physfs.h File Reference

Go to the source code of this file.

Data Structures

struct	**PHYSFS_File** A PhysicsFS file handle. More...
struct	**PHYSFS_ArchiveInfo** Information on various PhysicsFS-supported archives. More...
struct	**PHYSFS_Version** Information the version of PhysicsFS in use. More...
struct	**PHYSFS_Allocator** PhysicsFS allocation function pointers. More...

Defines

#define	**PHYSFS_file** **PHYSFS_File** 1.0 API compatibility define.
#define	**PHYSFS_VERSION**(x) Macro to determine PhysicsFS version program was compiled against.

Typedefs

typedef unsigned char	**PHYSFS_uint8** An unsigned, 8-bit integer type.
typedef signed char	**PHYSFS_sint8** A signed, 8-bit integer type.

typedef unsigned short	**PHYSFS_uint16** An unsigned, 16-bit integer type.
typedef signed short	**PHYSFS_sint16** A signed, 16-bit integer type.
typedef unsigned int	**PHYSFS_uint32** An unsigned, 32-bit integer type.
typedef signed int	**PHYSFS_sint32** A signed, 32-bit integer type.
typedef unsigned long long	**PHYSFS_uint64** An unsigned, 64-bit integer type.
typedef signed long long	**PHYSFS_sint64** A signed, 64-bit integer type.
typedef void(*	**PHYSFS_StringCallback**)(void *data, const char *str) Function signature for callbacks that report strings.
typedef void(*	**PHYSFS_EnumFilesCallback**)(void *data, const char *origdir, const char *fname) Function signature for callbacks that enumerate files.

Functions

void	**PHYSFS_getLinkedVersion** (**PHYSFS_Version** *ver) Get the version of PhysicsFS that is linked against your program.
int	**PHYSFS_init** (const char *argv0) Initialize the PhysicsFS library.
int	**PHYSFS_deinit** (void) Deinitialize the PhysicsFS library.
const **PHYSFS_ArchiveInfo** **	**PHYSFS_supportedArchiveTypes** (void) Get a list of supported archive types.
void	**PHYSFS_freeList** (void *listVar) Deallocate resources of lists returned by PhysicsFS.
const char *	**PHYSFS_getLastError** (void) Get human-readable error information.

const char *	**PHYSFS_getDirSeparator** (void) Get platform-dependent dir separator string.
void	**PHYSFS_permitSymbolicLinks** (int allow) Enable or disable following of symbolic links.
char **	**PHYSFS_getCdRomDirs** (void) Get an array of paths to available CD-ROM drives.
const char *	**PHYSFS_getBaseDir** (void) Get the path where the application resides.
const char *	**PHYSFS_getUserDir** (void) Get the path where user's home directory resides.
const char *	**PHYSFS_getWriteDir** (void) Get path where PhysicsFS will allow file writing.
int	**PHYSFS_setWriteDir** (const char *newDir) Tell PhysicsFS where it may write files.
int	**PHYSFS_addToSearchPath** (const char *newDir, int appendToPath) Add an archive or directory to the search path.
int	**PHYSFS_removeFromSearchPath** (const char *oldDir) Remove a directory or archive from the search path.
char **	**PHYSFS_getSearchPath** (void) Get the current search path.
int	**PHYSFS_setSaneConfig** (const char *organization, const char *appName, const char *archiveExt, int includeCdRoms, int archivesFirst) Set up sane, default paths.
int	**PHYSFS_mkdir** (const char *dirName) Create a directory.
int	**PHYSFS_delete** (const char *filename) Delete a file or directory.
const char *	**PHYSFS_getRealDir** (const char *filename) Figure out where in the search path a file resides.
char **	**PHYSFS_enumerateFiles** (const char *dir)

	Get a file listing of a search path's directory.
int	**PHYSFS_exists** (const char *fname) Determine if a file exists in the search path.
int	**PHYSFS_isDirectory** (const char *fname) Determine if a file in the search path is really a directory.
int	**PHYSFS_isSymbolicLink** (const char *fname) Determine if a file in the search path is really a symbolic link.
PHYSFS_sint64	**PHYSFS_getLastModTime** (const char *filename) Get the last modification time of a file.
PHYSFS_File *	**PHYSFS_openWrite** (const char *filename) Open a file for writing.
PHYSFS_File *	**PHYSFS_openAppend** (const char *filename) Open a file for appending.
PHYSFS_File *	**PHYSFS_openRead** (const char *filename) Open a file for reading.
int	**PHYSFS_close** (**PHYSFS_File** *handle) Close a PhysicsFS filehandle.
PHYSFS_sint64	**PHYSFS_read** (**PHYSFS_File** *handle, void *buffer, **PHYSFS_uint32** objSize, **PHYSFS_uint32** objCount) Read data from a PhysicsFS filehandle.
PHYSFS_sint64	**PHYSFS_write** (**PHYSFS_File** *handle, const void *buffer, **PHYSFS_uint32** objSize, **PHYSFS_uint32** objCount) Write data to a PhysicsFS filehandle.
int	**PHYSFS_eof** (**PHYSFS_File** *handle) Check for end-of-file state on a PhysicsFS filehandle.
PHYSFS_sint64	**PHYSFS_tell** (**PHYSFS_File** *handle) Determine current position within a PhysicsFS filehandle.
int	**PHYSFS_seek** (**PHYSFS_File** *handle, **PHYSFS_uint64** pos) Seek to a new position within a PhysicsFS filehandle.
PHYSFS_sint64	**PHYSFS_fileLength** (**PHYSFS_File** *handle)

	Get total length of a file in bytes.
int	**PHYSFS_setBuffer** (**PHYSFS_File** *handle, **PHYSFS_uint64** bufsize) Set up buffering for a PhysicsFS file handle.
int	**PHYSFS_flush** (**PHYSFS_File** *handle) Flush a buffered PhysicsFS file handle.
PHYSFS_sint16	**PHYSFS_swapSLE16** (**PHYSFS_sint16** val) Swap littleendian signed 16 to platform's native byte order.
PHYSFS_uint16	**PHYSFS_swapULE16** (**PHYSFS_uint16** val) Swap littleendian unsigned 16 to platform's native byte order.
PHYSFS_sint32	**PHYSFS_swapSLE32** (**PHYSFS_sint32** val) Swap littleendian signed 32 to platform's native byte order.
PHYSFS_uint32	**PHYSFS_swapULE32** (**PHYSFS_uint32** val) Swap littleendian unsigned 32 to platform's native byte order.
PHYSFS_sint64	**PHYSFS_swapSLE64** (**PHYSFS_sint64** val) Swap littleendian signed 64 to platform's native byte order.
PHYSFS_uint64	**PHYSFS_swapULE64** (**PHYSFS_uint64** val) Swap littleendian unsigned 64 to platform's native byte order.
PHYSFS_sint16	**PHYSFS_swapSBE16** (**PHYSFS_sint16** val) Swap bigendian signed 16 to platform's native byte order.
PHYSFS_uint16	**PHYSFS_swapUBE16** (**PHYSFS_uint16** val) Swap bigendian unsigned 16 to platform's native byte order.
PHYSFS_sint32	**PHYSFS_swapSBE32** (**PHYSFS_sint32** val) Swap bigendian signed 32 to platform's native byte order.
PHYSFS_uint32	**PHYSFS_swapUBE32** (**PHYSFS_uint32** val) Swap bigendian unsigned 32 to platform's native byte order.
PHYSFS_sint64	**PHYSFS_swapSBE64** (**PHYSFS_sint64** val) Swap bigendian signed 64 to platform's native byte order.
PHYSFS_uint64	**PHYSFS_swapUBE64** (**PHYSFS_uint64** val) Swap bigendian unsigned 64 to platform's native byte order.
int	**PHYSFS_readSLE16** (**PHYSFS_File** *file, **PHYSFS_sint16** *val)

	Read and convert a signed 16-bit littleendian value.
int	**PHYSFS_readULE16** (**PHYSFS_File** *file, **PHYSFS_uint16** *val) Read and convert an unsigned 16-bit littleendian value.
int	**PHYSFS_readSBE16** (**PHYSFS_File** *file, **PHYSFS_sint16** *val) Read and convert a signed 16-bit bigendian value.
int	**PHYSFS_readUBE16** (**PHYSFS_File** *file, **PHYSFS_uint16** *val) Read and convert an unsigned 16-bit bigendian value.
int	**PHYSFS_readSLE32** (**PHYSFS_File** *file, **PHYSFS_sint32** *val) Read and convert a signed 32-bit littleendian value.
int	**PHYSFS_readULE32** (**PHYSFS_File** *file, **PHYSFS_uint32** *val) Read and convert an unsigned 32-bit littleendian value.
int	**PHYSFS_readSBE32** (**PHYSFS_File** *file, **PHYSFS_sint32** *val) Read and convert a signed 32-bit bigendian value.
int	**PHYSFS_readUBE32** (**PHYSFS_File** *file, **PHYSFS_uint32** *val) Read and convert an unsigned 32-bit bigendian value.
int	**PHYSFS_readSLE64** (**PHYSFS_File** *file, **PHYSFS_sint64** *val) Read and convert a signed 64-bit littleendian value.
int	**PHYSFS_readULE64** (**PHYSFS_File** *file, **PHYSFS_uint64** *val) Read and convert an unsigned 64-bit littleendian value.
int	**PHYSFS_readSBE64** (**PHYSFS_File** *file, **PHYSFS_sint64** *val) Read and convert a signed 64-bit bigendian value.
int	**PHYSFS_readUBE64** (**PHYSFS_File** *file, **PHYSFS_uint64** *val) Read and convert an unsigned 64-bit bigendian value.
int	**PHYSFS_writeSLE16** (**PHYSFS_File** *file, **PHYSFS_sint16** val) Convert and write a signed 16-bit littleendian value.
int	**PHYSFS_writeULE16** (**PHYSFS_File** *file, **PHYSFS_uint16** val) Convert and write an unsigned 16-bit littleendian value.
int	**PHYSFS_writeSBE16** (**PHYSFS_File** *file, **PHYSFS_sint16** val) Convert and write a signed 16-bit bigendian value.

int	**PHYSFS_writeUBE16** (**PHYSFS_File** *file, **PHYSFS_uint16** val) Convert and write an unsigned 16-bit bigendian value.
int	**PHYSFS_writeSLE32** (**PHYSFS_File** *file, **PHYSFS_sint32** val) Convert and write a signed 32-bit littleendian value.
int	**PHYSFS_writeULE32** (**PHYSFS_File** *file, **PHYSFS_uint32** val) Convert and write an unsigned 32-bit littleendian value.
int	**PHYSFS_writeSBE32** (**PHYSFS_File** *file, **PHYSFS_sint32** val) Convert and write a signed 32-bit bigendian value.
int	**PHYSFS_writeUBE32** (**PHYSFS_File** *file, **PHYSFS_uint32** val) Convert and write an unsigned 32-bit bigendian value.
int	**PHYSFS_writeSLE64** (**PHYSFS_File** *file, **PHYSFS_sint64** val) Convert and write a signed 64-bit littleendian value.
int	**PHYSFS_writeULE64** (**PHYSFS_File** *file, **PHYSFS_uint64** val) Convert and write an unsigned 64-bit littleendian value.
int	**PHYSFS_writeSBE64** (**PHYSFS_File** *file, **PHYSFS_sint64** val) Convert and write a signed 64-bit bigending value.
int	**PHYSFS_writeUBE64** (**PHYSFS_File** *file, **PHYSFS_uint64** val) Convert and write an unsigned 64-bit bigendian value.
int	**PHYSFS_isInit** (void) Determine if the PhysicsFS library is initialized.
int	**PHYSFS_symbolicLinksPermitted** (void) Determine if the symbolic links are permitted.
int	**PHYSFS_setAllocator** (const **PHYSFS_Allocator** *allocator) Hook your own allocation routines into PhysicsFS.
int	**PHYSFS_mount** (const char *newDir, const char *mountPoint, int appendToPath) Add an archive or directory to the search path.
const char *	**PHYSFS_getMountPoint** (const char *dir) Determine a mounted archive's mountpoint.
void	**PHYSFS_getCdRomDirsCallback** (**PHYSFS_StringCallback** c, void *d) Enumerate CD-ROM directories, using an application-defined callback.

void	**PHYSFS_getSearchPathCallback** (**PHYSFS_StringCallback** c, void *d) Enumerate the search path, using an application-defined callback.
void	**PHYSFS_enumerateFilesCallback** (const char *dir, **PHYSFS_EnumFilesCallback** c, void *d) Get a file listing of a search path's directory, using an application-defined callback.
void	**PHYSFS_utf8FromUcs4** (const **PHYSFS_uint32** *src, char *dst, **PHYSFS_uint64** len) Convert a UCS-4 string to a UTF-8 string.
void	**PHYSFS_utf8ToUcs4** (const char *src, **PHYSFS_uint32** *dst, **PHYSFS_uint64** len) Convert a UTF-8 string to a UCS-4 string.
void	**PHYSFS_utf8FromUcs2** (const **PHYSFS_uint16** *src, char *dst, **PHYSFS_uint64** len) Convert a UCS-2 string to a UTF-8 string.
void	**PHYSFS_utf8ToUcs2** (const char *src, **PHYSFS_uint16** *dst, **PHYSFS_uint64** len) Convert a UTF-8 string to a UCS-2 string.
void	**PHYSFS_utf8FromLatin1** (const char *src, char *dst, **PHYSFS_uint64** len) Convert a UTF-8 string to a Latin1 string.

Detailed Description

Define Documentation

#define PHYSFS_file   PHYSFS_File

1.0 API compatibility define.

PHYSFS_file is identical to **PHYSFS_File**. This define is here for backwards compatibility with the 1.0 API, which had an inconsistent capitalization convention in this case. New code should use **PHYSFS_File**, as this define may go away someday.

See also:
PHYSFS_File

#define PHYSFS_VERSION (x)

Value:

```
{ \
    (x)->major = PHYSFS_VER_MAJOR; \
    (x)->minor = PHYSFS_VER_MINOR; \
    (x)->patch = PHYSFS_VER_PATCH; \
}
```

Macro to determine PhysicsFS version program was compiled against.

This macro fills in a **PHYSFS_Version** structure with the version of the library you compiled against. This is determined by what header the compiler uses. Note that if you dynamically linked the library, you might have a slightly newer or older version at runtime. That version can be determined with **PHYSFS_getLinkedVersion()**, which, unlike PHYSFS_VERSION, is not a macro.

Parameters:

x A pointer to a **PHYSFS_Version** struct to initialize.

See also:

PHYSFS_Version
PHYSFS_getLinkedVersion

Typedef Documentation

PHYSFS_EnumFilesCallback

Function signature for callbacks that enumerate files.

These are used to report a list of directory entries to an original caller, one file/dir/symlink per callback. All strings are UTF-8 encoded. Functions should not try to modify or free any string's memory.

These callbacks are used, starting in PhysicsFS 1.1, as an alternative to functions that would return lists that need to be cleaned up with **PHYSFS_freeList()**. The callback means that the library doesn't need to allocate an entire list and all the strings up front.

Be aware that promises data ordering in the list versions are not necessarily so in the callback versions. Check the documentation on specific APIs, but strings may not be sorted as you expect.

Parameters:

data	User-defined data pointer, passed through from the API that eventually called the callback.
origdir	A string containing the full path, in platform-independent notation, of the directory containing this file. In most cases, this is the directory on which you requested enumeration, passed in the callback for your convenience.
fname	The filename that is being enumerated. It may not be in alphabetical order compared to other callbacks that have fired, and it will not contain the full path. You can recreate the fullpath with $origdir/$fname ... The file can be a subdirectory, a file, a symlink, etc.

See also:
PHYSFS_enumerateFilesCallback

PHYSFS_sint64

A signed, 64-bit integer type.

Warning:
on platforms without any sort of 64-bit datatype, this is equivalent to PHYSFS_sint32!

PHYSFS_StringCallback

Function signature for callbacks that report strings.

These are used to report a list of strings to an original caller, one string per callback. All strings are UTF-8 encoded. Functions should not try to modify or free the string's memory.

These callbacks are used, starting in PhysicsFS 1.1, as an alternative to functions that would return lists that need to be cleaned up with **PHYSFS_freeList()**. The callback means that the library doesn't need to allocate an entire list and all the strings up front.

Be aware that promises data ordering in the list versions are not necessarily so in the callback versions. Check the documentation on specific APIs, but strings may not be sorted as you expect.

Parameters:

data	User-defined data pointer, passed through from the API that eventually called the callback.
str	The string data about which the callback is meant to inform.

See also:
  PHYSFS_getCdRomDirsCallback
  PHYSFS_getSearchPathCallback

PHYSFS_uint64

An unsigned, 64-bit integer type.

Warning:
  on platforms without any sort of 64-bit datatype, this is equivalent to PHYSFS_uint32!

Function Documentation

```
int PHYSFS_addToSearchPath ( const char *  newDir,
                             int           appendToPath
                           )
```

Add an archive or directory to the search path.

This is a legacy call in PhysicsFS 2.0, equivalent to: PHYSFS_mount(newDir, NULL, appendToPath);

You must use this and not PHYSFS_mount if binary compatibility with PhysicsFS 1.0 is important (which it may not be for many people).

See also:
  PHYSFS_mount
  PHYSFS_removeFromSearchPath

PHYSFS_getSearchPath

int PHYSFS_close (PHYSFS_File * handle)

Close a PhysicsFS filehandle.

This call is capable of failing if the operating system was buffering writes to the physical media, and, now forced to write those changes to physical media, can not store the data for some reason. In such a case, the filehandle stays open. A well-written program should ALWAYS check the return value from the close call in addition to every writing call!

Parameters:

handle handle returned from PHYSFS_open*().

Returns:

nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_openRead
PHYSFS_openWrite
PHYSFS_openAppend

int PHYSFS_deinit (void)

Deinitialize the PhysicsFS library.

This closes any files opened via PhysicsFS, blanks the search/write paths, frees memory, and invalidates all of your file handles.

Note that this call can FAIL if there's a file open for writing that refuses to close (for example, the underlying operating system was buffering writes to network filesystem, and the fileserver has crashed, or a hard drive has failed, etc). It is usually best to close all write handles yourself before calling this function, so that you can gracefully handle a specific failure.

Once successfully deinitialized, **PHYSFS_init()** can be called again to restart the subsystem. All default API states are restored at this point, with the exception of any custom allocator you might have specified, which survives between initializations.

Returns:

nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**. If failure, state of PhysFS is undefined, and probably badly screwed up.

See also:

PHYSFS_init
PHYSFS_isInit

int PHYSFS_delete (const char * filename)

Delete a file or directory.

(filename) is specified in platform-independent notation in relation to the write dir.

A directory must be empty before this call can delete it.

Deleting a symlink will remove the link, not what it points to, regardless of whether you "permitSymLinks" or not.

So if you've got the write dir set to "C:\mygame\writedir" and call PHYSFS_delete("downloads/maps/level1.map") then the file "C:\mygame\writedir\downloads\maps\level1.map" is removed from the physical filesystem, if it exists and the operating system permits the deletion.

Note that on Unix systems, deleting a file may be successful, but the actual file won't be removed until all processes that have an open filehandle to it (including your program) close their handles.

Chances are, the bits that make up the file still exist, they are just made available to be written over at a later point. Don't consider this a security method or anything. :)

Parameters:

filename Filename to delete.

Returns:

nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

char ** PHYSFS_enumerateFiles (const char * dir)

Get a file listing of a search path's directory.

Matching directories are interpolated. That is, if "C:\mydir" is in the search path and contains a directory "savegames" that contains "x.sav", "y.sav", and "z.sav", and there is also a "C:\userdir" in the search path that has a "savegames" subdirectory with "w.sav", then the following code:

```
char **rc = PHYSFS_enumerateFiles("savegames");
char **i;

for (i = rc; *i != NULL; i++)
    printf(" * We've got [%s].\n", *i);

PHYSFS_freeList(rc);
```

...will print:

```
* We've got [x.sav].
* We've got [y.sav].
* We've got [z.sav].
* We've got [w.sav].
```

Feel free to sort the list however you like. We only promise there will be no duplicates, but not what order the final list will come back in.

Don't forget to call **PHYSFS_freeList()** with the return value from this function when you are done with it.

Parameters:

dir directory in platform-independent notation to enumerate.

Returns:

Null-terminated array of null-terminated strings.

See also:

PHYSFS_enumerateFilesCallback

void PHYSFS_enumerateFilesCallback (const char * dir, PHYSFS_EnumFilesCallback c, void * d)

Get a file listing of a search path's directory, using an application-defined callback.

Internally, **PHYSFS_enumerateFiles()** just calls this function and then builds a list before returning to the application, so functionality is identical except for how the information is represented to the application.

Unlike **PHYSFS_enumerateFiles()**, this function does not return an array. Rather, it calls a function specified by the application once per element of the search path:

```
static void printDir(void *data, const char *origdir, const char *fname)
{
    printf(" * We've got [%s] in [%s].\n", fname, origdir);
}

// ...
PHYSFS_enumerateFilesCallback("/some/path", printDir, NULL);
```

Items sent to the callback are not guaranteed to be in any order whatsoever. There is no sorting done at this level, and if you need that, you should probably use **PHYSFS_enumerateFiles()** instead, which guarantees alphabetical sorting. This form reports whatever is discovered in each archive before moving on to the next. Even within one archive, we can't guarantee what order it will discover data. *Any sorting you find in these callbacks is just pure luck. Do not rely on it.*

Parameters:

dir	Directory, in platform-independent notation, to enumerate.
c	Callback function to notify about search path elements.
d	Application-defined data passed to callback. Can be NULL.

See also:

PHYSFS_EnumFilesCallback
PHYSFS_enumerateFiles

int PHYSFS_eof (PHYSFS_File * handle)

Check for end-of-file state on a PhysicsFS filehandle.

Determine if the end of file has been reached in a PhysicsFS filehandle.

Parameters:

handle handle returned from **PHYSFS_openRead()**.

Returns:

nonzero if EOF, zero if not.

See also:

PHYSFS_read
PHYSFS_tell

int PHYSFS_exists (const char * fname)

Determine if a file exists in the search path.

Reports true if there is an entry anywhere in the search path by the name of (fname).

Note that entries that are symlinks are ignored if PHYSFS_permitSymbolicLinks(1) hasn't been called, so you might end up further down in the search path than expected.

Parameters:

fname filename in platform-independent notation.

Returns:

non-zero if filename exists. zero otherwise.

See also:

PHYSFS_isDirectory
PHYSFS_isSymbolicLink

PHYSFS_sint64 PHYSFS_fileLength (PHYSFS_File * *handle*)

Get total length of a file in bytes.

Note that if the file size can't be determined (since the archive is "streamed" or whatnot) than this will report (-1). Also note that if another process/thread is writing to this file at the same time, then the information this function supplies could be incorrect before you get it. Use with caution, or better yet, don't use at all.

Parameters:

handle handle returned from PHYSFS_open*().

Returns:

size in bytes of the file. -1 if can't be determined.

See also:

PHYSFS_tell
PHYSFS_seek

int PHYSFS_flush (PHYSFS_File * *handle*)

Flush a buffered PhysicsFS file handle.

For buffered files opened for writing, this will put the current contents of the buffer to disk and flag the buffer as empty if possible.

For buffered files opened for reading or unbuffered files, this is a safe no-op, and will report success.

Parameters:

handle handle returned from PHYSFS_open*().

Returns:

nonzero if successful, zero on error.

See also:

PHYSFS_setBuffer

PHYSFS_close

void PHYSFS_freeList (void * *listVar*)

Deallocate resources of lists returned by PhysicsFS.

Certain PhysicsFS functions return lists of information that are dynamically allocated. Use this function to free those resources.

Parameters:

listVar List of information specified as freeable by this function.

See also:

PHYSFS_getCdRomDirs
PHYSFS_enumerateFiles
PHYSFS_getSearchPath

const char * PHYSFS_getBaseDir (void)

Get the path where the application resides.

Helper function.

Get the "base dir". This is the directory where the application was run from, which is probably the installation directory, and may or may not be the process's current working directory.

You should probably use the base dir in your search path.

Returns:

READ ONLY string of base dir in platform-dependent notation.

See also:

PHYSFS_getUserDir

char ** PHYSFS_getCdRomDirs (void)

Get an array of paths to available CD-ROM drives.

The dirs returned are platform-dependent ("D:\" on Win32, "/cdrom" or whatnot on Unix). Dirs are only returned if there is a disc ready and accessible in the drive. So if you've got two drives (D: and E:), and only E: has a disc in it, then that's all you get. If the user inserts a disc in D: and you call this function again, you get both drives. If, on a Unix box, the user unmounts a disc and remounts it elsewhere, the next call to this function will reflect that change.

This function refers to "CD-ROM" media, but it really means "inserted disc media," such as DVD-ROM, HD-DVD, CDRW, and Blu-Ray discs. It looks for filesystems, and as such won't report an audio CD, unless there's a mounted filesystem track on it.

The returned value is an array of strings, with a NULL entry to signify the end of the list:

```
char **cds = PHYSFS_getCdRomDirs();
char **i;

for (i = cds; *i != NULL; i++)
    printf("cdrom dir [%s] is available.\n", *i);

PHYSFS_freeList(cds);
```

This call may block while drives spin up. Be forewarned.

When you are done with the returned information, you may dispose of the resources by calling **PHYSFS_freeList()** with the returned pointer.

Returns:
Null-terminated array of null-terminated strings.

See also:
PHYSFS_getCdRomDirsCallback

void PHYSFS_getCdRomDirsCallback (PHYSFS_StringCallback c,
void * d

)

Enumerate CD-ROM directories, using an application-defined callback.

Internally, **PHYSFS_getCdRomDirs()** just calls this function and then builds a list before returning to the application, so functionality is identical except for how the information is represented to the application.

Unlike **PHYSFS_getCdRomDirs()**, this function does not return an array. Rather, it calls a function specified by the application once per detected disc:

```
static void foundDisc(void *data, const char *cddir)
{
    printf("cdrom dir [%s] is available.\n", cddir);
}

// ...
PHYSFS_getCdRomDirsCallback(foundDisc, NULL);
```

This call may block while drives spin up. Be forewarned.

Parameters:

c Callback function to notify about detected drives.

d Application-defined data passed to callback. Can be NULL.

See also:

PHYSFS_StringCallback
PHYSFS_getCdRomDirs

const char * PHYSFS_getDirSeparator (void)

Get platform-dependent dir separator string.

This returns "\\" on win32, "/" on Unix, and ":" on MacOS. It may be more than one character, depending on the platform, and your code should take that into account. Note that this is only useful for setting up the search/write paths, since access into those dirs always use '/' (platform-independent notation) to separate directories. This is also handy for getting platform-independent access when using stdio calls.

Returns:
READ ONLY null-terminated string of platform's dir separator.

const char * PHYSFS_getLastError (void)

Get human-readable error information.

Get the last PhysicsFS error message as a human-readable, null-terminated string. This will be NULL if there's been no error since the last call to this function. The pointer returned by this call points to an internal buffer. Each thread has a unique error state associated with it, but each time a new error message is set, it will overwrite the previous one associated with that thread. It is safe to call this function at anytime, even before **PHYSFS_init()**.

It is not wise to expect a specific string of characters here, since the error message may be localized into an unfamiliar language. These strings are meant to be passed on directly to the user.

Returns:
READ ONLY string of last error message.

PHYSFS_sint64 PHYSFS_getLastModTime (const char * filename)

Get the last modification time of a file.

The modtime is returned as a number of seconds since the epoch (Jan 1, 1970). The exact derivation and accuracy of this time depends on the particular archiver. If there is no reasonable way to obtain this information for a particular archiver, or there was some sort of error, this function returns (-1).

Parameters:
filename filename to check, in platform-independent notation.

Returns:
last modified time of the file. -1 if it can't be determined.

void PHYSFS_getLinkedVersion (PHYSFS_Version * ver)

Get the version of PhysicsFS that is linked against your program.

If you are using a shared library (DLL) version of PhysFS, then it is possible that it will be different than the version you compiled against.

This is a real function; the macro PHYSFS_VERSION tells you what version of PhysFS you compiled against:

```
PHYSFS_Version compiled;
PHYSFS_Version linked;

PHYSFS_VERSION(&compiled);
PHYSFS_getLinkedVersion(&linked);
printf("We compiled against PhysFS version %d.%d.%d ...\n",
          compiled.major, compiled.minor, compiled.patch);
printf("But we linked against PhysFS version %d.%d.%d.\n",
          linked.major, linked.minor, linked.patch);
```

This function may be called safely at any time, even before **PHYSFS_init()**.

See also:
PHYSFS_VERSION

int PHYSFS_getMountPoint (const char * dir)

Determine a mounted archive's mountpoint.

You give this function the name of an archive or dir you successfully added to the search path, and it reports the location in the interpolated tree where it is mounted. Files mounted with a NULL mountpoint or through **PHYSFS_addToSearchPath()** will report "/". The return value is READ ONLY and valid until the archive is removed from the search path.

Parameters:
dir directory or archive previously added to the path, in platform-dependent notation. This must match the string used when adding, even if your string would also reference the same file with a different string of characters.

Returns:

READ-ONLY string of mount point if added to path, NULL on failure (bogus archive, etc) Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_removeFromSearchPath
PHYSFS_getSearchPath
PHYSFS_getMountPoint

const char * PHYSFS_getRealDir (const char * filename)

Figure out where in the search path a file resides.

The file is specified in platform-independent notation. The returned filename will be the element of the search path where the file was found, which may be a directory, or an archive. Even if there are multiple matches in different parts of the search path, only the first one found is used, just like when opening a file.

So, if you look for "maps/level1.map", and C:\mygame is in your search path and C:\mygame\maps\level1.map exists, then "C:\mygame" is returned.

If a any part of a match is a symbolic link, and you've not explicitly permitted symlinks, then it will be ignored, and the search for a match will continue.

If you specify a fake directory that only exists as a mount point, it'll be associated with the first archive mounted there, even though that directory isn't necessarily contained in a real archive.

Parameters:

filename	file to look for.

Returns:

READ ONLY string of element of search path containing the the file in question. NULL if not found.

char ** PHYSFS_getSearchPath (void)

Get the current search path.

The default search path is an empty list.

The returned value is an array of strings, with a NULL entry to signify the end of the list:

```
char **i;

for (i = PHYSFS_getSearchPath(); *i != NULL; i++)
    printf("[%s] is in the search path.\n", *i);
```

When you are done with the returned information, you may dispose of the resources by calling **PHYSFS_freeList()** with the returned pointer.

Returns:
Null-terminated array of null-terminated strings. NULL if there was a problem (read: OUT OF MEMORY).

See also:
PHYSFS_getSearchPathCallback
PHYSFS_addToSearchPath
PHYSFS_removeFromSearchPath

```
void PHYSFS_getSearchPathCallback ( PHYSFS_StringCallback c,
                                    void *                d
                                  )
```

Enumerate the search path, using an application-defined callback.

Internally, **PHYSFS_getSearchPath()** just calls this function and then builds a list before returning to the application, so functionality is identical except for how the information is represented to the application.

Unlike **PHYSFS_getSearchPath()**, this function does not return an array. Rather, it calls a function specified by the application once per element of the search path:

```
static void printSearchPath(void *data, const char *pathItem)
```

```
{
    printf("[%s] is in the search path.\n", pathItem);
}

// ...
PHYSFS_getSearchPathCallback(printSearchPath, NULL);
```

Elements of the search path are reported in order search priority, so the first archive/dir that would be examined when looking for a file is the first element passed through the callback.

Parameters:

c Callback function to notify about search path elements.

d Application-defined data passed to callback. Can be NULL.

See also:

PHYSFS_StringCallback
PHYSFS_getSearchPath

const char * PHYSFS_getUserDir (void)

Get the path where user's home directory resides.

Helper function.

Get the "user dir". This is meant to be a suggestion of where a specific user of the system can store files. On Unix, this is her home directory. On systems with no concept of multiple home directories (MacOS, win95), this will default to something like "C:\mybasedir\users\username" where "username" will either be the login name, or "default" if the platform doesn't support multiple users, either.

You should probably use the user dir as the basis for your write dir, and also put it near the beginning of your search path.

Returns:

READ ONLY string of user dir in platform-dependent notation.

See also:

PHYSFS_getBaseDir

const char * PHYSFS_getWriteDir (void)

Get path where PhysicsFS will allow file writing.

Get the current write dir. The default write dir is NULL.

Returns:
READ ONLY string of write dir in platform-dependent notation, OR NULL IF NO WRITE PATH IS CURRENTLY SET.

See also:
PHYSFS_setWriteDir

int PHYSFS_init (const char * argv0)

Initialize the PhysicsFS library.

This must be called before any other PhysicsFS function.

This should be called prior to any attempts to change your process's current working directory.

Parameters:

argv0	the argv[0] string passed to your program's mainline. This may be NULL on most platforms (such as ones without a standard main() function), but you should always try to pass something in here. Unix-like systems such as Linux _need_ to pass argv[0] from main() in here.

Returns:
nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:
PHYSFS_deinit
PHYSFS_isInit

int PHYSFS_isDirectory (const char * fname)

Determine if a file in the search path is really a directory.

Determine if the first occurence of (fname) in the search path is really a directory entry.

Note that entries that are symlinks are ignored if PHYSFS_permitSymbolicLinks(1) hasn't been called, so you might end up further down in the search path than expected.

Parameters:

fname filename in platform-independent notation.

Returns:

non-zero if filename exists and is a directory. zero otherwise.

See also:

PHYSFS_exists
PHYSFS_isSymbolicLink

int PHYSFS_isInit (void)

Determine if the PhysicsFS library is initialized.

Once **PHYSFS_init()** returns successfully, this will return non-zero. Before a successful **PHYSFS_init()** and after **PHYSFS_deinit()** returns successfully, this will return zero. This function is safe to call at any time.

Returns:

non-zero if library is initialized, zero if library is not.

See also:

PHYSFS_init
PHYSFS_deinit

int PHYSFS_isSymbolicLink (const char * fname)

Determine if a file in the search path is really a symbolic link.

Determine if the first occurence of (fname) in the search path is really a symbolic link.

Note that entries that are symlinks are ignored if PHYSFS_permitSymbolicLinks(1) hasn't been called, and as such, this function will always return 0 in that case.

Parameters:

fname filename in platform-independent notation.

Returns:

non-zero if filename exists and is a symlink. zero otherwise.

See also:

PHYSFS_exists
PHYSFS_isDirectory

int PHYSFS_mkdir (const char * dirName)

Create a directory.

This is specified in platform-independent notation in relation to the write dir. All missing parent directories are also created if they don't exist.

So if you've got the write dir set to "C:\mygame\writedir" and call PHYSFS_mkdir("downloads/maps") then the directories "C:\mygame\writedir\downloads" and "C:\mygame\writedir\downloads\maps" will be created if possible. If the creation of "maps" fails after we have successfully created "downloads", then the function leaves the created directory behind and reports failure.

Parameters:

dirName New dir to create.

Returns:

nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_delete

```
int PHYSFS_mount ( const char *  newDir,
                   const char *  mountPoint,
                   int           appendToPath
                 )
```

Add an archive or directory to the search path.

If this is a duplicate, the entry is not added again, even though the function succeeds. You may not add the same archive to two different mountpoints: duplicate checking is done against the archive and not the mountpoint.

When you mount an archive, it is added to a virtual file system...all files in all of the archives are interpolated into a single hierachical file tree. Two archives mounted at the same place (or an archive with files overlapping another mountpoint) may have overlapping files: in such a case, the file earliest in the search path is selected, and the other files are inaccessible to the application. This allows archives to be used to override previous revisions; you can use the mounting mechanism to place archives at a specific point in the file tree and prevent overlap; this is useful for downloadable mods that might trample over application data or each other, for example.

The mountpoint does not need to exist prior to mounting, which is different than those familiar with the Unix concept of "mounting" may not expect. As well, more than one archive can be mounted to the same mountpoint, or mountpoints and archive contents can overlap...the interpolation mechanism still functions as usual.

Parameters:

newDir	directory or archive to add to the path, in platform-dependent notation.
mountPoint	Location in the interpolated tree that this archive will be "mounted", in platform-independent notation. NULL or "" is equivalent to "/".
appendToPath	nonzero to append to search path, zero to prepend.

Returns:

nonzero if added to path, zero on failure (bogus archive, dir missing, etc). Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_removeFromSearchPath
PHYSFS_getSearchPath
PHYSFS_getMountPoint

PHYSFS_File * PHYSFS_openAppend (const char * *filename*)

Open a file for appending.

Open a file for writing, in platform-independent notation and in relation to the write dir as the root of the writable filesystem. The specified file is created if it doesn't exist. If it does exist, the writing offset is set to the end of the file, so the first write will be the byte after the end.

Note that entries that are symlinks are ignored if PHYSFS_permitSymbolicLinks(1) hasn't been called, and opening a symlink with this function will fail in such a case.

Parameters:

filename File to open.

Returns:

A valid PhysicsFS filehandle on success, NULL on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_openRead
PHYSFS_openWrite
PHYSFS_write
PHYSFS_close

PHYSFS_File * PHYSFS_openRead (const char * *filename*)

Open a file for reading.

Open a file for reading, in platform-independent notation. The search path is checked one at a time until a matching file is found, in which case an abstract filehandle is associated with it, and reading may be done. The reading offset is set to the first byte of the file.

Note that entries that are symlinks are ignored if PHYSFS_permitSymbolicLinks(1) hasn't been called, and opening a symlink with this function will fail in such a case.

Parameters:

filename File to open.

Returns:

A valid PhysicsFS filehandle on success, NULL on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_openWrite
PHYSFS_openAppend
PHYSFS_read
PHYSFS_close

PHYSFS_File * PHYSFS_openWrite (const char * filename)

Open a file for writing.

Open a file for writing, in platform-independent notation and in relation to the write dir as the root of the writable filesystem. The specified file is created if it doesn't exist. If it does exist, it is truncated to zero bytes, and the writing offset is set to the start.

Note that entries that are symlinks are ignored if PHYSFS_permitSymbolicLinks(1) hasn't been called, and opening a symlink with this function will fail in such a case.

Parameters:

filename File to open.

Returns:

A valid PhysicsFS filehandle on success, NULL on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_openRead
PHYSFS_openAppend
PHYSFS_write
PHYSFS_close

void PHYSFS_permitSymbolicLinks (int *allow*)

Enable or disable following of symbolic links.

Some physical filesystems and archives contain files that are just pointers to other files. On the physical filesystem, opening such a link will (transparently) open the file that is pointed to.

By default, PhysicsFS will check if a file is really a symlink during open calls and fail if it is. Otherwise, the link could take you outside the write and search paths, and compromise security.

If you want to take that risk, call this function with a non-zero parameter. Note that this is more for sandboxing a program's scripting language, in case untrusted scripts try to compromise the system. Generally speaking, a user could very well have a legitimate reason to set up a symlink, so unless you feel there's a specific danger in allowing them, you should permit them.

Symlinks are only explicitly checked when dealing with filenames in platform-independent notation. That is, when setting up your search and write paths, etc, symlinks are never checked for.

Symbolic link permission can be enabled or disabled at any time after you've called **PHYSFS_init()**, and is disabled by default.

Parameters:

allow	nonzero to permit symlinks, zero to deny linking.

See also:
PHYSFS_symbolicLinksPermitted

PHYSFS_sint64 PHYSFS_read (PHYSFS_File * **handle,**
void * **buffer,**

```
                                   PHYSFS_uint32  objSize,
                                   PHYSFS_uint32  objCount
                                 )
```

Read data from a PhysicsFS filehandle.

The file must be opened for reading.

Parameters:

handle	handle returned from **PHYSFS_openRead()**.
buffer	buffer to store read data into.
objSize	size in bytes of objects being read from (handle).
objCount	number of (objSize) objects to read from (handle).

Returns:

number of objects read. **PHYSFS_getLastError()** can shed light on the reason this might be < (objCount), as can **PHYSFS_eof()**. -1 if complete failure.

See also:

PHYSFS_eof

```
int PHYSFS_readSBE16 ( PHYSFS_File *    file,
                       PHYSFS_sint16 *  val
                     )
```

Read and convert a signed 16-bit bigendian value.

Convenience function. Read a signed 16-bit bigendian value from a file and convert it to the platform's native byte order.

Parameters:

file	PhysicsFS file handle from which to read.
val	pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

int PHYSFS_readSBE32 (PHYSFS_File * file, PHYSFS_sint32 * val)

Read and convert a signed 32-bit bigendian value.

Convenience function. Read a signed 32-bit bigendian value from a file and convert it to the platform's native byte order.

Parameters:

file	PhysicsFS file handle from which to read.
val	pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

int PHYSFS_readSBE64 (PHYSFS_File * file, PHYSFS_sint64 * val)

Read and convert a signed 64-bit bigendian value.

Convenience function. Read a signed 64-bit bigendian value from a file and convert it to the platform's native byte order.

Parameters:

file	PhysicsFS file handle from which to read.
val	pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:
Remember, PHYSFS_sint64 is only 32 bits on platforms without any sort of 64-bit support.

```
int PHYSFS_readSLE16 ( PHYSFS_File *    file,
                       PHYSFS_sint16 *  val
                     )
```

Read and convert a signed 16-bit littleendian value.

Convenience function. Read a signed 16-bit littleendian value from a file and convert it to the platform's native byte order.

Parameters:
file PhysicsFS file handle from which to read.
val pointer to where value should be stored.

Returns:
zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_readSLE32 ( PHYSFS_File *    file,
                       PHYSFS_sint32 *  val
                     )
```

Read and convert a signed 32-bit littleendian value.

Convenience function. Read a signed 32-bit littleendian value from a file and convert it to the platform's native byte order.

Parameters:
file PhysicsFS file handle from which to read.

val pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

int PHYSFS_readSLE64 (PHYSFS_File * file, PHYSFS_sint64 * val)

Read and convert a signed 64-bit littleendian value.

Convenience function. Read a signed 64-bit littleendian value from a file and convert it to the platform's native byte order.

Parameters:

file PhysicsFS file handle from which to read.

val pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_sint64 is only 32 bits on platforms without any sort of 64-bit support.

int PHYSFS_readUBE16 (PHYSFS_File * file, PHYSFS_uint16 * val)

Read and convert an unsigned 16-bit bigendian value.

Convenience function. Read an unsigned 16-bit bigendian value from a file and convert it to the platform's native byte order.

Parameters:

file PhysicsFS file handle from which to read.

val pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_readUBE32 ( PHYSFS_File *   file,
                       PHYSFS_uint32 * val
                     )
```

Read and convert an unsigned 32-bit bigendian value.

Convenience function. Read an unsigned 32-bit bigendian value from a file and convert it to the platform's native byte order.

Parameters:

file PhysicsFS file handle from which to read.

val pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_readUBE64 ( PHYSFS_File *   file,
                       PHYSFS_uint64 * val
                     )
```

Read and convert an unsigned 64-bit bigendian value.

Convenience function. Read an unsigned 64-bit bigendian value from a file and convert it to the platform's native byte order.

Parameters:

file PhysicsFS file handle from which to read.

val pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

```
int PHYSFS_readULE16 ( PHYSFS_File *    file,
                       PHYSFS_uint16 *  val
                     )
```

Read and convert an unsigned 16-bit littleendian value.

Convenience function. Read an unsigned 16-bit littleendian value from a file and convert it to the platform's native byte order.

Parameters:

file PhysicsFS file handle from which to read.

val pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_readULE32 ( PHYSFS_File *    file,
                       PHYSFS_uint32 *  val
```

)

Read and convert an unsigned 32-bit littleendian value.

Convenience function. Read an unsigned 32-bit littleendian value from a file and convert it to the platform's native byte order.

Parameters:

file	PhysicsFS file handle from which to read.
val	pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_readULE64 ( PHYSFS_File *     file,
                       PHYSFS_uint64 *   val
                     )
```

Read and convert an unsigned 64-bit littleendian value.

Convenience function. Read an unsigned 64-bit littleendian value from a file and convert it to the platform's native byte order.

Parameters:

file	PhysicsFS file handle from which to read.
val	pointer to where value should be stored.

Returns:

zero on failure, non-zero on success. If successful, (*val) will store the result. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

int PHYSFS_removeFromSearchPath (const char * oldDir)

Remove a directory or archive from the search path.

This must be a (case-sensitive) match to a dir or archive already in the search path, specified in platform-dependent notation.

This call will fail (and fail to remove from the path) if the element still has files open in it.

Parameters:

oldDir	dir/archive to remove.

Returns:

nonzero on success, zero on failure. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_addToSearchPath
PHYSFS_getSearchPath

int PHYSFS_seek (PHYSFS_File * handle, PHYSFS_uint64 pos)

Seek to a new position within a PhysicsFS filehandle.

The next read or write will occur at that place. Seeking past the beginning or end of the file is not allowed, and causes an error.

Parameters:

handle	handle returned from PHYSFS_open*().
pos	number of bytes from start of file to seek to.

Returns:

nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:
PHYSFS_tell

int PHYSFS_setAllocator (const PHYSFS_Allocator * allocator)

Hook your own allocation routines into PhysicsFS.

(This is for limited, hardcore use. If you don't immediately see a need for it, you can probably ignore this forever.)

By default, PhysicsFS will use whatever is reasonable for a platform to manage dynamic memory (usually ANSI C malloc/realloc/calloc/free, but some platforms might use something else), but in some uncommon cases, the app might want more control over the library's memory management. This lets you redirect PhysicsFS to use your own allocation routines instead. You can only call this function before **PHYSFS_init()**; if the library is initialized, it'll reject your efforts to change the allocator mid-stream. You may call this function after **PHYSFS_deinit()** if you are willing to shut down the library and restart it with a new allocator; this is a safe and supported operation. The allocator remains intact between deinit/init calls. If you want to return to the platform's default allocator, pass a NULL in here.

If you aren't immediately sure what to do with this function, you can safely ignore it altogether.

Parameters:

allocator Structure containing your allocator's entry points.

Returns:

zero on failure, non-zero on success. This call only fails when used between **PHYSFS_init()** and **PHYSFS_deinit()** calls.

int PHYSFS_setBuffer (PHYSFS_File * handle, PHYSFS_uint64 bufsize)

```
int PHYSFS_setBuffer ( PHYSFS_File *   handle,
                       PHYSFS_uint64  bufsize
                     )
```

Set up buffering for a PhysicsFS file handle.

Define an i/o buffer for a file handle. A memory block of (bufsize) bytes will be allocated and associated with (handle).

For files opened for reading, up to (bufsize) bytes are read from (handle) and stored in the internal buffer. Calls to **PHYSFS_read()** will pull from this buffer until it is empty, and then refill it for more reading. Note that compressed files, like ZIP archives, will decompress while buffering, so this can be handy for offsetting CPU-intensive operations. The buffer isn't filled until you do your next read.

For files opened for writing, data will be buffered to memory until the buffer is full or the buffer is flushed. Closing a handle implicitly causes a flush...check your return values!

Seeking, etc transparently accounts for buffering.

You can resize an existing buffer by calling this function more than once on the same file. Setting the buffer size to zero will free an existing buffer.

PhysicsFS file handles are unbuffered by default.

Please check the return value of this function! Failures can include not being able to seek backwards in a read-only file when removing the buffer, not being able to allocate the buffer, and not being able to flush the buffer to disk, among other unexpected problems.

Parameters:

handle	handle returned from PHYSFS_open*().
bufsize	size, in bytes, of buffer to allocate.

Returns:

nonzero if successful, zero on error.

See also:

PHYSFS_flush
PHYSFS_read
PHYSFS_write
PHYSFS_close

int PHYSFS_setSaneConfig (const char * organization,
const char * appName,
const char * archiveExt,

```
                    int          includeCdRoms,
                    int          archivesFirst
                   )
```

Set up sane, default paths.

Helper function.

The write dir will be set to "userdir/.organization/appName", which is created if it doesn't exist.

The above is sufficient to make sure your program's configuration directory is separated from other clutter, and platform-independent. The period before "mygame" even hides the directory on Unix systems.

The search path will be:

- The Write Dir (created if it doesn't exist)
- The Base Dir (**PHYSFS_getBaseDir()**)
- All found CD-ROM dirs (optionally)

These directories are then searched for files ending with the extension (archiveExt), which, if they are valid and supported archives, will also be added to the search path. If you specified "PKG" for (archiveExt), and there's a file named data.PKG in the base dir, it'll be checked. Archives can either be appended or prepended to the search path in alphabetical order, regardless of which directories they were found in.

All of this can be accomplished from the application, but this just does it all for you. Feel free to add more to the search path manually, too.

Parameters:

organization	Name of your company/group/etc to be used as a dirname, so keep it small, and no-frills.
appName	Program-specific name of your program, to separate it from other programs using PhysicsFS.
archiveExt	File extension used by your program to specify an archive. For example, Quake 3 uses "pk3", even though they are just zipfiles. Specify NULL to not dig out archives automatically. Do not specify the '.' char; If you want to look for ZIP files, specify "ZIP" and not ".ZIP" ... the archive search is case-insensitive.
includeCdRoms	Non-zero to include CD-ROMs in the search path, and (if (archiveExt) != NULL) search them for archives. This may cause a significant amount of blocking while discs are accessed, and if there

are no discs in the drive (or even not mounted on Unix systems), then they may not be made available anyhow. You may want to specify zero and handle the disc setup yourself.

archivesFirst	Non-zero to prepend the archives to the search path. Zero to append them. Ignored if !(archiveExt).

Returns:

nonzero on success, zero on error. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

int PHYSFS_setWriteDir (const char * newDir)

Tell PhysicsFS where it may write files.

Set a new write dir. This will override the previous setting.

This call will fail (and fail to change the write dir) if the current write dir still has files open in it.

Parameters:

newDir	The new directory to be the root of the write dir, specified in platform-dependent notation. Setting to NULL disables the write dir, so no files can be opened for writing via PhysicsFS.

Returns:

non-zero on success, zero on failure. All attempts to open a file for writing via PhysicsFS will fail until this call succeeds. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:

PHYSFS_getWriteDir

const PHYSFS_ArchiveInfo ** PHYSFS_supportedArchiveTypes (void)

Get a list of supported archive types.

Get a list of archive types supported by this implementation of PhysicFS. These are the file formats usable for search path entries. This is for informational purposes only. Note that the extension listed is merely convention: if we list "ZIP", you can open a PkZip-compatible archive with an extension of "XYZ", if you like.

The returned value is an array of pointers to **PHYSFS_ArchiveInfo** structures, with a NULL entry to signify the end of the list:

```
PHYSFS_ArchiveInfo **i;

for (i = PHYSFS_supportedArchiveTypes(); *i != NULL; i++)
{
    printf("Supported archive: [%s], which is [%s].\n",
              (*i)->extension, (*i)->description);
}
```

The return values are pointers to static internal memory, and should be considered READ ONLY, and never freed.

Returns:
READ ONLY Null-terminated array of READ ONLY structures.

PHYSFS_sint16 PHYSFS_swapSBE16 (PHYSFS_sint16 val)

Swap bigendian signed 16 to platform's native byte order.

Take a 16-bit signed value in bigendian format and convert it to the platform's native byte order.

Parameters:
val value to convert

Returns:
converted value.

PHYSFS_sint32 PHYSFS_swapSBE32 (PHYSFS_sint32 val)

Swap bigendian signed 32 to platform's native byte order.

Take a 32-bit signed value in bigendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:
converted value.

PHYSFS_sint64 PHYSFS_swapSBE64 (PHYSFS_sint64 val)

Swap bigendian signed 64 to platform's native byte order.

Take a 64-bit signed value in bigendian format and convert it to the platform's native byte order.

Parameters:
val value to convert

Returns:
converted value.

Warning:
Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

PHYSFS_sint16 PHYSFS_swapSLE16 (PHYSFS_sint16 val)

Swap littleendian signed 16 to platform's native byte order.

Take a 16-bit signed value in littleendian format and convert it to the platform's native byte order.

Parameters:
val value to convert

Returns:
converted value.

PHYSFS_sint32 PHYSFS_swapSLE32 (PHYSFS_sint32 val)

Swap littleendian signed 32 to platform's native byte order.

Take a 32-bit signed value in littleendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:

converted value.

PHYSFS_sint64 PHYSFS_swapSLE64 (PHYSFS_sint64 val)

Swap littleendian signed 64 to platform's native byte order.

Take a 64-bit signed value in littleendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:

converted value.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

PHYSFS_uint16 PHYSFS_swapUBE16 (PHYSFS_uint16 val)

Swap bigendian unsigned 16 to platform's native byte order.

Take a 16-bit unsigned value in bigendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:

converted value.

PHYSFS_uint32 PHYSFS_swapUBE32 (PHYSFS_uint32 val)

Swap bigendian unsigned 32 to platform's native byte order.

Take a 32-bit unsigned value in bigendian format and convert it to the platform's native byte order.

Parameters:
val value to convert

Returns:
converted value.

PHYSFS_uint64 PHYSFS_swapUBE64 (PHYSFS_uint64 val)

Swap bigendian unsigned 64 to platform's native byte order.

Take a 64-bit unsigned value in bigendian format and convert it to the platform's native byte order.

Parameters:
val value to convert

Returns:
converted value.

Warning:
Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

PHYSFS_uint16 PHYSFS_swapULE16 (PHYSFS_uint16 val)

Swap littleendian unsigned 16 to platform's native byte order.

Take a 16-bit unsigned value in littleendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:

converted value.

PHYSFS_uint32 PHYSFS_swapULE32 (PHYSFS_uint32 val)

Swap littleendian unsigned 32 to platform's native byte order.

Take a 32-bit unsigned value in littleendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:

converted value.

PHYSFS_uint64 PHYSFS_swapULE64 (PHYSFS_uint64 val)

Swap littleendian unsigned 64 to platform's native byte order.

Take a 64-bit unsigned value in littleendian format and convert it to the platform's native byte order.

Parameters:

val value to convert

Returns:

converted value.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

int PHYSFS_symbolicLinksPermitted (void)

Determine if the symbolic links are permitted.

This reports the setting from the last call to **PHYSFS_permitSymbolicLinks()**. If **PHYSFS_permitSymbolicLinks()** hasn't been called since the library was last initialized, symbolic links are implicitly disabled.

Returns:
non-zero if symlinks are permitted, zero if not.

See also:
PHYSFS_permitSymbolicLinks

PHYSFS_sint64 PHYSFS_tell (PHYSFS_File * handle)

Determine current position within a PhysicsFS filehandle.

Parameters:
handle handle returned from PHYSFS_open*().

Returns:
offset in bytes from start of file. -1 if error occurred. Specifics of the error can be gleaned from **PHYSFS_getLastError()**.

See also:
PHYSFS_seek

```
void PHYSFS_utf8FromLatin1 ( const char *     src,
                             char *           dst,
                             PHYSFS_uint64    len
                           )
```

Convert a UTF-8 string to a Latin1 string.

Latin1 strings are 8-bits per character: a popular "high ASCII" encoding.

To ensure that the destination buffer is large enough for the conversion, please allocate a buffer that is double the size of the source buffer. UTF-8 expands latin1 codepoints over 127 from 1 to 2 bytes, so the string may grow in some cases.

Strings that don't fit in the destination buffer will be truncated, but will always be null-terminated and never have an incomplete UTF-8 sequence at the end.

Please note that we do not supply a UTF-8 to Latin1 converter, since Latin1 can't express most Unicode codepoints. It's a legacy encoding; you should be converting away from it at all times.

Parameters:

src	Null-terminated source string in Latin1 format.
dst	Buffer to store converted UTF-8 string.
len	Size, in bytes, of destination buffer.

```
void PHYSFS_utf8FromUcs2 ( const PHYSFS_uint16 *  src,
                           char *                 dst,
                           PHYSFS_uint64          len
                         )
```

Convert a UCS-2 string to a UTF-8 string.

UCS-2 strings are 16-bits per character: `TCHAR` on Windows, when building with Unicode support.

To ensure that the destination buffer is large enough for the conversion, please allocate a buffer that is double the size of the source buffer. UTF-8 never uses more than 32-bits per character, so while it may shrink a UCS-2 string, it may also expand it.

Strings that don't fit in the destination buffer will be truncated, but will always be null-terminated and never have an incomplete UTF-8 sequence at the end.

Please note that UCS-2 is not UTF-16; we do not support the "surrogate" values at this time.

Parameters:

src Null-terminated source string in UCS-2 format.
dst Buffer to store converted UTF-8 string.
len Size, in bytes, of destination buffer.

```
void PHYSFS_utf8FromUcs4 ( const PHYSFS_uint32 *  src,
                           char *                 dst,
                           PHYSFS_uint64          len
                         )
```

Convert a UCS-4 string to a UTF-8 string.

UCS-4 strings are 32-bits per character: `wchar_t` on Unix.

To ensure that the destination buffer is large enough for the conversion, please allocate a buffer that is the same size as the source buffer. UTF-8 never uses more than 32-bits per character, so while it may shrink a UCS-4 string, it will never expand it.

Strings that don't fit in the destination buffer will be truncated, but will always be null-terminated and never have an incomplete UTF-8 sequence at the end.

Parameters:

src Null-terminated source string in UCS-4 format.
dst Buffer to store converted UTF-8 string.
len Size, in bytes, of destination buffer.

```
PHYSFS_utf8ToUcs2 ( const char *      src,
                    PHYSFS_uint16 *   dst,
                    PHYSFS_uint64     len
                  )
```

Convert a UTF-8 string to a UCS-2 string.

UCS-2 strings are 16-bits per character: `TCHAR` on Windows, when building with Unicode support.

To ensure that the destination buffer is large enough for the conversion, please allocate a buffer that is double the size of the source buffer. UTF-8 uses from one to four bytes per character, but UCS-2 always uses two, so an entirely low-ASCII string will double in size!

Strings that don't fit in the destination buffer will be truncated, but will always be null-terminated and never have an incomplete UCS-2 sequence at the end.

Please note that UCS-2 is not UTF-16; we do not support the "surrogate" values at this time.

Parameters:

src	Null-terminated source string in UTF-8 format.
dst	Buffer to store converted UCS-2 string.
len	Size, in bytes, of destination buffer.

```
void PHYSFS_utf8ToUcs4 ( const char *        src,
                         PHYSFS_uint32 *     dst,
                         PHYSFS_uint64       len
                       )
```

Convert a UTF-8 string to a UCS-4 string.

UCS-4 strings are 32-bits per character: `wchar_t` on Unix.

To ensure that the destination buffer is large enough for the conversion, please allocate a buffer that is four times the size of the source buffer. UTF-8 uses from one to four bytes per character, but UCS-4 always uses four, so an entirely low-ASCII string will quadruple in size!

Strings that don't fit in the destination buffer will be truncated, but will always be null-terminated and never have an incomplete UCS-4 sequence at the end.

Parameters:

src	Null-terminated source string in UTF-8 format.
dst	Buffer to store converted UCS-4 string.

len	Size, in bytes, of destination buffer.

PHYSFS_sint64 PHYSFS_write (PHYSFS_File * handle, const void * buffer, PHYSFS_uint32 objSize, PHYSFS_uint32 objCount)

Write data to a PhysicsFS filehandle.

The file must be opened for writing.

Parameters:

handle	retval from **PHYSFS_openWrite()** or **PHYSFS_openAppend()**.
buffer	buffer to store read data into.
objSize	size in bytes of objects being read from (handle).
objCount	number of (objSize) objects to read from (handle).

Returns:

number of objects written. **PHYSFS_getLastError()** can shed light on the reason this might be < (objCount). -1 if complete failure.

int PHYSFS_writeSBE16 (PHYSFS_File * file, PHYSFS_sint16 val)

Convert and write a signed 16-bit bigendian value.

Convenience function. Convert a signed 16-bit value from the platform's native byte order to bigendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.
val Value to convert and write.

Returns:
zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeSBE32 ( PHYSFS_File *   file,
                        PHYSFS_sint32   val
                      )
```

Convert and write a signed 32-bit bigendian value.

Convenience function. Convert a signed 32-bit value from the platform's native byte order to bigendian and write it to a file.

Parameters:
file PhysicsFS file handle to which to write.
val Value to convert and write.

Returns:
zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeSBE64 ( PHYSFS_File *   file,
                        PHYSFS_sint64   val
                      )
```

Convert and write a signed 64-bit bigending value.

Convenience function. Convert a signed 64-bit value from the platform's native byte order to bigendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.

val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

```
int PHYSFS_writeSLE16 ( PHYSFS_File *   file,
                        PHYSFS_sint16   val
                      )
```

Convert and write a signed 16-bit littleendian value.

Convenience function. Convert a signed 16-bit value from the platform's native byte order to littleendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.

val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeSLE32 ( PHYSFS_File *   file,
                        PHYSFS_sint32   val
                      )
```

Convert and write a signed 32-bit littleendian value.

Convenience function. Convert a signed 32-bit value from the platform's native byte order to littleendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.

val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeSLE64 ( PHYSFS_File *   file,
                        PHYSFS_sint64  val
                      )
```

Convert and write a signed 64-bit littleendian value.

Convenience function. Convert a signed 64-bit value from the platform's native byte order to littleendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.

val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

```
int PHYSFS_writeUBE16 ( PHYSFS_File *   file,
                        PHYSFS_uint16  val
```

)

Convert and write an unsigned 16-bit bigendian value.

Convenience function. Convert an unsigned 16-bit value from the platform's native byte order to bigendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.

val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeUBE32 ( PHYSFS_File *   file,
                        PHYSFS_uint32   val
                      )
```

Convert and write an unsigned 32-bit bigendian value.

Convenience function. Convert an unsigned 32-bit value from the platform's native byte order to bigendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.

val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeUBE64 ( PHYSFS_File *   file,
```

PHYSFS_uint64 val
)

Convert and write an unsigned 64-bit bigendian value.

Convenience function. Convert an unsigned 64-bit value from the platform's native byte order to bigendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.
val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.

int PHYSFS_writeULE16 (PHYSFS_File * file,
PHYSFS_uint16 val
)

Convert and write an unsigned 16-bit littleendian value.

Convenience function. Convert an unsigned 16-bit value from the platform's native byte order to littleendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.
val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from

PHYSFS_getLastError().

```
int PHYSFS_writeULE32 ( PHYSFS_File *   file,
                        PHYSFS_uint32   val
                      )
```

Convert and write an unsigned 32-bit littleendian value.

Convenience function. Convert an unsigned 32-bit value from the platform's native byte order to littleendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.
val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

```
int PHYSFS_writeULE64 ( PHYSFS_File *   file,
                        PHYSFS_uint64   val
                      )
```

Convert and write an unsigned 64-bit littleendian value.

Convenience function. Convert an unsigned 64-bit value from the platform's native byte order to littleendian and write it to a file.

Parameters:

file PhysicsFS file handle to which to write.
val Value to convert and write.

Returns:

zero on failure, non-zero on success. On failure, you can find out what went wrong from **PHYSFS_getLastError()**.

Warning:

Remember, PHYSFS_uint64 is only 32 bits on platforms without any sort of 64-bit support.
