Device File Access Guide for Serial Devices

Hardware & Drivers > Serial

2005-12-06

Contents

Listings

Introduction to Device File Access Guide for Serial I/O

This document describes how to communicate with a serial 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 a Serial Device" (page 9) guides you through a sample application that communicates with a serial device that claims to be a modem.
- "Document Revision History" (page 25) lists the changes to this document.

See Also

Apple developer documentation provides documents that describe various types and aspects of device access.

- *I/O Kit Fundamentals* describes the I/O Kit (the object-oriented driver-development framework of Mac OS X) and provides an overview of application-level device access.
- *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 serial 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.

See Also

Working With a Serial Device

This chapter provides sample code that demonstrates how to use the device-file mechanism to find and communicate with a serial device that explicitly claims to be a modem, such as a built-in or USB modem. Because there is no safe, programmatic way to determine if a device attached to a serial port is indeed a modem, the sample code in this chapter does not find modems on the other side of a serial port.

The code snippets in this chapter are from the sample application `SerialPortSample`, available in its entirety at http://developer.apple.com/samplecode/SerialPortSample.

> **Note:** If you choose to develop a Cocoa application that accesses a serial modem, 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 is written to run in Mac OS X v10.0 or later.

Serial 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 serial 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

If your application uses the POSIX API in a standard manner, you should have no trouble developing a universal binary version of your application. As with any device-access application, however, if you read multibyte integer data (instead of or in addition to character streams) you need to be aware of potential differences in endian format.

If you determine that byte swapping is necessary, keep the following guidelines in mind:

■ Avoid unconditional byte-swapping code that always swaps from one endian format to the other. Instead, use the conditional byte-swapping macros defined in `libkern/OSByteOrder.h`. Even though this header file is in the Kernel framework, the macros are available to applications. Using these macros means the compiler optimizes your code so the routines are executed only if they are necessary for the architecture in which your application is running.

■ Be consistent about when you swap bytes. You might choose to 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 is ready to be written out to the device. Or, you might choose to perform all byte swapping on the data as it's being read into or written out from the buffer. The first option means the data in the buffer is in the correct endian format for the device and the second option means the data in the buffer is in the correct endian format for the architecture in which your application is running. Whichever alternative you choose, implement it consistently throughout your application.

Accessing a Serial Device

To communicate with a serial device from your Mac OS X application, use I/O Kit functions to obtain a path to the device file for that device. Then, implement traditional UNIX serial port access using the POSIX `termios` API. Your application can read and write data using the device file.

Specifically, the sample code in this chapter demonstrates how to:

■ Find all serial devices that claim to be modems

■ Obtain the path to the device file of the first modem found

■ Use the POSIX API to initialize the modem and communicate with it through the device file

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`. You can, of course, write similar code without these restrictions. For detailed documentation on using Xcode, see http://developer.apple.com/referencelibrary/Developer-Tools/index.html.

Many functions, data types, and constants used in the sample code in this chapter are defined in header files in `Kernel.framework`, `System.framework`, or in the directory `/usr/include` (whose contents you can examine using the Terminal application, located in `/Applications/Utilities/Terminal`). Specific header files are noted where appropriate. Some functions and data types, such as those for working with the `CFStringRef` type, are defined in header files in `CoreFoundation.framework`.

Some functions and data types used in this chapter are described in UNIX man pages. To view the reference documentation for these, see *Mac OS X Man Pages*. Alternatively, you can view the documentation by typing man*function_name* (for example, `man tcsetattr`) in a Terminal window. Many of the code snippets in this chapter refer to specific man pages in the code comments.

Including Header Files and Defining Macros and Constants

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 `Kernel.framework`

Listing 1-1   Header files to include for the serial port modem 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 <termios.h>
#include <sysexits.h>
#include <sys/param.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>

#include <CoreFoundation/CoreFoundation.h>

#include <IOKit/IOKitLib.h>
#include <IOKit/serial/IOSerialKeys.h>
#include <IOKit/IOBSD.h>
```

By default, Apple internal modems define `local echo` to be "on". If you're using the sample code in this chapter to find and communicate with another type of modem, you should undefine the macro shown in Listing 1-2.

Listing 1-2   Macro to define appropriate modem-response string

```
#define LOCAL_ECHO

#ifdef LOCAL_ECHO
#define kOKResponseString "AT\r\r\nOK\r\n"
#else
#define kOKResponseString "\r\nOK\r\n"
```

The sample code also defines the constants shown in Listing 1-3.

Listing 1-3   Constants used in the serial port modem sample code

```
#define kATCommandString        "AT\r"
#define kMyErrReturn            -1

enum
{
    kNumRetries = 3
};
```

Serial port attributes, such as timeouts and baud rates are stored in the `termios` structure. The sample code defines a global static structure to store the device's current attributes so it can restore them after changing them.

```
static struct termios gOriginalTTYAttrs;
```

Setting Up a Main Function

Listing 1-4 (page 12) shows a `main` function that uses I/O Kit functions to find a modem and POSIX functions to access it. The `main` function accomplishes its work by calling the following functions, which are shown in other sections:

- `MyFindModems` ("Finding All Modems" (page 13))
- `MyGetModemPath` ("Getting the Path to the Device File for a Modem" (page 15))
- `MyOpenSerialPort` ("Opening the Serial Port" (page 16))
- `MyInitializeModem` ("Communicating With the Modem" (page 20))
- `MyCloseSerialPort` ("Closing the Serial Port" (page 22))

The types `io_iterator_t` and `kern_return_t` are defined in the header files `IOTypes.h` and `std_types.h`, respectively.

The constants `EX_UNAVAILABLE`, `EX_IOERR`, and `EX_OK` are defined in the header file `sysexits.h`.

Listing 1-4   Setting up a main function for finding and accessing a modem

```
int main(void)
{
    int         fileDescriptor;
    kern_return_t   kernResult;

    io_iterator_t   serialPortIterator;
    char        deviceFilePath[MAXPATHLEN];

    kernResult = MyFindModems(&serialPortIterator);

    kernResult = MyGetModemPath(serialPortIterator, deviceFilePath,
                sizeof(deviceFilePath));

    IOObjectRelease(serialPortIterator);    // Release the iterator.

    // Open the modem port, initialize the modem, then close it.
    if (!deviceFilePath[0])
    {
        printf("No modem port found.\n");
        return EX_UNAVAILABLE;
    }

    fileDescriptor = OpenSerialPort(deviceFilePath);
    if (fileDescriptor == kMyErrReturn)
    {
        return EX_IOERR;
```

```
    }

    if (MyInitializeModem(fileDescriptor))
    {
        printf("Modem initialized successfully.\n");
    }
    else {
        printf("Could not initialize modem.\n");
    }

    MyCloseSerialPort(fileDescriptor);
    printf("Modem port closed.\n");

    return EX_OK;
}
```

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

Finding All Modems

The `MyFindModems` function, shown in Listing 1-5 (page 14), 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 `kIOSerialBSDServiceValue`. This sets up a dictionary that matches all devices with a provider class of IOSerialBSDClient.

A matching dictionary is a dictionary of key-value pairs that describes the properties of an I/O Kit device or other service. Each serial device object in the I/O Registry has a property with key `kIOSerialBSDTypeKey`. The possible values of this key are:

■   `kIOSerialBSDAllTypes`

■   `kIOSerialBSDModemType`

■   `kIOSerialBSDRS232Type`

This sample project is interested only in modems, so the `MyFindModems` function refines the matching dictionary by calling `CFDictionarySetValue` to add the key `kIOSerialBSDTypeKey` and value `kIOSerialBSDModemType`. (Remember, if there are modems on the other side of a serial port, this sample code will not find them.)

If you want to modify the sample code to find a different type of serial device, you can give the `kIOSerialBSDTypeKey` key one of the other values. The comments following the call to `CFDictionarySetValue` in Listing 1-5 describe how to do this.

Finally, `MyFindModems` passes the dictionary to the I/O Kit function `IOServiceGetMatchingServices` to obtain an iterator object that identifies all modem devices in the I/O Registry. If successful, `MyFindModems` uses its pointer parameter to return the iterator object. The calling function is responsible for releasing this object.

Constants such as `kIOSerialBSDTypeKey` and `kIOSerialBSDModemType` are defined in the header file `IOSerialKeys.h`. If you need more information on the process of finding devices in the I/O Registry, see *Accessing Hardware From Applications*.

The constant `KERN_SUCCESS` is defined in the header file `kern_return.h`.

Listing 1-5   Finding all serial port modems in the current system

```
static kern_return_t MyFindModems(io_iterator_t *matchingServices)
{
    kern_return_t        kernResult;
    mach_port_t          masterPort;
    CFMutableDictionaryRef  classesToMatch;

    kernResult = IOMasterPort(MACH_PORT_NULL, &masterPort);
    if (KERN_SUCCESS != kernResult)
    {
        printf("IOMasterPort returned %d\n", kernResult);
        goto exit;
    }

    // Serial devices are instances of class IOSerialBSDClient.
    classesToMatch = IOServiceMatching(kIOSerialBSDServiceValue);
    if (classesToMatch == NULL)
    {
        printf("IOServiceMatching returned a NULL dictionary.\n");
    }
    else {
        CFDictionarySetValue(classesToMatch,
                             CFSTR(kIOSerialBSDTypeKey),
                             CFSTR(kIOSerialBSDModemType));

        // Each serial device object has a property with key
        // kIOSerialBSDTypeKey and a value that is one of
        // kIOSerialBSDAllTypes, kIOSerialBSDModemType,
        // or kIOSerialBSDRS232Type. You can change the
        // matching dictionary to find other types of serial
        // devices by changing the last parameter in the above call
        // to CFDictionarySetValue.
    }

    kernResult = IOServiceGetMatchingServices(masterPort, classesToMatch,
matchingServices);
    if (KERN_SUCCESS != kernResult)
    {
        printf("IOServiceGetMatchingServices returned %d\n", kernResult);
        goto exit;
    }

exit:
    return kernResult;
}
```

Getting the Path to the Device File for a Modem

Listing 1-6 (page 15) shows the `MyGetModemPath` function. The caller of this function passes an iterator to a list of modems, 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 modem it finds in the iterator.

The main body of `MyGetModemPath` consists of a `while` loop that iterates over all the modem objects in the passed iterator. Until it finds a modem, the code in the `while` loop examines each iterator object, performing the following operations:

1. It calls the I/O Kit function `IORegistryEntryCreateCFProperty`, passing the key `kIOCalloutDeviceKey`, to obtain a `CFTypeRef` to the modem device file.

2. If successful in obtaining the modem's device file, it calls `CFStringGetCString` to obtain the full path to the device file as a C string, pointed to by the `deviceFilePath` parameter.

3. If it finds the name, it prints it, then releases the `CFTypeRef`. For example, the file path may be `/dev/cu.modem`.

The `while` loop in `MyGetModemPath` releases each iterator object it obtains while looking for a serial port modem, because the `IOIteratorNext` function retains each object it returns. The calling function (in this sample, `main`) is responsible for releasing the iterator itself, which also releases the iterator's objects.

Finally, `MyGetModemPath` returns a result value that indicates whether the function successfully obtained a device path for a modem.

Listing 1-6   Returning the device file path for the first modem in a passed iterator

```
static kern_return_t MyGetModemPath(io_iterator_t serialPortIterator, char
*deviceFilePath, CFIndex maxPathSize)
{
    io_object_t     modemService;
    kern_return_t   kernResult = KERN_FAILURE;
    Boolean     modemFound = false;

    // Initialize the returned path
    *deviceFilePath = '\0';

    // Iterate across all modems found. In this example, we exit after
    // finding the first modem.

    while !modemFound) && ((modemService = IOIteratorNext(serialPortIterator))
    {
        CFTypeRef   deviceFilePathAsCFString;

    // Get the callout device's path (/dev/cu.xxxxx).
    // The callout device should almost always be
    // used. You would use the dialin device (/dev/tty.xxxxx) when
    // monitoring a serial port for
    // incoming calls, for example, a fax listener.

        deviceFilePathAsCFString = IORegistryEntryCreateCFProperty(modemService,
                        CFSTR(kIOCalloutDeviceKey),
```

```
                        kCFAllocatorDefault,
                        0);
    if (deviceFilePathAsCFString)
    {
        Boolean result;

        // Convert the path from a CFString to a NULL-terminated C string
        // for use with the POSIX open() call.

        result = CFStringGetCString(deviceFilePathAsCFString,
                                    deviceFilePath,
                                    maxPathSize,
                                    kCFStringEncodingASCII);
        CFRelease(deviceFilePathAsCFString);

        if (result)
        {
            printf("BSD path: %s", deviceFilePath);
            modemFound = true;
            kernResult = KERN_SUCCESS;
        }
    }

    printf("\n");

    // Release the io_service_t now that we are done with it.

    (void) IOObjectRelease(modemService);
    }

    return kernResult;
}
```

Opening the Serial Port

To open a serial port, the `MyOpenSerialPort` function, shown in Listing 1-7 (page 17), calls the `open` function, passing the device file path, as well as the following constants:

- `O_RDWR`: open for reading and writing

- `O_NOCTTY`: don't assign a controlling terminal

- `O_NONBLOCK`: allow subsequent I/O on the device to be nonblocking

These constants and the `open` and `fcntl` functions are defined in `fcntl.h`.

If `open` returns a valid file descriptor, `MyOpenSerialPort` performs the following additional steps:

1. It calls the `ioctl` function, passing `TIOCEXCL`, to prevent additional opens on the device, except from a root-owned process.

2. It calls the `fcntl` function, passing the value `F_SETFL` to clear the `O_NONBLOCK` flag so subsequent I/O will block.

3. It calls the `tcgetattr` function to save the current file settings in the global static structure `gOriginalTTYAttrs`, of type `termios`. These values will be restored later by the `MyCLoseSerialPort` function (Listing 1-10 (page 23)). The `termios` structure and the `tcgetattr` and `tcsetattr` functions are defined in the header `termios.h`.

4. It sets some fields of `options`, a local `termios` structure, using values defined in the header `termios.h`. These options specify, among other things, raw input mode, a one second timeout value for blocking reads, and input and output baud rates. `MyOpenSerialPort` then passes the `options` structure to the `tcsetattr` function to set new values for the serial port (the changes won't take effect until the call to `tcsetattr`). The constant `TCSANOW` is also defined in `termios.h`, and indicates that the change should be made immediately.

5. Finally, it returns the file descriptor obtained from the call to `open`.

You can find the headers mentioned in this section in header files in `Kernel.framework`, `System.framework`, or the directory `/usr/include`.

Listing 1-7   Opening the serial port specified by the passed device file

```
static int MyOpenSerialPort(const char *deviceFilePath)
{
    int         fileDescriptor = -1;
    int         handshake;
    struct termios   options;

    // Open the serial port read/write, with no controlling terminal,
    // and don't wait for a connection.
    // The O_NONBLOCK flag also causes subsequent I/O on the device to
    // be non-blocking.
    // See open(2) ("man 2 open") for details.

    fileDescriptor = open(deviceFilePath, O_RDWR | O_NOCTTY | O_NONBLOCK);
    if (fileDescriptor == -1)
    {
        printf("Error opening serial port %s - %s(%d).\n",
               deviceFilePath, strerror(errno), errno);
        goto error;
    }

    // Note that open() follows POSIX semantics: multiple open() calls to
    // the same file will succeed unless the TIOCEXCL ioctl is issued.
    // This will prevent additional opens except by root-owned processes.
    // See tty(4) ("man 4 tty") and ioctl(2) ("man 2 ioctl") for details.

    if (ioctl(fileDescriptor, TIOCEXCL) == kMyErrReturn)
    {
        printf("Error setting TIOCEXCL on %s - %s(%d).\n",
            deviceFilePath, strerror(errno), errno);
        goto error;
    }

    // Now that the device is open, clear the O_NONBLOCK flag so
    // subsequent I/O will block.
    // See fcntl(2) ("man 2 fcntl") for details.

    if (fcntl(fileDescriptor, F_SETFL, 0) == kMyErrReturn)
```

```
{
    printf("Error clearing O_NONBLOCK %s - %s(%d).\n",
        deviceFilePath, strerror(errno), errno);
    goto error;
}

// Get the current options and save them so we can restore the
// default settings later.
if (tcgetattr(fileDescriptor, &gOriginalTTYAttrs) == kMyErrReturn)
{
    printf("Error getting tty attributes %s - %s(%d).\n",
        deviceFilePath, strerror(errno), errno);
    goto error;
}

// The serial port attributes such as timeouts and baud rate are set by
// modifying the termios structure and then calling tcsetattr to
// cause the changes to take effect. Note that the
// changes will not take effect without the tcsetattr() call.
// See tcsetattr(4) ("man 4 tcsetattr") for details.

options = gOriginalTTYAttrs;

// Print the current input and output baud rates.
// See tcsetattr(4) ("man 4 tcsetattr") for details.

printf("Current input baud rate is %d\n", (int) cfgetispeed(&options));
printf("Current output baud rate is %d\n", (int) cfgetospeed(&options));

// Set raw input (non-canonical) mode, with reads blocking until either
// a single character has been received or a one second timeout expires.
// See tcsetattr(4) ("man 4 tcsetattr") and termios(4) ("man 4 termios")
// for details.

cfmakeraw(&options);
options.c_cc[VMIN] = 1;
options.c_cc[VTIME] = 10;

// The baud rate, word length, and handshake options can be set as follows:

cfsetspeed(&options, B19200);    // Set 19200 baud
options.c_cflag |= (CS7        |// Use 7 bit words
        PARENB     |        // Enable parity (even parity if PARODD
                            // not also set)
        CCTS_OFLOW |        // CTS flow control of output
        CRTS_IFLOW);        // RTS flow control of input

// Print the new input and output baud rates.

printf("Input baud rate changed to %d\n", (int) cfgetispeed(&options));
printf("Output baud rate changed to %d\n", (int) cfgetospeed(&options));

// Cause the new options to take effect immediately.
if (tcsetattr(fileDescriptor, TCSANOW, &options) == kMyErrReturn)
{
    printf("Error setting tty attributes %s - %s(%d).\n",
        deviceFilePath, strerror(errno), errno);
    goto error;
```

```
    }

    // To set the modem handshake lines, use the following ioctls.
    // See tty(4) ("man 4 tty") and ioctl(2) ("man 2 ioctl") for details.

    if (ioctl(fileDescriptor, TIOCSDTR) == kMyErrReturn)
    // Assert Data Terminal Ready (DTR)
    {
        printf("Error asserting DTR %s - %s(%d).\n",
            deviceFilePath, strerror(errno), errno);
    }

    if (ioctl(fileDescriptor, TIOCCDTR) == kMyErrReturn)
    // Clear Data Terminal Ready (DTR)
    {
        printf("Error clearing DTR %s - %s(%d).\n",
            deviceFilePath, strerror(errno), errno);
    }

    handshake = TIOCM_DTR | TIOCM_RTS | TIOCM_CTS | TIOCM_DSR;
    // Set the modem lines depending on the bits set in handshake.
    if (ioctl(fileDescriptor, TIOCMSET, &handshake) == kMyErrReturn)
    {
        printf("Error setting handshake lines %s - %s(%d).\n",
            deviceFilePath, strerror(errno), errno);
    }

    // To read the state of the modem lines, use the following ioctl.
    // See tty(4) ("man 4 tty") and ioctl(2) ("man 2 ioctl") for details.

    if (ioctl(fileDescriptor, TIOCMGET, &handshake) == kMyErrReturn)
    // Store the state of the modem lines in handshake.
    {
        printf("Error getting handshake lines %s - %s(%d).\n",
            deviceFilePath, strerror(errno), errno);
    }

    printf("Handshake lines currently set to %d\n", handshake);

    // Success:
    return fileDescriptor;

    // Failure:
error:
    if (fileDescriptor != kMyErrReturn)
    {
        close(fileDescriptor);
    }

    return -1;
}
```

Communicating With the Modem

Listing 1-8 (page 20) shows a simple modem initialization function, `MyInitializeModem`. The caller of this function passes the file descriptor for a modem's device file. The modem serial port is assumed to be open. `MyInitializeModem` function performs the following steps:

1. It sends an "AT" command to the modem, using the `write` function defined in `unistd.h`.

2. It attempts to read a response from the modem, using the `read` command, checking it against the desired response of "OK".

The definitions for the constants kOKResponseString, `kATCommandString`, and `kMyErrReturn` are shown in Listing 1-3 (page 11) and Listing 1-3 (page 11).

Throughout, the `MyInitializeModem` function uses the `MyLogString` function (shown in Listing 1-9 (page 21)) to replace with printable equivalents the unprintable characters in the modem-command strings and in the data received from the modem.

Listing 1-8   Initializing a serial port modem by writing to and reading from its device file

```
static Boolean MyInitializeModem(int fileDescriptor)
{
    char    buffer[256];    // Input buffer
    char    *bufPtr;        // Current char in buffer
    ssize_t numBytes;       // Number of bytes read or written
    int     tries;          // Number of tries so far
    Boolean result = false;

    for (tries = 1; tries <= kNumRetries; tries++)
    {
        printf("Try #%d\n", tries);

        // Send an AT command to the modem
        numBytes = write(fileDescriptor, kATCommandString,
                    strlen(kATCommandString));

    if (numBytes == kMyErrReturn)
        {
            printf("Error writing to modem - %s(%d).\n", strerror(errno),
                    errno);
            continue;
        }
    else {
        printf("Wrote %d bytes \"%s\"\n", numBytes,
                    MyLogString(kATCommandString));
    }

    if (numBytes < strlen(kATCommandString))
    {
            continue;
    }

        printf("Looking for \"%s\"\n", MyLogString(kOKResponseString));
```

```
    // Read characters into our buffer until we get a CR or LF.
    bufPtr = buffer;
    do
    {
        numBytes = read(fileDescriptor, bufPtr, &buffer[sizeof(buffer)]
                - bufPtr - 1);
        if (numBytes == kMyErrReturn)
        {
            printf("Error reading from modem - %s(%d).\n", strerror(errno),
                    errno);
        }
        else if (numBytes > 0)
        {
            bufPtr += numBytes;
            if (*(bufPtr - 1) == '\n' || *(bufPtr - 1) == '\r')
            {
                break;
            }
        }
        else {
            printf("Nothing read.\n");
        }
    } while (numBytes > 0);

    // NULL terminate the string and see if we got a response of OK.
    *bufPtr = '\0';

    printf("Read \"%s\"\n", MyLogString(buffer));

    if (strncmp(buffer, kOKResponseString, strlen(kOKResponseString)) == 0)
    {
        result = true;
        break;
    }
    }

    return result;
}
```

The `MyInitializeModem` function uses a utility function called `MyLogString` that replaces unprintable characters with printable equivalents, using the '\' character. Listing 1-9 shows the `MyLogString` function.

Listing 1-9   Enabling printing of data traffic

```
static char *MyLogString(char *str)
{
    static char     buf[2048];
    char            *ptr = buf;
    int             i;

    *ptr = '\0';

    while (*str)
    {
        if (isprint(*str))
        {
            *ptr++ = *str++;
        }
```

```
else {
    switch(*str)
    {
    case ' ':
        *ptr++ = *str;
        break;

    case 27:
        *ptr++ = '\\';
        *ptr++ = 'e';
        break;

    case '\t':
        *ptr++ = '\\';
        *ptr++ = 't';
        break;

    case '\n':
        *ptr++ = '\\';
        *ptr++ = 'n';
        break;

    case '\r':
        *ptr++ = '\\';
        *ptr++ = 'r';
        break;

    default:
        i = *str;
        (void)sprintf(ptr, "\\%03o", i);
        ptr += 4;
        break;
    }

    str++;
    }
    *ptr = '\0';
    }
    return buf;
}
```

Closing the Serial Port

Listing 1-10 (page 23) shows the MyCloseSerialPort function. This function performs the following steps:

1. It blocks until all output has been sent from the device.

2. It restores the previous state of the serial port, using values that were saved in a static structure of type termios by the MyOpenSerialPort function (Listing 1-7 (page 17)). The termios structure is defined in the header termios.h in System.framework.

3. To close the serial port, MyCloseSerialPort calls the close function (defined in unistd.h), passing the file descriptor for the serial port device file (obtained by the MyOpenSerialPort function).

Listing 1-10   Closing the serial port specified by the passed file descriptor

```c
void MyCloseSerialPort(int fileDescriptor)
{
    // Block until all written output has been sent from the device.
    // Note that this call is simply passed on to the serial device driver.
    // See tcsendbreak(3) ("man 3 tcsendbreak") for details.
    if (tcdrain(fileDescriptor) == kMyErrReturn)
    {
        printf("Error waiting for drain - %s(%d).\n",
            strerror(errno), errno);
    }

    // It is good practice to reset a serial port back to the state in
    // which you found it. This is why we saved the original termios struct
    // The constant TCSANOW (defined in termios.h) indicates that
    // the change should take effect immediately.
    if (tcsetattr(fileDescriptor, TCSANOW, &gOriginalTTYAttrs) ==
                    kMyErrReturn)
    {
        printf("Error resetting tty attributes - %s(%d).\n",
                    strerror(errno), errno);
    }

    close(fileDescriptor);
}
```

Document Revision History

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

Date	Notes
2005-12-06	Made minor corrections.
2005-10-04	Added information about endian issues. Changed title from "Working With Serial I/O."
2005-04-08	Added links to man page documentation. Emphasized how to modify sample code to look for devices other than modems.
2003-05-01	First version of *Working With Serial I/O*. This document comprises one chapter from an earlier version of *Accessing Hardware From Applications*.