Diploma Thesis

USB for DROPS

Gerd Grießbach
gg5@os.inf.tu-dresden.de
Dresden University of Technology
Operating System Group

15th March 2003

Declaration

I declare that all parts of this work were autonomously written by the author while using only legal resources. All resources that were used within this work, are explicitly marked. To the best of my knowledge the content of this work is original and was not published before by me or another author.

Gerd Grießbach, Dresden, 15th March 2003

Contents

1 Introduction

The Universal Serial Bus (USB) provides the PC architecture with a low-cost, bi-directional and isochronous interface. Devices are dynamically attachable. Nearly all currently available PC chip-sets support USB and the number of available devices grows rapidly. Since over 500 companies joined the USB Implementers Forum until now, USB can be considered as an industry standard.

The capability to perform isochronous data transfers makes the USB attracting for real-time operating systems like DROPS (Dresden Real-Time Operating System [BBH+98]). Isochronous transfers are predestined for real-time applications (audio, video) that require guaranteed delivery time but need no error correcting in the transfer.

The wide variety of available Linux USB devices represents a rich resource to extend the functionality of DROPS. To make use of this wealth, the ported Linux USB stack ought to be compatible with any Linux USB device driver. As example, a web-cam driver will be ported too. The output of the web-cam driver will be shown by a viewer. The real-time properties of all ported components will be assured. Finally, all dispensable bandwidth left on the USB bus is to be made accessible to instances of *L*^{4}*Linux*.

Document structure

The following chapter lists all basics that are required to understand this document. First a detailed description of the Universal Serial Bus (USB) will be presented. The Dresden Real-Time Operating System (DROPS) as target platform and its components used for implementation of this work will be introduced as well.

Chapter 3 takes a close look at the components to be ported. This includes the Linux USB stack and the OVCam driver, which is used to demonstrate the efficiency of the resulting USB system.

At the beginning of Chapter 4 all requirements needed to start the development are gathered. Individual problems will be identified and separately discussed. This results in a concept for the realization of the ports.

Chapter 5 describes problems, which occurred during implementation and how some aspects of the concept had to be altered.

In Chapter 6 evaluates the achieved results and ends with a performance analysis of the presented implementation.

Finally, Chapter 7 provides a short summary of this work and suggests starting-points for future development.

1.1 Acknowledgements

The author of this document wants to thank Jork Löser for being a helpful tutor during the time of this work and Christian Helmuth for his patience telling me some of the secrets of the Linux kernel. Furthermore, a lot of thanks have to go to Prof. Dr. Hermann Härtig, Norman Fenske, Martin Pohlack, Adam Lackorzynski, Lars Reuther and all the other member of the OS group at the Dresden University of Technology. Last but not least, this work is based upon the excellent work done by the developers of the Linux USB stack and Mark McClelland's OVCam driver.

2 Background

2.1 USB

The USB was designed to bring more connectivity to the PC architecture. Ease-of-use is another important feature. Devices can be attached and detached at runtime (hot-plugging). The development of USB devices and the according device drivers is supported by published specifications [CIMN98, CHPI+00].

2.1.1 User Benefits

The introduction of USB brings a lot of benefits for the user. All the confusion caused by a variety of different cables for external devices like keyboard, mouse, joystick, printer, camera etc. will be avoided by one type of connector and one type of cable. Up to 127 devices (including hubs) can be attached to a USB, and most PCs are equipped with at least two buses. Hubs can be used to increase the number of physical attachment points.

Data can be transferred to different devices at the same time with a maximum bandwidth of 12 Mbit/s (USB 1.1) and 480 Mbit/s (USB 2.0). The revision 2.0 is fully downward compatible to revision 1.1. Since most motherboards are already equipped with USB controllers, the user has no need to install adapter cards into his computer. There are no technical oddities to deal with. No restart is needed to configure a newly attached device. Since one goal of the USB developers was to integrate communication devices, real time capabilities were added.

Up to 500 mA at 5 V will be provided by the bus. Small devices like web-cam, smart card reader and hub need no own power supply and are referred to as “host powered”, keyboard and mouse anyway. Last but not least, a power management function at bus level builds the base for a less power consuming computer periphery. The following list summarizes the most important features from the user’s point of view:

- ease-to-use
- port expansion
- plug & play, hot-plugging
- protocol flexibility: isochronous and asynchronous data transfer
- “host powered” devices & power management

USB devices are available for nearly all areas of application. An overview presents Figure 2.1. Depending on the requirements of the application, an appropriate transfer speed can be chosen

by the device developers. USB 1.1 conforming devices are restricted to the applications that are framed by the gray shaded box.

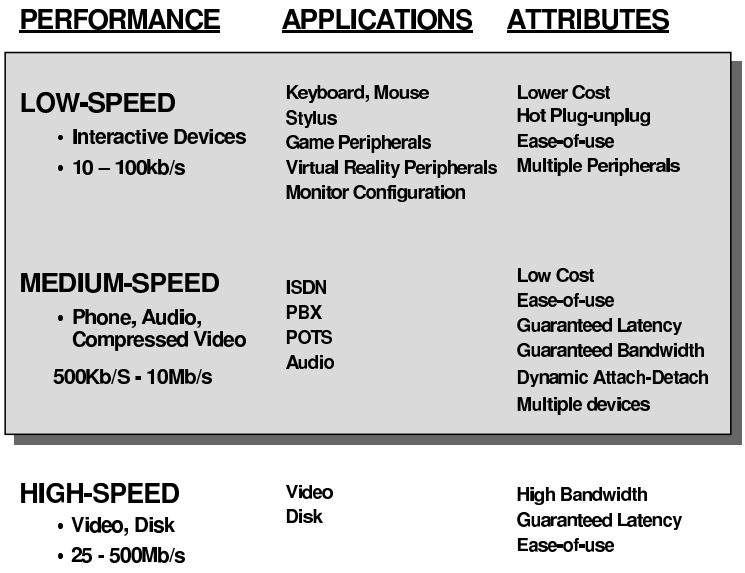

Figure 2.1: Application space taxonomy [CIMN98]

2.1.2 Developer Aspects

The USB specifications [CIMN98, CHPI+00] describe the bus attributes, communication protocol, types of transactions, bus management, and the programming interface required to design and build systems and peripherals that are compliant with this standard. The goal is to enable devices from different vendors to co-operate in an open architecture. This concept is accepted by the market and makes the architecture attractive for developers.

The specified protocol allows arbitrary data rates and transfer types. Flow control and error handling are included at minimal overhead. Failures caused by malfunctioning devices and cables, inappropriate user handling or transmission failures will not affect the system stability. This leads to a robust system. The legacy interrupt sharing problem of the PC architecture is extenuated by reducing the number of needed interrupts to one per bus. At last, the availability of inexpensive USB interface chips attracts the developers to integrate them in their devices. The following USB features are especially interesting for hardware developers:

- industry standard
- adaptive protocol
- robustness
- few system resources needed
- cheap hardware components

2.1.3 Architecture

After the introduction of common USB features this section enumerates all physical components that are interacting on a USB, and how they are connected.

Topology

In contrast to its name, the USB physically features a cascaded star structure (tiered-star) with the host as root (see Figure 2.2). From the user's point of view, devices are stackable in one serial cable. This gives the impression of a bus and makes it, for example, possible to attach a mouse to a keyboard that is directly connected to the computer.

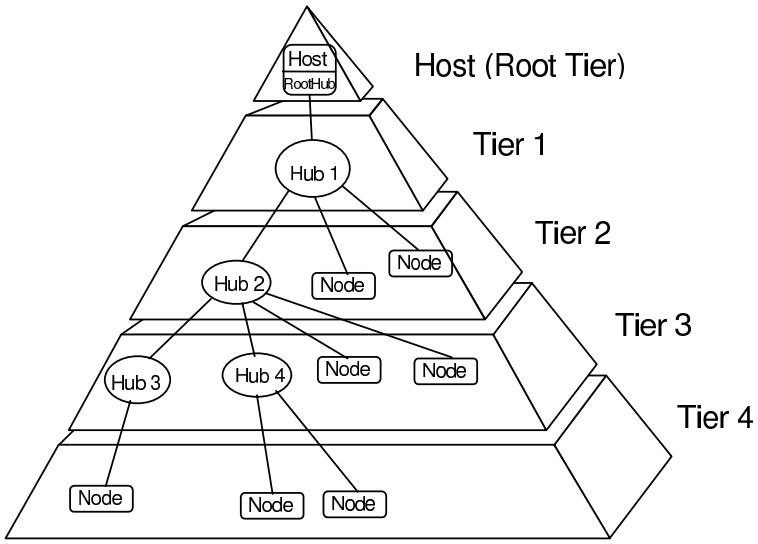

Figure 2.2: USB Topology [CIMN98]

Host

Every USB is controlled by exactly one host. The USB interface to the host computer system is referred to as the Host Controller (HC). The HC may be implemented in a combination of

hardware, firmware and software. There are only few different host controller implementations available on the market today. The most common are:

- UHCI (Universal Host Controller Interface) (Intel) [Int96]
- OHCI (Open Host Controller Interface)
  (Compaq, Microsoft and National Semiconductor) [CMS99]
- EHCI (Enhanced Host Controller Interface) (Intel) [Int02]

Controllers conforming to the UHCI and OHCI specifications are USB 1.1 compliant. Both controller types offer different trade-offs regarding the complexity of hardware and software. The UHCI specification has been designed to reduce hardware complexity by requiring the host controller driver to supply a complete transfer schedule. OHCI type controllers are much more independent by providing a more abstract interface and by doing a lot of work transparently. High-speed transfers coming with USB 2.0 can only be handled by EHCI controllers.

Several PCI (Peripheral Component Interconnect) cards are available to upgrade computers with USB functionality. These host controllers are mostly implemented with OHCI, while the chip-sets are usually implemented with UHCI. Intel as developer of the UHCI and EHCI standard equips its chip-sets with host controllers of this kind.

To provide one or more attachment point, every host controller incorporates a so-called *virtual root hub*.

Hubs

To obtain additional attachment points, external hubs will be needed. Although hubs are connected to the bus in the same way as any other devices, they are part of the USB infrastructure (see Figure 2.2). After hubs are configured by the host, they work as a repeater, multiplexer (transfer to host) and demultiplexer (transfer to device). Hubs are often integrated in other devices (monitor, keyboard).

Devices and hubs communicate with each other via peer-to-peer connections. Due to signal propagation delays, the topology is restricted to five hub levels and a cable length of five meters. Hubs have to protect the USB from invalid data transfers emitted by malfunctioning devices. Attaching and detaching of devices is first indicated by hubs, which propagate these events to the host for further handling.

USB 1.1 compliant hubs act as switches and forward all data without conversion (no store-and-forward). Transfers, coming from the host, pass through the hub only in direction of the addressee. Thus, slow speed devices will be protected from full/high-speed traffic. Hubs implementing the 2.0 specification contain additional buffers and convert traffic of different data rates if needed.

Devices

All USB specification compliant devices offer a self-description including vendor ID, device ID, version number, class ID, subclass, protocol etc. (see Section 2.1.5). With this information the host controller driver (HCD) is able to select a proper device driver. A unique address is

assigned to every device by the HCD. This number is needed to address a specific device and used by hubs to route the traffic. A device implementing more than one logical function is denoted as a composite device. Devices incorporating a hub are called compound devices.

2.1.4 Protocol

While devices physically connect to the USB in a tiered-star topology, the host communicates with each logical device as if it were directly connected to the root port. It is the host controller's responsibility to partition USB time into 1 ms (USB 2.0: 0.125 ms) quantities called *frames*, regardless of the other bus activity or lack thereof. Within a frame data packets addressing different devices can be placed. The host sends a token (request) with the address of the desired device on the bus and awaits an immediately reply resp. transfers data to the device.

The host controls all attached devices and initiates all transmissions. The devices are not allowed to communicate directly with each other. Data and control transfers occur only between host and a selected device. Hence, the scheduling of all bus activities is exclusively managed by the host. This simplifies the enforcement of real-time properties and avoids competing access. Data collisions are implicitly avoided and the communication is simplified. This leads to simplified device hardware as well. On the other side, the host has to handle nearly all kinds of errors (CRC, timeout) etc. itself.

To detect configuration changes the host controller periodically polls the state of all attached devices. The CPU (Central Processing Unit) is completely disencumbered and only notified by interrupt if something happened what requires handling by the HCD.

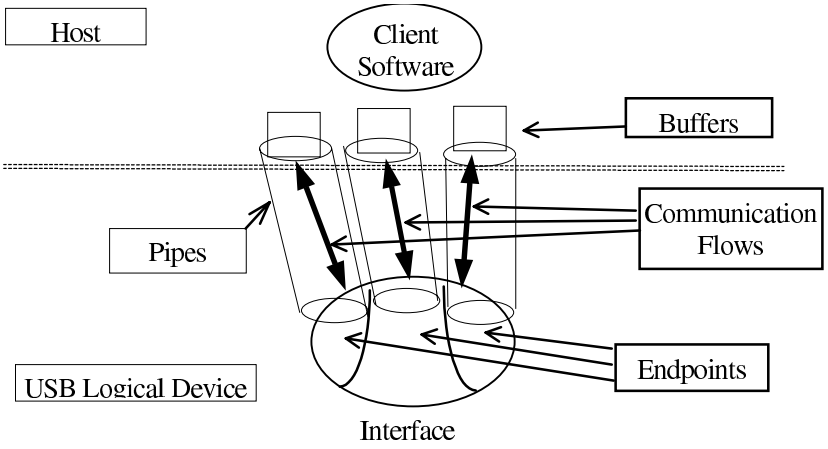

Figure 2.3: Communication Flow [CIMN98]

Pipes & Endpoints

From the view of the USB interfaces, all communication between a device and the host controller is performed by using so-called *pipes* (see Figure 2.3). These pipes are not to be confused

with UNIX pipes. A USB pipe can be considered as an association between an endpoint at a device and the software on the host (abstract peer-to-peer connection). Devices mostly provide more than one endpoint, depending on their configuration. Endpoint zero, also known as *default control pipe*, is used to determine device identification and configuration requirements, and to configure the device. The *default control pipe* is mandatory for every device and must always be available once a device is powered and has received a bus reset. A pipe basically consists of the following information:

- device address
- endpoint number
- direction (IN: to host, OUT: to device)
- speed (slow, full, high)
- transfer type
- status

Except for some predefined control transfers (standard device requests), data submitted via pipes appears as unstructured data stream at USB protocol level.

Transfer Types

The communication between a driver and the according device is restricted to the following four basic types of data transfer:

1. Control Transfer
    - mandatory, because it is needed for identification and configuration of devices at attach time
    - can be used for other device-specific purposes, including control of other pipes on the device
    - max. 10% of bus bandwidth reserved
    - lossless data delivery by protocol

2. Interrupt Transfer

   - notification of device state changes
   - guaranteed maximum service period (poll interval): 1-255 ms
   - used for signals, characters or coordinates with human-perceptible echo or feedback response characteristics

3. Isochronous Transfer

   - occupy a prenegotiated amount of bus bandwidth with a prenegotiated delivery latency (streaming real time transfers)
   - no error detection and handling (retrying) by protocol
   - together with interrupt transfer max. 90% of bus bandwidth can be reserved
   - requests to establish pipes with unsatisfiable requirements will be rejected

4. Bulk Transfer

   - consumes remaining bandwidth
   - generated or consumed in relatively large and burst-like quantities
   - wide dynamic latitude in transmission constraints

To provide a guaranteed delivery mechanism, interrupt, control, and bulk transfers are retried if they do not complete successfully. Slow speed devices are limited to small amounts of data payload and only support control and interrupt transfers.

USB supports no transmission retries for isochronous transfers. In other words, devices can perform isochronous transfers in an easy send-and-forget manner. It has to be pointed out that only interrupt and isochronous transfers are suitable to meet real-time requirements.

2.1.5 Descriptors

USB devices report their attributes using descriptors. The descriptor hierarchy includes devices, configurations, interfaces and endpoints (see Figure 2.4).

Device Descriptor

A device descriptor contains general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device has only one device descriptor.

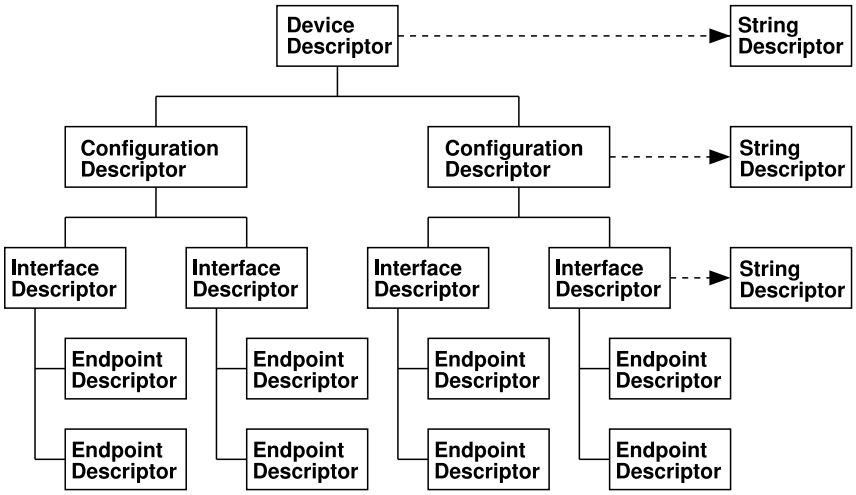

Figure 2.4: Descriptor Hierarchy [Kel01]

Configuration Descriptor

The configuration descriptor specifies a collection of interfaces provided by the configuration. Each interface may operate independently. For example, an ISDN device might be configured with two interfaces, each providing 64 KBit/s bi-directional channels that have separate data sources or sinks on the host. Another configuration might present the ISDN device as a single interface, binding the two channels into one 128 KBit/s bi-directional channel.

Interface Descriptor

USB interfaces represent the logical interfaces to device drivers. The host operating system tries to find an appropriate driver for each interface. It is possible that devices using configurations with more than one interface may be served by more than one driver.

An interface can consist of up to 15 endpoints and may include *alternate settings* that allow the endpoints and/or their characteristics to be varied after the device has been configured. For example a multifunctional device like a video camera with an internal microphone could have three alternate settings to change the bandwidth allocation on the bus: camera activated, microphone activated and both activated.

Endpoint Descriptor

An endpoint descriptor contains information required by the host to determine the bandwidth requirements of each endpoint. An endpoint represents a logical data source or sink of a USB device. The endpoint zero is used for all control transfers and there is never a descriptor for this endpoint. The USB specification uses the terms “pipe” and “endpoint” interchangeably.

String Descriptor

Where appropriate, descriptors contain references to optional string descriptors that provide displayable information about a descriptor in human-readable form, sometimes multilingual.

2.1.6 Software Layer

Regarding the communication model, the host and the device are divided into three distinct layers depicted as rows in Figure 2.5.

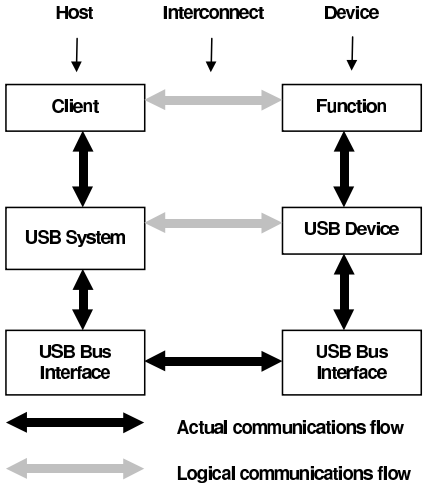

Figure 2.5: Interlayer Communications Model [CIMN98]

Vertical arrows indicate the actual communication on the host. The corresponding interfaces on the device are implementation-specific. All communications between the host and device ultimately occur on the physical USB wire. However, there are logical host-device interfaces between each horizontal layer. These communications, between client software resident on the host and the function provided by the device, are typified by a contract based on the needs of the application currently using the device and the capabilities provided by the device. All layers of the host software will be implemented by according drivers:

Host Controller Driver (HCD)

The HCD acts as USB bus interface and was introduced to more easily map the various host controller implementations (see Section 2.1.3) into the USB system, i.e. provides an abstraction of the host controller hardware. The interface between HCD and USB driver is never directly available to the client (device driver). Therefore, a client interacts with its device without knowing to which Host Controller the device is connected. Overall, the host layers provide the following capabilities [CIMN98]:

- detection of the attachment and removal of USB devices

- management of the USB standard control flow between host and devices
- management of the data flow between host and devices
- gathering of status and activity statistics
- controlling of the electrical interface between the Host Controller and USB devices, including the provision of a limited amount of power

USB Driver (USBD)

The USB system software is implemented by the USBD, which provides the basic host interface for clients (device drivers) to USB devices. This includes data transfer mechanisms in the form of I/O Request Packets (IRPs[1]), which consist of a request to transport data across a specific pipe. In addition, the USBD is responsible for presenting an abstraction of a USB device that can be manipulated for configuration and state management. As part of this abstraction, the USBD owns the *default control pipe* (see Section 2.1.4) through which all USB devices are accessed for the purposes of standard USB control.

Client Driver

The client driver layer describes all the software entities (clients) that are responsible for interacting with specific USB devices. One single client (driver) may even control different interfaces of several devices. Clients are only allowed to interact with the according peripheral hardware. The USB standard places USB system software between the client and its device; that is, a client cannot directly access the device's hardware.

A class code is assigned to a group of related devices or interfaces with similar attributes, requirements or services that has been characterized as a part of a USB Class Specification [SI97]. A class of devices may be further subdivided into subclasses and within a class or subclass a protocol code may define how the host software communicates with the device.

A complete class specification allows manufacturers to create implementations that may be managed by an adaptive device driver. These so-called *class drivers* are intended to be developed by operating system and third party software vendors as well as manufacturers supporting multiple products.

Until now the following device classes have been specified (excerpt): audio, smart card, security, communication, firmware, imaging, IrDA, HID (Human Interface Device, input devices), mass storage, monitor and printer. In spite of the availability of class drivers, in reality most of the USB devices are designed to be used with its own drivers, which are often exclusively available for MS Windows.

2.1.7 Resource Management

Whenever a pipe is to be established by the USBD for a given endpoint, the USB system must determine if it can support the pipe. The USB System makes this determination based on the

[1] An IRP is identifiable request by a software client to move data between itself (on the host) and an endpoint of a device in an appropriate direction.

requirements stated in the endpoint descriptor. There are two stages to check for available bandwidth. First the maximum execution time for a transaction is calculated from:

- number of data bytes to be transmitted
- transfer type
- depth in the topology (signal propagation delay)

Then, the frame schedule is consulted to determine if the indicated transaction will fit. The allocation of the guaranteed bandwidth for isochronous and interrupt pipes, and the calculation of whether a particular control or bulk transaction will fit into a given frame, can be determined by a software heuristic in the USB System. This calculation must also include any implementation-specific delays, such as preparation or recovery time required by the host controller itself.

2.2 Real-time Systems

Real-time means the ability of a computer system to react to any event under all circumstances within a specified period. Such events may be triggered by user inputs or devices, which support isochronous data transfers. Because there are limited resources in any system real-time activities have to be scheduled in advance including reservation of all necessary resources.

"Hard" real-time systems are designed to handle the "worst case". Missing the deadline could lead to an immediate damage as in the case of an engine control. To satisfy the viewer of a video application, the stream of frames has to be shown with a constant frame rate. Rare occurrences of dropped frames cannot be considered as dangerous. Such "soft" real-time systems provide often a much better utilization of the system resources.

Event or Time driven

Event driven systems will be primarily influenced by asynchronous events triggered from the outside (hardware interrupts). In case of such an event, resources will be allocated according to the thread priority.

The opposite approach to event driven systems is the complete planning of all system processes and consideration of process synchronization, data flow and failure model. Schedules for all critical resources will be set up.

Event-based scheduling is more flexible than time driven scheduling because it can adapt the schedule dynamically to react to changes in the execution environment. For example, a real-time application has not consumed its guaranteed worst-case CPU quantum. The remaining CPU time can be donated to other applications.

Events, Latency and Jitter

Real-time operating systems have to deal with events caused by internal time triggered and external device driven interrupts. In both cases, the delay between occurrence of event and start

of the interrupt handling routine is called latency or reaction time. In real systems with a certain complexity the latency will not show a deterministic behavior. Due other system activities latency varies in a statistical manner denoted as jitter.

2.3 DROPS

The operating system group at the Dresden University of Technology focuses its research and development on the Dresden Real-Time Operating System (DROPS). According to the terminology of real-time systems introduced in Section 2.2, DROPS can be placed between these two extreme approaches. It tries to unite predictability, dynamic and adaptability. It features a multi server architecture and supports application with QoS (Quality of Service) requirements. An overview of the DROPS architecture is shown by Figure 2.6.

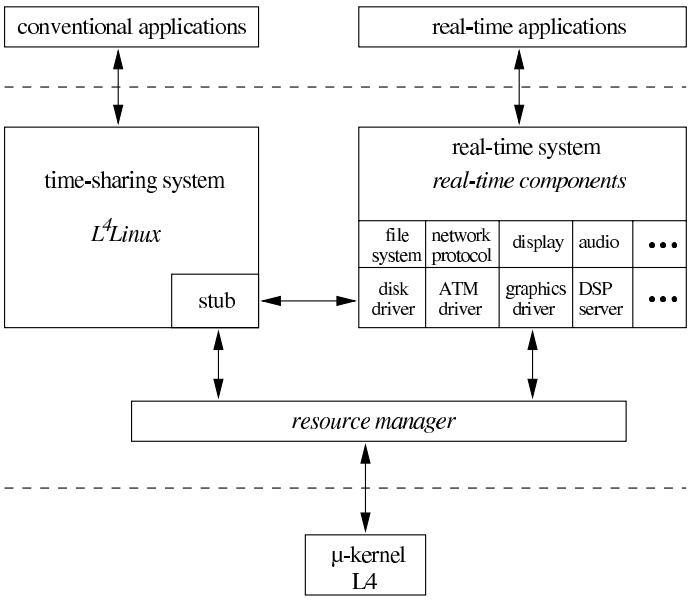

Figure 2.6: DROPS Architecture [BBH+98]

The DROPS project is based on an in-house implementation (Fiasco) of the second generation micro-kernel L4 [Lie96]. According to the micro-kernel philosophy, all classical operating system components like device drivers and resource managers are implemented as server processes at user level. The micro-kernel only provides thread control, address space protection and IPC (Inter Process Communication) mechanisms.

One of the project goals is to support applications that are used in multimedia systems. Multimedia applications can accept violations of QoS contracts, as long as they occur rarely. Multimedia systems are usually run on standard PC hardware, which features a limited suitability for

hard real-time applications. Not consequently considering the “worst case” at all costs leads to a much better utilization of the system resources while assuring the specific real-time requirements of multimedia applications at the same time.

Running Linux Applications

Another goal of DROPS is to run real-time and non real-time applications in parallel as demonstrated in [BBH+98]. According to the DROPS architecture most device drivers should provide a *L4Linux* stub, the USB stack port as well. *L4Linux* [Hoh96] is a server based Linux port running on an L4 micro-kernel which makes a large amount of non real-time applications available to DROPS. Only minimal changes on architecture dependent parts and device drivers of the Linux kernel were necessary. The ABI (Application Binary Interface) of this server offers binary compatibility to the Linux implementation for x86-CPUs.

2.4 DROPS Components

This chapter briefly introduces DROPS components that are essentially needed for the implementation of the port.

2.4.1 Common L4 Environment (L4Env)

L4Env consists of libraries providing the basic operating systems primitives like locks and semaphores for application programmers. Additionally it defines basic abstractions for thread control and memory management with according runtime support. L4Env applications are not L4-API dependent anymore.

2.4.2 Device Driver Environment (DDE)

One of the main tasks of an operating system is to provide an abstraction of devices. Obtaining adequate device drivers is problematic for most research based operating systems. Because of the complexity an in-house development of device drivers is often very time consuming and expensive, respectively. A promising approach is to port device drivers from other operating systems to DROPS with minimal adaptations to keep the maintenance effort as low as possible. The proof of this concept is shown in [Hel01], which introduces the Device Driver Environment (DDE). It acts as a wrapper around Linux device drivers (see Figure 2.7) emulating the Linux kernel environment. DDE provides all needed kernel calls for handling of processes, wait queues, timer, interrupts, memory and PCI bus.

Of course, the aim of DDE is not to implement the whole Linux kernel functionality. Therefore extensions of DDE will be done only on demand. This was necessary in the context of this work (see Chapter 5).

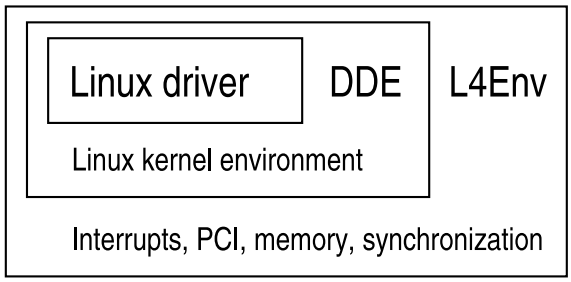

Figure 2.7: DDE Wrapper Layer

2.4.3 DROPS Streaming Interface (DSI)

DSI [LHR01] offers a framework for real-time interprocess communication. It defines a timed and packed-oriented zero-copy transport protocol at user-level between real-time components using shared memory. For actual data transfer, DSI uses a producer-consumer scheme on a ring buffer containing packet descriptors. In this work DSI will be used to establish a real-time communication between the web-cam driver and a video viewer.

2.4.4 Desktop Operating Environment (DOpE)

With DOpE [Fen02] an extensible windowed graphical user interface is available for the DROPS operating system. It is a foundation for capturing those application fields, where comfortable graphical user interfaces in connection with real-time demands are needed. Control widgets and frames displaying streaming data can be created easily. DOpE provides an easy to use API (Application Programming Interface) and might be the future platform for all graphic oriented DROPS applications. DOpE fits the needs of the required web-cam application and thus will be used.

2.5 Existing USB Stacks

Nearly all operating systems offer USB support today. As co-developer of the USB standard, Microsoft released the first full functioning USB stack within MS Windows. Presently most devices are shipped with MS Windows drivers.

Apple Computers even released its iMacs (Mac OS) without legacy peripheral connectors like serial and parallel ports. Hence, the USB support by this system software is well developed. The major drawback for further examinations of both mentioned operating systems is the lack of available source codes.

There are many other commercial USB stacks offered for instance by Intoto, MicroDigital, Phoenix Technologies, Simtec and SoftConnex, which partially are designed for use in embedded devices and are closed source too.

Due to the complexity of the development of a complete USB stack from scratch the decision was made to port an existing USB stack to DROPS. The widespread and freely distributable operating systems FreeBSD, NetBSD and Linux come with source codes. FreeBSD and NetBSD

use the same USB code base. The most USB device drivers are currently available for Linux. After a fundamental rework pushed by Linus Torvalds, the USB stack of Linux has matured over the last years and is now in stable state. Further maintenance can be expected.

The operating system group of the Dresden University of Technology has gathered a lot of experiences from porting parts from Linux into DROPS. Since a USB stub should be provided for *L*^{4}*Linux*, the Linux USB stack is chosen as porting base. The Device Driver Environment (DDE) provides a promising starting point.

Investigations for other related works brought no utilizable results, at least not for porting the Linux USB stack.

3 Porting Base

The aim of this work is to port the Linux USB stack under consideration of real-time properties. This chapter gives a detailed insight into the Linux USB stack. The reader is expected to be familiar with the Linux operating system in general.

3.1 The Linux USB stack

As reasoned in Section 2.5 the Linux USB stack [LUP] was chosen as porting base. To keep in synchronization with the development of L^4Linux, the current kernel version 2.4.20 will be the subject of all following examinations.

Within Linux exists a subsystem called the USB core with a specific API to support USB devices and host controllers. Its architecture follows closely the guidelines of the USB communications model (see Figure 2.5) and is shown in Figure 3.1. Analogous to Figure 2.5, USB device drivers are clients of the USB core resp. USB system. Host Controller Drivers (HCD) are assigned to physical bus interfaces.

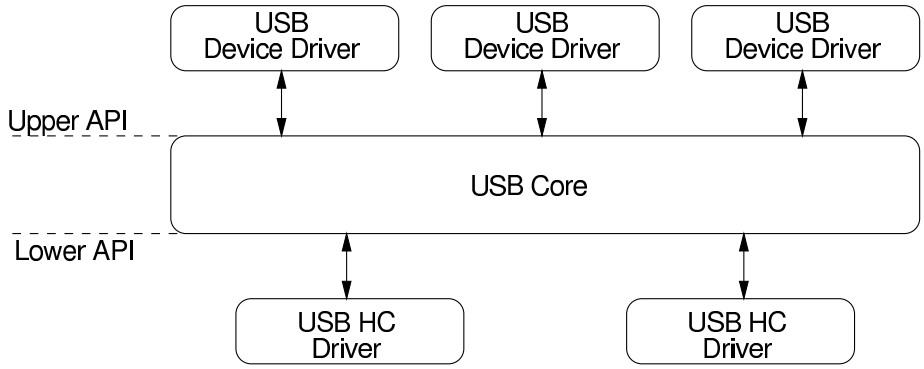

Figure 3.1: Linux USB Stack Architecture [Fli00]

The USB core abstracts hardware specific and USB specific issues. It provides an API with data structures, macros and functions and contains routines common to all USB device drivers and host controller drivers. These functions can be grouped into an upper and a lower API layer. The upper one, the device driver API [Fli00], is well documented while this is not the case for host controller interface. There is no official specification for the interface between the HCD and the USB core, except for the source code.

3.1.1 Device Driver API

Client drivers access the USB-API using a standard Linux header file (usb.h). The Linux USB subsystem uses only one data transfer structure called *USB Request Block* (URB), which is equivalent to the entity IRPs (I/O Request Packet) from the USB specification. This structure contains all parameters to setup any USB transfer type. The USB core asynchronously processes each request and signals the completion per callback function, if desired.

The USB-API features a quite lean structure. All functions can be grouped into the following four categories:

1. USB Device Driver Framework
   - device driver registration an deregistration
   - interface claiming
2. Standard Requests
   - device configuration
3. USB Transfers
   - URB handling (allocate, free, submit, unlink)
   - URB initialization macros
4. Compatibility (Convenience) Wrappers
   - control (configuration) and bulk transfers
   - caller is blocked until the request is finished (synchronous)

In the Linux environment there also exists the possibility to write user mode device drivers, even for the USB. This simplifies and speeds up the development. Drivers of this kind are represented as files in a so-called 'USB device file system' and can be controlled by common file operations. Porting Linux USB user mode drivers towards DROPS is currently not possible. A file system abstraction layer has to be available in DROPS to let user mode drivers access the devices. Not supporting user mode drivers does not lead to a significant disadvantage, because only few such drivers currently exist. In contrast to that, there are more than 50 kernel module drivers available.

3.1.2 USB Request Block (URB)

To make the API lean the URB structure will be used for every transfer type. Figure 3.2 provides an overview about the huge amount of parameters that are only be valid for certain transfer types. A detailed description provides [Fli00].

3.1.3 Architecture

This section describes all components that are necessary to build a running USB subsystem.

```
struct urb {
 spinlock_t lock;             // lock for the URB
 void *hcpriv;                // private data for host controller
 struct list_head urb_list;   // list pointer to all active urbs
 struct urb *next;            // pointer to next URB
 struct usb_device *dev;      // pointer to associated USB device
 unsigned int pipe;           // pipe information
 int status;                  // returned status
 unsigned int transfer_flags; // USB_DISABLE_SPD | USB_ISO_ASAP ...
 void *transfer_buffer;       // associated data buffer
 int transfer_buffer_length;  // data buffer length
 int actual_length;           // actual data buffer length
 int bandwidth;               // bandwidth for request (int,iso)
 unsigned char *setup_packet; // setup packet (ctl)
 //
 int start_frame;             // start frame (iso,irq)
 int number_of_packets;       // number of packets in request (iso)
 int interval;                // polling interval (irq)
 int error_count;             // number of errors in transfer (iso)
 int timeout;                 // timeout (in jiffies)
 //
 void *context;               // context for completion routine
 usb_complete_t complete;     // pointer to completion routine
 //
 iso_packet_descriptor_t iso_frame_desc[0];
};
```

Figure 3.2: URB Structure

USB Core

The USB core module provides the implementation of the USB-API building on different host controller driver implementations. The framework manages the attached devices for every bus, registers device drivers and parses descriptors. The USB core looks for an appropriate driver for each newly attached device. Bandwidth accounting is to be handled by the USB core, too.

While the USB core defines the resource managing policy, the actual implementation of mechanisms to deal with data transfers is left to the respective HCD. For example, URB related calls will be redirected to an according function implemented by the HCD.

Finally the USB core offers hot-plugging support. Every time a device is attached an application (hot-plug daemon) specified in /proc/sys/kernel/hotplug loads drivers modules and configures the device. The detachment of devices will be reported to the hot-plug daemon as well. If the USB file system is available, user-space drivers (a common Linux application) can directly read all the device descriptors under /proc/bus/usb. Since DROPS (resp. DDE) currently does not support no /proc file system, the Linux USB file system will not be ported.

Host Controller Driver

The host controller driver fills the gap between the USB core and host controller, which can only be accessed via memory mapped PCI registers. As lowest software layer, the HCD registers an interrupt handler (ISR[1]), called *top half* in Linux terminology. Since the frame intervals last 1 ms (full speed), up to 1000 hardware interrupts per second may occur. USB 2.0 introduces high-speed micro frames that last for 125 μs and may cause up to 8000 hardware interrupts per second. Since USB 2.0 simply features an eight time higher frame rate, in the rest of this document all considerations of frames can easily adapted to micro frames.

At the end of the ISR all drivers owning recently completed URBs will optionally be notified via completion calls. If any error occurs during URB submission, the completion handler will be called, too. *Bottom half* handlers can, if needed, be implemented in the device driver.

To perform its tasks, the HCD need some memory. Slab caches[2] will be used to hold a private version of the URB structure, extended by internal scheduling parameters. The communication with the UHCI host controller will be performed using so-called *transfer descriptors,* which will be stored in PCI pools to ensure DMA (Direct Memory Access) capability. These *transfer descriptors* will be processed by the HC asynchronously to the processor. As long as the transfer schedule is empty the HCD sends an idle host controller to suspended mode.

Linux provides two implementations for UHCI host controllers, one for OHCI and EHCI, and for embedded host controllers. One of the UHCI variants is a replacement, but does not implement all “features” of the older one that is needed to make some old drivers work. Thus, both UHCI implementations are currently supplied for compatibility reasons.

Hub Driver

The detection of hub events (attaching and detaching of devices) is accomplished by a hub daemon thread, which uses the services of USB core. In case of any hub event the USB core will be notified.

Every host controller contains an integrated virtual root hub, which is controlled with help of two PCI registers. To be able to use the hub driver even for the virtual root hub, the HCD maps the hub driver USB requests (control and interrupt transfers) to the corresponding virtual root hub registers. For handling the virtual root hub actually no real USB transfers are performed. The state of external hubs is polled periodically by an interrupt transfer.

3.1.4 Real-time Behavior

The real-time properties of the USB stack are preconditioned for real-time device drivers and have to be achieved at all cost. In this section, the Linux USB stack implementation will be reviewed. All factors influencing the real-time behavior in any way will be identified and requirements concerning the environment of the ported stack will be posted. It has to be reminded that only isochronous and interrupt transfers are suitable for real-time applications.

[1] Interrupt Service Routine

[2] Slab caches and PCI pools can be considered as caches for constant sized memory objects like structures, buffers etc.

Hardware

Before real-time applications can be successfully deployed, real-time capabilities have to be ensured at hardware level. Developers should be aware of so-called intelligent devices, which may relieve the processor from doing some work but show unpredictable behavior. For instance, mass storage devices may often implement hardware buffer for data caching.

There are many factors affecting the real-time performance that are out of interest in this work. Regarding the port of Linux USB stack, the only peripheral hardware device used is the host controller, typically integrated in the motherboard chip-set. USB host controllers are not equipped with hidden buffers. Devices producing isochronous data streams will surely implement a well-defined buffer based upon the sampling characteristics, which can be considered as unproblematic.

Interrupt Handling

The host controller provides interrupt capability based on two general groups of interrupt sources, those resulting from execution of transactions in the schedule, and those resulting from a Host Controller operation error. Additionally, individual transfers can be marked to generate an interrupt on completion (IOC). When a successfully processed transfer descriptor is encountered with the IOC bit set to 1, the IOC bit in the HC status register is set to 1 at the end of the frame. If interrupts are enabled, a hardware interrupt is signaled to the system. HC operation errors will signaled per interrupt, too. To catch these interrupts, the HCD registers an interrupt handler (ISR, *top half*).

In an interrupt context only the most urgent tasks will be performed: interrupt acknowledgement, schedule update and completion handling. The schedule update just removes the descriptors of the transferred data. The scheduling itself will be done in thread context (API call). If desired, an URB completion notification has to be sent to the concerning device driver. The control flow is transferred to the device driver. This mechanism causes problems that will be discussed later in this work.

Additionally, to detect hub status changes, the virtual root hub registers are polled periodically by a timer interrupt. The hub daemon thread that needs to react to port status changes in non-time-critical manner will be activated by this timer interrupt. No other dedicated threads are used by the Linux USB subsystem.

Thread Control

Threads commonly influence each other via synchronization mechanisms like semaphores and locks. Serious problems may occur when a thread is blocked until its deadline is exceeded. In the case of the Linux USB stack, there is a competition between interrupt serving and API serving threads to access URB and descriptor lists. A solution for this problem will be presented in Section 4.2.6.

Threads using blocking API calls (see Section 3.1.1) will be added to wait queues until the according transfer is finished. When a process is put to sleep, the USB core is still alive and can be called by another process.

The only API call concerning running real-time transfers, `submit_urb()`, is complete asynchronous. All it does is to schedule the given URB.

There are several `wait_ms()` calls placed in the source code, which cause delays up to 500 ms. These calls have no negative effects to the real-time behavior because they are only used for configuration handling that always will be performed in process (client) context. In other cases it is necessary to give devices some time to react to configuration changes (control transfer).

It is assured that the source code to be ported do not try to forbid system interrupts (`cli()`, `sti()`) as it can be assumed for all used DROPS environment resources.

Bus Scheduling - Bandwidth Reservation

On the USB, 1 ms frame times are used to transfer data. The host controller begins each frame by generating a start of frame (SOF) token. In UHCI, if there is isochronous data to be transferred, the HCD schedules this data first. The HCD ensures that there is enough time to complete all scheduled isochronous and interrupt (<90%) transfers with some time remaining for control (<10%) and bulk transfers as shown in Figure 3.3.

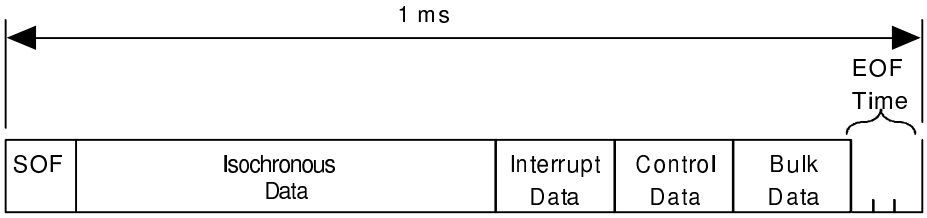

Figure 3.3: UHCI Transfer Type Schedule Order [Int96]

The host controller supports real-time data delivery by generating a SOF packet every 1 ms. If necessary, minor adjustments can be made to the frame time period to maintain real-time synchronization throughout a USB system.

The UHCI data structures include a frame list (FL), queue heads (QH), and queued transfer descriptors (TD). These data structures are used by HCD to construct a schedule in host memory for the host controller to execute (see Figure 3.4). The host controller is programmed with the starting address of the FL, then released to execute the schedule. During a frame period no other synchronization with the HCD is required. Transfer descriptors point to data buffers and include information about the addressing, data, and general behavior characteristics of the transaction. The flow through the schedule is based on link pointers in the FL, TDs, and QHs. Link pointers are the fundamental component used to connect all the scheduled data objects together. The host controller uses the link pointer to determine where to find the next TD to execute. Addresses in the link pointer fields must be a physical address and not a virtual address. At the start of a frame, the host controller repeatedly follows link pointers, beginning at the current FL index, pausing its traversal to perform transactions described in TDs, and stopping when the frame expires (or a terminate bit is set on a horizontal flow execution).

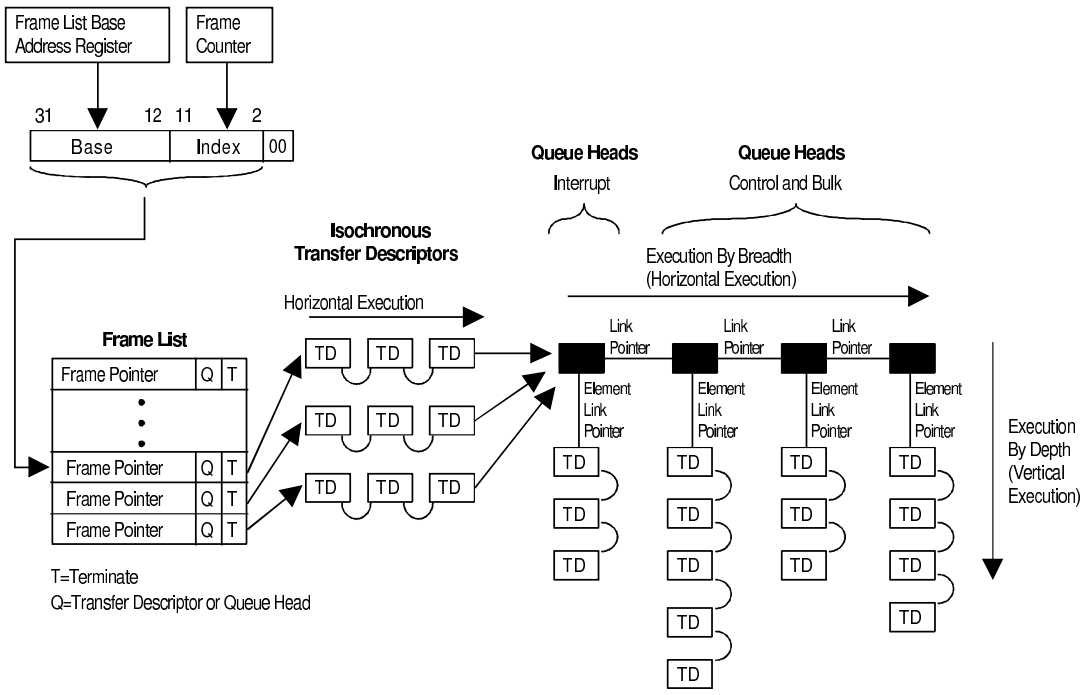

Figure 3.4: Example Schedule [Int96]

All traffic on a bus is under control of the according host controller. Depending on the current bus topology and utilization, the HC decides to schedule further real-time (isochronous) transfers or not (admission control). Once a transfer is permitted the compliance with the prenegotiated QoS parameters is guaranteed. For isochronous pipes, the bandwidth required is typically based upon the sampling characteristics of the associated function. The maximal acceptable latency is determined by the buffer size available at each endpoint. All isochronous pipes transfer exactly one data packet each frame. The USB limits the maximum data payload to 1023 bytes for each isochronous pipe. One pipe with 1023 bytes payload and 1000 frames per second consumes 69% of the full speed bus bandwidth, not considering the overhead. If an URB contains a *transfer buffer* that is larger then the maximum payload size of the destination pipe endpoint the HCD will split the buffer into small packets that fit in the frames. This avoids overload conditions at device side. Overload at host controller side is automatically avoided by USB specification. Devices send only if they were told to do so.

The USB bandwidth and bus accesses are granted based on a calculation of worst case bus transmission times (see Section 2.1.7) and required latencies. The bus time contribution is calculated as a constant although it is actually data-dependent[3]. Therefore, bus time will remain

[3]The actual bus time taken for a given transaction will almost always be less than that calculated because bit stuffing overhead is data-dependent. Worst case bit stuffing is calculated as 1.1667 (7/6) times the raw time. Bit stuffing is employed by the transmitting device in order to ensure adequate signal transitions. A zero is inserted after every six consecutive ones in the data stream. This gives the receiver logic a data transition at least once every seven bit times to guarantee the clock synchronization. For more details concerning the electrical and signaling details refer [CIMN98].

in each frame time versus what the frame transmission time was calculated to be. In order to support the most efficient use of the bus bandwidth, control and bulk transfers are candidates to be transferred over the bus as bus time becomes available. Both transfer types are not suitable for real-time applications. This feature just increases the throughput and is called *bus bandwidth reclamation*.

Error Handling

A timeout in jiffies[4] can be specified to automatically remove an URB from the host controller schedule. A timeout error will be notified via completion call, so an application (driver) can react in a proper way.

Bus errors caused by malfunctioning hardware may lead to unpredictable delay for recovery. In these cases, no warranty of real-time properties can be made any longer.

Other Resources

Sophisticated features like paged memory must be avoided. The L4 memory protection may cause page faults in unsuitable moments as well. These problems can be avoided by statically mapping (pinning) the memory pages before they will be accessed in real-time data transfers. Slab caches and PCI pools needed by the HCD have to be implemented this way as well.

As shown in [Sch02], the design of the PCI bus is variable enough to be utilized in real-time environments. The advantages of the round-robin arbitration scheme used by current host bridges are simplicity and low arbitration costs. However, it is too simple to provide adequate features for using the PCI bus in real-time systems where overload can occur.

3.2 OVCam Driver

The OVCam driver [McC] was taken from Linux to act as demonstration device driver for the ported Linux USB stack. Actually, the OVCam driver is a set of drivers for the OmniVision OV5xx series of chips. These are USB-only video capture chips used in many web-cam devices and some TV-capture devices. They support streaming and capture of color or monochrome video. The OVCam driver supports most image widths and heights that are multiples of 64, with a maximum resolution of 640x480. Additionally the driver offers a couple of module parameters, for example compression (4,6,8:1) support and optional filters (mirror, deflicker artificial light). The frame rate depends on the resolution and compression and can be up to 40 frames per second.

The next sections introduce the API and architecture of the OVCam driver followed by an analysis of its real-time behavior.

3.2.1 Video for Linux (V4L) API

V4L is an API that allows control of capture devices (cards) on Linux machines. There are a variety of capture cards, which can be web-cams, TV cards, radio cards, or devices used just

[4] Linux time unit, usually 10 ms.

to capture images from a camera. A device-specific driver controls a capture device and offers a semi-standard interface to the system. Application developers can use V4L's API without knowing much about the actual device or its driver.

The V4L implementation (videodev) has no own threads. It just acts as a registrar for capture device and provides a common API, which includes data structures and `ioctl()`[5] parameters as well. With help of these parameters a capture application can control multimedia devices, especially query device capabilities, define capture windows, configure the device and control capturing.

V4L comes with a precise documentation included in the Linux kernel distribution [LKS]. Meanwhile there is an overhauled API available, V4L two, which implements slight changes, but is not supported by OVCam driver.

The OVCam driver API conforms to the V4L-API extended by special `ioctl()` calls.

3.2.2 Architecture

The OVCam driver is structured like a common Linux device driver. The *top half* handler is collecting raw data packets until the whole frame of a video stream is received. Then the *bottom half* thread will be released, which converts the raw data into standardized frames. This includes processes like format conversion, decompression and filtering. Each pixel of a frame may be touched several times. To avoid at least unnecessary copying the OVCam driver shares a memory mapped V4L stream buffer with the capture application.

3.2.3 Real-Time Behavior

Once the shared buffer has been established the capture application submits a VIDIOCMCAPTURE *ioctl* to start the capture. When the VIDIOCMCAPTURE *ioctl* returns, the frame is not captured yet; the driver just instructed the hardware to start the capture. The application has to use the VIDIOCSYNC *ioctl* to wait until the capture of a frame is finished. VIDIOCSYNC *ioctl* takes as argument the frame number to wait for. The frame post-processing will be done in the process context of the capture application.

Since USB supports no transmission retries for isochronous transfers, special error handling is needed. In the case of the OVCam driver, the incoming packet stream will be scanned for synchronization patterns (start of frame). The length of a frame is predictable. In the case of any error, incomplete frames will just be dropped and the start token of the next frame will be awaited.

[5] input/output control

4 Concept

After the introduction of the Linux USB stack, the OVCam driver and components from the DROPS environment, this chapter describes the port itself.

4.1 Design Goals

At first some general design goals for this work have to be defined:

- modularity
- real-time capability
- performance
- security

Modularity means to follow the DROPS philosophy to put every functionality in a separate module. The user should be able to extend the system capabilities at runtime by loading the desired modules. All developed modules have to integrate themselves as common DROPS modules. Modularity in the broader sense is keeping the source code of the components to be ported untouched as far as possible, in particular APIs. This allows easy synchronization with the ongoing Linux development. To keep compatible with common Linux USB device drivers, the USB-API will not be altered.

An important issue is to consider real-time requirements for all approaches that will be developed. Issues like performance and security, which are often contrary to each other, have to be regarded. All required resources of the Linux USB stack will be provided by DROPS components that can be considered as real-time capable.

4.2 Porting the Linux USB Stack

As result of Chapter 3 the port of the Linux USB stack towards the DROPS environment seems to be feasible and saves a lot of development time. Of course, the port of drivers from a monolithic kernel environment into a component based system like DROPS causes some problems described in the following sections.

4.2.1 Modularization

Co-location of Core Components

USB 1.1 equipped systems contain mostly two host controllers of one type. Since USB 2.0, to support high-speed data transfers, a system has to be equipped with an additional controller implementing the EHCI. Slow and full speed transfers are handled by a companion host controller (UHCI or OHCI). The approach to place every HCD into an extra module to provide at least the same modularity as Linux kernel modules introduces the following problem:

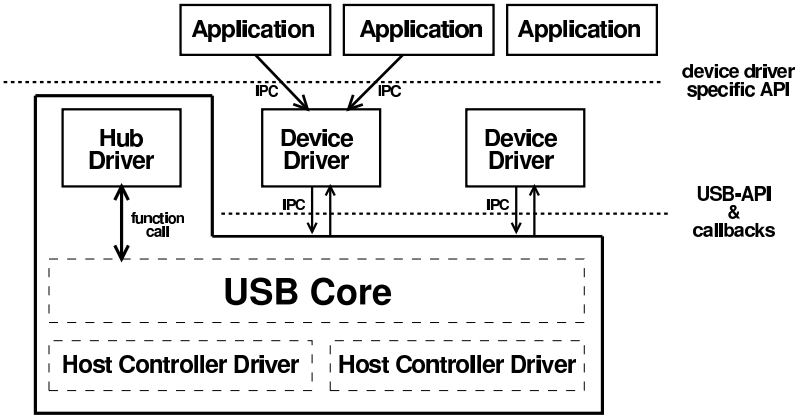

Figure 4.1: Component Architecture

Examinations revealed a strong linkage between the HCD and the core, which is very difficult to map to a client-server scheme with a lean interface. Every bus interrupt and device driver interaction would cause a large amount of IPC, which results in a significant performance loss. So, it is recommended to place (co-locate) the core and host controller drivers in the same module (address space) like it is done in the Linux kernel (see Figure 4.1). Furthermore there exists one hub driver that is always needed, so it is co-located, too. Finally, modularity can still be achieved by controlling every bus through a separate USB core module. From here, this module is called *USB core* or simply *core*.

Separate Device Drivers Modules

Since the separation of all USB device drivers is a design goal, to gain access to the API of the USB core a library is provided.

4.2.2 Wrapper Layers

The basic architecture of the DROPS USB port consists of two wrapper layers (see Figure 4.2): one wraps the USB core, and another one (USB library) has to be linked against the ported USB device driver, e.g. L^4Linux stub. For this reason the communication between the USB core and the USB device drivers is always under control of the USB port.

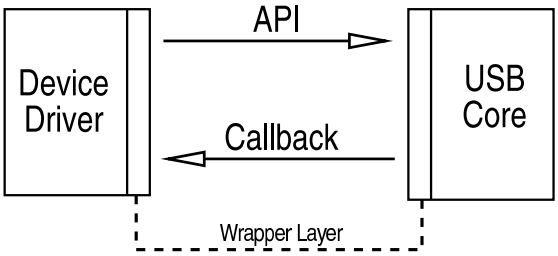

Figure 4.2: Wrapper Layers and Control Flow

4.2.3 Control Flow

Normally, a USB device driver can be viewed as client and the core as server. Callback functions defined in the USB-API break this simple client-server scheme. Sometimes the roles have to be changed. Clients pass pointers to probe and disconnect functions, file operations (fops) and URB completion handlers to the core. The core has to be able to call this functions which means the clients have to act as servers, too. In short, it is necessary to emulate a pure function call semantic as conventional in the origin of the ported components, the Linux kernel. It has to be possible to transfer the control flow boundless between the USB core and the drivers. A solution for this scenario is not yet available in DROPS and will be developed later in this section.

In Linux, core and device drivers are located within the same address space (kernel) and interact via simple function calls. In an L4 environment, IPCs are the usual way to pass the control flow to code in other modules. Suppose the core is implemented as a common server with one thread waiting for requests. Due to the synchronous manner of IPC the client is blocked until the server sends a reply. While a server is already handling a request it cannot accept another one. Thus, a second client is blocked for an undefined time. Only one thread can enter the core server at the same time, which is not satisfying. In order to handle synchronous compatibility wrapper calls (see Section 3.1.1), the ported USB stack may not be blocked until such calls are finished.

Multiple Worker Threads

Currently, a DROPS server can register only one interface serving thread at *names*. This is a naming service to help other components find the available servers. As consequence, servers in DROPS cannot handle multiples request at the same time. Therefore other ways have to be found to allow multiple clients to enter the core simultaneously. Beside the assumption that future L4 kernel APIs may offer IPC with auto-propagation, today the only way to handle multiple IPC requests in parallel is the introduction of worker threads. *Worker threads* executes exactly the same code but at each case in a different context, i. e. as proxy for a different client.

One approach to implement parallel server threads is to use a single interface thread that delegates each request to a currently unused worker thread to be immediately ready to receive another request. The worker thread will enter the core code in context of the request. Fortunately, the flick stub generator[1] produces reentrant code that can be executed by several worker

[1] Flick is an IDL (Interface Defi nition Language) compiler that creates client and server stubs for a given interface

threads at the same time (thread-safe). All requests from a certain client must be serialized. After finishing its job a worker thread cannot directly reply to the client that is doing a close wait for the interface thread's reply. So, the reply is send back to the interface thread, which has to redirect the reply back to the client. All in all, at least two additional IPCs are needed for every client request.

Another option to solve the problem is to put a little more "intelligence" into the client (lib). During the initialization of the USB library the client receives a number of dedicated worker thread IDs from the server. When a client wants to call the core the library choose a free dedicated worker thread as addressee for the request. A drawback is creating many threads in advance. On the other side, as an advantage, this approach costs no additional IPCs. Because performance is an important issue, this approach exceeds the first one and is therefore implemented.

Problems with Worker Threads

When the limit is reached more dedicated workers have to be reserved at the core server. This event, of course, may affect the real-time properties in a negative manner and should be avoided by serving a sufficient number of worker threads before any transaction starts.

A general proposition how many worker threads will be needed cannot be made. It depends on the implementation of the particular device driver and its thread structure. The design of the USB-API itself does not cause deep recursion. All USB-API calls cause at most one callback, considering further API calls by completion handlers. An example for one of the worst case scenarios is: client driver registers at core, core calls probe callback, driver calls core for configuration issues. In consequence, the number of dedicated worker threads a client has to reserve is: two times (recursion depth 1) the number of parallel running and USB-API using driver threads.

One problem with multiple worker threads is left to be discussed. On single processor systems, Linux kernel threads are not preemptible in kernel, except for disruptions caused by interrupts. Linux can be compiled for SMP (Symmetric Multiprocessing) environments, too. To avoid inconsistency caused by threads parallel accessing the same data, all critical structures are protected by *spin locks*. To provide better SMP performance, current Linux device drivers offers fine-grained locking mechanism, fortunately the Linux USB stack as well. Since the USB stack is aware of multiple threads running in the Linux kernel, so no special treatment is needed to make it multi-threading safe.

Callbacks

To accomplish callbacks the client library has to provide dedicated worker threads in the same way as the core server does. The mechanism is completely transparent for the device drivers. The worker threads of the client library will execute the callback functions as proxy for the USB core.

The dependencies between the blocked worker threads in the scenario shown in Figure 4.3

defi nition.

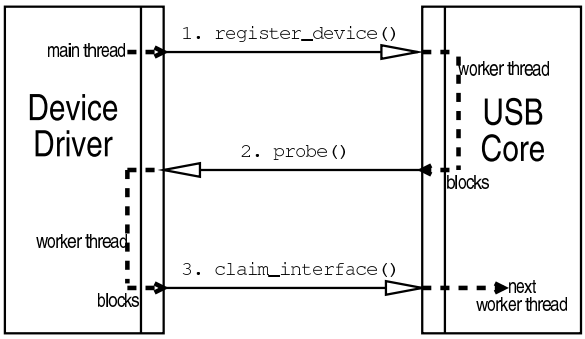

Figure 4.3: Providing Conventional Call Semantic

form a conventional function call semantic. All involved worker threads cooperate within the same request context without knowing of it.

4.2.4 Control Data Structures

The amount of control data structures and all other data that are transferred as arguments for USB-API calls is relatively small. So the decision was made to build up copies of all needed control data structures at driver and core side. Different types of data will be identified and treated in a specific way.

USB entities

The `usb_device` structure is the root of all USB specific structures and will be used as argument for all configuration calls. This structure completely describes the current state of a device including all descriptors.

During registration a device driver passes the `usb_driver` structure to the core. This structure contains among other things references to callback functions.

Descriptor structures

Descriptor structures are only used for device configuration, which will never be performed during data transfers. Thus performance need not be achieved at all cost. The descriptor structures form a tree (see Section 2.1.5) as described in the USB specification. This tree will be created by the core once a device is attached to the bus. To send the tree to the device driver, it has to be serialized.

URBs

Most control data is exchanged via URBs (see Section 3.1.2). The number of URBs to pass per second strongly depends on transfer speed and transfer buffer size. Especially devices controlling isochronous data transfers tend to produce up to 100 URBs per second. Bulk data transfers

mostly make use of more bandwidth than isochronous data transfers but use larger *transfer buffers,* which need more time to fill. Hence, less completion calls are needed.

URBs are relatively small (currently 108 bytes). Most of their elements have to be inspected by the wrapper layer anyway. Some elements of the URB structure are exclusively used at the server or the client side, respectively. Generally, having copies of a structure at both sides makes it possible to use these elements for other purposes such as storing context information.

URB structures are normally reused by the device drivers. They have to be completely build up at the server side once only at URB allocation time. Every time the control flow passes the wrapper layer, URB structures on the particular side have to be updated to preserve consistency. This is sufficient because, once a driver has submitted an URB the exclusive control over the URB is given to the core until the URB completion call.

Consistency

Device drivers are not allowed to alter shared USB structures autonomously. The driver has to use the API calls instead. Unfortunately, the API allows explicitly the parsing of the descriptor structures by the device driver.

Since there are copies of all structures at both sides, structures, which are used as an argument for an API call have to be synchronized by the wrapper layer before a call is performed. The same has to be done when the call returns. Depending on the implementation of the Linux USB core, many optimizations can be applied. In the most cases only few structure elements have to be updated. Most elements are only relevant in the particular context (client or server side).

Reference Mapping

The USB-API was designed to be used by components that share the same address space (Linux kernel). Thus, handing over references to data structures and buffers is very common. In DROPS each component has its own address space. Virtual addresses provided as argument for USB-API calls are only valid in the according address space. The usage of replicated structures in different address spaces involves an approach to map references, what has to be done by the wrapper layer. The following example illustrates how it works:

A device driver registers itself at core using the `register_device()` call. A device driver structure will be created and sent back to the driver. Now two copies of the same structure exist. Later, the client wants to submit a `usb_control_msg()` call and sends its device structure reference as an argument. The server side wrapper layer translates this address to the corresponding core structure reference and passes it to the core. Nearly all USB-API calls (see Section 3.1.1) uses either a reference to a driver, device or URB structure.

For API communication those references that are valid in the address space of the caller will be used. The according mapping information will only be stored on the other side. The callee has to find the address for the according structure in its address space. So, any reference can be addressed in both directions.

Fast address mapping can be achieved using hash tables. The unique hash key consists of the caller's task id associated with the submitted reference (virtual address).

4.2.5 Transfer Buffers

At maximum load the USB bus provides a bandwidth of up to 15 resp. 480 Mbit/s. Thus, large *transfer buffers* (TB) are needed, which demands the use of shared memory. In contrast to the control data structures, TBs contain just unstructured byte streams not containing references or anything that has to be interpreted by the USB core.

Of course, it is possible to share address space regions manually by sending *flexpages* [2]. A more promising approach is to use components of the L4 environment, which implements the concept of *dataspaces* and *region mappers* [ALE+01]. Dataspaces are unstructured data containers that can be attached to regions of an address space. TBs managed by a dataspaces can easily mapped[3]. Instead of a reference to the TB in client's virtual address space an URB request now specifies a dataspace - offset pair.

Most of the inspected Linux device drivers reuse few TBs or parts of it for all URB transfers. Generally it is not necessary to allocate new TBs for every transfer. This includes the registration of statically mapped TBs at initialization time.

If otherwise an URB transfer request wants to make use of an unattached dataspace, the TB has to be attached dynamically. After the transfer is completed, the dataspace has to be detached. This approach leads to performance drawbacks and is named as sharing a dynamically mapped TB.

Some drivers provide unattached dataspaces with relatively small size. Then it is better to copy the content of the TB. The USB core holds a statically attached TB for this case. There is a break even when copying is as fast as mapping shared memory at 500-2000 bytes, depending on processor and memory performance. This value will be used to decide whether a TB will be copied or not.

[2] In L4 flexpages are regions of the virtual address space, which can be attached to other address spaces via IPC.
[3] Before a dataspace can be used at core side, the rights to access the dataspace has to be transferred.

Recapitulating, there are three ways to transfer the content of a TB between the USB core and the drivers:

- sharing a statically mapped TB
- sharing a dynamically mapped TB
- copying the TB

TBs are allocated by device drivers and not by the USB core. Since USB device drivers are coming from the Linux environment they know nothing about dataspaces. Sure, it is possible for the USB lib to determine to which dataspace a given TB belongs. But as long it is not possible to share subdataspaces, more memory has to be shared than actually needed. This is the case for the *L*^{4}*Linux* stub, which currently shares to whole *L*^{4}*Linux* kernel memory with the USB core. The OVCam driver's allocation of the TBs was modified to use dataspaces. In the most cases, this adaption can be easily applied.

Physical Addresses

TBs are allocated by the driver as non-pageable contiguous physical memory blocks, which will be accessed per DMA by the host controller. The DROPS USB port is intended to run in DDE. Per default, only the physical address of memory allocated by `kmalloc()` can be calculated by DDE. The physical base address of other memory regions (dataspaces) have to be registered at DDE, otherwise the `__pa()` call would fail. The physical base address of an attached dataspace can easily be determined. After the dataspace is attached and DDE is updated, the USB transfer can be started.

Since the content of TBs dataspace never will be interpreted by the HCD, TBs will never directly accessed by the processor. The dataspace only needs to be attached to USB core's virtual address space. Although the dataspace is not mapped, its physical address can be calculated because *dm_phys*[4] assigns a chunk of physical memory to each dataspace at creation time.

Scatter/Gather

A last issue concerning TBs has to be mentioned. Once again, TBs are contiguous chunks of memory. USB drivers can perform *scatter/gather* transfers using linked URBs pointing to different (scattered) memory chunks. These high level mechanisms are not recognizable for the USB core. In contrast to that, the way USB core handles isochronous transfers can be considered as a kind of *scatter/gather*. Isochronous packets will be transferred every frame. Each of these packets is transferred from/to a specified offset and length within the TB. This feature can be used to synchronize isochronous data streams.

4.2.6 Assuring Real-Time Capabilities

To view a web-cam stream in real-time, a lot of components have to work together without negative interferences. In this scenario, the data stream passes a chain of software components

[4] The L4Env Physical Memory Dataspace Manager (DMphys) manages the available physical memory of a system.

including a host controller driver, a devices driver and a viewer application. To maintain hard real-time capabilities all involved components have to show a predictable behavior, which implies the availability of sufficient system resources[5]. Therefore a common resource management model is needed.

This section models the requirements regarding the most critical system resource: CPU time. An adequate scheduling policy is required to fulfill each thread's demand.

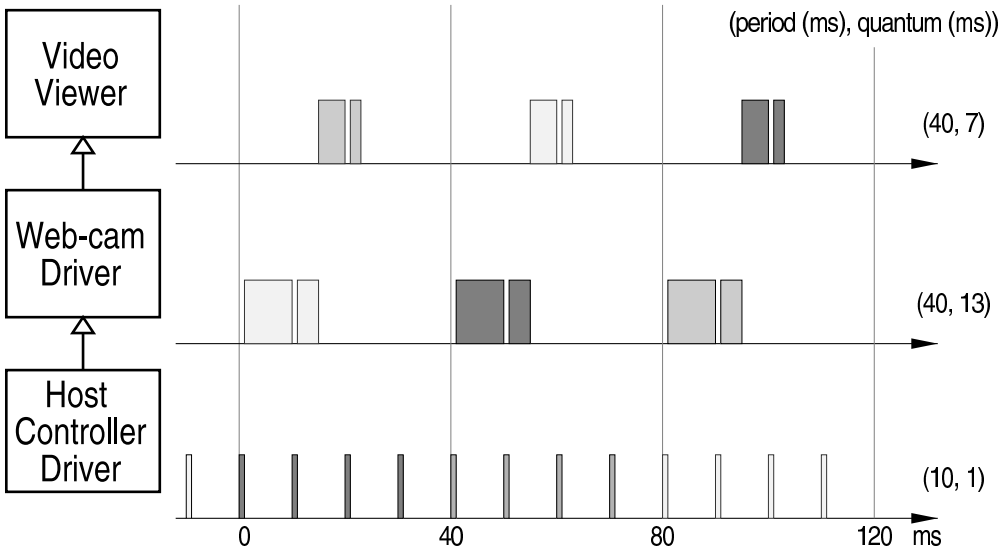

Figure 4.4: Scheduling Scenario

CPU Utilization Model

The following model is derived from the scenario of a video viewer application showing the frame stream provided by a USB web-cam. For example, the viewer displays 25 video frames per second. Each of these frames is the result of a processed raw data stream, which is assembled of a resolution dependent number of URB transfer buffers. To complete an URB request the host controller has to receive 10 iso-packets from the web-cam. Every 1 ms one iso-packet arrives from the USB. The data stream flows though the chain of components, which are synchronized to each other, but each requiring a different quantum of CPU time and period. Except for the iso-packet receiving process the described scenario is illustrated in Figure 4.4. The boxes show the CPU utilization for the respective threads.

In the underlying scenario every processing stage uses double buffering for URB transfer buffers and video frames as well. This approach adjusts the jitter at the cost of a small and well-defined delay. Right from the beginning of its period, each component can expect a ready input buffer, which has to be processed until the end of the period. Preemption while processing

[5]Refer Chapter 3 for an analysis of all needed resources.

is permitted. The different gray shadings of the CPU utilization boxes in Figure 4.4 show the way one resulting frame takes through all processing stages. This explains the delay between a frame is received from the camera and displayed on the screen. The periods of the web-cam driver and the video viewer should ideally be a integer multiple of each other (harmonic), or the problem as described in Section 5.4.3 will occur.

The analysis of the ported components (see Chapter 3) shows that most of the CPU time is consumed by device drivers, not by the USB core. Thanks to the usage of shared transfer buffers, the task of the core component is restricted to resource management (bus scheduling) and control of the HCD. In most cases simply setting and linking of transfer descriptors will be performed.

Scheduling API Requirements

Based on the presented scenario, the following requirements for the upcoming DROPS scheduling API can be derived by the illustrated scenario:

1. threads request a quantum of CPU time within a given period
2. these periods can be synchronized

Appropriate values for CPU time quantum and period have to be determined by experiments. The second requirement maintains the synchronization of all threads working on the same stream as shown in Figure 4.4. Synchronized periods provide threads with a mutual time base.

Defining thread priorities is the job of the scheduling API. Arbitrary thread preemption can be performed as long requirement number 1 is preserved. To attach higher priorities to threads with short periods is a common scheduling policy.

The preconditions for the usage of classic real-time scheduling methods RMS or EDF [LL73] are fulfilled. These methods only allow sets of independent periodic processes (threads) that do not require inter-process synchronization. Due to double buffering, threads are independent of each other within their synchronized periods. Each thread is ready at the beginning of each period and wants to consume a CPU time quantum that of course is less or equal than the period time. The deadline is the end of a period. The CPU time quantum and the period are constants. To apply RMS static priorities have to be assigned. To implement EDF, the scheduler always has to switch to the thread with the earliest deadline. Instead of these two approaches, another, most likely more complex scheduling policy may gain advantages like a reduced number of context switches, as long as all requirements are met.

Even hardware interrupts can be handled by the described scheduling scheme, as long as they occur periodically that is true in the case by USB host controllers.

In the case of video streams a quality of 100% is not always needed. Then the model specified in [HLR+01] can be applied, which splits the requirements in a mandatory and an optional part, which should be available as often as possible but at least with a certain percentage. This distinction will not be made in this work because it only deals with hard real-time components.

A more general approach for resource reservation and management is presently developed at the Dresden University of Technology within the scope of the COMQUAD [COM03] project. In short, besides interfaces for offered services, COMQUAD components provide additional interfaces for description and negotiation of quantitative properties. Once the COMQUAD project

is realized, it may offer adequate support for time slice donation and can be used to implement the scheduling policy needed for the scenario mentioned above.

Locks - Always a Problem

The USB core uses some lists containing processing (scheduling) information, which are protected by locks. The access to these list is performed in very short critical sections called in interrupt and interface context. Thus there is always a small chance for interferences between a running interrupt handler and API calls. So it is necessary to prevent priority inversion where a low-priority thread that blocks a high-priority thread is preempted by a mid-priority thread.

As stated in [Hoh02], wait-free synchronization can also be implemented using locks, albeit with a nonblocking helping scheme. Helping is a family of mechanisms that implement wait-free synchronization. When a higher-priority thread A's critical section detects an interference with a lower-priority thread B, A helps B to finish its critical section first, effectively lending its own CPU time to B. During helping, A also lends B its priority to ensure that no other, lower-prioritized activities can interfere. When B has finished, A executes its own critical section. For instance, helping can be implemented by the Priority Inheritance Protocol or the Priority Ceiling Protocol [SRL90]. This concept is successfully implemented by the Fiasco micro kernel to protect critical sections. As result, Fiasco achieves good preemptibility.

A locking scheme with priority inheritance can be considered as a wait-free synchronization scheme as long as critical sections never block. In the case of the examined Linux USB stack this precondition is fulfilled. Critical sections are very short and not nested. Device driver code will not be entered within critical sections. So, to avoid interferences, locks have to be implemented together with priority inheritance.

4.2.7 Security Issues

The placement of USB core and the device drivers in different components (separate address spaces) provides a basic protection from undesirable memory access. In contrast to the original Linux environment, the control data structures will not be shared anymore. This allows to insert effective security checks at API call level. But there are other security issues left:

For security reasons data sharing between the USB core and the device drivers has to be restricted to the actual need. As described in Section 4.2.5, the device drivers share their transfer buffers with the USB core only for performance reasons. These buffers need only to be attached and not to be mapped and thus cannot be inspected or modified by an untrusted USB core.

In L^4Linux, transfer buffers can be located elsewhere in the kernel address space, which is currently completely shared with the USB core. Since real sharing (mapping) is not performed, both components are protected from each other.

While transfer buffers cannot be accessed by USB core directly, the host controller is still able to perform direct memory access (DMA). An extensive approach to use the untrusted USB core in a trusted environment is to introduce an additional control instance that checks the USB schedules for illegal physical addresses before they will be processed by the host controller.

How will denial-of-service attacks, started by malicious or malfunctioning drivers, affect the USB core? Due to the usage of a limited number of preallocated worker threads for each client driver, the services of the core cannot be completely claimed by a single client. Sooner or later a

driver will reach the worker thread limit and will be blocked. The USB core can treat this event as an attack and is free to disconnect the driver. Sending IPCs to worker threads dedicated to other clients can also be detected by the USB core and treated in the same way.

4.3 L^4Linux Stub

According to Figure 3.1, stubs for L^4Linux can be supplied at all three levels of the USB stack.

- host controller driver (HCD) stub
- USB core stub
- device driver stub

As already stated in Section 3.1 the host controllers are interacting with the USB core using a undocumented API. This and the reasons for co-locating the HCD with the USB core as enumerated in Section 4.2.1 advise not to implement a virtual HCD for L^4Linux.

The ported USB core already implements the complete USB-API. To export this API by a USB core stub allows to run all available USB drivers within L^4Linux without modifications. All requests by L^4Linux that will not endanger the real-time capabilities at DROPS side can be scheduled. Because of its advantages the USB core stub was implemented. Features like hot-plugging and USB device file system are currently not implemented in the USB core and thus cannot provided for L^4Linux.

If there are plans to use a USB device by DROPS and L^4Linux in parallel, developing a stub for USB drivers is always possible and does not effect the stubs at other levels. For example, the OVCam driver may export its V4L interface to L^4Linux.

4.4 Porting the OVCam Driver

The OVCam driver comes with a separate decompression module, which is very small. Compression is strongly needed for higher frame rates at higher resolutions. To simplify the port the compression module will be co-located to the web-cam driver. As in the case of USB core, the Linux kernel environment will be provided by DDE. The USB functionality can be accessed by linking against the USB library.

There is one question left: how to deal with the V4L (see Section 3.2.1) interface? All OVCam driver properties except for the module parameters can be controlled by capture applications via the V4L interface, which turned out to be quite usable and well documented. This suggests not to create another new interface. So an IPC wrapper exports a subset of the V4L interface to the V4L application. For the transmission of the data stream DSI (see Section 2.4.3) is used, which establishes shared buffers and extends the original V4L interface about real-time properties.

Because it is possible to use the OVCam driver under L^4Linux, there is no need to implement an own L^4Linux stub for it. The web-cam driver can only be used by one application at the same time. It provides no benefit to feed non real-time applications (L^4Linux) with the output of real-time components (OVCam under L4).

5 Implementation

In this chapter selected problems that have occurred during the implementation will be discussed. More detailed information can be gathered from the source code.

5.1 DDE Extensions

The ported Linux drivers need some Linux kernel functionality that was not implemented in the DDE. Therefore, DDE has to be extended to support memory management functions for kmem caches and PCI pools. The kmem caches were easily mapped to l4slab functionality, inheriting their limitation to a maximum slab object size of the page size (4096 bytes). This limit is sufficient in the most cases and in particular for the ported Linux USB subsystem.

Additionally, DDE was updated to support Linux conforming kernel threads. The Linux USB subsystem installs a kernel hub daemon, which is permanently listening for port status changes.

5.2 Scheduling

To get the port running a fixed scheduling scheme based on priorities is used. This is generally hard-coded into the system implementation and cannot be easily adapted. The introduction of new components makes it necessary to rethink the priorities of all threads running in the system.

L4, and so its implementation Fiasco that underlies DROPS, currently has a prioritized multi-level round-robin scheduling. This alone is insufficient to construct a non-trivial real-time system [Elp01]. An appropriate processor scheduling framework is needed, which includes at least the CPU time reservation (see Section 4.2.6) and avoids priority inversion. Since even thread preempters are not yet supported by Fiasco, user level scheduling is currently not possible, priority inheritance (helping) as well.

5.2.1 Thread Switching Pitfalls

While sending an IPC, the micro kernel Fiasco automatically switches to the receiver regardless of the thread priority of both threads. The caller thread donates the rest of its time slice to the receiver. This works fine in standard client server scenarios. In special cases, even if they are performance critical, this behavior is not desired:

In its completion handler (interrupt context) the OVCam driver collects raw data buffers until a frame end token is received. Then the *bottom half* handler is started to process the data. The *bottom half* thread sleeps in a waiting queue until the completion handler wakes it up. Because under Linux no scheduling will be done in interrupts, the *bottom half* handler will just be moved to the ready queue and the time critical completion handler will proceed. In DDE, which

provides the wait queues functionality for the OVCam driver, waiting queues are implemented using semaphores. Thus, waking up a thread implies sending an IPC. The completion handler is suspended regardless of its higher priority. Priority driven scheduling is not preserved. After the *bottom half* handler is woken up, it has relatively time consuming calculations to perform (decompression, format conversions). In other words, it will not release the CPU until its time slice expires. In the meantime the interrupt thread (completion handler) is blocked. In Fiasco a time slice may last up to 1 ms what is the same period in that already a new USB interrupt may occur.

To cope with this problem there are two options. Inserting a `schedule()` right after the sleeping operation of the *bottom half* handler will immediately switch back to the completion handler. The desired thread switching semantic can also be achieved by using a special Fiasco option. The send descriptor contains a so-called *deceit bit*. Since the mechanism according to this bit is not implemented in Fiasco, the bit was rededicated and only the name remained. If the *deceit bit* is set, an IPC leads not to a switch if the destination thread's priority is lower than the priority of the sending thread. In this case, the destination thread is only enqueued into the ready queue and the sender can proceed. To activate the alternative switching behavior, Fiasco and the semaphore package have to be compiled with according options.

5.2.2 Critical Interrupt Path

In the worst case scenario the completion call of an URB has to be performed every 1 ms. To avoid data loss the maximum completion call processing time must be significantly smaller then 1 ms. If a completion call is not delivered in time, the host controller hardware may overwrite transfer buffers not yet processed by the according device driver. Fast completion calls are even more important if the transfer of several URBs is finished at the end of the same frame.

While running in interrupt context the USB core passes the control flow to the driver and blocks for an unpredictable time period. This problem can be solved by setting the send and receive timeout for the completion call IPC. Since the completion call provides no return values, all possible damage is bounded to the driver, which can miss a completion call. If a send timeout occurs, the USB core may retry to send the completion message later, which is at least not reasonable for isochronous transfers that are delivered continuously. Just sending completion messages (not calling the completion handler) works in the most cases, but breaks with the conventional call semantic in the same way as deferred completion calls.

5.3 L^4Linux Stub

The L^4Linux stub uses nearly the same USB library implementation as DROPS USB device drivers do. This library runs within the same task together with L^4Linux and exports the USB-API symbols to the kernel. Some of L4Env functionality used by the library (threads, semaphores, memory management) needs to be initialized, what is done at the start of L^4Linux. The regarding symbols have to be exported by the L^4Linux kernel. To share the access to L4Env functionality with L^4Linux only the L4Env variant of L^4Linux can be used.

From the view of the USB core, L^4Linux virtually behaves as a client implementing several driver interfaces. Since the USB core is capable to register a almost unlimited amount of clients,

it is possible to run several instances of *L⁴Linux* with USB support in parallel. The number of required worker threads for each *L⁴Linux* instance can only be estimated (see Section 4.2.3).

Another problem is to obtain equivalent dataspaces for the provided transfer buffers. An interim solution is to share the whole kernel address space with USB core as mentioned in Section 4.2.5.

5.3.1 Transferring USB-API Structures

In Section 4.2.4 the decision was made to build up extra copies of the used USB structures at client and server side. Even though the source codes were not modified, the binary compatibility cannot be assured because both components are compiled with different header files. While the USB core is compiled against the DDE Linux kernel sources, the *L⁴Linux* stub uses of course the *L⁴Linux* kernel sources. As consequence, semaphores and locks, which are part of some structures, have different sizes at both sides. So it is inconvenient but necessary to convert all transferred structures.

Structures used by the USB-API have to be serialized (see Section 2.1.5). There are a lot of tools for serializing dynamically linked structures. Those are available in DROPS do unfortunately not meet the requirements for three reasons. First, the structures have to be converted as mentioned in the last paragraph. Second, the USB-API uses dynamically sized structures (e. g. URB). This means, that the size of a structure only can be determined at run-time (`sizeof()` will not work). References to certain substructures, which are only valid in the address space of the sender, have to be stored for future reference mapping. Finally, a self-made implementation for serialization of the required structures was developed.

5.3.2 Stub Call Propagation

In general, the *L⁴Linux* USB stub propagates all USB-API calls to the USB core. The same is true for the callbacks. Problems arise from the transition of the control flow from *L⁴Linux* to the DROPS environment, where the USB library runs, and vice versa.

In the case of USB-API calls, *L⁴Linux* kernel threads cannot call flick IPC stubs directly. The kernel would be blocked until the call returns. This would only affect the performance of *L⁴Linux*, but there are callbacks that can occur in the context such a stub call. In this case, *L⁴Linux* is not ready to receive callbacks.

The solution is to block the calling *L⁴Linux* kernel thread in a Linux conformable way and let an L4-thread do the call instead. The *L⁴Linux* thread has to wait until the L4 thread has finished the call. Only *L⁴Linux* synchronization primitives can be used to block a *L⁴Linux* thread properly, but L4 threads cannot access them without the risk of calling *L⁴Linux* scheduling functions.

The implemented solution for this problem is a synchronization mechanism using a memory cell, which is periodically polled for state changes. The waiting *L⁴Linux* thread loops around a `schedule_timeout()` call until the memory cell has changed. Depending on the specified timeout period, this solution introduces a delay for all non-time-critical calls which may cause callbacks: `usb_register()`, `usb_unregister()` and `unlink_urb()`.

All remaining API calls never cause callbacks and can be invoked by the *L⁴Linux* kernel directly. The only time-critical API call `submit_urb()` can thus be called directly, too.

Callbacks to *L*^{4}*Linux*

The also non-time-critical callback calls `probe()` and `disconnect()` are expected to be called from the process context. This can only be done by a valid *L*^{4}*Linux* thread, which is created by the *L*^{4}*Linux* stub at initialization time. The synchronization works in the same way as for the API calls.

Completion calls, which always have to be considered as time critical, can be directly processed by the worker threads of the callback server, which is part of the USB library. This approach works, because the completion handlers of the device drivers are designed to run in Linux interrupt context and will therefore not use any Linux scheduling function.

5.4 Web-cam Viewer

5.4.1 Source File Modifications

The OV5Cam driver shares the frame buffer with V4L applications and provides transfer buffers for the USB subsystem. In the DROPS world, shared buffers between applications will be established by sharing dataspaces. In the case of the V4L frame buffers the driver tries to access Linux low-level internals like page table entries. To emulate such Linux details in DDE would go too far. Therefore, the memory allocation calls for all shared buffers were identified and replaced.

5.4.2 Module Parameter Passing

The behavior of kernel modules can sometimes be significantly influenced by module parameters. For example, the configuration of the OVCam driver filters will be performed this way. In Linux, kernel module parameters will be set during loading. Modules in the DROPS world will be loaded by GRUB[1] or the L4 loader, which both not are able to set module parameters. To specify module parameters at least at compile time without modifying the driver's source code, the driver is compiled with an adapted `MODULE_PARM` macro definition that creates a freely accessible variable for every module parameter. These parameters can be set during initialization by the wrapper layer.

5.4.3 DOpE Application

The web-cam viewer was integrated in the DOpE example application *vscreentest*, which originally shows four attractive visual effects in separate windows (widgets). One of these windows is now used to view the video stream provided by the OVCam viewer. Buttons to change picture properties like brightness and color were added to this window. Additionally, the web-cam viewer makes use of DOpE's sophisticated features like window content scaling.

[1] GNU Grand Unifi ed Boot loader

6 Evaluation

As result of the development both UHCI and the OHCI host controller driver implementations were successfully tested together with the OVCam driver. Further details are described in this chapter.

Design Goals Review

At this place the achieved results has to be reviewed considering the design goals defined in Section 4.1. The USB core and the OVCam device driver were placed in separate modules and interact via IPC interface. This modular concept allows to load device drivers only if needed. The goal to use unchanged Linux USB source codes was successfully achieved. The OV5Cam had to be minimally modified (see Section 5.4.1).

Real-time properties were at least achieved for systems with minimal workload. Problems that had to be solved are discussed in Section 5.2. To assure real-time characteristics even in systems with higher workload, a scheduling API that meets the requirements presented in Section 4.2.6, has to be provided by the run-time environment.

The achieved security level is described in Section 4.2.7 and a performance analysis will be presented in the next sub-chapter.

Resource Consumption

The USB library, which is linked against every device driver, needs approx. 60 KB for code and 10 KB for dynamically allocated memory. The USB core consists of approx. 300 KB code and additional 10 KB dynamically allocated memory for each registered device driver. Compared to the original Linux drivers, the progression of the memory footprint can be marked as low.

In contrast to the memory footprint, the usage of threads is not very economical. The need to introduce worker threads (see Section 4.2.3) leads to the reservation of at least two worker threads per registered device driver. These threads are statically allocated and will be used very rarely, if at all.

6.1 Measurements

Before quantitative results will be presented, the environment properties for all following benchmarks will be reported. The test system featured following characteristics: Intel Celeron (Coppermine), 900 MHz, 128 MB main memory, Intel i815 chip-set (Camino 2), USB controller implementing UHCI. The Fiasco configuration was: L4-V2 ABI, gcc 2.95, assembler IPC short cuts. The implementation uhci.c of the UHCI host controller driver was used.

Call Overhead

Since no changes to the source code were applied, the ported USB stack is expected to consume only slightly more CPU cycles than the original in the Linux environment. Sure, each API call and each callback costs at least one context switch (call IPC). Additionally, all transferred parameters have to be encoded and decoded. All other activities inside the USB wrapper layer involve only accessing data structures and parameter conversions. There are no algorithms (loops) executed featuring unpredictable processing time. The way that DDE influences the performance cannot be estimated.

The USB-API functions called most often are `usb_submit_urb()` and the completion callback (see also Section 5.2.2). The latter one is optional but requested in nearly all cases. The reason is that applications want to stay closely synchronized to the hardware activities. It is sufficient to measure only the performance drawbacks of this calls, because during running real-time URB data transfers all interactions between the core and the driver will be performed using these functions. Since actual processing times depend on the implementation of the USB core (`usb_submit_urb()`) and the driver (completion call), it would be reasonable to measure only the call overhead caused by the USB wrapper layer (see table 6.1). All other API calls are only used for configuration and thus can be considered as not performance relevant.

	USB layer	IPC & stub	$\sum$
`usb_submit_urb()` (μs)	8.9	4.6	13.5
completion call (μs)	0.6	6.8	7.3

Table 6.1: Call Overhead

The term IPC costs summarizes the time needed for encoding, copying and decoding of the IPC parameters (about 100 bytes) and the context switch. The USB layer needs 6 μ of 8.9 μ to call `l4rm_lookup()` for determining the offset of a given transfer buffer within the according dataspace. This IPC costs can be saved by caching the dataspace parameters locally. Without doubt: the IPC costs are the limiting bottleneck of the presented implementation.

Since the measured call overhead is negligible compared to the USB frame rate of 1 ms, the additional CPU time needed by the USB layer will never impact the performance.

If a driver wants to share dynamically mapped transfer buffers (see Section 4.2.4) three additional IPCs are required: getting subdataspace, attaching of the dataspace at USB core side and determining of the physical address. Because getting subdataspaces is currently not possible, this benchmark was skipped.

OVCam Driver Benchmarks

In all tests the web-cam captures frames with a resolution if 576x432 pixel and 16 bit color depth. About 94% of the limit for isochronous transfers was used. This implies an overall USB utilization of 65%. All benchmarks were conducted on otherwise idle Linux systems with minimal workload. To void interference all unneeded activities were omitted. The results of course depend on the web-cam used, which was a Terratec TerraCAM USB Pro in this case.

Table 6.2 allows to compare the characteristics of the USB subsystem (Linux kernel 2.4.20) and the OVCam (version 1.63) web-cam driver in different environments.

Operating System USB subsystem	DROPS USB port	L4Linux USB stub	L4Linux native	Linux native
av. `usb_submit_urb()` execution time (μs)	30	40	7	7
av. completion call execution time (μs)	80	72	67	67
frame rate (fps)	15.0	15.0	15.0	15.0

Table 6.2: OVCam Driver Capturing Performance

The presented values are driver specific. Since the processing times varies from call to call, the table presents average values. Extremal values (criterion: 10 times higher than the average) caused by preemption by higher prioritized threads (interrupts) were not considered.

Regarding to the used measurement precision, the results for the native Linux USB core running in the Linux and *L*^{4}*Linux* environment are the same. This is comprehensible since exactly the same code was executed.

The performance loss for the ported USB subsystem matches with the call overhead in Table 6.1. Although thousands of test loops were performed, the impreciseness may be caused by the average value approach and different test environments.

All tests achieved the same frame rate of 15 Hz, which is limited by the chosen capture resolution and the web-cam hardware. Resolutions up to 320x240 would allow frame rates of 30 Hz.

Measurements in configurations that not made use of the *deceit bit* (see Section 5.2.1) brought unacceptable results. In worst cases, completion calls stalled the USB core for more than 1 ms.

Visual impression

To assure real-time performance DOpE does not allow applications to access the frame buffer memory directly. Instead the application writes in a hidden buffer that is drawn by DOpE to the screen periodically. This period is determined at compile time. Currently the reasonable value of 25 Hz is standard.

The frame-rate delivered by the OVCam driver depends on the resolution and varies between 3 and 40 Hz. As result, the window frame rate and the capture frame rate are not synchronized to each other and the time a frame is shown varies by one frame. This may explain the slightly smoother animation in *L*^{4}*Linux*, where the video stream is directly copied to the frame buffer. The measured results presented in Figure 6.2 do not show any significant performance differences of the OVCam driver in both environments.

7 Conclusions and Future Work

The result of this work is a nearly 100% Linux USB-API compatible USB stack for DROPS. Only the way to handle the transfer buffer management had to be changed. Linux USB stack's asynchronous callback mechanism makes the current implementation vulnerable for unpleasant delays caused by inappropriate thread scheduling. All remaining USB bandwidth can be used up by *L*^{4}*Linux*.

Altogether, to port further Linux USB device drivers should be easily feasible as the OVCam driver port shows. The ported drivers demonstrate again the usability of DDE and DSI.

Future Work

There is a need for using USB mass storage devices (hard disk, USB key) in DROPS. Mass storage devices will be handled by a single class driver and only support bulk transfers, which are not designed to meet real-time constraints by the standard and the current Linux implementation. A USB-to-serial adaptor device would allow debugging in cases where a serial interface is missing or already used. For instance, the DOpE mouse pointer can get its input by a USB mouse.

The support of USB 2.0 compliant EHCI host controllers seems to be possible even with the current implementation, but has not been tested yet. With a current market share of 50%, USB 2.0 will dominate in the future.

The OVCam driver may be used as data stream source for the upcoming DROPS video recorder application. If more Linux device drivers that provide the V4L-API will be ported in the futures, the demand for a V4L registrar application will increase.

The current Linux kernel development version (2.5.) introduces small USB-API changes but no completely new mechanisms. Thus, an update seems to be present no difficulties. The further development of the L4 USB stack depends on the Linux kernel versions supported by DDE and *L*^{4}*Linux*.

Acronyms

ABI Application Binary Interface

API Application Programming Interface

CPU Central Processing Unit

DDE Device Driver Environment

DMA Direct Memory Access

DOpE Desktop Operating Environment

DROPS Dresden Real-Time Operation System

DSI DROPS Streaming Interface

EDF Earliest Deadline First

EHCI Enhance Host Controller Interface

HC Host Controller

HCD Host Controller Driver

IDL Interface Definition Language

IPC Inter Process Communication

IRP I/O Request Packet

L4 Second generation micro-kernel interface

L4Env Library collection providing operating system primitives for DROPS

OHCI Open Host Controller Interface

OVCam Web-cam driver for the OmniVision OV5xx series of chips

PCI Peripheral Component Interconnect

RMS Rate Monotonic Scheduling

SMP Symmetric Multiprocessing

URB USB Request Block, Linux USB stub analog of IRP

USB Universal Serial BUS

UHCI Universal Host Controller Interface

V4L Video for Linux

Bibliography

[ALE+01] Mohit Aron, Jochen Liedtke, Kevin Elphinstone, Yoonho Park, Trend Jaeger, and Luke Deller. The SawMill Framework for Virtual Memory Diversity. January 2001.

[BBH+98] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, Michael Hohmuth, Lars Reuther, Sebastian Schönberg, and Jean Wolter, editors. *Dresden Realtime Operarting System*. Dresden University of Technology, March 1998. http://os.inf.tu-dresden.de/drops/doc.html.

[CHPI+00] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. *Universal Serial Bus Specification - Revision 2.0*, April 2000. http://www.usb.org.

[CIMN98] Compaq, Intel, Microsoft, and NEC. *Universal Serial Bus Specification - Revision1.1*, September 23 1998. http://www.usb.org.

[CMS99] Compaq, Microsoft, and National Semiconductor. *Open Host Controller Interface Specification for USB - Revision 1.0a*, September 1999. http://www.usb.org/developers/docs/.

[COM03] COMQUAD - Components with quantitative properties, February 2003. www.comquad.org.

[Elp01] Kevin Elphinstone. Proposed L4 scheduling behavior to support real-time system construction, February 2001.

[Fen02] Norman Fenske. DOpE - a graphical user interface for DROPS. Master's thesis, Dresden University of Technology, October 2002. http://os.inf.tu-dresden.de/project/finished/finished.xml.

[Fli00] Detlef Fliegl. *Programming Guide for Linux USB Device Drivers - Revision 1.32*, 2000. http://usb.cs.tum.edu/usbdoc/.

[Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Master's thesis, Dresden Univerity of Technology, July 2001. http://os.inf.tu-dresden.de/project/finished/finished.xml.

[HLR+01] Claude-Joachim Hamann, Jork Löser, Lars Reuther, Sebastian Schönberg, Jean Wolter, and Hermann Härtig. Quality-Assuring Scheduling - Using Stochastic Behavior to Improve Resource Utilization. Technical report, Dresden University of Technology, Dezember 2001. http://os.inf.tu-dresden.de/drops/doc.html.

[Hoh96] Michael Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, Dresden University of Technology, August 1996. http://os.inf.tu-dresden.de/project/finished/finished.xml.

[Hoh02] Michael Hohmuth. *Pragmatic nonblocking synchronzation for real-time systems*. PhD thesis, Dresden University of Technology, October 2002. http://os.inf.tu-dresden.de/drops/doc.html.

[Int96] Intel. *Universal Host Controller Interface (UHCI) Design Guide - Revision 1.1*, March 1996. http://www.usb.org/developers/docs/.

[Int02] Intel. *Enhanced Host Controller Interface for Universal Serial Bus - Revision 1.0*, March 2002. http://www.usb.org/developers/docs/.

[Kel01] Hans Joachim Kelm, editor. *USB 2.0*. Number 3-7723-7965-6. Franzis, 2001.

[LHR01] Jork Löser, Hermann Härtig, and Lars Reuther. A streaming interface for real-time interprocess communication. Technical report, Dresden University of Technology, 2001. http://os.inf.tu-dresden.de/drops/doc.html.

[Lie96] Jochen Liedke. L4 Reference Manual - Version 2.0. Technical report, GMD - German National Research Center for Information Technology / IBM Watson Technical Report, September 1996. http://os.inf.tu-dresden.de/L4/l4doc.html.

[LKS] Linux kernel sources. http://www.kernel.org/.

[LL73] C. L. Lui and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. 1973.

[LUP] Linux USB project home page. http://www.linux-usb.org/.

[McC] Mark McClelland. Linux OVCam Drivers. http://alpha.dyndns.org/ov511/.

[Sch02] Sebastian Schönberg. *Using PCI-Bus Systems in Real-Time Environments*. PhD thesis, Dresden University of Technology, June 2002.

[SI97] SystemSoft and Intel. *Universal Serial Bus Common Class Specification - Revision 1.0*, December 1997. http://www.usb.org/developers/devclass_docs.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance proctocols; an approach to real-time synchronsization, September 1990.