4 Java USB API for Windows

This chapter will give an overview of how the Java USB API for Windows will be implemented. To understand this approach, we give a short introduction to the USB driver stack for Windows. At the end, we present the final framework which we are going to implement as part of this project.

4.1 USB Driver Stack for Windows

The developers of Microsoft Windows indeed transformed the USB specification as close as possible to the Windows operating system environment. Therefore, we find some layers of drivers that support USB to the Windows operating system. All layers shown in Figure 5 are closely related to Figure 2 in Chapter 3.2. Figure 5 illustrates the USB driver stack for Windows.

USB Layers

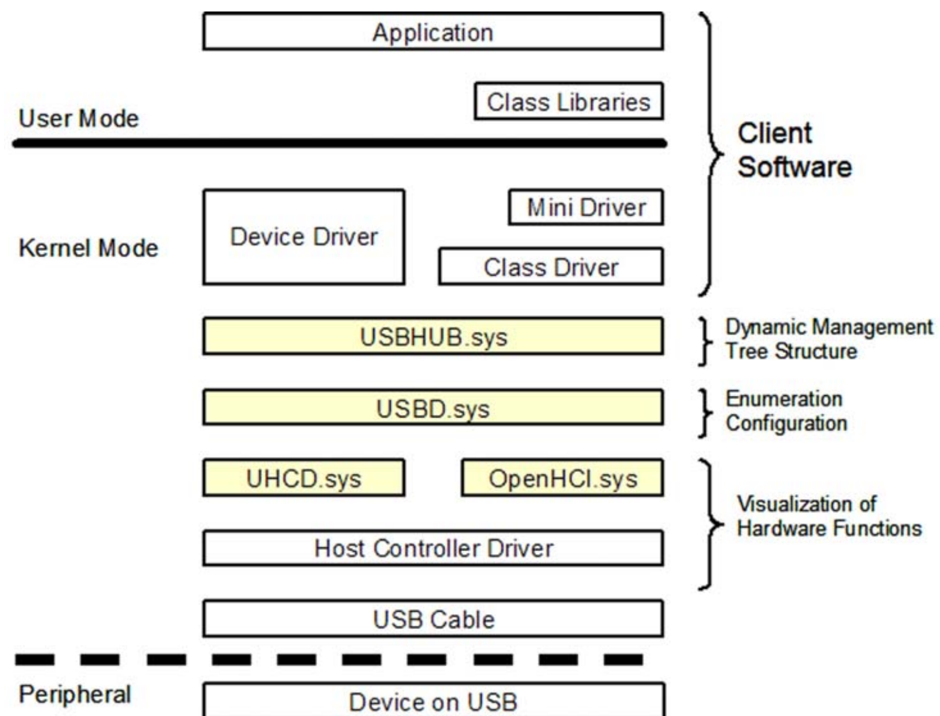

Figure 5: USB driver stack for Windows

Host Controller Driver

The visualisation of the hardware functions of the USB Host Controller for operating system components takes either place through the USB host controller driver **UHCD.sys** or the open host controller Interface **OpenHCI.sys** driver. The interfaces to those drivers are not documented by Microsoft and therefore not usable for end users [1].

USB Driver (USBD.sys)

The driver above the host controller driver is **USBD.sys** called the USB driver. This driver plays an important role in the USB driver model. Configuration of the attached devices, requests of device information, monitoring and control of the bus structure is all part of this driver. Further, it is responsible for allocation and monitoring of the available resources such as bandwidth and power management. Another task of the USB driver is to control the data stream in both directions and exporting interfaces to controlling several USB devices.

Configuration

The Configuration of the USB devices is handled by the default pipe number zero (EP0). For each USB device, the USB driver creates a data channel to endpoint zero (EP0) after the operating system has been booted. Through this channel, configuration is done beginning at the root hub. The configuration process encompasses requesting the descriptor of each USB device and assigns a unique address to the device. With help of the descriptor data (especially the vendor id and the product id) the corresponding device driver can

be localized, loaded, and further configuration can be applied to the device using the loaded device driver.

Interface to the USB Driver

The interface to the USB driver (USBD.sys) is documented by Microsoft to user mode direction (see DDK [6] for more information). It establishes the initial point for the utilization of USB through applications. User programs running in user mode do not directly have access to this interface. The realisation of such access involves an additional driver module (either a **Device Driver** or a **Class Driver** as shown in Figure 5) that react to certain function calls from the user mode and pass them down to the USB driver (as I/O request packet (IRP) containing an USB request block (URB)).

Hub Driver (USBHUB.sys)

The tree structure of the USB is managed by the hub. The configuration of the hubs and their dynamic administration of the tree structure is handled by the hub driver (**USBHUB.sys**). The major tasks of a hub driver are:

- configuration of the hubs
- controlling and power management for each port
- to initiate the signals suspend, resume and reset at each port.

Class Driver

Mini Driver

On top of the hub driver we find a lot of drivers that belong either to a device specific driver, a mini driver or a class driver. A class driver manages a group of devices which have similar functions. A mini driver is used when a device nearly fits into a class driver but some functions differ from the class driver. The mini driver implements only the extra features that are not supported by the class driver. If a device does not fit to any class driver then the vendor has to supply a device specific driver. This results in supplying a ".sys" driver file for installation of the USB device.

Knowing the USB driver stack for Windows leads to the framework of the Java USB API.

4.2 Framework of the Java USB API

Conceptual Design

The core Java USB API provides a singleton host that monitors all USB busses. The host is responsible for enumerating the USB devices on the Java side and update its listeners as soon as a device has been attached or removed. We can see a close correlation to the work of the USB hub driver (USBHUB.sys) and the USB driver (USBD.sys) in the USB driver stack. In fact, they are responsible for the tree structure and to enumerate the devices.

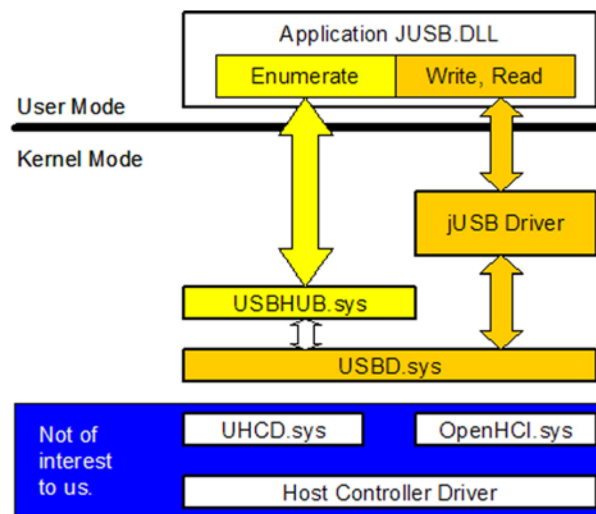

Figure 6: Java USB API layer for Windows

According to the usbview example delivered with the DDK [6], we know that it is possible to enumerate all the devices (hubs included) and even the host controllers. John Hyde shows another example how to display the USB tree

structure in Windows [2]. The common thing both examples have in common is that they are executed in user mode. The conclusion is that we do not have to write a driver to enumerate and control the USB tree structure for the Java USB API. Of course, these user mode functions are performed with the *DeviceIoControl* WinAPI function which uses the handle to the corresponding hub driver. A driver is still required but it is already supplied by the Microsoft operating system. A small disadvantage is that undocumented I/O Control (IOCTL) codes are used. This forces one to use the examples as documentation, which is far away from an optimal documentation. Anyway, creating a framework using existing user mode functions simplifies the writing of the Java USB driver. We use the Win API user mode function as shown in the usbview example to enumerate and monitor the USB tree structure as shown in Figure 6.

To perform device specific operations we need to write a device driver mapping the user mode function to the related kernel mode function as shown in the right part of Figure 6. This involves the jUSB driver to handle different kinds of IOCTL codes to maintain all the functionality supported by the Java USB API.
The Question may arise of how to assign the jUSB driver to any kind of USB device. Usually, a USB device is plugged in and the driver is loaded automatically. This is still preferable but instead of loading the original driver for the USB device we want the system to load the jUSB driver (details about the installation of a new device are given in Appendix D). Of course, this will take away all the functionality the origin driver supported but this functionality should now be provided by the Java USB API. Using the new API we can build the functionality we want from the device in Java and do no longer have to care about C, JNI and driver writing on the Windows platform. Chapter 5 is going to present in a first part the implementation of functions not using the JUSB driver while the other part describes the driver implementation for the Java USB API.

Replace the Origin Driver through the JUSB Driver