Device File Access Guide for Storage Devices

Hardware & Drivers > Storage

2007-03-06

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to Device File Access Guide for Storage Devices

This document describes how to communicate with a storage device from an application running in Mac OS X. Before you read this document, you should be familiar with the I/O Kit's device interface mechanism and device matching in particular. To learn about these things, read *Accessing Hardware From Applications*.

Organization of This Document

This document contains the following chapters:

- "Working With Device Files for Storage Devices" (page 9) guides you through a sample application that uses I/O Kit functions to find a storage device and POSIX functions to communicate with it.
- "Document Revision History" (page 17) lists the changes to this document.

See Also

The ADC Reference Library contains documents that describe various types and aspects of device access, as well as numerous sample applications.

- *I/O Kit Fundamentals* describes the I/O Kit (the object-oriented driver-development framework of Mac OS X) and provides an overview of device-access options for applications.
- *Accessing Hardware From Applications* describes many ways applications can access devices and provides in-depth information on the device interface mechanism of the I/O Kit.
- *I/O Kit Framework Reference* contains API reference for I/O Kit methods and functions and for specific device families, such as USB.
- Mac OS X Man Pages provides access to existing reference documentation for BSD and POSIX functions and tools in a convenient, HTML format.

If you're ready to create a universal binary version of your storage device-access application to run in an Intel-based Macintosh, see *Universal Binary Programming Guidelines, Second Edition*. That document describes the differences between the Intel and PowerPC architectures and provides tips for developing a universal binary.

A detailed description of the UNIX file system is beyond the scope of this document, but there are many books and websites you can refer to. In particular, you can get information on the POSIX standard at http://standards.ieee.org.

Working With Device Files for Storage Devices

This chapter describes how to develop an application that uses I/O Kit and POSIX functions to locate a CD-ROM storage device on Mac OS X and open it for reading.

The code snippets in this chapter are based on the sample application *CDROMSample*, available in its entirety at Sample Code > Hardware & Drivers > Storage.

Note: If you choose to develop a Cocoa application that accesses a storage device, be aware that Objective-C does not provide interfaces for I/O Kit or POSIX functions. However, because the I/O Kit and POSIX APIs are C APIs, you can call them from a Cocoa application.

Although the sample code in this chapter has been compiled and tested to some degree, Apple does not recommend that you directly incorporate this code into a commercial application. For example, only limited error handling is shown—you should develop your own techniques for detecting and handling errors.

Important: The sample code in this chapter requires Mac OS X v10.1 or later to build, but the resulting application will run in Mac OS X v10.0 or later.

Storage Device Access in an Intel-Based Macintosh

This section briefly outlines some of the issues related to developing a universal binary version of a Mac OS X application that uses device files to access a storage device. Before you read this section, be sure to read *Universal Binary Programming Guidelines, Second Edition*. That document covers architectural differences and byte-ordering formats and provides comprehensive guidelines for code modification and building universal binaries. The guidelines in that document apply to all types of applications, including those that access hardware.

Before you build your application as a universal binary, make sure that:

- You port your project to GCC 4 (Xcode uses GCC 4 to target Intel-based Macintosh computers)
- You install the Mac OS X v10.4 universal SDK
- You develop your project in Xcode 2.1 or later

An application that reads from and writes to storage media frequently handles data structures that contain multibyte integer data. It's vital that these data structures remain in the correct endian format on the disk so the disk can be used with both PowerPC-based and Intel-based Macintosh computers. Depending on the native endian format of the computer in which the application is running, therefore, the application may need to byte swap the data structures it handles.

If you've determined that byte-swapping is required in your application, you can implement it in one of two ways:

- Perform the appropriate byte swap on the data after it's read into a buffer and perform the opposite byte swap on the data that's ready to be written out to disk. This scheme allows your application to access the buffers without having to worry about the endian format of the data in them.
- Do not byte swap the data in the buffers, but perform the appropriate byte swap each time your application accesses the buffers. This preserves the data's correct endian format while it resides in the buffers, which means your application does not have to byte swap the data while reading it in or writing it out.

To avoid confusion, it's best to choose only one of these two schemes and be consistent in its implementation throughout your application. Whichever you choose, however, be sure to use the conditional byte-swapping macros defined in `libkern/OSByteOrder.h` (even though this header file is in the Kernel framework, its macros are available to applications). When you use these macros, the compiler optimizes your code so the routines are executed only if they are necessary for the architecture in which your application is running.

Accessing a CD-ROM Storage Device

To communicate with a storage device (such as a CD-ROM device) from your Mac OS X application, you use I/O Kit functions to find the device and obtain a path to its device file. You can then use POSIX functions to perform such operations as opening and closing the device and reading from it.

The sample code in this chapter demonstrates how to find all ejectable CD media, obtain the path to the device file for a CD-ROM drive, and use POSIX functions to open the device, read a sector of the media and close the device. Your application can read data using POSIX functions because, depending on the permissions, the file system may allow multiple users to open a file for reading. However, you should not assume you can use this mechanism to write data, because the file system itself may have opened all writable mounted storage devices with restrictive write access.

The sample code shown in this chapter is from an Xcode "CoreFoundation Tool" project. The project builds a tool that has no user interface and sends its output to the console. You can view the output either by running the tool within Xcode or by running the Console utility, which you can find at `/Applications/Utilities/Console`, before launching the tool.

Note: By convention, application functions in this chapter, such as `MyOpenDrive` in Listing 1-5 (page 15), start with `My` to distinguish them from I/O Kit functions and other Mac OS X functions.

If you are using a version of Mac OS X prior to v10.1, this tool must be run with root privileges, because the `/dev/rdisk*` nodes are owned by root in those versions. In Mac OS X v10.1 and later, the `/dev/*disk*` nodes for removable media are owned by the currently logged-in user (nodes for nonremovable media are still owned by root). If necessary, you can use the `sudo(8)` command to launch the tool with root privileges, as shown below (you will be asked to supply your `admin` password):

```
sudo open /YourDirectoryPath/CDROMSample.app
```

Including Header Files and Setting Up a Main Function

Listing 1-1 shows the header files you'll need to include in your main file for the sample code in this chapter. (Some of these headers include others; a shorter list is possible.) Except for `CoreFoundation.h`, these headers are generally part of `IOKit.framework` or `System.framework`.

Listing 1-1 Header files to include for the storage device sample code

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <paths.h>
#include <sys/param.h>
#include <IOKit/IOKitLib.h>
#include <IOKit/IOBSD.h>
#include <IOKit/storage/IOCDMedia.h>
#include <IOKit/storage/IOMedia.h>
#include <IOKit/storage/IOCDTypes.h>
#include <IOKit/storage/IOMediaBSDClient.h>
#include <CoreFoundation/CoreFoundation.h>
```

Listing 1-2 shows a `main` function for finding a CD-ROM device with the I/O Kit and accessing it with POSIX functions. The `main` function accomplishes its work by calling the following functions, which are shown in other sections:

- `MyFindEjectableCDMedia` ("Finding All Ejectable CD Media" (page 12))
- `MyGetDeviceFilePath` ("Getting the Path to the Device File for the CD-ROM Device" (page 13))
- `MyOpenDrive` ("Opening the Device" (page 15))
- `MyReadSector` ("Reading a Sector From the Device" (page 15))
- `MyCloseDrive` ("Closing the Device" (page 16))

The type `kern_return_t` is defined in `std_types.h`.

The constant `KERN_SUCCESS` is defined in `kern_return.h`.

Listing 1-2 Finding a CD-ROM device and reading a sector

```
int main( void )
{
    kern_return_t kernResult;
    io_iterator_t mediaIterator;
    char deviceFilePath[ MAXPATHLEN ];

    kernResult = MyFindEjectableCDMedia( &mediaIterator );
    if ( kernResult != KERN_SUCCESS )
        return 0;

    kernResult = MyGetDeviceFilePath( mediaIterator, deviceFilePath,
                    sizeof( deviceFilePath ) );
    if ( kernResult != KERN_SUCCESS )
```

```
        return 0;

    // Now open the device we found, read a sector, and close the device.
    if ( deviceFilePath[ 0 ] != '\0' )
    {
        int fileDescriptor;

        fileDescriptor = MyOpenDrive( deviceFilePath );
        if (fileDescriptor != -1 )
        {
            if ( MyReadSector( fileDescriptor ) )
                printf( "Sector read successfully.\n" );
            else
                printf( "Could not read sector.\n" );

            MyCloseDrive( fileDescriptor );
            printf( "Device closed.\n" );
        }
    }
    else
        printf( "No ejectable CD media found.\n" );

    // Release the iterator.
    IOObjectRelease( mediaIterator );

    return 0;
}
```

The `main` function releases the iterator returned by the `MyFindEjectableCDMedia` function, which also releases the iterator's objects.

Finding All Ejectable CD Media

The `MyFindEjectableCDMedia` function, shown in Listing 1-3 (page 13), establishes a connection to the I/O Kit by calling the `IOMasterPort` function, which returns a Mach port. It then creates a matching dictionary by calling `IOServiceMatching`, passing the constant `kIOCDMediaClass` (defined in `IOCDMedia.h`). This sets up a dictionary that matches all devices with a provider class of IOCDMediaClass; all CD media devices in the I/O Registry are instances of this class or a subclass.

A matching dictionary is a dictionary of key-value pairs that describe the properties of an I/O Kit device or other service. Each IOMedia object in the I/O Registry has a property with key `kIOMediaEjectableKey` and a value that is `true` if the media is indeed ejectable. In this sample, we are interested only in ejectable media, so the `MyFindEjectableCDMedia` function refines the matching dictionary by calling `CFDictionarySetValue` to add the key `kIOMediaEjectableKey` and value `kCFBooleanTrue`.

The constants `kIOMediaEjectableKey` and `kIOCDMediaClass` are defined in `IOMedia.h` in `Kernel.framework`. The constant `kCFBooleanTrue` is a Core Foundation constant. If you need more information on the process of using matching dictionaries to find devices in the I/O Registry, see *Accessing Hardware From Applications*.

Next, `MyFindEjectableCDMedia` passes the dictionary to the I/O Kit function `IOServiceGetMatchingServices` to obtain an iterator object that identifies all CD-ROM devices with ejectable media in the I/O Registry. If successful, `MyFindEjectableCDMedia` uses its pointer parameter to return the iterator object. The calling function is responsible for releasing this object.

Finally, `MyFindEjectableCDMedia` returns a result value that indicates whether it found any ejectable CD media. The constant `KERN_SUCCESS` is defined in `kern_return.h`.

Listing 1-3 Finding all ejectable CD media

```
kern_return_t MyFindEjectableCDMedia( io_iterator_t *mediaIterator )
{
    mach_port_t         masterPort;
    kern_return_t       kernResult;
    CFMutableDictionaryRef   classesToMatch;

    kernResult = IOMasterPort( MACH_PORT_NULL, &masterPort );
    if ( kernResult != KERN_SUCCESS )
    {
        printf( "IOMasterPort returned %d\n", kernResult );
        return kernResult;
    }
    // CD media are instances of class kIOCDMediaClass.
    classesToMatch = IOServiceMatching( kIOCDMediaClass );
    if ( classesToMatch == NULL )
        printf( "IOServiceMatching returned a NULL dictionary.\n" );
    else
    {
        // Each IOMedia object has a property with key kIOMediaEjectableKey
        // which is true if the media is indeed ejectable. So add this
        // property to the CFDictionary for matching.
        CFDictionarySetValue( classesToMatch,
                        CFSTR( kIOMediaEjectableKey ), kCFBooleanTrue );
    }
    kernResult = IOServiceGetMatchingServices( masterPort,
                                classesToMatch, mediaIterator );
    if ( (kernResult != KERN_SUCCESS) || (*mediaIterator == NULL) )
        printf( "No ejectable CD media found.\n kernResult = %d\n",
                    kernResult );
    return kernResult;
}
```

Getting the Path to the Device File for the CD-ROM Device

Listing 1-4 (page 14) shows the `MyGetDeviceFilePath` function. The parameters to this function specify an iterator over ejectable CD media devices, a pointer to storage for the device file path, and the maximum size of the path. The function returns, in the `deviceFilePath` parameter, the path to the device file, including filename, for the first such device it finds in the iterator.

The `MyGetDeviceFilePath` function examines the first object in the passed iterator. Although many computers have just one CD-ROM device, the iterator could actually contain objects for multiple devices; however, this function looks at only the first.

The `MyGetDeviceFilePath` function performs the following steps:

1. It calls the I/O Kit function `IORegistryEntryCreateCFProperty`, passing the key `kIOBSDNameKey` (defined in `IOBSD.h`), to obtain a `CFTypeRef` to the device file name.

2. If the call to `IORegistryEntryCreateCFProperty` is successful, `MyGetDeviceFilePath` constructs a device path to the device. To do this, the function:

- Copies the string '/dev/' (defined by the constant _PATH_DEV in the header paths.h) to the storage location specified by the deviceFilePath parameter
- Concatenates the string 'r' to the end of the device path to ensure that the code accesses the raw device
- Calls the CFStringGetCString function to encode the Core Foundation representation of the device name as a C string

3. If MyGetDeviceFilePath is able to create the device-file name successfully, it prints the string and releases the CFTypeRef. The full device file name will be something like /dev/rdisk0.

The IOIteratorNext function retains each media object it returns, so the MyGetDeviceFilePath function releases the iterator objects it examines. The calling function is responsible for releasing the iterator itself, which also releases the iterator's objects.

Finally, MyGetDeviceFilePath returns a result value that indicates whether the function successfully obtained a device path for a CD-ROM device.

Listing 1-4 Getting the device file path for the first ejectable CD media in a passed iterator

```
kern_return_t MyGetDeviceFilePath( io_iterator_t mediaIterator,
                        char *deviceFilePath, CFIndex maxPathSize )
{
    io_object_t nextMedia;
    kern_return_t kernResult = KERN_FAILURE;

    *deviceFilePath = '\0';
    nextMedia = IOIteratorNext( mediaIterator );
    if ( nextMedia )
    {
        CFTypeRef   deviceFilePathAsCFString;
        deviceFilePathAsCFString = IORegistryEntryCreateCFProperty(
                                nextMedia, CFSTR( kIOBSDNameKey ),
                                kCFAllocatorDefault, 0 );
       *deviceFilePath = '\0';
        if ( deviceFilePathAsCFString )
        {
            size_t devPathLength;
            strcpy( deviceFilePath, _PATH_DEV );
            // Add "r" before the BSD node name from the I/O Registry
            // to specify the raw disk node. The raw disk node receives
            // I/O requests directly and does not go through the
            // buffer cache.
            strcat( deviceFilePath, "r");
            devPathLength = strlen( deviceFilePath );
            if ( CFStringGetCString( deviceFilePathAsCFString,
                                    deviceFilePath + devPathLength,
                                    maxPathSize - devPathLength,
                                    kCFStringEncodingASCII ) )
            {
                printf( "BSD path: %s\n", deviceFilePath );
                kernResult = KERN_SUCCESS;
            }
            CFRelease( deviceFilePathAsCFString );
        }
```

```
    }
    IOObjectRelease( nextMedia );

    return kernResult;
}
```

Opening the Device

To open a CD media device, the `MyOpenDrive` function, shown in Listing 1-5, calls the `open` function, passing a device-file path and the constant `O_RDONLY`, which indicates the device should be opened for reading only. The `open` function and `O_RDONLY` are both defined in `fcntl.h`, which is part of `System.framework`. You can get more information about the `open` function by typing `man 2 open` in a Terminal window.

The `MyOpenDrive` function returns the value it gets from the `open` function; on error, it also prints an error message.

Listing 1-5 Opening a device specified by its device file path

```
int MyOpenDrive( const char *deviceFilePath )
{
    int fileDescriptor;

    fileDescriptor = open( deviceFilePath, O_RDONLY );
    if ( fileDescriptor == -1 )
    {
        printf( "Error opening device %s: \n", deviceFilePath );
        perror( NULL );
    }
    return fileDescriptor;
}
```

Reading a Sector From the Device

Listing 1-6 shows a function, `MyReadSector`, that reads a sector of the media. The caller of this function passes the file descriptor for a device file. The device is assumed to be open. `MyReadSector` first uses the `DKIOCGETBLOCKSIZE` ioctl to get the preferred block size for the media. Then, it allocates a buffer of the preferred block size and attempts to read a sector, using the `read` function defined in the `unistd.h`.

Listing 1-6 Reading a sector of the media, given the file descriptor

```
Boolean MyReadSector( int fileDescriptor )
{
    char *buffer;
    size_t numBytes;
    u_int32_t blockSize;

    if ( ioctl( fileDescriptor, DKIOCGETBLOCKSIZE, &blockSize ) == -1)
    {
        perror( "Error getting preferred block size." );
        // Set a reasonable block size instead.
        // kCDSectorSizeCDDA is defined in IOCDTypes.h as 2352.
        blockSize = kCDSectorSizeCDDA;
    }
```

```
    buffer = malloc( blockSize );
    numBytes = read( fileDescriptor, buffer, blockSize );
    free( buffer );
    return numBytes == blockSize ? true : false;
}
```

Closing the Device

Listing 1-7 shows the `MyCloseDrive` function. To close the CD-ROM device, `MyCloseDrive` calls the `close` function (defined in `unistd.h`), passing the file descriptor for the device file. The file descriptor was obtained by the `MyOpenDrive` function.

Listing 1-7 Closing a device, given its file descriptor

```
void MyCloseDrive( int fileDescriptor )
{
    close( fileDescriptor );
}
```

Document Revision History

This table describes the changes to *Device File Access Guide for Storage Devices*.

Date	Notes
2007-03-06	Made minor corrections.
2005-12-06	Made minor corrections.
2005-09-08	Added information about endian issues. Changed title from "Working With Device Files for Storage Devices."
2003-05-01	First version of *Working With Device Files for Storage Devices*. This document comprises one chapter from an earlier version of *Inside Mac OS X: Accessing Hardware From Applications* .

Document Revision History