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.

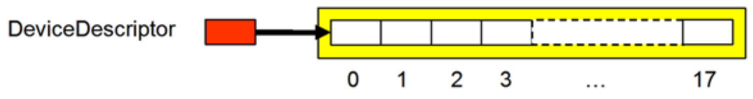

Figure 11: DeviceDescriptor memory allocation (yellow: allocated memory)

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)

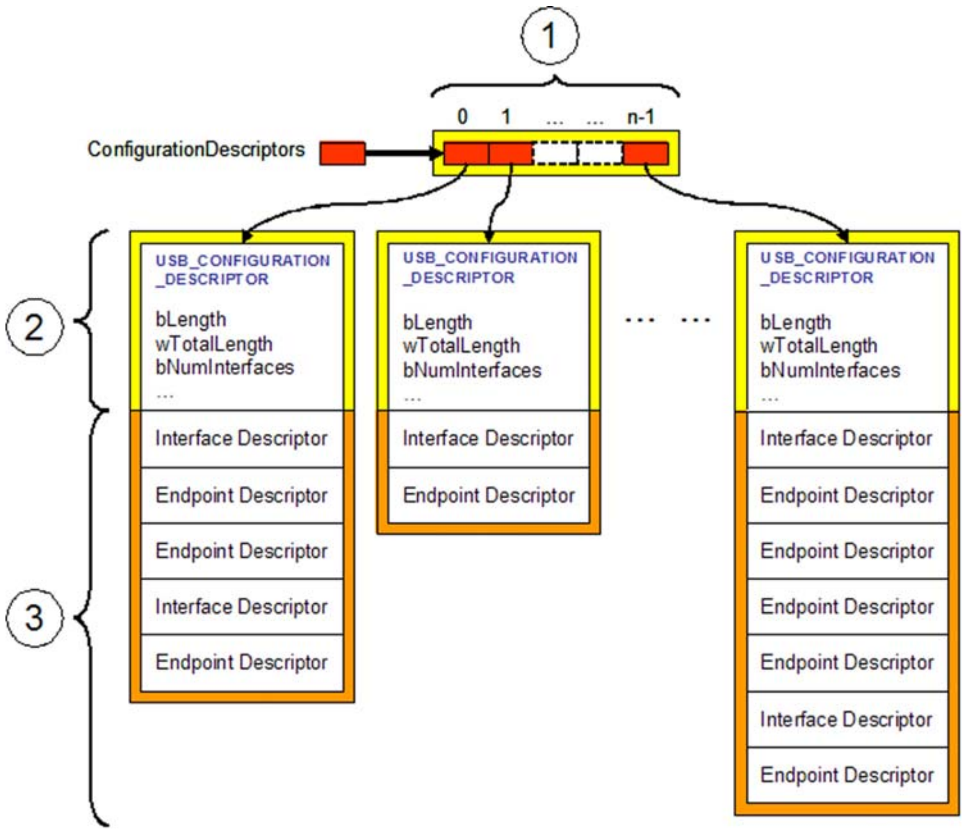

Figure 12: ConfigurationDescriptors memory allocation structure (yellow: allocated memory, orange: additional allocated memory)

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.

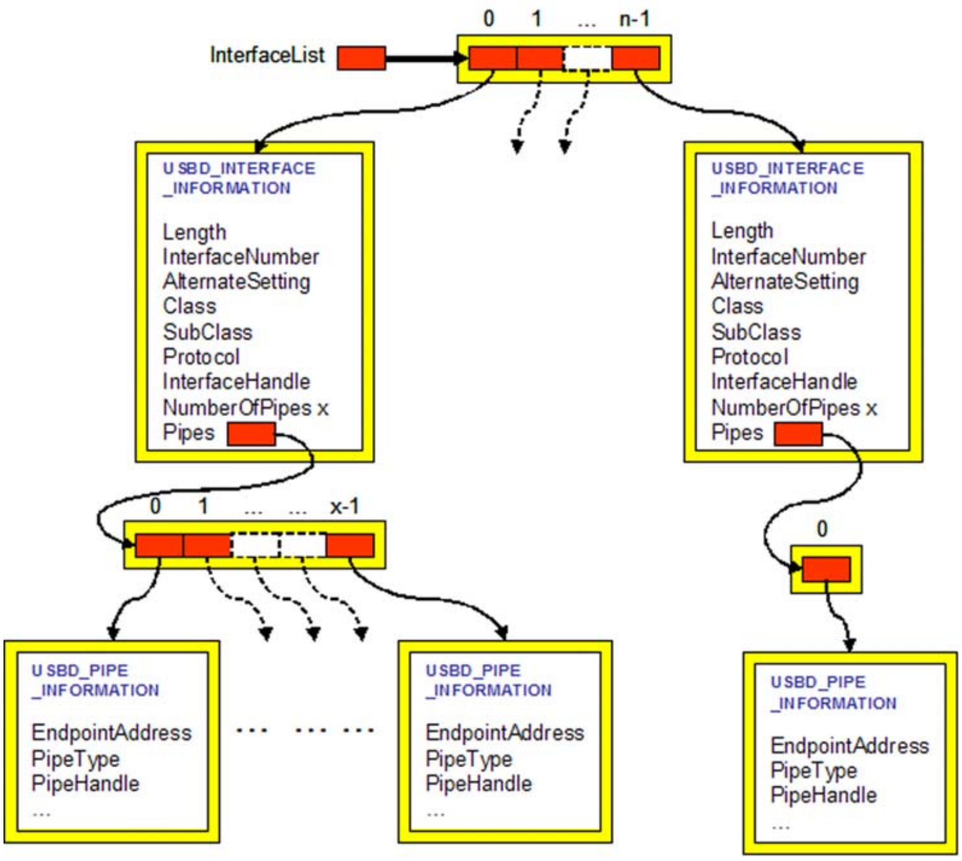

Figure 13: InterfaceList memory allocation structure (yellow: allocated memory)

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.

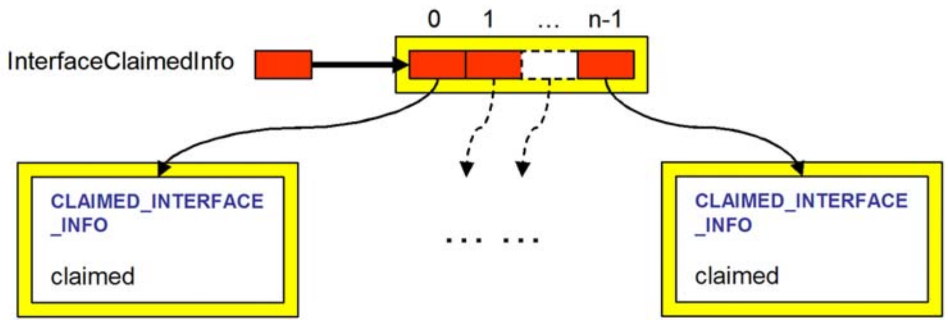

Figure 14: InterfaceClaimedInfo memory allocation structure (yellow: allocated memory)

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.

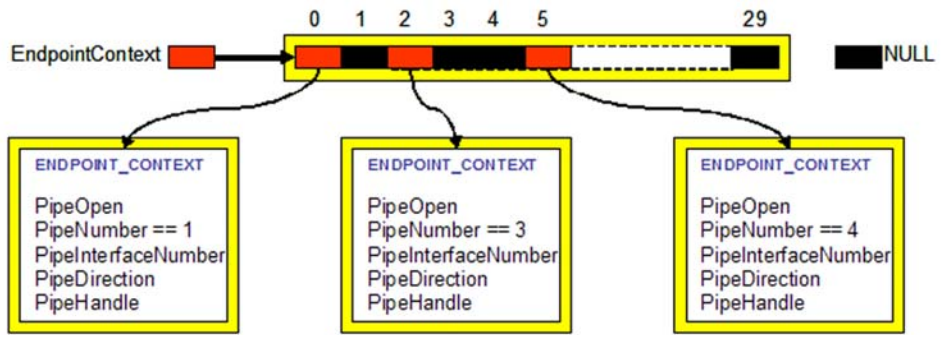

Figure 15:EndpointContext memory allocation structure (yellow: allocated memory)

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:

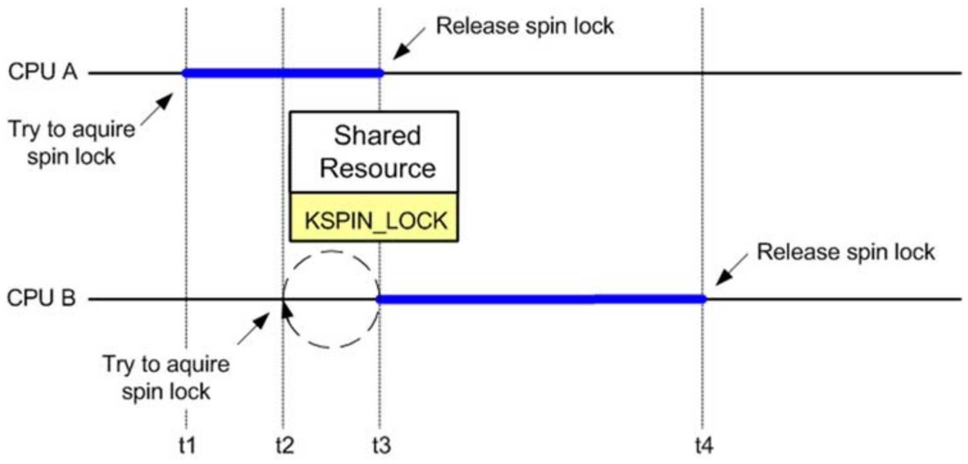

Figure 16: Using a spin lock to guard a shared resource

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.

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).

Spin Lock Initialization

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.

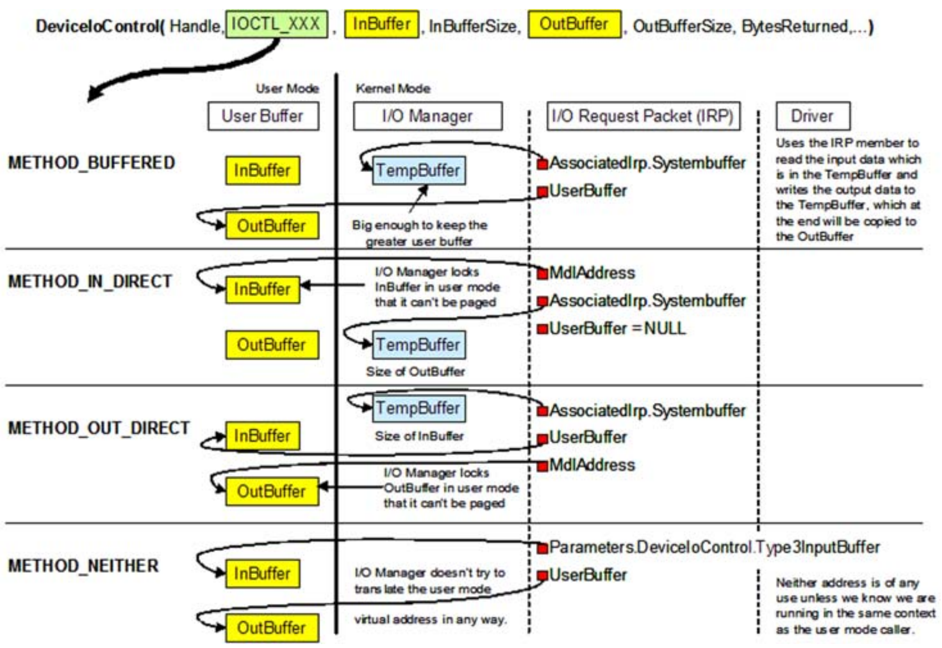
DeviceIoControl(Handle, IOCTL_XXX , InBuffer , InBufferSize, OutBuffer , OutBufferSize, BytesReturned,...)
User Mode
Kernel Mode
User Buffer
I/O Manager
I/O Request Packet (IRP)
Driver
METHOD_BUFFERED
InBuffer
TempBuffer
AssociatedIrp.Systembuffer
UserBuffer
OutBuffer
Big enough to keep the greater user buffer
Uses the IRP member to read the input data which is in the TempBuffer and writes the output data to the TempBuffer, which at the end will be copied to the OutBuffer
METHOD_IN_DIRECT
I/O Manager locks InBuffer in user mode that it can't be paged
MdlAddress
AssociatedIrp.Systembuffer
UserBuffer =NULL
Size of OutBuffer
METHOD_OUT_DIRECT
Size of InBuffer
I/O Manager locks OutBuffer in user mode that it can't be paged
METHOD_NEITHER
Parameters.DeviceIoControl.Type3InputBuffer
I/O Manager doesn't try to translate the user mode virtual address in any way.
Neither address is of any use unless we know we are running in the same context as the user mode caller.

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.

```
UsbBuildInterruptOrBulkTransferRequest(
        &urb,
        sizeof(struct _URB_BULK_OR_INTERRUPT_TRANSFER),
        deviceExtension->EndpointContext[pipeNum]->PipeHandle,
        ioBuffer,
        NULL,
        inputBufferLength,
        transferFlag | USBD_SHORT_TRANSFER_OK,
        NULL
);

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.