

GROßER BELEG

USB for the L4 Environment

Dirk Vogt

23. September 2008

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur für Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Björn Döbel

Aufgabenstellung

Ziel der Aufgabe ist der Entwurf und die Implementierung einer USB-Infrastruktur für das L4 Environment. Hierbei sollen bestehende Vorarbeiten zur Integration von Gerätetreibern, Nutzung von USB und Sicherheit bei Gerätetreibern berücksichtigt werden.

Insbesondere sind folgende Teilaufgaben zu lösen:

- Analyse der bestehenden Arbeiten, insbesondere unter dem Gesichtspunkt von Sicherheit und dem Einsatz in Eingebetteten Systemen
- Einbindung eines USB-Stacks (FreeBSD, Linux o.ä.) in das L4 Device Driver Environment
- Entwurf und Implementierung einer geeigneten Infrastruktur für Server, die USB-Dienste nutzen:
- generische Unterstützung der USB-Hotplugging-Fähigkeit innerhalb des L4Env
- Beispiel-Implementierung eines Treibers unter Nutzung der geschaffenen Infrastruktur

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 23. September 2008

Dirk Vogt

Acknowledgments

I would like to thank Prof. Hermann Härtig for the opportunity to work at the TU-Dresden Operating Systems Group and all members of the group who contributed to this work. Especially my supervisor Björn Döbel for his professional support, patience, and 'this and these' hints to improve my writing.

Further I would like to thank the students in the lab for their companionship and fast help.

Last but not least I would like to thank Christine Oeser, not only for her patience.

Contents

1	About this Document	1
2	Background and Related Work	3
2.1	The Universal Serial Bus	3
2.1.1	USB Hardware	4
2.1.2	USB Protocol	6
2.2	DROPS	12
2.2.1	The Microkernel Approach	13
2.2.2	L4Env—An Environment for L4 Applications	13
2.2.3	DDE—Device Driver Environment	14
2.3	The Linux USB Stack	16
2.3.1	Linux USB Stack Architecture	16
2.3.2	Linux USB Device Driver API	17
2.3.3	Linux USB Host Controller API	17
2.4	Related Work	19
2.4.1	USB for DROPS	19
2.4.2	USB/IP	20
3	Design	23
3.1	Design Goals	23
3.2	DDEUSB - A USB Framework for L4Env	24
3.2.1	The DDEUSB Server	24
3.2.2	DDEUSB Client Side	29
3.2.3	Access Control	31
3.2.4	Forwarding USB devices	31
3.2.5	Communication Mechanisms	32
4	Implementation	37
4.1	Porting the Linux USB Stack	37
4.1.1	Platform Devices	37
4.1.2	DMA from the Stack	38

Contents

4.1.3	Physical to Virtual Address Translation	38
4.2	The DDEUSB Implementation	38
4.2.1	Caching of URB Data Structures	38
4.2.2	Improving DDEUSB's IPC Performance	39
4.3	DDEUSB Example Applications	42
4.3.1	USBHID	42
4.3.2	USBCAM	44
4.4	L ⁴ Linux as DDEUSB Client	45
4.5	Isochronous Transfer	46
5	Evaluation	49
5.1	Evaluation of Time and Effort	49
5.1.1	Porting the Linux USB Stack to dde_linux26	49
5.1.2	DDEUSB Server Side	49
5.1.3	DDEUSB Client Side	50
5.1.4	Porting Linux USB device drivers to dde_linux26	50
5.2	Performance Evaluation	51
5.2.1	Test Setup	51
5.2.2	Discussion of the Test Results	52
6	Conclusion and Future Work	61
	Index	63
	Bibliography	67

List of Figures

2.1	USB Transactions	8
2.2	USB 1.x Frameneration	9
2.3	USB Descriptor Hierarchy	11
2.4	L4Env Components	14
2.5	DDE Architecture	15
2.6	Linux USB Stack	16
2.7	USB for components	19
2.8	USB/IP Architecture	21
3.1	DDEUSB Example Session	26
3.2	DDEUSB Architecure	29
3.3	Communication Methods	33
4.1	USBHID Architecture	43
4.2	USBCAM Architecture	45
5.1	System Comparison, Hard Disk, Bonnie++	52
5.2	Transfer Mode Comparison, Hard Disk, Bonnie++	54
5.3	Comparsion of dd, Hard Disk	56
5.4	Comparison of dd, Flash Drive	57
5.5	Relative Comparison of dd	58

1 About this Document

The universal serial bus (USB) is a widely used bus to connect peripheral devices to today's computer platforms. DROPS, the Dresden real-time operating system, has support for USB, but as I will show in this work, the current solution does not fulfill today's requirements.

In the fast changing world of software, there is a need for generic and maintainable solutions that can easily be adapted to new use cases and are easy to integrate into existing and new projects. Thus, there is a requirement for a new USB infrastructure for DROPS.

In this work, I will present DDEUSB, a replacement of DROPS' current USB infrastructure, which is designed to fulfill the aforementioned requirements, and is able to offer access to USB with an overhead of 2.4 percent compared to L⁴Linux and 6.4 percent compared to native Linux.

Structure of this Document

In the next chapter I explain the foundations that are required to understand the rest of this document. This includes an overview on USB as well as an introduction to DROPS. Further, I explain the fundamentals of the Linux USB stack and finally, I give an overview of related work.

Chapter 3 presents the design of DDEUSB, the new USB infrastructure for DROPS, and explains how this solution can easily be integrated into existing projects.

In Chapter 4, I show, which steps were necessary to implement DDEUSB, and which problems occurred during implementation.

Finally, in Chapter 5, my project is evaluated regarding aspects of effort and performance and in the end, Chapter 6 summarizes this work, and suggests future improvements.

2 Background and Related Work

In this chapter, I give an overview of important foundations for my work. First, I present the basics of the universal serial BUS (USB) and how the USB protocol is defined. Afterward, I introduce the Dresden real-time operating system (DROPS) and its device driver framework DDE. Then, I give an introduction to the Linux USB stack and at the end of this chapter I present other projects related to my work.

2.1 The Universal Serial Bus

The interfaces used in the original IBM PC designs of the early 1980s had a number of problems. For example, there existed a wide diversity of connectors. Furthermore, most of these interfaces were not hot pluggable. Limited system resources, which had to be shared by more and more devices, appeared to be another problem.

One approach to handle these problems is USB, whose specification was first published in early 1996 by a consortium of IT companies including IBM and Microsoft [CDI⁺96].

To overcome the shortcomings of traditional peripheral interfaces, the designers of USB were striving for the following design goals:

- A single connector type for all PC peripherals,
- Hot plug support,
- Preventing system resource conflicts,
- Low cost for system and peripheral implementations,
- Automatic detection and configuration of peripherals,

Name	Speed	Available since
Low Speed	1.5 Mb/s	USB 1.x
Full Speed	12 Mb/s	
High Speed	480 Mb/s	USB 2.0
Super Speed	4.8 Gb/s	USB 3.0 (planned)

Table 2.1: USB transfer speeds

- Support for legacy hardware and software, and
- Low-power implementation.

USB can be used to connect a wide variety of peripheral devices ranging from input devices like keyboards and mice to more complex hardware like video frame grabbers and mass storage devices.

The first revision of USB introduced two transfer speeds, namely low speed using a transfer rate of 1.5 Mb/s and full speed using 12 Mb/s.

In the year 2000, a second revision [CHPI⁺00] of USB was published, which included high speed transfer at a data rate of 480 Mb/s. Nevertheless, USB 2.0 is fully backward compatible to USB 1.x.

It is to be expected that the third major revision of USB will be published in in the year 2009. This revision will also be fully backward compatible to previous revisions. Additionally, it will offer a yet higher transfer speed called super speed. Devices supporting this transfer speed will be able to transfer data at rates of up to of 4.8 Gb/s.

USB uses twisted-pair data wires for data transmission. Additionally, there are two more wires in a USB cable providing a supply voltage of 5 V to a USB device.

A good overview of USB can be found in the book *Universal Serial Bus Architecture* by Anderson and Dzatko [AD01].

2.1.1 USB Hardware

The hardware part of the USB mainly consists of three elements. These include the host controller with its root hub, USB hubs, and USB devices and are explained in the following.

Host Controllers

USB is a single master bus, which is managed by a single host controller (HC). Thus, all communication on the bus is controlled by the HC. The HC's task is to perform transactions that have been scheduled by the HC driver. For that, the HC's counter part, the HC driver, generates transfer descriptors (TD) and enqueues them for execution by the HC into the HC's transfer queue.

A HC always comes with a root hub to provide USB ports for one or more USB devices. The amount of USB ports can be increased by using USB hubs.

There are three important HC designs:

Universal Host Controller Interface (UHCI) This design was developed by Intel [Int96] and fulfills the USB 1.1 specification. It is mainly used in products by Intel and VIA Technologies. In comparison to other HC designs, more work is done in software, in order to reduce hardware complexity.

Open Host Controller Interface (OHCI) This design, developed by Compaq, Microsoft, and National Semiconductors [CMN99], also supports the USB specification up to revision 1.1. In contrast to the UHCI design, OHCI-type HCs do more work in hardware, which allows to provide a more abstract interface for driver developers.

Enhanced Host Controller Interface (EHCI) The EHCI design, which supports the USB 2.0 Specification was created by Intel [Int02]. Usually, an EHCI HC is equipped with one or more companion HCs for backward compatibility. Thus, if a USB 1.x Device is connected to a port of the root hub of an EHCI HC, this port will be forwarded to the corresponding companion HC. However, it is also possible to ensure backward compatibility without a companion HC. For that the HC's root hub has to support so called split transactions.

USB Hubs

Hubs can be used, to extend the number of available USB ports. They can be integrated into devices like keyboards and monitors or can be implemented as standalone devices. Furthermore, USB hubs can be bus-powered or self-powered. If a USB device is bus-powered, it does not need an external power source and will be powered by the bus.

Because a USB port only provides a limited amount of power (500 mA at 5 V), a bus-powered hub may only provide up to four USB ports. Otherwise, the hub has to be self-powered.

USB hubs play an important role in USB's hot-plugging mechanism, because they are also used to detect connection changes on the bus.

USB Devices

USB devices provide the actual functionality to the user. Their attributes are stored in descriptors. Whenever a new device is attached to the bus, the HC driver first reads these descriptors. Afterward, the HC driver uses the information contained in the descriptors to find a corresponding USB device driver for this device. This device driver can also use the device's descriptors to obtain more information on the device.

USB devices can provide one or more configurations, each of which can contain one or more interfaces. These interfaces have a default setting and may contain one or more alternate settings. These settings in turn can contain one or more numbered endpoints, which can be understood as source or sink of data. Each of these endpoints can either be ingoing, which means information flows from the device to the HC or, outgoing.

Again, like hubs, which are actually normal USB devices, USB devices can be bus-powered or self-powered, whereas, if a device is bus-powered, it may not consume more current than 500 mA.

2.1.2 USB Protocol

In this section I give an overview of how communication works on USB. This overview includes the existing transfer types and their purpose, how USB transactions are executed by the HC, and how the USB time-base, called frames, is generated. Afterward, I describe how USB devices present themselves to the USB software stack with the help of descriptors.

Transfer Types

USB provides the following transfer types, not all of which need to be implemented by USB devices.

Control Transfer Control transfer is used to configure a USB device and to control aspects of its operation. This transfer type is mandatory for USB devices. Every USB device must implement at least one control endpoint, which is endpoint zero.

HC drivers and USB device drivers use special request to endpoint zero for getting information on a device and managing it.

Interrupt Transfer This transfer type is used to periodically poll USB devices for data that needs to be transmitted. For that, the USB device driver specifies an interval, in which the USB device should be polled. This polling is done by the HC, and the USB device driver is not informed until data is available. The minimal polling interval for low speed devices is 10 ms, for full speed devices it is 1 ms, and for high speed devices it is 125 μ s. Interrupt transfer is often used for input devices, like keyboards and mice.

Isochronous Transfer Isochronous transfer is used whenever guaranteed bandwidth is needed, like in video frame grabbers or audio interfaces. Only full and high speed devices support this transfer type. The default setting of an interface may not include an isochronous endpoint. Furthermore, when an alternate setting of an interface is activated that includes an isochronous endpoint, the HC driver reserves the bandwidth needed, according to the USB specification.¹ If the required bandwidth is not available the activation of the alternate interface setting fails.

Up to 90 percent of the bus bandwidth are reserved for periodical transfer types, like isochronous and interrupt transfer.

Bulk Transfer Bulk transfer is used when there is no need of guaranteed bandwidth. It uses the remaining bandwidth of the bus, and is, for example, used for printers, scanners, and USB storage devices.

Transactions

A USB device driver's request to send or receive data on the bus is represented by an input–output request package (IRP). Whenever a device driver submits

¹ In this point Linux does not follow the USB specification: The Linux USB stack reserves the bandwidth not until the first isochronous transfer is taking place.

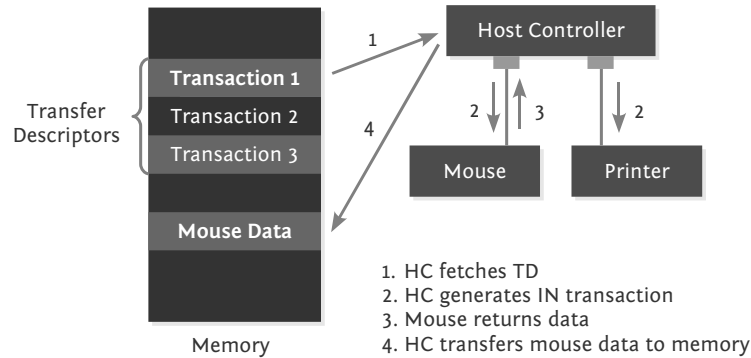


Figure 2.1: Conceptual view of a USB transaction

an IRP to the USB subsystem, the HC driver translated the IRP into one or more USB transactions. These USB transactions, in turn, are represented by transfer descriptors (TDs) and are scheduled by the HC driver.

A TD contains all the information needed by the HC to perform a transaction, including:

- The address for this transaction, consisting of the device's address on the bus, and the endpoint ID,
- The transaction type (e.g., ingoing/outgoing),
- The transfer speed,
- The number of bytes to be transfered, and
- The memory location of the transfer buffer, containing the actual data.

The HC driver enqueues the TDs into a linked list, which is called the frame list. During a certain interval (USB 1.x: 1 ms, USB 2.0: 125 μ s), called a frame, the HC fetches the TDs belonging to the current frame and executes them.

Figure 2.1 illustrates this process: On the left side, we can see the system's memory containing several transfer descriptors, and a memory location where data of the USB mouse should be stored. On the right side, there is a simple

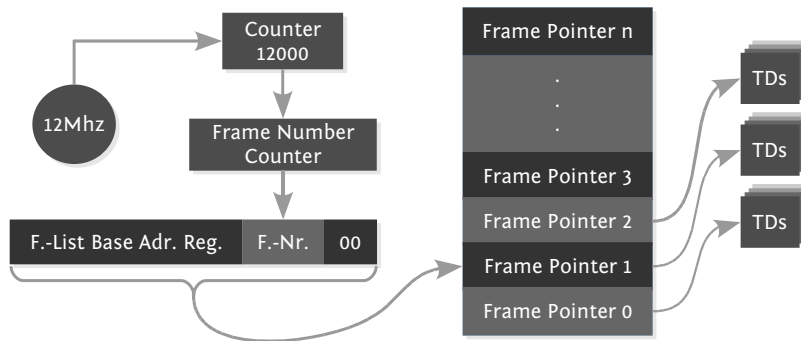


Figure 2.2: Conceptual view of frame generation in USB 1.x systems

USB setup, including a HC and two USB devices attached to it, a mouse and printer.

As the first step of this USB transaction, the HC grabs the transfer descriptor. Second, the HC generates the ingoing (i.e. reading) transaction, described in this TD. This transaction is addressed to the USB mouse. Although the transaction is received by all devices attached to the same USB port as the mouse, only the mouse is allowed to send a response, which it does in the third step. In the end, the host controller transfers the data sent by the mouse to memory.

Frame Generation

The HC is responsible to partition time into USB frames.

To generate the USB frames, a clock and a counter are used. Each clock tick increments the counter. When this counter reaches its limit, the frame number counter increments.

Because in USB 1.x systems a frame lasts 1ms, while the transfer rate is 12 Mb/s, a 12 MHz clock and a counter that counts up to 12 000 are used. According to the higher transfer rate of 480 Mb/s, and the frame interval of 125 μ s, in USB 2.0, a 480 MHz clock and a counter with a limit of 60 000 is used for frame generation. This setup is illustrated in Figure 2.2. On the left side we can see a 12 MHz clock incrementing a counter, whose carry output increments the frame number counter.

As we can further see, the frame number is used as offset, to generate the address of the next frame pointer, which points to a list, containing the transfer descriptors for the current time frame. The base for this address is to be set in the HC's frame list base register by the HC driver.

Configuration and Interfaces

USB devices have two levels of configurability: configurations and interfaces.

Configurations can, for example, be used to support a low power mode in a high power device. So, if no external power supply is attached to the device, the low power configuration can be used, only supporting a subset of the functionality. Otherwise the high power configuration can be used, then offering full functionality.

Interfaces, on the other hand, are used to access different functionalities of a device, for instance, the video function of a web camera and the built-in microphone. Furthermore, the different interfaces of a device can be driven by different drivers.

Descriptors

As mentioned before, USB devices use descriptors to present their features to software. These descriptors include:

Device Descriptor Every device provides one descriptor containing information on the device such as the manufacturer, the USB device ID and whether it is a full or a low-speed device. Furthermore, it contains information on the number of and references to the configuration descriptors this device contains.

Configuration Descriptor A USB device provides one configuration descriptor per configuration it supports. These descriptors contain information on the number of interfaces provided by the corresponding configuration. Further, they include references to the corresponding interface descriptors.

Interface Descriptor The interface descriptors hold general information on the corresponding interface, and on the number of endpoints included in an interface. An interface may include up to 15 endpoints, and further

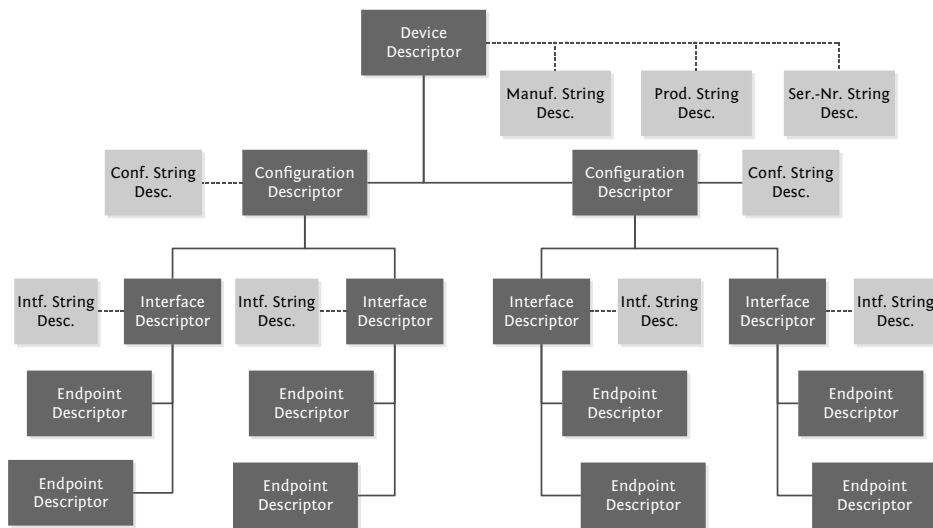


Figure 2.3: Example descriptor hierarchy for a USB device with two configurations

may include alternative settings. Alternative settings can for example be used to switch between different transfer bandwidths or to enable and disable endpoints.

Endpoint Descriptor An interface may contain multiple endpoint descriptors, each of which describes one endpoint, and this endpoint's attributes. These descriptors contain information on the transfer types supported by this endpoint (e.g., isochronous, control, ...), and on the maximum transfer rate.

String Descriptor String descriptors may be defined for the whole device, for certain configurations, and interfaces. They contain human-readable information about the corresponding device, configuration, or interface.

Class Descriptor If a device is implementing a USB device class, it may contain class descriptors containing class-specific information used by the USB-class device driver.

2 Background and Related Work

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video
0Fh	Interface	Personal Healthcare
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous
FEh	Interface	Application Specific
FFh	Both	Vendor Specific

Table 2.2: USB device class codes [USB06]

Device Classes

Device classes describe the exact behavior of devices belonging to these classes. So, devices implementing the behavior of a device class do not need a custom USB device driver, but may be driven by the corresponding class device driver.

The affiliation to a device class can be defined in the device descriptor, or in an interface descriptor. Table 2.2 gives an overview about the existing device classes.

2.2 DROPS

DROPS [HBB⁺98], the Dresden real-time operating system is a research operating system, developed at TU Dresden. Its main focus has changed from

real-time to security over the preceding years. However, security aspects also include real-time issues.

2.2.1 The Microkernel Approach

DROPS' kernel, called Fiasco [Hoh02, Hoh98], implements the L4-microkernel application binary interface specification, designed by Jochen Liedtke [Lie96]. First-generation microkernels, like MACH [ABG⁺86], had a rather bad performance compared to monolithic kernels like Unix and were still rather complex. Second generation microkernels [Lie95], like Fiasco, offer a better IPC performance, and are less complex.

In contrast to monolithic operating system kernels, like Unix or Linux, microkernels just offer basic primitives like threads, memory isolation through address spaces, and inter-process communication (IPC). However, microkernels do not implement policies. These policies have to be implemented as userland tasks, called servers, which communicate through IPC. The servers run in isolated address spaces and can be replaced at run time.

Andy Chou and colleagues [CYC⁺01] showed that device drivers form the biggest part of the code running in kernel mode, in today's operating systems. Further they showed that device drivers have the largest bug-density compared to other parts of the kernel. This implicates that moving drivers from the kernel into isolated user-level components, would lead to more reliable systems, because a defective device driver will not anymore crash the whole system.

With the microkernel approach, device drivers are removed from the kernel. Beside making the system more reliable, this makes also reduces the amount of code that has to be trusted. This altogether, allows to build small, flexible, modular, and secure systems, with a small trusted computing base.

2.2.2 L4Env—An Environment for L4 Applications

L4Env [Ope03] aims to provide a set of common functions to build userland processes on top of an L4-mikrokernel.

L4Env implements a set of L4 servers and libraries making use of these servers. Additionally, various development tools, like the DICE IDL compiler [Aig07] are included in L4Env.

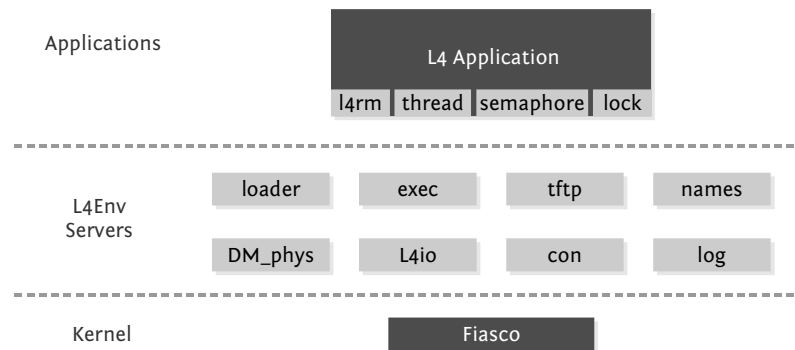


Figure 2.4: The components of L4Env. The lighter parts belong to L4Env.

Also memory management functionalities are provided by L4Env. So L4Env provides a dataspace manager (DM_phys) providing access to physical memory to L4 applications, and a region mapper library (l4rm), which manages the virtual memory regions of the address space and acts as the applications pagefault handler.

Further, L4Env offers libraries for thread creation, and synchronization mechanisms such as semaphores and locks. Also a small C-library and a utility library (L4util) are included.

In addition, there is a log server (log), a graphical console server (con), a root name server (names), and an input-output resource manager (L4io). L4io provides an interface to access input-output memory regions, input-output ports, ISA DMA channels, and the PCI configuration space.

In Figure 2.4, we can see an example setup of L4Env running on top of the Fiasco microkernel together with one application using L4Env.

2.2.3 DDE—Device Driver Environment

Device drivers are an integral part of operating systems, because they allow high level applications to access hardware devices in an abstract way. For most hardware devices, the drivers are delivered by the hardware manufacturer. However, these drivers are typically specific to the operating System.

For small operating systems, like DROPS, it is impracticable to implement new device drivers for each and every device. On the one hand, it is often hard

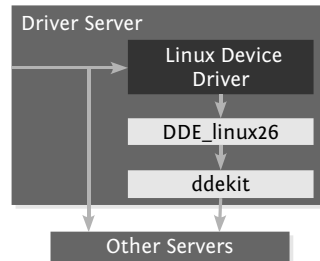


Figure 2.5: The DDE architecture. The actual DDE is illustrated by the lighter parts. The arrows stand for functional dependencies.

to get the specification of the hardware. On the other hand, scarce resources could be spent in a better way than writing device drivers.

Therefore, the reuse of existing device drivers seems to be more practicable. The Device Driver Environment (DDE) [Hel01] was developed for easy reuse of legacy device drivers. It implements the kernel application programming interface (API) of a driver’s usual operating system and maps these calls to L4Env primitives. The first version, developed by Christian Helmuth, was based on Linux version 2.4 and will be referred to as `dde_linux24` in this document.

A first port of the DDE for Linux 2.6 was done by Marek Menzer [Men04] in 2004. This version of the DDE will be referred to as `old_dde_linux26` in this document.

As a result of Thomas Friebel’s implementation of `dde_fbsd` [Fri06], the DDE was split into two parts, as illustrated in Figure 2.5. The first one, called `ddekit`, is independent of the legacy operating system and only relies on L4Env. It offers abstractions for commonly used functionality, like memory management and synchronization. Further, it can also be used as a foundation for writing non-legacy device drivers.

The operating-system-dependent part is located on top of the `ddekit`. This part implements the device driver’s familiar kernel API. At the time of this writing, there exist such back-ends for Linux (Version 2.6.20-19), referred to as `dde_linux26` and for FreeBSD, referred to as `dde_fbsd` in this document.

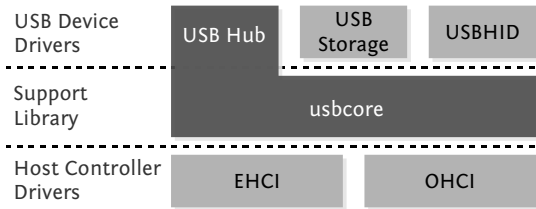


Figure 2.6: Illustration of the Linux USB stack's architecture.

2.3 The Linux USB Stack

As I explain in Chapter 3, the Linux USB stack was chosen as base of this work. Therefore, in this section, I the architecture of the Linux USB stack and introduce its APIS.

2.3.1 Linux USB Stack Architecture

The Linux USB stack can be divided into three parts, as illustrated in Figure 2.6:

- The actual USB device drivers,
- The host controller drivers, and
- A support library called `usbcore`.

The `usbcore` abstracts hardware specific and USB specific issues. It is implemented as a Linux subsystem. Further, the `usbcore` includes a device driver for USB hubs.

The actual USB drivers use the `usbcore` APIS. The `usbcore` implements two USB APIS targeting different types of USB drivers. These APIS are a high level API, used by USB device drivers, and a low level API, used by HC drivers. Thereby, the HC drivers are the only parts of the system touching the HC hardware (e.g., reading and writing registers, handling interrupts, doing direct memory access (DMA), ...). Like that, USB device drivers do not require knowledge about the used HC hardware.

2.3.2 Linux USB Device Driver API

The upper `usbcore` API is used by USB device drivers. The API is accessed by drivers calling functions that are defined in the Linux header file `linux/usb.h`. In this header file, the USB Request Block (URB) data structure is defined (see Listing 2.1 on the following page). This data structure represents an IRP, and is the only data structure used by USB device drivers to perform USB transactions. If a USB device driver wants to communicate on the bus, it first has to create a URB, and then has to submit it to the `usbcore`. After processing of this URB, a completion handler, specified in the URB, is called.

The functionality of the upper `usbcore` API can be divided into the following three categories:

USB Device Driver Framework The API offers functions to register and unregister USB device drivers. When a new USB device is attached to the bus, the `usbcore` configures this device. This includes assigning this device an address and reading its descriptors. Finally, the `usbcore` searches for an appropriate USB device driver and if found, call its `probe()` function.

Memory Management There exist functions for allocating and freeing URBS and transfer buffers. Transfer buffers that were allocated with the function provided by the `usbcore` are ready for use by the host controller in DMA transfers. However, not all USB device drivers use the `usbcore`'s function for allocating transfer buffers (see Section 3.2.5 on page 32).

Convenience Wrappers There also exist some convenience wrappers for performing control and bulk transfers (including scatter-gather transfer) which are synchronous, unlike the normal call for submitting an URB (`submit_urb()`).

2.3.3 Linux USB Host Controller API

HC drivers are Linux device drivers, which register themselves at the `usbcore`. They provide several callback functions, which are called by the `usbcore` to perform actions on the HC. These callback functions include —amongst others— functions for submitting and unlinking a URB, to get the number of the current USB frame, and functions to poll the current status of the HC and its root hub.

Listing 2.1: The URB data structure

```

struct urb
{
    /* private: usb core and host controller
       only fields in the urb */
    struct kref kref; /* reference count
                       * of the URB */
    spinlock_t lock; /* lock for the URB */
    void *hcpriv; /* private data for
                  * host controller */
    int bandwidth; /* bandwidth for
                   * INT/ISO request */
    atomic_t use_count; /* concurrent submissions
                        * counter */
    u8 reject; /* submissions will fail */
    /* public: documented fields in the urb that
       can be used by drivers */
    struct list_head urb_list; /* list head for use
                               * by the urb's
                               * current owner */
    struct usb_device *dev; /* (in) pointer to
                             * associated device */
    unsigned int pipe; /* (in) pipe information */
    int status; /* (return) non-ISO status */
    unsigned int transfer_flags; /* (in) URB_SHORT_NOT_OK | ... */
    void *transfer_buffer; /* (in) associated data buffer */
    dma_addr_t transfer_dma; /* (in) dma addr for
                              * transfer_buffer */
    int transfer_buffer_length; /* (in) data buffer length */
    int actual_length; /* (return) actual
                       * transfer length */
    unsigned char *setup_packet; /* (in) setup packet
                                  * (control only) */
    dma_addr_t setup_dma; /* (in) dma addr for
                           * setup_packet */
    int start_frame; /* (modify) start frame (ISO) */
    int number_of_packets; /* (in) number of ISO packets */
    int interval; /* (modify) transfer interval
                  * (INT/ISO) */
    int error_count; /* (return) number of ISO
                     * errors */
    void *context; /* (in) context for
                   * completion */
    usb_complete_t complete; /* (in) completion routine */
    struct usb_iso_packet_descriptor iso_frame_desc[0];
    /* (in) ISO ONLY */
};

```

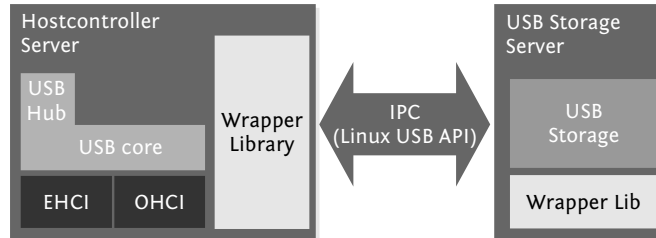


Figure 2.7: The components of Gerd Griessbach's work *USB for DROPS*.

The HC driver is also responsible to return a completed URB back to the usbcore, which then will execute the URB's completion handler.

2.4 Related Work

In this section I present two works related to my thesis' topic. The first one is *USB for DROPS*, which also provides a USB infrastructure for DROPS, but has some shortcomings, which I explain in this section. The other one is *USB/IP*, which inspired the solution I present in the next chapter.

2.4.1 USB for DROPS

In 2003, Gerd Griessbach implemented a first USB infrastructure for DROPS, using the USB stack of Linux 2.4.20 [Gri03]. This work introduces an L4 server including the Linux usbcore and HC drivers. USB device drivers communicate with the HC server using an interface mapping the Linux USB API to L4 IPC (Figure 2.7).

From today's perspective, this work appears to have the following four shortcomings:

Outdated USB stack The first problem is the outdated USB stack. Gerd Griessbach used the USB stack of Linux 2.4.20 as base for his work. At the point of this writing, the current Linux Version is 2.6.26 and Linux' USB stack has matured over time. Bugs were fixed, and features such as high speed isochronous transfer were added. Furthermore, many USB device drivers were written for Linux 2.6 and were not back-ported to Linux 2.4.

Thus, there is a need to move the DROPS USB infrastructure to a more recent USB stack.

Outdated DDE As the work is based on the Linux 2.4 kernel, `dde_linux24` has been used to port the Linux USB stack to L4. However, as shown by Thomas Friebel [Fri06], the `ddekit` approach seems to be more promising for porting device drivers from legacy operating systems to L4, because it leads to a cleaner architecture and to less changes of legacy source code.

Complexity Choosing the Linux USB API as IPC interface leads to another problem. This API consists of over 20 calls, which all have to be implemented as IPC functions, both on client and on server side. In addition, the Linux USB API is reentrant, implicating that the server side has to be multi-threaded.

Because the data structures of the Linux USB API have to be accessible on server and client side, USB for DROPS uses copies of these data structures on both sides. Because this data structures do not necessarily have to be binary compatible, they have to be serialized. This leads to an extremely complex architecture, which is a problem for the maintainability of this project.

Platform dependency Using the Linux USB API as IPC interface, also prevents using non-Linux USB device drivers. To forgo this problem, a more generic interface is desirable.

As I will show in the remainder of my thesis, it is possible to provide a less complex, generic, portable, and maintainable solution for a USB infrastructure for DROPS.

2.4.2 USB/IP

The USB/IP project [HKFS05] aims at providing a network transparent USB-device-sharing system. USB messages are encapsulated into IP packets and then transmitted between computers.

Therefore, USB/IP implements a virtual-host controller interface (VHCI) driver on the client side and a stub USB device driver on the server side, as shown in Figure 2.8. The VHCI driver acts like a normal USB host controller, so it is possible to use unmodified USB device drivers on the client side.

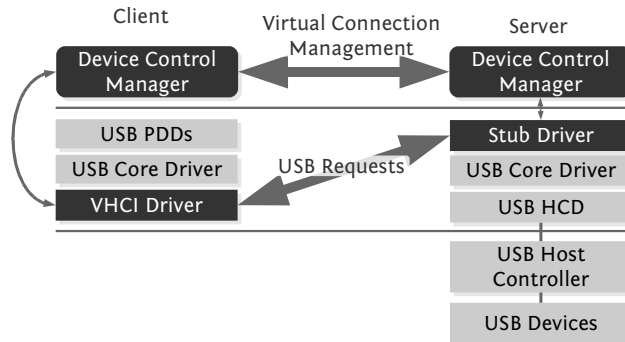


Figure 2.8: Architecture of the USB/IP project. The darker parts belong to USB/IP.

When a USB device is forwarded to the client side, the VHCI driver informs the core driver of a root-hub-port status change, so that the core driver can configure the new USB device. All generated USB requests are encapsulated in IP packets and sent to the USB stub driver on the server side, where they are handled by the USB core driver, which submits them to the physical host controller. The responses are then sent back to the VHCI driver, which informs its USB core driver about the completion of the USB request.

Currently there exists a VHCI driver and stub USB device drivers for Linux, but there are plans to port them to other operating systems including Microsoft Windows.

3 Design

As explained in Chapter 2, the existing implementation of DROPS' USB infrastructure needs to be replaced by a new one. This replacement should offer USB functionality to L4 applications in a generic way. This way it will be easy to integrate this solution into existing projects, like L⁴Linux, a Linux port to L4, or DDE. Further, it should be possible for different L4 applications to access USB devices concurrently, which suggests a client-server separation.

In this chapter, I introduce DDEUSB, a flexible and maintainable replacement for USB for DROPS, which fulfills the aforementioned requirements.

In the first part, I outline the design goals of DDEUSB. Then, after presenting the DDEUSB server, and its IPC interface, I introduce an extension to `dde_linux26` that uses this server to provide USB functionality to legacy Linux USB device drivers.

As the client-server separation introduces IPC overhead, in the final part of this chapter I discuss communication methods that are suitable to keep this overhead as low as possible.

3.1 Design Goals

In order to create a flexible USB framework, the following design goals have to be taken into account.

Modularity and flexibility In conjunction with the microkernel approach, modularity is a major design goal. The user should be able to replace and reconfigure all parts of the running system.

Flexibility also plays an important role. Flexibility here means that the interface used to access the DDEUSB server should be generic and not specialized to a single client, like it was the case in USB for DROPS.

Maintainability The result of this work should be easy to maintain. Because DDEUSB will make use of a legacy USB stack, updates to newer versions of this USB stack should require low effort.

Security DDEUSB aims to be secure, meaning that no part of this solution should affect other parts of the system with defective behavior. Because of the complexity of these topics, other security aspects like confidentiality and reliability are not covered in this work.

Performance The use of DDE, and the introduction of a client–server separation will introduce overhead. One goal of DDEUSB is to keep this overhead as small as possible. This requires efficient communication mechanisms and the possibility to forgo the client–server separation for the benefit of high performance applications.

3.2 DDEUSB - A USB Framework for L4Env

This section describes the design of DDEUSB. First, I explain, why the Linux USBstack was chosen as base for this work. Afterward, I show the DDEUSB's IPC interface and finally I introduce the VHCD approach, which allows existing projects, like DDE to act as client of DDEUSB.

3.2.1 The DDEUSB Server

To achieve security and modularity, and to provide concurrent access to USB, the separation of USB components is essential. Because of that, DDEUSB is divided into a server, multiplexing access to USB, and clients of this server, forming the actual USB device drivers.

USB Stack for the DDEUSB Server

To provide USB functionality to other L4 applications, it is necessary to allow the DDEUSB server to access USB. For that reason, a USB stack needs to be included into the DDEUSB server.

Due to the complexity of the USB architecture, a complete rewrite of a USB stack is not feasible. So it stands to reason, to reuse an existing USB stack.

The USB stack used in the DDEUSB server has to match three criteria. It should be well documented, well tested, and easy to port. Portability includes

the requirement of freely available source code. Further, there should exist a DDE backend for the USB stack's operating system, in order to minimize porting effort.

At the time of this writing, there are DDE backends available for Linux and FreeBSD. Thus, the USB stacks of these two operating systems are shortlisted. Both of them fulfill the aforementioned requirements.

I chose the Linux USB stack as base of the DDEUSB server for two reasons. First, when this work started the FreeBSD backend of the DDE was not fully functional, even though these issues were solved during the time of this writing. Second, I am more familiar with the Linux operating system. So, with choosing the Linux USBstack, time could be saved that would have been necessary to dive into the FreeBSD kernel.

DDEUSB Server IPC Interface

In this section, I describe the interface the DDEUSB server offers to clients. As mentioned before, this interface has to be generic and provide efficient communication between the server and its clients.

USB for DROPS used the Linux USB API for that purpose, but as pointed out in Chapter 2 this leads to a complex design. Thus, to get a simple and generic solution, DDEUSB has to use a thinner interface.

Examining Linux' USB stack, I found that all communication between USB device drivers and the usbcore is performed using four functions:

- `usb_register()`,
- `usb_unregister()`,
- `usb_submit_urb()`, and
- `usb_unlink_urb()`.

The first two functions are used to register and unregister a driver at the usbcore, whereas `usb_unlink_urb()` is used to cancel the submission of a URB. All other functions of the usbcore API that generate traffic on the bus, like `usb_bulk_msg()` or `usb_control_msg()`, depend on `usb_submit_urb()`.

Building an IPC interface from these four functions leads to a small and generic interface. In spite of its small size, it offers all required functionality to build USB device drivers on top of it.

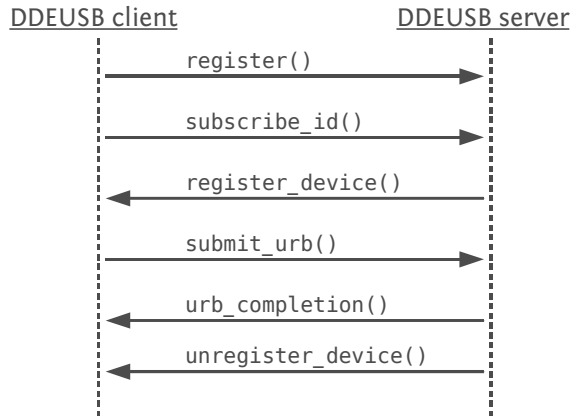


Figure 3.1: Illustration of a DDEUSB example session.

Concluding from these observations, I designed the server's interface to provide the following operations:

int register() This call is the first one to be issued by a client. The server will check if the calling thread, is allowed to act as a client (see Access Control, Section 3.2.3 on page 31) and then setup the client data structure in its client database.

int subscribe_id(ddeusb_usb_device_id id) With this call, a registered client subscribes for a specific USB device ID. The server will check, if the client is allowed to subscribe for that specific ID and then add this device ID to the client's entry in its client database.

void unsubscribe_id(ddeusb_usb_device_id id) This cancels a subscription for a USB device ID. If a USB device was forwarded to the client because of this subscription this device will be revoked.

int submit_urb(ddeusb_urb *d_urb, ...) This call allows a registered client to submit a URB. The server will do several sanity checks before it submits this URB to his usbcore.

For example, it will check if the USB device the URB refers to is valid and owned by the client. Additionally, control messages that the client is not

allowed to send, have to be trapped (see Section 3.2.1 on the following page).

int unregister() With this call a client unregisters from the server. The server will revoke all devices from this client and remove the client from the its client database.

Additionally, the client has to implement a notification thread. The purpose of this notification server is to inform the client about forwarded or revoked devices and URB completions. All these calls are one-way calls, so the server does not have to wait for the client to handle the notification. Thus, if a client is not fast enough to handle all notifications, the function of the server will not be affected. However the client has to be aware that this might happen and has to deal with it, for example, using time-outs and resubmits. The notification calls that have to be implemented on the client side include:

void register_device(int dev_id, ...) This notification is sent, when a device is forwarded to a client by the DDEUSB server. It includes a device ID, which, together with the client's thread ID, identifies the device on the server side.

void unregister_device(int dev_id) This notification informs the client about the revocation of a USB device. After this notification, the client is not allowed to send URBS to the corresponding device anymore.

void urb_completion(int urb_id, ...) This call informs the client about the completion of a specific URB. Its parameters include a `urb_id`, used by the client for identifying the URB. This ID has to be set by the client in the `submit_urb()` call. Further, the calls parameter include the return values of the URB and, if necessary, transfer buffers (for ingoing transfers).

In Figure 3.1, we see the sequence diagram of the communication between one client and the DDEUSB server during an example session. First, the client registers itself at the server and afterward, it subscribes for a USB device ID. Then, the server is forwarding a device to the client using the `register_device()` call, provided by the client's notification server. After that, the client is sending a URB to the server, using `submit_urb()`. When this URB is completed, the server sends a `urb_completion()` notification to the

client. Finally, the server revokes the device from the client. The reason for that could be, for example, that the device was physically disconnected from the USB.

DDEUSB Server Architecture

After describing the DDEUSB server's interface, I now explain its internal architecture.

The DDEUSB server consists of a complete Linux USB stack with HC drivers, and a usbcore based on `dde_linux26`. The usbcore takes care of configuration and enumeration of USB devices. On top of the usbcore there is a stub driver that implements the actual server functionality.

The stub driver is forgoing the driver matching algorithm of the usbcore, by leaving those flags of the USB device ID empty that indicate which fields of the USB device ID should be matched against. Thereby, the stub driver gets access to every newly attached USB device. However, USB hubs are still handled by usbcore's hub driver.

As mentioned before, the stub driver acts as a server for DDEUSB clients (i.e., L4 USB device drivers). It forwards URBS received from a client to the usbcore and sends completions back to the client.

Whenever a new device is attached to the bus, the stub driver will search for a suitable client in its client database and if found, forward the device to this client.

DDEUSB's server architecture is shown in Figure 3.2. On the left side we can see the DDEUSB server including the Linux USB stack on top of `dde_linux26`, and the stub device driver. On the right side we can see a client using the VHCD approach to communicate with the server. The VHCD approach is introduced in the next section.

Trapping Control Messages

As mentioned before, a USB interface may contain several alternative settings. To switch between these settings the USB device driver sends a special control message to the device. Because an alternative setting may contain different endpoints the internal representation of the device may change.

In order to keep the internal representation of USB devices on server and client side in sync, a DDEUSB client is not allowed to send these control

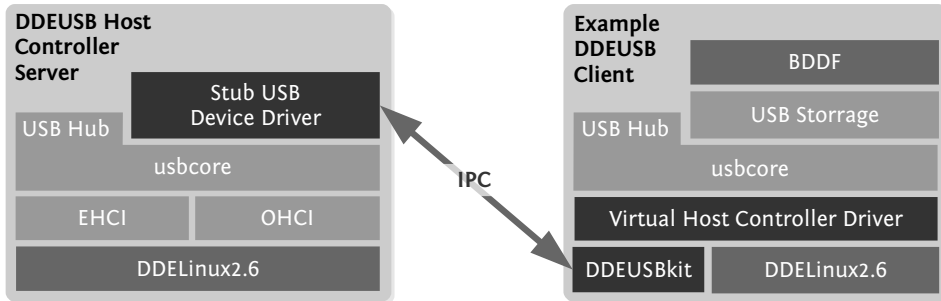


Figure 3.2: Illustration DDEUSB's architecture. The darker parts are the DDEUSB components.

messages directly. Instead they are trapped by the DDEUSB server, which then executes the corresponding `usb_set_interface()` call of its `usbcore`.

Trapping of control messages, can also be used to modify the client's view of a USB device (see device forwarding, Section 3.2.3 on page 31).

3.2.2 DDEUSB Client Side

In this part of the work, I present the DDEUSB client side. First, I introduce DDEUSBkit, a small library, abstracting the communication between clients and the server. Afterward, I present the VHCD approach for integrating DDEUSB into existing projects. This integration has been done for `dde_linux26` and L⁴Linux.

DDEUSBkit

To abstract communication between the DDEUSB server and its clients, an IPC wrapper library, called DDEUSBkit, was created. This library also implements the client-side notification server, described in Section 3.2.1 on page 27.

Thus, DDEUSBkit provides the minimally required client-side functionality to access USB devices. In this regard, it can be compared to the `ddekit`, which offers minimal functionality to write device drivers. Hence, the DDEUSBkit together with the `ddekit` can be used to write legacy-free USB device drivers.

DDEUSB DDE Integration

On the right side of Figure 3.2, we can see a `dde_linux26`-based device driver acting as a client of the DDEUSB server. This client uses the `VHCD` approach to access DDEUSB's functionality.

As the figure shows, the solution consists of a normal Linux USB device driver paired with a `dde_linux26`-based Linux USB stack. The only difference to a normal USB stack is that it does not make use of Linux' legacy HC drivers. Instead, it uses a virtual host controller driver (`VHCD`), which mimics the behavior of a normal HC driver. However, instead of splitting up URBS into TDs, which are then linked to the transmission queue of a physical HC, the `VHCD` forwards the URBS to the server-side's stub driver. When a URBS is completed the DDEUSB server sends a notification to the `VHCD`, which then returns the URBS to the client side's `usbcore`.

The `VHCD` also emulates a root hub and when a USB device is forwarded to the client, it mimics a root hub port status change, causing the `usbcore` to configure the new device.

This mechanism allows seamless integration of DDEUSB in existing DDE backends. Furthermore, because the virtual host controller driver is a normal Linux driver, integration in existing projects like L⁴Linux [Hoh96, HHW98], a Linux port to L4, requires low effort.

High-Performance Applications

There exist applications, where high performance is essential. This is, for example, the case for audio recording applications. In this case the throughput to a USB hard disk needs to be as high as possible, so that the number of simultaneously recorded audio tracks can be maximized.

Christian Helmuth [Hel01] proposes colocation to enhance performance in such scenarios. For that, the `usbcore`, the USB device driver, and the host controller drivers have to be linked together to a single binary.

However, this way a new problem occurs, namely that low-performance applications, which are not colocated with the high-performance components, do not have access to USB. Nevertheless, this problem can be solved by integrating DDEUSB's stub driver into the high-performance application as well

3.2.3 Access Control

When a DDEUSB client registers at the server for a specific USB device type, the server somehow needs to decide if this certain client is authorized to access that device. To achieve this, a mechanism similar to the ioguard approach introduced by Lukas Hänel [Hän07] could be used, introducing a new system resource consisting of a USB device id and the address of a USB device on a bus. The integration of DDEUSB into ioguard is straight forward and thus is not covered in this work.

3.2.4 Forwarding USB devices

As mentioned in Section 2.1.2 on page 10, USB devices present themselves to software with the help of descriptors. When a new device is attached to the bus, the usbcore reads the descriptors of this device and generates the data structures representing the device. Then, the usbcore searches for a driver for this device. The entity that the drivers receive from the usbcore is a USB interface.

There are two possible ways to forward devices to DDEUSB clients. The first one is to forward only USB interfaces to the client. The second one is to forward the whole USB device. Both approaches are discussed in the following.

Forwarding on Interface Layer

Using this approach, the DDEUSB client only gets the interfaces for which it includes drivers. To achieve this, the client at first has to be informed about a new device. Then, when the usbcore tries to get the descriptors of the new device, the DDEUSB server has to trap these `get_descriptor` USB control messages and modify the descriptors in such a way that they only include the interfaces the client is allowed to bind to.

Forwarding on Device Layer

When forwarding whole devices to a client, the client gets access to all interfaces provided by this device. The major drawback is that this client has to include drivers for all the interfaces and gets access to all the USB device's functionalities. For example, a webcam can also include a microphone. This device then provides two interfaces representing these two functionalities. A

DDEUSB client driving this device has to include the webcam driver and the audio driver.

The advantage of this method is its simplicity. The client side just has to be informed about the new device, the rest will be done by the client's usbcore. Thus, there is no need to trap the clients `get_descriptor` messages on server side.

Although forwarding on interface layer is desirable, because of its simplicity, for now, I implemented forwarding on device layer.

3.2.5 Communication Mechanisms

Efficient communication between the server and the clients is a must to keep DDEUSB's overhead as low as possible. The largest part of the ongoing communication consists of `submit_urb()` calls. Furthermore, this call may include a large amount data: the URB's transfer buffer. Thus, it is necessary to find a good way to transmit these transfer buffers from the client to the server (outgoing transfer), and back (ingoing transfer). Four imaginable methods are illustrated in Figure 3.3 on the facing page and discussed in the following.

Copying of Transfer Buffers

The simplest way to transfer data is to copy the transfer buffer of an URB using string IPC upon submission.

However, that way the transfer buffers have to be copied twice. The first copy is done by the kernel from the senders message buffer to the receivers message buffer. After that the receiver has to copy it out of the message buffer into his own memory, because the buffer is only valid during the IPC. However, DDEUSB cannot know in advance how long the transfer buffer will be needed.

Although this solution is the slowest, it is easy to implement and secure for both the DDEUSB server and its clients, because no memory has to be shared and no physical addresses are submitted.

Transfer Buffers in Shared Memory

In this alternative, memory used for transfer buffers is taken out of a memory pool shared between client and server. Instead of copying the whole buffer, it just is necessary to submit the position of the buffer in the shared memory location. This saves time and leads to a zero copy protocol.

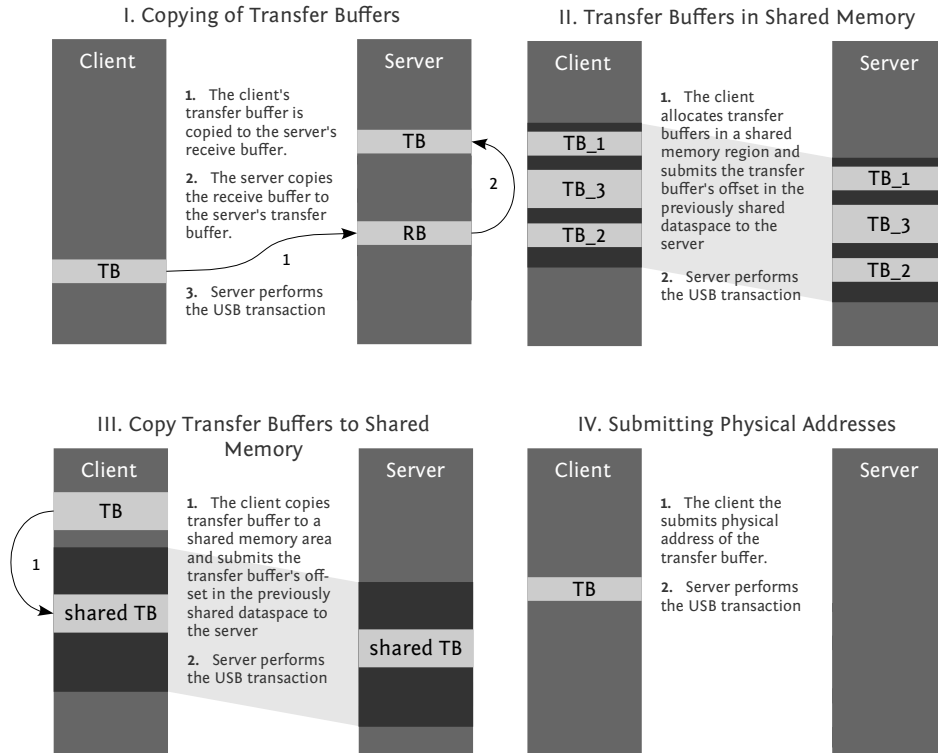


Figure 3.3: Illustration of the communication methods discussed in Section 3.2.5.

Although, this variant would be the first choice for legacy free USB drivers, built with the ddekit and DDEUSBkit, it is not suitable for ported legacy Linux USB device drivers. The reason for this is that Linux USB device drivers do not necessarily allocate their transfer buffers from a special memory pool.

Indeed, the usbcore offers a function to allocate USB transfer buffers, but many Linux USB device drivers rely on other frameworks for memory allocation.

For example, the Linux USB cPIA webcam driver is built upon the cPIA module, which abstracts functionalities of the cPIA chipset. This module uses `kmalloc` for allocating the video buffers when the video device is opened.

These buffers are then filled by the USB `cpia` driver. Another example is Linux' USB storage driver. This driver gets the transfer buffers from the Linux storage subsystem, so it also does not use `usbcore`'s buffer allocation function.

Thus, it is a hard task, to adapt legacy Linux USB device drivers to use this mechanism. The alternative of taking all memory allocated with `kmalloc()` out of the shared memory pool is impracticable, because the whole address base of the Linux driver would be accessible by the server, which would be a security drawback.

It is also possible to temporarily share the memory region of the current URB transfer buffer for the time between submit and completion. However, it is likely that, without caching of these shared regions, the mapping overhead would be too big to gain performance with this method. Although, this method may be applicable for exceptionally large buffers, for which the copying overhead would be higher than the mapping overhead. In Chapter 5, I show that mapping buffers larger than 4 kB improves the performance compared to copying into shared memory (see below).

For reasons mentioned above, this way using shared memory is not an alternative to copying the transfer buffers for legacy Linux USB device drivers, although it would be an alternative for non-legacy USB device drivers built upon the `DDEUSBkit`.

Copy Transfer Buffers to Shared Memory

Another alternative is to copy the buffers to a DMA-capable memory region shared between client and server.

When a client submits an outgoing URB, the client first needs to copy the transfer buffer into this shared dataspace. After that, he submits the position of the transfer buffer's offset in this dataspace to the server. Now, the server can directly use this transfer buffer to perform the request.

For a URB that receives data from a USB device, the client also has to submit an offset to a position in that data space. The server will then perform the request using that memory location as transfer buffer. When the request is completed, the client has to copy the transfer buffer out of the shared memory area to the URB's actual transfer buffer.

That way, the number of involved copy operations can be reduced to one, but an additional IPC for setting up the shared data space is necessary, which is no problem, because this only has to be done once the driver is loaded.

The DROPS Streaming Interface (DSI) [LRH01] offers a similar communication method to L4 applications. However, because DSI is mainly used for streaming applications, and offers much more than the required functionality (like QoS, multiple receivers, ...), it has a complex interface. In order to keep the complexity of DDEUSB as small as possible, an own implementation of this transfer method seems to be better suited.

Submitting Physical Addresses

Most USB host controllers use DMA to access the transfer buffers, so it is possible for a client to just transfer the physical address of the transfer buffer to the server. Although this is fast, this method yields a security drawback, because the server has no chance of knowing whether that piece of physical memory is really mapped by the client and therefore needs to trust the client.

Conclusion

From the alternatives presented in this section, simple copying is indeed the slowest method but easy to implement. Transmitting physical addresses should only be used when both the DDEUSB server and client are totally trustworthy and performance has the highest priority. Copying transfer buffers into a shared data space seems to be a good alternative to simple copying, but introduces a security drawback, because the owner of the shared dataspace, can revoke right to access this dataspace at any time.

4 Implementation

In this chapter, I first show the steps necessary to port the Linux USB stack to `dde_linux26`. Afterward, I explain some details of the `DDEUSB` implementation. Thereafter, I introduce the example applications that have been created during this work. Finally, I explain a problem considering isochronous transfer.

4.1 Porting the Linux USB Stack

As first part of this work, the Linux USB stack had to be ported to `dde_linux26`. Therefore, some functionality had to be added to `dde_linux26` and the `ddekit`. This functionality includes support for platform devices, some timer functions, parts of the Linux parser library and functions for firmware loading.

Most of the functionality could be added by simply merging the corresponding Linux source files, thereby incorporating the needed functionality to `dde_linux26`.

4.1.1 Platform Devices

Linux 2.6 introduced a unified device model. In this driver model every device is bound to a specific bus and, thereby, a device hierarchy is created. However, some legacy devices are not physically bound to a bus. To maintain the bus-device topology for those devices, Linux introduced a virtual bus, the platform bus.

Although the platform bus was designed to serve as a parent for legacy devices, it is also suitable as parent for virtual devices, like the virtual host controller of the virtual host controller driver (`VHCD`). The `VHCD` registers itself as a platform device driver and generates a virtual platform device in its initialization function, representing the `VHCD`.

4.1.2 DMA from the Stack

There are USB device drivers doing DMA from their stack. Mainly, that is the case for setup packages of USB control messages. That is no problem when acting as DDEUSB client because the setup packages are copied. However, when colocating USB device drivers and HC drivers, the HC driver will do DMA also for the setup packages.

The ddekit creates threads by calling a thread creation function of the L4Env thread library, which is also responsible for creating a stack for the newly created thread. There is no entry for the stack's memory location in ddekit's page table and, therefore, no address translation is possible using `virt_to_phys()`.

I solved this problem by modifying the thread creation mechanism of ddekit in a way, so that newly created threads get their stacks created directly from the ddekit.

4.1.3 Physical to Virtual Address Translation

The Linux SCSI subsystem used by `usbstorage` needs support for physical-to-virtual-addresses translation, which was not supported by ddekit. This feature was also added to the ddekit by adding a list to ddekit's pagetable, which contains the physical starting addresses of the mapped regions.

4.2 The DDEUSB Implementation

In this section, I explain some details of the DDEUSB server implementation.

4.2.1 Caching of URB Data Structures

Linux device drivers normally allocate URB data structures once and reuse them by retransmitting them in the completion handler. Because of the client-server separation, this type of reuse is impossible in the DDEUSB server's stub driver. Thus, the DDEUSB server has to allocate a new URB data structure every time a client submits a URB.

The `usbcore` provides the function `usb_alloc_urb(int nof)` for URB allocation. This function takes the number of isochronous frames as argument. For memory allocation itself, it uses `kmalloc()`.

Because URB submissions occur frequently, a server side caching mechanism for the URB data structure is desirable.

However, because the size of the URB data structures differs according to the number of isochronous frames, a normal Linux look-aside cache is not applicable. So, I decided to implement a dedicated URB caching mechanism.

For this mechanism, the DDEUSB server implements its own URB allocation and deallocation functions. Whenever a free URB is needed, the server calls its URB allocator, which tries to lookup a URB in a free-list. If there is no free URB in this list, then a URB has to be allocated using `usb_urb_alloc()`. When the URB is not needed anymore, the DDEUSB server frees the URB, using its own deallocator, adding this URB to the free-list.

A special case is looking up URBS that are used for isochronous transfers, because the URB data structure, which also contains the frame descriptors, needs to be larger. Therefore, an extra list containing only URBS used for isochronous transfer was introduced. The URB allocator then iterates over that list looking for a URB data structure large enough for the current transfer.

In my experiments, it turned out that for an isochronous transfer there have to be 15 URBS cached until no new URB's need to be allocated by the stub driver.

4.2.2 Improving DDEUSB's IPC Performance

During first test runs, DDEUSB showed a dissatisfying performance. My measurements—the exact test setup is explained in Chapter 5—showed that `usbstorage` under DDEUSB only reached about 70 percent (15.8 KB/s) of the throughput of native Linux (22.7 KB/s). I suspected that the reason for this deficiency can be a found in large IPC overhead and actually it turned out that improving DDEUSB's IPC performance drastically improved the overall performance.

Short Versus Long IPC

The Fiasco microkernel provides two different types of IPC, namely short IPC and long IPC. Using short IPC, two machine words can be transmitted using machine registers, which leads to good performance. The other way, using string IPC, involves copying and is thus much slower, but more data can be transferred.

The `submit_urb()` and the `urb_complete()` calls cause the most of the ongoing IPC during a DDEUSB session. Hence, it is desirable that these two calls use short IPC. However, two machine words do not suffice for transmitting the large number of arguments. Therefore, I investigated the following mechanisms

Shared Arguments Stacks

Bershad and colleagues [BALL90] introduced *lightweight remote procedure calls* (LRPC) to improve the performance of remote procedure calls (RPC). LRPC uses four techniques:

1. LRPC uses a *simple control transfer mechanism*. The requested procedure is executed by the client's thread in the server's domain.
2. LRPC uses a *simple data transfer mechanism*. Parameters are passed to procedures using a shared argument stack.
3. Because of LRPC's simple control and data transfer model, it is possible to generate *highly optimized stubs*.
4. LRPC is *designed for concurrence*. It avoids shared data structure bottlenecks by minimizing the use of shared data structures on the critical domain transfer path, allowing better performance on multi processor platforms. Further, LRPC reduces the context-switch overhead by caching domain contexts on idle processors.

Because DDEUSB relies on the L4 IPC mechanism, the method of control transfer cannot be influenced. Further, design for concurrence will not be regarded in this work. However, simple data transfer seems to be a good starting point, to optimize DDEUSB's IPC performance.

LRPC is using argument stacks (A-stacks) to transfer the parameters of a remote procedure call. These A-stacks are shared read-writable between caller and callee. Whenever a caller makes an RPC to the callee, the arguments of the call are copied to the shared A-stack. That way redundant data copying is avoided.

Adopting this mechanism to DDEUSB, I created a dataspace, which is shared between client and server. This dataspace, in its current implementation, contains 50 argument containers. One of these argument container provides enough space to contain the arguments of one `submit_urb()`

call. Further, because it is likely that after a `submit_urb()`, exactly one `urb_complete()` call will follow, the same container is also used for the arguments of `urb_complete()`.

This way, during a `submit_urb()` and during a `urb_complete()` call, only the offset to the corresponding container entry in the shared dataspace has to be transmitted. This has two advantages. First, as with LRPC, redundant data copying is avoided. The second, and more important advantage is, that the offset to the container is small enough to fit in a short IPC. That way, the DICE IDL compiler is able to generate stub functions using short IPC for communication.

To use this mechanism the IPC interface has to be extended in the following way: On the server side, two new calls are needed. The first call is required to setup the shared dataspace. Another IPC call that has to be implemented, is a call, corresponding to the `submit_urb()` call, using this technique, which takes the index to the corresponding argument container as argument. This function is called `fast_submit_urb`. On the client side a new `fast_urb_complete()` notification call has to be implemented.

For now, there are 50 of these argument containers in one shared dataspace, so if the number of concurrently submitted URBS exceeds this number, the normal `submit_urb()` call has to be used as fall-back method, which uses string IPC. In my test cases 50 containers were always sufficient, so the fall-back method will be used rarely. The decision when to use the fall-back method can be made transparently to the client in the DDEUSBkit library.

Transfer Buffer Transmission Mechanisms

As stated in the design chapter (see Section 3.2.5 on page 32), there are several ways to transfer the URBS' transfer buffers from the client to the server, and vice versa. The current implementation of DDEUSB is able to perform transactions using all four methods:

1. Simple copying of transfer buffers,
2. Temporarily mapping of transfer buffers,
3. Copying of transfer buffers into a shared dataspace, and
4. Transmission of the physical addresses of the transfer buffers.

The normal `submit_urb()` and `urb_complete()` calls use string IPC to transmit the transfer buffers from the caller to the callee. These calls are used as fall-back methods, when their pendants that use short IPC are not applicable.

Using the `fast_submit_urb()`, and `fast_urb_complete()` calls, the method used for buffer transfer can be selected by setting an argument in the argument container. These methods include transmitting physical addresses, temporally mapping the transfer buffers, and copying them into a buffer in the container. The last mentioned method is only applicable for small buffer sizes, due to the limited size of the container.

The method for transferring the buffer is selected by the client but the server easily can be extended in a way that it can deny the use of certain methods. So, if a client tries to submit a URB using a prohibited method, the `fast_submit_urb()` call fails.

4.3 DDEUSB Example Applications

As part of this work, I implemented two example applications for DDEUSB. These include USBHID, a human interface device driver, and USBCAM, a webcam driver including graphical output using the Desktop Operating Environment (DOPE) [FH03, Fes02], a window server, which is part of DROPS.

4.3.1 USBHID

USBHID (Fig. 4.1) is a port of the Linux USB human-interface device driver to L4. All parts of the driver considering USB have been reused from Linux. The driver reads input events from USB devices such as keyboards and mice and forwards them to the L4Env input library L4Input.

USB HID Class

The USB HID class [USB01] is a USB device class aggregating human-interface devices such as keyboards and mice. Device drivers for this class are provided by every modern operating system, including Linux.

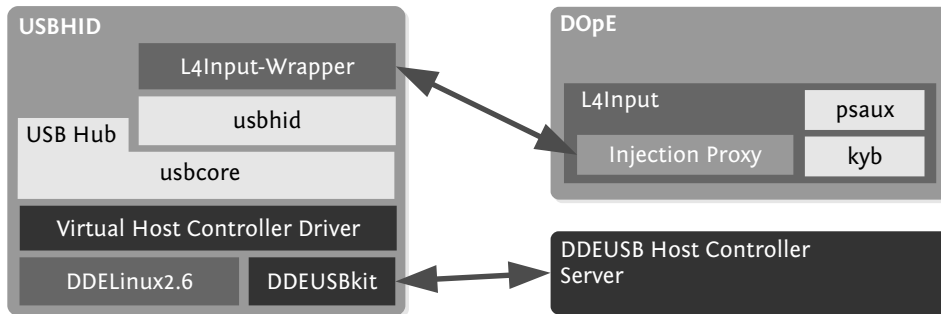


Figure 4.1: Architecture of USBHID.

The Linux Input Subsystem

The Linux input subsystem manages a large range of input devices, which are connected through different kinds of interfaces (PS/2, USB, ...), offering a consistent, and device-independent interface to userspace. The main elements of the input subsystem are the input core, drivers, and handlers.

The event handlers offer interfaces to common input device classes like keyboards, mice, and joysticks to userspace, whereas the input drivers are responsible to interact with the input devices at low-level, that is, on USB, PCI memory regions, and input-output-ports. Communication between these components is done using input events, sent by the input drivers to the input core, which then distributes them amongst the different event handlers.

L4Input

The L4Input package provides an API for input device drivers, used for example by DOpE and con. It uses the Linux 2.6 input subsystem including input device drivers, amongst others for PS/2 input devices. It also provides an input event injection proxy, enabling L4Input to receive input from other tasks. This proxy acts as input device driver and forwards received events to L4Input's core.

USBHID

Like L4Input, USBHID is using a complete Linux input subsystem with an unmodified input core and the usbhid device drivers but makes no use of the

legacy event handlers. Instead it implements an own event handler, which forwards the events to L4Input's injection proxy.

Using `dde_linux26` and `DDEUSB` all Linux parts have been reused. Only a new event handler had to be created, which is a modified version of the Linux `evdev` driver. In most parts, this event handler, was already existing as part of the `usbhid` package, which also provides support for USB HIDS on L4.

4.3.2 USBCAM

USBCAM is a webcam viewer application for DOPE, which includes a USB device driver for webcams based on the color processor interface ASIC (CPIA) [VLS97].

Video4Linux

Video4Linux, or short `v4L`, is a Linux kernel API for analog video capture and output drivers, which offers an abstraction of video hardware. Thus, the application developer does not need to know about the hardware used. It was introduced in the late 2.1.x development cycle of the Linux kernel. Recently it has been replaced by its successor `Video4Linux2`, or `v4L2`, which also provides a compatibility layer for applications that still use `v4L`. Communication with the video hardware works through the `write()/read()` Linux system calls on a Linux device file or, alternatively, through shared memory.

Architecture of the Linux CPIA Driver

The CPIA capture driver consists of two parts. First, there is a library, to be called `CPIA-core` here. The `CPIA-core` controls common functions of the CPIA chipset and manages device registration and deregistration. It uses the older `v4L` API.

Second, there exist two low-level drivers, encapsulating data transmission to and from the CPIA chipset, one for USB and one for the parallel port.

USBCAM

As we can see in Figure 4.2, USBCAM consists of a small DOPE application colated with a `v4L` subsystem, the `CPIA-core`, and the CPIA USB device driver. I was able to reuse the unmodified source code. I just had to create a few wrapper functions to memory functions used by the `CPIA-core`. For example the

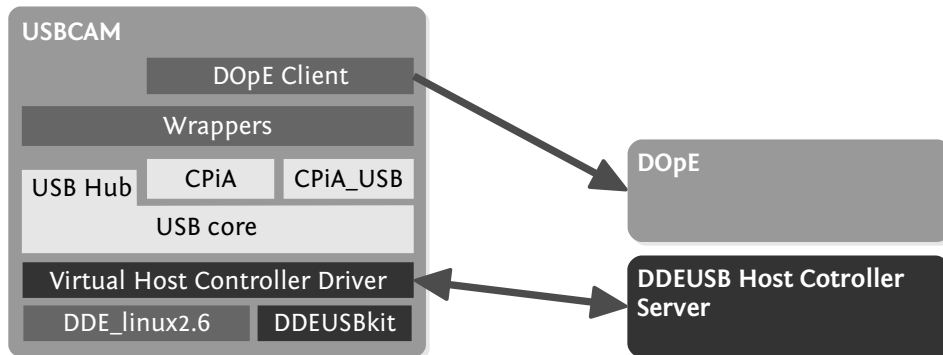


Figure 4.2: Architecture of USBCAM. The lighter parts are legacy Linux parts.

CPiA-core makes use of `vmalloc()`, which is not implemented in `dde_linux26`. Therefore, a wrapper function has been written mapping calls to this function to `kmalloc()`. I created a device-file wrapper, offering file operations to the viewer application.

The viewer application is requesting a framebuffer, called `Vscreen`, from `DOpE`, to which it writes the data read from the video device.

4.4 L⁴Linux as DDEUSB Client

To use USB functionality in L⁴Linux, the current version of L⁴Linux needs access to the whole PCI bus. This allows use of the legacy HC drivers, but impairs security as the system resources can no longer be managed by L4io. Furthermore sharing of USB bandwidth amongst L⁴Linux and other L4 tasks is impossible.

As shown previously (see Section 3.2.2 on page 30), the VHCD approach can be used to integrate DDEUSB into existing projects. Thus, I created a VHCD for L⁴Linux. Thereby I was able to reuse most of `dde_linux26` VHCD's source code. However, the following changes to the notification mechanism of the VHCD were necessary.

In the `dde_linux26` version of the VHCD, the notification server loop was started by the main function of the driver server. Thereby a `dde_linux26`

kernel thread is created, which is running in the context of `dde_linux26` and represented by an L4 thread. This way, Linux functions calls can be issued directly in the component functions of the notification server.

Because L⁴Linux' kernel threads are multiplexed by a single L4 thread, it is impossible to run the notification server as L⁴Linux kernel thread. This is because the notification server waits for IPC without timeout, which would block the whole system. Thus, an extra L4 thread had to be created for the notification server. Because this thread does not run in the L⁴Linux kernel context, calls to L⁴Linux, that is, also to the USB API, cannot be made directly.

Because of that, the component function of the server, has to copy the arguments of the notification calls to a memory region known to L⁴Linux and then has to raise an interrupt. L⁴Linux's interrupt handler then handles the notifications and returns the URBS to the Linux `usbcore`.

4.5 Isochronous Transfer

In order to keep a continuous data stream running, isochronous transfer in Linux is done using multiple URBS (normally two), which are resubmitted in their completion handler. This way, double buffering is implemented.

During the first attempts to run `USBCAM` the transmission of isochronous URBS was not working. It turned out that the `CPiA` USB device driver was not able to get a continuous data stream from the webcam.

The reason for this malfunction was, that the interrupt handler of the `HC` driver gets interrupted by the completion IPC call to the client. Therefore the interrupt handler of the `DDEUSB` server's `HC` driver could not finish fast enough to handle all completions in time.

A first attempt to avoid this problem was buffering isochronous transfers. All parameters of an isochronous transfer are stored in the `URB` data structure that the client submits. Therefore, the server has all necessary information to start the transfer itself. Instead of submitting the clients `URBS` to its `usbcore`, the server creates a ring-buffer for the data of the transfer and starts the transfer itself using normal double buffering.

For ingoing transfers, the server fills the ring buffer with received data. Whenever enough data is collected to complete a client's `URB`, an extra server thread submits a completion to the client. This way, the server's interrupt handler does not get interrupted. If there are not enough `URBS` to answer,

which can be the case if the client is too slow or has stopped transmitting URBs, the ring buffer will overflow. If this is the case, the server stops the isochronous transfer.

For outgoing transfers, the ring buffer is filled with the content of the client URBs' transfer buffers. Whenever a client submits a new URB, the transfer buffer's data is copied into the ring buffer, and a completion event is sent to the client, so that he will submit the next URB. If there is not enough data in the ring buffer to keep the outgoing stream running, the transmission is stopped.

A similar buffering mechanism is used by USB/IP [HKFS05].

The second approach was to change the priorities of DDE's interrupt handler threads. The problem is that, if the notification server runs at the same, or higher priority than the DDEUSB server's interrupt threads, the interrupt handler is intercepted, when a notification is sent to a client.

By raising the priority of the interrupt thread, this interruption could be avoided and thus isochronous transfer worked. As this approach only involved a small change in the ddekit, this solution was more suitable for this work.

5 Evaluation

In this chapter, I first give an overview on how much work was needed to create DDEUSB. Finally, I evaluate DDEUSB with respect to performance aspects regarding the throughput of USB storage devices.

5.1 Evaluation of Time and Effort

During this work, several components were created. As first part, I ported the Linux USB stack to L4 with the help of `dde_linux26`. Afterward, the DDEUSB server was created, offering USB access to L4 applications. Further, I created two virtual host controller drivers, one for `dde_linux26` and one for L⁴Linux.

5.1.1 Porting the Linux USB Stack to `dde_linux26`

Porting the Linux USB stack to `dde_linux26` was straightforward. I only had to make minor changes to the source code of the `ddekit` and `dde_linux26`.

The changes to the `ddekit` amount to 63 lines of code (LOC) and are related to `ddekit`'s page table, adding support for physical-to-virtual address translation (50 LOC), and adding the possibility to translate stack addresses to physical addresses (13 LOC).

Several functionalities (see previous chapter, Section 4.1 on page 37) had to be added to `dde_linux26`. This was done by adding 9 source-code files of the Linux kernel tree to `dde_linux26`.

5.1.2 DDEUSB Server Side

Linux' `usbcore` and `HC` drivers were not modified. `usbcore`'s part in the DDEUSB server amounts to about 8 500 LOC and the host controller drivers make up about 13 000 LOC.

Part	SLOC
usbcore	8 500
HCD (UCHI, OHCI and EHCI)	13 000
stub driver	1 500

Table 5.1: LOC count of the ddeusb server

The stub driver itself consists of about 1 500 LOC. For now, the stub driver only provides basic functionality, so it's complexity will increase when additional features are added. Such a feature, for example is forwarding USB devices on interface layer. This feature will need approximately 250 LOC.

5.1.3 DDEUSB Client Side

As base for DDEUSB client-side applications I created a small library, the DDEUSBkit. This library contains IPC wrapper and a notification server. It consists of about 200 LOC. With this library, non-legacy USB device drivers and VHCDs for legacy systems can be built.

Further I implemented two VHCDs, one for L⁴Linux and one for dde_linux26. Because these VHCDs differ only in their notification method, the complexity of both is nearly equal and lies at about 1 000 LOC.

Because the IRP representation of FreeBSD (the data structure `usb_xfer`) is similar to Linux' URB data structure it is also possible to write a VHCD for dde_fbsd. It is likely, that the complexity of this driver is equal to its Linux pendant.

5.1.4 Porting Linux USB device drivers to dde_linux26

In this thesis I ported two Linux USB device drivers to dde_linux26. Once the VHCD for dde_linux26 was created, there was no need to deal with the USB issues of these drivers anymore. The ported drivers are Linux' usbhid driver and a driver for CPiA based webcams.

The complexity of these drivers can be seen in Table 5.2.

Part	SLOC
usbcore	8 500
VHCD	1 000
legacy USB device driver part of USBCAM (V4L, CPiA, CPiA-USB)	7 700
legacy USB device driver part of USBHID	3 500

Table 5.2: SLOC count of the DDEUSB client side

5.2 Performance Evaluation

In order to evaluate DDEUSB's performance, a benchmark is required that represents the overall USB performance of the system. The results of this benchmark than can be compared between different test systems.

The FX2 USB microcontroller [Cyp] offers good way to measure a system's USB performance. Equipped with a special firmware, this device is able to offer isochronous and bulk endpoints acting as dumb data sources and sinks. This microcontroller would have been ideal for measuring the performance of DDEUSB, but regrettably such a device was not available to me during the time of this writing and another test scenario has to be found.

Because the throughput of USB storage devices also heavily depends of the overall USB performance of the system, I used USB storage to evaluate DDEUSB's performance.

First, I measured the throughput of several USB storage devices in L⁴Linux, using the L⁴Linux version of the VHCD. These values then are compared to the throughput on L⁴Linux with legacy USB drivers and native Linux. Afterward, I compare the different buffering mechanisms introduced in Section 3.2.5 on page 32.

5.2.1 Test Setup

The USB storage devices used in the experiments are a USB hard disk enclosure with a Myson Century CS8813G-103 chipset, equipped with a Hitachi IC25N040 ATMR04-0 40 GB hard disk, and a low budget 4 GB USB flash drive. To measure the throughput of the storage devices, I used two benchmarks: First, Bonnie++ [BON], a widely used file-system benchmark, was used. The

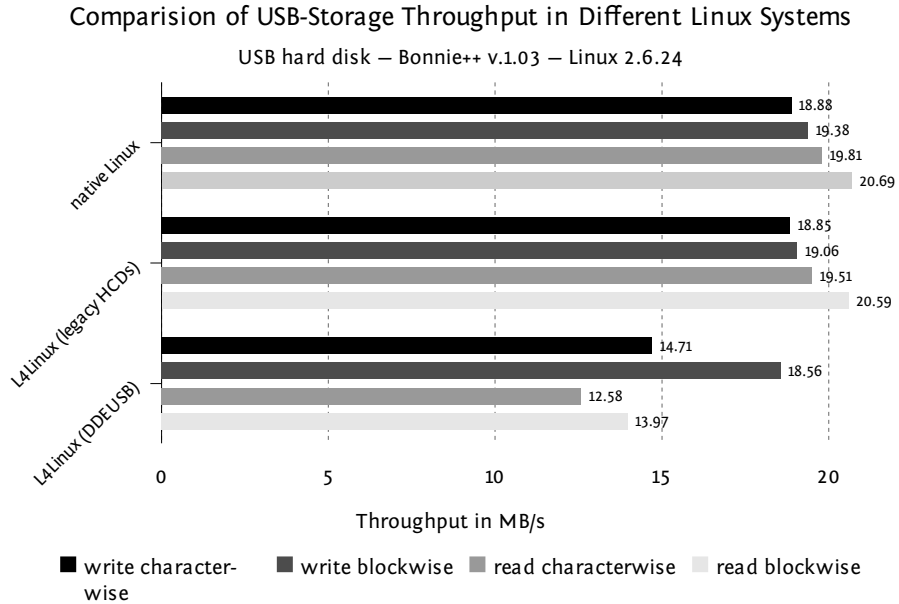


Figure 5.1: Comparison of the different Linux Systems.

second benchmark used was copying 1GB of data from /dev/zero to disk and from disk to /dev/null. All benchmarks were run in Linux respectively L⁴Linux 2.6.24 on a 2 GHz Pentium Celeron with 512 MB system memory, whereas L⁴Linux got 400 MB of the system memory.

5.2.2 Discussion of the Test Results

In the following the I discuss the results of the tests.

Comparison of Different Linux Systems

At first, I explain Figure 5.1, which represents the test readings of Bonnie++ on the different Linux systems. Expectedly, native Linux has the highest throughput of all systems followed by L⁴Linux with the legacy USB system. The reason for the little performance decrease of L⁴Linux can be explained

with a context-switch from the L⁴Linux userland task (i.e., Bonnie++) to the L⁴Linux kernel task when the userland task does the read and write system calls.

However, the biggest performance loss can be observed in the L⁴Linux system using the DDEUSB VHCD. This is the case for several reasons. First, there is a context-switch from Bonnie++ to L⁴Linux, whenever it does a read or write request. Second, there is a context-switch from L⁴Linux to the DDEUSB server, whenever a URB is submitted, and additionally, a context-switch from the DDEUSB server to L⁴Linux whenever a URB is completed. Therefore, there are three times as many context-switches leading to the translation lookaside buffers being flushed than in unmodified L⁴Linux.

The next observation is, that DDEUSB is performing better on writes than on reads. This behaviour will be explained during the discussion of the dd-benchmark test readings in Section 5.2.2 on page 55.

The higher performance loss when reading or writing characterwise is easy to explain: The performance loss is caused by the higher number of `submit_urb()` calls that are necessary to transfer the same amount of data, each of which leading to the previously explained context-switches.

Comparison of Different Transfer Modes

In Figure 5.2 we can see the throughput of DDEUSB using different methods to transfer the URBs transfer buffer, measured with Bonnie++. Again, we can see in the diagram that reads perform better than writes on the same system. Further, for all transfer modes blockwise transfer is performing better than characterwise transfers.

Fall-back method (fall-back) The fall-back method, which uses string IPC has the lowest throughput as expected.

Using Physical Addresses (phys adr.) The highest throughput can be seen, when the `fast_submit_urb()` call using physical addresses for buffer transfer is used. This behaviour was also expected

Using temporary mappings (sh.-mem.) When temporary mapping of the transfer buffers is used, DDEUSB's performance is only slightly better than using the fall-back method. This is the case, because of the introduced mapping overhead. For each submitted URB, the dataspace containing

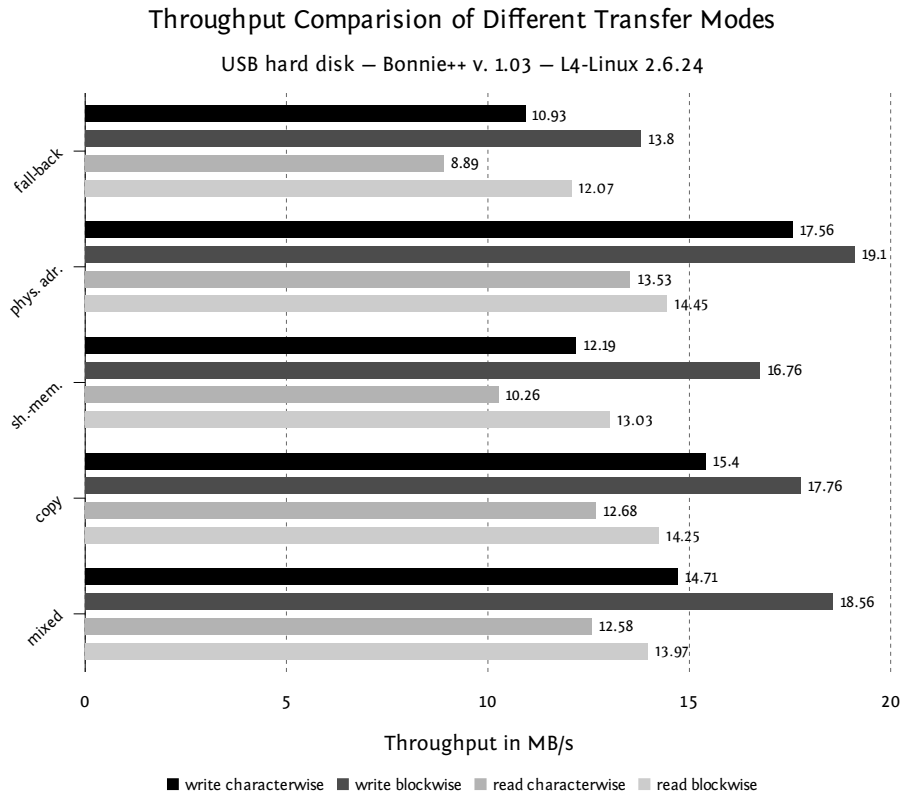


Figure 5.2: Comparison of the different transfer mechanisms.

the transfer buffer has to be found. Then the client needs to make this dataspace accessible to the server, and further the server has to map and unmap this dataspace. All these tasks lead to a high overhead, and thus it only makes sense to use this method for large chunks of data.

Copy transfer buffers into argument container (copy) For testing purposes the size of transfer buffers fitting in an argument container was enlarged to 64 KB, which is the largest transfer buffer size the Linux usbstorage driver uses. As we can see in the diagram, this method produces better performance, than the fall-back method using string IPC.

Mixing the previous two approaches (mixed) Because it is impracticable to have 64 KB buffers in each argument container, a mixed approach is used. The maximal buffer size that is copied into an argument container is 4 KB. Larger buffers are transferred using temporary mappings. As we can see, the performance is similar to the previous method. For reading blockwise it is 1KB/s faster than copying, which indicates that temporary mappings are useful for transferring big transfer buffers.

Test Readings of the dd-Benchmark

In the next experiment 1 GB of data was copied from `/dev/zero` to the storage device on block layer using `dd`. Afterward, 1 GB was copied from the block device to `/dev/null` in order to measure read performance.

The test readings are illustrated in Figure 5.3 for the USB hard disk and in Figure 5.4 for the USB flash drive.

On the whole, the previously mentioned observations are also applicable here. However, during these tests, I investigated, why writes perform significantly better than reads.

Upon examination it showed, that Linux submits during reading about 4.5 times as many URBS to the USB core as for writing. I measured, that for copying 1 GB to the USB device 10 459 URBS were submitted. Upon reading, 46 536 URBS were submitted. Because of the additional context-switches, the throughput becomes smaller.

Regrettably, the reason for the higher number of submitted URBS could not be found during this work. However, this problem is not related to DDEUSB, but to the Linux block layer, the file-system layer or the SCSI subsystem.

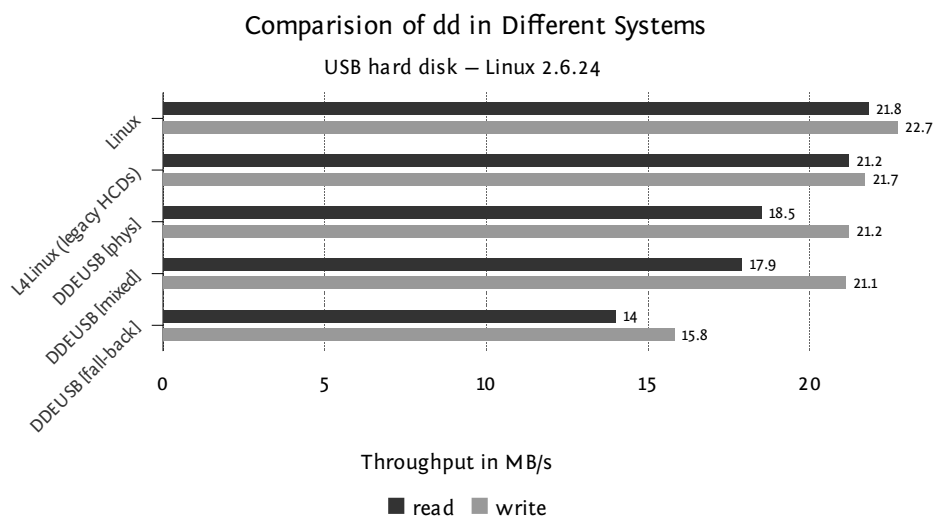


Figure 5.3: Comparison of the dd-troughput on different Linux systems – hard disk.

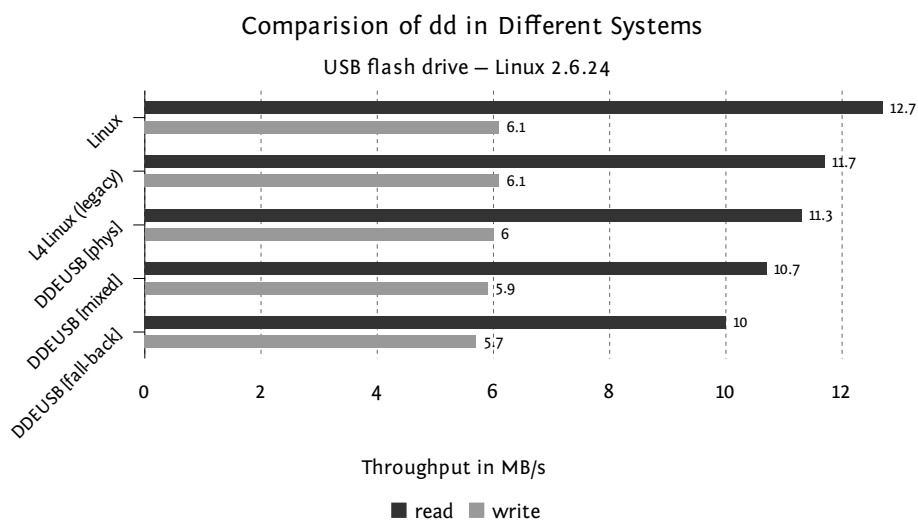


Figure 5.4: Comparison of the dd-throughput on different Linux systems – flash drive.

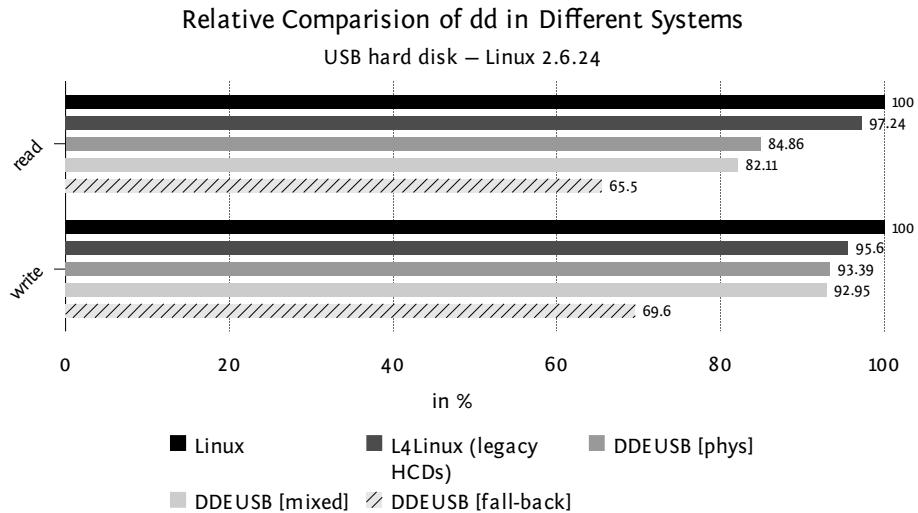


Figure 5.5: Relative comparison of dd-throughput on the different Linux systems – hard disk.

From the test readings of the flash drive (Figure 5.4), it is visible that the throughput for writing is nearly constant at 6 MB/s. This behaviour is also not caused by DDEUSB, but by the limited write rate of the flash memory.

Figure 5.5 shows the relative throughput of the USB hard disk of different L⁴Linux based systems compared to native Linux.

Relative Comparison of the Linux Systems

As we can see, with DDEUSB it is possible to reach at most 93.39 percent of native Linux' throughput, which is about 2.31 percent points less than in L⁴Linux with the legacy HC drivers.

Further, we can see, that using the mixed transfer method, which offers a good trade-off between performance and security, it is possible to reach at most 92.95 percent of native Linux' throughput. This is 2.77 percent points less than in L⁴Linux with legacy HC drivers.

The fall-back method only reaches 69.6 percent of native Linux' throughput. However, this transfer method is only used, when outside influences, like

security requirements, do not allow the use of the faster transfer methods. Thus the decreased performance of the fall-back method is the price to be paid for security.

6

Conclusion and Future Work

In this thesis I introduced DDEUSB, a maintainable replacement for USB for DROPS. The solution offers USB functionality to DROPS in a generic way because the resulting interface is not bound to a specific use case.

Further, I evaluated my design by implementing three applications:

1. USBHID, offering a USB human-input device driver to L4Input, the input library of L4Env,
2. USBCAM, a webcam viewer application for CPiA based USB webcams running under DOPE, and
3. Finally, a DDEUSB VHCD for L⁴Linux was created, which enables L⁴Linux to use USB functionality without drawbacks to the system's security.

Additionally, mechanisms for efficient communication were discussed and evaluated and it was shown, that with DDEUSB it is possible to multiplex USB functionality between several L4 tasks with up to 93 percent of native Linux' performance.

Future Work

As DDEUSB is well integrated into dde_linux26, now the whole range of Linux USB device drivers waits to be ported to L4. So, for example, the Linux' USB storage can be integrated into windhoek, a block device server for L4Env.

Also integration of DDEUSB into other projects like dde_fbsd is desirable. Like that, it can be proven that the interface of DDEUSB is generic enough, to be used for non-Linux-based client systems.

Although it is possible to forward USB devices to clients, a mechanism is required that decides which client is allowed to subscribe for what types of USB devices (see section 3.2.3 on page 31). Such a solution exists with `ioguard` [Hän07], but `DDEUSB` has not yet been integrated into this solution.

Further forwarding devices as whole may not be sufficient under certain circumstances. For example, regarding security aspects, it might be useful that a USB device driver only gets access to certain functionalities of a device. So, a method has to be created which allows to forward USB devices on interface layer (see Section 3.2.4 on page 31).

Finally, it is interesting to investigate the possibilities to increase the performance of `DDEUSB`. One imaginable way to do that is implementing a special `URB` submit method for scatter-gather transfers. Using a special scatter-gather transfer method only one call to the `DDEUSB` server would be necessary where now multiple `submit_urb()` calls are necessary, leading into less context switches and thus better performance.

Index

A

argument stack 40

B

bandwidth reservation 7
Bonnie++ 52
bulk transfer 7
bus-powered 5

C

class descriptor 11
con 14
configuration 10
configuration descriptor 10
control transfer 6

D

DDE 14–15
dde_fbsd 15
dde_linux24 15
dde_linux26 15
ddekit 15, 49
DDEUSB design goals 23
DDEUSB example applications 42
DDEUSBkit 29, 50
descriptors 10–11
device descriptor 10
device ID 10
DM_phys 14
DMA 38
DOpE 44
DROPS 12–15

DROPS streaming interface 34
DSI . . *see* DROPS streaming interface

E

EHCI . . *see* Enhanced Host Controller Interface
endpoint 6
 zero, 7
endpoint descriptor 11
Enhanced Host Controller Interface . 5

F

Fiasco 13
frame list 8
FreeBSD 50

H

HC *see* USB host controller
high speed 4

I

I/O request package 7
inter process communication 13
interface 10
interface descriptor 10
interrupt transfer 7
ioguard 31, 62
IPC . *see* inter process communication
IRP *see* I/O request package
isochronous transfer 7, 46

Index

- L**
- L4 13
 - L4Env 13–14
 - L4input 43
 - l4io 14
 - L⁴Linux 51
 - L⁴Linux 45
 - l4rm 14
 - l4util 14
 - Linux input subsystem 43
 - Linux USB stack 16
 - long IPC 39
 - low speed 4
- M**
- microkernel 13
- N**
- names 14
- O**
- OHCI *see* Open Host Controller Interface
 - old_dde_linux26 15
 - Open Host Controller Interface 5
- P**
- platform device 37
- R**
- root hub 5
- S**
- self-powered 5
 - short IPC 39
- string descriptor 11
 - stub driver 28
- T**
- TD *see* transfer descriptor
 - transfer buffer 8, 32, 41
 - transfer descriptor 5, 7–9
 - transfer speeds 4
- U**
- UHCI *see* Universal Host Controller Interface
 - Universal Host Controller Interface 5
 - URB 17
 - URB cache 38
 - USB 3
 - USB 2.0 4
 - USB 3.0 4
 - USB cable 4
 - USB design goals 3
 - USB device classes 12
 - USB Devices 6
 - USB for DROPS 19–20
 - USB frames 9–10
 - USB HID class 42
 - USB host controller 5
 - USB hub 5
 - USB Request Block *see* URB
 - USB Transfer Types 6
 - USB/IP 20–21, 47
 - USBCAM 44
 - usbcore 16
 - USBHID 42
 - usbstorage 51
- V**
- V4L *see* Video4Linux
 - VHCD 45, 50

VHCI	20
Video4Linux	44

Bibliography

- [ABG⁺86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. MACH: A new kernel foundation for UNIX development. Technical report, Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA, 1986. 13
- [AD01] Don Anderson and Dave Dzatko. *Universal Serial Bus System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 4
- [Aig07] Roland Aigner. *DICE Version 3.3.0 User's Manual*, November 2007. Available at URL: <http://www.inf.tu-dresden.de/content/institutes/sya/os/forschung/projekte/dice/manual-3.3.0.pdf>. 13
- [BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990. 40
- [BON] <http://www.coker.com.au/bonnie++/>. 51
- [CDI⁺96] Compaq, DEC, IBM, Intel, Microsoft, NEC, and Northern Telecom. Universal serial bus specification revision 1.0, 1996. 3
- [CHPI⁺00] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Phillips. Universal serial bus specification revision 2.0, 2000. 4
- [CMN99] Compaq, Microsoft, and National Semiconductor. OpenHCI - open host controller interface specification for USB - revision 1.0a, 1999. 5
- [CYC⁺01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. pages 73–88, 2001. 13

- [Cyp] Cypress Semiconductor Corporation. EZ-USB FX2 USB microcontroller. Company website: <http://www.cypress.com>. 51
- [Fes02] Norman Feske. DOpE – a graphical user interface for DROPS. Diploma Thesis, TU Dresden, Chair for Operating Systems, November 2002. Available at URL: http://os.inf.tu-dresden.de/papers_ps/feske-diplom.pdf. 42
- [FH03] Norman Feske and Hermann Härtig. Dope - a window server for real-time and embedded systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 74, Washington, DC, USA, 2003. IEEE Computer Society. 42
- [Fri06] Thomas Friebe. Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebssystemkerns, March 2006. Diploma thesis, TU-Dresden. 15, 20
- [Gri03] Gerd Griessbach. USB for DROPS. Diploma thesis, TU-Dresden, Chair of Operating Systems, March 2003. 19
- [Hän07] Lucas Hänel. ACPI for L4Env. Großer Beleg, TU-Dresden, Chair of Operating Systems, June 2007. 31, 62
- [HBB⁺98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998. 12
- [Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Diploma thesis, TU-Dresden, Chair of Operating Systems, August 2001. 15, 30
- [HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998. 30
- [HKFS05] Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara. USB/IP: a peripheral bus extension for device sharing

- over IP network. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 42–42, Berkeley, CA, USA, 2005. USENIX Association. 20, 47
- [Hoh96] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: <http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-14/>. 30
- [Hoh98] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD-FI-12, TU Dresden, December 1998. Available from URL: http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz. 13
- [Hoh02] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002. 13
- [Int96] Intel. Universal host controller interface (UHCI) design guide - revision 1.1, 1996. 5
- [Int02] Intel. Enhanced host controller interface for universal serial bus - revision 1.0, 2002. 5
- [Lie95] J. Liedtke. Towards real μ -kernels. Arbeitspapiere der GMD No. 958, GMD — German National Research Center for Information Technology, Sankt Augustin, December 1995. 13
- [Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996. 13
- [LRH01] Jork Löser, Lars Reuther, and Hermann Härtig. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09-August-2001, TU Dresden, August 2001. Available from URL: http://os.inf.tu-dresden.de/papers_ps/dsi_tech_report.pdf. 35

Bibliography

- [Men04] Marek Menzer. Portierung des DROPS Device Driver Environment (DDE) für Linux 2.6 am Beispiel des IDE-Treibers. Großer Beleg, TU Dresden, Chair for Operating Systems, January 2004. Available at URL: http://os.inf.tu-dresden.de/papers_ps/menzer-beleg.pdf. 15
- [Ope03] Operating System Research Group, TU Dresden. — L4Env — An Enviroment for L4 Applications, June 2003. Available at URL: <http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf>. 13
- [USB01] USB Implementers' Forum. *Device Class Definition for Human Interfaces Devices (HID)*, June 2001. 42
- [USB06] USB Implementers' Forum. USB.org - Defined 1.0 Class Codes, March 2006. http://www.usb.org/developers/defined_class. 12
- [VLS97] VLSI Vision Ltd. *Software Developer's Guide for CPiA Cameras*, 1997. Available at URL: <http://webcam.sourceforge.net/docs/developer.pdf>. 44