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

Watcher 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 *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

Naming Convention

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.

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!

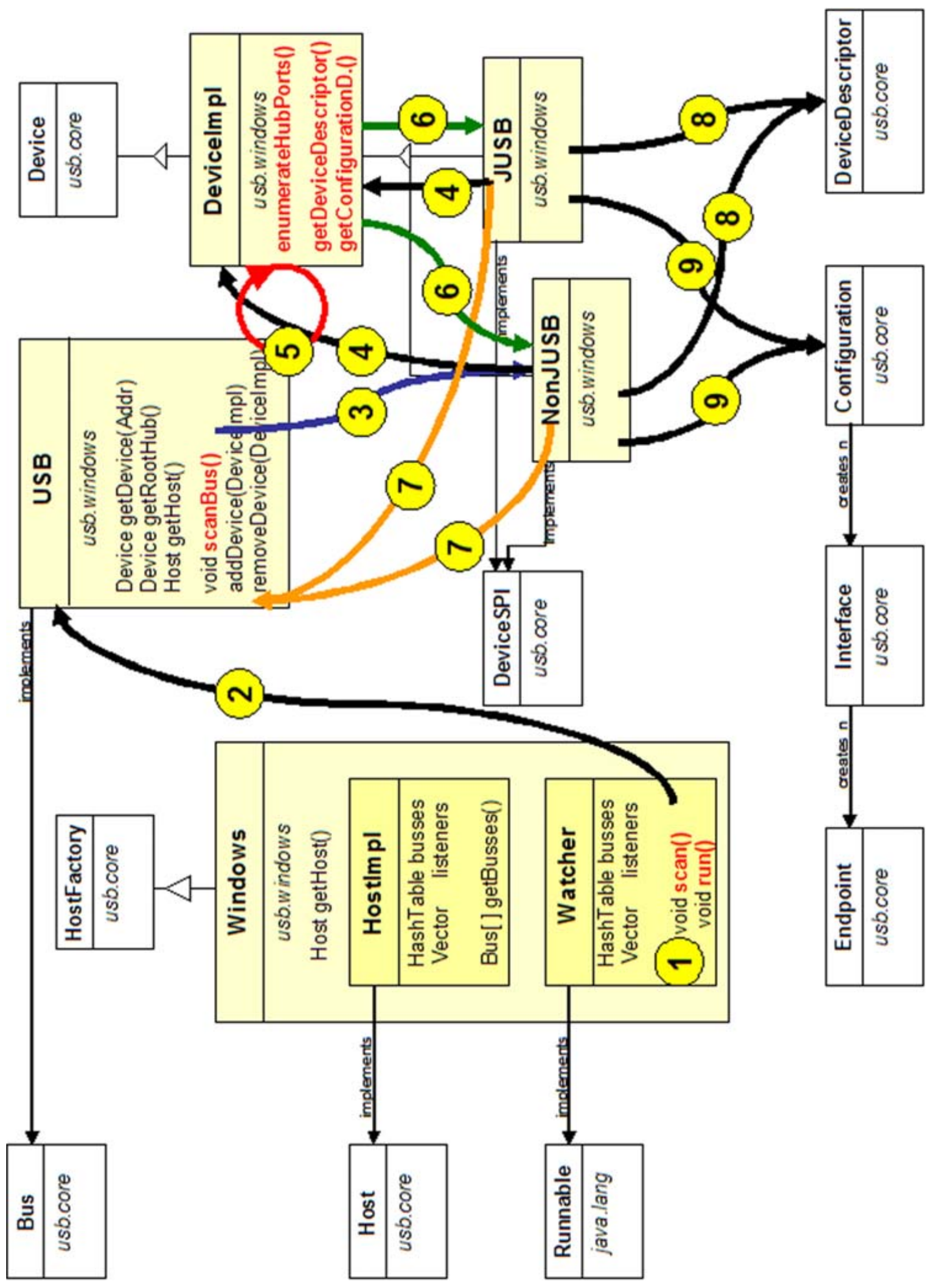


Figure 7: Class overview with its interaction

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.

Table 9: Additional method getBusNum in the USB class

Host Controller Name

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

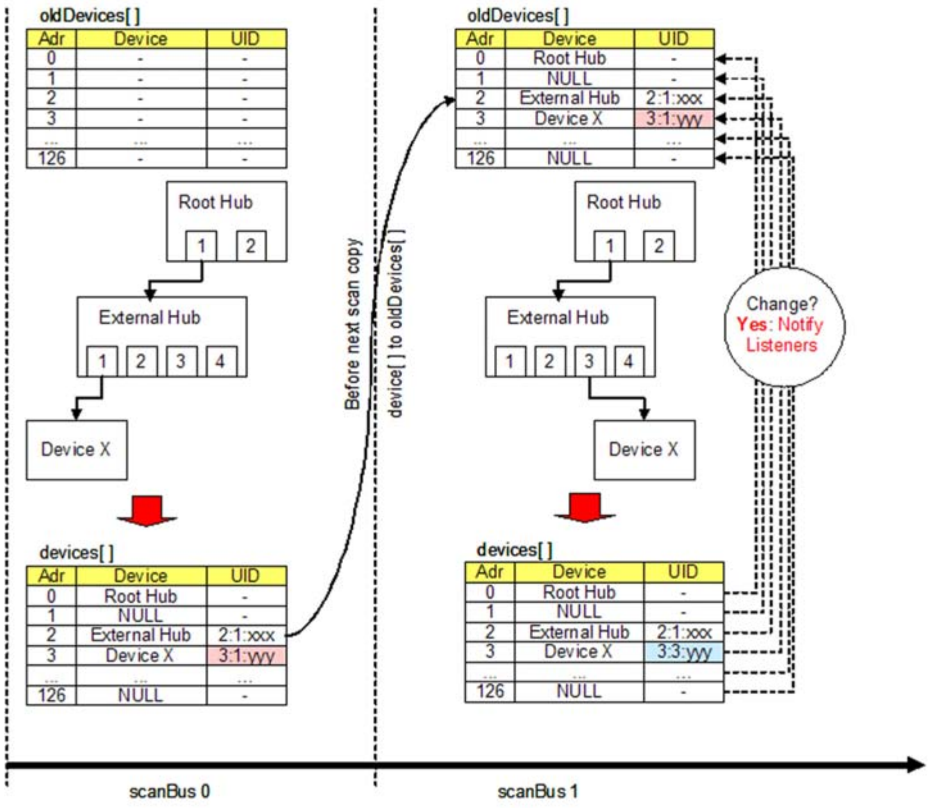
getUniqueDeviceID

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.



oldDevices[ ] (scanBus 0)

Adr	Device	UID
0	-	-
1	-	-
2	-	-
3	-	-
...	...	...
126	-	-

devices[ ] (scanBus 0)

Adr	Device	UID
0	Root Hub	-
1	NULL	-
2	External Hub	2:1:xxx
3	Device X	3:1:yyy
...	...	...
126	NULL	-

oldDevices[ ] (scanBus 1)

Adr	Device	UID
0	Root Hub	-
1	NULL	-
2	External Hub	2:1:xxx
3	Device X	3:1:yyy
...	...	...
126	NULL	-

devices[ ] (scanBus 1)

Adr	Device	UID
0	Root Hub	-
1	NULL	-
2	External Hub	2:1:xxx
3	Device X	3:3:yyy
...	...	...
126	NULL	-

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

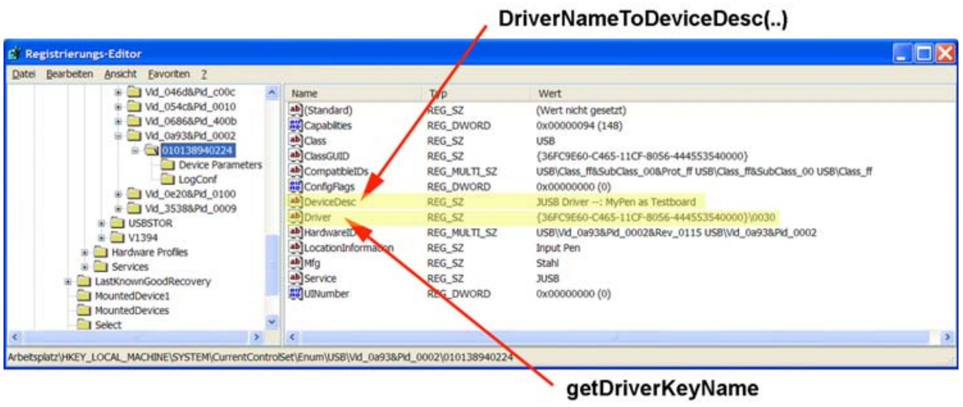
A handle for a device we get with the known device path and the Windows API function *CreateFile*.
This native function returns either a INVALID_HANDLE_VALUE by failure or a device handle by success. The INVALID_HANDLE_VALUE is defined in the error.h file from the Microsoft SDK [24].

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).


DriverNameToDeviceDesc(..)
getDriverKeyName
Registrierungs-Editor
Datei Bearbeiten Ansicht Favoriten ?

```
Vid_046d&Pid_c00c
Vid_054c&Pid_0010
Vid_0686&Pid_400b
Vid_0a93&Pid_0002
    010138940224
        Device Parameters
        LogConf
Vid_0e20&Pid_0100
Vid_3538&Pid_0009
USBSTOR
V1394
Hardware Profiles
Services
LastKnownGoodRecovery
MountedDevice1
MountedDevices
Select
```

Name	Typ	Wert
(Standard)	REG_SZ	(Wert nicht gesetzt)
Capabilities	REG_DWORD	0x00000094 (148)
Class	REG_SZ	USB
ClassGUID	REG_SZ	{36FC9E60-C465-11CF-8056-444553540000}
CompatibleIDs	REG_MULTI_SZ	USB\Class_ff&SubClass_00&Prot_ff USB\Class_ff&SubClass_00 USB\Class_ff
ConfigFlags	REG_DWORD	0x00000000 (0)
DeviceDesc	REG_SZ	JUSB Driver --: MyPen as Testboard
Driver	REG_SZ	{36FC9E60-C465-11CF-8056-444553540000}\0030
HardwareID	REG_MULTI_SZ	USB\Vid_0a93&Pid_0002&Rev_0115 USB\Vid_0a93&Pid_0002
LocationInformation	REG_SZ	Input Pen
Mfg	REG_SZ	Stahl
Service	REG_SZ	JUSB
UINumber	REG_DWORD	0x00000000 (0)

Arbeitsplatz\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\Vid_0a93&Pid_0002\010138940224

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:

```
typedef struct _NODE_INFORMATION{
            USB_HUB_NODE NodeType;
1           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 *read-Contol* 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

guidgen

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).

```
\??\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.

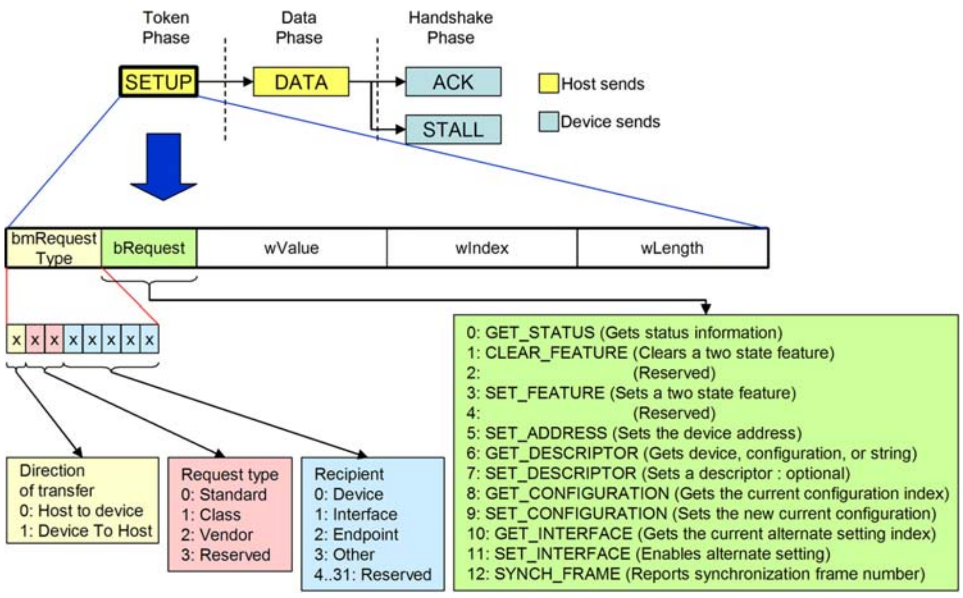
```
JNIEXPORT jstring JNICALL Java_usb_windows_JUSB_getDevicePath
(JNIEnv *env, jobject obj, jstring pidAndVid){
…
  PCHAR deviceIdentity = (PCHAR)env->GetStringUTFChars(pidAndVid, 0);
…
  while(!found){
    devPath =
        getDevicePath((LPGUID)&GUID_DEFINTERFACE_JUSB_DEVICES, i);

    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



Token Phase
Data Phase
Handshake Phase
SETUP
DATA
ACK
STALL
Host sends
Device sends
bmRequest Type
bRequest
wValue
wIndex
wLength
x x x x x x x x
Direction of transfer
0: Host to device
1: Device To Host
Request type
0: Standard
1: Class
2: Vendor
3: Reserved
Recipient
0: Device
1: Interface
2: Endpoint
3: Other
4..31: Reserved
0: GET_STATUS (Gets status information)
1: CLEAR_FEATURE (Clears a two state feature)
2: (Reserved)
3: SET_FEATURE (Sets a two state feature)
4: (Reserved)
5: SET_ADDRESS (Sets the device address)
6: GET_DESCRIPTOR (Gets device, configuration, or string)
7: SET_DESCRIPTOR (Sets a descriptor : optional)
8: GET_CONFIGURATION (Gets the current configuration index)
9: SET_CONFIGURATION (Sets the new current configuration)
10: GET_INTERFACE (Gets the current alternate setting index)
11: SET_INTERFACE (Enables alternate setting)
12: SYNCH_FRAME (Reports synchronization frame number)

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.