Java USB API for Windows

Diploma Thesis at the Institute for Information Systems, ETH Zürich

Michael Stahl

September 18th 2003

Diploma Professor:
Prof. Moira C. Norrie

Supervisor:
Beat Signer

Contents

List of Tables

List of Figures

1 Introduction

The goal of this diploma thesis is to extend the Java USB API to the Windows operating system as a part of the open source project jUSB [18].

This documentation presents an overview of the universal serial bus (USB) to provide the fundamental understanding of the Java USB API. Common USB terminologies are also explained in detail.

The concept of the jUSB API for Windows will be introduced which includes a presentation of the USB driver stack for Windows and the principal framework of the Java USB API.

The design approach to implement the usb.windows package for the Java USB API is separated into two parts. One part deals with the enumeration and monitoring of the USB while the other part looks into the aspects of communicating with USB devices in general. Both parts are implemented using Java Native Interface (JNI) to access native operation on the Windows operating system. The jUSB dynamic link library (DLL) provides the native functions that realise the JNI interface of the Java usb.windows package.

Communication with an USB device is managed by the jUSB driver. The structures and important aspects of the jUSB driver are introduced in chapter 6. The chapter itself is a summary and covers only some fraction of the driver implementation. A lot of useful information about driver writing and the internal structures can be looked up in Walter Oney's book "Programming The Microsoft Driver Model" [4].

A lot of important programs and resources are used to work with the Java USB API for Windows project. Therefore, two chapters have been included to simplify the installation for end users and developers.

Acknowledgement

I am very grateful to my supervising assistant Beat Signer for always having the time to answer my question. Thanks also to the other members of the GlobIS group, the members of the OMS-Lab and especially to Prof. Moira C. Norrie, for the opportunity of my diploma thesis.

2 Motivation

The European project Paper^{++} (Disappearing Computer Programme, IST-2000-26130) develops various technologies to enhance physical paper by digital augmentation [23]. As a part of the project, we are evaluating different kind of reading devices for position detection on paper. Due to the lack of an existing USB driver for Java programming environment, at the moment we are restricted to using only readers transmitting information over the serial port (RS232).

The goal of this diploma thesis is to develop a USB driver for the J2SE 1.4 programming environment [9] supporting any kind of USB device (different device classes). Based on experience from an earlier project Java HID-USB API for Windows), a Windows USB binding conforming to the already existing jUSB interface definition [18] should be implemented.

The idea is to provide some kind of a Java wrapper classes based on the Java native Interface (JNI) [17] mapping the corresponding Java USB calls to the underlying Windows driver system. The existing jUSB API will be used as a guideline and the final Java USB driver should support reading from the USB port as well as transmitting information to USB devices.

3 USB Overview

3.1 USB Terminology

The USB specification introduced new terms that are used throughout the USB literature. This section introduces those terms and presents an overview.

Host

Root Hub

A typical configuration has a single PC host with multiple devices interconnected by USB cables. The PC host has an embedded hub, also called the root hub, which typically contains two or more USB ports.

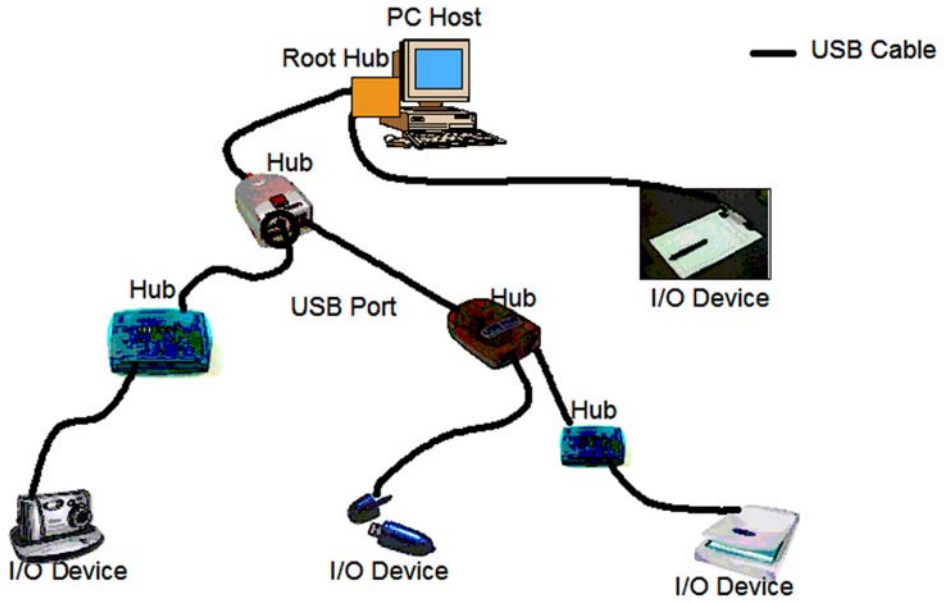

Figure 1: Standard USB designation

Device configuration range from simple to complex:

Hub

- **Hub**: If a device contains only additional downstream USB ports, then it is called simply a hub.

I/O Device

- **I/O device**: An I/O device adds capability to the host. It has a single upstream connection and interacts with the real world to create or consume data on behalf of the PC host.

Compound Device

- **Compound device**: If a device includes both I/O and hub functionality, it is called a compound device. A keyboard that includes additional USB downstream ports is such an example.

Composite Device

- **Composite device**: If a single device implements two or more sets of device functions, it is called a composite device. For example an eyecam camera with a camera and dual audio channels and a microphone is a composite device

As far the PC host is concerned, devices are the important feature, and as many as 126 devices can be interconnected using external hubs up to five levels deep (in Figure 1 the hub level is three levels deep).

Speed

USB 2.0 supports three device speeds. The USB specification version 1.1 defined only two device speeds, such as **low speed** at 1.5 Mbps and **full speed** at 12 Mbps. The **high speed** at 480 Mbps has been added in USB specification 2.0. Low speed devices are the cheapest to manufacture and are adequate for supporting low data rate devices such as mice, keyboards, and a wealth of other equipment designed to interact with people. The introduction of high speed data rate enables high bandwidth devices such as full colour page scanners, printers and mass storage devices.

3.2 PC Host

A typical configuration has a single PC host. The PC host runs an USB aware operating system software that supports two distinct functions: initialization and monitoring of all USB devices.

The USB initialization software is active all the time and not just during the PC host powered-on. Because the initialization software is always active, USB devices can be added and removed at any time, also known as Plug and Play. Once a device is added to the host, the device is enumerated by the USB initialization software and assigned a unique identifier that is used at run time.

Figure 2 shows how the USB host software is layered (layering supports many different software solutions).

Client Software

On the top **Client Software** is being executed to achieve the desired USB device functionality. The application software is written in user mode and therefore does not harm the operating system when errors occure. Class libraries gain access in user mode to class driver functions. A class is a grouping of devices with similar characteristics that can be controlled by a generic class device driver. Examples of classes include mass-storage devices, communication devices, audio devices, human-interface devices (HID) and some more that can be found at www.usb.org. If a device does not fit into one or more of these predefined classes, then a specific device driver has to be written for the device. Device drivers are executed in the kernel mode and must therefore be validated and tested to be error free. Next to the Client Software layer follows the **USB System Software** layer. Enumerating processes and USB monitoring is the major task of this layer. It is also responsible for recognising removal and attachments of devices. The deepest layer is the **USB Host Controller**. It is the hardware and software that allows USB devices to be attached to the host.

System Software

Host Controller

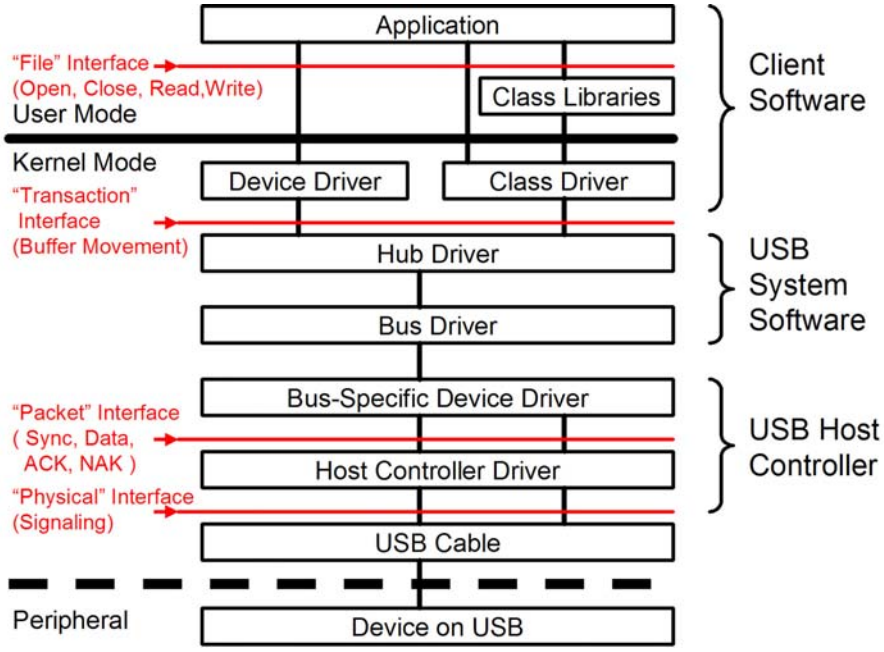

Figure 2: PC host software for USB is defined in layers

3.3 USB Cable

A USB Cable transports both power supply and data signals.

The power supplied by the USB cable is an important benefit of the USB specification. A simpler I/O device can rely on the USB cable for all its power needs and will not require the traditional “black brick” plugged into the wall. The power resource is carefully managed by the USB, with the hub device playing the major role. A hub or an I/O device can be self-powered or bus-powered.

- **Self-powered** is the traditional approach in which the hub or I/O device has an additional power cable attached to it.
- A **bus-powered** device relies solely on the USB cable for its power needs and is often less expensive.

Connectors

The USB cable connectors were specifically designed with the power pins longer than the signal pins so that power would always be applied before signals. Two different connector types were defined, to ensure that illegal configurations could not be made. An **“A”-type connector** defines the downstream end of the cable, and a **“B”-type connector** defines the upstream end (Figure 3).

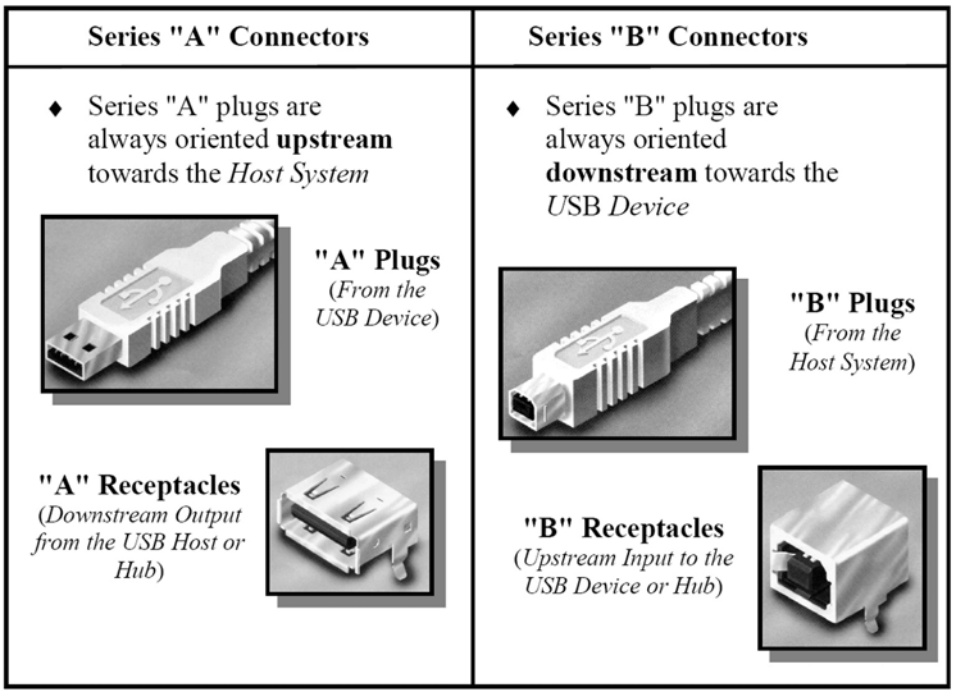

Figure 3: USB cable connector types [27]

Maximum Length

The maximum cable length is 5 meters between a hub and a device. With up to five levels of hubs we reach a length of 30 meters from the PC host to the device.

3.4 Hub Device

The hub has two major roles: power management and signal distribution.

External Hub

An external hub has one upstream connection and multiple downstream connections. The USB specification does not limit the number of downstream connections. The most popular hub size has four downstream ports.

A hub can be self-powered or bus-powered. The self-powered hub can provide up to 500mA to each of its downstream ports while a bus-powered has a maximum of 500mA to all the ports.

3.5 I/O Device

A PC host creates data for or consumes data from the real world as shown in Figure 1. A scanner is a good example of a data-creating I/O device and a printer of a data consuming I/O device.

Logical View

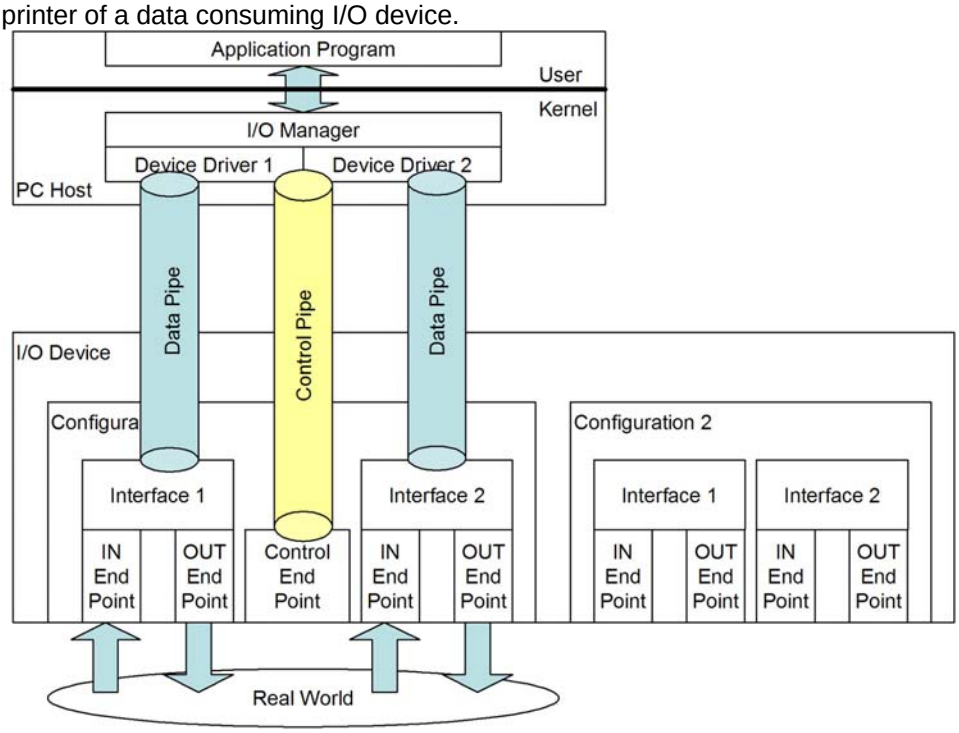

Figure 4: Logical view of an I/O device

The software (or logical) view of the USB connection is shown in Figure 4. This diagram is best explained bottom up.

Endpoint

The term **endpoint** is used to describe a point where data enters or leaves a USB system. An IN endpoint is a data creator, and an OUT endpoint is a data consumer. Note that the data direction is relative to the PC host – if we remember that the PC host is the "master" controlling all data movements, then the data direction is easy to understand.

Interface

A typical real-world connection may need multiple IN and/or OUT endpoints to implement a reliable data-delivery scheme. This collection of endpoints is called an **interface** and is directly related to a real-world connection. The operating system will have a software driver that corresponds to each interface. The

Pipes

operating system uses the term pipe to describe the logical connection between a software driver on the PC host and the interface on the I/O device. There is always a one-to-one mapping between software drivers and interfaces.

Configuration

A collection of interfaces is called a **configuration**, and only one configuration can be active at a time. A configuration defines the attributes and features of a specific model. Using configuration allows a single USB connection to serve many different roles, and the modularity of this system solution saves in development time and support costs.

3.6 Information Flow

Transfer Types

USB defines four methods of transferring data, as summarized in Table 1. The methods differ in the amount of data that can be moved in a single transaction in whether any particular periodicity or latency can be guaranteed, and in whether errors will be automatically corrected. Each method corresponds to a particular type of endpoint.

Transfer Type	Description	Lossless?	Latency Guarantee?
Control	Used to send and receive structured information of control nature	Yes	Best effort
Bulk	Used to send or receive blocks of unstructured data	Yes	No
Interrupt	Like a bulk pipe but includes maximum latency	Yes	Polled at guaranteed minimum rate
Isochronous	Used to send or receive blocks of unstructured data with guaranteed periodicity	No	Read or written at regular intervals

Control Transfer

Bulk Transfer

Interrupt Transfer

Isochronous Transfer

Table 1: USB data transfer types

Endpoints have several attributes in addition to their type. One endpoint attribute is the maximum amount of data that the endpoint can provide or consume in a single transaction. Table 2 indicates the maximum values for each endpoint type for each speed of device. In general, any single transfer can involve less than the maximum amount of data that the endpoint is capable of handling.

Transfer Type	High Speed	Full Speed	Low Speed
Control	64	8,16,32 or 64	8
Bulk	< 512	8,16,32 or 64	not allowed
Interrupt	< 1024	< 64	< 8
Isochronous	< 3072	< 1023	not allowed

Table 2: Allowable endpoint maximum packet sizes in bytes

3.7 Descriptors

USB devices maintain on-board data structures known as descriptors to allow for self-identification to host software. Table 3 lists the different descriptor types.

Descriptor Type	Description
Device	Describes an entire device
Configuration	Describes one of the configurations of a device
Interface	Describes one of the interfaces that is part of configuration
Endpoint	Describes one of the endpoints belonging to an interface
String	Contains a human readable Unicode string describing the device, a configuration, an interface, or an endpoint.

Table 3: Descriptor types

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

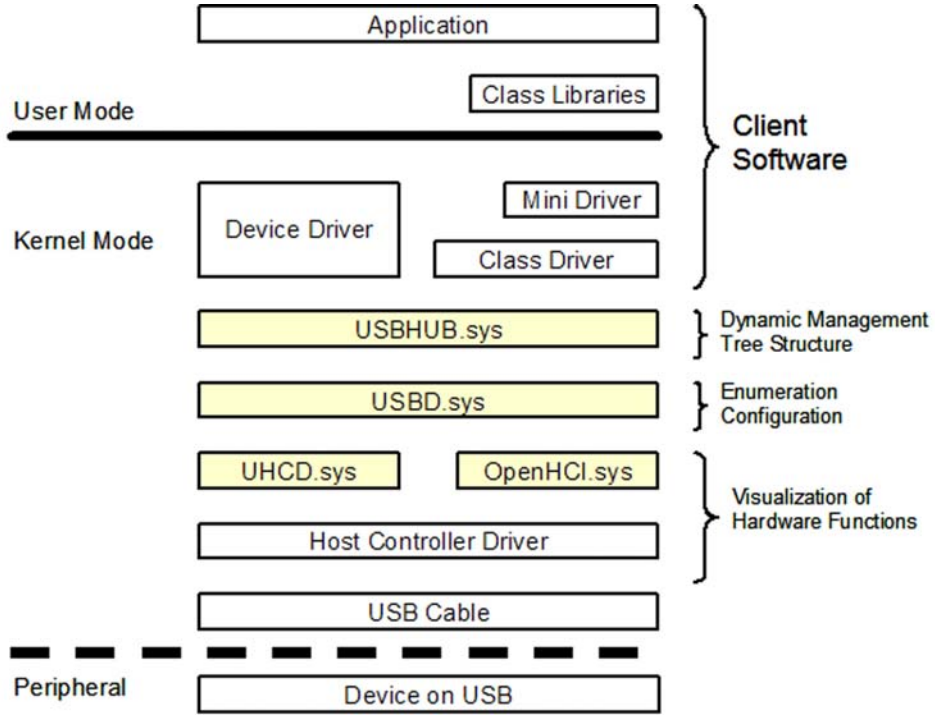

Figure 5: USB driver stack for Windows

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.

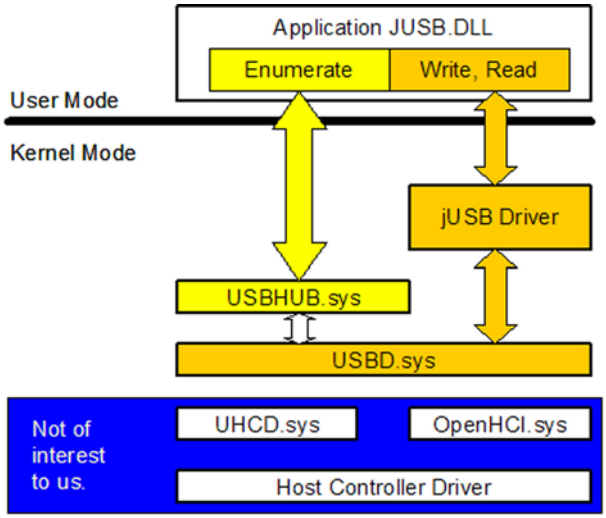

Figure 6: Java USB API layer for Windows

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.

Replace the Origin Driver through the JUSB Driver

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.

5 Java USB API usb.windows Design

Introduction

The design of the usb.windows package for the Java USB API is built on the usb.core package. This is a constraint of my diploma thesis and therefore I am bound to some given relation. The functionality is as noted in the design phase separated in two parts. The first part contains all work to create a Host and enumerate all devices on the busses. We even get the device descriptor and the default configuration descriptor (index equal to zero) of every USB device attached to the bus. The first part does not depend on the driver the device uses. To access all information as mentioned before we must have objects of a *DeviceImpl* instance. The functionality of *DeviceImpl* objects is restricted. There is no way to use functions which are part of the *DeviceSPI* class (see javadoc of Java USB API [10]).
The second part depends on the driver the device uses. All devices that are not configured to use the jUSB driver will be put into the *NonJusb* class. The *NonJusb* class does as well implement the *DeviceSPI* interface, because it is necessary according to the usb.core API, but it will throw only *IOExceptions* indicating that *DeviceSPI* cannot be used to access the device. In the opposite way we put all devices using the jUSB driver in the *JUSB* class. The *DeviceSPI* interface is partly implemented and supports reading data and doing control transfer to the device.
In the following section we are going to describe how the first part of the Java USB API application is implemented and how the communication looks like underneath the responsible objects.

5.1 Host and Enumeration Processes

Windows Class

The *Windows* class which extends the *HostFactory* class contains to inner classes the *HostImpl* and the *Watcher* class. The *Watcher* class (Figure 7: ①) implements the *Runnable* interface and is therefore used for thread activity. The

Watcher Class

Windows class runs the *Watcher* thread as a daemon thread. This means as soon as our main application is finished, the Watcher thread will terminate as well. Within the *run* method, method *scan* should be called anytime when something has changed on the bus. The notification of USB structure changes we wanted to implement with a call-back function to the USB native on Windows with the aid of the *RegisterDeviceNotification* function. It is better if the *Watcher* only starts the *scan* method when really something has changed on the bus. In the usbview example this notification is done with *RegisterDeviceNotification*. This will call an event which can be handled in the *WindowProc* call-back function. Unfortunately this call-back depends on a window application. This means we can only fetch that event in a window object. So far we have only native calls in the jUSB DLL. We tried to make a fake window, which was not visible for the user to catch than the WM_DEVICECHANGE event, which is broadcasted when a change on the USB happened. This topic has some related aspects in user forums, but it seems to be that every one fight with the same problem. *RegisterDeviceNotification* is not usable in a DLL! If someone gets a solution we will be happy to here it.

Polling

Anyway the *Windows* class polls now every two seconds the bus to look for changes on the USB structure. The major task of the *scan* method in the *Watcher* class is to find out through a native call how many USB host controllers the current machine supports. It creates for every USB host controller a new *USB* object. It checks first of all if there already exist a *USB* object. In the case of already having a *USB* object it will just call the *scanBus* method of this *USB* object to monitor if changes have been occurred. In the other case a *USB* object will be created and put to the *HashTable* of current busses (Figure 7: ②).

HostImpl Class

The *HostImpl* class takes care over all USB busses found on the computer. It implements all methods from the *usb.core.Host* class.

The *USB* class is responsible to keep control over its bus. This means an *USB* object knows about all its devices and can access them through an address that

USB Class

is given at enumeration time. The major work is done through the *scanBus* method. At first, it creates the root hub which exists only once for a USB bus (host) (Figure 7: ③). To avoid misunderstanding between the names of the *Host* class as they can be found in the Java USB API and the host of an USB bus, we briefly explain their differences. The *Host* class is an abstract definition for all USB busses on a system. Every USB bus consists always of a host and a root hub. The *USB* class does implement a host as defined in the USB specification. Therefore we can say that the USB class itself represents a host as known in common sense. Every *USB* object contains a root hub. This fact indicates that the *USB* class itself must be a host in USB topology.

Naming Convention

scanBus

The *scanBus* method in the *USB* object gets now the native rootHubName of that given hostcontroller and creates a new *NonJUSB* for that root hub (Figure 7:③). This will be done in either way if we have already an existing root hub or not. This design is made in that way because we call the *enumerateHubPorts* method in *DeviceImpl* recursively to get all devices on the bus and the bus structure itself. The root hub creation starts this recursion and therefore we need to create it for each scan. In other words the enumeration is done by the devices itself and every device that exists on the current bus will add itself to the *USB* object. To avoid always getting notified by the *USBListener* that a new root hub is created we check if we already have a root hub for that bus and do only notify the *USBListener* when there was no root hub before. In the other case we create the root hub again without notifying its listeners.

Enumeration

The *NonJUSB* and the *JUSB* class delegate most work to their superclass *DeviceImpl* (Figure 7: ④). The *DeviceImpl* class has two constructors, one is used for the root hub and the other one is used when the device is either a USB device or an external root hub. The *DeviceImpl* object will get some information about the USB device itself. Through native calls it will get to know how many ports it has and what kind of device type is connected to each of their ports. The port can be free (no device connected) or a device or even a hub can be connected. If a hub is connected we recursively call the *enumerateHubPorts* method to get all the children of the hub (Figure 7:⑤).

The *enumerateHubPorts* method is in charge to update the bus structure. It knows the recent structure from the oldDevices member. It compares those members with the currently processed device. If the currently processed device can be found at the same address in the oldDevices member, we check the device to make sure that it is still the same. In case of a device (not a hub) the decision is made in one step. We only check if the device's current unique ID correspond to the recent device unique ID. If not, we remove the recent device from the bus and add the current device to the bus and inform the listeners about a removal and an attachment. In case of having a hub we have to check all its ports recursively down, to make sure that everything remains the same. If we remove a hub, we have to inform the listeners about a removal of the hub and all its children that used to be connected to it. In both cases when a device is new to the bus or it has changed we create a new *NonJUSB* or *JUSB* object depending on the friendly driver name and add it to the *USB* object (Figure 7: ⑦).

JUSB or NonJUSB

At the moment when a friendly driver name starts with "JUSB Driver --:", this is a public string constant, called A_JUSB_DRIVER declared in the *Windows* class, we create a *JUSB* object and otherwise a *NonJUSB* object.
In either way whether we created a *JUSB* or *NonJUSB* object we get the device descriptor through the *getDeviceDescriptor* method from their superclass. This method is not part of the *DeviceSPI*, but makes it possible to read the descriptor of any device (Figure 7 : ⑧). Furthermore, we get the default configuration descriptor in the same way (Figure 7: ⑨). If a device has multiple configurations, we only get the configuration with index zero!

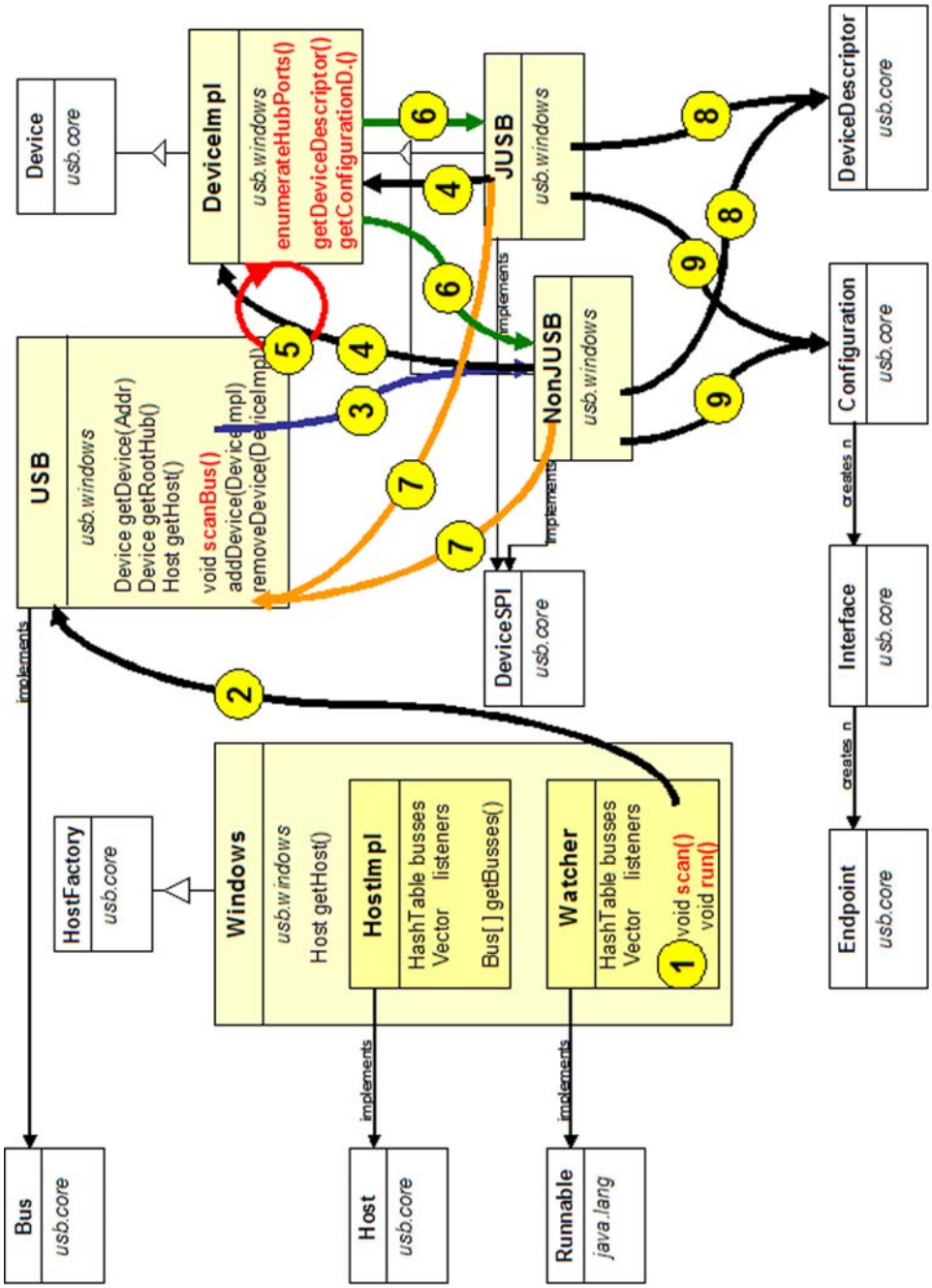

Figure 7: Class overview with its interaction

5.2 Windows Class

The *Windows* class which extends the *HostFactory* contains two inner classes the *HostImpl* and the *Watcher* class. This section explains the *Window* class.

Creating a Host

The Java usb.core package supports classes that need to be implemented in the usb.windows package. The core subject of his API is the *HostFactory* itself. The *HostFactory* class is responsible to setup an OS-specific environment. The *HostFactory* in the usb.windows package is in charge to instantiate a valid environment for Windows XP and 2000. This is all done with the following code:

```
Host host = HostFactory.getHost();
```

Table 4: Creating an USB host

With the method *getHost* of the interface *HostFactory* we get a *Host* object returned. The Host is responsible to monitor all universal serial busses on a given

machine. There can be more than one USB host controller on a computer. Every USB host controller can manage up to 126 USB devices.

Terminology

To prevent a misunderstanding in the USB topology of Windows operating systems, one universal serial bus is managed by one Host Controller. So the host we create through the Java USB API as in the example above, has nothing to do with the Host Controller from the Windows operating system. In fact a host controller in the Windows OS corresponds to a Bus object according to this Java USB API. This means that if we have more than one host controller on our Windows PC, we will also have more USB busses. The number of busses is equal to the amount of host controllers.

HostImpl Class

The *Windows* class in the usb.windows package has to implement the *Host* interface. This includes all methods in the *Host* interface. The following table will list those methods:

Bus []	getBusses()	Returns an array of Bus instances. Remember the number of Bus instance will be equal to the number of host controllers on your computer.
void	addUSBListener(USBListener)	Adds a call back for USB structure changes. As soon as a device or a bus gets removed or attached to the bus, any class which extends the class USBListenerAdapter gets notified. The abstract class USBListenerAdapter implements already the USBListener interface. This is the reason why we have to extend our class that will be in charge to handle USB structure changes from the USBListener-Adapter.
void	removeUSBListener(USBListener)	Will remove the callback for USB structure changes.
Device	getDevice(PortIdentifier)	not implemented yet!

Table 5: Host Interface of the Java USB API

Bug

The *HostFactory* dynamically loads the host for the operating system it runs on. There is a bug in the usb.core package. The method *getHost* checks for the operating system name and then tries to load the class usb.<os-name>.<OS-name>. In Java this looks as follows:

```
String os = System.getProperty ("os.name");
String classname = “usb." + os.toLowerCase () + "." + os);

Object   temp;
temp = Class.forName (classname);
temp = ((Class)temp).newInstance ();
return ((HostFactory) temp).createHost ();
```

Table 6: Dynamically loading of HostFactory

Operating Sytsem Names

This works perfectly for Linux. Linux becomes the OS name “Linux” and this results to a correct classname: “usb.linux.Linux”.
Among Windows XP the OS name is “Windows XP” and the classname would look like this: “usb.windows xp.Windows xp”. This is not a valid Java package name and even not a valid class name, because spaces within a class or package name are not allowed. For windows 2000 the OS name is “Windows 2000”!

We fixed this bug by checking the operating system’s name for windows and if “windows” is a substring of the operating system name, we will load usb.windows.Windows package. The Windows version should run among Windows 2000/XP/2003. There is not a guarantee that this Windows class supports Windows 95,98 and ME.

Watcher

The *Watcher* class is a daemon thread which is responsible of the current Host.

Watcher

It monitors all changes of the USB structure. At initialization, we scan all USB busses on the system and create the appropriate *USB* object. In a later scan we check if the busses, exactly there host controller names, remains. When a bus has been removed we notify the listeners about a removal or attachment of a bus. This case will hardly ever happen, because the removal or attachment of a bus is usually done by exchanging a piece of hardware which should be done while power off.

To get all host controller on the Windows operating system we use the native method *getHostControllerDevicePath(i)*. This method returns a device path to the ith host controller. The variable i has to be incremented from zero to the amount of host controllers. It will return *null* as soon the variable i is to high.

5.2.1 Windows Class Native Side Design

SetupDiXxx-Function

The implementation of *getHostControllerDevicePath* is in the file jusbJNIwindows.cpp. The new guid interface is used to get all host controllers on the Windows operating system. The most work is done in using the *SetupDiXxx* API function. Because Windows 2000 does not support the GUID interface for host controller we need another way to get the device path on Windows 2000 operating system. This is solved in that the device path always starts with "\\.\HCD**x**" where **x** is the ith host controller.

```
// Windows XP
getDevicePath(  &GUID_DEVINTERFACE_USB_HOST_CONTROLLER,
                (int)number);

// Windows 2000
getHostControllerPath(int number);
/*
This function is being called, when we execute getDevicePath with number 0
and fail, which means that we are not able to find at least one host. Then we
try the Windows 2000 function. If we fail again, we do not have a host con-
troller on the system or are running under another operating system.
*/
```

Table 7: getDevicePath and getHostControllerPath function in jusb.cpp

Device Path

What is a device path?
A device path is used to execute *CreateFile* WinAPI function which returns a device handle to the specific device. With the device handle we can call *DeviceIoControl* with an appropriate IOCTL code to get more information about the device itself. In that case the device would be the host controller.

5.3 USB Class

The *USB* class implements the *Bus* interface from the usb.core API. This involves to implement the following methods shown in Table 8:

Bus Interface

String	getBusId()	Returns a host specific stable identifier for this bus.
Device	getDevice(address)	Returns an object representing the device with the specified address (1 through 127), or null if no such device exists.
Host	getHost()	Returns the USB host to which this bus is connected.
Device	getRootHub()	Returns the root hub of this bus, if it is known yet. The root hub is always the device with address 0. This is according to the USB implementation of the Java USB Windows API

Table 8: Bus interface of the Java USB API

An additional method is implemented to the *USB* class. This method is called *getBusNum* and listed in Table 9.

int	getBusNum()	Returns the number assigned to this bus. This number is from 1 to the number of host controller on the Windows machine.

Host Controller Name

Table 9: Additional method getBusNum in the USB class

The method *getBusId* returns the host controller name of the Windows operating system. This name may as follows:
„Intel(R) 82801DB/DBM USB Universal Host Controller - 24C4"
This name is unique according to the other host controllers on a Windows machine.

The only native method in *USB* class is *getRootHubName*. *ScanBus* uses this method to start the enumeration process. It creates with the given root hub name a *NonJusb* object, which itself starts the recursive enumeration by calling its superclass *DeviceImpl*. The *enumerateHubPorts* method of *DeviceImpl* is responsible to let the recursion run or terminate. As soon there are not more hubs found on a port the recursion will stop.

5.3.1 USB Class Native Side Design

getRootHubName

The file jusbJNIusb.cpp contains the implementation of *getRootHubName*. To succeed the *getRootHubName* method a valid host controller device path has to be given as input parameter. The hostControllerDevicePath is a private member of the *USB* object. Its initialization is done in the constructor of the *USB* object. Refer to Table 10 to see the code fragment of *getRootHubName*.

```
1  JNIEXPORT jstring JNICALL Java_usb_windows_USB_getRootHubName
2  (JNIEnv *env, jobject obj, jstring hcdDevicePath)
   {
3    …
4    hcdHandle = CreateFile( hcdDevPath, …, …. );
   …
5    rootHubName = getRootHubName(hcdHandle);

6    if(!CloseHandle(hcdHandle)) {  …} // an error occured
7  return rootName;
   }
```

Table 10: getRootHubName JNI function

Description of a JNI function

1. The head of a Java Native Interface method looks mostly this fashion. The complicated and not very readable function name is coming from the JNI naming convention. Every native method starts with **Java**_ followed by the package names (**usb_windows_USB**_), separated by a **'_'** instead of a **'.'** as we are used to on Java side. Finally we append the native method name to the previous name.
2. The parameter *env* stands for the Java environment and the *obj* parameter refers to the Java class this method belongs to. The third parameter in that case is now the host controller device path as a type of jstring.
3. Some initialisation has to be done at this point. We have to convert the jstring hcdDevicePath to a type of PCHAR hcdDevPath variable. Look at the source code to see how this is done.
4. To get some information from the host controller we need at first a host controller handle, which is done through *CreateFile* WinAPI function.
5. We call *getRootHubName* method with a valid host controller handle. See at the next section how this function succeeds the demanded action.
6. Finally we always close a handle. Open handles can slow down the

operating system.
7. We return the rootName as type of jstring.

The interesting thing in the *getRootHubName* JNI function is the C method *getRootHubName* that takes a host controller handle as its argument. The *DeviceIoControl* API function is used to send the IOCTL_USB_GET_ROOT_-HUB_NAME command to the host controller. This IOCTL code is undocumented by Microsoft but used in the usbview example in the DDK. Its use can be summarised as:

Call *DeviceIoControl* WinAPI function with function code IOCTL_USB_GET_ROOT_HUB_NAME to receive the USB_ROOT_HUB_-NAME structure. We will receive a structure which contains only 6 bytes, 4 bytes for the AuctualLength and 2 bytes for the RootHubName which is an array of wide chars. Both are members of the USB_ROOT_HUB_NAME structure. In a second way we have to allocate memory in the size of ActualLength for the output buffer and call *DeviceIoControl* WinAPI function again to obtain the whole root hub name

The following code snippet shows the important parts. Error handling is omitted to clarify the main aspects.

```
   PCHAR getRootHubName(HANDLE HostController)
   {
      BOOL              success;
      ULONG              nBytes;
1     PUSB_ROOT_HUB_NAME  pRootHubNameW;
2     PCHAR              rootHubNameA;
   …
3     pRootHubNameW = (PUSB_ROOT_HUB_NAME)
                        GlobalAlloc( GPTR, sizeof(USB_ROOT_HUB_NAME));

4     success = DeviceIoControl(HostController,
                        IOCTL_USB_GET_ROOT_HUB_NAME,
                        0,
                        0,
5                       pRootHubNameW,   //&rootHubName
6                       sizeof(USB_ROOT_HUB_NAME),  //sizeof(rootHubName)
                        &nBytes,
                        NULL);

7     nBytes = pRootHubNameW->ActualLength;   //rootHubName.ActualLength
8     GlobalFree(pRootHubNameW);

9     pRootHubNameW = (PUSB_ROOT_HUB_NAME)GlobalAlloc(GPTR,nBytes);

10    success = DeviceIoControl(HostController,
                        IOCTL_USB_GET_ROOT_HUB_NAME,
                        NULL,
                        0,
                        pRootHubNameW,
                        nBytes,
                        &nBytes,
                        NULL);

11    rootHubNameA = WideStrToMultiStr(pRootHubNameW->RootHubName);
12    GlobalFree(pRootHubNameW);

13    return rootHubNameA;
   }
```

Table 11: IOCTL_USB_GET_ROOT_HUB_NAME

1. A pointer to a USB_ROOT_HUB_NAME structure, which is declared in usbioctl.h [28]
2. A pointer to the return value
3. Allocate memory to keep a USB_ROOT_HUB_NAME structure. The GPTR Flag indicates that the memory is fixed and its content initialized with zeros.

4. Get the root hub name
5. Output buffer
6. Output buffer size
7. The length of the root hub name
8. free the recently allocated memory for the pRootHubNameW pointer
9. Allocate memory to keep the entire root hub name
10. Get the root hub name with an output buffer big enough to keep the entire root hub name
11. convert the wide string to a 8 bit string
12. free the memory of pRootHubName
13. return the root hub name

5.4 DeviceImpl Class

Description

The *DeviceImpl* class is one of the core classes for enumerating the USB. It is only used for hubs. The whole enumerating process is done to search for a hub and then to check its ports to determine what kind of devices are attached to it. We get the device and configuration descriptor by asking the hub driver about the devices that are attached to the ports. We do never access the device directly, but rather through the hub itself. We gain a lot of useful information in asking the hub about the devices that are attached to it. By means of that information a decision can be made whether the device uses the jUSB driver or not. This will result in creating a *JUSB* object for a device using the jUSB driver and for all other ones we get a *NonJUSB* object. To satisfy all this constraints help, is needed from a native method to get access to a hub by *openHandle* and *closeHandle* method. This methods dispatch just to WinAPI functions *CreateFile* and *CloseFile*. As soon as we got a handle to a hub, we can gather information about the ports of the hub and the hub itself. At first we want to know how many ports the current hub actually has. This request will be satisfied with the native method *getNumPorts*.

getNumPorts

In a second step we iterate now over all ports of this hub and do the following steps at each port:

getAttachedDevice

1. *getAttachedDevice* implemented as a native method is first called. it returns a constant depending on the ports connectivity. This is either the value EXTERNAL_HUB, USB_DEVICE or NO_DEVICE_CONNECTED.
2. If we have no device connected we go to the next port.
3. For an external hub or a device we call the native method *getDriverKeyNameOfDeviceOnPort* to get the driverkeyname which will look similar to this:

getDriverKeyNameOf-DeviceOnPort

```
{745A17A0-74D3-11D0-B6FE-00A0C90F57DA} \0001
{<device interface class>}\<number>
```

Table 12: DriverKeyName example

getFriendlyDevicName

4. With the driver key name and the native method *getFriendlyDeviceName* we receive a readable name for the driver key. This looks in the case we have just a normal USB device with its own driver as shown in Table 13 first line. In case we use a jUSB driver it looks like the second line in Table 13

```
Logitech WheelMouse (USB)
JUSB Driver --: MyPen as Testboard
```

Table 13: FriendlyDeviceName example

getUniqueDeviceID

5. To identify devices, hubs included, and recognise modification on the bus, a unique id is used. The native method *getUniqueDeviceID* will return a unique id for a device. This unique id consists of the current device address, the port it is connected to, the vendor id, the product id, the revision number, version number and some class and configuration issues.
6. At the end we either create a new *JUSB* object, when the friendly device name starts with "**JUSB Driver --:**" or otherwise a *NonJUSB* object

There are two more native methods to get the device and configuration descriptor. Those methods are called from the subclass *NonJUSB* and *JUSB* to initialize their appropriate device and configuration descriptor. The sub classed objects do not get their device and configuration descriptor. They ask the hub they are connected to, to retrieve their descriptors.

Why the address given at enumerating by the USBD driver is not unique enough in the Java USB API implementation?
Every device that is found during enumeration is put into the member devices in the *USB* class, which is an array of *DeviceImpl* objects elements. The index of this array corresponds to the device's addresses. The root hub which is a device too, has always the address zero. The other addresses for devices are given through the USBD driver by the operating system. To detect modification on the bus, we compare the currently found devices with the oldDevices member which represents the devices from a recent scan. If the devices on the same address are identical, no listeners are notified. But if the devices are different we have to notify the listener about a removal and an attachment of a new device.

Suppose we have the following situation on the USB bus. A root hub, an external hub attached on port 1 to the root hub and a device attached on port 1 to the external hub. The first time the member oldDevices of the *USB* class is null and after the first scan we have the following devices in the member devices of *USB* class as shown in Figure 8: scanBus 0.

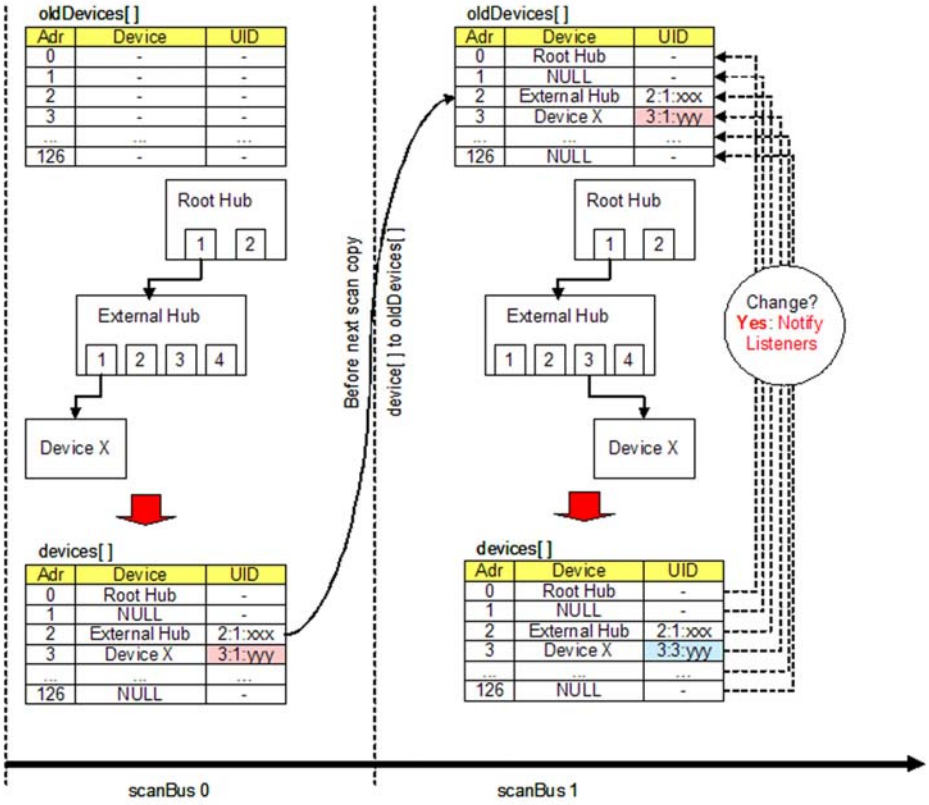

Figure 8: How to recognise modification on the bus structure

If we now detach device X from the external hub and attach it again to the external hub, but rather on port 3, we got the same enumeration in respect to the addresses of the devices (Figure 8: scanBus 1). If we compare the member oldDevices and devices to each other according to its address, we would find the same device at address 2 and therefore not notify the *USBListener*. In fact that would be incorrect since we had a modification on the bus. If every device has a unique device id, we are able to recognise modification on the

bus. Consider the scanBus 1 Figure 8 and look at the unique device id (UDI). We have still a device at address 2 but this time it is not the same with respect to the unique device id to the member oldDevices. The corollary is to inform the *USBListeners* that device X has been removed and attached again to the bus, but this time on port 3 of the external hub.

5.4.1 DeviceImpl Class Native Side Design

The native functions of the *DeviceImpl* class are implemented in jusbJNIdeviceImpl.cpp. In the next section we explain those native functions.

jint JNICALL Java_usb_windows_DeviceImpl_**openHandle**
jint JNICALL Java_usb_windows_DeviceImpl_**closeHandle**
jstring JNICALL Java_usb_windows_DeviceImpl_**getFriendlyDeviceName**
jint JNICALL Java_usb_windows_DeviceImpl_**getAttachedDeviceType**
jint JNICALL Java_usb_windows_DeviceImpl_**getNumPorts**
jstring JNICAL Java_usb_windows_DeviceImpl_**getDriverKeyNameOfDeviceOnPort**
jstring JNICALL Java_usb_windows_DeviceImpl_**getExternalHubName**
jbyteArray JNICALL Java_usb_windows_DeviceImpl_**getDeviceDescriptor**
jbyteArray JNICALL Java_usb_windows_DeviceImpl_**getConfigurationDescriptor**
jstring JNICALL Java_usb_windows_DeviceImpl_**getUniqueDeviceID**

Table 14: JNIEXPORT function for deviceImpl class

5.4.1.1 openHandle

A handle for a device we get with the known device path and the Windows API function *CreateFile*.
This native function returns either a INVALID_HANDLE_VALUE by failure or a device handle by success. The INVALID_HANDLE_VALUE is defined in the error.h file from the Microsoft SDK [24].

5.4.1.2 closeHandle

It closes open handles. The *CloseHandle* WinAPI function takes as argument a handle and closes it. We have to take care about open handles, because some open handles that are never closed can affect that the device may not be opened again through another application or the same application. The best way is to close open handle as soon as we got the appropriate information or we do not need any access to the device.

5.4.1.3 getFriendlyDeviceName

The friendly device name is closely related to the *getDriverKeyNameOfDevice-OnPort* function (see 5.4.1.6). Every device has one ore more entries in the registry, where parameters and device specific matters are stored. Because USB is hot pluggable we need a way to gather information about the device that is attached to the USB. The Bus driver recognises when a device is being attached and requests the device for the device descriptor. Out of this information the vendor id and the product id is extracted. The operating system checks if that information fits to an entry in the registry. If there is no concordance, the operating system checks all the INF-files for a match. As a last resort, the hardware assistant will ask the user to provide the driver information on a disk.
The *getFriendlyDeviceName* function looks up the *DeviceDesc* entry in the registry, which contains a human readable and understandable name for a given driver. The driver name look as follows {36FC9E60-C465-11CF-8056-444553540000}\0030, where the whole part between the brackets {…} suits to an interface class for a device and next to the slash is a number that identifies exactly one instance of the device. *getFriendlyDeviceName* is the JNI export function and the major task is done by the *DriverNameToDeviceDesc* function (Figure 9).

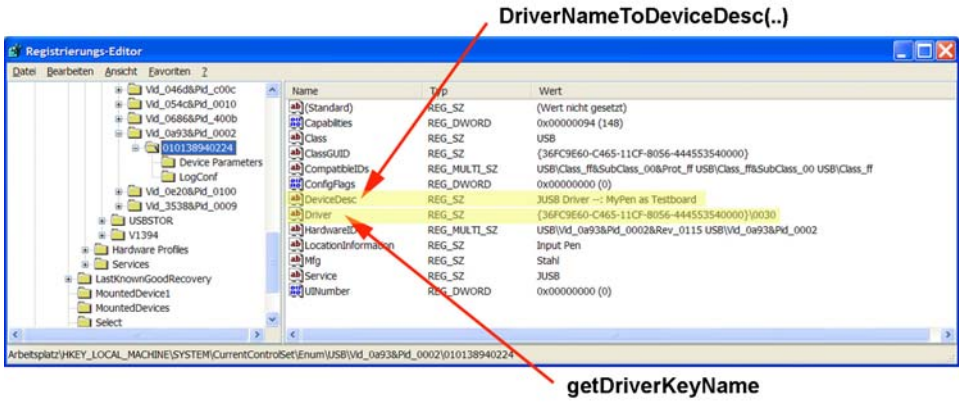

Figure 9 :Registry entry of driver and device description

With the aid of that friendly driver name, the decision is possible whether a USB device belongs to the *JUSB* class or to the *NonJUSB* classes. When the device description starts with “JUSB Driver --:” a *JUSB* object will be created.

The *DriverNameToDeviceDesc* function is in the file devnode.cpp which comes with the usbview example from the DDK. *CM_Get_DevNode_Registry_Property* is available for use in Windows 2000 and Windows XP. It may be altered or unavailable in subsequent versions. Applications should use the *SetupDiGetDeviceRegistryProperty* function. The *doDriverNameToDeviceDesc* in the helperFunction.cpp file uses those *SetupDiXxx* function but does not work properly together with the Java Native Interface. The modification and correction of this function is put to future work (see at conclusions).

5.4.1.4 getAttachedDeviceType

While we enumerate all devices through the root hub and external hub, we need to know what kind of device is attached to each of the hub ports. This function returns some symbolic constants as NO_DEVICE_CONNECTED, EXTERNAL_HUB or a USB_DEVICE is connected to the asked port. These constants are defined on the Java side, in the *DeviceImpl* class. To gain the attached type information of a hubs port, we use the undocumented IOCTL_USB_GET_NODE_CONNECTION_INFORMATION in a *DeviceIoControl* call (This function is shown in the usbview example but more precise and clearly arranged in an example of Intel by John Hyde [5].

The structure sent to and returned from the hub driver provides the following information.

```
typedef struct _NODE_CONNECTION_INFORMATION{
1           ULONG ConnectionIndex;
            DEVICE_DESCRIPTOR DeviceDescriptor;
            UCHAR CurrentConfigurationValue;
            BOOLEAN LowSpeed;
2           BOOLEAN DeviceIsHub;
            UCHAR DeviceAddress[2];
            UCHAR NumOfOpenPipes[4];
3           UCHAR ConnectionStatus[4];
            USB_PIPE_INFO PipeInfo[32];
} NODE_CONNECTION_INFORMATION,
PNODE_CONNECTION_INFORMATION;
```

Table 15: Node connection information of a hub

1. Specifies the port we look at
2. Is true when the device connected to this hub on port ConnectionIndex is an external hub. False denote that it is a USB device.

3. ConnectionStatus contains some info about the connection itself. This value can have one of the following values:

    - DeviceConnected
    - NoDeviceConnected
    - DeviceGeneralFailure
    - DeviceCauseOverCurrent
    - DeviceNotEnoughPower
    - DeviceNotEnoughBandwith
    - DeviceHubNestedToDeeply (not defined in Windows 2000)
    - DeviceInLegacyHub (not defined in Windows 2000)
    - DeviceFailedEnumeration

5.4.1.5 getNumPorts

If a hub is found we need to know how many ports it supports. *GetNumPorts* sends an IOCTL_USB_GET_NODE_INFORMATION to the hub driver and fills in the following structure Table 16:

```
1   typedef struct _NODE_INFORMATION{
            USB_HUB_NODE NodeType;
            HUB_DESCRIPTOR HubDescriptor;
            BOOLEAN HubIsBusPowered;
    } NODE_INFORMATION, *PNODE_INFORMATION;
```

Table 16: Node information of a hub included the hub descriptor

1. The HUB_DESCRIPTOR structure (Table 17) contains among other things the bNumberOfPorts member which we were looking for.

```
typedef struct _HUB_DESCRIPTOR{
        UCHAR bDescriptorLength;
        UCHAR bDescriptorType;
        UCHAR bNumberOfPorts;
        UCHAR wHubCharacteristics[2];
        UCHAR bPowerOnToPowerGood;
        UCHAR bHubCbontrolCurrent;
        UCHAR bRemoveAndPowerMask[64];
} HUB_DESCRIPTOR, *PHUB_DESCRIPTOR;
```

Table 17: Hub Descriptor structure and its members

The following paper was very helpful to get information how to query a hub with IOCTL codes [5]. The annoying thing is that most of this hub IOCTL codes are not documented but used in a lot of examples how to get access to a hub. This involves a lot of reading of source code, but makes it hard to vary the code sample, because we do not get out of the example, how the IOCTL code handles its input and output buffer nor the structure members.

5.4.1.6 getDriverKeyNameOfDeviceOnPort

GetDriverKeyNameOfDeviceOnPort returns the driver name from the registry for that USB device (see 5.4.1.3 and Figure 9

Figure 9 :Registry entry of driver and device description

). With the aid of the IOCTL_USB_-GET_NODE_CONNECTION_DRIVERKEY-_NAME and the USB_NODE_-CONNECTION_DRIVERKEY_NAME structure can the hub provide the driver name of the device that is attached on a given port.

5.4.1.7 getExternalHubName

GetExternalHubName returns a readable name for the hub device. The information is received while sending an IOCTL_USB_GET_NODE_CONNECTION_-NAME to the hub driver with the *DeviceIoControl* WinAPI function. The buffer returned from *DeviceIoControl* contains the desired external hub name.

5.4.1.8 getDeviceDescriptor

This function enables to retrieve the device descriptor of the USB device attached at a given downstream port of the hub. The good thing is we do not need to know the device path of the USB device to get the device descriptor, we only need to request the hub which does gathering the desired information. This is the reason we do not need a driver to enumerate the devices, but still able to access the functionality the USB device supports. At least we learn what device is connected to. The *getDeviceDescriptor* function uses the hub specific IOCTL code, IOCTL_USB_GET_NODE_CONNECTION_INFORMATION, and the port number to succeed the request.

5.4.1.9 getConfigurationDescriptor

The *getConfigurationDescriptor* function returns the configuration descriptor from the device attached to a given port number of the hub. It uses the IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION IOCTL code to obtain the complete configuration descriptor included all interface and endpoint descriptors.

5.4.1.10 getUniqueDeviceID

The unique device id is a string which consists of some attributes from the device descriptor and port information. The function is implemented similar to *getDeviceDescriptor* described in 5.4.1.8. The composition of the unique id is explained in Table 18.

Unique id composition:
USB/Adr_**AAA**&Port_**BBB**&Vid_**CCCC**&Pid_**DDDD**&Rev_**EEEE**&Ver_**FFFF**&DevClass_**GG**&DevSubClass_**HH**&NumC_**JJ** **AAA:** The device address assigned by the operating system (1…126) **BBB:** Port number where the device is attached to (usually 1…4) **CCCC:** Vendor id from the device descriptor **DDDD:** Product id from the device descriptor The following members of the unique id specify more precise the device and therefore the id should really be unique. **EEEE:** The revision number **FFFF:** The version number of the device **GG:** The device class it belongs to **HH:** The subclass the device belongs to **JJ:** The number of configurations

Table 18: Unique id

5.5 JUSB Class

The *JUSB* class contains all USB devices that are running with the jUSB driver. All native method will need the device path of the device to provide access to the driver. The device path is searched by means of the VID and PID which is passed as argument to the native method *getDevicePath* to retrieve the Windows device path of that device.

The final implementation of the jUSB driver should support all methods of the *DeviceSPI* class listed in Table 19. The highlighted methods in Table 19 are implemented. The method *readControl* is partly implemented. If we call a *readContol* request to the device it will answer the request or throw an exception depending on the setup packet we sent (see 5.5.1.2).

DeviceSPI Methods

```
public byte [] getConfigBuf (int n) throws IOException;
public void setConfiguration (int n) throws IOException;
public byte [] readControl (byte type, byte request, short value,
                    short index, short length);
public void writeControl (byte type, byte request, short value,
                    short index, byte buf []);
public byte [] readBulk (int ep, int length);
public void writeBulk (int ep, byte buf []);
public int clearHalt (byte ep);
public byte [] readIntr (int ep, int len);
public void writeIntr (int ep, byte buf []);
public String getClaimer (int ifnum);
public void claimInterface (int ifnum);
public void setInterface (int ifnum, int alt);
public void releaseInterface (int ifnum);
public Device getChild (int port);
```

Table 19: DeviceSPI methods

At the moment there is only interrupt transfer and a part of the control transfer available in the *JUSB* class. All the other transfer types (bulk and isochronous) are not implemented yet. For future work the bulk transfer can analogous be built to the interrupt transfer. Possible steps to implement bulk transfer in the jUSB API for Windows:

1. Define a native method such as *doBulkTransfer* in the *JUSB* class which extends the signature of *readBulk* with the argument device path.
2. Create the new JNI header file with javah.
3. Implement the *doBulkTransfer* JNI function in the jUSB DLL.
4. Define a new IOCTL code for *doBulkTransfer*.
5. Implement the IOCTL functionality in the jUSB driver.

This effort needs knowledge in driver writing.

5.5.1 JUSB Class Native Side Design

The native functions of the JUSB class are implemented in jusbJNIjusb.cpp. The following function in Table 20 will be explained in the next subchapters.

```
JNIEXPORT jstring JNICALL Java_usb_windows_JUSB_getDevicePath
JNIEXPORT jbyteArray JNICALL Java_usb_windows_JUSB_JUSBReadControl
JNIEXPORT jbyteArray JNICALL Java_usb_windows_JUSB_getConfigurationBuffer
JNIEXPORT jbyteArray JNICALL Java_usb_windows_JUSB_doInterruptTransfer
```

Table 20: JNIEXPORT functions for JUSB class

5.5.1.1 getDevicePath

The *getDevicePath* function takes as input parameter a string containing product id (PID) and vendor id (VID) of the device. It calls the *getDevicePath* (C/C++) function which returns the ith device path of a given device interface. The device interface we call for is GUID_DEFINTERFACE_JUSB_DEVICES which is defined in guids.h file and was created with the help of guidgen (Appendix B contains more information about GUID and the guidgen program).

guidgen

```
\??\USB#Vid_<VID>&Pid_<PID>#<Instance-Num>#{<Device Interface Class>}

<VID>: The vendor id of the USB device
<PID>:The product id of the USB device
<Instance-Num>: An automatic generated number by the operating system
<Device Interface Class>:        The GUID of a device interface class
        (e.g. as defined in guids.h)
```

Table 21: Device path of an USB device in Windows 2000/XP

Every USB device using the jUSB driver belongs to this device interface class (to retrieve more information about device interface classes refer to Appendix C).

The device path of each USB device in the Windows operating system 2000 and XP looks as follows shown in Table 21.

When retrieved the device path of the ith USB device, we compare the VID and PID to the VID and PID of the searched USB device. If the comparison corresponds to the VID and PID then a device has been found and we return its device path. This implies if we have to identical devices, we return only the first one. To distinguish between two identical devices is subject of future work. Table 22 presents a fragmentation of the *getDevicePath* JNI function.

```
JNIEXPORT jstring JNICALL Java_usb_windows_JUSB_getDevicePath
(JNIEnv *env, jobject obj, jstring pidAndVid){
…
  PCHAR deviceIdentity = (PCHAR)env->GetStringUTFChars(pidAndVid, 0);
…
  while(!found){
    devPath =
          getDevicePath((LPGUID)&GUID_DEFINTERFACE_JUSB_DEVICES, i);

    if(devPath != 0){
      //try to find the substring deviceIdentity in the devPath
      find = strstr(devPath,deviceIdentity);
      // find won't be NULL if we found such a string
      if(find != NULL) found = TRUE;  // we found a devicePath
       i++;           // look for the next device
    }
    else found = TRUE; // we did not find a matching, but quit the while loop
  }
…
  env->ReleaseStringUTFChars(pidAndVid, deviceIdentity);
  return devicePath;
}
```

Table 22: GetDevicePath JNI function

5.5.1.2 JUSBReadControl

First about the name of this function. Why JUSBReadControl and not just ReadControl as is used in the Java DeviceSPI class? The reason is to avoid a mangled function naming by the Visual C++ compiler (see Appendix E for more information). With naming that function as it is called now, the compiler did give the right export name as we defined it.

Setup Packet

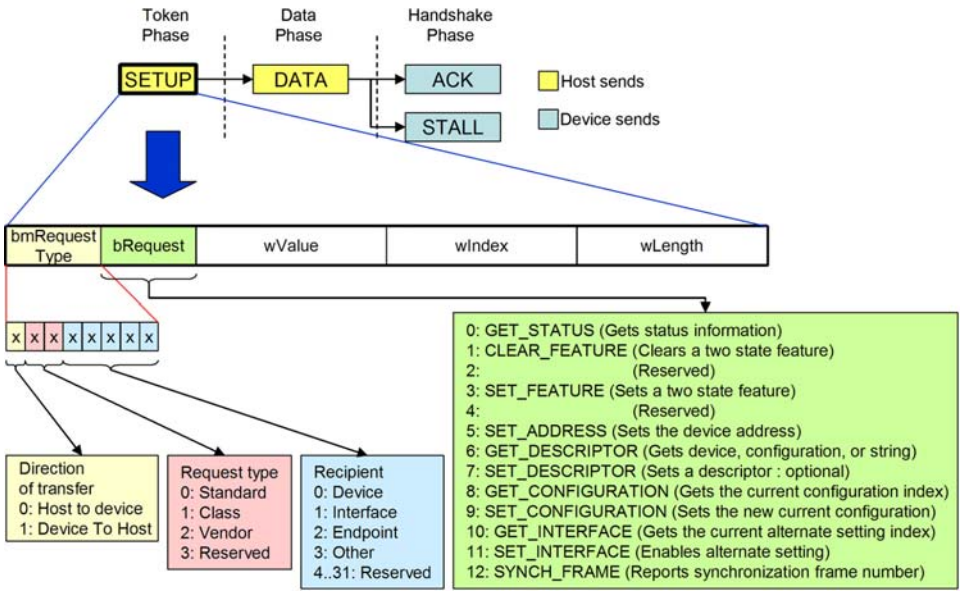

Figure 10 : Control transfer process with its setup packet

The control transfer includes a setup stage, which can be followed by an optional data stage in which additional data moves to or from the device, and a status stage, in which the device either response with an ACK packet or a STALL packet or does not response at all (Figure 10). The content of a setup packet contains 8 bytes and its members are shown in Figure 10 (corresponds to USB specification [27] chapter 9.3 and 9.4 which contains information about USB device request and standard device requests).

The standard USB 2.0 specification and the usb.core *ControlMessage* class process the control transfer as follows:
*All USB devices respond to request from the host on the device's Default Pipe. These requests are made using control transfers which contain all parameters in a **Setup packet** of exactly eight bytes.*

The Windows implementation of control transfer is far away from the USB standard. The setup packet for control transfer is separated depending on the request code (Request type in Figure 10). This fact is stated in the DDK by Microsoft as follows:
*All USB devices support endpoint zero for standard control requests. Devices can support additional endpoints for custom control requests .For endpoints **other than endpoint zero**, drivers issue the URB_FUNCTION_CONTROL_-TRANS-FER URB request. The UrbControlTransfer.SetupPacket member of the URB specifies the initial setup packet for the control request. See the USB specification for the place of this packet in the protocol.*

In other words while having a request type (as defined in Figure 10) of class or vendor, we are able to use the setup packet as it is used in usb.core *ControlMessage* class. In the case the value of request type is set to standard then we need to unpack the setup packet and according to the bRequest (second byte of the setup packet) use one of the following DDK macro (listed in Table 23) in the jUSB driver to achieve the request.

USB Feature Requests : (CLEAR_FEATURE, SET_FEATURE)
USB devices support feature requests to enable or disable certain Boolean device settings. Drivers use the UsbBuildFeatureRequest support routine to build the URB feature request.

USB Status Requests : (GET_STATUS)
Devices support status requests to get or set the USB-defined status bits of a device, endpoint, or interface. Drivers use the UsbBuildGetStatusRequest to build the URB status request.

Get or Set the Configuration : (GET_CONFIGURATION, SET_CONFIGURATION)
Use UsbBuildGetDescriptorRequest. Drivers use the URB_FUNCTION_GET_CONFIGURATION URB to request the current configuration. The driver passes a one-byte buffer in UrbControlGetConfiguration.TransferBuffer, which the bus driver fills in with the current configuration number.

Get USB Descriptors : (GET_DESCRIPTOR)
The device descriptor contains information about a USB device as a whole. To obtain the device descriptor, use UsbBuildGetDescriptorRequest to build the USB request block (URB) for the request.

Get or Set Interfaces: (GET_INTERFACE, SET_INTERFACE)
To select an alternate setting for an interface, the driver submits an URB_FUNCTION_SELECT_INTERFACE URB. The driver can use the UsbBuildSelectInterfaceRequest routine to format this URB. The caller supplies the handle for the current configuration, the interface members, and the new alternate settings. Drivers use the URB_FUNCTION_GET_CONFIGURATION URB to request the current setting of an interface. The UrbControlGetInterface.Interface member of the URB specifies the interface number to query. The driver passes a one-byte buffer in UrbControlGetInterface.TransferBuffer, which the bus driver fills in with the current alternate setting.

USB Class and Vendor Requests

To submit USB class control requests and vendor endpoint zero control requests, drivers use one of the URB_FUNCTION_CLASS_XXX or URB_FUNCTION_VENDOR_XXX requests. Drivers can use the UsbBuildVendorRequest routine to format the URB.

Table 23: Control request for endpoint zero in Windows driver stack

That fact described above does not make the implementation of control transfer easy. We have two possibilities to execute a setup request.

1. Define one IOCTL code and send the full setup packet with the help of *DeviceIoControl* WinAPI function to the jUSB driver and let the driver do the work.
2. Unpack the setup packet in the jUSB DLL and define a lot of IOCTL codes that activate a specific request as described in Table 23.

We decided to implement the second approach. This allows error handling in the jUSB DLL which still runs in user mode and therefore does not end up in a blue screen if we missed a point.
The following IOCTL codes (in Table 24) are used to execute the control requests. How to use those IOCTL code is described in Appendix A.

bRequest	**IOCTL code**
GET_STATUS	IOCTL_JUSB_GET_STATUS
CLEAR_FEATURE	n.i.
SET_FEATURE	n.i.
SET_ADDRESS	n.i.
GET_DESCRIPTOR Device Descriptor: Configuration Descriptor: String Descriptor:	 IOCTL_JUSB_GET_DEVICE_DESCRIPTOR IOCTL_JUSB_GET_CONFIGURATION_DESCRIPTOR IOCTL_JUSB_GET_STRING_DESCRIPTOR
SET_DESCRIPTOR	n.i.
GET_CONFIGURATION	n.i.
SET_CONFIGURATION	n.i.
GET_INTERFACE	n.i.
SET_INTERFACE	n.i.
SYNCH_FRAME	n.i.

Table 24: Corresponding IOCTL code for control request (n.i.: not implemented yet)

5.5.1.3 getConfigurationBuffer

The *getConfigurationBuffer* function is implemented with a *DeviceIoControl* function call (IOCTL = IOCTL_JUSB_GET_CONFIGURATION_DESCRIPTOR) to the jUSB driver. Refer to Appendix A to get more information about this IOCTL code.

5.5.1.4 doInterruptTransfer

The *doInterruptTransfer* function is implemented with a *DeviceIoControl* function call (IOCTL = IOCTL_JUSB_INTERRUPT_TRANSFER) to the jUSB driver. Refer to Appendix A to get more information about this IOCTL code.

6 jUSB Driver

Driver writing and driver development is very complex. We refer to the book written by Walter Oney “Programming The Microsoft Windows Driver Model” [4] to get into driver development within the Windows operating system. The following sections highlight some aspects of the jUSB driver. We have to mention that the jUSB driver is built out of the bulkusb driver delivered with the DDK.

6.1 DeviceExtension

The structure DEVICE_EXTENSION contains information about the device’s state (its current configuration. The initialization should be done in the *AddDevice* routine. This routine will be called only once for each device, exactly when we attach the device to the host.
The members of DEVICE_EXTENSION and the management are free to invent, so that they satisfy our hardware, in our case we should be able to handle all request from and to the jUSB API.
There are some common members that can be found in most drivers (refer to part 1 in Table 25) and in the part 2 of Table 25 there are jUSB driver specific members.

```
   typedef struct _DEVICE_EXTENSION{
     /*      Part 1      */
1    PDEVICE_OBJECT DeviceObject;
2    PDEVICE_OBJECT LowerDeviceObject;
3   ..PDEVICE_OBJECT PhysicalDeviceObject;
4    UNICODE_STRING ifname;
5    IO_REMOVE_LOCK RemoveLock;
6    DEVSTATE devState;
     DEVSTATE previousDevState;
     DEVICE_POWER_STATE devicePower;
     SYSTEM_POWER_STATE systemPower;
     DEVICE_CAPABILITIES deviceCapabilities;
     /*      Part 2      */
7    USBD_CONFIGURATION_HANDLE CurrentConfigurationHandle;
    ..USBD_CONFIGURATION_HANDLE PreviousConfigurationHandle;
8    PUSB_DEVICE_DESCRIPTOR DeviceDescriptor;
9    PUSB_CONFIGURATION_DESCRIPTOR * ConfigurationDescriptors;
10   ULONG CurrentConfigurationIndex;
11   ULONG PreviousConfigurationIndex;
12   PUSBD_INTERFACE_INFORMATION * InterfaceList;
13   PCLAIMED_INTERFACE_INFO * InterfaceClaimedInfo;
14   PENDPOINT_CONTEXT * EndpointContext;
15   KSPIN_LOCK IoCountLock;

   } DEVICE_EXTENSION, *PDEVICE_EXTENSION
```

Table 25: Common members within a DEVICE_EXTENSION structure

1. It is useful to have the DeviceObject pointer.
2. The address of the device object immediately below this device object. This is used for passing IRP down the driver stack.
3. A few service routines require the address of the PDO instead of some higher device object in the same stack.
4. The member ifname records the interface name to that device. This will always be set to GUID_DEFINTERFACE_JUSB_DEVICES.
5. It is used to solve the synchronization problem, when it is safe to remove this device object by calling *IoDeleteDevice*.
6. We need to keep track of the current plug and play state and the current

power status state of our device. DEVSTATE is an enumeration that we declare elsewhere.

7. Records the current ConfigurationHandle. This will be used if the method *getConfiguration* is invoked by the jUSB API. This value must be updated as soon *getConfiguration(n)* gets called. If there is no nth Configuration, return an error and set the ConfigurationHandle to the old one.
8. Contains the current Device Descriptor for this USB device. So far the *setDeviceDescriptor* method in jUSB API is not implemented and therefore the device descriptor will remain the same (for detailed information see 6.2.1).
9. Keeps an array of all configuration descriptors for this device (more info see 6.2.2).
10. Holds the current Configuration index.
11. Holds the previous Configuration index.
12. An array that contains information about every interface of the current configuration in this device. The USBD_INTERFACE_INFORMATION structure itself contains information about all pipes that belong to that interface (more info see 6.2.3).
13. An array of CLAIMED_INTERFACE_INFO values to indicate if a specific interface is claimed or not and who is the current claimer (for more information see 6.2.4).
14. An array which keeps information about all the endpoints in the current configuration (for more information see 6.2.5).
15. see at 6.4

6.2 Important Members of DeviceExtension Structure

To keep device information and its state current of a device using the jUSB driver, we need some useful member in the DeviceExtension structure. The following section will present those members and its structure. All of those members are always initialized when a jUSB device is attached to the bus. We set always the first configuration of an USB device as default. Most USB devices have just one configuration. The reason to configure the device at initialization is to gain access to the device. Preciously we will handle IRP_MN_START_DEVICE that is a minor function of the IRP_MJ_PNP.
The function *DispatchPnP* in PlugPlay.c processes those IRPs. The function *HandleStartDevice* will call a sub function *ReadAndSelectDescriptors* which starts all setup settings for the the jUSB device.

6.2.1 DeviceDescriptor

The DeviceDescriptor is a pointer that points to a USB_DEVICE_DESCRIPTOR structure. This structure can be type casted to PCHAR for giving this value back to JNI. Device descriptor has always a size of 18 bytes.

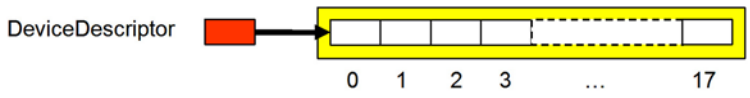

Figure 11: DeviceDescriptor memory allocation (yellow: allocated memory)

6.2.2 ConfigurationDescriptors

The ConfigurationDescriptors variable is an array of pointers to a USB_CONFIGURATION_DESCRIPTOR. It is initialized in the function *ConfigureDevice*. The following steps have to be done to correctly initialize this variable. The number of configuration we get from the DeviceDescriptor structure member bNumConfiguration. The first step is to allocate enough memory to keep all the pointers that point to a possible configuration of the device (Figure 12: position 1). In a second step we get all those configuration descriptors by repeating the following procedure:

1. Allocate enough memory to keep only the configuration descriptor. This can be done because the USB_CONFIGURATION_DESCRIPTOR

structure is predefined in usb100.h and therefore we can calculate its size.
After having received the configuration descriptor we can get the information about the total size of the whole configuration descriptors including all interface and endpoint descriptors through the member wTotalLength. (Figure 12: position 2)

2. Allocate memory in the exact size of wTotalLength and call the _URB_CONTROL_DESCRIPTOR_REQUEST again to get the complete configuration descriptor (Figure 12: position 3)

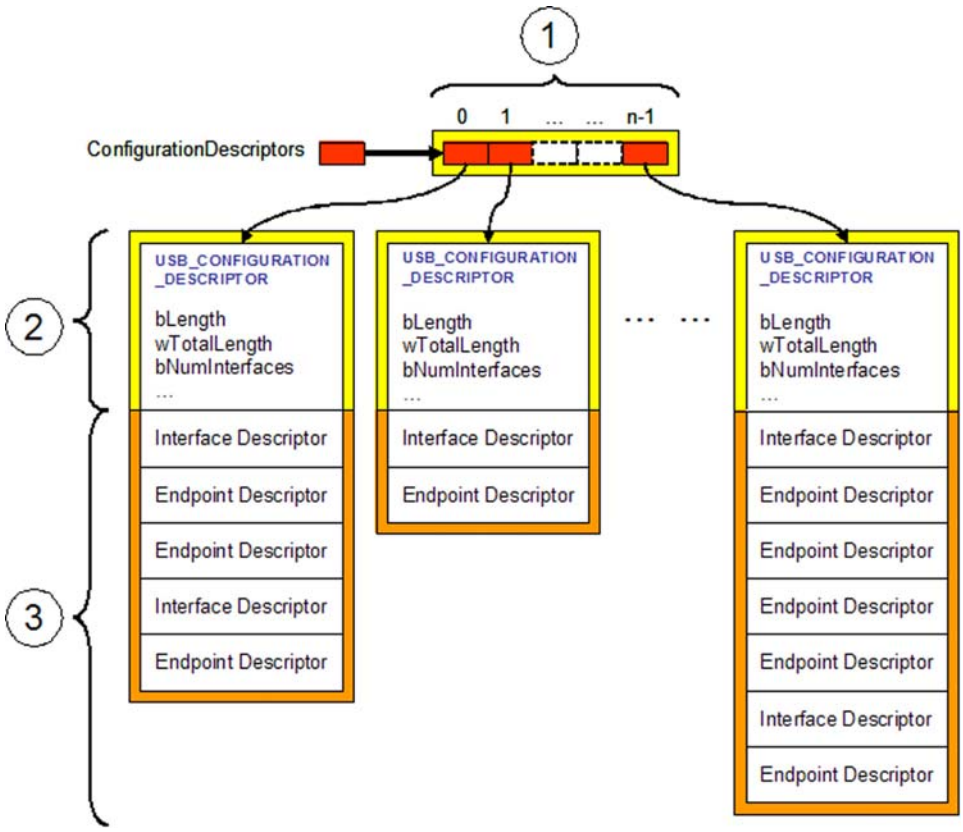

Figure 12: ConfigurationDescriptors memory allocation structure (yellow: allocated memory, orange: additional allocated memory)

6.2.3 InterfaceList

The InterfaceList variable is an array of pointers to a USBD_INTERFACE-_INFORMATION structure that is predefined in the DDK in usb100.h. The size of InterfaceList is exactly the number of currently available interfaces in the current configuration of the device. Every USBD_INTERFACE_INFORMATION structure contains information about the interface and about all pipes. The pipe information is kept in the USBD_PIPE_INFORMATION structure.

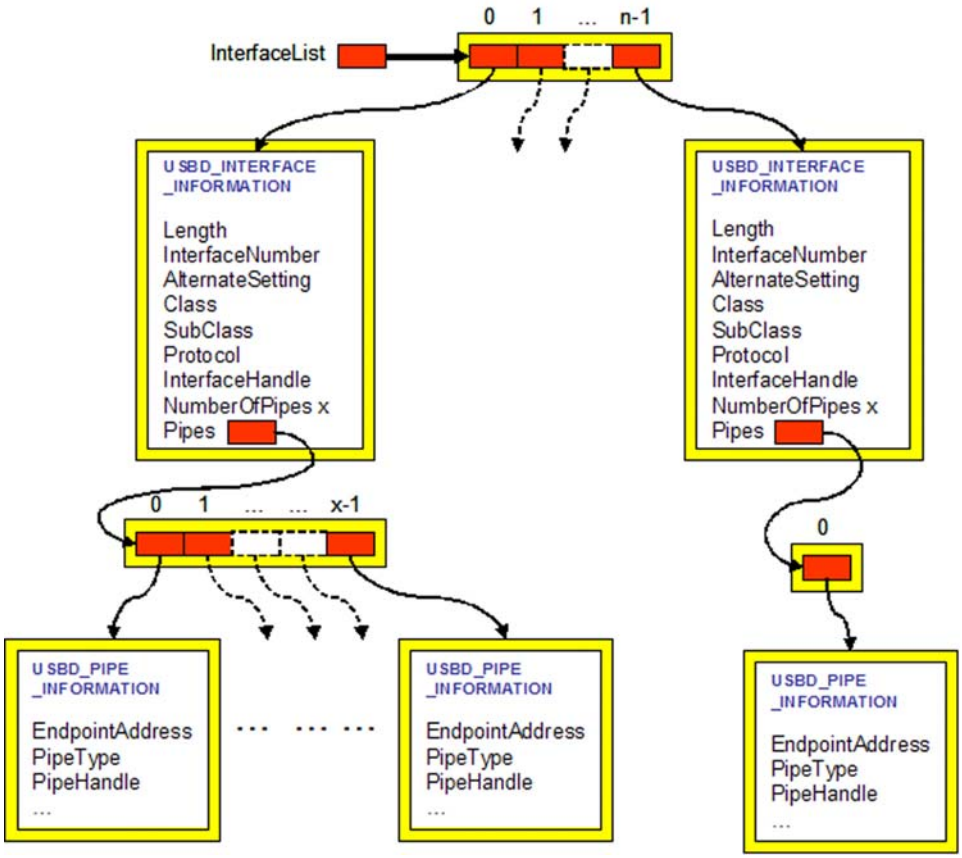

Figure 13: InterfaceList memory allocation structure (yellow: allocated memory)

6.2.4 InterfaceClaimedInfo

InterfaceClaimedInfo is an array of pointers to a CLAIMED_INTERFACE_INFO structure. That structure contains so far only one member claimed, that indicates if an interface has been claimed by a user or not. The size of the Interface-ClaimedInfo array is the same as the size of InterfaceList array.

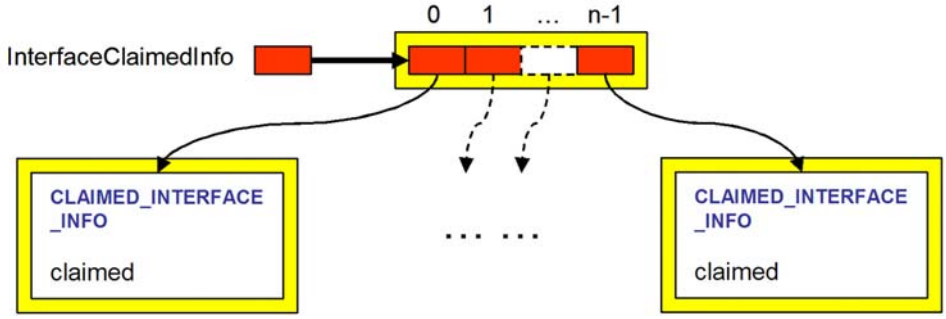

Figure 14: InterfaceClaimedInfo memory allocation structure (yellow: allocated memory)

6.2.5 EndpointContext

EndpointContext is an array of pointers which points to an ENDPOINT_CONTEXT structure. The size of this array is always 30. This means we can have a maximum of 30 endpoints, so called pipes, for an USB device. This is related to the USB 2.0 specification chapter 5.3.1.2 (Non Endpoint Zero Requirements) and chapter 8.3.2.2 (Endpoint Field).
The endpoint numbers correspond to the index of the EndpointContext array, except that we have to add one to the index of the array to get the pipe number.

For every endpoint the configuration of the device supports, we fill in such a ENDPOINT_CONTEXT structure at the exact position. All other entries of the EndpointContext array point to NULL.

Further we assume that every endpoint is unique to a configuration and its interfaces.
A short excursus to interfaces and endpoints. Related to the USB 2.0 specification, a configuration can have one or more configuration. Each configuration can have one or more interfaces and a maximum amount of 30 endpoints (the two endpoints for the default pipe are excluded). The endpoints have to be unique in the configuration, which means that they can not be shared through several interface in the same configuration.

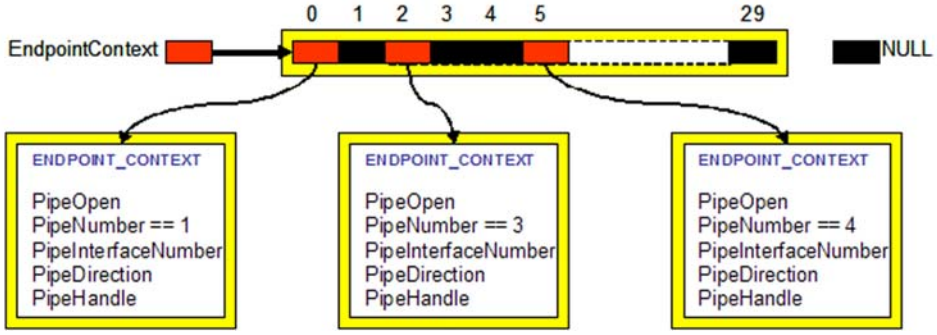

Figure 15:EndpointContext memory allocation structure (yellow: allocated memory)

6.3 Dispatch Routine

Before a driver can process I/O request, it has to define what kind of operation it supports. This section describes the meaning of the dispatch mechanism of the I/O Manager and how a driver activates some I/O function codes that it receives.

Every I/O operation of Windows 2000/XP is managed through packets. For every I/O request an associated I/O request packet (IRP) exists, that is created by the I/O Manager. The I/O Manager writes a function code in the MajorField of the IRP, which uniquely identifies the request. Furthermore the MajorField serves the I/O Manager to decide which dispatch routine should be loaded. In case a driver does not support a requested operation, the I/O Manager will return an error message to the caller. Dispatch routines have to be implemented by the developer. What kind of dispatch routine the driver supports and will process is in the developer decision.

6.4 Synchronization Techniques

Spin Lock Objects

To support synchronously access shared data in the symmetric multiprocessing world of Windows XP, the kernel lets us define any number of spin lock objects. To acquire a spin lock, code on the CPU executes an atomic operation that tests and then sets a memory variable in such a way no other CPU can access the variable until the operation completes. If the test shows that the lock was previously free, the program continues. If the test indicate that a lock was previously held, the program repeats the test-and-set in a tight loop: it “spins”. Eventually the owner releases the block by resetting the variable, whereupon one of the waiting CPUs’ test-and-set operation will report the lock as free.
The next figure shows the concept of using a spin lock:

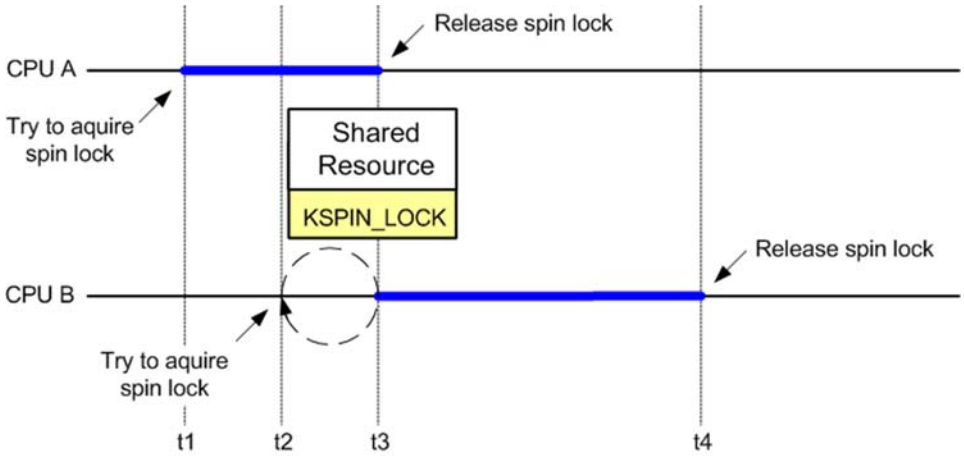

Figure 16: Using a spin lock to guard a shared resource

Consideration about SpinLock

There are some facts about spin locks we have to be aware while writing code. First of all, if a CPU already owns a spin lock and tries to obtain it a second time, the CPU will deadlock. No usage counter or owner identifier is associated with a spin lock; somebody either owns a lock or not.
In addition, acquiring a spin lock raises the IRQL to DISPATCH_LEVEL automatically and must therefore be in nonpaged memory.

Spin Lock Initialization

In the jUSB driver we find some variable identifiers that are of type KSPIN_LOCK in the driver's device extension. The type KSPIN_LOCK is defined in wdm.h as ULONG_PTR. We have for example an IOCountLock spin object in the device extension of the jUSB driver (see 6.1).
This object has to be initialized in the *AddDevice* routine for later use.

```
NTSTATUS AddDevice(…){
   …
   PDEVICE_EXTENSION deviceExtension = …;
   KeInitializeSpinLock(&deviceExtension->IOCountLock);
   …
}
```

Table 26: Initialization of a spin lock object

After the spin lock object has been initialized it can be used in any dispatch routine. The following example shows how to use the spin lock object.

Use of Spin Lock

```
LONG IoIncrement(…){
   KIRQL oldIrql;  // to keep the Kernel Interrupt Request Level
   PDEVICE_EXTENSION deviceExtension = …;

   KeAcquireSpinLock( &DeviceExtension->IOCountLock, &oldIrql);
   …
   …// code between those SpinLock routines is atomarly executed
   …// no other process which calls IoIncrement enter this section
   …// as long IOCountLock spin is not released by the current process.
   …
   KeReleaseSpinLock( &DeviceExtension->IOCountLock, oldIrql);
   …
}
```

Table 27: Use of a spin lock object

6.5 I/O Control Codes

To communicate with the driver without using *ReadFile* or *WriteFile* from the Windows API, we can use the supported *DeviceIoControl* function. This allows

user mode access to driver specific features. I/O Control codes (IOCTL) are depending on the developer. It is the developer's charge to manage and handle the IOCTL. In the jUSB driver we need as well IOCTL codes to modify or get some information about the driver states. The definition of all IOCTL the jUSB driver supports can be found in the ioctls.h file and Appendix A contains more detailed information about those IOCTL's. A fragment of this header file is presented in the following table.

IOCTL Definition

```
#ifndef CTL_CODE
#pragma message("CTL_CODE undefined. Include winioctl.h or wdm.h")
#endif

#define IOCTL_JUSB_GET_DEVICE_DESCRIPTOR CTLCODE( \
                        (FILE_DEVICE_UNKNOWN, \
                         0x8000, \
                         METHOD_BUFFERED, \
                         FILE_ANY_ACCESS)
```

Table 28: Definition of an IOCTL

The pragma message is just a help in case someone forget to include the header file winioctl.h that defines the CTL_CODE macro for user program. The "\" represents just a new line without any new line character!

The structure of an IOCTL code is a 32 bit value and it is defined as follows:

IOCTL Structure

31 - 16	15 - 14	13 - 2	1 - 0
DeviceType	RequiredAccess	ControlCode	TransferType

DeviceType	0x0000 to 0x7FFF reserved for Microsoft 0x8000 to 0xFFFF free to use
RequiredAccess	FILE_ANY_ACCESS FILE_READ_DATA FILE_WRITE_DATA FILE_READ_DATA | FILE_WRITE_DATA
ControlCode	0x000 to 0x7FF reserved for Microsoft 0x800 to 0xFFF free to use
TransferType	METHOD_BUFFERED METHOD_IN_DIRECT METHOD_OUT_DIRECT METHOD_NEITHER

Table 29: CTL_CODE macro parameters

Each user mode call to a *DeviceIoControl* WinAPI function causes the I/O Manager to create an IRP with the major function code IRP_MJ_DEVICE_CONTROL and to send that IRP to the driver dispatch routine at the top of the stack for the addressed device.

DispatchControl

A skeleton dispatch function for control code operation looks like this:

```
NTSTATUS DispatchControl( IN PDEVICE_OBJECT DeviceObject,
                                    IN PIRP Irp)
{
   ULONG            code;
   PVOID            ioBuffer;
   ULONG            inputBufferLength;
   ULONG            outputBufferLength;
   ULONG            info;
   NTSTATUS         ntStatus;
   PDEVICE_EXTENSION  deviceExtension;
   PIO_STACK_LOCATION irpStack;
   info = 0;
   irpStack = IoGetCurrentIrpStackLocation(Irp);
1  code = irpStack->Parameters.DeviceIoControl.IoControlCode;
   deviceExtension=
         (PDEVICE_EXTENSION)DeviceObject->DeviceExtension;
   ioBuffer = Irp->AssociatedIrp.SystemBuffer;
   inputBufferLength =
         irpStack->Parameters. DeviceIoControl.InputBufferLength;
   outputBufferLength =
         irpStack->Parameters.DeviceIoControl.OutputBufferLength;
   …
   switch(code) {
     case IOCTL_JUSB_GET_DEVICE_DESCRIPTOR:
2    …  // do something here
     break;
   case IOCTL_...:
     …  // do something here
     break;
   default :
      ntStatus = STATUS_INVALID_DEVICE_REQUEST;
   }

   Irp->IoStatus.Status = ntStatus;
   Irp->IoStatus.Information = info;
   IoCompleteRequest(Irp, IO_NO_INCREMENT);
   return ntStatus;
}
```

Table 30: Skeleton of DispatchControl

1. The next few statements extract the function code and buffer sizes from the parameters union in the I/O stack. We often need this value no matter which specific IOCTL we are processing.
2. Handles all the various IOCTL operation we support

6.5.1 IOCTL TransferType

METHOD_BUFFERED

With METHOD_BUFFERED, the I/O Manager creates a kernel-mode temp buffer which is big enough to hold the larger of the user-mode input and output buffers. When the dispatch routine gets control, the user mode input data is available in the temp buffer. Before completing the IRP, we need to fill the copy buffer with the output data that we want to send back to the application. The IoStatus.Information field in the IRP is equal to the numbers of output bytes written. Always check the length of the buffers, because we are the only one who knows how long the buffers should be. Finish processing the input data before overwriting the copy buffer with the output data.

METHODE_IN_DIRECT

METHODE_OUT_DIRECT

Both METHODE_IN_DIRECT and METHOD_OUT_DIRECT are handled the same way in the driver. METHODE_IN_DIRECT needs read access. METHOD_OUT_DIRECT needs read and write access. With both of these methods, the I/O Manager provides a kernel-mode temp buffer for the input data

and for the output data.

METHOD_NEITHER

METHOD_NEITHER is often used when no data transfer for a current IOCTL is used.

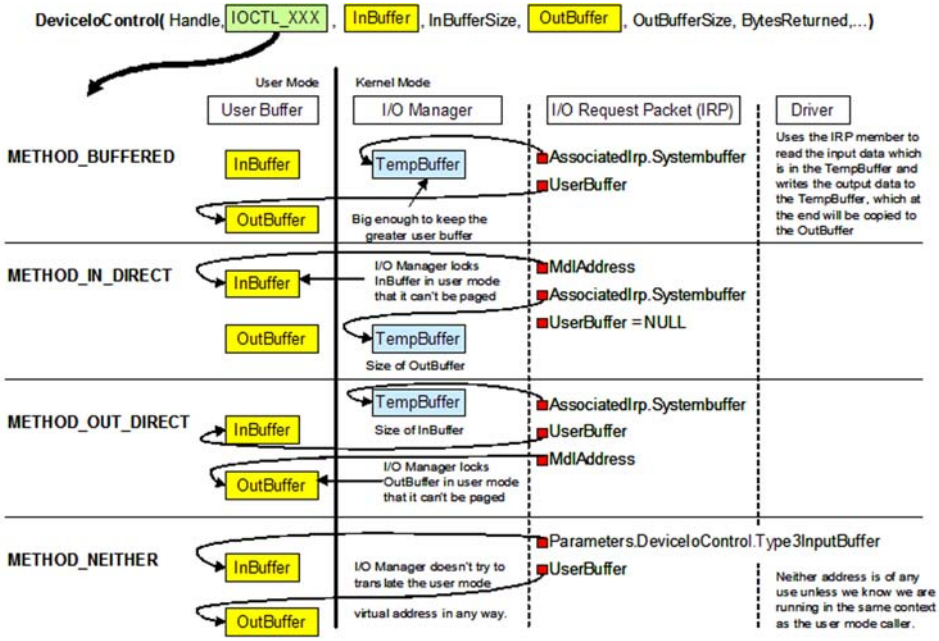

Figure 17: IOCTL transfer types and DeviceIoControl WinAPI functions

6.6 Control Transfer

The design of handling control transfer in user mode is described in chapter 3.5.1.2. The decision we made uses different IOCTL codes for the different kind of request types. In the jUSB driver we have to handle those IOCTL codes that will be sent by means of the *DeciceIoControl* WinAPI function to the jUSB driver (see chapter 6.5 and Figure 17 for more information about IOCTL codes).

Table 24 at chapter 5.5.1.2 list the IOCTL code we have to implement in the driver. The input and output parameter of all those IOCTL are explained in Appendix A. The DDK macro function to handle standard request are all listed in Table 23 at chapter 5.5.1.2.

6.7 Interrupt Transfer

Interrupt transfer is implemented with the help of an IOCTL code namely the IOCTL_JUSB_INTERRUPT_TRANSFER code. As input we have the endpoint address to which we want process an interrupt transfer. The address contains the pipe number and the direction of data flow. The transferFlag variable (line 8 in Table 31) is either USBD_TRANSFER_DIRECTION_IN for an IN endpoint or USBD_TRANSFER_DIRECTION_OUT for an out endpoint. Because we keep all information up to date in the deviceExtension, we know all about each pipe the device supports. With the help of the member EndpointContext (chapter 6.2.5) we are able to get the desired pipe handle to process the interrupt transfer. Of course, if the input request tries to execute an interrupt transfer to a pipe that either does not exist nor the direction nor the type corresponds to the endpoint descriptor, than an invalid status has to be returned.

If all input checks are successful the UsbBuildInterruptOrBulkTransferRequest macro from the DDK can be used to build an USB interrupt request. The request is stored in the urb variable (line 2 in Table 31). This USB request Block (URB) will be sent to the lower driver in this case to the USB driver (usbd.sys) which does the duty.

We do not have to be concerned about the intervals of executing this request in the jUSB driver. This is the task of the person which uses the jUSB API. The interval time is known from the endpoint descriptor and the Java programmer has to take care to execute periodically the *readIntr* method.

```
1   UsbBuildInterruptOrBulkTransferRequest(
2           &urb,
3           sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER),
4           deviceExtension->EndpointContext[pipeNum]->PipeHandle,
5           ioBuffer,
6           NULL,
7           inputBufferLength,
8           transferFlag | USBD_SHORT_TRANSFER_OK,
9           NULL
    );

10  ntStatus = SendAwaitUrb(DeviceObject,&urb,&ulLength);
```

Table 31: UsbBuildInterruptOrBulkTransferRequest macro

6.8 BulkTransfer

Bulk transfer is not implemented yet. We think it should be possible to implement bulk transfer corresponding to the interrupt transfer. A design using IOCTL codes allows us to send input parameters such as endpoint address to the driver to do bulk transfer on the desired pipe. It should be considered that for bulk transfer we may better define an IOCTL code with transfer type METHOD_OUT_DIRECT or METHOD_IN_DIRECT to get rid of copying a temp buffer to the kernel mode.

If we figure out how to use *ReadFile* and *WriteFile* WinAPI function to perform a bulk transfer to a given pipe, we better use these functions. At the moment we did not find a solution for this idea.

7 User Installation

This section describes the installation of the Java USB API for Windows for end users.

7.1 Resources

JavaUSB.ZIP
The Java USB API includes the binaries of the jUSB DLL and the jUSB driver. JavaUSB.ZIP can be downloaded from http://www.steelbrothers.ch/jusb/. Be sure to download the Java USB resources for end users which is called JavaUSB.ZIP

Netbeans IDE
Is a full-featured integrated environment for Java Developers [21]. We used Netbeans to develp the Java side of the Java USB API. Netbeans is freely available on http://www.netbeans.org. Of course any other environment with a Java compiler can be used to extend or run the Java USB API.

7.2 Installation of the jUSB Driver and jUSB DLL

jUSB DLL
Copy the jusb.dll from the folder *InstallationFiles**JusbDll* to the *system32* folder of your Windows directory.

Compile the usb.windows package in Netbeans. Attach a USB device to the USB and run *RunUSBControllerTest*.

jUSB Driver
First, we have to register the JUSB driver in the Windows registry. Therefore, we need to double click on the jusb.reg file which is located in the *InstallationFiles**JusbDriver* folder. A Window will pop up asking if we are sure to add the information of jusb.reg to the registry. After clicking on yes, a confirmation will be displayed and the information will be added to the registry. This process of registering the jUSB driver has to be executed only once.

After the jUSB driver has been registered, we have to copy the jusb.sys file (the driver) to the *system32**drivers* folder of the Windows directory.

jUSB driver test
The following procedure can be used to test the driver:

1. Connect an USB mouse to the USB port.
2. Run RunUSBControllerTest and note down the VID and PID which is displayed out, next to the uniqueID.
3. Download Debug View v4.21 from http:/www.sysinternals.com/ntw2k-/utilities.shtm and start the debugger. This is only for controlling purposes.
4. Change the registry entries as described in Appendix D.
5. Disconnect the USB mouse and connect it again.
6. The debugger should display some information. This information are generated by the jUSB driver

8 Developers Installation

The purpose of an open source project is that other developers modify and enhance the existing framework. This section should provide a help to install and setup the environment to rapidly start with developing. The most annoying thing in developers work is spending a lot of time to install the programming environment. We try to make this step as easy as possible.

8.1 Resources

To develop on the current Java USB API project some resources are required:

JavaUSBComplete

JavaUSBComplete.ZIP
The Java USB API sources including the native libraries and the jUSB driver. This source can be downloaded on http://www.steelbrothers.ch/jusb/. Be sure you download the Java USB complete resources for developers which is called JavaUSBComplete.ZIP

Visual C++

Microsoft Visual C++ or the new version Microsoft Visual .NET
Microsoft Visual C++ or the new version Microsoft Visual .NET programming environment has to be used. We developed a big part of the Java USB API for Windows on Microsoft Visual C++ Version 6.0.

Platform SDK 2003

Microsoft Software SDK 2003 or later
The Microsoft software developer kit is needed to support the core libraries. The latest SDK can be downloaded from the MSDN developers site (http://www.microsoft.com/msdownload/platformsdk/sdkupdate/). Only the Core SDK is needed (168 MB). The required size for the complete installation is 480MB.

DDK XP

Microsoft Driver Development Kit (DDK) XP or 2003
The DDK is used to build the JUSB driver. Unfortunately, the DDK is not made available for download by Microsoft. An order is demanded, but they will send the DDK for free apart of the expenses for delivery. The DDK page can be found at the following url: http://www.microsoft.com/whdc/ddk/.

External Debugger

DbgView
We used the debug view program DbgView which can be freely downloaded from http://www.sysinternals.com. This program intercepts calls made to *DbgPrint* and *KdPrint* by device drivers and *OutputDebugString* made by Win32 programs. It allows for viewing and recording of debug session output on your local machine or across the Internet without an active debugger [25]. This tool has been a big benefit for developing either user or kernel mode programming.

Netbeans

Netbeans IDE
Is a full-featured integrated environment for Java Developers [21]. We used Netbeans to develp the Java side of the Java USB API. Netbeans is freely available from http://www.netbeans.org. Of course any other environment with a Java compiler can be used to extend or run the Java USB API.

JDK 1.4.1

Java Runtime Environment J2SE 1.4.1
The premier solution for rapidly developing and deploying mission-critical, enterprise applications, J2SE provides the essential compiler, tools, runtimes, and APIs for writing, deploying, and running applets and applications in the Java programming language [9]. We compiled the Java USB API with the JDK version 1.4.1 [8].

8.2 Setting the Environment Variables

Different kinds of environment variables have to be adjusted. Some of them are

specific for the Netbeans IDE and others for settings in the Visual C++ project. If those environment variables are not set correctly, parts of the software environment will not work correctly. Table 32 shows where the environment variables can be set.

Environment Variables

Windows 2000

1. Start→Settings→Control Panel→System
2. choose category: Advanced
3. choose: Environment Variables…
4. Edit, delete or make a new system variable

Windows XP

1. Start→Control Panel→System
2. choose category: Advanced
3. choose: Environment Variables
4. Edit, delete or create a new system variable

Table 32: Setting the environment variables

The following environment variables have to be set. The value given in this context can be seen as an example. This belongs to the settings we specified in our computer and they will vary on other system. The example should give a hint of what the path may look like.

CLASSPATH

CLASSPATH

The class path tells SDK tools and applications where to find third-party and user-defined classes - that is, classes that are not Java extensions or part of the Java platform. The class path needs to find any classes you have compiled with the javac compiler - its default is the current directory to conveniently enable those classes to be found. We may need to extend the CLASSPATH variable, because other settings are defined too. Extension are made by a semicolon '**;**'.

Variable	Value
CLASSPATH	F:\Studium\JavaUSB\JavaSources

Table 33: CLASSPATH setting

Path

Path

While trying to compile a Java source with javac or creating a JNI header file with javah and receiving the following error message: *'javac' is not recognized as an internal or external command* or *'javah' not found* then the Path variable need to be set to the path where the binaries of those commands are. This variable needs in all probability to be extended.

Variable	Value
Path	C:\Programme\s1studio_jdk\j2sdk1.4.1_02\bin;

Table 34: Path setting

JAVAHOME

JAVAHOME

This variable points to the root directory of the Java runtime environment. This setting enables the Visual C++ programming environment to find the Java Native Interface header files, such as jni.h.

Variable	Value
JAVAHOME	C:\Programme\s1studio_jdk\j2sdk1.4.1_02

Table 35: JAVAHOME setting

JUSBPATH

JUSBPATH

This variable points to the JavaUSB directory. This setting is used when a developer does not have the DDK [6] installed but still wants to try compiling and

JUSBPATH

modifying the JUSB DLL. This provides the Visual C++ project settings where to find the additional header file which would have been on the DDK from Microsoft. These DDK header file we use can be found in the *JusbDll\external-header-file\ddk* folder.

Variable	Value
JUSBPATH	F:\Studium\JavaUSB

Table 36: JUSBPATH setting

DDKPATH

DDKPATH
Points to the root directory of the current installed DDK.

Variable	Value
DDKPATH	C:\WINDDK\2600.1106

Table 37: DDKPATH setting

8.3 Unzip the JavaUSBComplete.Zip File

To work with the JUSB DLL the JavaUSBComplete.Zip file needs to be extracted. For further examples we assume that the JavaUSBComplete.Zip file is unzipped in a folder, named *JavaUSB*.

Make sure not to copy the *JavaUSB* folder within a folder path containing spaces in the name for example *"C:\Documents and Setting\..."*. This will lead to error in the build environment for the driver (see 8.6.1.1).

After successfully unzipping the JavaUSBComplete.Zip file, four folders should be seen in the JavaUSB folder:

- *Installation Files*: Contains all files for end users that want to use the Java USB API in application.
- *JavaSources* : Complete Java source code of the Java USB API. Chapter 8.4 gives an overview of the folder contents.
- *JusbDll*: All C/C++ source files which are used to create the JUSB DLL and the JNI implementation of the Java USB API. Chapter 8.5 will explain the contents of this folder in detail.
- *JusbDriver*: Driver relevant resources to build the driver. Chapter 8.6 presents al the files belonging to the JUSB driver and how the driver can be built.

8.4 Java USB API for Windows

All the needed files to implement or extend the Java USB API can be found in the usb.windows package which is in the *\JavaSources\usb\windows* folder. As development environment any text editor can be used or Netbeans IDE as we did. Make sure that you start compiling the classes from the root directory of the package. For example if we want to compile the Windows class, we need to be in the *JavaSources* directory. Run the command line program and enter the following command: javac usb.windows.Windows.java.

8.4.1 Creating the Java Native Headers

javah

To implement the native methods which are specified in the Java classes, we need to create the appropriate C-header files. This is done in using the command **javah**. Suppose we add a new native method to the *JUSB* class and want to create the C-header file then use the following command:

```
javah –jni usb.windows.USB
```

The corresponding header file usb_windows_USB.h is put into the Java root directory, which should be *JavaSources.* Remember while creating the JNI

header file that we always have to write the whole package name of the class and need to call the command from the Java root directory. If we disregard this restriction and call javah just in the current Windows directory then we would get a JNI header file named as USB.h. The information about the package is lost and this leads to error while loading the JUSB DLL [3].

8.4.2 Directory and File Description

We describe only the files belonging to the Windows package. For the description of the whole Java USB project refer to [18].

The usb.windows package files are in the following directory path: \JavaSources\usb\windows. Table 38 lists those files.

Filename	Description
DeviceImpl.java	Implements the class to which all USB devices belongs to either running as a JUSB device or not.
JUSB.java	Contains methods that can be used for devices that are adapted for the JUSB driver. This class does implement the DeviceSPI class.
NonJUSB.java	Does implement the DeviceSPI method in throwing only IOExceptions otherwise it does nothing else.
USB.java	This class implements the Bus class and provide access to a USB bus.
USBException.java	Contains the USBExceptions that are thrown when having a USB specific error.
Windows.java	This class implements a singleton USB host.

Table 38: Files in the usb.windows package

8.5 jUSB DLL

Installation of JUSB DLL

This section provides information for developer interested in extending the jUSB DLL for the Java USB API. There is a little operating system version conflict between Windows 2000 and Windows XP that we encounter while creating the DLL on Windows 2000. All the development has been made on Windows XP Professional and a full installation of the DDK [6] and SDK [24]. To give developers on Windows 2000 the opportunity to extend this Java USB API a specific section introduce the different project settings in the Visual C++ environment. We recommend new developers to use Windows XP or higher environments for developing the Java USB API for Windows.

At first, the Microsoft SDK 2003 has to be installed on the computer.

Start the Visual C++ environment and open the workspace (File→Open Workspace…) jusb.dsw which can be found in the *JusbDll\jusb*\ folder. Before start editing and working on the files, we need to set the project setting depending on the operating system we are currently running. Next subsection is going to explain the project settings in detail. To do the setting in the Visual C++ environment choose Project→Settings and then choose the appropriate rubric.

8.5.1 Visual C++ 6.0 Project Setting

Some project settings need to be done to successfully build the JUSB dynamic link library. The setting depends on the Windows Version (2000 or XP) and if we have installed the driver development kits or not.

In all cases we need the proper installation of the Microsoft SDK 2003. To make sure the SDK is added to the Visual C++ project check (Tools→Options.. →Directories) check if the following entry "C:\Programme\Microsoft SDK\include" depending on where we installed the SDK is on the **top** by the include files.

8.5.1.1 Project Settings without the DDK

To make it possible to build the dynamic link library without the driver development kit from Microsoft, the used header file from the DDK are made available in the following folder: *JusbDll**external-header-file**ddk*.

Attention of the Environment Variables

The next paragraph describes the project settings for Windows 2000 and XP. Because we use environment variables, we have to be aware of one important fact for the settings. Usually the environment variable are used in the project settings as follows:

```
$(JUSBPATH)\JusbDll\external-header-file\java\include\
```

but if the environment variable contains spaces in the string, for example JUSBPATH defined as "C:\Documents and Settings\JavaUSB" then we need to quote the entry in the project settings!

```
"$(JUSBPATH)\JusbDll\external-header-file\java\include\"
```

If we do not care about this fact, the compiler will not build the DLL and ends with an error like : *error LNK 1104: cannot open file "Documents.obj"*.
Thereby belongs the Documents.obj file to the the name of C:\Documents where in fact does not exist. To make sure that such an error does not occur, choose an environment variable value with no spaces within the path or quote all the project settings where an environment variable is used.
In the following example we assume to have environment variable values without any spaces within the string.

Project Settings in Windows 2000 without the DDK

8.5.1.2 Windows 2000

Add the SDK to Options as described in 8.5.1 and then set the following project settings as in Table 39.

Rubric **C/C++**: Category: Preprocessors **Additional include directories:** *$(JUSBPATH)\JusbDll\external-header-file\java\include\,* *$(JUSBPATH)\JusbDll\external-header-file\java\include\win32\,* *$(JUSBPATH)\JusbDll\external-header-file\ddk\inc\,* *$(JUSBPATH)\JusbDll\jni\,* ***$(JUSBPATH)\JusbDll\external-header-file\ddk\inc\w2k*** Rubric **Link** Category: General **Object/Library modules:** (add the following entries to the existing) ***$(JUSBPATH)\JusbDll\external-lib-file\w2k\setupapi.lib*** *$(JUSBPATH)\JusbDll\external-lib-file\hid.lib*

Table 39: Project settings in Windows 2000 without DDK

Comment Out Functions

If we now build the JUSB DLL we get some error of SPDRP_XXX undeclared identifier. This happens because we tried to use the new SetupDiXxx API function to retrieve registry information. Windows 2000 does not support completely those functions. Therefore we need to comment out 3 functions in the helperFunction.cpp file. These are:

- getRegistryPropertyString
- doDriverNameToDevicesDesc

- getRegistryInfo

Further we need to comment out the three function prototype in the jusb.h file.
We do not use these function either in Windows XP, but they are already coded to be used for future work on the Java USB API.
After having done those changes the JUSB DLL should be built without any errors.

Project Settings in Windows XP without DDK

Windows XP
The jusb.dsw project contains all the current settings and therefore no additional settings should be necessary. The following setting should be predefined:

Rubric **C/C++**: Category: Preprocessors **Additional include directories:** *$(JUSBPATH)\JusbDll\external-header-file\java\include\,* *$(JUSBPATH)\JusbDll\external-header-file\java\include\win32\,* *$(JUSBPATH)\JusbDll\external-header-file\ddk\inc\,* *$(JUSBPATH)\JusbDll\jni* Rubric **Link** Category: General **Object/Library modules:** (add the following entries to the existing) *setupapi.lib* *$(JUSBPATH)\JusbDll\external-lib-file\hid.lib*

Table 40: Project settings in Windows XP without DDK

8.5.1.3 Project Settings with an Installed DDK

If the DDK is installed, it does not make sense to use the DDK header provided in the *JavaUSB* folder and is appropriated to use the original DDK header files. The paragraph “Attention to environment variables” in 8.5.1.1 still has its validity.

Project Settings in Windows 2000 with DDK

Windows 2000
Next to the project settings there must be done additional changes in different source files. First, set the project settings according to Table 41.

Rubric **C/C++**: Category: Preprocessors **Additional include directories:** *$(JAVAHOME)\include\,* *$(JAVAHOME)\include\win32\,* *$(DDKPATH)\inc\w2k\,* *$(DDKPATH)\inc\ddk\w2k\,* *$(JUSBPATH)\JusbDll\jni* Rubric **Link** Category: General **Object/Library modules:** (add the following entries to the existing) *$(DDKPATH)\lib\w2k\i386\setupapi.lib* *$(DDKPATH)\lib\w2k\i386\hid.lib*

Table 41: Project settings in Windows 2000 with the DDK installed

Function need to be out documented in Windows 2000

If we now build the JUSB DLL we get some error of the form SPDRP_XXX undeclared identifier. This happens because we tried to use the new SetupDiXxx API function to retrieve registry information. Windows 2000 does not completely support those functions. Therefore, we need to comment out 3 functions in the helperFunction.cpp file. These are:

- getRegistryPropertyString
- doDriverNameToDevicesDesc

- getRegistryInfo

Further, we have to comment out the three function prototypes in the jusb.h file. We do not use these function either in Windows XP, but they are already coded to be used for future work on the Java USB API.

New Constants definition in jusb.h running on Windows 2000

We also need to activate the comment out definition in jusb.h for bmRequest.Dir, bmRequest.Type and bmRequest.Recipient which will be found almost on the top. The reason for that definition is because we already used those constants which are defined in usb100.h in the DDK. The DDK file usb100.h in Windows 2000 does not define those constants. In the newer version of DDK XP they are defined in usb100.h. Table 42 shows the correct settings for Windows 2000.

```
// Only used when running on Windows 2000!
//bmRequest.Dir
#define BMREQUEST_HOST_TO_DEVICE       0
#define BMREQUEST_DEVICE_TO_HOST       1

//bmRequest.Type
#define BMREQUEST_STANDARD            0
#define BMREQUEST_CLASS               1
#define BMREQUEST_VENDOR              2

//bmRequest.Recipient
#define BMREQUEST_TO_DEVICE           0
#define BMREQUEST_TO_INTERFACE        1
#define BMREQUEST_TO_ENDPOINT         2
#define BMREQUEST_TO_OTHER            3
```

Table 42: Definition of bmRequest constants only under Windows 2000

Out document some line in getAttachedDeviceType function

Furthermore, we need to comment out some *else* branches in the *getAttached-DeviceType* function in jusb.cpp. The reason for that is the modification from the usbioctl.h header file in the DDK. The USB_CONNECTION_STATUS enumeration type has been extended with two members (DeviceHubNestedTooDeeply and DeviceInLegacyHub). Those members are not known in the Windows 2000 environment and we therefore have to comment out those lines as shown in Table 43.

```
int getAttachedDeviceType(HANDLE hubHandle, int portIndex){
…
  if(…){
  …
  }else if(connectionInfo.ConnectionStatus[0] == DeviceNotEnoughBandwidth){
   return -5;
  }else /* if(connectionInfo.ConnectionStatus[0] == DeviceHubNestedTooDeeply){
   return -6;
   }else if(connectionInfo.ConnectionStatus[0] == DeviceInLegacyHub){
    return -7;
   else */ if(connectionInfo.ConnectionStatus[0] == DeviceFailedEnumeration){
   return -8;
   }else return 0;
…
}
```

Table 43: Modified getAttachedDeviceType function (Windows 2000)

After having done those changes, the JUSB DLL should be built without any errors.

Windows XP

Table 44 shows all the additional settings.

Project Settings in Windows XP with the DDK

Rubric **C/C++**: Category: Preprocessors

Project Settings in Windows XP without the DDK	**Additional include directories:** *$(JAVAHOME)\include\,* *$(JAVAHOME)\include\win32\,* *$(DDKPATH)\inc\wxp\,* *$(DDKPATH)\inc\ddk\wxp\,* *$(JUSBPATH)\JusbDll\jni* Rubric **Link** Category: General **Object/Library modules:** (add the following entries to the existing) *setupapi.lib* *$(DDKPATH)\lib\wxp\i386\hid.lib*

Table 44: Project settings in Windows XP with installed DDK

8.5.2 Directory and File Description

All the files required to build the jUSB DLL are available in the *JusbDll* folder. The folders within the *JusbDll* folder are listed in Table 45.

Foldername	Description
jusb	Contains all C++ source files, C-header files and all the files to create the project workspace for the Visual C++ environment.
jni	Contains all JNI header files that are created with javah. The header files are named automatically when executing javah. These header files should not be modified. If new native methods have been added to a Java class, run javah with the modified Java class to acquire the corresponding JNI header file, and copy that header file if necessary in that folder.
external-lib-file	Contains libraries which are used when no DDK is installed (see 8.5.1.1).
external-header-file	Contains header files that are used when no DDK is installed (see 8.5.1.1).

Table 45: Folders in JusbDll Folder

The files in the *jni*, *external-lib-file* and *external-header-file* folder are not explained in detail. The important files for developers are in the *jusb* folder which will be described in the following paragraphs.

Figure 18 provides an overview about the files which are related to each other. The general framework decision was to provide for every automatically created JNI header file its own file where the implementation is done. Furthermore, the decision was made to keep the function code within these implementation files as short as possible to keep it readable. If a function implementation becomes more complex, an external function was made to process the desired request. These external functions are put in the jusb.cpp, helperFunction.cpp and devnode.cpp files. Figure 18 represents this partitioning in the block B. All three blocks A, B and C represent one function body.

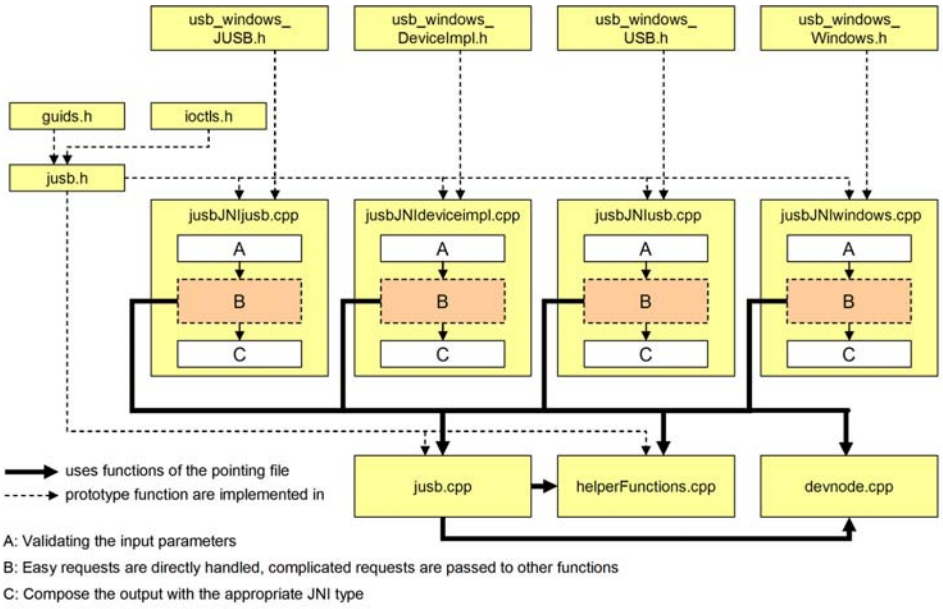

Figure 18: File composition of jUSB DLL project

Filename	Description
guids.h	Contains the GUID for GUID_DEFINTERFACE_JUSB_DEVICES which the jUSB driver registers when a device uses the JUSB driver. With the aid of this GUID we are able to locate JUSB devices. More about GUID can be found in Appendix X .
ioctls.h	Contains all the IOCTL codes which are available in the jUSB driver.
jusb.h	Definition of structures, variables and constants. Further, it contains also all function prototypes that are used in more than one file.
devnode.cpp	Contains one function that retrieves the DeviceDesc registry entry from a given driver name. This file may be obsolete for future work and should be replaced with the new registry function, such as getRegistryPropertyString, getRegistryInfo and doDriverNameToDeviceDesc which are already implemented in the helperFunctions.cpp.
helperFunctions.cpp	Contains function to process complicated requests.
jusb.cpp	Is the entry point for the DLL. It contains also function to process complicated requests.
jusbJNIdeviceimpl.cpp	Implements the JNI function of the Java *DeviceImpl* class.
jusbJNIjusb.cpp	Implements the JNI function of the Java *JUSB* class.
jusbJNIusb.cpp	Implements the JNI function of the Java *USB* class.
jusbJNIwindows.cpp	Implements the JNI function of the Java *Windows* class.
jusb.dsw	Visual C++ Workspace
jusb.dsp	Visual C++ Project

jusb.html jusb.mak jusb.opt jusb.ncb	The next four files belonging to the Visual C++ environment.

Table 46: Descriptions of files in the jusb folder

8.6 JUSB Driver

This section describes all parts of the jUSB driver. Useful information is provided to build and develop the jUSB driver. It is absolute necessary that the SDK [24] and the DDK [6] from Microsoft is installed and the relevant environment variables are correctly set.

The build process depends not on the operating system we develop. The procedure is identical in Windows 2000 and Windows XP, because we assume having installed on both operating systems the DDK XP version.

The Microsoft® Windows® Driver Development Kit (DDK) release for Microsoft® Windows® XP incorporates a number of significant design changes. The most important of these include: a new setup program, new build environments, a redesign of the layout of the installed headers and libraries, and new build tools that make the new DDK a stand-alone product. A feature has also been added to the build environment to help developers identify the use of deprecated functions in their code at build time [22].

8.6.1 How to Build the Driver

As introduction we present a section (Table 47) of the paper *New in the DDK for Windows XP* [22].

New Build Environment
A number of important changes have been made to the Windows DDK build environment. For one, the Windows DDK now includes a complete set of tools for building drivers. Microsoft® Visual C++® is no longer required to be installed to use the DDK. Use of the included tools for all Windows 2000 and Windows XP drivers is expected within the shipping build environment. This version of the Windows DDK does not support building Windows XP or Windows 2000 drivers using a version of Microsoft Visual C++ earlier than the one supplied with the DDK. Attempts to use an incorrect version of Visual C++ will result in the following error message from the compiler:

error C1189: #error : Compiler version not supported by Windows DDK

This requirement is due to the reliance on many new features within this tool set for proper functioning of the include build environment. The compiler, linker, and supporting files, such as C Run-Times (CRTs), should be considered an atomic unit within the build environment. It is likely that mixing the supporting tool files of Visual C++ versions 5 or 6 with those in this DDK release, which are based on the new Visual C++ version 7 code base, will result in errors in the binaries. Using the provided build environment and build tools is thus strongly recommended to ensure generation of valid binaries [22].

Table 47: Build environment from the DDK

The conclusion of the section in Table 47 is that we can not build the jUSB driver within Visual C++ Version 6.0 because the compiler version does not correspond to the DDK version. To compile the jUSB driver within Visual C++ environment, an update of Visual C++ is required (Visual .NET). We started developing the driver in Visual C++ environment and did not change the environment during the project. Therefore we build the driver with the build environment delivered within the DDK but still edit the code within the Visual C++ environment.

Building Steps

Build the jUSB driver:

1. Start the Win XP Checked Build Environment (also in Windows 2000) (Start→Programs→Development Kits→Windows DDK 2600.1106→ BuildEnvironments→Win XP Checked Build Environment)
2. Change the directory path so that it points to the *sys* folder which is a subfolder of *JusbDriver*. For example:
   - change the drive: **F:** <enter>
   - change the folder path: **cd JavaUSB\JusbDriver\sys** <enter>
3. Enter command: **build –cZ** <enter>
   Some states of the building process are printed on the output screen (an output example is shown in Table 48).
   The most important statement is : 1 executable built
   If this statement is missing check chapter 8.6.1.1 for more information
4. The compiled jUSB driver (jusb.sys) can be found in the following sub-folder of the *sys* folder: ***\objchk_wxp_x86\i386***
5. Copy the jusb.sys file into the *system32\drivers* folder of the Windows main directory.
   If there is already a registered jUSB driver in the directory, only a re-placement of the jsub.sys file needs to be done. Otherwise if it is the first time using the jUSB driver refer to chapter 7.2 (user installation) for more information about how to register the jUSB driver.

Screen Shot of Building Process

```
F:\Studium\JavaUSB\JusbDriver\sys>build -cZ
BUILD: Adding /Y to COPYCMD so xcopy ops won't hang.
BUILD: Object root set to: ==> objchk_wxp_x86
BUILD: Compile and Link for i386
BUILD: Examining f:\studium\javausb\jusbdriver\sys directory for files to compile.
BUILD: Building generated files in f:\studium\javausb\jusbdriver\sys directory
BUILD: Compiling f:\studium\javausb\jusbdriver\sys directory
Compiling - jusb.rc for i386
Compiling - driverentry.c for i386
Compiling - plugplay.c for i386
Compiling - power.c for i386
Compiling - control.c for i386
Compiling - wmi.c for i386
Compiling - readwrite.c for i386
Compiling - generating code... for i386
BUILD: Linking f:\studium\javausb\jusbdriver\sys directory
Linking Executable - objchk_wxp_x86\i386\jusb.sys for i386
BUILD: Done

   8 files compiled
   1 executable built

F:\Studium\JavaUSB\JusbDriver\sys>
```

Table 48: Output of jUSB driver build process

8.6.1.1 No Driver Executable Built

In case of missing the statement: "**1 executable built**" as shown in Table 48 check the **buildchk_wxp_x86.log** file. If the string *"'jvc' is not recognised as an internal or external command"* then your folder path to the *JusbDriver* folder contains somewhere spaces.

Spaces are not allowed in the driver path!

Solution: Copy the *JusbDriver* folder in a path with no spaces and do build the driver again as described in 8.6.1.

8.6.2 Directory and File Description

The files for the jUSB driver are all in the *JusbDriver\sys* folder. The following Table 49 lists all those file.

Filename	Description

Control.c	Implements the DispatchControl function which handle all I/O request packet (IRP) with function code IRP_MJ_DEVICE_CONTROL in the major field of the IRP.
Driver.h	Header file containing all definition of structs and global variables used in the driver.
DriverEntry.c	The entry point to the jUSB driver. Similar to a main file.
guids.h	The definition of the device interface, a global unique identifier named GUID_DEFINTERFACE-_JUSB_DEVICES
ioctls.h	The definition of IOCTL codes handled by the jUSB driver.
jusb.inf	The INF file used to register the jUSB driver to a device using an INF file.
jusb.reg	File to register the jUSB driver in the Windows registry.
jusb.bmf jusb.mof	Used in the makefile to build the jUSBdriver.
jusb.rc	Resources file containing information about the jUSB driver.
PlugPlay.c	Implements the DispatchPnP function which handles all I/O request packet (IRP) with function code IRP_MJ_PNP in the major field of the IRP.
Power.c	Implements the DispatchPower function which handles all I/O request packet (IRP) with function code IRP_MJ_POWER in the major field of the IRP.
ReadWrite.c	Implements the DispatchReadWrite function which handles all I/O request packet (IRP) with function code IRP_MJ_READ or IRP_MJ_WRITE in the major field of the IRP. This file is leftover from the bulkusb project and is not used so far in the jUSB driver. Some functions are used in other files and therefore this file has not been removed.
Wmi.c	Implements the WmiRegistration function.
All the other file in this folder belongs either to the project settings for Visual C++ or to the build process. There are not further described.	

Table 49: Files and its description in the JusbDriver folder

9 Conclusion

The goal of this diploma thesis was to extend the Java USB API fort the Windows operating system.

The goal could not be reached but a part of the jUSB project is working.

The enumeration and monitoring facility of the universal serial bus with the Java USB API is complete and working. Communication to a jUSB device such as interrupt transfer and control transfer are partly implemented and has been successfully tested on USB devices. Bulk transfer and isochronous transfer are not supported at the moment but are subjects of future work. The project, as it is, provides a basic framework for the Java USB API for Windows and developers are welcomed to modify and build a stable jUSB distribution.

The following subjects have put to future work:

- Writing a stable jUSB driver including documentation
- Implement the methods of the *DeviceSPI* interface in the jUSB DLL and in the jUSB driver.
- Implement the *DriverNameToDeviceDesc* function in devnode.c with the *SetupDiXxx* function to read registry entries.

The difficulty was to understand what is going on in a driver and how all the requests have to be handled that they correspond to the current Windows Driver Model. The project time was too short to understand completely driver writing and modelling but still a very interesting topic.

Appendix A: IOCTL codes used by the JUSB framework

IOCTL codes are used within the *DeviceIoControl* user mode API function to retrieve information from a device. These IOCTL codes allow programmers to access kernel mode functionality from within the user mode,. The following list presents all jUSB driver IOCTL codes, The supplied input structure and its corresponding output structure. In some cases, the input structure is the same as the output structure but this is not always the case.

I: JUSB IOCTL codes

The following IOCTL codes are used in the jUSB driver and defined in ioctls.h file.

I a) IOCTL_JUSB_GET_DEVICE_DESCRIPTOR

Returns the device descriptor in the output buffer. The output buffer must have the size of the USB_DEVICE_DESCRIPTOR structure. The input buffer is not being treated, so we can put this parameter to NULL. The members of this structure are described in the DDK or on the online MSDN documentation [19].

```
typedef struct _USB_DEVICE_DESCRIPTOR {
  UCHAR  bLength ;
  UCHAR  bDescriptorType ;
  USHORT  bcdUSB ;
  UCHAR  bDeviceClass ;
  UCHAR  bDeviceSubClass ;
  UCHAR  bDeviceProtocol ;
  UCHAR  bMaxPacketSize0 ;
  USHORT  idVendor ;
  USHORT  idProduct ;
  USHORT  bcdDevice ;
  UCHAR  iManufacturer ;
  UCHAR  iProduct ;
  UCHAR  iSerialNumber ;
  UCHAR  bNumConfigurations ;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR ;
```

Table 50: USB_DEVICE_DESCRIPTOR structure

I b) IOCTL_JUSB_GET_CONFIGURATION_DESCRIPTOR

Returns the ith configuration descriptor of the device. Two steps are necessary to successfully execute this IOCTL code. The input and output buffer belongs to the *DeviceIoControl* WinAPI function.

1. Step
Input buffer: Index of type USHORT
Output buffer: NULL
nReturnedBytes: Contains the length of the ith configuration descriptor

2. Step
Input buffer: Index of type USHORT
Output buffer: The size of nReturnedBytes, type of UCHAR
nReturnedBytes: Contains the length of the ith configuration descriptor

I c) IOCTL_JUSB_GET_STRING_DESCRIPTOR

Retrieves the string descriptor from a given index and language. To dynamically allocate memory for the string descriptor we will need two *DeviceIoControl* calls. The first call returns the USB_STRING_DESCRIPTOR structure without the driver key name, but tells in the bLength member, how many bytes the string descriptor needs. In a second call we provide a buffer big enough to hold the

entire length of the string descriptor and the STRING_REQUEST structure. To tell the driver which string descriptor we look for a STRING_REQUEST structure is always put at the beginning of the input and output buffer to send the input parameters to the jUSB driver.

```
typedef struct _STRING_REQUEST{
  UCHAR ucDescriptorIndex;
  USHORT usLangId;
} STRING_REQUEST, * PSTRING_REQUEST;

typedef struct _USB_STRING_DESCRIPTOR {
  UCHAR  bLength ;
  UCHAR  bDescriptorType ;
  WCHAR  bString[1] ;
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR ;
```

Table 51: STRING_REQUEST and USB_STRING_DESCRIPTOR structures

I d) IOCTL_JUSB_GET_STATUS

Returns the status for the specified recipient (bmRequestType) which is always two bytes (further information can be found in the USB specification chapter 9.4.5). The input buffer consists of two bytes of type USHORT. The first byte contains the bmRequestType and the second byte the wIndex field. The output buffer will contain the data that is returned by the GET_STATUS request.

I e) IOCTL_JUSB_INTERRUPT_TRANSFER

This IOCTL code invokes the jUSB driver to do a synchronously interrupt request to the driver. The input buffer must be as big as the number of bytes we want to read. To tell the driver of which endpoint we want to read, we set the first byte of the input buffer with the endpoint address. When successfully complete the request the output buffer contains the bytes readed. For more information look at the source, which is in the DoInterruptTransfer function in the Control.c file.

II: Other IOCTLs

The following IOCTL:

- IOCTL_USB_GET_NODE_CONNECTION_NAME
- IOCTL_USB_GET_NODE_CONNECTION_INFORMATION
- IOCTL_USB_GET_NODE_CONNECTION_DRIVERKEY_NAME
- IOCTL_USB_GET_NODE_INFORMATION
- IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION
- IOCTL_USB_GET_ROOT_HUB_NAME

are applied in the usbview example (DDK) or very clearly arranged in an example by John Hide which is online available on:

www.intel.com/intelpress/usb/examples/DUSBVC.PDF

We do not explain this IOCTL codes any further.

Appendix B: Global Unique Identifier GUID

The Windows driver model introduces a new naming scheme for devices which is language neutral. The schema relies on the concept of a device interface, which is basically a specification of how software can access hardware. Each device interface is uniquely identified by a 128-bit GUID [4].

GUIDGEN

The procedure of creating a GUID to be used in a device driver involves running either UUIDGEN or GUIDGEN and then capturing the resulting identifier in a header file. GUIDGEN is easier to use because it allows to format the GUID for use with the DEFINE_GUID macro and to copy the resulting string to the Clipboard (see Figure 19).

The created GUID is saved in the guids.h file which can be found in the *JusbDll**jusb* folder or \JusbDriver\sys and which looks as follows:

```
// {07D25E7A-CBDD-4f69-9BE1-FCCF14F4B299}
DEFINE_GUID(GUID_DEFINTERFACE_JUSB_DEVICES,
0x7d25e7a, 0xcbdd, 0x4f69, 0x9b, 0xe1, 0xfc, 0xcf, 0x14, 0xf4, 0xb2, 0x99);
```

Table 52: GUID_DEFINTERFACE_ JUSB_DEVICES

The name of the GUID is GUID_DEFINTERFACE_JUSB_DEVICES and it is used to identify the JUSB devices within the JUSB DLL.

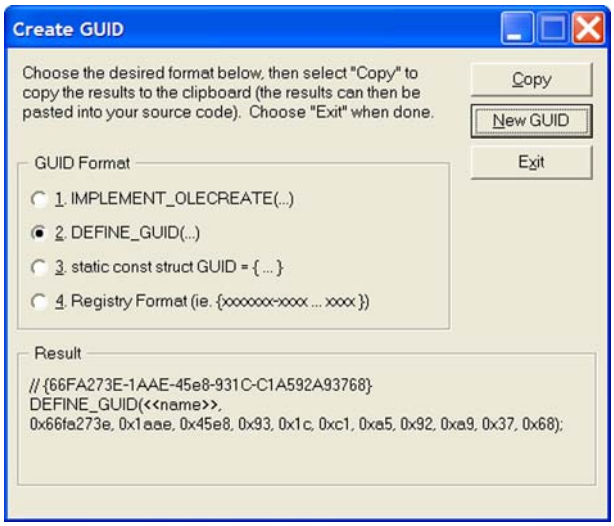

Figure 19: Using GUIDGEN to generate GUID

The output in Figure 19 does not correspond to the output in Table 52 because the guidgen has been executed again for documentation purposes and will hopefully never return the same result.

Start GUIDGEN

Start GUIDGEN with: Start→Run and enter *guidgen*

The guidgen programm is part of the SDK and is installed in the *MicrosoftSDK**Bin folder*.

Appendix C : Device Interface Classes

Device interface classes are the means by which drivers make devices available to applications and other drivers.

I: Introduction to Device Interfaces

Any driver of a physical, logical, or virtual device to which user-mode code can direct I/O requests must supply some sort of name for its user-mode clients. Using the name, a user-mode application (or other system component) identifies the device from which it is requesting I/O.
In Windows NT® 4.0 and earlier versions of the NT-based operating system, drivers named their device objects and then set up symbolic links in the registry between these names and a user-visible Win32® logical name.
For Windows® 2000 and later, drivers do not name device objects. Instead, they make use of device interface classes. A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, than enable an instance of the class for each device object to which user-mode I/O requests might be sent.
Each device interface class is associated with a GUID. The system defines GUIDs for common device interface classes in device-specific header files. Vendors can create additional device interface classes.
For example, three different types of mice could be members of the same device interface class, even if one connects through a USB port, a second through a serial port, and the third through an infrared port. Each driver registers its device as a member of the interface class GUID_DEVINTERFACE_MOUSE. This GUID is defined in the header file ntddmou.h.
Typically, drivers register for only one interface class. However, drivers for devices that have specialized functionality beyond that defined for their standard interface class might also register for an additional class. For example, a driver for a disk that can be mounted should register for both its disk interface class (GUID_DEVINTERFACE_DISK) and the mountable device class (MOUNTDEV_MOUNTED_DEVICE_GUID).
When a driver registers an instance of a device interface class, the I/O Manager associates the device and the device interface class GUID with a symbolic link name. The link name is stored in the registry and persists across system boots. An application that uses the interface can query for instances of the interface and receive a symbolic link name representing a device that supports the interface. The application can then use the symbolic link name as a target for I/O requests.

II: Register Device Interfaces in a Driver

IoRegisterDeviceInterface registers a device interface class, if it has not been previously registered, and creates a new instance of the interface class. A driver can call this routine several times for a given device to register several interface classes and create instances of the classes. A function or filter driver typically registers device interfaces in its *AddDevice* routine.
If the device interface class has not been registered previously, the I/O Manager creates a registry key for it, along with instance-specific persistent storage under the key.
A driver registers an interface instance once and then calls *IoSetDeviceinterfaceState* to enable and disable the interface.
Registered interfaces persist across operating system reboots. If the specified interface is already registered, the I/O Manager passes its name in SymbolicLinkName and returns the informational success status STATUS_OBJECT_NAME_EXISTS.

Most drivers use a NULL reference string for a device interface instance. If a driver uses a non-NULL reference string, it must do additional work, including

possibly managing its own namespace and security.
Callers of this routine are not required to remove the registration for a device interface when it is no longer needed. Device interface registrations can be removed from user mode, if necessary.

Callers of *IoRegisterDeviceInterface* must be running at IRQL = PASSIVE_LEVEL in the context of a system thread.

Appendix D: Replacement of origin driver with the jUSB driver

Once we want to develop a device with the Java USB API it comes to the point where we need to replace the origin driver from the device and force the device to use the JUSB driver. This section explains how we can configure any USB device to the JUSB driver. There are two situations we come across when we attach a device to the USB bus:

- The hardware assistant starts and looks for an appropriate INF file that contains further information about the driver to be installed.
- The operating system automatically loads the appropriate driver for the USB device. This is usually done for devices associated with a class driver. Windows supports all known class drivers from the standard USB specification.

Both situations demand for a different procedure to install the JUSB driver. There are some aspects to concern about, which will be explained in the next two sections.

I: Install the jUSB driver

Register the JUSB Driver

Before we can configure a USB device to use the JUSB driver, we need to install the driver. If the USB device is configured with an INF file, we can pass to 0. In all other cases, we need to register the JUSB driver to the Windows registry and put the driver file JUSB.sys into the \WINDOWS\SYSTEM32\DRIVERS folder. To register this driver double click on the jusb.reg file in the "\JUSBDriver\Treiber Installations Dateien". This will add the JUSB driver to the registry. For additional devices we do not have to repeat this process, because the JUSB driver remains in the registry as long as no changes to the registry are made.
The entry can be found in HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001 \Services\JUSB

II: USB Device with an INF file

If an USB device comes with its own driver attached on an external disk, then in most cases an INF-file is supplied for that driver. The first thing we have to do if we do not know the vendor and product id of the device is reading the first paragraph of 0 to get the id's. In a second step we do prepare the INF file for the JUSB driver for our own needs. Open the jusb.INF file (should be concerned as a default file) from the *InstallationFiles**JusbDriver* folder (Table 53 shows the content of jusb.inf file). The INF file contains several sections which are denoted within brackets ('[', ']'). To distinguish this INF file from other INF files using the JUSB driver we save the default jusb.INF file under an appropriate name which belongs to the USB device we want to add, e.g. the device we want to work with is called MyPen [20].

1. Save jusb.inf as e.g. jusb-mypen.inf
2. section **[SourceDiskFiles]**:
   change JUSB.inf to jusb-mypen.inf
3. section **[JavaUSBDevices]**
   change two times the VID and PID according to the VID and PID the USB device is identified.
4. section **[JUSB.Files.Inf]**
   change JUSB.inf to jusb-mypen.inf
5. section **[String]**
   change VID and PID. The DeviceDesc string has to be set, so that it starts with the "**JUSB Driver --:**".
6. Save jusb-mypen.inf in the *InstallationFiles**JusbDriver* folder.

```
; Installation inf for the JUSB (Java USB) driver
```

```
;
; (c) Copyright 2003 ETH Zürich Institute for Information Systems
;

[Version]
Signature="$CHICAGO$"
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
provider=%ETHGLOBIS%
DriverVer=07/28/2003

[SourceDisksNames]
1="JUSB Installation Disk",,,

[SourceDisksFiles]
JUSB.sys = 1
JUSB.inf = 1

[Manufacturer]
%MfgName%=JavaUSBDevices

[JavaUSBDevices]
%USB\VID_0A93&PID_0002.DeviceDesc%=JUSB.Dev, USB\VID_0A93&PID_0002
…

[JUSB.Files.Ext]
JUSB.sys

[JUSB.Files.Inf]
JUSB.Inf

 [Strings]
ETHGLOBIS="ETH Zürich Institute for Information Systems"
MfgName="Stahl"
;FriendlyDeviceName has always to start with "JUSB Driver --:"
;
;In case of a device having a friendly device name starts with "JUSB Driver --:"
;the Java USB API will put that device in the JUSB class, In the other case
;the device will be put to the NonJUSB class.
;
USB\VID_0A93&PID_0002.DeviceDesc="JUSB Driver --:  JUSB Device"
JUSB.SvcDesc="JUSB.Sys Java USB Driver"
```

Table 53: Fragment of the jusb.inf file. Highlighted are the sections that have to be modified to fit for other USB devices

Replace the driver using the Device Manager in Windows. According to the example we are looking for a device called MyPen. At this point we can update the driver. The sources for the jUSB driver are found in the *InstallationFiles\JusbDriver*.

III: Class USB Devices

A class USB device is usually automatically recognised by the Windows operating system and the corresponding driver is loaded for that device. The hardware assistant may inform that it found a new USB device and its driver has been successfully loaded. If we have such a device, then we need to edit the registry to make that device available to the JUSB driver. The steps are related to a mouse which is the class of a Human Interface Device (HID). A mouse can be attached to the computer and without any settings the mouse can be used. This is exactly what we do first of all, we attach the USB device we want to configure for the JUSB driver to the USB bus. In a second step we need to know the vendor (VID) and product id (PID) of the device. This can be done in the following ways:

- Start RunUSBControllerTest.java. This will do a scan of the USB bus

and display its attached devices. With the friendly device name we are able to identify the device we just attached. If we have two identical devices, detach one of them, so we are sure to see only the connected device we are looking for. Look at the uniqueID string to get the vendor id (Vid_xxxx) and the product id (Pid_xxxx).

- Use the usbview executable from the DDK to get those informations.
- Look in the registry under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB. This approach is like look and guess. The first two approaches are recommended to retrieve the VID and PID.

As soon we know the PID and the VID we start the registry editor (use Start->Run and type “regedit”). Go to the following folder:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB.
In that folder are all USB devices that once have been attached to the USB bus. It contains a ROOT_HUB and a ROOT_HUB20 folder in which the settings and information about the root hubs on the system can be found. Plenty of folders named like Vid_xxxx&Pid_xxxx represent devices belonging to those VID and PID. The mouse we are going to configure for the JUSB driver has VID 046d and PID c00e (in Figure 20 (1.) a folder name containing the VID and PID).

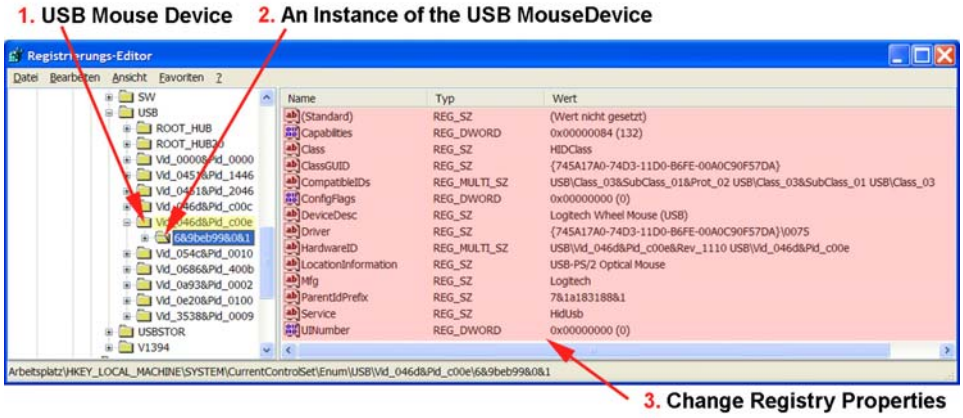

Figure 20: Registry entries in HKLM\SYSTEM\CurrentControlSet\Enum\USB

This means we have found the mouse device in the registry. The subfolders (Figure 20 (2.)) contain instances of a mouse device. We may have more USB devices with the same VID and PID and for each of those an instance will be created. If we select this instance (in our example there is only one) we get some registry properties on the right hand side To change the properties of that device instance, the security attributes of the instance folder may have to be changed to write access otherwise the registry entry cannot be changed (for more information see 0). Table 54 lists the entries that have to be changed to configure the device for the JUSB driver.

Class	**USB**
ClassGUID	**{36FC9E60-C465-11CF-8056-444553540000}** Is the system supplied setup class for USB [26].
DeviceDesc	**JUSB Driver --: <the recent DeviceDesc name>**
Driver	**{07D25E7A-CBDD-4f69-9BE1-FCCF14F4B299}\nnnn** {07D25E7A-CBDD-4f69-9BE1-FCCF14F4B299}: is the GUID for the JUSB Driver Interface Class. This GUID is defined in JUSBDriver\sys\guids.h. nnnn: a number like 0001. For every JUSB Driver increment this number. There should not exist another JUSB driver with the same number.
ParentIdPrefix	**delete** the complete entry
Service	**JUSB**

Table 54:Change of Regitry Entries for a JUSB Device

The complete list of settings for a JUSB device is presented in Figure 21

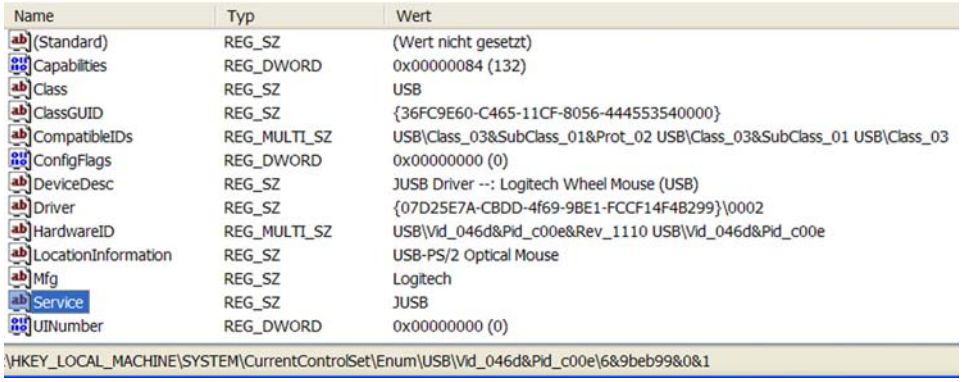

Figure 21: Registry entries for a jUSB device

In Windows XP a reboot of the system is not required, but at least the device has to be unplugged and plugged again to the bus so the change in the registry takes effect. As soon these properties are set, the device is now being recognised as a JUSB device and will process all request to the JUSB driver. In fact the assumption is made that the JUSB driver is already in the C:\WINDOWS\SYSTEM32\DRIVERS folder. Otherwise it needs to be copied to that location.
The example with the mouse shows the effect of not working correct anymore. The mouse can be moved but no cursor can be seen. In other words, the mouse device is in the control of the JUSB driver, hopefully not out of control!

IV: How to change Registry Security Attributes

We may encounter the following error messages as displayed in Table 55 when trying to modify the registry values.

Windows 2000/XP:	Error Editing Value Cannot edit <value>: Error writing the value's new content

Table 55: Error while trying to modify registry entries

The reason for that failure is because the Windows operating system in the basic setting only grants read access to registry entries. To change the security permission we must log in as system administrator and perform the following steps depending on the Windows version:

Windows 2000:

1. Security permissions have to be set with RegEdt32 (Start→Run, enter RegEdt32).
   Select the HKEY_LOCAL_MACHINE window. Open the following folders: *SYSTEM*, *ControlSet001*, *Enum*. Select the *USB* folder and choose Security→Permissions… in the registry editor menu. Give "Everyone" full control access and apply the new changes. Close RegEdt32.
2. Start regedit (Start→Run, enter regedit) and do the necessary changes as described in 0.

Windows XP:

1. Start regedit (Start→Run, enter regedit) and choose the folder where new settings have to be applied. Right click and go to Permissions… and set "Everyone" to full control access.
2. In case no Permissions… field is available in the context menu, other settings in the folder options have to be done (Explorer→Tools→Folder Options→View) under Advanced settings, clear "*Use simple file sharing [Recommended]*". After this setting the security tag should be visible.

Appendix E: Java Native Interface Linking Error

The steps to implement the Java native methods on the C/C++ side are the following:

1. Declare a method in a Java class as native.

```
private native byte[] JUSBReadControl(String devicePath, byte type,byte request,
        short value, short index, short length);
```

2. Create the JNI header file with javah.

```
javah –jni usb.windows.JUSB
```

3. Include the header file in the C/C++ file that implements this native function.
4. Build the DLL

Usually we assume that the DLL function name corresponds to the function name we supplied in the JNI header file. In fact the compiler creates the function name according to the JNI header file.
Unfortunately, we encountered a problem that a Java native method got a fatal linking error while running the Java main program. We did step 1 to 4 as usual but the main application still claimed that no native method of that name exists in the DLL. The reason of such a failure is in mangling the function name by the Visual C++ Compiler. In other words: the compiler creates a new name for the JNI native function which does not correspond to the origin header file and therefore cannot be found by the Java native interface. To eliminate this malfunction of the compiler we used the **dumpbin** [7] command to get the names of the DLL supported functions. According to the dump function names we are able to check if a name mangling has happened or not. Look at the source of *JUSBReadControl* method in the *JUSB* class for further information.

DUMPBIN command

The **dumpbin** command can be used as follows:

```
dumpbin /EXPORTS /LINKERMEMBER C:\WINDOWS\SYSTEM32\jusb.dll
```

Table 56: Dumpbin command to see the export function of a DLL

The output of that command is shown in **Table 57**.

Output of DumpBin

```
Dump of file C:\WINDOWS\SYSTEM32\jusb.dll
  Section contains the following exports for jusb.dll

1    0 000010AF _Java_usb_windows_DeviceImpl_closeHandle@12
2    1 00001050 _Java_usb_windows_DeviceImpl_getAttachedDeviceType@16
3    2 00001069 _Java_usb_windows_DeviceImpl_getConfigurationDescriptor@16
…
11    A 0000110E _Java_usb_windows_JUSB_JUSBReadControl@32
12    B 00001005 _Java_usb_windows_JUSB_doInterruptTransfer@20
13    C 00001104 _Java_usb_windows_JUSB_getConfigurationBuffer@16
14    D 000010F5 _Java_usb_windows_JUSB_getDevicePath@12
15    E 00001064 _Java_usb_windows_USB_getRootHubName@12
16    F 0000106E _Java_usb_windows_Windows_getHostControllerDevicePath@12
17   10 0000100F _Java_usb_windows_Windows_getHostControllerName@12
```

Table 57: Output of dumpbin command

A mangled function name has the following structure:

```
_Java_usb_Windows_JUSB_readControl@@YGPAV_jbyteArray@@PAUJNIEnv_@
@PAV_jobject@@PAV_jstring@@PAV1@@Z
```

Mangled function name

Table 58: A mangled function name by the compiler

Appendix F: A sample of DbgView with HP Scanjet 4100C

Table 59 shows the output of DbgView when we attach a device that is using the JUSB driver. This output is just for information. No further explanation will be given.

	000	0.00000000	- Running under NT
	001	0.00037351	- ENTER AddDevice: DriverObject 821B0ED0, pdo 81E9C5D0
AddDevice create DeviceObject	002	0.00042743	- AddDevice : deviceObject created --> 81D88D80
	005	0.00046263	- AddDevice DELEGATE : to WMI
	006	0.00047352	- ENTER WmiRegistration
	007	0.00050872	- EXIT WmiRegistration
	008	0.00052213	- AddDevice INITIALIZE : attach our driver to device stack
	009	0.00053526	- AddDevice INITIALIZE : Register device interfaces
	011	0.00073920	- AddDevice INITIALIZE : WDM version: Win XP or better
	012	0.00080792	- END: AddDevice
Start Device	013	0.00084340	- ENTER DispatchPnp IRP:IRP_MN_?????
	014	0.00087190	- DispatchPnp (IRP_MN_START_DEVICE) will be sent to the lower driver
	015	0.00099957	- ENTER DispatchPnp IRP_MN_QUERY_RESOURCE_REQUIREMENTS
	016	0.00107332	- DispatchPnp (IRP_MN_START_DEVICE) will be sent to the lower driver
	017	0.00113813	- ENTER DispatchPnp IRP_MN_FILTER_RESOURCE_REQUIREMENTS
	018	0.00115545	- DispatchPnp (IRP_MN_START_DEVICE) will be sent to the lower driver
Configure the DeviceExtension from the Device	019	0.00117305	- ENTER DispatchPnp IRP:IRP_MN_START_DEVICE
	020	0.00118423	- ENTER HandleStartDevice
	021	0.00241204	- ENTER ReadAndSelectDescriptors
	034	0.02080823	- ENTER ConfigureDevice: ConfigIndex:0
	053	0.08289461	- Entering DispatchPower: IRP_MN_WAIT_WAKE
	054	0.08292646	
The configured DeviceDescriptor	055	0.08293819	>>>>>>>>>>>>>>> DEVICE DESCRIPTOR <<<<<<<<<<<<<<<<<<
	056	0.08296026	18 0x01 0x0100 0x00 0x00 0x00 0x08 0x03F0 0x0101 0x0100 0x01 0x02 0x03 1
	057	0.08296892	
The configured ConfigurationDescriptors	058	0.08298150	>>>>>>>>>>>>>>> CONFIGURATION DESCRIPTOR [0] <<<<<<<<<<<<
	059	0.08299742	9 0x02 wTotalLength(39 Bytes) bNumInterfaces(0x01) 0x00 0x60 0x00
	060	0.08300608	
	061	0.08301502	
The current ConfigurationDescriptor and its interfaces and their endpoints	062	0.08302815	>>>>>>>>>>>>>>> CURRENT CONFIGURATION DESCRIPTOR [0] <<
	063	0.08304072	CurrentConfigurationHandle : 0X81E60ED0
	064	0.08305162	NumberTotalEndpoints : 3
	065	0.08306195	INTERFACE [0]
	066	0.08307452	- InterfaceHandle : 0X81DAEA08
	067	0.08308542	- InterfaceNumber : 0
	068	0.08309632	- Class : 0
	069	0.08310749	- SubClass : 0
	070	0.08311866	- Protocol : 0
	071	0.08312956	- AlternateSetting : 0
	072	0.08314046	* Claimed : 0
	073	0.08315135	ENDPOINT (Pipe) [0]
	074	0.08316252	- Address : 0X81
	075	0.08317510	- PipeHandle : 0X81DAEA24
	076	0.08318851	- Type : 0X2 (00:Control 01:Isochronous 02:Bulk 03:Interrupt)
	077	0.08319940	- Interval : 0
	078	0.08321058	- MaxPacketSize : 0X40
	079	0.08322231	- MaxTransferSize : 0XFFC0
	080	0.08323320	ENDPOINT (Pipe) [1]
	081	0.08324438	- Address : 0X2
	082	0.08325639	- PipeHandle : 0X81DAEA44
	083	0.08326980	- Type : 0X2 (00:Control 01:Isochronous 02:Bulk 03:Interrupt)
	084	0.08328098	- Interval : 0
	085	0.08329243	- MaxPacketSize : 0X10
	086	0.08330416	- MaxTransferSize : 0XFFC0
	087	0.08331506	ENDPOINT (Pipe) [2]
	088	0.08332623	- Address : 0X83
	089	0.08333852	- PipeHandle : 0X81DAEA64
	090	0.08335193	- Type : 0X3 (00:Control 01:Isochronous 02:Bulk 03:Interrupt)
	091	0.08336339	- Interval : 0XFA
	092	0.08337428	- MaxPacketSize : 0X1
	093	0.08338602	- MaxTransferSize : 0XFFC0
	094	0.08339496	
The content of the EndpointContext member	095	0.08340557	ENDPOINT CONTEXT [0]
	096	0.08341619	- PipeOpen : 0
	097	0.08342708	- PipeNumber : 1
	098	0.08343770	- PipeInterfaceNumber : 0
	099	0.08344999	- PipeDirection : 1 (0:HostToDevice 1:DeviceToHost)
	100	0.08346200	- PipeHandle : 81DAEA24
	101	0.08347066	
	102	0.08348100	ENDPOINT CONTEXT [1]

103	0.08349162	- PipeOpen : 0	
104	0.08350223	- PipeNumber : 2	
105	0.08351285	- PipeInterfaceNumber : 0	
106	0.08352514	- PipeDirection : 0 (0:HostToDevice 1:DeviceToHost)	
107	0.08353743	- PipeHandle : 81DAEA44	
108	0.08354609		
109	0.08355643	ENDPOINT CONTEXT [2]	
110	0.08356705	- PipeOpen : 0	
111	0.08357766	- PipeNumber : 3	
112	0.08358856	- PipeInterfaceNumber : 0	
113	0.08360085	- PipeDirection : 1 (0:HostToDevice 1:DeviceToHost)	
114	0.08361342	- PipeHandle : 81DAEA64	
115	0.08362404	NO ENDPOINT CONTEXT [3]	
116	0.08363465	NO ENDPOINT CONTEXT [4]	
…			
141	0.08389865	NO ENDPOINT CONTEXT [29]	
143	0.08437190	- ENTER DispatchPnp IRP:IRP_MN_QUERY_CAPABILITIES	
144	0.08452890	- ENTER DispatchPnp IRP:IRP_MN_QUERY_PNP_DEVICE_STATE	
145	0.08455376	- DispatchPnp IRP_MN_START_DEVICE) will be sent to the lower driver	
146	0.08457220	- ENTER DispatchPnp IRP:IRP_MN_QUERY_DEVICE_RELATIONS	
147	0.08461550	- DispatchPnp:(IRP_MN_START_DEVICE) will be sent to the lower driver	
148	5.08966503	- Entering DispatchPower: IRP_MN_SET_POWER	Device running
		At this point the JUSB Device is initialized and ready to handle request from the Java USB API	
		…	
		(Device is attached)	
		…	
		…	
		The following lines show what happens when the device is removed from the USB bus.	
149	11.50900799	- ENTER DispatchPower: DeviceObject 81D88D80,IRP_MN_SET_POWER	
150	11.50906666	- ENTER DispatchPower: DeviceObject 81D88D80, IRP_MN_WAIT_WAKE	
151	11.50908453	Lower drivers failed the wait-wake Irp	
152	11.50934127	- ENTER IdleNotificationRequestComplete	
153	11.50960443	- ENTER DispatchPnp IRP:IRP_MN_QUERY_DEVICE_RELATIONS	
154	11.50962762	- DispatchPnp(IRP_MN_START_DEVICE) will be sent to the lower driver	
155	11.50964215	- ENTER DispatchPnp IRP:IRP_MN_QUERY_DEVICE_RELATIONS	Surprise Removal
156	11.50971227	- DispatchPnp (IRP_MN_START_DEVICE) will be sent to the lower driver	
157	11.50973518	- ENTER DispatchPnp IRP:IRP_MN_SURPRISE_REMOVAL	
158	11.55099629	- ENTER DispatchPnp IRP:IRP_MN_REMOVE_DEVICE	
159	11.55100271		
160	11.55102227	- ENTER WmiDeRegistration	
161	11.55116363	- EXIT WmiDeRegistration ntStatus:	
162	11.70122935	- FREE MEMORY OF : EndpointContext[0]	Important: Free all allocated memory for DeviceExtension members
163	11.70124834	- FREE MEMORY OF : EndpointContext[1]	
164	11.70126036	- FREE MEMORY OF : EndpointContext[2]	
165	11.70127153	- FREE MEMORY OF : EndpointContext	
166	11.70128438	- FREE MEMORY OF : InterfaceList[0]	
167	11.70129556	- FREE MEMORY OF : InterfaceList	
168	11.70130813	- FREE MEMORY OF : InterfaceClaimedInfo[0]	
169	11.70131930	- FREE MEMORY OF : InterfaceClaimedInfo	
170	11.70133215	- FREE MEMORY OF : ConfigurationDescriptors[0]	
171	11.70134361	- FREE MEMORY OF : ConfigurationDescriptors	
172	11.70135478	- FREE MEMORY OF : DeviceDescriptor	
173	11.72126402	- ENTER DriverUnload	
174	11.72128050	- FREE registryPath->Buffer	
175	11.72129167	- EXIT DriverUnload	Unload Driver

Table 59: DdgView output of a device using the JUSB driver

Literature

Books:

[1] H.J Kelm (Hrsg.), 1999.
USB Universal Serial Bus, Franzis-Verlag , Poing.
ISBN 3-7723-7962-1

[2] John Hyde, 2001.
USB Design by Example 2nd Edition. Intel Press.
ISBN 0-9702846-5-9

[3] Rob Gordon, 1998.
Essential JNI: Java Native Interface, Prentice Hall PTR, New Jersey.
ISBN 0-13-679895-0

[4] Walter Oney, 2003.
Programming the Microsoft Windows Driver Model, 2nd Edition. Microsoft Press, Redmond, Washington.
ISBN 0-7356-1803-8

Internet:

[5] DisplayUSB.cpp:
http://www.intel.com/intelpress/usb/examples/DUSBVC.PDF

[6] Driver Development Kits (DDK):
http://www.microsoft.com/whdc/ddk/

[7] DumpBin:
http://h18009.www1.hp.com/fortran/docs/vf-html/pg/pg4exdmb.htm

[8] Java 2 Platform, Standard Edition (J2SE), Version 1.4.1:
http://java.sun.com/j2se/1.4.1/index.html

[9] Java 2 Platform, Standard Edition (J2SE):
http://java.sun.com/j2se/

[10] Java API (javadoc):
http://jusb.sourceforge.net/apidoc/overview-summary.html

[11] Java Book Online - German:
http://www.galileocomputing.de/openbook/javainsel2/

[12] Java Book Online - German:
http://www.javabuch.de/

[13] Java Native Interface How To Use:
http://www.acm.org/crossroads/xrds4-2/jni.html

[14] Java Native Interface Introduction:
http://www.javaworld.com/javaworld/jw-10-1999/jw-10-jni.html

[15] Java Native Interface Specification:
http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html

[16] Java Native Interface Tutorial:
http://java.sun.com/docs/books/tutorial/native1.1/

[17] Java Native Interface:
http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html

[18] Java USB Project Page:
http://jusb.sourceforge.net/

[19] MSDN Platform:
http://msdn.microsoft.com/library/

[20] MyPen:
https://entry2.credit-suisse.ch/cs/business/p/s/de/mypen.pdf

[21] Netbeans DIE:
http://www.netbeans.org

[22] New in DDK for Windows XP:
http://msdn.microsoft.com/library/en-us/dndevice/html/newinwinxpddk-.asp

[23] Paper++ :
http://www.paperplusplus.net/

[24] Platform Microsoft SDK 2003:
http://www.microsoft.com/msdownload/platformsdk/sdkupdate/

[25] Sysinternals (Debug Viewer):
http://www.sysinternals.com

[26] System-Supplied Device Setup Classes:
http://www.osr.com/ddk/install/setup-cls_2i1z.htm

[27] Universal Serial Bus Specification 2.0:
http://www.usb.org/developers/docs

[28] USB_ROOT_HUB_NAME structure:
http://www.osr.com/ddk/buses/usbstrct_1iya.htm

Index

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W