Design and Implementation of a USB Stack for the Java-based JX Operating System

Bachelor Thesis in Computational Engineering

vorgelegt von

Dreweke Alexander

geb. 08.01.1980 in Ingolstadt

angefertigt am

Institut für Informatik
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen–Nürnberg
(Prof. Dr. Wolfgang Schröder-Preikschat)

Betreuer: Dipl-Inf. Meik Felser

Beginn der Arbeit: 01.07.2003
Abgabe der Arbeit: 03.11.2003

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Dreweke Alexander
Erlangen, den 3. November 2003

Acknowledgement

Thanks to all the numerous assistants at the Friedrich-Alexander University for being so patient with me and answering all my questions. And god knows there where and still are a lot of them.
Especially Meike Felser for being the patient of all.

Thanks to Michael Klemm for helping me getting familiar with TeX and LaTeX, for giving tips and tricks to me, that made writing this bachelor thesis much easier and for proofreading my stuff.

Thanks to my parents, Edeltraud and Klaus, for making me what I am, for setting high goals and expectations, for making my computational engineering and computer science study possible and for supporting me in so many different ways.

Last but not least thanks to Sabine Guth, for building me up again every time study, this bachelor thesis or life brought me down. For being always there when I needed someone to talk or lean on and for being a friend. For helping to solve all of my problems and also for sometimes being the biggest problem of all.

Abstract

This Bachelor Thesis targets to create a general design of an USB stack for the use on USB devices through the JX operating system. At the time of the thesis USB 2.0 devices are not so widely used at all, so the work concentrated on USB 1.0/ 1.1 devices. Therefor the Java-based operating system JX will be extended for the general integration of USB through Universal-, Open- or Enhanced-Host Controller Interfaces. Afterwards the general capabilities are available all layers of the USB stack for the use of low-speed and full-speed USB devices (USB 1.0/1.1 specification) at a Intel 82801DB-ICH4 USB Controller with Universal Host Controller Interface (UHCI Controller) will be implemented. The work mainly concentrates on UHCI on Intel architecture but the interface design is made in a way it can easily be extended to also support OHCI and EHCI on different architectures.

Ziel dieser Bachelor Arbeit ist es das generelle Design eines USB Stacks zum Betrieb von USB Geräten durch das JX Betriebssystem zu entwerfen. Zum jetztigen Zeitpunkt sind USB 2.0 Geräte noch nicht sonderlich weit verbreitet. Aus diesem Grund konzentriert sich die Arbeit hauptsächlich auf USB 1.0/1.1 Geräte. Zu diesem Zweck musste das java-basierte JX-Betriebssystem um die generellen Integration von USB durch Universal-, Open- oder Enhanced-Host Controller Schnittstellen erweitert werden. Nachdem die grundlegenden Voraussetzungen geschaffen sind, müssen alle weiteren Schichten eines USB-Stacks zum Betrieb von Low-speed and Full-speed USB Geräten (USB-1.0/1.1-Spezifikation) durch einen Intel 82801DB-ICH4 USB Controller mit Universal Host Controller Schnittstelle (UHCI-Controller) implementiert werden. Die Arbeitet richtete ihr Hauptaugenmerk auf die UHCI Schnittstelle auf Intel Architektur. Alle Schnittstellen wurden aber derart gestaltet, dass sie einfach um OHCI und EHCI auf verschiedensten Architekturen erweitert werden können.

Contents

Acknowledgement **v**

Abstract **vii**

1 Introduction **1**

2 USB **3**

2.1 Introduction . . . 3
2.2 USB Host . . . 3
2.3 USB Device . . . 4
2.4 Attachment/Removal of USB Devices . . . 4
2.5 Power Distribution . . . 4
2.6 USB 1.0/1.1 Controller . . . 4
2.7 Bus Protocol . . . 5
2.8 Pipes . . . 6
2.9 Data Flow Types . . . 7
2.10 Summary . . . 8

3 Universal Host Controller Interface **9**

3.1 Introduction . . . 9
3.2 Overview . . . 9
3.3 Data Structures . . . 10
3.3.1 Frame List . . . 10
3.3.2 Frame List Pointer . . . 11
3.3.3 Queue Header . . . 12
3.3.4 Transfer Descriptor . . . 12
3.4 Mode of Operation . . . 13
3.5 Summary . . . 15

4 Design and Implementation **17**

4.1 Introduction . . . 17
4.2 USBDevice Interface . . . 18
4.2.1 USBDevice . . . 18

4.2.2 USBFullSpeedDevice 19
4.2.3 USBHighSpeedDevice 19
4.2.4 Device Drivers 20
4.3 USBTransfer 20
4.4 USBQueueManager 21
4.5 USBController 22
4.5.1 FrameListCreater 22
4.5.2 FrameListCollector 24
4.6 Summary 25

5 Design Comparison **27**
5.1 Introduction 27
5.2 JX 27
5.3 JX vs. Linux 27
5.4 JX vs. LibUSB 28
5.5 Summary 29

6 Summary and Future prospects **31**

List of Figures

2.1 USB Bus. 3
2.2 Broadcast from Host to Functions. 5
2.3 Unicast from Function to Host. 6
2.4 Parts of a Transaction. 6
2.5 User view of Host to Function communication via Pipes. 7

3.1 UHCI Frame. 10
3.2 UHCI Communication Date Structures. 10
3.3 UHCI Start-Of-Frame. 11
3.4 UHCI Frame List Pointer. 11
3.5 UHCI Queue Header. 12
3.6 UHCI Transfer Descriptor. 14
3.7 Processing of data structures. 15

4.1 Design overview. 17
4.2 USBDevice class diagram. 18
4.3 USBFullSpeedDevice class diagram. 19
4.4 USBTransfer class diagram. 20
4.5 USBQueueManager class diagram. 21
4.6 USBController class diagram. 22
4.7 FrameListCreater - FrameListCollector interaction. 23
4.8 FrameListCreater class diagram. 23
4.9 FrameListCollector class diagram. 24

5.1 Various configurations of the USBStack. 28

1 Introduction

One of the main tasks of modern operating systems is to provide an interface to applications and the user to interact with peripheral devices. In this area USB is getting more and more important. Because of its special properties, like hot-plugging and the possibility to simply enlarge the number of attached devices (to just call some of the advancements for the user), USB is more and more replacing serial and parallel interfaces.

The USB protocol has not only advantages for the user but also for programmers: The possibility of sharing interrupts and not being depend on special I/O-address makes driver development much easier.

At the moment there are only two different types of controllers supported by USB 1.0/1.1, namely Universal Host Controller Interface (UHCI) and Open Host Controller Interface (OHCI), and for the USB 2.0 specification only the Enhanced Host Controller Interface (EHCI). Most of the current USB devices fully support USB 1.0/1.1, USB 2.0 is not so widely used at the time of writing. So the main focus lays on extending the JX to support USB 1.0/1.1 functionally. USB 2.0 is fully backwards compatible to version 1.0 and 1.1 of the protocol specification. Therefore USB 1.0 and 1.1 devices can always be attached and used with EHCI controllers.

On the Intel architecture only UHCI USB controllers can be found. To be able to support the most widespread architecture today the Universal Host Controller Interface was implemented as part of this work. Nevertheless, the USB design has been made so clean that with a minimum amount of time it can be extended to support also OHCI and EHCI USB controllers.

In the next chapter, the USB protocol will be discussed, followed by a more in depth insight of the Universal Host Controller Interface. After the chapters about the USB protocol and the UHCI specification, the USB design and the implementation for the JX operating system will be discussed.

2 USB

2.1 Introduction

This chapter gives a short overview over the most important aspects of the USB 1.0/1.1 bus architecture as far as it concerns this thesis. For a more complete and detailed description please see [Com98].

2.2 USB Host

The USB interface to the host computer system is called the Host Controller. It may be implemented in a combination of hardware, firmware, and software. A root HUB is integrated within the host system to provide one or more attachment points for other HUBs or functions. Figure 2.1 illustrates the USB bus topology.

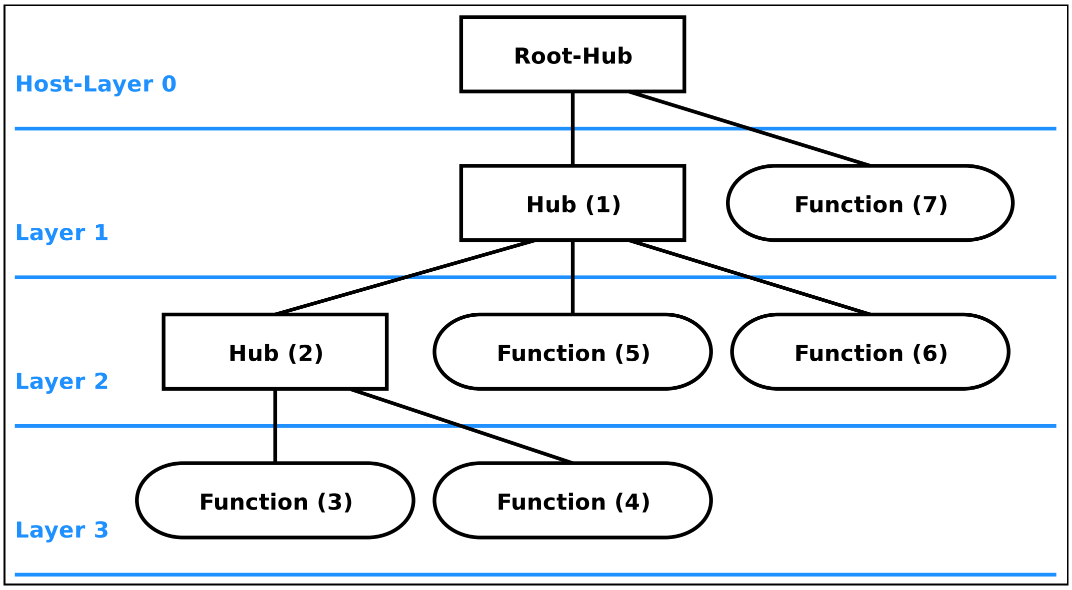

Figure 2.1: USB Bus.

2.3 USB Device

There are two device types that are distinguished. Firstly, there are HUBs, which provide additional attachmentpoints to the USB bus, and secondly, there are functions which provide capabilities to the system, such as a mouse, keyboard, data-stick, ... Devices are separated in low-speed- and full-speed classes which differ in the type of communication pipes they offer. Low-speed devices just provide control and interrupt transfer pipes. Full-speed devices provide additionally bulk and isochronous transfer pipes. HUBs are always full-speed-devices. USB functions can be low-speed- (like mice, keyboards, ...) or full-speed-devices (like data-sticks, isdn-systems, ...). Full-speed-devices have larger buffers compared to low-speed-devices. Thus more data can be transferred in the same amount of time.

2.4 Attachment/Removal of USB Devices

All USB devices attach to the USB through ports on specialized USB devices known as HUBs. HUBs have status indicators that indicate the attachment or removal of a USB device on one of its ports. The host queries the HUBs to retrieve these indicators. In the case of an attachment, the host enables the port and enumerates the device through the device's control pipe. If the attached USB device is a functional device, attachment notifications will be handled by host software that is appropriate for the function. When a USB device has been removed from one of a HUB's ports, the HUB disables the port and provides an indication of the device removal to the host. The removal indication is then handled by appropriate the USB system software.

2.5 Power Distribution

The host supplies power for USB devices that are directly connected to it. In addition, any USB device may have its own power supply. USB devices that totally rely on power from the cable are called bus-powered devices. In contrast, those that have an additional power supply are called self-powered devices. A HUB can only additional supply power to its connected USB devices if it is self-powered. The host controls the amount of energy that is supplied to each function.

2.6 USB 1.0/1.1 Controller

There are two different types of USB 1.0/1.1 controllers. On the one hand there are controllers which implement the Universal Host Controller Interface (UHCI), developed and propagated by Intel. On the other hand Open Host Controller Interface (OHCI) was developed by Compaq. Both controllers offer the same interfaces for the programmer and can communicate with the same devices. They only differ in their internals. They process different internal data structures, and the data order during the transfer differs from controller type to controller type.

2.7 Bus Protocol

The whole communication over the bus is separated in frames. A frame has a fixed length of one millisecond and is initialized by a Start-Of-Frame packet (SOF). After this SOF packet all other transactions (there are always multiple transactions per frame) take place until the end of the frame is reached. If there isn't enough data to fill the frame the host spends the remaining frame time idle. All bus transactions involve the transmission of three packets (see figure 2.4. All packets from the host are broadcasted to all functions on the bus (see figure 2.2) whereas each packet sent by a function is unicasted to the host (see figure 2.3).
Each transaction begins with a so called token packet, describing the type and direction of the transaction, the USB device-address, and endpoint number. The USB device that is addressed selects itself by decoding the appropriate address fields. In a given transaction, data is transferred either from the host to a device or from a device to the host. The direction of data transfer is specified in the token packet. After the data has been sent the transaction ends with a handshake packet indicating whether the transfer was successful or not. All these different packets are generated from transfer descriptors by the USB controller hardware.

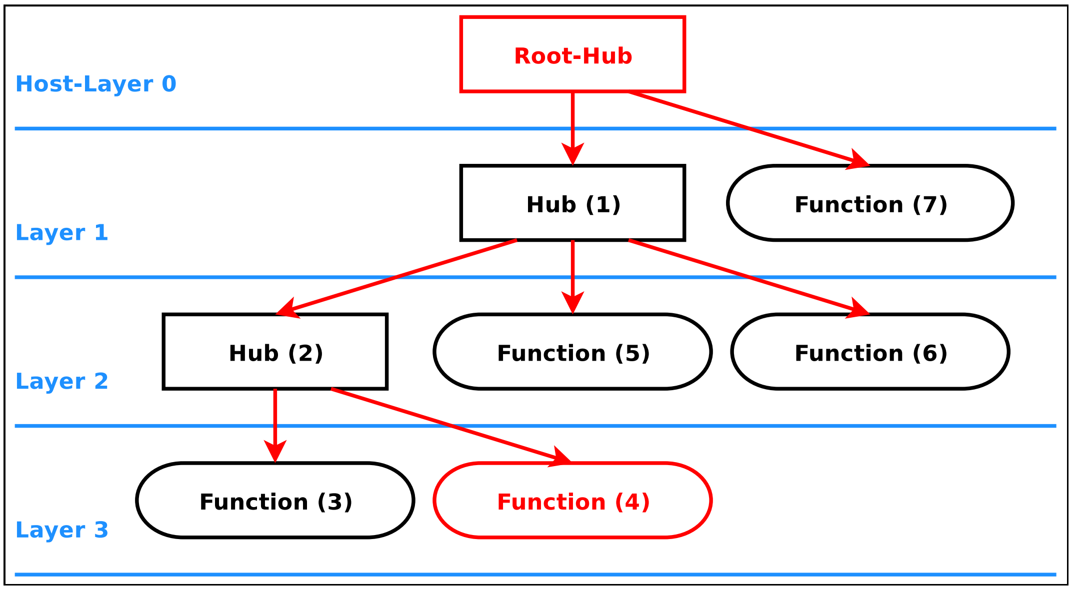

Figure 2.2: Broadcast from Host to Functions.

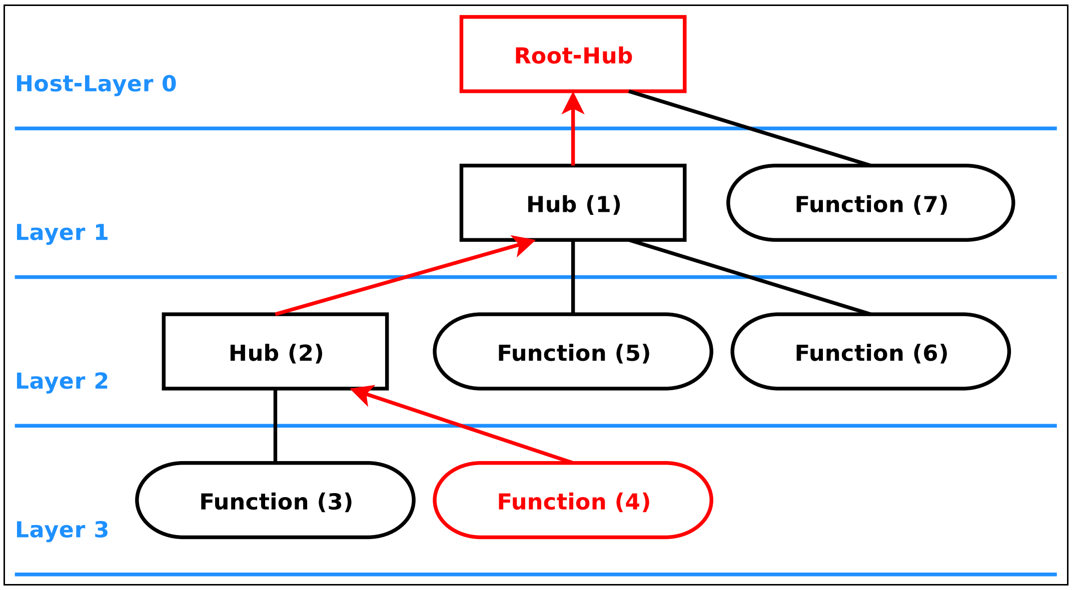

Figure 2.3: Unicast from Function to Host.

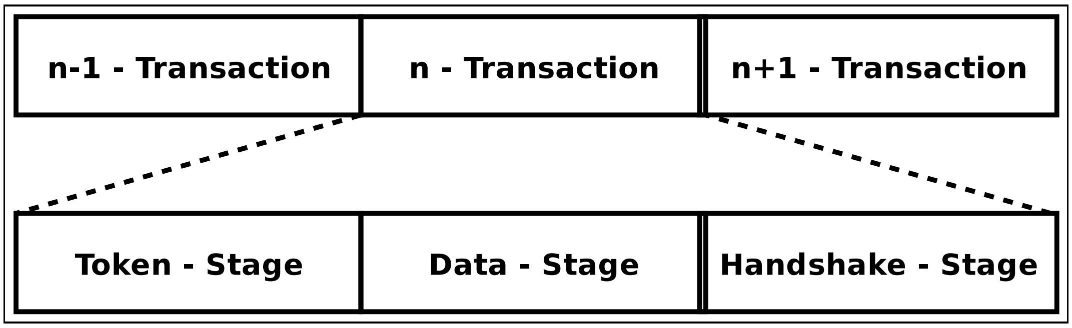

Figure 2.4: Parts of a Transaction.

2.8 Pipes

The USB data transfer model between the host and an endpoint on a device is called a pipe. There are two types of pipes: stream and message. Stream data has no USB defined structure, while message data does. Pipes have bandwidth, transfer service type, and endpoint characteristics like directionality and buffer sizes associated with them. Before the USB device isn't totally configured only the default control pipe exists, in order to provide access to the devices configuration, status, and control information. Also, device with all possible pipes can be seen in figure 2.5.

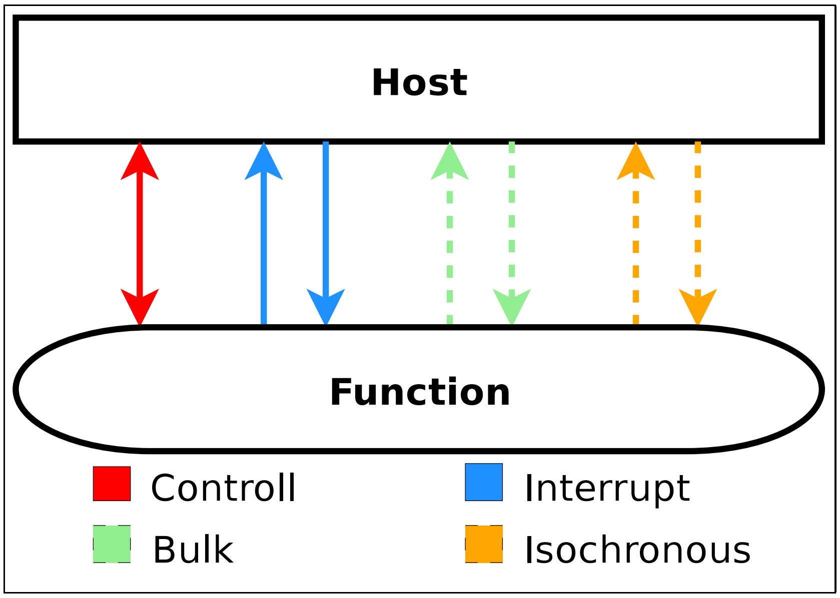

Figure 2.5: User view of Host to Function communication via Pipes.

2.9 Data Flow Types

USB devices support uni-directional or bi-directional pipes. USB data transfers take place between host software and a particular endpoint on a USB device. In general, data movement through one pipe is independent from the data flow in any other pipe. A given USB device may have more than one pipe. The USB architecture describes four basic types of data transfers:

Control Transfer: Control data is used by the USB System Software to configure devices when they are first attached to the bus. Other driver software can choose to use control transfers in vendor-specific ways. Data delivery is lossless.

Interrupt Transfer: A small, limited-latency transfer of data to or from a device is done via the interrupt transfer pipe. Interrupt data consists of event notification, characters, or coordinates that are organized as one or more bytes.

Bulk Transfer: Bulk data consists of larger amounts of data, such as that used for printers or scanners. Bulk data is sequential. Reliable exchange of data is ensured at the hardware level by using error detection in hardware and invoking a limited number of retries in hardware.

Isochronous Transfer: Isochronous data is continuous and real-time. Isochronous data must be delivered at the rate received to maintain its timing. In addition to delivery rate, isochronous data may also be sensitive to delivery delays.

2.10 Summary

USB offers a clear structured and simple way of communicating with peripheral devices. Through its hot-plugin capapility it suits for devices that are not always connected to the pc or not from the booting on like digital camcorders and mp3 players. Smaller devices with a low power consumption can be attached without the need for additonal power cables because their power requirements can be fulfilled over the USB bus. This brings more comfort to the end user and is therefor one of the reasons why USB has become so popular. With the four data flow types, control, interrupt, isochronous and bulk transfer, the various requirements of different devices like mice, keyboards, ... but also external hard discs can be fulfilled.

3 Universal Host Controller Interface

3.1 Introduction

This chapter introduces all data structures needed for a successful communication between the UHCI host controller and a function as far as they can be controlled by the host controller driver. All data structures generated and controlled by the hardware are not object of this chapter (See [Com98] for information about the hardware generated data structures).

3.2 Overview

As mentioned in chapter 2 the USB specification [Com98] separates the USB bus into frames, of length one millisecond each and are started by a SOF-packet. The structure of the different transfer types in such frames (see figure 3.1) is given by the UHCI specification [Int96]. After the SOF-packet the isochronous transfers are sent followed by the interrupt transfers. According to the USB specification [Com98], these two transfer types achieve at most 90 percent of the total bandwidth. After the interrupt transfers the control transfers follows.
At least 10 percent of the bandwidth is reserved for the control transfers to guarantee that configuration of the attached USB devices can be assured. The rest of the bandwidth is used for bulk transfers. If there are not enough transfers to completely fill a frame the remaining time is spent idle.
It is in the responsibility of the host controller driver to achieve the correct order of the different transfer types. Therefor a data structure (see figure 3.2) must be created in main memory. Interrupt-, control and bulk transfers begin with a Queue Header. This data structure stores control data needed for the correct processing by the hardware. A linked list consisting of Queue Headers is build, whereas each Queue Header represents a single transaction. The list of Transfer Descriptors which is linked by each Queue Header represents the transactions itself. For the isochronous transfer no Queue Headers are used. Instead all the Transfer Descriptors of all isochronous transfers in this frame are linked together.

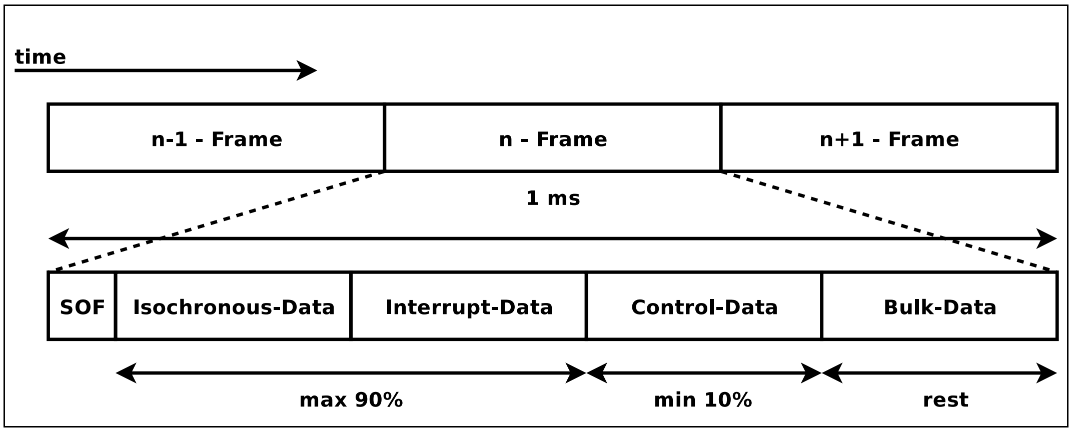

Figure 3.1: UHCI Frame.

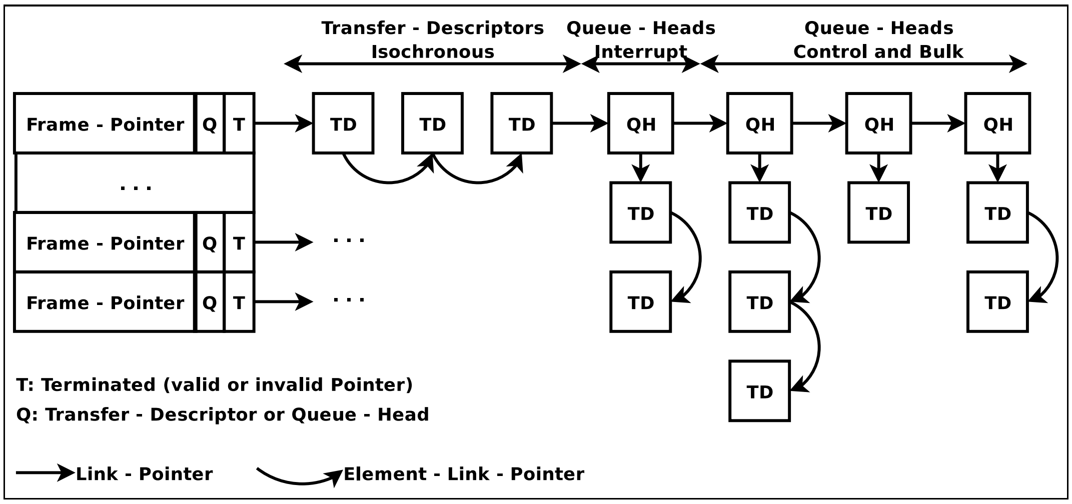

Figure 3.2: UHCI Communication Date Structures.

3.3 Data Structures

3.3.1 Frame List

During initialization, a list of 1024 Frame List Pointers has to be allocated in main memory. Each Frame List Pointer stores the start address of the data structures that represent the data which will be transferred during a frame (see figure 3.2). The current entry in the frame list is determined by the Start-Of-Frame counter (see figure 3.3) which is responsible for generating the Start-Of-Frame packet each millisecond (signals the beginning of a new frame). It gets decremented by a 12 MHz clock until it reaches zero, taking into account the Start-Of-Frame modify register which provides an adjusted starting value for the Start-Of-Frame counter. Every time the Start-Of-Frame counter is decremented to zero a new frame is generated and therefor the frame counter gets incremented. The frame list base address register contains the base address of the whole frame list. The combination of the frame counter and the frame list base register results in the address of the current processed frame list pointer.

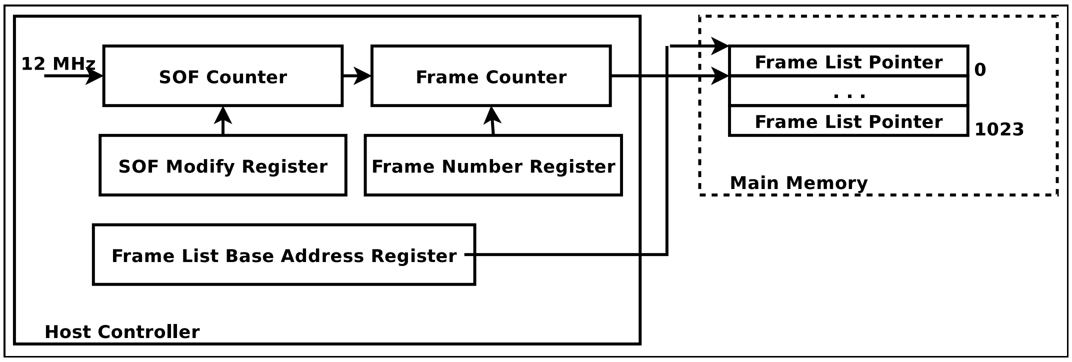

Figure 3.3: UHCI Start-Of-Frame.

3.3.2 Frame List Pointer

A Frame List Pointer (see figure 3.4) consists of three fields. The address field stores the memory address of the first data object that will be processed by the hardware during this frame. This must be a Transfer Descriptor or a Queue Header. To distinguish this two kinds of data structures the Q-field of the Frame List Pointer is used. It is set if the following data structure is the Queue Header. If the host controller driver was not able to generate a correct frame data structure before it is processed by the hardware or if there are no transactions left, the frame gets not marked as valid, by the host controller driver by setting the T-field to zero. This causes the hardware to skip this frame and to immediately process the next frame. All transactions, beside the isochronous transfers, are realized as a linked list of Queue Headers (each Queue Header symbolizes one transaction) in the main memory. The individually parts of each transaction are attached as a linked list of Transfer Descriptors to each Queue Header. This is true for interrupt, control and bulk transfers. Whereas, the isochronous transfers whereas are an exception to this strict composition, they are a linked list of all the needed Transfer Descriptors. If there are no isochronous transfers during a frame the Frame List Pointer would have to point to a Queue Header of one of the other transfer-types.

Figure 3.4: UHCI Frame List Pointer.

3.3.3 Queue Header

Queue Headers as shown in figure 3.5, are used to separate the individual transactions from each other. They consist of two main parts a queue head part and a queue element part. First, the queue head field is used to generate the linked list of the Queue Headers. The end of this list is signalled to the hardware by marking the memory address in the address field invalid through the T-field. This corresponds to the way a Frame List Pointer gets marked inactive. The Q-field which is also available in the queue head region must always be set to one, because after a control, interrupt or bulk transfer is reached only transfer types that make use of Queue Headers will follow. The only transfer type which does not make use of Queue Headers is the isochronous transfer and these transfers have to be always at the beginning of a frame.
The second part of a Queue Header is used to point to the next Transfer Descriptor, that belongs to this transfer. After processing this Transfer Descriptor, the hardware updates this part of the Queue Header to point to the next Transfer Descriptor of this transfer. This part of the Queue Header is structured the same way as the queue head part, but since the queue element part only points to next Transfer Descriptor (see [Int96]) its Q-field must always be zero.

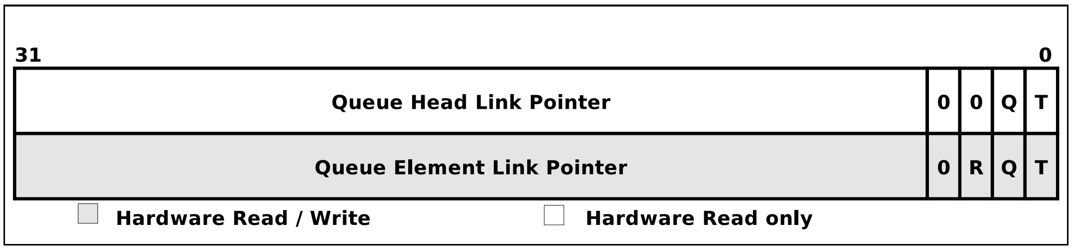

Figure 3.5: UHCI Queue Header.

3.3.4 Transfer Descriptor

Transfer Descriptors (figure 3.6) represent a transaction between the host controller and a function. They can be separated into four parts. The first part, the so called link pointer, is used to generate a linked list of several Transfer Descriptors which belong together. Therefor it consists of an address field and a Q-field that determines the type of the element the address-field points to. If a Transfer Descriptor is the last one in a interrupt, control or bulk transfer transaction the address-field gets marked invalid by setting the T-field. Beside these fields there is the Vf-field. This field determines the next Transfer Descriptor, after processing the current Transfer Descriptor and updating the corresponding Queue Header, that will be processed whether of the current transaction (depth-first). Or whether the next available Queue Header is processed instead (breath-first).

The next part is the so called control section. It gives the hardware additional information about the function, with whom the communication will take place (if the function is a low-speed device the LS-field has to be set), and about how to react on certain events during the communication. The host controller driver can be told after how many errors the transaction gets marked as invalid through the ERR-field. Via the IOC-field, whether there should be an interrupt at the end of the whole frame in which this Transfer Descriptor was processed or not, can be controlled. The hardware can determine through the ISO-field if the current Transfer Descriptor is part of an isochronous transfer or not. During processing the host controller gives the host controller driver feedback about the status of the transfer through the STATUS-field. This field tells the host controller driver if the host controller has already started processing the current Transfer Descriptor, or if the processing was stopped or canceled because of errors that occurred. Beside the STATUS-field the ACTLEN-field tells the host controller driver how many bytes of the Transfer Descriptor have already been processed by the hardware.
The third part is called token section which stores the total number of bytes that must be processed by the hardware in the MAXLEN-field. The D-field determines if the so called toggle bit must be set or not. The toggle mechanism is used as an alternating bit sequence of one bit to synchronize the data transfer between host and a pipe on a function. The pipe is clearly identified through the function id (which is stored in the DEVICE ADDRESS-field) and the endpoint number (ENDPT-field) on this function. The PID-field specifies the type of transfer that must be generated by the controller for this Transfer Descriptor. Possible values are IN (this is a transaction which moves data from a function to the host controller), OUT (moves data from the host controller to the function) and SETUP (with a SETUP-transfer the configuration of a function can be changed by the host controller or by other higher layers).
The last part of a Transfer Descriptor only contains the starting-address of the memory block that must be transfer to the function or that stores the data, sent from the function to the host, after the transfer has been successfully determined.
Beside these four parts the USB specification [Int96] reserves another 32 bytes in the Transfer Descriptor. This part is not used by the host controller and can therefor be used by the host controller driver.

3.4 Mode of Operation

The hardware begins the execution of a frame by determining the current Frame List Pointer through the Start-Of-Frame Register and broadcasting a Start-Of-Frame packet to all connected functions signalling that the previous frame has ended. Meanwhile the hardware analyzes the Frame List Pointer. If it is not valid (T-field of the Frame List Pointer is set) the hardware idles one millisecond and starts all over again. But, if the Frame List Pointer is valid the hardware checks whether the referenced

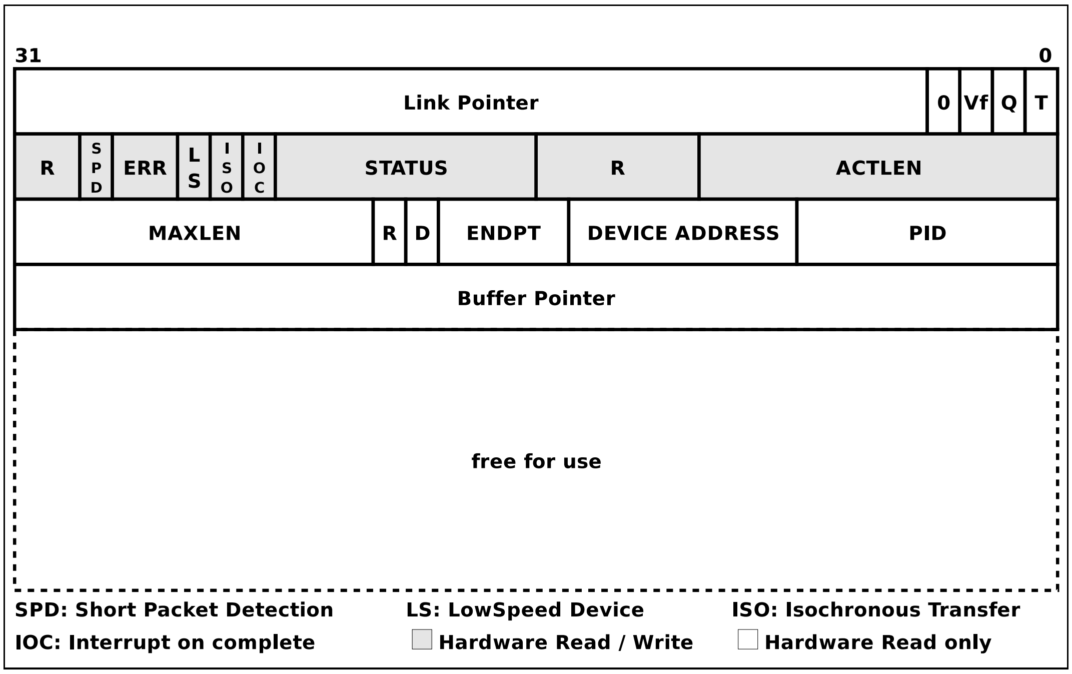

Figure 3.6: UHCI Transfer Descriptor.

data structure is a Transfer Descriptor or a Queue Header and begins processing.
If the referenced data structure is a Queue Header the hardware determines whether to follow the Transfer Descriptors striping through all transfers of this frame (in the UHCI specification [Com98] this is called horizontal context) or along the current transfer (the so called vertical context) by analyzing the Vf-filed of the Transfer Descriptor referenced by the Queue Header. After processing this Transfer Descriptor and updating the Queue Header (the element link pointer points to the next not jet processed Transfer Descriptor of the transfer) the hardware fetches the next Transfer Descriptor. In vertical context this is the next Transfer Descriptor of the current transfer of the current transfer. If the processing context is horizontal, the next Transfer Descriptor of the following Queue Header is fetched and processed. The vertical context offers a successive way of processing all the transfers in a frame whereas the horizontal context offers a concurrent processing model of the frame data.
During the processing of a Transfer Descriptor the hardware transmits the transfer to the function. The ACTLEN-field of the Transfer Descriptor gets updated during processing the memory. In case of errors the transfer may be canceled (depending on the ERR-field, defining the number of errors that may occur without canceling the transfer) and the STATUS-field gets updated with an error code. If the transfer is not canceled and the end of the Transfer Descriptor is reached the STATUS-field gets marked as inactive. If the Transfer Descriptor is part of a isochronous transfer (not Queue Header controlled) the next Transfer Descriptor gets fetched and processed,

because isochronous transfers do not make use of Queue Headers, until the frame is terminated or a Queue Header is reached.

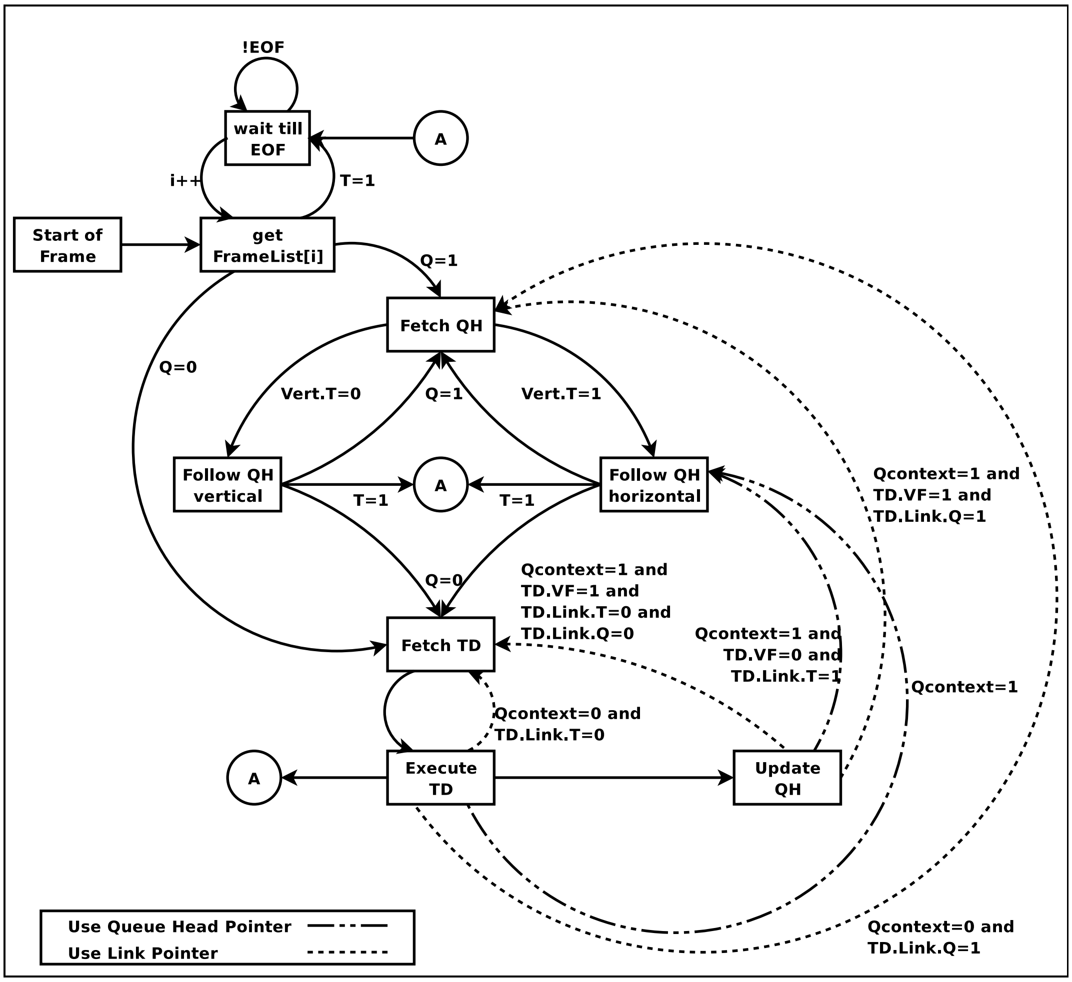

Figure 3.7: Processing of data structures.

3.5 Summary

The UHCI specification describes a well defined and structured concept to fulfill the requirements that are defined by the USB specification. The way of constructing the frames leads to data structures that can be easily understood and reproduced. The linked list out of *UHCIQueueHeaders* devides the frame into the various transfer type classes as there are control, interrupt, isochronous and bulk transfer. Each of this transfers itself is represented through a linked list of *UHCITransferDescriptors*. This kind of doubled linked list, is suitable for fast processing linear data structures of variable length.

4 Design and Implementation

4.1 Introduction

This chapter discusses the implementation of the USB protocol and the UHCI specification in the JX operating system. Therefor the basic classes of the USB and the UHCI library will be discussed. Afterwards we analyze the interaction of all theses classes.

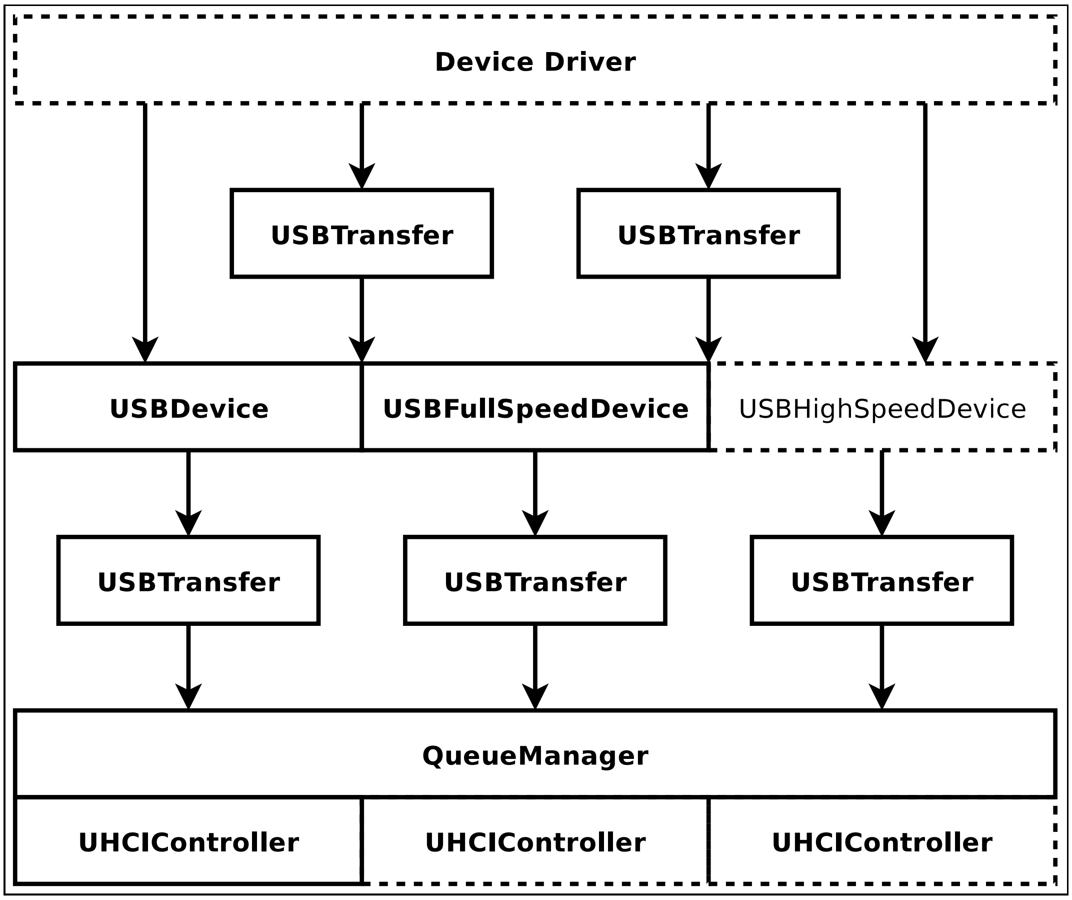

Figure 4.1: Design overview.

4.2 USBDevice Interface

4.2.1 USBDevice

```
USBDevice
+USBDevice(naming:Naming,manager:USBQueueManager)
+open(): void
+close(): void
+activatePoller(): void
+deactivetePoller(): void
+clearFeature(type:int,value:int,index:int): void
+getConfiguration(mem:Memory,offset:int,length:int): void
+getDescriptor(mem:Memory,offset:int,value:int,index:int,length:int): void
+getInterface(mem:Memory,offset:int,value:int,lenght:int): void
+getStatus(mem:Memory,offset:int,type:int,index:int,length:int): void
+setAddress(value:int): void
+setConfiguration(value): void
+setDescriptor(mem:Memory,offset:int,value:int,index:int,length:int): void
+setFeature(type:int,value:int,index:int): void
+setInterface(value:int,index:int): void
+synchFrame(mem:Memory,offset:int,index:int,length:int): void
+interruptIN(trans:USBTransfer,mem:Memory,offset:int,length:int): void
+interruptOUT(trans:USBTransfer,mem:Memory,offset:int,length:int): void
```

Figure 4.2: USBDevice class diagram.

According to the USB specification [Com98] the class *USBDevice* makes the basic functionalities a USB device has to offer to device drivers. It offers control communication calls which have to be supported by each device. Such as getDescriptor, which returns all device-, interface- or endpoint-descriptors as demanded, getStatus which returns information whether a device is attached, ready or sleeping, setAddress to assign a unique address from 1 to 127 that identifies the function on the bus, or setConfiguration which activates one of the configuration profiles that are provided through the device, ... Beside these communication calls it also offers the possibility to communicate with the device over the interrupt pipe. The interrupt pipe is mostly used to poll status information from the attached device the class *USBDevice* offers the possibility to automatically poll this information in a user defined polling interval. This polling interval can vary from the minimal polling interval supported by the attached device to several minutes. If the automatic polling is activated a new thread is started which controls the interrupt pipe communication for this device. In the time interval between two polling intervals always the last values are returned to the caller. If it is not necessary to poll this information, for example because it is well known that this status information do only change on certain actions initialized by the host, it is also possible to fetch this information only when needed. As already mentioned there are predefined functions for each possible communication via the control pipe, all calls to the device via the control pipe are executed synchronous and the *USBDevice* class handles the generation of the corresponding *USBTransfers* (a communication object, that will be described later) and control communication descriptors. According to the USB specification, no other than the defined control calls are allowed. therefor there is no need to provide an interface to the control pipe of the device to any higher layer. Because only one control command at a time is valid the *USBDevice* only has to use one *USBTransfer* instance which can be reused after each control command. The reuse of the *USBTransfer* instance leads to a smaller memory footprint and also speeds up the whole USB system because garbage collector runs can be reduced to a minimum.

4.2.2 USBFullSpeedDevice

USBFullSpeedDevice
+USBFullSpeedDevice(naming:Naming,manager:USBQueueManager)
+bulkIN(trans:USBTransfer,mem:Memory,offset:int,length:int): void
+bulkOUT(trans:USBTransfer,mem:Memory,offset:int,length:int): void
+isochronousIN(trans:USBTransfer,mem:Memory,offset:int,length:int): void
+isochronousOUT(trans:USBTransfer,mem:Memory,offset:int,length:int): void

Figure 4.3: USBFullSpeedDevice class diagram.

The *USBFullSpeedDevice* class only enhances the *USBDevice* interface to bulk and isochronous transfers. Like for the interrupt pipe there are functions to sent and receive data from the attached device via these control pipes. The *USBDevice* and *USBFullSpeedDevice* need not to about the data that is exchanged through the interrupt, bulk and isochronous communication pipes the classes just offer a very general interface for sending and receiving. This interface just requires a correct *USBTransfer* (which is archived through the filling method of the *USBTransfer* class, as we will see later) and a memory object. This memory object stores the data to be sent or provides the space for the data that will be transferred from the device to the host. Several vettings are performed by the *USBDevice* class to ensure that the memory object provides enough space, so that there will be no problems when the hardware accesses the specified memory addresses.

4.2.3 USBHighSpeedDevice

The USB 1.0/1.1 specification has only support for so called *USBLowSpeedDevices* (represented by the *USBDevice* class) and *USBFullSpeedDevices*. With the USB 2.0 specification becoming more widely accepted, the design of the USBDevice class was designed in such a way that the *USBHighSpeedDevices* would only have to extend the interface of a *USBFullSpeedDevice* and can reuse the implementations of the *USBLow-* and *USBFullSpeedDevices*.

4.2.4 Device Drivers

Because of the very powerful interface that is provided to the device driver developer the current driver which is settled over the *USBDevice* class must only provide a mapping between the device interface and the interface of the *USBDevice*. For a mouse driver, for example, this would mean, that the events like a button click or the movement of the mouse be automatically polled by the *USBDevice* with the highest polling rate supported. The actual mouse driver would only have to report any changes of these data to the AWT or any other drawing interface.

4.3 USBTransfer

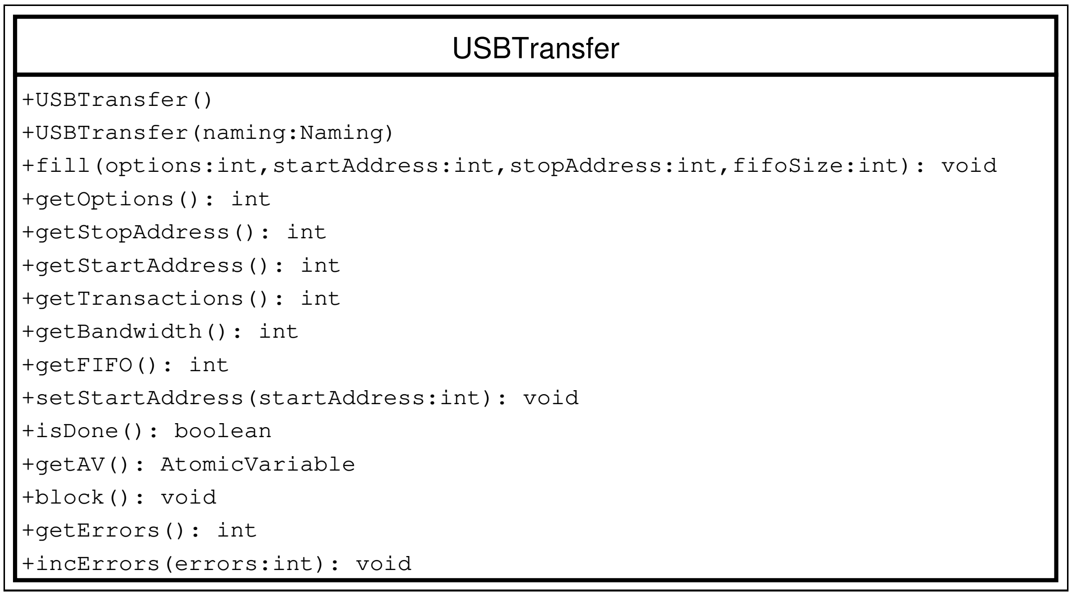

Figure 4.4: USBTransfer class diagram.

This class is exclusively used for communication between the *USBDevice* and lower-level layers like the QueueManager and the *USBController*. These objects are passed from one of the *USBDevice* instances, that represent each one attached devices, to the QueueManager where they are stored for buffering and scheduling reasons. Each *USBTransfer* represents one host-to-function or function-to-host communication. It contains information which is needed to create the data structures (*UHCITransferDescriptors* and *UHCIQueueHeaders* on a UHCI system) that are to be processed by the hardware. Most of this information remains constant for the whole transfer, like the identification (through the device id and the pipe on that device) of the *USBDevice* to which the data shall be transferred or from which it shall be gathered, the ISO-flag - indicating an isochronous data transfer - , ... Besides this constant information, the correct TOGGLE bit for the next *UHCITransferDescriptor* in that specific transfer is automatically altered by the *USBTransfer* class, therefor the underlying classes can rely on these information. Additional information like the amount of bandwidth (only the payload of the *USBTransferDescriptors*) and the number of *USBTransferDescriptors* needed to accomplish the transfer is calculated during the initialization of a *USBTransfer* by the device driver or the *USBDevice*. During processing the *USBTransfer* by the *USBController* this information is kept correct because it is used to determine the optimal transfer order of all transfers on the USB bus during one frame. After an *USBTransfer* has been submitted from an *USBDevice* to the *USBQueueManager*, the *USBDevice*, respectively the device driver, has the choice whether they want to wait for the *USBTransfer* to be processed and transferred to the devices, or not. In case that the *USBDevice* wants to wait a *USBTransfer* offers an AtomicVariable on which the process can block itself and gets notified after the *USBTransfer* has been completed. If the *USBDevice* does not want to block itself after the *USBTransfer* has been submitted, it can either block itself after some work has been accomplished or it can check from time to time if the *USBTransfer* has been transferred.

4.4 USBQueueManager

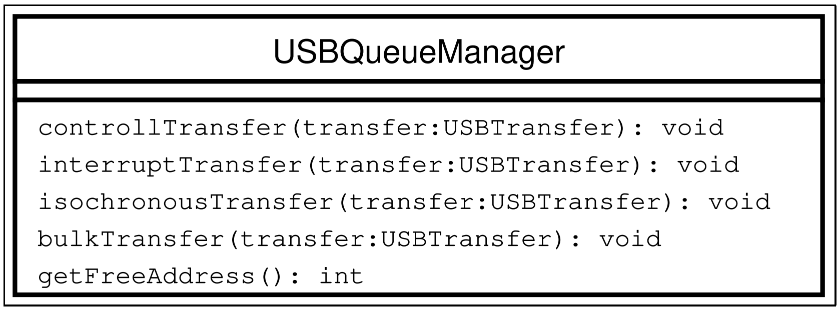

Figure 4.5: USBQueueManager class diagram.

The *USBQueueManager* interface defines methods for the *USBDevices* to submit generated *USBTransfers* to the *USBController*. For each communication pipe a single queue exists.
Newly supplied *USBTransfers* are get appended to the end of the according queue. *USBTransfers* that were started, but could not be fulfilled, during a frame get appended to the head of the queue, so that they will be processed at first during the next frame. At the moment a first-in-first-out strategy is used, because it is the most

simple, fast and clearest method and archives a reasonable load on the USB bus. To archive a even better load other selection algorithms like, smallest-job-first or best-fit, could also be implemented. To keep the single queues in a consistent state, all push, pop and append operations must be synchronized. At the moment a global queue-lock must be acquired before any changes can be submitted. To avoid locking a more sophisticated method would be to implement a synchronization via a compare-and-swap mechanism.

4.5 USBController

USBController
+USBController(pci:PCIDevice)
+getSupportedConfigurations(): DeviceConfigurationTemplate[]
+open(conf:DeviceConfiguration): void
+close(): void
+interrupt(): void
+controllTransfer(trans:USBTransfer): void
+interruptTransfer(trans:USBTransfer): void
+isochronousTransfer(trans:USBTransfer): void
+bulkTransfer(trans:USBTransfer): void
+getFreeAddress(): int

Figure 4.6: USBController class diagram.

The *USBController* is responsible for creating the data structures that can then be processed by the hardware.
The *UHCIController* therefor implements two private inner classes *FrameListCreater* and *FrameListCollector*.

4.5.1 FrameListCreater

The *FrameListCreater* thread traverses all communication queues and creates the needed *UHCIQueueHeader* and *UHCITransferDescriptors* accordingly to the UHCI specification. The collector thread processes all queues of the *USBQueueManager*, starting with the control queue. This is necessary because at least 10% of the USB bus bandwidth has to be reserved for the control data transfers. After the control queue was processed it processes the interrupt and isochronous transfer queues, because the UHCI specification assures a maximal bandwidth of 90% to these transfer types. Last but not least the bulk transfer queue is processed if there is still bandwidth left in the

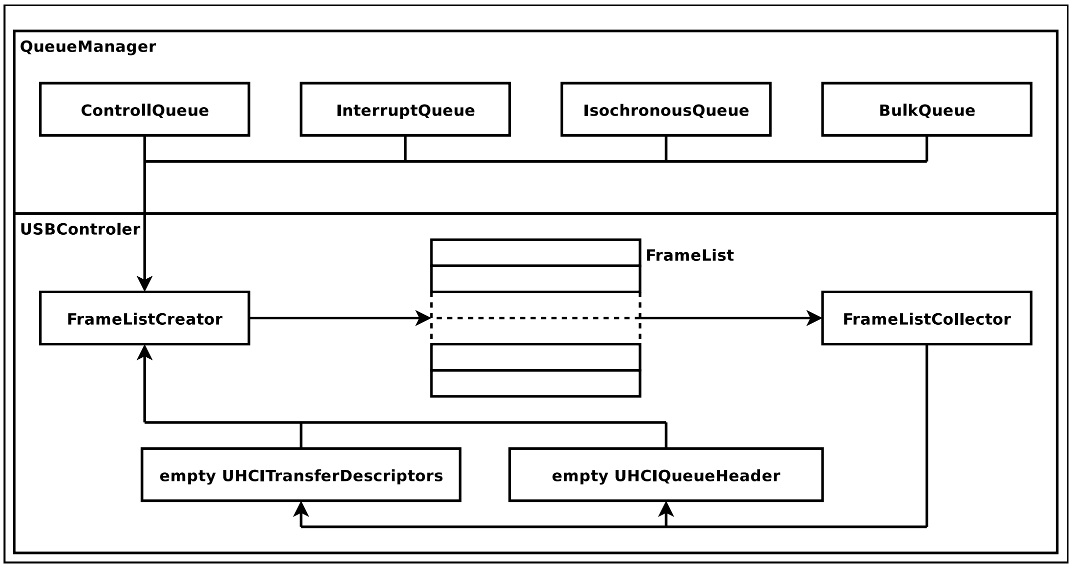

Figure 4.7: FrameListCreater - FrameListCollector interaction.

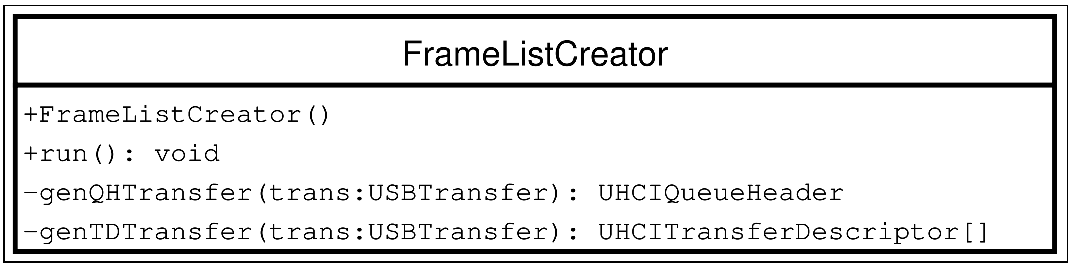

Figure 4.8: FrameListCreater class diagram.

current frame. The sequence in which the different transfer types are sent in a frame (see figure 3.1) differs from the sequence in which the queues are processed through the *FrameListCreater* thread. This because processing the queues in the order as they are sent afterwards, would make it more complex to keep the various bandwidth standards.

After constructing a complete frame out of the information given in the various transfer queues, the creator thread starts all over until there is no data left in any of the queues or until there are no more *UHCITransferDescriptors* or *UHCIQueueHeaders* left from which a frame could be constructed. In either of these cases the thread would block itself until new data has arrived in one of the transfer queues or another frame has been transmitted successfully and therefor the *UHCITransferDescriptors* and *UHCIQueueHeaders* from this frame can be reused to build a new frame. The limitation of the number of *UHCITransferDescriptors* and *UHCIQueueHeaders* the creator thread can use is important to prevent the system from using the entire system memory for UHCI data structures in times of high load on the USB bus.

4.5.2 FrameListCollector

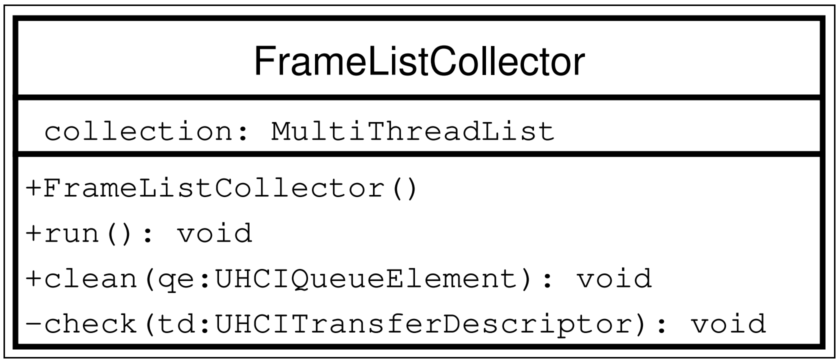

Figure 4.9: FrameListCollector class diagram.

The *FrameListCollector* waits until transferred frames are submitted to its work queue, and then performs a check whether all transfers in this frame have been executed correctly or if errors occurred. For these purposes the collector thread has to traverse the whole frame again and check each *UHCITransferDescriptor*. Each *UHCITransferDescriptor* has a number of fields that are updated by the hardware. These fields are check by the *FrameListCollector* before a *UHCITransferDescriptor* gets marked a ready for reuse again. The thread checks whether all bytes of the transfer have been processed, and whether the hardware has marked the *UHCITransferDescriptor* as done. Besides these fields, also the error counter that is decremented by the controller for each error that occurred, gets checked and its value gets reported to the corresponding *USBTransfers*, the *UHCITransferDescriptors* and *UHCIQueueHeaders* are marked as ready for reuses and can then be used again by the creator thread. There are two possible ways to fill the working queue of the collector can be filled. On the one hand all or at least one *UHCITransferDescriptor* in a frame had the IOC-bit (Interrupt-On-Complete) set, in this case the hardware would cause an interrupt on the end of that frame. On the other hand the Frame Register can be polled in intervals of one millisecond. At the moment only the IOC-method is implemented for the JX operating system, because it doesn't require additional locking but and can be implemented very easily. The best performance results could probably be achieved by combining the two approaches. In times with high bandwidth appearance the polling method could be made the default behavior for the *USBController* and then switch dynamically to the IOC-mode in times with low bandwidth appearance. Through this dynamical changing between these two methods, the interrupt overhead could be minimized by polling the frame pointer register.

4.6 Summary

One of the main goals during the design and the implementation was to create classes that are based on the memory structures that are described in the various specifications. Therefor the UHCI data structures like the *UHCIQueueHeader*, the *UHCITransferDescriptor*, the *UHCIFrameList* and the *UHCIFrameListPointer* only encapsulate the memory structures, that are needed by the hardware, to achieve a good interface for the programmer. Another goal was to define the interfaces in a way that they do not only fit the requirements of the UHCI controllers but can be reused for other controller types like OHCI and EHCI.

5 Design Comparison

5.1 Introduction

USB is getting more and more common. Thus more and more operating systems are supporting the USB protocol nowadays. In this chapter we will take a closer look at the USB implementations of other operating systems, for example the USB stack of the Linux operating system. Beside the low-level functionality that is provided through an operating system, we will also take a closer look at the more flexible user space implementation of LibUSB.

5.2 JX

The JX operating system is based on a small microkernel which is responsible for system initialization, CPU context switching, and low level protection-domain management. The Java code is organized in components, which are loaded into domains, verified, and translated to native code. The unit of protection and resource management is called a domain. Each domain has its own heap with its own garbage collector. The garbage collectors run independently and they can use different garbage collecting algorithms. Every domain also has its own threads. A thread does not migrate between domains during inter-domain communication. Memory for the thread control blocks and stacks is allocated from the domains memory area. Domains are allowed to share code - classes and interfaces - with other domains. Inside a domain protection between the various Java object instances is assured through the protection mechanisms of the Java language, like scope rules.
The domain concept allows the JX operating system be be configured in a way that fits best the actual needs. As can be seen in figure 5.1 the USB stack can be put in a single domain (monolithic design) or can be split in controller, queue manager and device driver parts.

5.3 JX vs. Linux

The modular design of the JX USB implementation allows a general apportionment in two layers. The low level layer consists of the corresponding *USBController* imple-

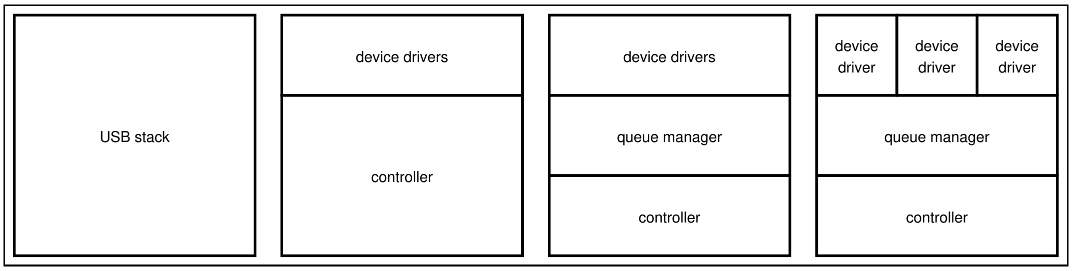

Figure 5.1: Various configurations of the USBStack.

mentation and the helper classes which belong to it. For the UHCI controller this are the FrameListCreator and FrameListCollector classes, that were described in chapter 4. These classes can be integrated into a single domain. This corresponds to the implementation of the USB stack as it can be found in a Linux operating system for example (see figure 5.1). The USB concept for the JX system features a simple but therefor also multi functional interface for higher layers. A direct comparison between the implementation and design of the USB stack of Linux and JX shows that the object-oriented design of the individual parts like the hardware machine-intimate components to which classes like the UHCIFrameListPointer, UHCITransferDescriptor and UHCIQueueHeader belong, are less complex and therefor more maintainable, thorough their structured object-oriented design. Hence, a direct consequence is that single classes can extensively be tested. The separation in functionally encapsulated units entail more security and stability to the whole operating system. In the Linux kernel beside the low level data structures, so called URBs (Uniform-Request-Brokers) must also be created for each transfer. These URBs have to supply certain information about the transfer and to take care that the transfers has been accomplished correctly. If that was not possible that at least the application that initiated the transfer gets informed that the transfer was not completed correctly. The JX system totally passes on the that additional layer that was introduced in the Linux implementation through the URB, which leads to a higher efficiency. Of course the functionality that was introduced by the URB is also available in the USB implementation of the JX operating system, but not as part of the kernel or low level layer but as part of the device driver layer. By providing generic USB drivers such as *USBDevice* and *USBFullSpeedDevice* each device driver can revert to this functionality and does not have to deal with it on it's own.

5.4 JX vs. LibUSB

The goal of the LibUSB is to provide a consistent user-space interface for programming USB device drivers and communicating with attached devices across various operating system platforms. At the moment Linux, OpenBSD, NetBSD, FreeBSD, Darwin and MacOS X are supported. The classes USBDevice and USBFullSpeedDevice are comparable to the functionality of the LibUSB in terms of USB 1.0/1.1. But in contrast to LibUSB they have the advantage to be able to communicate directly with the underlying layers like the QueueManager or the USBController. LibUSB is only capable of communicating indirect with the actual USB system of the operating system. On Linux systems LibUSB uses the usbfs-file-system, for accessing the USB stack, which brings overhead and several security risks out to the system. Incorrect or malicious USB device drivers can therefor affect all *USBdevices* or even the whole system. Through the modular composition of the USB stack in the JX operating system outsourcing of each driver in a separated domain is supported. This leads to maximal possible security and flexibility. Incorrect drivers can only affect their own domain, without affecting any other domain.

5.5 Summary

The design of the USB stack as it is implemented for the JX operating system, offers composition of the fast and efficient data processing like it is achieved thru the simple and general interface that is offered by the *USBController* and the flexible interface which the *USBDevice* classes offer to the device drivers. Developers that choose to develop drivers or components of the USB stack have the freedom of choice on which level of abstraction they want to develop their components.

6 Summary and Future prospects

This thesis had the target to design and integrate USB support to the JX operating system. Therefor the basic principles of USB where discussed. After the general USB principles were evident, the actual requirements for the universal host controller interface were analyzed. With this information the general requirements for an USB support through the JX system was described. In this discussion all the design principles were singled out. In detail that were:

modularity: The design of the classes was geared to the data structures that are introduced by the USB and UHCI specification. These data structures are the smallest components in the complete USB stack. Thus the maintenance of these classes is easier and by emulating data types the USB and UHCI specify, the introduction into the USB codebase is simplified.

flexibility: Through the object-oriented design, class implementations can change, without affecting other parts of the USB stack. It is for example possible to implement a new USB device layer without changing the device drivers or the USB controller. Thus it is also possible to adjust the USB stack to special needs for example of embedded systems, by only exchanging some layers of the USB stack.

security: Through the modularity only classes that have to work close together must be loaded into a single domain. On a monolithic system like Linux the device drivers, the USB devices and the USB controller can not be separated from each other and therefor can affect each other. Through the Domain concept in JX not only different layers can be separated (like it is done by the LibUSB) but also several USB device drivers (all part one layer) can be separated.

efficiency: By the modularity and the flexibility of the design the user has the possibility to balance efficiency and security as it fits his needs. A very efficient, but not very secure configuration for example would be to pack all USB classes into one Domain. A secondary possibility would be to pack each USB class into its own Domain, this would lead to a very secure but not very efficient configuration. So the user can choose exactly that configuration that fits its needs in best in terms of security and efficiency.

In chapter 5 then a comparison between the JX USB implementation and an existing kernel space implementation, considering Linux as example, and an user space, namely the LibUSB, have been made. In this chapter it became clear, that the design goals under whom the JX USB stack had been designed and implemented, lead to good compromise between flexibility and security one the one hand and efficiency one the other.
The USB device classes provide an abstraction level that is similar to the abstraction layer that is provided through the LibUSB, whereas the underlying layers provide a similar interface as the Linux kernel. Thus JX provides the developer various layers of abstraction to develop USB device drivers.
Also the JX system now has a fully operational USB 1.0/1.1 stack further work can be made by developing various USB device drivers, to get a better and more complete device support.
Beside the device driver development an other field of activity can be the additional support of other hardware architectures that only support the open host controller interface (OHCI). Or the extend of supporting the USB 2.0 specification and with that the possibility to run USBHighSpeedDevices at full speed.
A last point would also be to implement the improvements that where mentioned in chapter 4, like removing the global-queue lock in the transfer queues of the *USBQueue-Manager* and replacing it with more fine granular locks or maybe even omitting the locks by implementing the linked lists with compare and swap.

Bibliography

[And01] Don Anderson. *Universal Serial Bus system architecture*. 2001.

[Com98] Compaq, Intel, Microsoft and NEC. *Universal Serial Bus Specification*, 1.1 edition, September 1998.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich, and Juergen Kleinoeder. The jx operating system. *USENIX Annual Technical Conference*, pages 45–58, 2002.

[GK02] Michael Golm and Juergen Kleinoeder. Ubiquitous computing and the need for a new operating system architecture. 2002.

[GKB01] Michael Golm, Juergen Kleinoeder, and Frank Bellosa. Beyond address spaces - flexibility, performance, protection, and resource management in the type-safe jx operating system. 2001.

[Int96] Intel Corporation. *Universal Host Controller Interface (UHCI) Design Guide*, 1.1 edition, March 1996.