

Migrate device control applications from Windows to Linux

Overcome migration challenges by understanding differences in Windows, Linux device control

Skill Level: Intermediate

[Sun Ling \(featherlet.sun@gmail.com\)](mailto:featherlet.sun@gmail.com)
Software Engineer Intern
IBM

[Yang Yi \(yangyi830320@hotmail.com\)](mailto:yangyi830320@hotmail.com)
Software Engineer Intern
IBM

24 Jun 2008

Ease the pain of migrating device control applications from Microsoft® Windows® to Linux® by understanding how device control works in both operating systems. The authors outline these differences and give you a C/C++ migration sample.

If you develop device control applications on different platforms, you know that Windows and Linux have different ways of doing device control, and migrating applications from one to the other can be a pain. In this article, we analyze how device control works in both operating systems, examining everything from architecture to system calls and focusing on the differences. We also give you a migration sample (in C/C++) to demonstrate the migration in detail.

Assumptions:

For the purposes of this article, "Windows" refers to Windows 2000 or later, and Microsoft Visual C++® version 6 or later should be installed. For Linux, the kernel should be based on 2.6, and GNU GCC should be installed.

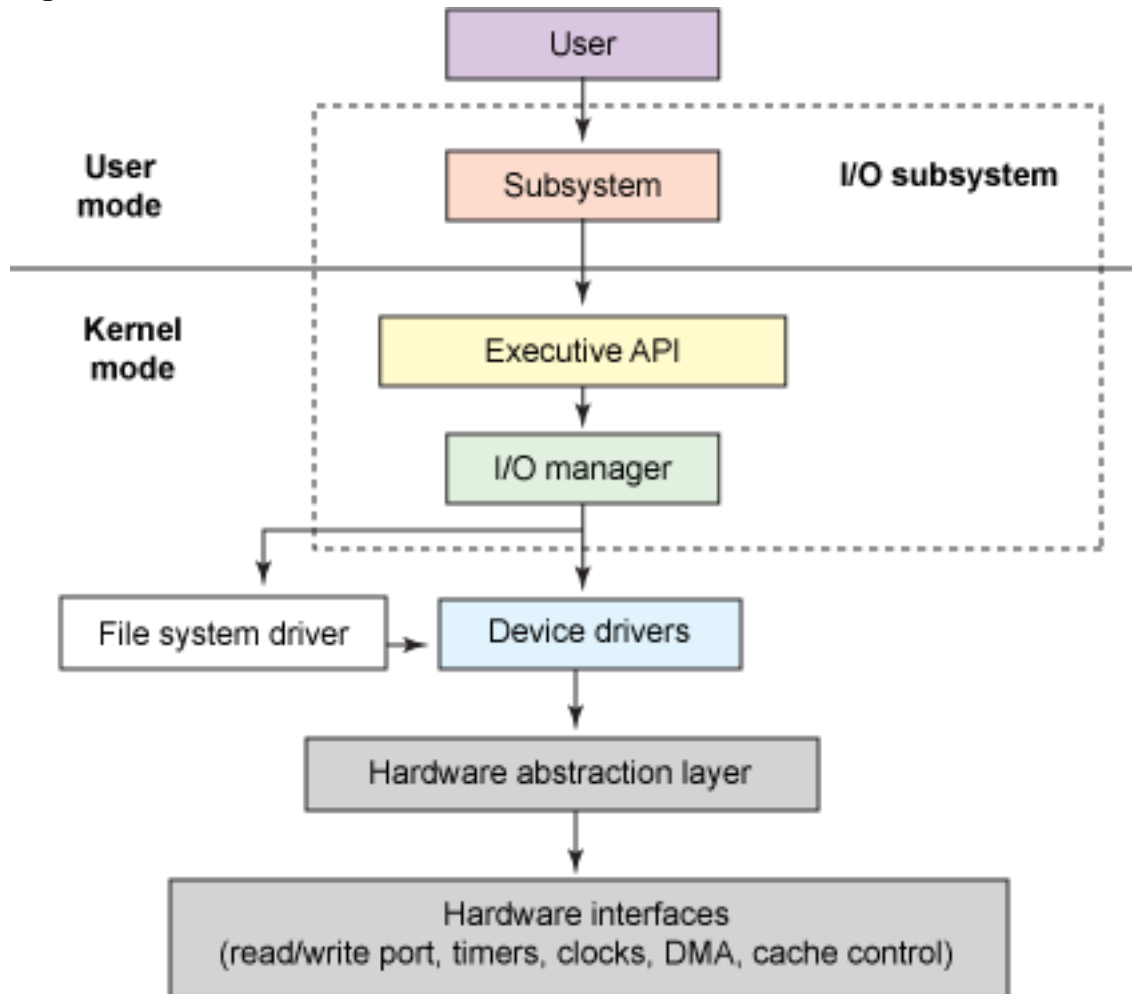
Comparing architectures of device control

Ways to control devices differ between Windows and Linux.

The Windows device control architecture

In Windows, the I/O subsystem connects user applications with device drivers and defines the infrastructure to support device drivers. Device drivers provide an I/O interface to particular devices (see Figure 1).

Figure 1. The Windows device control architecture

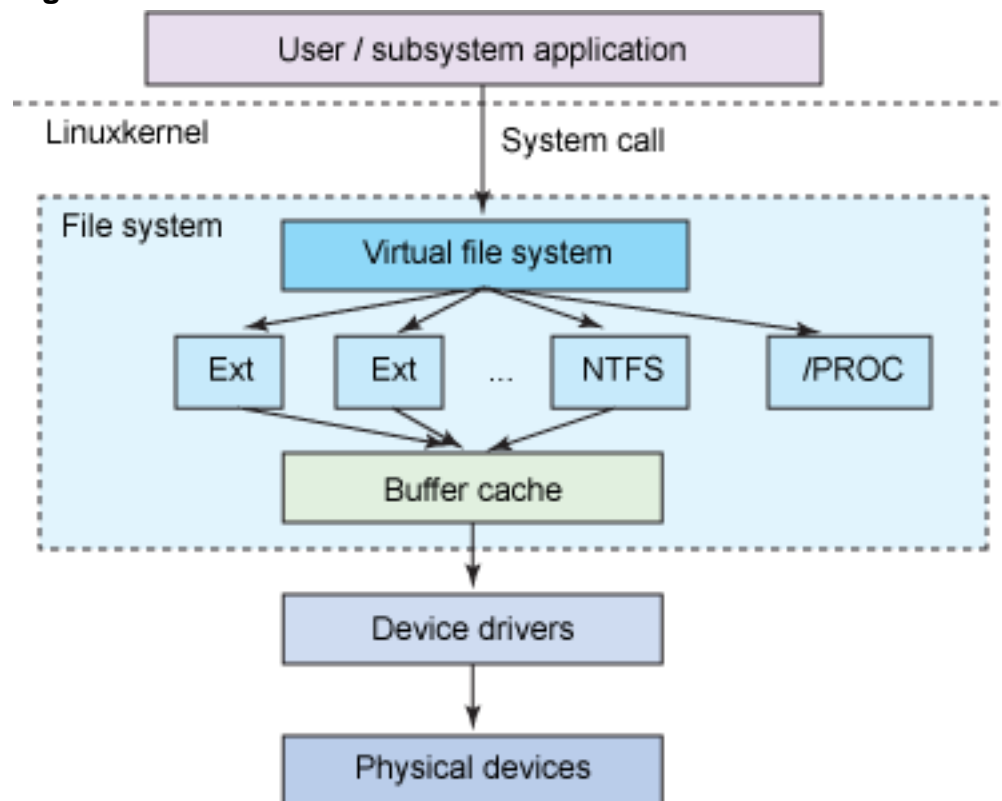


In the process of device control, I/O operations are encapsulated into an IRP (I/O Request Packet). The I/O manager creates the IRP and sends it to the top of the stack. Device drivers then get the stack location of an IRP, which contains parameters for the I/O request. According to the requirement in IRP (such as create, read, write, devioctl, cleanup, or close), each driver does its work through hardware interfaces.

The Linux device control architecture

The device control architecture is a little different in Linux, with the main difference being that normal files, directories, devices, and sockets are all files—everything is a file in Linux. To visit the device, the Linux kernel maps the device operation call to the device driver via the file system. There is no I/O manager in Linux: all I/O requests go to the file system at the beginning (see Figure 2).

Figure 2. The Linux device control architecture



Comparing device file names and path names

From a development point of view, getting the device handle is a prerequisite to device control; however, because the device control architecture varies, how you get the device handle is a different story depending on whether you're using Windows or Linux.

Generally speaking, the device handle is determined by the name of a particular device driver.

On Windows, the file name of a device driver is different from that of a common file; it's usually called the *device pathname* instead. It has a fixed format like `\\.DeviceName`. In C/C++ programming, this character string should be `\\.\\DeviceName`. And in code, we make it `\\\\.\\DeviceName`. `DeviceName` should be the same as the device name defined in the corresponding device driver

program.

Some device names are defined by Microsoft and will not be changed (as listed in Table 1).

Table 1. Device names on Windows (x = 0, 1, 2, etc.)

Device	Pathname
Floppy drive	A: B:
Hard disk logic sub-area	C: D: E: . . .
Physical drive	PhysicalDrivex
CD-ROM, DVD/ROM	CdRomx
Tape drive	Tapex
COM port	COMx

For example, we use device pathnames such as `\\\\\\. \\PhysicalDrive1`, `\\\\\\. \\CdRom0`, and `\\\\\\. \\Tape0` in C/C++ programming. For details on other devices not in this general list, check the [Resources](#) section later in this article.

Because the devices are described as files in Linux, you can find all of these device files in the directory `./dev`. The device drivers in this directory include:

- IDE (Integrated Drive Electronics) hard drives, such as `/dev/hda` and `/dev/hdb`
- CD-ROM drives, some of which are IDE; others are CD-RW (CD read/write) drives emulated as SCSI (Small Computer Systems Interface) devices such as `/dev/scd0`
- Serial ports, such as `/dev/ttyS0` for COM1, `/dev/ttyS1` for COM2, and so on
- Pointing devices, including `/dev/input/mice` and others
- Printers, such as `/dev/lp0`

Most common device files can be found according to the description above. For other device file names and detailed device information, try the command `dmesg`.

Comparing main system calls

Main system calls for device control include the following operations: open, close, I/O control, read/write, etc. See the Windows/Linux mapping shown in Table 2.

Table 2. Device control function mapping

Windows	Linux
CreateFile	open
CloseHandle	close
DeviceIoControl	ioctl
ReadFile	read
WriteFile	write

Now let's dig deeper into three of the most common functions: `create`, `close`, and `devioctl`.

Opening and closing a device in Windows

In Windows, we're talking about `CreateFile` and `CloseHandle`. You use the function `CreateFile` to open a device. The function returns a handle that can be used to access the object as shown in Listing 1.

Listing 1. The `CreateFile` function in Windows

```

HANDLE CreateFile (LPCTSTR lpFileName,           //File name of the device
                 DWORD dwDesiredAccess,        // (Device Pathname)
                 FILE_SHARE_FLAGS dwShareMode, //Access mode to the object (read,
                 SECURITY_ATTRIBUTES lpSecurityAttributes, // or both)
                 DWORD dwCreationDisposition,  //Sharing mode of the object
                 FILE_ATTRIBUTES dwFlagsAndAttributes, //Security attribute determining
                 HANDLE hTemplateFile);        // whether the returned handle can be
                                             // inherited by child processes
                                             // Action taken on files that exist
                                             // do not exist
                                             // File attributes and flags
                                             // A handle to a template file

```

The parameter `lpFileName` is the device path name that has been specified earlier. Generally, to open a device, we can set `dwDesiredAccess` to 0 or `GENERIC_READ|GENERIC_WRITE`, `dwShareMode` to `FILE_SHARE_READ|FILE_SHARE_WRITE`, `dwCreationDisposition` to `OPEN_EXISTING`, `dwFlagsAndAttributes` and `hTemplateFile` to 0 or `NULL`. The returned handle will be used in further device control operations.

To close a device, use the function `CloseHandle`. Set the parameter `hObject` as the handle returned when the device is open: `BOOL WINAPI CloseHandle (HANDLE hObject);`.

Opening and closing a device in Linux

In Linux, we're talking about `open` and `close`. As we mentioned earlier, opening a device is just like opening a common file. Listing 2 shows how we use `open` to get a device handle.

Listing 2. The `open` function in Linux

```
int open (const char *pathname,
         int flags,
         mode_t mode);
```

The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process. If failed, -1 is returned. The file descriptor is used as the device handle.

The parameter flags must include one of these: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. Other flags are optional. The argument mode specifies the permissions to use in case a new file is created.

The function `close` closes a device on Linux, just like closing a file: `int close(int fd);`

DeviceIoControl in Windows

Device control (`DeviceIoControl` in Windows and `ioctl` in Linux) is the most common function used for device control, fulfilling such tasks as accessing devices, getting information, sending orders, and exchanging data. Listing 3 illustrates `DeviceIoControl`:

Listing 3. The `DeviceIoControl` function in Windows

```
BOOL DeviceIoControl (HANDLE hDevice,
                    DWORD dwIoControlCode,
                    LPVOID lpInBuffer,
                    DWORD nInBufferSize,
                    LPVOID lpOutBuffer,
                    DWORD nOutBufferSize,
                    LPDWORD lpBytesReturned,
                    LPOVERLAPPED lpOverlapped);
```

This system call sends control code and other data to a specified device. The corresponding device drivers work according to what the control code `dwIoControlCode` tells them to do. For example, we can use `IOCTL_DISK_GET_DRIVE_GEOMETRY` to get structure parameters from physical drives (type of medium, number of cylinders, track number on each cylinder, number of sectors on each track, etc.). You can find all the control code definitions, header files, and other detailed information on the MSDN Web site (see [Resources](#) for links).

Whether the input/output buffers are required and what their structure and size is depends on the device and operation the actual `ioctl` process relates to. They are also determined by `dwIoControlCode` specified in the call.

If the pointer for an overlapped operation is set to `NULL`, `DeviceIoControl` will work in a blocked (synchronous) way. Otherwise, it will work asynchronously.

ioctl in Linux

In Linux, you use `ioctl`—`int ioctl(int fildes, int request, /* arg */ ...)`;—to send the control information to a specified device. The first parameter `fildes` is the open file descriptor returned from the function `open()`, referring to a specific device.

Unlike the corresponding system call `DeviceIoControl`, the input parameter list in `ioctl` is not fixed. It depends on what kind of request the `ioctl` performs and what is specified by the parameter `request`, just like `dwIoControlCode` in Windows `DeviceIoControl`. However, during migration, you need to pay attention when choosing the correct request parameter because `dwIoControlCode` in `DeviceIoControl` and `request` in `ioctl` are not of the same value, and there is no explicit mapping list for `dwIoControlCode/request`. Normally, you choose the value of a parameter request by looking into its definition in its header file. All the definitions of the control codes are in `/usr/include/{asm,linux}/*.h`.

The parameter `arg` is left to deliver detailed command information needed by the specific device to do its required work. The data type of `arg` depends on the particular control request. We can use this argument to both deliver detailed commands and receive returned data.

Migration sample

Let's look at an example of the migration process from Windows to Linux. This example involves reading the SMART log from the main IDE hard drive on a personal computer.

Step 1. Identify the type of device

As we discussed, each device on Linux is regarded as a file. The first step is to figure out the file name of the device on Linux. Only by using this file name can we get the device handle that is required for device control.

In this example, the object is an IDE hard drive. In Linux, it is described as `/dev/hda`, `/dev/hdb`, etc. The device path name of the hard disk we are going to migrate in this example is `\\\\.\\PhysicalDrive0`. `/dev/hda` is the corresponding file name of the device on Linux.

Step 2. Change the include headers

We must change the `#include` header files to their Linux forms (see Table 3):

Table 3. #include header files

Windows	Linux
<code>#include <windows.h></code>	<code>#include <sys/types.h></code> <code>#include <sys/stat.h></code> <code>#include <fcntl.h></code>
<code>#include <devioctl.h></code>	<code>#include <sys/ioctl.h></code>
<code>#include <ntddscsi.h></code>	<code>#include <linux/hdreg.h></code>

`windows.h` is used for functions opening and closing the device (`CreateFile` and `CloseHandle`). Accordingly, header files needed for `open()` and `close()` on Linux should be included. They are `sys/types.h`, `sys/stat.h`, and `fcntl.h`.

`devioctl.h` in Windows is for function `DeviceIoControl`, and we change it to `sys/ioctl.h` to make sure the function `ioctl` can work.

In `ntddscsi.h` (this is the header file from DDK), a set of control codes are defined for device controlling. Because this sample deals only with the IDE hard drive, we just need to add `linux/hdreg.h` into the Linux program.

In other situations, make sure all the header files with definitions of required control codes are included. For example, to access a CD-ROM rather than a hard drive, include `linux/cdrom.h` instead.

Step 3. Revise functions and parameters

Now let's see the code in detail. Listing 4 shows the detailed command information.

Listing 4. Command details

```
unsigned char cmdBuff[7];
cmdBuff[0] = SMART_READ_LOG; // Used for specifying SMART "commands"
cmdBuff[1] = 1; // IDE sector count register
cmdBuff[2] = 1; // IDE sector number register
cmdBuff[3] = SMART_CYL_LOW; // IDE low order cylinder value
cmdBuff[4] = SMART_CYL_HI; // IDE high order cylinder value
cmdBuff[5] = 0xA0 | (((Dev->Id-1) & 1) * 16); // IDE drive/head register
cmdBuff[6] = SMART_CMD; // Actual IDE command
```

The command information is from the ATA command specification. Because no changes are needed to transplant the code to Linux, no further analysis is needed.

The code shown in Listing 5 opens the main hard drive on Windows.

Listing 5. Opening the main hard drive on Windows

```
HANDLE devHandle = CreateFile("\\\\.\\PhysicalDrive0",           //pathname
                              GENERIC_WRITE|GENERIC_READ,    //Access Mode
                              FILE_SHARE_READ|FILE_SHARE_WRITE, //Sharing Mode
                              NULL,OPEN_EXISTING,0,NULL);
```

Remember from the section on opening and closing that we need two parameters (file path name and access mode to the device) to open a device on Linux. According to the previous original code, the first should be `/dev/hda`, and the second `O_RDONLY|O_NONBLOCK`. The changed code looks like this: `HANDLE devHandle = open("/dev/hda", O_RDONLY | O_NONBLOCK);`. Accordingly, change `CloseHandle(devHandle);` to `close(devHandle);`.

The main part is about how to use `ioctl` to get access to the particular device and the information we want. The original Windows code is shown in Listing 6:

Listing 6. Source of DeviceIoControl on Windows

```
typedef struct _Buffer{
    UCHAR    req[8];                // Detailed command information other than
                                   // control code
    ULONG    DataBufferSize;       // Size of Data Buffer, here is 512
    UCHAR    DataBuffer[512];     // Data Buffer
} Buffer;

Buffer regBuffer;
memcpy(regBuffer.req, cmdBuff, 7); //req[7] is reserved for future use. Must be
zero.
regBuffer.DataBufferSize = 512;
unsigned int size = 512+12;       // Size of regBuffer
                                   // 8 for req, 4 for DataBufferSize, 512 for
data
DWORD bytesRet = 0;              // Number of bytes returned
int retVal;                       // Returned value

retVal = DeviceIoControl(devHandle,
                        IOCTL_IDE_PASS_THROUGH, //Control code
                        regBuffer, // Input Buffer, including detailed command
size,
                        regBuffer, // Output Buffer, use the same buffer here
size,
                        &bytesRet, NULL);

if (!retVal)
    cout<<"DeviceIoControl failed."<<endl;
else
    memcpy(data, regBuffer.DataBuffer, 512);
```

`DeviceIoControl` requires more parameters than `ioctl` does. The device handle is the first parameter on both platforms, returned from `CreateFile/open()` for Linux. But control code on Windows and requests on Linux are defined in such a different way that there is no fixed rule to find the mapping relations between these two parameters, as we discussed earlier. `IOCTL_IDE_PASS_THROUGH` is defined in header file `ntddscsi.h` as `CTL_CODE (IOCTL SCSI_BASE, 0x040a, METHOD_BUFFERED, FILE_READ_ACCESS | FILE_WRITE_ACCESS)`. By

looking at the definitions in header file `/usr/include/linux/hdreg.h`, we choose the corresponding control code `HDIO_DRIVE_CMD` for Linux.

In addition, detailed command information is needed for the device to fulfill a specific task. The command is included in a buffer being exchanged in the process along with the space left for returned data. We use the same buffer to both send the command and get log information we need. In Linux, the space for data buffer size can be removed; not all eight bytes are required. In this sample, only four bytes of the command are used.

The corresponding code in Linux (Listing 7) seems much simpler because the structure and function arguments are simpler than in Windows.

Listing 7. Source of `ioctl` on Linux

```
int retval;
unsigned char req[4+512]; // Enough for returned data and the 4 byte detailed
                          // command information
req[0]= cmdBuff[6];      // Consider the requirement in this sample, only 4 bytes
                          // are used
req[1]= cmdBuff[2];
req[2]= cmdBuff[0];
req[3]= cmdBuff[1];

retval = ioctl(devHandle, HDIO_DRIVE_CMD, &req);
if(retval)
    cout<<"ioctl failed."<<endl;
else
    memcpy(data, &req[4], 512);
```

Step 4. Testing in the Linux environment

After revising header files, functions, and parameters, the program is ready for a Linux run. Now the task is to compile it on a Linux platform and correct any remaining syntax errors. Some additional changes may be needed according to the edition of Linux and compiling environment.

Resources

Learn

- In the [Windows-to-Linux roadmap](#) series, e-business architect Chris Walden guides you through a nine-part developerWorks series on moving your operational skills from Windows to Linux, covering everything from logging to networking, from the command-line to help systems, even compiling packages from available source code. Of special interest may be "[Windows-to-Linux roadmap: Part 6. Working with partitions and file systems](#)" (developerWorks, November 2003), which gives advice on working with devices on Linux.
- The [Migration station](#) resources will help you port your applications from Windows, Solaris, and OS/2 to run natively on Linux on x86-based, POWER-based, and System z systems.
- See [Device File Names](#) on MSDN for details on Windows device names.
- See [CreateFile Function](#) on MSDN for details on how Windows functions work with devices.
- The "[UNIX Application Migration Guide](#)" includes guidelines and best practices for migrating UNIX applications.
- In the [developerWorks Linux zone](#), find more resources for Linux developers, and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and spaces.

About the authors

Sun Ling

Sun Ling is an intern software engineer at the IBM China System and Technology Lab, pursuing a master's degree in Information Security Engineering at Shanghai

Jiaotong University. At present, she is working in the CIM Convergence team, and has experience in device control development and migration on Linux.

Yang Yi

Yang Yi is an intern software engineer at the IBM China System and Technology Lab. Currently, he is pursuing a master's degree in Electronic Engineering at Shanghai Jiaotong University. He has experience with device management on different platforms.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.