Java and USB

Although the Universal Serial Bus (USB) is an integral part of many computers, Java does not officially support USB. Getting your Java programs to interact with arbitrary USB devices thus requires either a third-party Java/USB API or your own Java/USB API. This article introduces two third-party APIs and my own API, which provides a partial USB interaction.

Before it introduces the APIs, this article explores some USB fundamentals. The article next introduces JSR-80 (an API that attempts to bring official USB support to Java) and jUSB (a rival API). Moving on, the article presents my own bus-enumeration API, which only works in a Windows context. The article closes by pondering whether or not Java should officially support the USB.

I developed and tested this article’s code with Sun’s J2SE 5.0 SDK and Borland C++ 5.5.1. Windows 98 SE and Windows ME were the underlying OSes.

USB in a Nutshell

USB is a serial bus standard for connecting devices to computers, video game consoles, televisions, and so on. This standard has advantages over the legacy personal computer (PC) architecture (expansion cards, I/O ports, interrupts, device drivers) previously used to connect devices to PCs. These advantages include:

- The serial bus, in partnership with plug and play, eliminates I/O port and interrupt conflicts that often arise when adding new devices to a legacy PC.
- There is no need to open a USB-enabled PC and insert expansion cards into its system bus. Devices plug into the serial bus through external connectors.
- Devices can be connected to or removed from the serial bus without having to first power down or reboot the USB-enabled PC. This is known as hot swapping.

Compaq, Intel, Microsoft, and NEC started to work on the USB standard in 1994. In January 1996, they released version 1.0. Following revisions, these companies released version 1.1 in September 1998. And in April 2000, after Hewlett-Packard, Lucent, and Philips joined the group, version 2.0 was released.

USB 1.1 was the first widely implemented version of the USB standard. It featured two data-transfer speeds: full speed (12 megabits per second) and low speed (1.5 megabits per second). It is now fading into history as version 2.0 takes over. This version adds a third data-transfer speed: high speed (480 megabits per second).

To fully understand the USB standard, you’ll need to review the USB 1.1 specification and the USB 2.0 specification. For the purpose of understanding this article, however, you only need to grasp a few fundamentals, starting with an overview of host controllers, hubs, and functions.

Host Controllers, Hubs, and Functions

USB is implemented in terms of software drivers (which I don’t discuss here) and hardware. The hardware divides into host controllers, hubs, and functions:
The host controller connects a tree of USB devices to a host (computer, console, and so on). Multiple host controllers can be present. Each host controller anchors one device tree.

A hub is a device with multiple ports, into which are plugged other hubs and functions. The root hub integrates with the host controller. All other hubs are external hubs.

A function is a device that provides a capability to the host. It plugs into a hub (external or root). Mice, keyboards, cameras, scanners, and printers are examples.

This hardware arranges into a tiered-star topology where each star’s center is a hub: both the root hub and external hubs sit at the center of their connected external hubs/functions. Figure 1 reveals this topology.

![Figure 1. Three external hubs (Hub 1/Hub 2/Hub 3 or Hub 1/Hub 2/Hub 4) are chained together from the root hub](image)

**Configurations, Interfaces, and Endpoints**

A function can have one or more configurations that control how the function behaves. Configurations can differ in how much power they consume, whether and how they remotely wake up a suspended computer, and more.

A trackball device that can be configured as a mouse or as a joystick is an example of a function with multiple configurations. Another example is an Integrated Services Digital Network (ISDN) communications device that offers one 128Kb channel or two 56Kb channels.

Each configuration contains one or more interfaces that specify how software accesses the hardware. Interfaces frequently have alternate settings corresponding to different bandwidth requirements.

An interface exposes endpoints that each serve as a data transfer source or a data transfer destination. Each endpoint supports data transfer in a single direction and has a unique numeric identifier. Figure 2 relates endpoints to interfaces and configurations.
Figure 2. The relationship between a function’s configurations, interfaces, and endpoints

Note: Microsoft drivers tend to work with only the first configuration. As a result, multi-configuration functions are rare and are discouraged by Microsoft.

Data Flow Model

USB supports the flow of data between applications running on a host and physical devices attached to the USB. This data flow can be understood in terms of a data flow model--such as Figure 3’s USB data flow model.
Actual data transfers take place in a physical device’s lowest bus layer. This layer consists of the hubs and cables (and associated connectors) that connect the host controller to the physical device’s bus interface.

Data physically flows vertically up and down both sides of the model. Data virtually flows horizontally across the corresponding sides of the upper two layers by way of pipes, logical channels that associate host software memory buffers with endpoints.

Pipes can be categorized as message pipes and stream pipes. Message pipes transfer data with some USB structure. In contrast, stream pipes transfer data with no USB structure.

The device layer has a logical view of and controls the physical device. It uses a special message pipe, the Default Control Pipe, to read configuration data from input endpoint 0 and output control data to output endpoint 0. These endpoints are always present.

The function layer has a function view of the physical device. This layer uses data pipes (stream pipes) to transfer data to and from the function.

Transfer Types

A pipe transfers data according to an endpoint’s transfer type, which governs how much data transfers in a transaction (part of a data transfer), whether or not the transfer is lossless, and other factors:

- **Bulk transfers** are used to transfer files and perform other kinds of transfers that involve large blocks of unstructured data. These transfers guarantee that all data will be delivered.
- **Control transfers** are used to control the physical device. These transfers send and receive structured data in the form of commands and statuses.
- **Isochronous transfers** are used to transfer realtime audio, video, and other kinds of unstructured data at a guaranteed speed. However, there is the possibility of data loss.
- **Interrupt transfers** are used by pointing devices, keyboards, and any other physical devices that need a guaranteed quick response. Interrupt transfers are similar to bulk transfers.

**Note:** Although I don’t focus on data transfer in this article, you need to be aware of these transfer types when working with the JSR 80 and jUSB APIs.

Descriptors

USB devices present onboard data structures known as descriptors. These data structures let a device’s function (or functions) identify itself (or themselves) to software running on the host:

- **Device descriptors** describe an entire device.
- **Device qualifier** descriptors are used with USB 2.0 devices.
- **Configuration** descriptors describe a device’s configurations.
- **Other speed configuration** descriptors are used with USB 2.0 devices.
- **Interface** descriptors describe a configuration’s interfaces.
- **Endpoint descriptors** describe an interface’s endpoints.
String descriptors contain human-readable Unicode strings that each describe a device, a configuration, an interface, or an endpoint.

Later in this article, I present a bus-enumeration example, whose C++ code accesses device and string descriptors during the enumeration process.

Device Classes

USB devices belong to device classes, which define an expected behavior in terms of device and interface descriptors. A device class's device driver can be used for any device belonging to the device class.

OSes usually provide generic drivers for all device classes, to support all USB devices. When a device attaches to the USB, its device class identifier is read from a specific descriptor and used to load the proper driver.

The USB-IF standardizes device class identifiers. Examples include 1 (audio class: a sound card), 3 (human interface device class, such as keyboards and mice), 8 (mass storage device class: flash drives and portable hard drives), and 9 (hub class). Identifiers 0 and 255 are reserved.

If the class belongs to the entire device, the identifier is assigned to the device descriptor's 8-bit bDeviceClass field. If the class is for a single interface, the identifier is assigned to the interface descriptor's 8-bit bInterfaceClass field (0 is assigned to bDeviceClass).

JSR-80

In 1999, IBM's Dan Streetman began a Java/USB API project to let Java programs interact with the USB. In 2001, via the Java Community Process (JCP), this project was accepted as a candidate standard extension to the Java language. The JCP identifies this project as Java Specification Request (JSR) 80--you can determine JSR 80's current status by visiting the JCP home page for JSR 80 (Java USB API).

The official development site for JSR 80 is JSR080 - javax.usb. This site provides links to the JSR 80's Common Public License, a FAQ, API implementations in the CVS repository, and Javadoc for the API. The repository provides jsr80.jar (the core javax.usb package implementation), jsr80_ri.jar (the OS-independent reference implementation), and Linux/Windows implementations.

The Linux implementation has been certified with Sun's test compliance kit. You'll find this implementation's C and Java source files, and a Linux-specific javax.usb.properties file, in the repository's javax-usb-ri-linux directory tree. It appears that you will have to build this implementation from the source code (assuming that Linux is your OS).

The pre-alpha Windows implementation is not certified and requires a kernel driver. This implementation's C and Java source files, and a Windows-specific javax.usb.properties file, are located in the repository's javax-usb-ri-windows directory tree. If Windows is your OS, you can build this implementation from the source code (after possibly making changes), but you must supply the kernel driver.

You might prefer the alpha Windows implementation. This uncertified implementation is located in the repository's javax-usb-libusb directory tree. Along with deployment files,
you'll need to install the Windows version of libusb. Visit the libusb Project Page and the libusb-win32 Project Page to learn about libusb and obtain the Windows version.

The JSR 80 API specification PDF document, located in the repository as jsr80.pdf, describes this API's architecture. This document reveals that the USBHostManager class is the entry point into the API. This class lets you obtain an OS-specific instance of the USBServices interface (based on information in javax.usb.properties), from which you can obtain the virtual root hub:

```java
UsbServices services = USBHostManager.getUsbServices ();
UsbHub vroothub = services.getRootUsbHub ();
```

The virtual root hub, obtained by calling UsbServices's public UsbHub getRootUsbHub() method, provides access to all of the available host controllers (and their root hubs). UsbHub's public boolean isRootUsbHub(), public byte getNumberOfPorts(), and public List getUsbPorts() methods are useful in verifying, enumerating, and obtaining root hubs.

I won't dig deeper into JSR 80 because "Access USB Devices from Java Applications" presents a nice overview of this API. This overview includes example bus-enumeration and data-transfer code, which demonstrates obtaining an endpoint's transfer type, direction, and connected pipe. Instead, I've identified four tasks that you need to perform to achieve a successful installation of JSR 80 on your OS:

- Add jsr80.jar to your CLASSPATH. Your Java programs interact with this .jar file's classes and interfaces. You can find jsr80.jar in the javax-usb-libusb directory's lib directory.
- Add jsr80_ri.jar to your CLASSPATH. Almost all of the javax.usb API is implemented in this .jar file. You can also find jsr80_ri.jar in the javax-usb-libusb directory's lib directory.
- Install either the Linux or Windows libusb implementation. For Windows libusb, don't forget to install the Windows version of libusb. Furthermore, you must also add jsr80_windows.jar to your CLASSPATH when installing the Windows libusb implementation. This .jar file is in the previous lib directory.
- Add the appropriate javax.usb.properties file (either the Linux or Windows libusb version) to your CLASSPATH. This file contains a javax.usb.services property, which identifies the fully qualified name of a class that implements the javax.usb.UsbServices interface. This class is the entry point to the Linux and Windows implementations.

The lib directory also contains log4j.properties, along with the commons_logging.jar and log4j.jar jar files. I believe you can avoid adding these .jar files to your CLASSPATH by modifying the javax.usb.properties file.

jUSB

"Mojo Jojo" and David Brownel began a rival Java/USB API project in June 2000: jUSB. They developed a version for Linux, which is available from their project page. For his thesis, Mike Stahl created a Windows version; visit the jUSB: Java USB API for Windows project page to obtain this version. jUSB is licensed under the Lesser GNU Public License.
While researching jUSB, I focused on Mike's Windows version. From Mike's jUSB site, I downloaded two archives: JavaUSB.zip and JavaUSBComplete.zip. The former archive contains the Java source code to jUSB's packages, the jUSB DLL, and the jUSB kernel driver (along with information and registry files) for Windows XP and Windows 2000. The latter archive includes the DLL and driver source code.

I also downloaded Mike's "Java USB API for Windows" diploma thesis--a 2.2MB PDF file. This PDF file introduces the USB and provides a detailed account of how Mike implemented jUSB's usb.windows package for Windows 2000 and Windows XP. This document is a must-read for anyone wanting to create a Java/USB API that accesses the Windows USB architecture.

According to Mike's thesis, jUSB's usb.core package contains a HostFactory class, whose public static Host getHost() method is the entry point into jUSB. This method returns a Host object, which monitors all busses (device trees) on a given machine. Rather than dig deeper into jUSB, I recommend exploring jUSB in "Access USB Devices from Java Applications."

Create Your own Java/USB API

Although you can use the JSR 80 and jUSB APIs to give your Java programs the ability to interact with USB, you might decide to create something different. To help you with this task, I present my own API. This API is limited to bus enumeration. Furthermore, the Java Native Interface (JNI) C++ code limits the API to Windows 98 SE, Windows ME, and (although it has not been tested) Windows 2000.

I built the API's C++ code using the Borland C++ 5.5.1 compiler. You'll need to obtain a copy of this compiler (which is free) if you plan to modify the code--I provide a prebuilt DLL with this article's code (see Resources). Click the Compiler link in the Downloads table on the C++ Builder Downloads--Borland C++ 5.5.1 compiler page to obtain Borland C++ 5.5.1.

My "Java Tech: Acquire Images with TWAIN and SANE, Part 1" article, which I wrote for java.net, briefly introduces Borland C++ 5.5.1. For a more substantial introduction, check out "Get Acquainted with Borland C++ 5.5.1" in my article "Build Screensavers with a Custom Screensavers Library in Borland C++ ."

JavaUSB

My JavaUSB API consists of five top-level unpackaged classes: JavaUSB, HCInfo, Device, Hub, and Function. Consider placing these classes into their own package. The JavaUSB class serves as the entry point into this API. It provides three methods that load a DLL and access DLL functions:

- **public static boolean init()** initializes JavaUSB by loading javausb.dll. It returns Boolean true if the DLL successfully loads. If an UnsatisfiedLinkError occurs, false returns. You must call this method before calling the other two methods.
- **public static native HCInfo getHostControllerInfo(int index)** returns an HCInfo object that provides the name of the indexth host controller and the name of its root hub. Indexes start at 0; null is returned if no host controller associates with index.
- **public static native Device [] getAttachedDevices(String hubName)** returns an array (possibly with zero length) of all Devices currently
attached to the hub identified by hubName; null is returned if this method cannot accomplish its task.

To keep JavaUSB simple, I haven't provided an exception class and modified each native method to throw an instance of this class whenever something goes wrong. However, these methods can throw JNI-related errors ("classfile not found," for example). Instead, a problem is revealed when the native method returns a null value. Consider enhancing these methods to throw a suitable exception.

HCInfo describes host controller information returned by the getHostControllerInfo() method. This information is stored in two String fields: name contains the host controller's name and rootHubName contains the host controller's integrated root hub's name. Pass the rootHubName field to getAttachedDevices() to begin enumerating a device tree.

Device describes one of the hub or function devices that is attached to a hub. This class contains three String fields that are common to hubs and functions: manufacturer identifies the device's manufacturer, product identifies the device as a product, and serialNumber identifies the device's serial number. These fields may be null.

Device is also the abstract superclass for the Hub and Function classes. The Hub class presents a single String field, hubName, that names an external hub. In contrast, Function is an empty class. In the future, I plan to expand Hub and Function with fields and methods that are unique to these classes.

To demonstrate how easy JavaUSB's classes and the JavaUSB class's three methods are to use, I have created a TestJavaUSB application. This application's source code employs four of the classes (there is no point in using Function until this class is more fully developed) to enumerate each host controller's root hub's device tree. The application's source code is presented below.

```
// TestJavaUSB.java

public class TestJavaUSB
{
    public static void main (String [] args)
    {
        // Attempt to initialize the JavaUSB
        // library.
        
        if (!JavaUSB.init ())
        {
            System.err.println ("Unable to " +
                               "initialize JavaUSB");
            return;
        }
        
        // Enumerate all attached host controllers,
        // outputting their names and the names of
        // their root hubs. For each root hub,
        // enumerate the device tree.
        
        for (int i = 0; ; i++)
        {
            HCInfo hcinfo =
                JavaUSB.getHostControllerInfo (i);
            if (hcinfo == null)
```
break;

System.out.println ("\nHost " +
"controller name = " + hcinfo.name);
System.out.println ("Root hub " +
"name = " + hcinfo.rootHubName);

enumerate (hcinfo.rootHubName, 1);
}
}

public static void enumerate (String hubname,
   int depth)
{
   Device [] devices =
   JavaUSB.getAttachedDevices (hubname);
   if (devices == null)
      return;

   for (int i = 0; i < devices.length; i++)
   {
      System.out.println ("
      for (int j = 0; j < depth; j++)
         System.out.print (" ");
      System.out.println (devices [i]);
      for (int j = 0; j < depth; j++)
         System.out.print (" ");
      System.out.printf ("Manufacturer = %s\n", devices [i].manufacturer);
      for (int j = 0; j < depth; j++)
         System.out.print (" ");
      System.out.printf ("Product = %s\n", devices [i].product);
      for (int j = 0; j < depth; j++)
         System.out.print (" ");
      System.out.printf ("Serial number " +
"= %s\n", devices [i].serialNumber);
      if (devices [i].isHub ())
         enumerate (((Hub) devices [i]).hubName, depth+1);
   }
}

Executing java TestJavaUSB on my Windows 98 SE OS caused the following output to appear:

Host controller name = USB\0000
Root hub name = \\.\0000000000000000#{f18a0e88-c30c-11d0-8815-00a0c906bed8}

Function
The output below appeared when I executed `java TestJavaUSB` under Windows ME OS:

Host controller name = USB\0002
Root hub name = \\.\0000000000000005#{f18a0e88-c30c-11d0-8815-00a0c906bed8}

Hub
Manufacturer = ALCOR
Product = Generic USB Hub
Serial number = ?

Function
Manufacturer = LEXAR MEDIA
Product = JUMPDRIVE
Serial number = L210442203100M

This source code raises an interesting question: What happens if a host controller is attached or removed during enumeration? For an attachment, I assume the OS will append a host controller entry to some structure of entries and enumeration will not be affected. For a removal, either the host controller was already enumerated or a native method will return null (depending on where the code is executing).

Under the Hood

JavaUSB works with Windows 98 SE and Windows ME. Although I haven't tested this API with Windows 2000 (I do not have access to this OS), I believe that JavaUSB works with Windows 2000. If you need to modify this API's `javausb.cpp` code to support another Windows OS (or even a non-Windows OS), you'll be interested in the following `javausb.cpp` details:

- Various Windows Driver Development Kits (DDKs) define assorted constants, macros, and types for working with the USB. Rather than require you to install a copy of the DDK (the Windows 98 SE DDK is no longer available), I've created an extensively commented `usb.h` header file, with constants, macros, and types relevant to and included by `javausb.cpp`.
- The `javausb.cpp` source code uses four Win32 API functions to interact with the Windows USB: `CreateFile()` (open a host controller or hub device), `DeviceIoControl()` (interact with the device), `WideCharToMultiByte()` (convert Unicode characters to ANSI 8-bit characters), and `CloseHandle()` (close the device).
- Windows 98 SE, Windows ME, and Windows 2000 use the `\\.\HCDx` syntax (`x` is a zero-based integer) to identify a host controller device. The first host controller's device name is `\\.\HCD0`, the second host controller's device name is `\\.\HCD1`, and so on. This device name is passed to `CreateFile()` to open the host controller.

For information on building `javausb.dll` with Borland C++ 5.5.1, examine the articles I referred to earlier. Also, read the comments in the `makedll.bat` file--included with this article's code.

Should Java Officially Support the USB?
While researching this article, I encountered various discussions on whether or not Java should officially support the USB. Several individuals want Sun to introduce USB support into Java, whereas other individuals do not think it's a good idea for Java to officially support the USB. Below are two typical reactions:

- One respondent to JavaLobby's What would you like to do with Java and cannot? topic wants Java to officially support the USB: "Seriously, Sun if you are reading this, pay attention! This [javax.comm] sucks badly. And while you're fixing serial port IO, how about getting USB and FireWire working while you're at it?"
- According to one java.net forum topic: USB support (especially for Windows) response, Java shouldn't officially support the USB because this is "similar to having direct access to the physical layer on network cards or directly accessing the SCSI bus." Furthermore: "You don't need direct USB access to use card readers or USB mass storage devices--that should be handled by the OS. You should be able to simply create a file in the appropriate section of the file system, write to it, and the OS should take care of the rest."

If Java is to officially support the USB, this support should be present on all OSes that support Java--even on OSes that don't support the USB. After all, Java is a cross-platform technology. But what happens if a Java-supported OS supports FireWire (or some other technology) instead of the USB?

Some developers have expressed a desire to see Java support FireWire and Bluetooth (which I like to think of as a wireless USB) in addition to USB. Although JSR 82, Java APIs for Bluetooth, is in the works (there appears to be no FireWire JSR), does it make sense to have separate USB and Bluetooth APIs?

USB, FireWire, and Bluetooth are three technologies that communicate with devices. Because Java has a long history of abstracting over similar technologies, maybe it's feasible to devise a single serial-based API that abstracts over these and similar future technologies. Perhaps Java should officially support USB this way.

**Conclusion**

Because Java doesn't officially support USB, you need to work with JSR 80, jUSB, or some other third-party Java/USB API--or even create your own API--to bring USB interaction to your Java programs. Should you plan to create your own API for use with Windows, JavaUSB can help get you started with the bus-enumeration part. I deliberately avoided supporting data transfer for two reasons:

- I don't want to support data transfer until I complete the bus-enumeration code. According to a Microsoft article, I need to use a technique based on globally unique identifiers (GUIDs) to ensure that USB 2.0 host controllers are enumerated on Windows 2000 (and possibly on Window 98 SE and Windows ME).
- I need to create a kernel-mode driver to support data transfer on Windows. Although I recently obtained a copy of Walter Oney's highly-regarded book *Programming the Microsoft Windows Driver Model* (Microsoft Press, 2003), which explores kernel-mode driver creation, it will take time to fully absorb this book's content. If you're interested in the Microsoft Windows driver model, I recommend this book.
My work on JavaUSB has shown me that it's not easy to develop a Java/USB API. The API must be useful while not exposing low-level hardware details. It must also be abstract enough to be usable across OSes. Because I chose to target the Windows 98 SE/ME/2000 OSes (for practical reasons), I leave you with the following homework: try porting JavaUSB to Linux or some other OS.

Resources

- Sample code for this article
- "Access USB Devices from Java Applications"
- "Build Screensavers with a Custom Screensavers Library in Borland C++"
- C++ Builder Downloads: Borland C++ 5.5.1 compiler
- JCP home page for JSR 80 (Java USB API)
- JCP home page for JSR 82 (Java APIs for Bluetooth)
- JSR080: javax.usb
- "Java Tech: Acquire Images with TWAIN and SANE, Part 1"
- Javadoc for javax.usb API
- jUSB: Java USB project page
- jUSB: Java USB API for Windows project page
- java.net forum topic: USB support (especially for Windows)
- libusb project page
- libusb-win32 project page
- "The New Java USB API"
- "The USBView.exe Sample Program Does not Enumerate Devices on Pre-Windows-XP-SP1-based Computers"
- USB 1.1 specification (.zip, 1.7MB)
- USB 2.0 specification (.zip, 7.86MB)
- Universal Serial Bus Wikipedia entry
- "What Would You Like to Do with Java and Cannot?"