Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 88 b

Systems Group, Department of Computer Science, ETH Zurich

Barrelfish USB Subsystem

by
Reto Achermann

Supervised by
Prof. Timothy Roscoe, Dr. Kornilios Kourtis

August 2013

inf | Informatik
Computer Science

Abstract

In the early days, connecting devices with a computer involved many different connectors and the attachment of the devices had to be done before the host computer was powered on. The universal serial bus (USB) has replaced many of those old-fashioned connectors and provides a unified and universal connector with hot-plug support. The versatility of the USB makes it an indispensable way of connecting devices with a host computer. Therefore having USB support in an operation system is a precondition for extensibility and usability in the sense of extending the storage capacity with mass storage devices or to provide a way of user input with a USB keyboard. In the context of the multi-kernel operating system Barrelfish, the USB subsystem has to be designed in a distributed fashion and run as pure user-level services. The design idea is to separate the USB stack into two parts. An USB Manager that provides the USB driver interface and handles the configuration of the host controller hardware and a USB client driver for each attached USB device. The USB subsystem should provide hot-plug capabilities that the USB client drivers are started and terminated when a device is attached respectively detached on an USB port.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction and Motivation

1.1 Introduction

In the early days, different devices were connected using various different kind of ports such as PS/2 or RS-232 for keyboards / mice, parallel port for printers and Firewire for cameras. The problems with these ports were, that they could not be used interchangeably limiting the number of devices and their lack of hot-plug functionality.

The inventors of the USB had a clear vision in mind. The USB specification [19] gives three main points motivating why USB was introduced:

- Connection of the PC to the telephone
- Ease-of-use with just one connector for many devices
- Port expansion with hubs to provide more attachment points

The success of the Universal Serial Bus (USB) is indisputable. Today, most of these old-fashioned ports such as the serial interface are hard to find on modern computers especially on portable devices. These ports were replaced by USB ports providing a small and uniform interface for connecting various kinds of devices to the host computer.

Overall, the USB standard is getting more and more important as many of today's peripheral devices are connected to a computer using an USB cable. One of the main benefits of the USB is, that the USB wires can not only be used for pure communication purposes between host and device, but also provides power up to 500mA[19] to the device over the same wire[1]. This enables the devices to become usable by using just a single cable connection. Hence portable devices can be charged why there are connected to the host computer during the synchronization process for instance.

Furthermore the not only peripheral devices can be connected via USB, but also two computers may use the USB for communication as it is done with smartphones or tablets nowadays (mostly with the smartphone or tablet in the

[1] The Wireless USB standard is the exception [23]

device mode). With a transfer speed of up to 10 Gbps, as announced in a press release early this year [7], together with support of up to 127 different devices concurrently [19] the USB provides a fast, uniform and extensible way to connect the different devices with a host computer.

Despite the versatility of devices using USB as a connection to the host computer they all must[2] obey the standards defined by the USB specification [19]. If the host operating system provides a reasonable framework for accessing the USB, the development of a new driver for a device is expected to be much easier than for traditional devices. This claim is backed by the fact that USB device drivers do not access any hardware registers directly and operate only via the USB interconnect. The only part of the USB subsystem that has hardware registers is the USB host controller. The operation system has to provide the needed abstractions of the underlying host controllers [13, 12, 3, 14] as well as for managing the bus.

To sum up, having USB support in an operating system is crucial and an important factor of success. Devices such as keyboards, network cards and mass storage devices can be attached to the USB and enable the system to communicate, react to user input and store files.

In order to solve the engineering problem of the USB subsystem in a distributed operating system, the following questions have to be answered

- How is the USB subsystem initialized?
- How the devices are configured and which instance is responsible for it?
- How the device drivers can access the devices?
- How the data is moved from the driver to the device and vice versa?
- How are the device drivers found and started?
- How are the host controllers abstracted?
- How are the detach / attach events handled?
- How must the interfaces be defined to meet the specification requirements?

Answering these questions form a basis to come up with a design and semantics of a USB Subsystem. Of course, this list is not complete and many other questions arise while implementing the system.

1.2 Thesis Outline

The thesis consists of several parts. In Chapter 2 an overview of related work is given. Chapter 3 outlines some specialties in Barrelfish. Chapter 4 briefly introduces the main aspects of the USB architecture. Chapter 5 shows the PandaBoard hardware configuration and the changes that were made to Barrelfish. Chapter 6 talks about the design of the USB subsystem followed by a discussion of limitations and future work in Chapter 7.

[2] As always there are devices that have some bugs and that do not follow the specifications entirely, referring to the USB quirks of FreeBSD [8]

Chapter 2

Related Work

Modern operating systems such as FreeBSD [8], Linux [15] or Microsoft Windows [5] have their own implementation of the USB stack providing different interfaces and usage semantics. However, all of the mentioned operating systems fall into the category of monolithic kernels and their drivers live mostly in kernel space. Hence, they not really suite the design principles of a microkernel based operating system such as Barrelfish [2, 20].

2.1 USB for DROPS

The L4 operating system [17] is based on a micro kernel and uses user-level services to provide the necessary OS services. In the year 2003 an attempt to bring USB to the L4 system was attempted by Gerd Griessbach [11]. He took the Linux 2.4.20 USB Stack [15] and mapped the API to the message passing interface of the L4 kernel. That way the Linux USB stack could be used as is and a wrapper library ensured the communication between the different servers. The number of different API calls made the overall system complex.

2.2 The USB/IP Project

In 2005, the USB/IP project [10] came up with a different approach by abstracting the communication via the network stack. The USB requests are encapsulated inside the IP packets and transmitted over the network. Basically there are two components in the system a client, which makes use of a virtual host controller interface and a server that uses a stub device driver library. The USB requests are exchanged between the two modules.

2.3 USB for the L4 Environment

In 2008, Dirk Vogt investigated in his Master Thesis [30] how the Linux [15] USB stack can be ported to the L4 environment. He took up the idea of the USB/IP project with the stub driver and the virtual controller interface. As a result of his work, he ported the Linux 2.6 USB Stack to L4 by preserving the Linux API such that the Linux client drivers can be ported easily to L4.

Chapter 3

Barrelfish

The Barrelfish operating system[1] is a research project of the Systems Group at ETH Zurich[2] in collaboration with Microsoft Research. As a multi-kernel operating system [20] supporting many different architectures, the kernel is viewed as a CPU driver which exports the hardware abstraction of the underlying hardware architecture. The kernel provides only a basic set of necessary services such as memory mappings, context switching and message passing. Other functionality such as memory management is provided by user-level servers. The system design follows the trend of a distributed systems architecture [2].

This section will outline some special aspects that are related for implementing a USB subsystem on Barrelfish.

3.1 USB on Barrelfish

In 2009, there was an attempt to bring USB functionality to the Barrelfish operating system. Animesh Trivedi provided an USB implementation for Barrelfish in the context of his master thesis about hot plugging in a multikernel operating system[28]. In his master thesis he solved the problem stated in the introduction with tree different processes and a library. First the USB manager which is responsible for management related activities i.e. the bus driver, secondly the EHCI host controller driver which handles the hardware specific part of the USB stack and third the USB client driver which is responsible for the operation of the USB device. In addition to that, the USB memory library takes control of the memory allocation and management, since there are many constraints about the alignment of the different data structures used by the host controller hardware.

His implementation focused on the EHCI [13] host controller and a mass storage driver. Therefore, the support of HID devices in particular the USB keyboard is missing in his implementation.

With the evolution of Barrelfish and the changes that were made to the API, this USB implementation is not working anymore. According to the comments in the code and the FIXME file, the code contains some bugs as well as inconsistencies with the coding guidelines.

[1] `http://www.barrelfish.org`
[2] `http://www.systems.ethz.ch`

3.2 Capabilities

Barrelfish manages critical system resources such as memory or page tables by the use of capabilities [21]. The capabilities are stored in a dedicated region called the `CSPACE` in each domain. Only the kernel can access and manipulate the capabilities in the `CSPACE` directly. User-level domains only deal with references (`caprefs`) to those capabilities stored in their own `CSPACE`. Not only resources are tracked via capabilities, but also many privileged system calls are in fact capability invocations where the caller needs to present the capability to get the system call executed.

All capabilities are typed and there exists rules how to retype them into a more specific type such as creating a frame capability out of a RAM capability. The rules prohibit certain retype operations because of security related issues e.g. a page table capability must never be mapped as a virtual memory page. In order to have access to the memory range of the device a RAM capability of that range can be retyped to a device frame capability and then mapped into the virtual address space of the driver domain.

3.3 Device Manager

Since we are dealing with device drivers, we are also have to consider how they are managed and started. Gerd Zellweger elaborated in his master thesis [31] how the startup of the device drivers can be coordinated and how they are managed. The main focus of his thesis was the coordination and synchronization between processes. The device manager called Kaluga was the result of a case study in his thesis. Initially, Kaluga was designed to run together with PCI and the system knowledge base.

3.4 Device Drivers in Barrelfish

Device drivers in Barrelfish are implemented as user-level servers. Each server exports an interface to provide access to device related features. The interface can be invoked by the clients using message passing. To simplify the implementation of message passing Barrelfish comes with a tool called Flounder (Section 3.5). The device driver domain can access the hardware registers directly as soon as the capability specifying the memory range of the device is mapped in the virtual memory space of the driver domain. That way the kernel is out of the data path and the hardware related issues can be tackled by the user-level server directly.

As stated above, in order to have access to the hardware registers the driver domain needs a device frame capability. Therefore it is crucial for the driver domain to obtain it either the capability is already supplied as a spawn parameter or can be requested afterwards.

x86 Architecture On x86, besides the IO bus, most of the devices are located on the PCI bus. The PCI domain is responsible for the initial configuration of the PCI and hence has the capability for the whole memory range of the PCI bus. When the PCI driver is finished with configuring the bus, the device driver

domains for the device located on the bus are spawned by Kaluga. The device driver domain requests the needed capability from the PCI domain afterwards.

ARM Architecture System on a Chip (SoC) based architectures like the Texas Instruments OMAPP44xx [27] have every supported device at a predefined memory location inside the chip and hence do not have a PCI bus. With no PCI bus there is no PCI domain which hands out the device capabilities and thus obtaining a device capability on the ARM architecture was an open issue which had to be solved. Section 5.3.2 describes the changes to the Barrelfish kernel that were needed to obtain the device capability.

Mackerel

Once the hardware registers are accessible by the device driver domain the device can be configured. Accessing hardware registers involves a lot of shifting and masking which is tedious and error prone. Barrelfish solves this problem by a tool called Mackerel [24]. Mackerel comes with a domain specific language (DSL) that allows transcribing the register interface as it is defined in the specification documents in a natural way. After some basic checks such as uncommon register sizes or overlapping registers, Mackerel generates C code from the Mackerel file. The framework hides all the bit shift and masking behind a consistent interface and therefore simplifies and reduces the bugs when accessing the hardware registers. The only thing that has to be done is initializing it with the virtual base address of the device.

3.5 Flounder

As explained in Section 3.4, in Barrelfish every device driver runs in its own user-space domain. Each driver domain acts as a server and exports a service interface. Other domains, the clients, need a way to invoke the service interface. This can be abstracted by a communication channel between server and client each invocation involves sending messages between the two domains.

In Barrelfish the message passing is called inter-dispatcher communication (IDC) [1]. As with Mackerel for hardware register access, Barrelfish uses a tool called Flounder with its own DSL to abstract the low level implementation details. The service interface is defined using the Flounder interface DSL and fed into the interpreter. Out of the DSL interface specification, Flounder generates C code which is then included by both client and server domains. The generated C code contains stubs for each function of the interface.

The server exports the interface and gets a reference of its own service back where as the client binds to the service using this reference. In general the communication is two way and in order to handle the messages, callback functions have to be registered which are then called by the generated stubs.

Flounder supports asynchronous messages and synchronous messages with remote procedure call (RPC) semantics. A message is sent like a normal function invocation. The sender of the message calls one of the generated transmit functions on the Flounder binding.

3.6 Naming and Interface References

We have seen in the previous section that Flounder provides a framework for message passing in Barrelfish. To send a message, a channel between the two domains has to be opened during the binding process. In order to bind, the location where the service is running has to be known. From that, two questions arise: how are the services referenced and how can one get such a reference?

Interface references (iref) A service is started by exporting the implemented interface. As a return value of the exporting process, the server gets an interface reference (`iref`) back. The `iref` is an integer value that uniquely identifies the exported service. The client uses this `iref` as a parameter for the binding process when the communication channel is initialized.

Naming The client does not know the `iref` of the server's interface a priori. After the export process, the server has to register the returned `iref` with the name service. The service name-`iref` association is then stored in the system knowledge base (SKB) [25] which runs as another OS service[3]. A client that wants to bind with the exported service can obtain the `iref` by doing a name service lookup which queries the SKB for the `iref` information.

[3]The SKB is a database where every piece of information of the system can be stored, analyzed and queried.

Chapter 4

USB Architecture

The USB architecture is completely defined in the Universal Serial Bus Specification [19] and the different host controller specifications [3, 12, 13, 14]. This chapter briefly describes some of the most important points concerning the topology, protocols and interfaces of the USB specification revision 2.0.

4.1 USB Topology

The USB topology can be viewed as a tree rooted at the so-called host. The USB specification [19] describes the topology as a "tired start topology" where the tiers correspond to the depth levels of the tree. Figure 4.1 shows an example setup with 10 connected device, three of which are forming a compound device and another three are hub devices. The maximum depth of the topology tree is 7 starting from the root at depth 1. This implies that only function devices (Section 4.1.2) can be attached at depth 7 (hubs would be unusable). This is due to timing constraints and the USB driver has to ensure that these constraints are not violated.

4.1.1 Host

The host is the root of the USB topology tree. It is not possible to have two hosts within the same USB topology. The host system consists of two main parts: the host controller and the root hub.

Host Controller

The host controller is a physical device which resides either on the PCI bus or within a SoC. In contrast to the other USB devices is the host controller is the only device in the USB topology whose hardware registers are memory mapped and can be directly accessed by software. Currently there are four different host controllers available supporting different USB revisions and register interfaces:

1. Universal Host Controller Interface (UHCI) proprietary standard by Intel for USB 1.0 [12]
2. Open Host Controller Interface (OHCI) is an open standard by several companies for USB 1.1 [3]

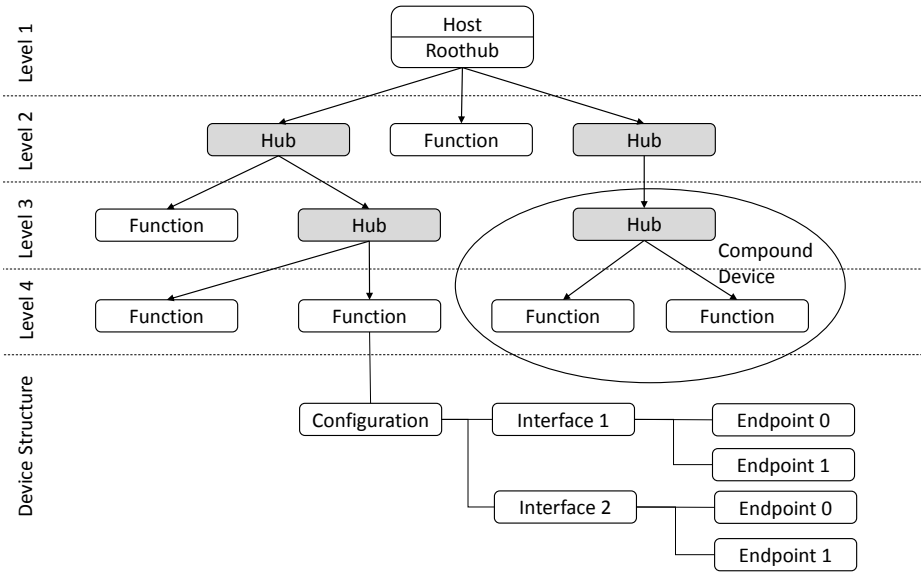

Figure 4.1: USB Topology

3. Enhanced Host Controller Interface (EHCI) was introduced for USB 2.0 [13] and defined by collaboration of several companies.

4. Extensible Host Controller Interface (XHCI) is the 3rd generation of host controllers and the only one that supports all USB revisions 1.0 to 3.0 [14]

Referring to Section 4.2.3 for more details about the host controller driver interface or Section 4.2.4 for EHCI specific details. The companies that were involved in the development of the host controllers can be obtained by the respective host controller specifications [12, 3, 13, 14].

Root Hub

The root hub is a virtual USB device which is emulated by the host system and provides at least one port where other devices can be attached. In contrast to other USB devices, requests are not sent to the device but are rather handled virtually by the host software accessing the registers on the host controller. Thus the root hub is the only hub device which has memory mapped registers. The host controller revision defines the operational capabilities of the root hub.

4.1.2 USB Devices

USB Devices can be split up into two fundamental categories each having different usage types and operational capabilities. Despite the elementary difference there are several things that all USB devices have in common like they are residing on the network like USB interconnect and have no directly accessible

registers. One of the biggest advantage of USB devices is the fact that they can be attached or detached to the system at any time (hot-plug functionality).

USB Hub Devices

The number of ports provided by the root hub is limited. To overcome this limitation additional USB hub devices are connected to increase the number of attachment points in the USB topology as shown in Figure 4.1. That way, from a single root hub port, attachment points for up to 127 devices can be provided under the condition to meet the USB specification requirements. This implies that hubs cannot be attached at depth 7 in the topology because if so they would provide attachment points that would be unusable[1].

High speed hubs contain a transaction translator (TT) that enables the attachment of full or low speed devices to a high speed hub by translating a micro-frame (125 us) to a normal frame (1ms).

Besides extra attachment points, USB hubs do not provide new functionality.

USB Function Devices

USB devices that provide some form of functionality such as storage of sensor data are called functions. Function devices do not provide any additional attachment points and act as the sink or source of a data transfer between host and USB device. As an example one may want to consider keyboards or mass storage devices.

Compound Devices It is also possible to combine the two categories above into a compound device. Compound devices look like a single device with a single cable, but are in fact a combination of multiple USB devices each of which having a different address. As an example, one may consider the USB keyboard with integrated USB hub. Compound devices cannot be attached at depth 7 since they occupy two depth levels at once.

The Device Descriptor

Each device contains an 18 byte data structure called the device descriptor as shown in Table 6.4 on page 52. The device descriptor contains all the relevant information about the connected device such as USB revision number or device class code. The descriptor can be obtained by executing the `GET_DESCRIPTOR` request on the default control pipe (see Section 4.4 for an overview of transfer types). USB devices fall into predefined device classes which are assigned by the USB-IF[2]. The class codes are used to identify the device and find a matching function driver[3].

Device Configurations

Each device has at least one configuration. The actual number of configurations is defined in the device descriptor. During the attachment process the configu-

[1] It may work, but the USB specification prohibits it due to timing reason
[2] http://www.usb.org/about
[3] For some classes, the relevant information is found in the interface descriptor

ration value is set. Each configuration is defined by a configuration descriptor and has a different set of interfaces and endpoints (Sections 4.1.3 and 4.1.4).

Device Speeds

A device falls into one of the following four speed categories depending on the USB revision and the capabilities of the device.

- Low-Speed (LS). USB 1.x standard with 1.5 Mbps [19].
- Full-Speed (FS). USB 1.x standard with 12 Mbps [19].
- High-Speed (HS). This was introduced by USB 2.0 and supports up to 480 Mbps [19].
- Super-Speed (SS). This was introduced by USB 3.0 and supports up to 10 Gbps [7].

The speed categories are backward compatible e.g. a high speed device may operate as a full speed device when connected to an OHCI controller.

Device States

Once a device is attached to the USB it has always a clear defined state. Depending on the state, the reaction to transfer requests is either undefined or results in an error condition. The possible device states are

- Attached: This is the initial state when a device is plugged in in a port of a hub which is already configured.
- Powered: The port was enabled and the device got power. The reset sequence has not yet been executed.
- Default: The reset sequence was executed and the device responds to the default address (`0x00`).
- Address: The device was an unique address assigned using a `SET_ADDRESS` request.
- Configured: The device is configured by executing a `SET_CONFIGURATION` request with a valid configuration value. The device is now usable.
- Suspended: Power save mode if there is no bus activity for some time. The device is not usable in the suspended state.

State transitions usually happen, when a device request is executed. Referring to the USB specification [19] chapter 9 for a complete description of state transitions.

4.1.3 Interfaces

Each device configuration has at least one interface and each interface contains at least one endpoint (Section 4.1.4). An interface is characterized by the endpoints it groups together and can be viewed as a representation for a single feature of a device: if a device has two different features then it also has two interfaces e.g. a keyboard/track-pad configuration may consist of two interfaces one for the keyboard and one for the track pad. Figure 4.1 shows an example with to interfaces and each of which has two endpoints.

4.1.4 Endpoints

Each interface has one or more endpoints associated with it. Each endpoint has a defined, device dependent endpoint address and data flow direction i.e. a particular endpoint may be the source or the sink of data flow through the USB. Endpoints have an associated type supporting pipes of exactly this type referring to Section 4.4 for an overview of transfer types.

Endpoint Zero There is a special, dedicated endpoint zero, which is used as the default control pipe for executing device requests such as setting the device address. The data-flow of the control pipe is bidirectional.

4.1.5 Attachment and Detachment of Devices

Any USB device can be attached to a free port of a hub device. Once a new device is discovered the host software configures it and updates the USB topology accordingly. If the attached devices was a hub, then the whole procedure may be repeated recursively for the potential devices connected to that hub.

When a device is detached an interrupt may be fired again and the status bits signal that something has changed. The USB software handles the removal from the topology. If the detached device was a hub, then the whole process need to be repeated recursively on all child devices.

Section 6.4 will give a more detailed view on the steps to perform when a device is attached or detached and how it is handled in the suggested USB system architecture.

4.2 USB Stack

Like the network stack, the USB stack also consists of several layers which provide different levels of abstractions and ensure independence of the particular host controller implementation. The USB stack is shown in Figure 4.2.

4.2.1 USB Device Drivers

The top most layer of the USB stack is the USB device driver for a particular USB device like a USB keyboard. USB device drivers are often referred to as USB client drivers. This layer knows the particular configuration and usages of its function device and hence is responsible for setting up the needed transactions and specific configurations.

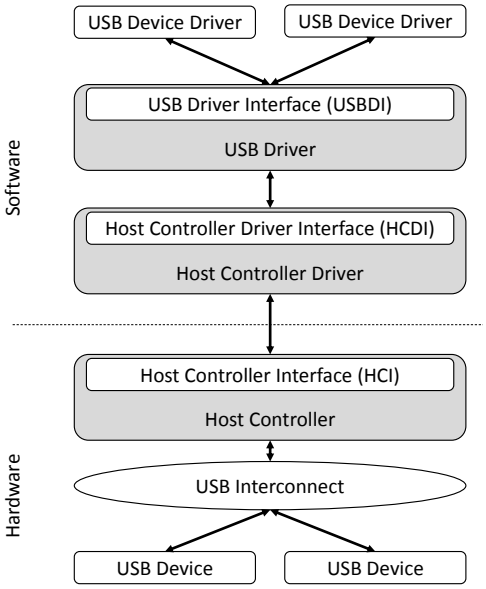

Figure 4.2: USB Stack

USB Device Class Drivers As an alternative option, it is also possible to introduce another layer at this level. Every USB device falls exactly into one device class which resembles common functionality by all devices of this class. A device class driver handles common device class functionality.

USB Hub Drivers The drivers for the USB hubs play a little bit different role. USB Hub drivers are responsible for recognizing the attach and detach events on their ports and inform the bus driver that something has changed in the USB topology. Hub drivers may be implemented as a normal client driver or within the USB driver.

4.2.2 The USB Driver Interface (USBDI)

The next layer is the USB driver with the USB driver interface (USBDI). This layer provides services for device configuration, transfer management, event notification as well as status reporting and error recovery. In fact, the USB driver manages the entire bus. USB client drivers use the exported USB driver interface to issue new USB transfers and manage the device. The USBDI can be divided into two different interfaces:

Pipe Interface This interface provides methods for managing pipe state such as opening, closing and starting pipes. This involve setting up the corresponding data structures. The setup of the data structure involves allocating memory resources and therefore the pipe interface is on the performance critical path.

Command Interface This interface provides management services that allows the client driver to configure its device. The device driver can execute device requests and change the power state of the device with the management services.

4.2.3 The Host Controller Driver Interface (HCDI)

The main part of this layer is to abstract the underlying host controller hardware and provide a host controller independent interface that is accessed by the USB driver. The HCDI has to implement the following abstractions:

- Host Controller hardware: abstracts the different host controller implementations and provides an unified interface
- Data transfers: abstracts the data flow across the USB interconnect i.e. status reporting of the active USB transfers
- Resources: Abstracts the allocation and de-allocation of the resources such as USB bandwidth for periodic transfers
- root hub: emulates the necessary functionality of a hub device which acts as the root hub

How the abstractions are implemented and how the exported interface looks like is up to the host software system. There is only one client that uses the HCDI and this is the USB Driver. This implies that no USB client driver directly accesses the host controller driver interface. This ensures that the USB driver is always aware of what is going on.

4.2.4 The Host Controller Interface (HCI)

Each host controller is a piece of hardware which implements one of the four different interfaces listed in Section 4.1.1. The USB revision and vendor define which HCI is used[4]. The different host controller interface implementations differ in the number and representation of the hardware registers. In this thesis, we focus on the Enhanced Host Controller Interface (EHCI).

The Enhanced Host Controller Interface (EHCI)

With the revision 2.0 of the USB specification a new register level interface was designed called Enhanced Host Controller Interface (EHCI). The EHCI Specification [13] defines the registers layout and data structures as well as the operational model for this particular host controller interface. The EHCI with USB revision 2.0 introduced high-speed (HS) capable devices supporting up to 480 Mbps.

Companion Controllers One has to be aware of the fact that the EHCI controllers only support HS devices connected directly to the root hub ports. There are two options for operating full-/low-speed (FS/LS) devices with a EHCI controller:

[4]This plays a role for USB revision 1.x compliant host controllers

1. Using a companion controller (OHCI or UHCI)

2. Using a HS hub with transaction translator in between

When the FS/LS device is connected to an HS hub, the root hub only sees the HS hub devices attached to its port. The HS hub then translates the USB transactions for the FS/LS devices.

When the host controller driver (HCD) recognizes that the attached device is not HS capable it releases the ownership of that port and hands it over to the companion host controller (cHC). The cHC is either an OHCI or UHCI controller and is now responsible for this port. The particular cHC type depends on the vendor of the EHCI controller[5]. Further a EHCI controller may have multiple virtual companion host controllers.

EHCI Register Interface The EHCI register interface can be split up into three parts and may have additional, vendor specific registers as it is the case on the PandaBoard. The overall registers size also depends on the number of root hub ports.

1. PCI Configuration Registers

2. Host Controller Capability Registers

3. Host Controller Operational Registers

It is important to emphasize that the EHCI hardware registers not form a contiguous memory region. Each of the three parts may be located somewhere else. However, location of the operational registers are not completely independent of the capability registers because the `CAPLENGTH` specify there the operational registers start relative to the capability register base.

4.3 Human Interface Device (HID) Class

As mentioned above, every USB device falls into one device class with its corresponding subclasses and protocols[6]. The human interface device (HID) class [6] contains generic definitions and operational models for (mostly[7]) input devices such as keyboard and mouse.

The good thing about the device classes is the fact that every device of a certain class should obey the class specification. For instance, the HID device class defines the minimum requirements that every HID device should provide. This enables to have universal drivers e.g. an USB keyboard driver works for every USB keyboard which follows the HID specification no matter how many fancy buttons it has. The basic functionality stays the same.

The HID class basically distinguishes two different protocols that can be used: keyboard and mouse (or none). All HID devices are either low-speed devices or high-speed devices the latter is more of an exception.

[5]The cHC implements the UHCI if it is produced by Intel or VIA and implements OHCI otherwise.

[6]If there is a special device, it may fall into the "Vendor Specific" class.

[7]force feedback of joysticks and LED indicators may be viewed as output

4.3.1 Accessing the Device

Every HID class device must implement two different pipes (Section 4.4). First the default control pipe that is used for configuration of the device and secondly an interrupt pipe that transfers information about the events (e.g. key-down / key-up) to the client driver.

4.3.2 Class Specific Details

Since the HID class deals with human input, there are special definitions concerning the different type / sources of input. HID devices use a special data format which is called a report. A report can be composed of different items and collections and can be parsed by the USB client driver to extract the related information. Referring to the HID specification [6] for a complete list of class specific details. The following three paragraphs emphasize some specialties.

Country Codes The HID reports may contain location information of the device represented as a country code. This information can be used to load a language dependent driver setting for instance the Japanese keyboard layout when the Japanese country code was read.

Units For sensor devices which are also belonging to the HID class it is important to know the unit of the absolute values in the report. The unit item of the report defines how the value should be interpreted e.g. a temperature as Kelvin or Fahrenheit. The HID specification has a set of predefined units.

Physical Information In order to distinguish what triggered the input event, physical descriptors contain an information qualifier and a designator value of the event e.g. the left hand. That way multiple sensors of the same type can be attached and the driver can distinguish if the event happened at the left or the right hand.

4.4 Transfer and Endpoint Types

The dataflow between host and the USB device is abstracted through so-called pipes. Each pipe has an associated endpoint (Section 4.1.4) and may be unidirectional or bi-directional. Pipes support either message based communication or stream based communication depending on the transfer type. There are four different USB transfer types each of which has its own characteristics.

4.4.1 USB Transfer Types

Control Transfers Transfers of type control are message based and are used to control the device e.g. set a new device address or read descriptors. The messages over the control pipe have a clear defined format and may contain a data stage that can go in either direction. Every device must provide a control endpoint which is referred to as the default control pipe or endpoint zero.

Interrupt Data Transfers Transfers of type interrupt are of event based nature and have stream characteristics. Interrupt transfer have a maximum payload. Once the transfer is started it is polled every x ms. If an event happens the device initiate the transfer and finishes the transfer. Otherwise the transfer stays in unfinished state.

Isochronus Data Transfers Transfers of type isochronus are stream based and are used to transfer data on a regular basis. The bandwidth for the isochronus transfers are pre-negotiated in advance to ensure that the data can be transferred within the required latency. As an example for this one may consider a video camera that transfers a video frame every 40ms.

Bulk Data Transfers Transfers of type bulk data are stream based and can transfer relatively large amounts of data from or to the device. Bulk data transfers have no guaranteed bandwidth and use the remaining i.e. the one that is not occupied by interrupt or isochronus transfers.

Chapter 5

Pandaboard

This part of the documentation first gives an overview of the hardware related issues using the PandaBoard [18] and secondly how they are solved in the PandaBoard specific implementation of Barrelfish. The content of this section briefly describes on a high level point of view the hardware initialization steps. For detailed register set up one may consider the OMAPP44xx SoC TRM [27] and the PandaBoard reference manual [18].

5.1 The PandaBoard

The PandaBoard is an ARMv7 based development board using the Texas Instruments OMAP44xx SoC [27]. The detailed hardware specifications can be obtained from the PandaBoard project website[1].

First, consider the x86 architecture where the processor die contains the central processing unit (CPU) and possibly a memory controller. The other devices such as the PCI bus reside on the main board. In contrast to that, SoC based architectures have every device already integrated in the chip die. In SoC designs such as the OMAP44xx the devices in the chip are connected to receivers on the board via connector pins.

These connectors pins are shared and have to be muxed with the correct `DPAD` configurations in order to connect the wires correctly. These input / output pins can either be connected to an integrated device in the SoC e.g. the USB host subsystem or driven manually by the general purpose input output (GPIO) registers. In order to make USB work on the PandaBoard both ways are needed.

5.2 The USB Subsystem

The Pandaboard's USB subsystem can be divided into two major parts. On the one hand there are the devices which are integrated in the OMAP44xx SoC and on the other hand the board itself which provides the corresponding endpoints and receivers to the devices in the SoC.

[1] `http://www.pandaboard.org`

Even the PandaBoard looks quite simple at a first glance, getting the USB subsystem running involves quite an amount of work and the particular steps have t be executed in the right order. We will first have a look at the SoC related setup and then on the board specific initialization sequence.

5.2.1 USB Host Subsystem in the OMAP44xx SoC

The OMAPP44xx technical reference manual [27] describes the three different USB subsystems of the OMAP44xx each of them serves different purposes. For enabling USB host functionality only the first of them is of interest.

- High-Speed Multiport USB Host Subsystem
- High-Speed USB OTG Controller
- Full-Speed USB Host Controller

In contrast to the high-speed multiport USB host subsystem, the OTG (on-the-go) controller is a device mode controller which is connected to the mini USB port of the board. This port is used to provide power and boot Barrelfish on the PandaBoard via usbboot [16]. The full-speed USB host controller is only connected to the finger print reader and hence not to the USB ports on the board.

The following section describes the building blocks of the HS USB host subsystem and how initialize it.

5.2.2 High-Speed Multiport USB Host Subsystem

The HS USB host subsystem is composed of several modules as shown in Figure 5.1. It is important to keep in mind, that there are many other modules which are related to the USB subsystem and have also to be initialized because of the dependencies.

HS USB Host Controller

This module is the heart of the USB subsystem and contains two different host controller implementations. The EHCI controller for high-speed devices and an OHCI controller for full-/low-speed devices. The OHCI controller can be viewed as the companion controller. However, since there is only a HS USB hub connected to one of the root hub ports the OHCI controller can be left aside. The root hub has tree ports, two of which are connected to the channels.

USBTLL and Channels

The USBTLL module on the OMAP44xx SoC can be used to translate the signal from the host controllers to various other encodings for full-/low-speed modes to match the receiving modules on the board.

There are two channels connected to the ports of the root hub. Each channel is owned by exactly one of the two host controllers and therefore provide either high-speed or full-/low-speed support[2].

[2]On full-/low-speed mode the channel is connected to a serial controller (TTL) or a serial PHY

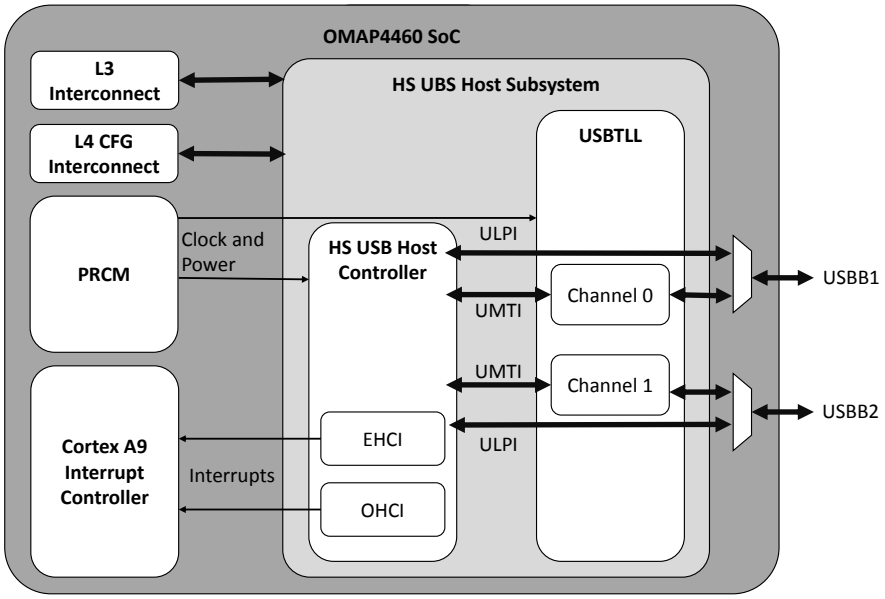

Figure 5.1: OMAP44xx HS USB Subsystem

Each of the two channels can be configured independently. For using the EHCI controller with high-speed USB access, the USBTLL module is bypassed and the channel mode is left in its default state (UMTI-to-UPLI controller). In general, the non-ULPI channel modes differ in the number and configuration of the serial lines.

If a channel is connected to the EHCI controller it has to be configured to use the ULPI mode and hence bypasses the USBTLL module entirely. When the OHCI controller is used, the channel needs to be in one of the different serial modes and the USBTLL module is used. For the PandaBoard only channel 0 is relevant.

Errata According to the OMAP44xx Silicon Errata [26] there is a bug in this part of the HS USB subsystem. The problem is the following: Assume a channel is configured to use the ULPI interface and provides high-speed functionality. Then that channel cannot be reconfigured to use the UMTI mode and full-/low-speed functionality anymore i.e. it cannot be handed over to the OHCI controller.

This implies that if there is a high-speed device connected to a channel and after that a full speed device is connected, the channel fails to operate [3]. As a workaround, one can use a HS hub with transaction translator to connect full-speed USB devices or to configure the channel as full-speed in first place.

[3] A signal will stay high forever

Register	Setting
CM_SYS_CLKSEL	Set the system clock to 38.4 MHz (0x7)
CM_L4PER_CLKSTCTRL	L4PER interconnect clock forced wakeup
CM_L3INIT_HSUSBHOST_CLKCTRL	enable the HS USB host module with all functional clocks
CM_L3INIT_HSUSBTLL_CLKCTRL	enable the USB TLL module and the optional channel 0 clock.
CM_L3INIT_FSUSB_CLKCTRL	explicitly enable the FS USB module (0x2)
CM_L3INIT_USBPHY_CLKCTRL	Enable the USBPHY and enable the optional functional clock.
SCRM_AUXCLK3	enable (bit 8) and set divider to 2 (bit 16)

Table 5.1: Clock and Power Settings

Initializing the HS USB Host Subsystem

To enable a module it basically needs power and a clock. The HS USB host subsystem module is not supplied with a clock and power by default. Therefore we need to set the correct values into the power, reset and clock management (PRCM) registers.

Table 5.1 on page 27 shows an overview of the registers and a description of the settings to apply. Currently is done at the platform dependent kernel initialization stage of the boot process (refer to Section 5.3.1 for more details). Since, the module initialization is in fact the work of a PRCM driver, it is supposed to be moved into a SoC driver (refer to the section about future work in the end of the documentation)

Clock Requirements It has to be mentioned, that simply applying a clock to the USB subsystem may not work. The PandaBoard ES manual [18] states that the clock has to be exactly 19.2 MHz. To achieve this, the system clock is set to 38.4 MHz and then setting the divider for the reference clock to 2 giving the required 19.2 MHz.

Module Initialization Sequence As soon as the module has clock and power, the hardware can be accessed and configured. The OMAP44xx TRM enumerates the sequence of operations to enable and reset the modules of the HS USB host subsystem inside the OMAP44xx. The following enumeration summarizes the steps.

1. Reset the USBTLL Module: `USBTLL_SYSCONFIG = 0x2`
2. Wait until the reset is done.
3. Set up USBTLL Module by writing `USBTLL_SYSCONFIG` register
4. Enable the USBTLL interrupts

5. Reset the host controller module: `UUH_SYSCONFIG = 0x1`
6. Wait until the reset is done.
7. Set the host controller module features by writing to the `UUH_SYSCONFIG` register
8. Set the host configuration to use the external PHY by writing to the `UUH_HOSTCONFG` register

Connection with the board

The steps in the previous section only enable the USB submodule hardware inside the OMAPP44xx SoC. However, the wires of the channel are ending up in a dead end and will not reach the IO pins of the SoC and hence there is no connection to the board hardware. By activating the correct configuration of the `DPAD` registers, the signals will be propagated to the IO pins and further be received by the corresponding hardware on the board. Table 5.2 shows an overview over all the needed `DPAD` configuration settings with the corresponding mux mode.

Mux Register	**Muxmode**	**IO Settings**
USBB1_ULPITLL_CLK	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DIR	4 (usbb1)	input + pull-down
USBB1_ULPITLL_STP	4 (usbb1)	output
USBB1_ULPITLL_NXT	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT0	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT1	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT2	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT3	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT4	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT5	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT6	4 (usbb1)	input + pull-down
USBB1_ULPITLL_DAT7	4 (usbb1)	input + pull-down
FREF_CLK3_OUT	0 (fref clk 3)	output
KPD_COL2	3 (GPIO_1)	output
GPMC_WAIT1	3 (GPIO_62)	output

Table 5.2: Settings of the CONTROL_CORE_PADs

5.2.3 USB system on the board

The USB system on the PandaBoard consists of two parts as shown in Figure 5.2. First the ULPI receiver (U7) which connects to the IO pins of the OMAP44xx. The purpose of the receiver is to transform the ULPI signal from the OMAP44xx into the USB signal format needed by the USB hub (U8). The ULPI receiver is connected to the upstream port of the USB/Ethernet hub (U8).

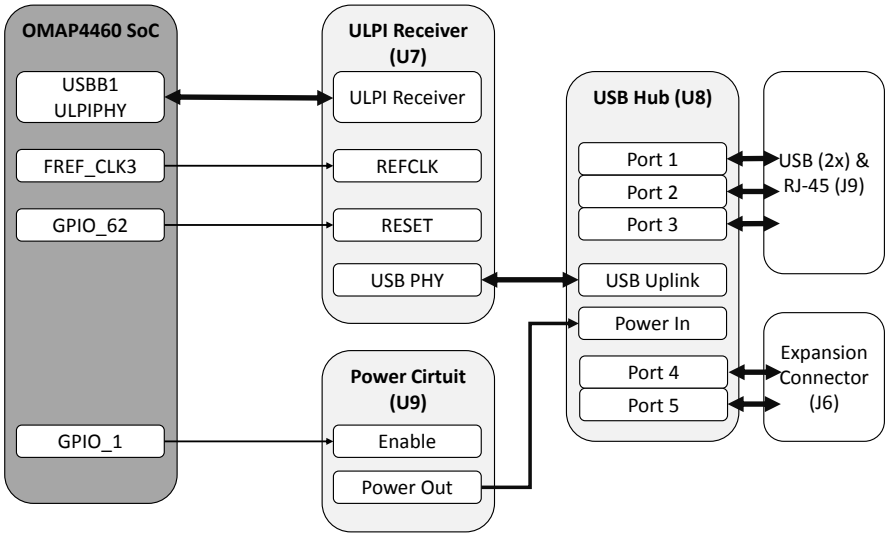

Figure 5.2: Pandaboard USB System

The PandaBoard ES Input Power Circuitry

The USB specification allows devices to consume up to up to 500mA from the bus. With multiple ports on a hub, this sums up and therefore the power supplied by the USB OTG port is not enough because the same rule applies here again: USB delivers only 500mA. This implies that the USB hub needs an additional power source.

The HS USB hub gets its power by an additional power circuit which is connected to an external power supply. However, just plugging the cable into the 5V connector of the PandaBoard does not provide power to the USB/Ethernet hub. To forward the power to the hub, the circuitry has to be enabled explicitly i.e. driving the `GPIO_1` pin to high in this case.

System overview

The big picture of the USB subsystem is shown in Figure 5.2. The PandaBoard has four different main modules that are related to the USB subsystem:

1. the OMAPP44xx SoC which the USB host controllers
2. the ULPI receiver (U7) which connects to the SoC.
3. a Power circuit (U9) which provides power to the hub
4. the high speed USB hub with 5 ports.

Not all ports of the USB hub can be used freely. One of them is connected to a Ethernet device and hence is already occupied. The next two of ports are

wired to the physical ports (J9) and are usable. The last two are only usable, when an expansion connector is present on the board (J6).

Initialization of the USB hub

The initialization sequence for the USB hub on the PandaBoard involves several steps which have to be executed in the correct order. Otherwise the hub may not be reset correctly and fails to operate.

As mentioned before, there are specific requirements for the clock and hence it is one of the most crucial steps to provide the ULPI transceiver with the correct clock. This is done by setting the `SYSCLK` to 38.4 MHz and derive the `AUXCLK3` from it divided by 2. The `AUXCLK3` is then forwarded to the `FREF_CLK3` which is muxed to one of the IO pins and in the end connected to the ULIP transceiver.

Initialization Sequence As soon as all the power, mux and clock configuration of the devices inside the OMAP44xx SoC is completed, the USB devices on the board itself can be reset and enabled. The following list shows the steps to be performed:

1. Reset the USB hub: drive `GPIO_1` to low
2. Reset the USB PHY: drive `GPIO_62` to low
3. Give the hardware time to reset
4. Initialze the HSUSB Host System of the OMAP44xx
5. Enable the power to the USB hub again: drive `GPIO_1` to high
6. Enable the USB hub again: drive `GPIO_62` to high
7. Perform a soft reset of the ULPI PHY through the ULPI interface

One may have noticed that the initialization of the HS USB subsystem inside the OMAP44xx is encapsulated in the board's initialization sequence.

The last step will perform a reset of the ULPI PHY between the OMAP44xx and the ULPI receiver using the ULPI protocol. The following section shows an example how to execute ULPI commands.

5.2.4 Verifying the Setup using the ULPI interface

The EHCI controller on the OMAP44xx contains a additional registers that can be used for debugging and configuring purposes of the ULPI transceiver. Table 5.3 shows the layout of the ULPI register of the EHCI controller[4]. To read or write something from the external ULPI receiver on the board involves some steps that have to be done. Listing 5.1 shows an example how the vendor ID can be read[5]. The use of Mackerel for this is possible here, but plain register accesses simplify the example here. For a Mackerel version one may consider the initialization sequence in the code.

[4] the ports correspond to the two channels.

[5] The address of the vendor register depends on the receiver

Bits	Field Name	Description
31	CONTROL	Control/Status of the ULPI register access
30:28	RESERVED	-
27:24	PORTSEL	port selector (1 or 2)
23:22	OPSEL	register access is write or read
21:16	REGADDD	direct ULPI register address
15:8	EXTREGADD	extended register addrss
7:0	RDWRDATA	data field

Table 5.3: EHCI INSNREG05_ULPI Register

```
volatile uint32_t* ulpi_reg = (uint32_t *)EHCI_ULPI_REG;

/* initiate a new ULPI access */
*(ulpi_reg) = (ULPI_REG_VENDOR | ULPI_OP_READ |
                ULPI_PORT_1 | ULPI_CTRL_START);

/* wait till the ULPI access is finished */
while (*(ulpi_reg) & ULPI_CTRL_START) {
        /* no-op */
}

/* read out the vendor id */
uint8_t vendorid = *(ulpi_reg) & 0xFF;
```

Listing 5.1: Reading out the ULPI Vendor id

The supported access modes and registers are specified in the manual of the respective ULPI transceiver. For the PandaBoard this is the SMSC USB33200 [22].

5.3 Barrelfish on PandaBoard

As already mentioned before, Barrelfish on the ARM architecture has not the same functionality as on the x86 architecture. For the development of user-level device drivers, the missing implementations in the ARM version of Barrelfish had to be added. This involved changes in the kernel as well as special domains such as init and monitor.

There are tree main points that have to be implemented:

- Hardware initialization (enabling of clocks and power)
- Obtaining the device capability
- Interrupt handling

This section will briefly discuss the missing functionality and how the needed functionality is added to the system.

5.3.1 HS USB Host Subsystem Initialization

On x86 the devices are located on the PCI bus in general. Therefore the PCI domain takes over the initial configuration. On ARM this is not the case and the SoC sub modules need to be initialized otherwise.

The steps needed to initialize the HS USB Host Subsystem as shown in the previous section, are currently performed in the kernel during system boot. The code is executed in the architecture dependent part of the kernel before the MMU is getting enabled.

However, it has to be emphasized that initializing hardware should not be part of the kernel. Therefore, that part will be subject to change soon. The hardware dependent initialization sequences are to be moved out of the kernel into a chip dependent SoC driver. The other driver domains will only be started when the SoC driver is done with initializing the hardware. Figure 5.3 shows how the startup procedure may will look like in the future. How the SoC driver is started and its responsibilities would fall out of the scope of this thesis and hence only one idea is given.

In any case, to guarantee a platform independent and reusable device driver, the device driver domains should not deal with platform specific initialization tasks such as the HS USB subsystem initialization and therefore expect the hardware to be in accessible state.

5.3.2 Getting the Device Capability

As explained before, on the x86 architecture the PCI domain gets the capability of the whole PCI address space and hands over the capabilities of suitable range to the device driver domains upon request. This is not possible on the ARM architecture, because there is no PCI bus on the SoC and hence no PCI domain that has the capability for all devices.

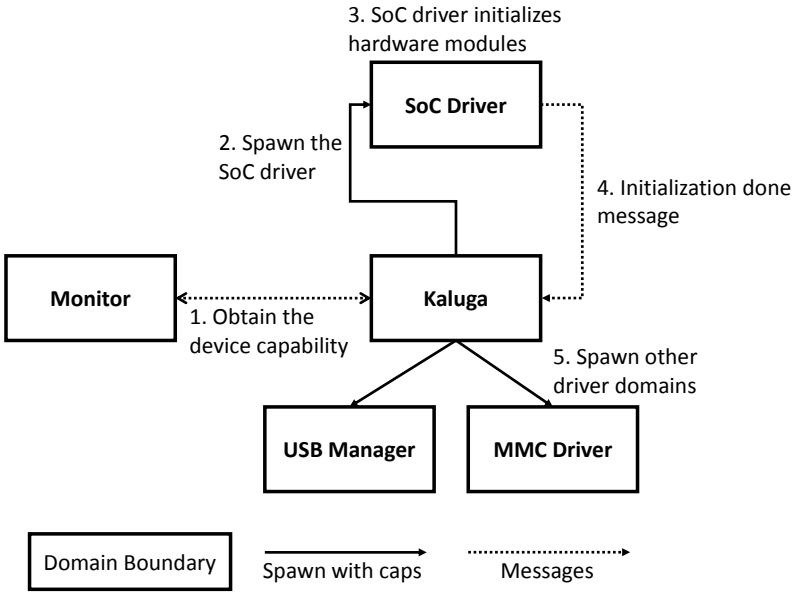

Figure 5.3: Barrelfish Startup with SoC Driver (Planned)

The capability space (`CSPACE`) is designed in such a way, that there are some pre-defined slots where special can capabilities are stored. Without an PCI bus, the ARM architecture does not make use of the `IO` slot in the `TASKCN`. This unused slot can be used to store a capability spanning the whole device range of the OMAP44xx[6]. This involves modifications in the kernel, init and monitor domains.

Kernel Modifications

Creating new capabilities can only be done by the kernel. Listing 5.2 shows the two additional code lines in the platform specific kernel startup. One has to notice that the address ranges may differ between the two SoC designs. In case the SoC has more than one core, this code has to be executed at least on the core where Kaluga is running in the end. For the PandaBoard, the created capability is of type `DevFrame` and spans the whole device range from 1GB to 2GB.

Init Modifications

The kernel places the capability in the `CSPACE` of the init domain. Init is responsible for spawning system critical OS services such as the monitor and memserver. The monitor is a privileged user-level domain which is responsible for handling privileged operations. The init domain needed to be slightly mod-

[6]It's a point of discussion whether using a new slot instead of reusing the existing one is the better design choice

```
struct cte *iocap = caps_locate_slot(
                        CNODE(spawn_state.taskcn),
                        TASKCN_SLOT_IO);
errval_t err = caps_create_new(ObjType_DevFrame,
                        0x40000000, 30, 30, iocap);
```

Listing 5.2: Kernel Modification for Adding the Device Range Capability

```
/* variable definitions */
struct monitor_blocking_rpc_client *cl;
errval_t err;

/* get the monitor binding */
cl = get_monitor_blocking_rpc_client();

/* variables storing the return values */
struct capref requested_caps;
errval_t e_code;

/* invoke the monitor service */
err = cl->vtbl.get_io_cap(cl, &requested_caps, &e_code);
```

Listing 5.3: Requesting the Capability from Monitor

ified, that it copies the `IO` capability in to the very same slot in the `CSPACE` of the monitor domain.

Obtaining the Capability

The monitor domain has now the needed capability in its `CSPACE` (`IO` slot of the `TASKCN`). The monitor implements the `monitor_blocking` interface which already has a function to obtain the IO capability. Listing 5.3 shows how to obtain it.

It has to be said, that device drivers should not request the capability from monitor by their own. Instead they should be spawned with only a capability spanning the device range. This is actually the way, how the USB subsystem is spawned. (Section 6.2 and limitations below)

Limitations

The current implementation allows every domain to obtain the full device capability by invoking monitor. This may be a safety critical: there should be just one process that is allowed to obtain this capability in order to prevent that two domains are using the same device range concurrently. In general only Kaluga i.e. the device manager should be able to obtain the device capability exactly

Host Controller	**x86 / x64**	**ARM**
VFS initialization	Executed	Executed
Environment initialization	Executed	Executed
SKB Client connect	Executed	Skipped
SKB execute	Executed	Skipped
Octopus initialization	Executed	Executed
Boot modules initialization	Executed	Executed
Octopus barrier enter	Executed	Skipped
USB Manager start	Skipped	Executed
Watch for Cores	Executed	Skipped
Watch for PCI Root Bridge	Executed	Skipped
Watch for PCI Devices	Executed	Skipped
THCFinish	Executed	Executed

Table 5.4: Overview of Steps Executed by Kaluga

once. Kaluga then starts the device drivers with a capability that spans only the needed sub range.

5.3.3 Driver Startup

On Barrelfish, Kaluga is responsible for starting up driver domains. So far, Kaluga was not running on the OMAP44xx because it performed operations that were not compatible with the OMAP44xx SoC. Most of them are not needed anyway since the OMAP44xx does not have a PCI bus and the number of cores is static[7] and known in advance.

Kaluga Modifications

On the OMAP44xx, the incompatible steps are skipped and Kaluga is now working on ARM. Table 5.4 shows the different steps and if they are executed or skipped.

It has to be said, that the Kaluga API is currently subject to change and the startup of the device drivers will be handled slightly different by Kaluga. However, the high level steps that are executed when a device driver is started are expected to stay more or less the same. The following list resembles the startup process:

1. Request the device capability from monitor service (once)
2. Look up the driver modules
3. Split up the capability into smaller chunks
4. Marshall the command line arguments
5. Spawn the driver domain with the capability.

[7] x86 is not static in the sense that you may have an x86 machine with 2 and one with 4 cores. In contrast, a specific SoC design has always the same cores

The command line arguments for the USB Manager form a triple containing the host controller type, the offset from the beginning of the capability and the interrupt number. Currently these values are hard coded.

The device capability is stored in the `ARGCN` slot[8]. That way, each driver domain is started with the needed capability and therefore does not need to request it afterwards. The driver domain knows exactly where to look for the capability and can map it in its own virtual address space. This not only simplifies the initialization but also increases the security since the device manager is aware of the spawned driver domains with the supplied capabilities i.e. no two domains can request the same capability.

5.3.4 Interrupt handling

So far there was not support for registering interrupt handlers on ARM and the only interrupt that could be enabled was the timer interrupt. Interrupts are very important for driver design, because polling a register costs CPU cycles those can be used in a more useful way. In this section, we will first have a look on how it is done on x86 and then how this was adapted to work on ARM.

Interrupts on x86 On x86, when a new interrupt handler is registered, a new interrupt vector is allocated and returned to the caller. These vectors are allocated one after another. The caller then associates its own device interrupt with the allocated vector by setting up interrupt routing entry in the APIC [4]. This way of interrupt routing cannot be adapted to the OMAP44xx platform, since there is no APIC that supports routing of the interrupts.

Interrupts on ARM On the OMAP44xx SoC, each device already has its interrupt number assigned. Hence, to setup a handler for a device interrupt, the particular interrupt number must be activated explicitly. This implies that instead of returning an interrupt vector, the interrupt number must be supplied to the setup procedure.

Kernel Modifications

Every interrupt in Barrelfish is transformed into a message by the kernel. So far, in the ARM version of the kernel the code to setup and handle the interrupts has not been implemented.

The capability invocation handling of the ARM kernel was extended with the functionality to deal with the capability for the interrupt table. Basically, the x86 version could be adapted to work on ARM without much changes: The handler function in the kernel stores interrupt number together with the supplied local message passing (LMP) endpoint in the interrupt table. When an interrupt occurs, the kernel does a lookup to get the endpoint in the table and sends a message to it. The driver domain will eventually receive the message.

[8]With the API change, the ARGCN slot will contain a reference to another CNODE containing all the capabilities needed instead of the just one

```
/* Allocate an IRQ on the arm plattform */
rpc arm_irq_handle(in cap ep, in uint32 irq, out errval err);
```

Listing 5.4: Added Function to the Monitor Blocking Interface

Init Modifications

Barrelfish has a special capability for accessing the interrupt table. The init domain was adapted to pass this capability to the monitor domain upon spawn. can be done exactly the same way it is done on x86.

Monitor Modifications

So far, the changes that have been made do not differ fundamentally from their respective x86 counterparts. This does not hold for the kernel monitor. To allow activating a specific interrupt, the monitor blocking interface had to be extended by an additional remote procedure call. Listing 5.4 shows the signature of the new function. In addition to that, also the message handler functions for the new RPC function needed to be created.

The way Monitor handles the new RPC call is similar to the existing one: It uses the IRQ capability it got from init and passes over the endpoint and the interrupt number to the kernel. However, monitor does not allocate a new interrupt number, but uses the supplied `irq` parameter as the interrupt number. Hence the caller does not get a vector back.

Barrelfish Library Modifications

In addition to the added RPC function in the monitor interface the corresponding library functionality also had to be added. The library function abstracts the whole message passing steps behind a single function call.

The signature of the new function is shown in Listing 5.5. The new API function is very similar to the existing `inthandler_setup` function in terms of signature and semantics. The only difference is in the last argument, which is now a normal integer and not a pointer i.e. the argument is used to pass the interrupt number instead of getting the vector back.

The future Implementation

As one can see, the way interrupts are implemented on the ARM architecture is as closely as possible to the x86 implementation. However the current implementation has a severe pitfall: Any process can initialize any interrupt number. This may end up in conflicts where two domains want to set up the very same interrupt number. The problem can be solved by introducing a new type of capability which represents a single interrupt number. This interrupt capability is then presented to the kernel upon calling the `inthandler_setup` function. Handling it that way also solves the problem that a driver domain has to know its particular interrupt number of the device. However, this is out of the scope of this thesis.

```
/**
 * \brief Setup an interrupt handler function to receive
 *              device interrupts on the ARM platform
 *
 * \param handler Handler function
 * \param handler_arg Argument passed to #handler
 * \param irq the IRQ number to activate
 */
errval_t inthandler_setup_arm(interrupt_handler_fn handler,
                              void *handler_arg, uint32_t irq);
```

Listing 5.5: Added Function to the Barrelfish Library

Chapter 6

System Design

The following chapter describes the architectural design of the USB subsystem software and gives some details about the implementation. The USB subsystem is based on the FreeBSD [8] implementation and was adapted to suite the distributed architecture of Barrelfish.

6.1 USB Subsystem Architecture

This section shows how the USB subsystem is composed from a high level point of view and briefly explains the responsibilities of the different modules. Figure 6.1 gives an overview of the system architecture. At the coarsest scale, the system is composed of at least two domains running purely user space:

- USB Manager, a single instance in the whole system
- USB client drivers, one particular instance for each connected device

Each of the domains contains different interfaces and sub modules. The following subsections give a brief explanation of them and outline their responsibilities. Later in this chapter a more detailed description for the most important aspects will be given.

6.1.1 USB Manager

The USB Manager implements the required functionality of the two lowest layers in the USB stack as shown in Figure 4.2 on page 19. As the name indicates, the USB Manager domain is responsible for managing the different USB requests, transfers and devices i.e. all the required USB host functionality.

The exported USB driver interface (USBDI) allows the USB client driver domains to start/stop the transfers and to execute requests on the device. Section 6.6 will give a detailed documentation of the interface. The USB Manager itself is structured in a layered way with the host controller driver at the bottom and the USB driver interface at the top. The layers basically correspond to the layers of the USB stack.

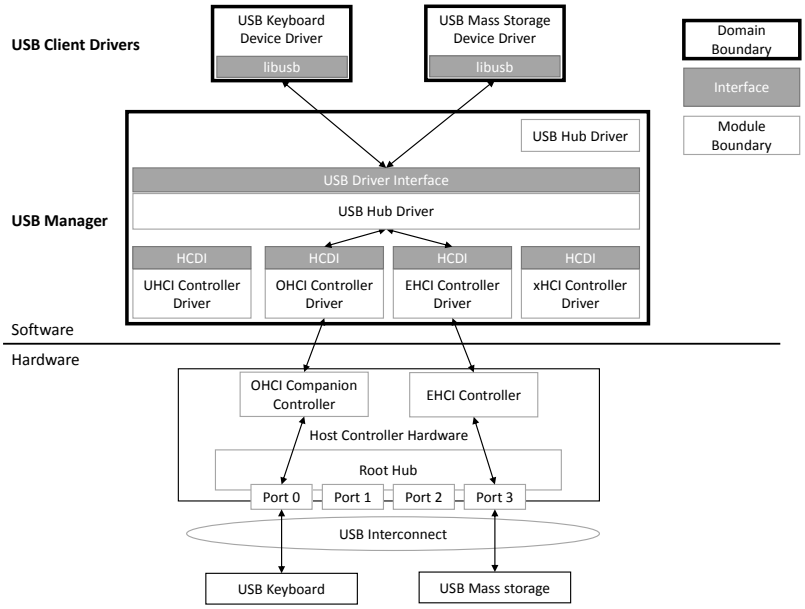

Figure 6.1: USB Subsystem Architecture

Host Controller Driver (HCD)

To ensure the functionality of the USB Manager, at least one USB host controller has to be configured before the USB Manager service is started. There exist four different modules at the lowest layer of the USB Manager stack. The modules handle the hardware specific details for each of the four different host controllers[1].

Each of the host controller modules implements the generic host controller driver interface (HCDI) which abstracts the different host controller hardware interfaces and operational models. The HCDI is only used by the USB driver, i.e. only within the USB Manager, and is never directly accessed from any USB client driver. Besides the initialization, the handling of interrupts has to be done host controller specific as well.

Device Management

Each host controller maintains a list of attached devices on its owned ports[2]. As soon as a new device is attached or a device is detached, the USB manager handles the (de-)allocation of the device resources as well as the dealing with the startup and shutdown of the corresponding USB client driver.

[1] Currently, there is only the EHCI controller implemented.

[2] With USB Rev. 2.0 each port of the root hub can be owned by either the EHCI controller or one of its companion controller.

USB Hub Driver

As already explained in Section 4.1.2, hub devices are special USB device. The USB Manager has an integrated USB hub driver that handles all the configuration and state management of the attached hub devices, hence there is never a separate domain for a hub device driver. Whether or not to separate the USB hub driver from the USB Manager may be a point of argument. Section 6.4 will show why it might be beneficial to treat the USB hub drivers special.

Transfer and Request Management

The USB Manager handles the different transfers and USB device requests issued by the USB client drivers or by the integrated hub driver. This involves managing the different transfer queues, checking for transfer completion and setting up the corresponding data structures for tracking the transfers. The exported HCDI is used to deal with the host controller specific details of the transfer management process.

6.1.2 USB Client Driver

Every USB client driver runs in a separate domain and uses the exported USB manager interface (Section 6.6) to communicate with its device. In general, the USB driver does not deal with the USB Manager interface directly but make use of the functionality provided by the USB library.

Each USB client driver exports the USB device driver interface (Section 6.8) to allow the USB Manager sending notifications when the device is detached or the transfer has completed.

In addition to that, every USB client driver exports the device specific interface to allow other domains to use the functionality of the device e.g. the keyboard interface. Device specific interfaces fall not into the scope of this thesis and are hence just mentioned here.

6.1.3 USB Library

Each domain in the USB subsystem, the USB Manager and the USB client drivers, links to the USB library. The USB library is used for two main purposes.

First, the USB library contains general data structure definitions such as descriptors and error codes as well as the related functions to manipulate them. It has to be emphasized, that some of the data structures exist also in an extended form in the USB Manager.

The second purpose of the USB library is to provide a simplified way to invoke the USB Manager service. The underlying details such as the connection state are hidden from the USB client drivers. By doing that, added sanity checks ensure correct data format, request validation and error handling. This will be given in more detail in Section 6.7.

6.2 USB Subsystem startup

The startup process of the whole USB subsystem involves several steps and the collaboration of different system domains. In general, the startup process can

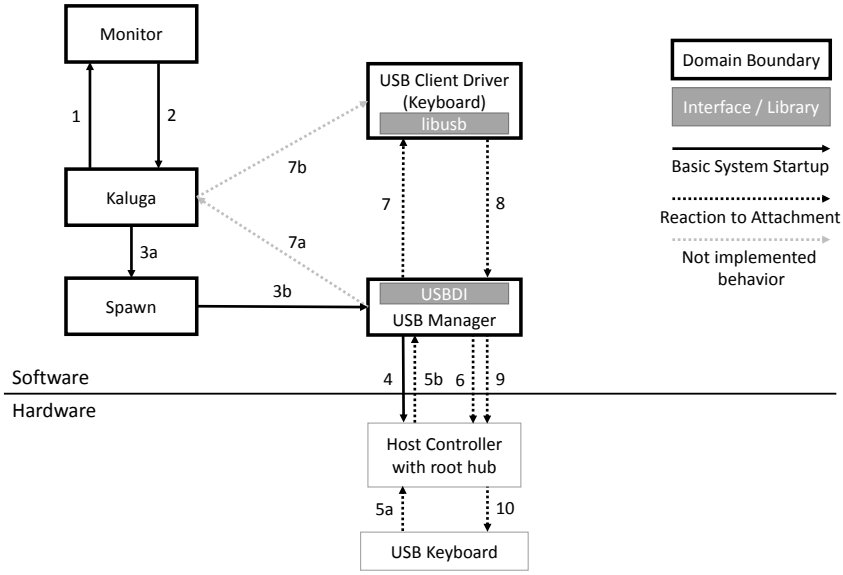

Figure 6.2: USB Subsystem startup

be divided into two parts which happen as a reaction to different events:

- USB Manager startup on system start
- USB client driver startup on a device-attach-event

Figure 6.2 shows the sequence scheme with the steps to be executed when the USB subsystem is spawned (solid arrows) and the reaction to a device attached event (dotted arrows). This section gives only a high level description of the reaction to the attach event. For further details how the attachment works refer to Section 6.4.

Initial System State

The initial situation for the figure is as follows: all the necessary OS services have been started and are ready to serve requests. The hardware is initialized and can be accessed i.e. the SoC specific initialization sequence was executed. However the host controller hardware has not yet been configured. Kaluga is running and is about to spawn the device driver domains.

6.2.1 Preparations to Spawn in Kaluga

As already explained in Section 3.4, Kaluga [31] is responsible for starting the device drivers in the Barrelfish operating system. During the initialization process, Kaluga parses the modules and finally reach the point where the USB Manager is about to be started. Currently, the startup routine is hard coded in Kaluga and will be subject to change to support a more general way for starting

device drivers on SoC[3]. Kaluga locates the device driver binary by looking it up in the modules and gets the related driver information. Next, the required arguments for the spawning procedure have to be compiled.

In the following sections, the numbers in brackets (x) denote the step indicated by an arrow in Figure 6.2.

Prepare the Device Capability

After the driver information of the USB Manager module are found, Kaluga requests the needed capability from the kernel monitor by invoking `get_io_cap()` (1) and gets it back as a return value from the RCP invocation (2). After this step Kaluga possesses a device frame capability of the whole 1GB device range of the OMAP44xx[4].

Next, Kaluga spits up the big capability into smaller chunks. Since we are dealing with memory ranges we can make use of the memory manager library (`libmm`) which will do the splitting of the capability.

Command Line Arguments

After the previous step has completed, the capability has now a range that is suitable for the USB Manager. The physical start address represented by the capability is known and the offset can be calculated[5].

The next step is compiling the command line arguments together. Table 6.1 shows how the USB Manager expects the command line arguments to be formatted. The number of expected arguments differ depending on which architecture the USB Manager is executed. The reason for the two different formats is the way interrupts are handled on the two architectures.

As already explained in Section 5.3.4, the OMAP44xx SoC does not have an advanced programmable interrupt controller (APIC) [4] and thus cannot route device interrupts to arbitrary interrupt lines to the processor. Hence the interrupt number of the device has to be supplied via an argument[6].

- The first argument tells the USB Manager which host controller is to be initialized. The values for the first argument are either `"ohci"`, `"uhci"`, `"ehci"` or `"xhci"`. Currently there is just the EHCI implementation available (referring to Section 7.2 for further details on limitations)
- The next parameter is the offset from the beginning of the supplied device capability to the location where the host controller registers start. Obviously this value should be as close to zero as possible, to avoid that a driver can write into other devices registers.
- The third parameter is architecture dependent and contains the corresponding interrupt number of the device. The device driver domain, here the USB Manager, parses the interrupt number and enables the interrupt with that number.

[3] See Section 7.3 on page 70 for a discussion about that topic

[4] It is also possible on other SoC, the kernel has to ensure that the correct range is covered and the capability is placed into the `IO` slot

[5] An offset of 0 would be preferable, but due to 4kb page constraints this is not always possible

[6] An alternative would be to pass a capability representing the interrupt

Architecture	Parameters	Example
ARM	[host controller] [offset] [IRQ]	ehci 0x1000 109
x86	[host controller] [offset]	ehci 0x0

Table 6.1: USB Manager Command Line Arguments

The parsing of the command line arguments by the USB Manager is designed in such a way that multiple host controllers can be started. Each host controller needs an argument tuple with 2 respectively 3 elements. Thus, by concatenating multiple such tuples one after another, the USB Manager will parse those and initializes a host controller for each tuple[7]. The current implementation just deals with the EHCI host controller type and ignores others.

6.2.2 Spawn

Kaluga has now everything ready to initiate the spawn of the USB Manager. The spawning of new driver domains is not done by Kaluga itself, but is deferred like any other domain to the spawn domain. The only difference is that the driver domain is spawned with the capability. The request is sent via a message (3a) containing the name of the driver binary as well as the command line arguments and the capability for the `ARGCN` slot. The spawn domain starts the USB manager (3b).

6.2.3 USB Manager

The first step of the USB Manager is checking the command line arguments for the correct count and mapping the capability in the `ARGCN` slot in its own virtual address space. If the USB Manager fails to map the device frame capability or there is a wrong number of supplied arguments, the domain exists with an error condition.

As soon as the device capability is mapped, the command line parameters are parsed. The Mackerel base address is set as the address of the mapped capability plus the offset from the command line arguments. Depending on the architecture, the specific interrupt is activated or a new interrupt vector is allocated and the interrupt routing is set up.

The USB Manager starts accessing the host controller hardware registers and executes the needed configuration steps depending on the host controller type (4). Section 6.3 shows the steps for the EHCI controller. After this step the host controller hardware is operational. During the configuration process, the first USB device is also initialized: The root hub. The root hub is always present and emulated by the host controller driver.

In the last step, the USB Manager exports its service to enable the client driver domains connecting to the USB manager (refer to Section 6.6 for the interface specification). The USB Manager is now ready to use and waits for new devices to be attached.

[7]The device manager has to ensure, that no controller is initialized twice

6.2.4 Device Attachment

This section gives a high level view of steps involved when a new device is attached to the USB. Section 6.4 will outline some specific points in more detail. A new device is attached (5a) on one of the root hub ports[8] and an interrupt (port change detected) is risen (5b) by the host controller hardware. The interrupt handler function of the USB Manager defers the handling of the interrupt to the host controller specific interrupt handler.

The host controller specific handler function reads the interrupt status register of the host controller hardware to determine the type of the interrupt. With a port change detected interrupt, the root hub ports need to be explored and the USB Manager initiates the exploration process.

There are two possibilities how the exploration process can end: either a device was removed or a device was connected[9]. We are interested in the latter case. The information from the exploration process on the root hub (6) tells the USB Manager to allocate a new device in its device list.

The newly allocated device gets its address and is configured with its initial configuration by the USB Manager. During the configuration process, the device information are read and parsed such that the USB client driver can be determined. The USB Manager performs a lookup for the client driver binary using the class and vendor information read from the device. A new domain is spawned (7) if a suitable binary exists.

Detour via Kaluga The way the USB Client drivers are spawned is a simplified way. The USB Manager should not be aware of the different client driver binaries available in the system and therefore pass the information about the new device to Kaluga (7a). Kaluga processes the information, keeps track of the devices and looks up the most suitable device driver binary. If there is one, Kaluga initiates the spawning process of the new driver domain (7b). This way, Kaluga is always aware of the current driver domains running in the system. However, this functionality is currently not implemented.

6.2.5 USB Client Driver

The first step that every USB client driver has to do is initializing the USB library. As part of the initialization sequence the USB library binds to the USB Manager (8). Section 6.7 will give more details about the initialization and connecting process. As soon as the connection to the USB Manager is established the USB client driver starts configuring the device by allocating the needed transfers and registering of the callback functions. The USB Manager translates the requests and transfers using the HCDI (9) to the host controller specific formats and sends notifications about the outcome of the transfers to the client driver. The transfers are executed on the device via the USB interconnect (10).

Capabilities USB client drivers differ from normal device drivers in the way that they do not need a device frame capability for accessing their device, since

[8] In general this can be any port of any hub. The steps on normal hubs differ to the root hub in terms of interrupt handling

[9] This can be distinguished by evaluating the port status, see Section 6.4.2

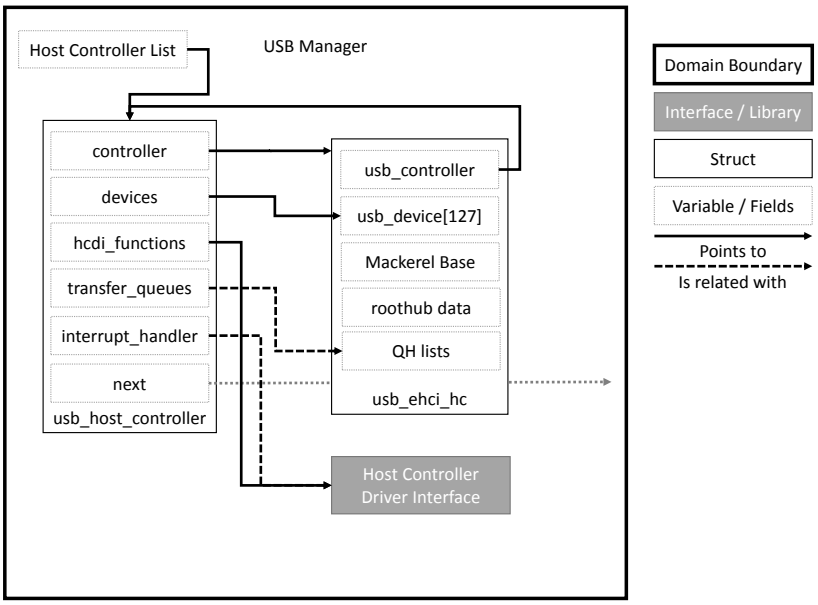

Figure 6.3: Host Controller Software Struct

USB function devices do not have any memory mapped registers and are accessed entirely through the USB interconnect. The USB host controller is the only device of the USB subsystem that has memory mapped registers. It is an open point of discussion whether or not each USB client driver should have a capability for its own device.

6.3 USB Host Controller

Each host controller is represented in the USB Manager with two different data structures.

- The generic USB host controller structure contains hardware independent data which is used at the USB driver level.
- The hardware dependent host controller structure contains all the hardware specific details which are abstracted by the generic part.

Figure 6.3 shows a simplified structure. The generic host controllers form a doubly linked list, since the USB Manager may be in charge of managing multiple host controllers, e.g. companion controllers, at once.

6.3.1 Initializing the Generic Part

The generic part contains pointers and values which are filled in during the initialization procedure of the specific host controller. The generic host controller

Offset	Size	Identifier	Description
00h	1	CAPLENGTH	Capability Register Length
01h	1	Reserved	N/A
02h	2	HCIVERSION	Interface Version Number
04h	4	HCSPARAMS	Structural Parameters
08h	4	HCCPARAMS	Capability Parameters
0Ch	8	HCSP-PORTROUTE	Companion Controller Port Route

Table 6.2: EHCI Capability Registers

contains a pointer to the hardware specific host controller as well as a field indicating the type of the controller. The transfer queues track the created USB transfers and are filled upon setting up new USB transfers.

6.3.2 Initializing the EHCI Controller

This section explains how the hardware dependent part is initialized by using the example of the EHCI controller. The data structure of the hardware specific host controller contains many hardware dependent fields such as the Mackerel base or the queue heads (QH) lists for the transfers. Since the host controller driver is required to emulate the root hub device, the related data structures and values for the root hub also have to be stored. One may notice, that the two controller structures are connected in a circular fashion.

Initialize the Mackerel Base

The register interface of the EHCI controller consists of three parts. First, there are the PCI related registers, which are not of interest here as they are used by the PCI domain. Next, there are the capability registers (Table 6.2) and the operational registers (table 6.3). This two registers sets can be arbitrary far apart from each other and therefore the Mackerel base initialization has to be done in tree steps.

1. Initialize the capability register base part of the Mackerel base
2. Read the `CAPLENGTH` register to get the offset
3. Initialize the operational register base part of the Mackerel base

After the initialization of the Mackerel base, both register regions are accessible via Mackerel and the configuration sequence of the EHCI controller can be started.

Host Controller Initialization

Section 4.1 of the EHCI specification [13] defines the steps that have to be done for a proper initialization of the host controller hardware. The following list summarizes how these steps are implemented in the initialization procedure.

1. Halting[10] the controller and reset the controller hardware

[10]It may be the case, that the hardware was used by the BIOS on system boot

Offset	Size	Identifier	Description
00h	4	USBCMD	USB Command
04h	4	USBSTS	USB Status
08h	4	USBINTR	USB Interrupt Enable
0Ch	4	FRINDEX	USB Frame Index
10h	4	CTRLDSSEGMENT	4G Selector (64 Bit Addressing)
14h	4	PERIODICLISTBASE	Frame List Base Address
18h	4	ASYNCLISTADDR	Next Asynchronus List Address
1C-3Fh	4	Reserved	-
40h	4	CONFIGFLAG	Configured Flag Register
44h	4	PORTSC(1-N_PORTS)	Port Status / Control

Table 6.3: EHCI Operational Registers

2. Allocate the initial queue heads for the transfer lists.
3. Initialize interrupt transfer queues each with a terminate queue head.
4. Link the queue heads that they form the 2ms intervals
5. Allocate and initialize the periodic frame list and store address in register
6. Initialize the asynchronous queue and store address in register
7. Allocate and initialize the root hub device
8. Set the `USBCMD` register values
9. Take over port ownership of all ports
10. Enable the interrupts in the `USBINTR` register
11. Enable power on the root hub ports

As soon as these steps are executed, the host controller is up and running. For a detailed sequence of instructions for the steps in the enumeration above, please refer to the code. Keep in mind that the USB Manager will have to deal with interrupts after this step completes thus the interrupt handler must be set beforehand.

6.4 Device Attachment Process

The previous section about subsystem startup outlined the steps to be done on a domain level view. This section gives some implementation specific details as well as some detailed steps to be performed inside the USB library and the USB manager.

6.4.1 New Device Attached Event

When a new device is attached to a hub port, an interrupt is risen. However, depending on the point of attachment i.e. directly at the root hub or on another hub, a different interrupt will happen and therefore the interrupt handling is a bit different:

1. root hub If a device is directly attached to one of the root hub ports then the port change detected bit of the `USBSTS` register is driven high. The host controller knows that something changed on the ports of the root hub and can initiate the exploration process.

2. Normal hub Devices that are connected to a hub other than the root hub, do not change anything on the ports of the root hub. Each hub - except the root hub - starts an interrupt transfer upon initialization. As soon as a new device is attached on one of the ports, the hub device executes the transfer and the host controller recognizes a transfer complete interrupt. The data of the interrupt transfer contains the hub status.

6.4.2 Exploring Hub Ports

As soon as the interrupt was fired and determined on which hub and port something has changed, the hub port can be explored to see what has happened. In the exploration process of a hub, the hub driver executes a `GET_STATUS` requests to obtain the state of each of a port. Listing 6.1 shows an example how to such a request is executed. The return value of the request is a four byte data structure containing two words: `wPortStatus` and `wPortChange` (see [19] for a detailed definition). To determine if there is a new device or not, two bits of interest:

- `C_PORT_CONNECTION`: This bit indicates whether there was a change on the connect status (bit = 1) of this port or not (bit = 0).
- `PORT_CONNECTION`: This bit indicates if there is something connected to that port (bit = 1) or not (bit = 0)

By a combination of the information from these two bits it is clear that a new device was attached on that port when there was a change in the current connect status (`C_PORT_CONNECTION` = 1) and the there is something connected to that port (`PORT_CONNECTION` = 1).

```
/* pointer to a hub device, must be a valid pointer */
struct usb_device *hub;

/* port to get the status, must be a valid port */
uint8_t port;

/* return value for the port status */
struct usb_hub_port_status ps;

/* execute the request */
err = usb_hub_get_port_status(hub, port, &ps);
```

Listing 6.1: Getting the Port Status of a Hub

6.4.3 Device Allocation and Initialization

As soon as there is a new device attached condition detected, we know all the needed information (especially which port and which hub it is attached) to allocate new device. During the allocation process, the device gets its initial configuration.

Reset To ensure that the device is in a known state it has to be reset first. Resetting an USB device is different than resetting a traditional hardware device. The device reset is not done on the device itself but rather on the hub port it is connected to the USB.

In order to do the actual reset, a `SET_FEATURE` request with the `PORT_RESET` feature selector is executed. The request is sent to the port of the hub where the device is located. It has to be mentioned that the device must be given enough time to complete the reset sequence (there is no register that can be polled).

USB Topology Updates Each device has its position in the USB topology which has to be kept up to date. This implies that each hub must know the child devices connected to its ports and every device must know its controller and parent hub. The USB device data structure stores the most important topology information such as

- The associated host controller
- The parent hub device (`NULL` for the root hub)
- The port number of the attachment port in the parent hub
- The depth of the device in the USB topology tree
- For FS / LS devices: the parent HS hub with the transaction translator.

These values are set when the device is allocated an can be obtained directly or by traversing the USB topology.

Addressing Every USB device needs a unique address on the USB. This involves keeping track of already used addresses and free addresses. Each host controller has to maintain a list of connected devices. The USB specification [19] defines the maximum number of devices to be 127 which is stored as a one byte number. The address 0 is treated special since new devices have address 0 by default. With that constant maximum device number, an array for the devices can be pre-allocated.

Therefore we can set up a one-to-one mapping between array index and device address and hence find an unused device address simply by going through the array and pick the first `NULL` entry (see Listing 6.2). After the addressing step, the device has its own unique address and the array entry is filled with the pointer to the new device.

```
/* pointer to the current host controller */
struct usb_host_controller *hc;

/* start at the root hub address */
uint8_t device_index = USB_root hub_ADDRESS;

/* loop over the devices */
while (device_index < hc->devices_max) {
   if (hc->devices[device_index] == NULL) {
      break;
   }
   device_index++;
}

/* allocate memory for the new device */
struct usb_device *device = malloc(sizeof(struct usb_device));

/* set the device address */
err = usb_req_set_address(device, device_index);

/* set the device entry */
hc->devices[device_index] = device;
```

Listing 6.2: Finding a free device address

Configuration After setting the address, the device is in the addressed state. To make a device usable, it has to be configured. Each device has at least one configuration and exactly one that is active. The actual number of available configurations is stored in the `bNumConfigurations` field of the device descriptor. The device descriptor contains many other device specific information and is shown in Table 6.4. To obtain it, a `GET_DESCRIPTOR` request has to be executed on the device.

The USB Manager will set the first configuration value by default[11]. As a result of the `SET_CONFIGURATION` request the device transitions to the configured state. Each change of configuration is tied with a change in the available interfaces and endpoints.

When the configuration value is set, the current configuration descriptor is read. The configuration descriptor is of variable size and contains all the interface and endpoint descriptors of this configuration. (See Figure 4.1 on page 15). While parsing the different descriptors of the configuration, the interface-endpoint tree of the device is set up.

6.4.4 Driver Startup

Once the device is set up with its initial configuration the device is ready to be used by the USB client driver. It is up to the client driver to change the

[11] In rare cases, this may cause problems and a quirk has to be applied. See Section 7.2.4

Offset	Field	Size	Value
0	bLength	1	Length of the Descriptor
1	bDescriptorType	1	Type DEVICE
2	bcdUSB	2	Version of the USB
4	bDeviceClass	1	Device Class
5	bDeviceSubClass	1	Device Subclass
6	bDeviceProtocol	1	Device Protocol
7	bMaxPacketSize0	1	Maximum Packet Size
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Version of the device
14	iManufacturer	1	Manufacturer string index
15	iProduct	1	Product string index
16	iSerialNumber	1	Serial number string index
17	bNumConfigurations	1	Number of configurations

Note: The string index determines which string descriptor stores the value.

Table 6.4: USB Device Descriptor

configuration to another value. The next step is to find a suitable client driver which can make use of the functionality provided by the device.

Like PCI devices, USB devices can be distinguished by their class codes, subclass codes and so on. These codes are stored either in the device descriptor (Table 6.4) or the interface descriptor depending on the actual device class. Table 6.5 shows the different class codes defined by the USB-IF [29] and where the needed class code information is stored (device or interface descriptor).

Example for an USB keyboard From Table 6.5 we see that we have to look up the information from the interface descriptor:

- device class: 0x03 (HID Class)
- device subclass: 0x01 (Boot Device)
- device protocol: 0x01 (Keyboard)
- vendor id: .. product dependent
- product id: .. product dependent

The USB Manager uses the information of the device descriptor to lookup a suitable device driver binary. In general, the class, subclass and protocol information is enough to find a suitable driver and a single driver can be used for devices from different vendors e.g. a Cherry or a Logitech keyboard. The vendor and product information can be used to load a specific driver to support additional features of the device such as special keys. If a suitable device driver is found, then the corresponding driver-to-device binding can be set up upon binding.

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface
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
10h	Interface	Audio/Video Devices
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous
FEh	Interface	Application Specific
FFh	Both	Vendor Specific

Table 6.5: USB Device Classes by USB-IF [29]

Device Configuration The client driver knows its device best and therefore may not agree with the USB Manager with respect to the initial device configuration. The library initialization takes a configuration value as argument. This value is used to change the configuration of the device to the supplied value upon client driver binding.

The configuration is only changed, if the client driver supplies a value other than `USB_CONFIGURATION_DEFAULT` as the configuration value to set. The device configuration can also be changed after the binding procedure is over by using the `SET_CONFIGURATION` request.

Hubs Hubs are a special type of USB devices and are treated separately. In contrast to normal function devices, hub drivers are not separate domains but are rather managed inside the USB manager. This simplifies the handling of attach and detach events. As soon as a hub device is detected the hub related data structures are initialized and the spawning of the driver domain aborted. It is a point of discussion whether or not to take out hub driver out of the USB Manager is beneficial. Clearly having the driver out of the USB Manager would introduce more inter domain communication traffic.

6.4.5 The USB Device Tree

Each attached USB device has its place in the USB device tree as shown in Figure 4.1 on page 15. With exception of the root hub, every other device has a parent hub i.e. a device where it is attached to the USB interconnect. Each

device is aware of its depth in the tree and knows its parent hub as well as the port it is attached to. Further each hub knows the devices that are attached to its ports.

It is important to keep the topology tree always consistent with the real hardware situation on the USB interconnect. Especially the depth information is important because there is a maximum depth constraint by the USB specification.

Configuration Tree Not only the devices form a tree, but also the endpoints and interfaces within a device configuration are organized like a tree. Each devices has a tree with the configuration descriptor as the root, interfaces as inner nodes and endpoints as leaves. This tree has to be updated whenever the configuration is changed.

Non High-Speed Devices Full-/Low-speed devices can be attached to the USB in two different ways. If they are attached directly to the root hub, the port is handed over to the companion controller which then takes the responsibility of managing the device as well as the transfers[12].

If the FS/LS device is attached to a high speed hub with transaction translator the HS hub does the translation from HS micro-frames to FS/LS frames. Transactions targeting a FS/LS device, are marked as full speed and requires two additional fields to be set. This fields are the address of the parent HS hub and the port number which the FS/LS device (sub-tree) is connected to. This is because the parent HS hub needs to be aware that the translation has to be applied.

This implies that not only the parent hubs but also the parent high-speed hub with the port has to be known for each FS/LS device. Upon device allocation, the topology tree needs to be walked up until a HS hub is found.

6.4.6 Driver-to-Device / Device-to-Driver Association

When a USB client driver starts, it initializes the USB library as shown in Figure 6.4. As explained in Section 6.2.4 during the initialization process, the USB client driver connects to the USB Manager. For future requests, the USB Manager needs a way to figure out on which USB device the request has to be executed. To ensure this, a Flounder binding - device association (and vice versa) is established. This is done using the following sequence of steps after the device class is known:

1. Look up device driver binary and store the path with the device
2. if there is a binary found, put the device on a pending list[13]
3. Dequeue the next device from the pending list
4. Update the currently processed device pointer
5. Spawn the client driver domain

[12] Remember: the EHCI controller does not support FS/LS devices.

[13] It may be possible, that more devices are identified before the driver connects to the manager

6. Wait until the client driver connects

7. Associate the binding with the currently processed device

Limitations It has to be clarified, that only one USB device driver is being spawned at any point of time and the next is spawned only when the connect procedure is over.

USB Capabilities As an alternative way of identifying the USB device would be to supply the USB client driver with a special USB capability representing the USB device. The capability is then presented to the USB Manager with each request. This may be part of a future extension of the system.

6.5 Device Detachment Process

A detach event is recognized by the USB Manager the same way an attach event is recognized either by the port change detected bit of the `USBSTS` register or the completed interrupt transfer of the USB hub device. The only difference the port connect status bit will be zero now.

6.5.1 Client Driver Shutdown

In order to prevent the USB client driver to issue further transfers, the detach notification has to be sent to the client driver at first place. When the client driver domain receives the detach notification, the client driver domain initiates the cleanup steps such as shutting down the own services and reverse their export. Obviously potential other domains that may make use of the driver's service, should be informed. After the cleanup the domain will exit. The detachment process waits till the notification has been sent successfully.

6.5.2 Freeing up Resources

When the detach notification has been sent successfully, the USB Manager starts with freeing up the allocated resources of this device. This includes the existing USB transfers which are un-setup now and the data structures. In the end the device entry of the host controller is set to `NULL` to free up the address and the device data structure is freed.

6.5.3 Hub detachment

The detachment process of hub devices involves a bit more work. Hub devices provide attachment points for multiple USB devices and therefore the removal of a hub device may trigger a whole sub-tree in the USB topology to be disconnected from the root. Therefore for every detached hub device, the whole detachment process must be done recursively on every device at the hub's ports. It has to be said, that this is currently not implemented.

```
interface usb_manager "USB Manager Interface" {
  /* connecting to the manager */
  rpc connect(in iref driver_iref, in uint16 init_config,
          out uint32 ret_error, out uint8 ret_desc[length]);

  /* request handling */
  rpc request_read(in uint8 request[req_length],
          out uint8 data[data_length], out uint32 ret_status);
  rpc request_write(in uint8 request[req_length],
          in uint8 data[data_length],  out uint32 ret_status);
  rpc request(in uint8 request[req_length],
          out uint32 ret_status);

  /* transfer management */
  rpc transfer_setup(in uint8 type, in setup_param params,
          out uint32 ret_error, out uint32 ret_tid);
  rpc transfer_unsetup(in uint32 tid, out uint32 ret_error);
  rpc transfer_start(in uint32 tid, out uint32 ret_error);
  rpc transfer_stop(in uint32 tid, out uint32 ret_error);
};
```

Listing 6.3: USB Manager Interface Definition

6.6 USB Manager Interface

The USB Manager interface provides the functionality required by the USB driver interface (USBDI). This section will outline the semantics of the most important functions of this interface. Listing 6.3 gives an overview of the basic functions which are needed to make the USB subsystem work followed by a description of the semantics. There are non-essential functions which are not describe here. For a complete interface description refer to the Flounder interface definition `usb_manager.if`. It has also to be mentioned, that most of the functions are abstracted by their library counterpart to simplify the use.

6.6.1 Connect

This function is invoked just once during the library initialization process. The USB client driver supplies two arguments during the call. There is the initial configuration that is used to update the device configuration. The second argument is the iref of the USB driver service. This service is not associated with a name thus the USB client driver has to tell the iref to the USB Manager at the connection time.

The USB Manager returns with two arguments. First the error value, that informs the client driver about the outcome of operation. Second an extended descriptor is returned consisting of the device descriptor as well as the whole configuration descriptor. That way the client driver has access to the most important data structures and device information right at the beginning.

6.6.2 Request Handling

Every device requests is executed on the default control pipe and has to be formatted with a special format called a device request. Even there are several different standard device requests they all fall into just three different categories which differ in the direction and the length of the data flow.

The basic request, i.e. without a data stage, is used to execute simple commands on the device such as setting the address or clearing a feature. The other two requests are used to read or write data from or to the device respectively. For this, also the data for the data stage have to be transferred between the USB Manager and the client driver or vice versa.

The device requests are treated as remote procedure calls and hence only return if the request on the device is completed. This is because, usually the client drivers need the outcome of the request right afterwards.

6.6.3 Transfer Management

To initiate a new USB transfer on a device two things have to be done. First, the transfer has to be set up i.e. all the needed resources have to be allocated in advance. The USB Manager will return a unique transfer ID for reference in the future. Secondly, when this is done the USB client driver can start the transfer simply by sending the transfer start message with the transfer id to start.

Even though the calls are also in RPC style, the semantics are different. The RPC returns with the outcome of the operation e.g. if a transfer could be started or set up. The transfer done notification is sent asynchronously.

Interrupt transfers have the options of automatic restart when the transfer completes successfully. This lets the client driver just receive asynchronous messages with the data and does not have to mess around with restarting the transfer.

6.6.4 Referencing Devices and Transfers

In the USB topology, each device has its address and each endpoint has its number and there are many more things to reference. The USB Manager interface tries to hide those from the USB client driver as much as possible. The client driver does not have to deal with its own device address as well as with the transfer queues and so on. The devices are identified using the USB Manager binding and the USB transfers are identified using a transfer id. For example, the client driver can issue a start command for transfer with id 12 and the USB Manager is able to figure out the device and which particular transfer to start.

6.7 USB Library

The USB library is used by the USB Manager and every USB client driver and provides functionality of for four different kinds:

1. USB Manager binding
2. USB Manager interface abstractions
3. Class specific functionality

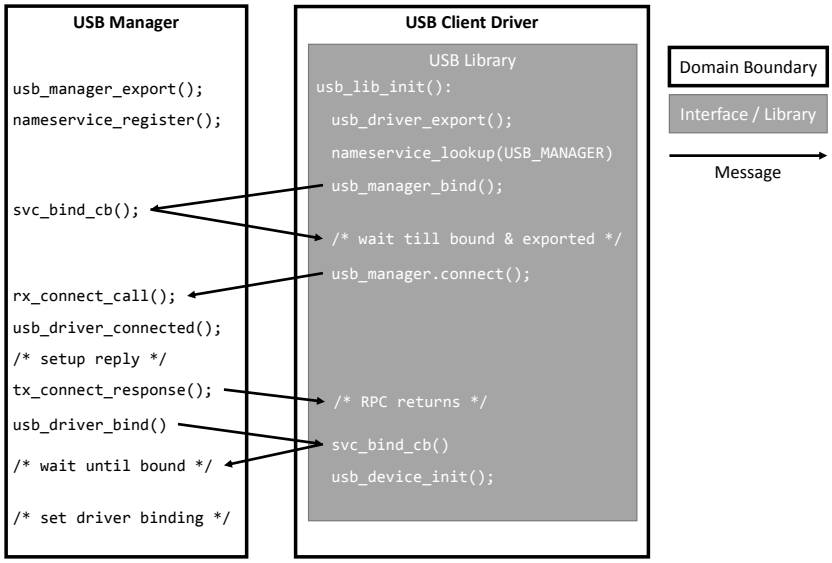

Figure 6.4: USB Library Initialization Sequence

4. General data structure definitions

This section will give a brief example for each of the four types. For a complete list of functions provided by the USB library, refer to the implementation.

6.7.1 USB Manager Binding

The first step that a USB client driver has to do is initializing the USB library. This can be done with a call to `usb_lib_init()`. The library will then establish the two-way binding between the USB Manager an the USB client driver. Figure 6.4 shows the connect sequence between the two domains.

Preparation Phase In each of the two domains there is a preparation phase where the two services are getting exported. This has to be done before any communication between the two domains happen. Obviously, the USB Manager has started his service before the USB client driver is spawned. It has to be emphasized that only the USB Manager registers its service with the name service. As soon as both services are exported, the binding process can be initiated.

Client Driver to USB Manager Binding The USB client driver looks up the iref of the USB Manager service and binds to it. As soon as the binding process is completed, the client driver executes the `connect()` RPC call and supplies the iref of its service as one of the parameters. The USB Manager will then setup the binding-device association and replies with the extended device descriptor.

USB Manager to Client Driver Binding At the point where the reply to the `connect()` call is sent successfully, the USB Manager binds to the iref it got as an argument to the connect call. This establishes the second communication channel used for sending notifications to the client driver. As the last step the USB Manager associates the USB device with that binding.

6.7.2 USB Manager Interface Abstraction

Section 6.6 already gave an overview of some of the USB Manager interface functions. In general, rather than dealing with that interface directly, the USB client drivers use the interface through the USB library. In general, the USB client drivers do not have access to the USB Manager Flounder binding which is hidden in the library.

This section shows at an example, why it is useful to access the USB Manager interface through the library instead of direct invocation. Listing 6.4 shows the library code of the `usb_get_configuration()` function which returns the current configuration value.

There are several reasons why a client driver should not deal with the USB Manager interface directly but use the USB library instead. As one can see with setup of the `usb_device_request` structure[14], it is crucial that every field has its correct value otherwise the request would be invalid and lead to undefined behavior of the device.

Further, the USB library function provides some basic checks to avoid getting corrupted data or invalid values for the fields in the `usb_device_request` structure. This function for instance checks the returned length of the request which has to be one byte, since the configuration value is expressed by a one byte number.

One may have noticed that this function itself does not use the USB Manager interface directly but rather uses a wrapper function for one of the tree request types. The USB Manager interface supports calls to one of the three basic types and those are abstracted inside the library as well by factored out code.

6.7.3 Class Specific Functionality

Each device belongs to a certain class and each class has some class specific requests or data structures which may need to be manipulated. The HID class for instance has the notion of reports and a request for setting the idle rate just to mention two of them. The USB library should contain factored out code which is common to all devices of a certain USB class.

To prevent naming clashes, each class specific function should follow the naming convention `usb_<class>_<function-name>`. Listing 6.5 shows two examples of class specific functions for the HID class and the hub class.

6.7.4 General Definitions

The USB library contains also general data structure definitions such as the different descriptor types as well as defined constants such as the USB speeds or the error codes. The most important and most frequently used files are included when the `<usb/usb.h>` header file is included.

[14]The format of this structure is defined by the USB specification

```
/**
 * \brief this request returns the current device
 *        configuration value
 *
 * \param *ret_config   the current configuration value if
 *                      zero then
 *                      the device is not configured yet
 *
 * \return USB_ERR_OK   on success
 */
usb_error_t usb_get_configuration(uint8_t *ret_config)
{
    struct usb_device_request req;

        /* setup the request data structure */
    req.bType.direction = USB_REQUEST_READ;
    req.bType.type = USB_REQUEST_TYPE_STANDARD;
    req.bType.recipient = USB_REQUEST_RECIPIENT_DEVICE;
    req.bRequest = USB_REQUEST_GET_CONFIG;
    req.wValue = 0;
    req.wIndex = 0;
    req.wLength = 1;

        /* return values of the request */
    uint16_t ret_length;
    void *ret_data;

        /* execute the request */
    usb_error_t err = usb_do_request_read(&req, &ret_length,
                                          &ret_data);

    if (err != USB_ERR_OK) {
        return (err);
    }

    if (ret_length != 1) {
        return (USB_ERR_IOERROR);
    }

    if (ret_config) {
        *ret_config = *((uint8_t *)ret_data);
        free(ret_data);
    }

    return (USB_ERR_OK);
}
```

Listing 6.4: USB Library: USB Manager Interface Abstraction

```
usb_error_t usb_hid_set_idle(uint8_t iface, uint8_t duration,
                             uint8_t id);
usb_error_t usb_hub_get_port_status(uint16_t port,
                    struct usb_hub_port_status *ret_status);
```

Listing 6.5: USB Library: Class Specific Functions

```
/* send a notification that a device is gone */
message device_detach_notify();

/* send a transfer done notification with the data */
message transfer_done_notify(uint32 tid, uint32 error,
                        uint8 data[length]);
```

Listing 6.6: USB Client Driver Interface

6.8 USB Client Driver Interface

Each USB client driver implements the interface shown in Listing 6.6. The connect process of the USB library hides the initialization of the client driver interface from the client driver programmer. The data and message flow of this interface is always from the USB Manager to the USB client driver.

6.8.1 Detach Notification

During the detachment process the USB Manager sends a detach notification to the USB client driver whose corresponding device was removed from the USB. The library handles the receive event and initiates the shutdown processing. When this message is received, the client driver domain will be exited.

6.8.2 Transfer-Done Notification

When a USB transfer completes, the USB Manager sends a transfer complete notification to the client driver. The first two arguments contain the most important information such as the ID of the transfer that has finished (`tid`) and the outcome of the transfer (`error`). The `error` is important, since the transfer may be stopped by other reasons than successful termination. The parameter data contains the payload of the USB transfer if it is completed successfully.

When a new transfer is setup the user may specify a callback function which is called when the transfer completes. The library handles the receive event and invokes the callback function if there is one set or simply discards the data otherwise.

6.9 USB Keyboard Driver

This section describes a USB client driver at the example of a USB keyboard. The client driver consists a USB driver service part and a keyboard service part. The client driver is initialized in four steps:

1. Initialize the USB library
2. Initialize the USB keyboard, setup the related data structures
3. Start the keyboard service
4. Start the USB transfers

The following subsections will explain how the USB keyboard is initialized and hence give an example on how the USB transfers are set up. This section will not give details on how to convert the USB code to keyboard scancodes refer to the implementation for this.

6.9.1 Setting up Transfers

Each HID device has at least one control transfer and one interrupt transfer as specified by the HID specification [6]. The control transfers usually are executed on the default control pipe and hence do not need to be set up explicitly by the client driver. Listing 6.7 shows an example how to set up interrupt transfers (other types are set up analogously)

First a data structure containing the setup information has to be initialized. One can see, that the transfer is of type interrupt. In this case and we don't care which of the endpoints is used as long as the direction of data flow is from the device to the host (IN)[15]. There are several flags that can be set, the `auto_restart` flag will initiate an automatic restart of the interrupt transfer when it completes successfully.

The actual setup is initiated by sending the transfer setup information to the USB Manager. For every transfer that is set up that way a transfer ID is returned back. The callback function is associated with the transfer ID and is called when the transfer done notification is received with this ID.

6.9.2 Transferred Data

For HID devices, the data transferred by the interrupt transfer have a certain format which is referred to as a so-called report. The report has to be parsed in order to get the desired information. Concerning the USB keyboard, there are two types of events that are reported via the interrupt transfer. Every key press and key release action will generate a new report containing the USB codes of the pressed key or zero when it is released again. The keyboard driver has to figure out which keys are pressed, released or hold. Refer to the HID specification [6] for a complete specification of the report format.

[15] the USB Manager will look up the number by the type and direction

```
/* transfer setup information */
static usb_transfer_setup_t
 keyboard_tconf[USB_KEYBOARD_NUM_TRANSFERS] = {
    [USB_KEYBOARD_DATA] = {
        .type = USB_TYPE_INTR,
        .iface = 0,
        .endpoint = USB_ENDPOINT_ADDRESS_ANY,
        .direction = USB_ENDPOINT_DIRECTION_IN,
        .max_bytes = 0,
        .interval = 5,
        .flags = {
            .short_xfer_ok = 1,
            .pipe_on_falure = 1,
            .auto_restart = 1,
        },
    },
};

/* call to setup the interrupt transfer */
usb_error_t err = usb_transfer_setup_intr(
            &keyboard_tconf[USB_KEYBOARD_DATA],
             usb_keyboard_transfer_cb,
            &keyboard.xferids[USB_KEYBOARD_DATA]);
```

Listing 6.7: USB Transfer Setup

Setting the LEDs

In order to change the LED state on the keyboard, data has to be written to the keyboard. Again the data has to be formatted as a report and passed via data stage of a `SET_REPORT` request. The USB keyboard does not automatically set the LED-indicators when one of the lock keys is pressed and the client driver has to keep track of the state.

6.9.3 Learning the Modifier Keys

A standard keyboard contains several keys to modify the actual typed keys. The positions of `SHIFT`, `CTRL` and friends may not necessarily always be at the very same place on every keyboard. To learn the locations of these modifier keys, the HID descriptor needs to be parsed and the location information extracted.

6.9.4 The Idle Rate

HID class device deliver new reports either on a `GET_REPORT` request, by the interrupt when an input event was recognized or at a specified rate every x ms. This rate is called the idle rate. If an idle rate is set the USB HID device will generate a new report via the interrupt transfer even if no key was pressed or released. This may be useful, when packets are lost since the idle rate will simply transfer the last packet again.

6.10 Example Usage

Figure 6.5 shows an example usage of the whole system and the messages involved.

1. A key is pressed on the keyboard
2. The interrupt transfer completes and an interrupt is generated
3. The USB Manager gets the interrupt and processes the transfer
4. Transfer done notification is sent to the keyboard driver
5. Keyboard driver converts the USB code from the report to scancode
6. The scancodes are transmitted to the subscribers (Fish)
7. Fish shows the pressed key on the screen.

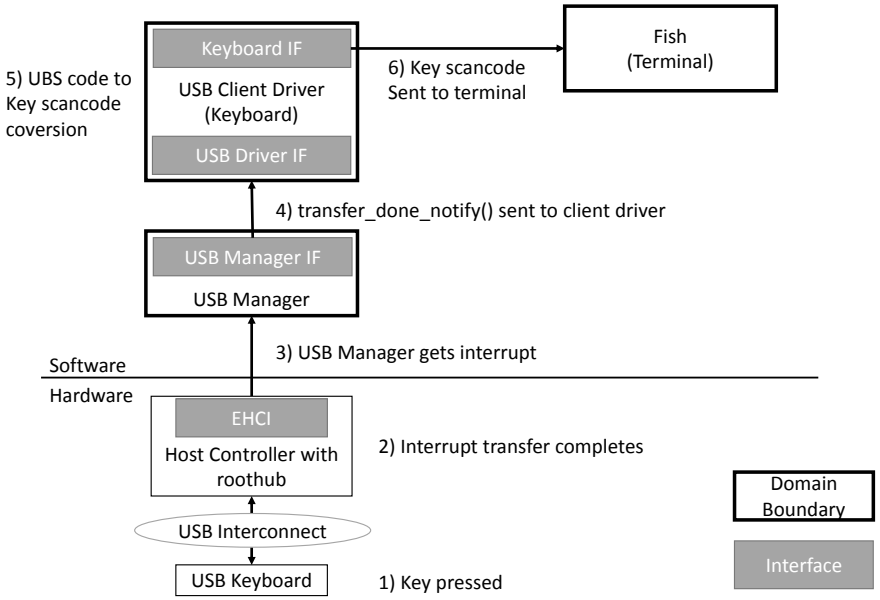

Figure 6.5: USB Usage Example

Chapter 7

Discussion

This chapter gives an overview of the USB subsystem implementation state on Barrelfish and its limitations. This chapter is divided into three parts. The first section explains how the current implementation can be used in Barrelfish, followed by a section outlining the limitations and the last section will give anchor points for future work related to the USB subsystem as well as Barrelfish.

7.1 Setting up Fish

Obviously just having a device driver idling around in an OS does not really bring benefits. A consumer for the features provided by the driver has to exist in the system. The most basic way to use the keyboard input is a shell: So, let's get the Fish[1] started. However, Fish had to be slightly modified to make it run since the OMAP44xx does not know how to handle power off and reset commands. They are simply removed in the ARM version.

Fish looks for the keyboard service and connects to it. The terminal library had to be modified in such a way the initialization procedure does not complete until the binding with the keyboard service completed. Therefore, Fish will be blocked until the keyboard service is started. Together with an attached USB keyboard fish can be used as a shell. Figure 6.5 in the previous chapter shows the data flow when a key is pressed.

It has to be mentioned, that the terminal implementation is to be replaced by a new one. Raphael Fuchs [9] created a new session capable terminal system. The code of this is not yet merged with the main tree but is expected to be merged soon. The USB keyboard driver implements already the `keyboard.if` interface.

7.2 Limitations

The current implementation has several limitations which originate either from a missing implementation or a hard coded sequence of instructions. This section will give a brief overview of the limitations.

[1] The shell is called Fish on Barrelfish

Transfer Type	State
Control Transfers	Implemented and working.
Interrupt Transfers	Implemented and working.
Bulk Transfers	Support is there, but not tested so far.
Isochronus Transfers	Special cases not implemented and not supported

Table 7.1: Overview of Implemented Transfer Types

7.2.1 USB Transfer Types

In order to get a USB keyboard work, only two of the four different transfer types need to be supported

- Control Transfers for configuring the device
- Interrupt Transfers for getting the key events

Therefore, the implementation focus was on those two types. Table 7.1 summarizes the support for each transfer type.

Bulk Transfers Due to the generality of the design and the fact that bulk transfer do not need a special treatment, the support for bulk type transfers is expected to work without much additional effort. However, currently only devices of the HID class are supported and hence no devices which may use of bulk type transfers can be used because there is no such client driver. This implies that the functionality of bulk transfer types is not tested.

Isochronus Transfers At several stages of the transfer management, the isochronus transfer types need a special case. In each of the special cases, an assertion will be risen where the code is missing. Further, there is no single isochronus type: the EHCI specification distinguishes HS isochronus transfers and FS isochronus split transfers. Therefore there are two missing cases in the EHCI specific handling of isochronus transfers.

7.2.2 USB Manager and USB Driver Interface

Currently not every USB Manager interface function does something useful i.e. only stubs exists. In particular the functions for getting the current state of a transfer are not implemented. They are basically not essential but may be a nice-to-have when longer running transfers such as bulk type transfers are added to the system.

7.2.3 Host Controllers

Table 7.2 summarizes the state of the different host controller implementations. The OMAP44xx SoC has two integrated host controllers: an EHCI and an OHCI controller. Only the EHCI controller is used in the current implementation and hence only this code is expected to work.

Host Controller	State
UHCI	not implemented, some stubs exist
OHCI	partially implemented and tested
EHCI	implemented and running
XHCI	not implemented, some stubs exist

Table 7.2: Overview of Supported Host Controllers

OHCI Host Controller The current implementation supports some functionality related to the OHCI host controller such as initializing the host controller hardware. The implementation of most of the functions already exists. However, the behavior of the code towards functionality beyond initialization is not tested and expected to contain bugs.

UHCI and XHCI Host Controllers The support for UHCI and XHCI host controllers is not implemented, but there exists stub signatures for the initialization process in order to make the code compile.

Companion Controller The EHCI controller is not capable of serving FS/LS devices. Thus the port should be handed over to the companion controller as soon as the device is recognized as FS/LS. The disown functionality of a root hub port is not present and thus FS/LS devices are only supported through a HS hub with transaction translator.

7.2.4 USB Quirks

Some USB devices do not strictly follow the USB specifications, need a special way of initialization to work properly[2] or simply have bugs such as wrong device descriptor. The current implementation treats every device as a fully USB compliant device with no bugs and no special handling needed. Thus if a device is attached which has a known issue in one of the tree points the device may not work properly. However, since the issue is known, a quirk could be applied to make it work again. Thus the implementation is limited to bug-free USB devices.

7.2.5 Error Handling

There are some cases where an operation may fail inside the USB Manager especially during the attach or detach process of new devices. Error recovery such as re-enumeration or handling of stalled endpoints is very limited and the execution is skipped.

7.2.6 Power Requirement Check

The USB specification also contains guidelines on the power management of the USB. Several restrictions protect the USB from an over-current situation such as connecting too many devices at a bus-powered hub. The current implementation

[2]Referring to freeBSD [8]

skips the check if there is sufficient power available to support all connected devices on a hub.

7.2.7 Device Driver Lookup

If a new device is attached to the USB and the device information is read out, a suitable device driver binary has to be found. The current implementation has a hard-coded lookup procedure. Thus adding new client driver binaries always involves to change the hard-coded lookup table in the USB Manager. This complicates the adding of new USB client drivers or the possible use of alternate drivers. However, the lookup procedure is designed in such a way, that it can easily be modified to involve Kaluga/SKB.

7.2.8 Hot-Plug

The hot plugging of devices basically works. However, the de-registration of the client driver is currently not implemented. Therefore when a keyboard is attached the second time, it tries to register the keyboard interface a second time which will fail.

7.2.9 Single Threaded USB Manager

The current implementation of the USB manager is single threaded. This may become a bottle neck when there are several client drivers want to execute device requests simultaneously. The device requests are implemented as RPCs and hence only one RPC invocation can be handled at a time. This means the USB Manager is blocked until the request completes. The duration of a request depends on the responsiveness of the USB device. However most of the requests are used for configuration purposes. Starting and setting up normal USB transfers only involve the USB Manager and hence do not have to wait for the USB device.

7.2.10 Different SoC Support

The way the USB Manager is started in Kaluga is hard-coded. Thus if the system is running on another SoC than the OMAP44xx, it is not expected that the host controllers are located at the same physical address. This implies that only the OMAP44xx SoC is currently supported.

7.2.11 64-bit Support

If the EHCI controller is running on machine with a 64-bit addressing, the addresses have to be extended to 64 bits. The EHCI controller supports only 32 bit register sizes. The problem is solved in such a way that the queue heads, elements and so on are allocated within the same 4GB region of memory. The address of this memory region is then stored in a separate register and added upon access. Further the data structures have 64 bit extensions which are currently not implemented. Therefore it is expected that the implementation is limited to 32 bit only.

7.2.12 PCI

On the x86 architecture, the devices reside on the PCI bus in general. Thus the host controller will also be somewhere located on the PCI bus. The OMAPP44xx does not have a PCI bus and therefore it was not checked if it works on a PCI bus as well.

7.2.13 Keyboard

Basically the keyboard can be used, however the behavior is not always that smooth. The reason for this seems to be the missing interrupts. The first key stroke and release is recognized as expected but the following ones only one of the two are captured. As a work around, the idle rate ensures that the missing event is transferred again and the USB keyboard driver get the event data.

Further with the current implementation of the terminal, the support of the LEDs is not implemented. However, a driver option exists to enable the integrated parsing of the codes and disabling the scancode generation. With this option chars are directly generated with working modifiers and the LED states are updated accordingly.

7.3 Future Work

This section outlines some anchor points that may be used for future projects on the USB subsystem or Barrelfish in general.

7.3.1 USB Class Support

The current implementation supports just the human interface device class besides the special hub class. Adding support for other device classes such as mass storage devices (MSD) or networking devices is a good starting point to extend the device support of the current implementation. For every new device class, the corresponding USB library functions have to be added as well.

Mass Storage Device Class As the name describes, this class contains devices such as hard drives or flash drives. In the context of the PandaBoard, additional memory can only be provided using the MMC slot or an attached USB mass storage device. Bringing file system support to the PandaBoard would become possible if mass storage devices are supported. That way, applications could store their data or the executable can be placed on the USB mass storage device and dynamically loaded.

Communication Device Class Networking is considered to be one of the most important parts of an operating system. The PandaBoard has an Ethernet port, which is attached to the USB. Thus in order to get network support on the PandaBoard, the support for Communication Class Devices is required.

7.3.2 Flounder Interfaces: Using THC

There are quite a lot functions in the `usb_manager.if` interface. To simplify the handling of the messages, it may be beneficial to switch to the THC library.

7.3.3 Adding USB 3.0 Support

With USB Revision 3.0 a powerful new standard has been established recently. The new XHCI controller not only brings a new register interface but also requires some additional management effort from the host controller driver side. Implementing the XHCI host controller driver would avoid the problem of the companion controllers, since the XHCI controller is able to support devices of all speeds.

7.3.4 Resource Allocation

In the current implementation all the needed resources are allocated by the USB Manager. It may be beneficial that the USB client driver or even the application domain that uses the driver allocates the needed memory and passes the capability to the USB Manager. It has to be evaluated whether or not host controller specific data structures should be allocated with USB Manager resources or with memory of the driver supplied during transfer setup.

Bulk Transfers There may be a problem when bulk transfer support is fully added to the system because by using bulk transfers big files may be loaded into main memory. Then the USB manager has to allocate big chunks of memory on behalf of other processes.

Desired Resource Allocation It makes sense that the source / sink of a data transfer - especially if the data size is big - allocates the resources before the transfer is started. As an example consider an application that wants to load a file from a mass storage device. The application then allocates the memory and passes the capability to the MSD driver, which then passes the capability to the USB Manager. The device then does a direct DMA transfer to this memory region. That way, neither the USB Manager nor the client driver does have to deal with the resource allocation for the frame buffers.

NUMA-aware Resource Allocation If the system runs on a machine with non-uniform memory access, it may make sense to consider the issue of where to allocate the resources. For some applications it may be better to allocate it close to the host controller or close to the requesting application. This may be taken into consideration when the resource allocation is redesigned.

OS Bulktransfer Integration An operating system usually has support for bulk transfer of data between domains. Extending the interfaces of the USB subsystem with the OS bulk system may be beneficial for MSD usage as well as networking.

7.3.5 Muxing and Power/Clock Management

The muxing process of the wires on a SoC such as the OMAP44xx is tedious and error prone. If the muxing code is hard coded, changing the muxing involves a lot of work and does not necessarily work on other SoC. Therefore, setting the pad configuration to the correct mux values should be handled in a general way within a special SoC initialization driver which runs before any other driver

domain is started. This may be handled analogue to the PCI configuration but with fixed address ranges.

7.3.6 Capabilities

Currently the client driver - device association is done on the basis of the IDC binding and relies that the connection invocation is done by the correct client driver domain. This is partially enforced to be the case, but there may be other domains which accidentally invoke the connect function and hence may screw up the whole association. Therefore, it may make sense to introduce a special USB capability which is supplied to the USB client driver when it gets spawned. The client driver presents the capability with each invocation.

7.3.7 Hot Plugging and Multiple Devices

In contrast to traditional devices, the USB devices have a slightly different usage pattern. The biggest difference is that the can be hot plugged into the running system and removed again. Also there may be multiple devices which are of the same type i.e. two flash drives connected to the same host. This may result in the case that the driver service is exported multiple times. Thus having de-registration support or a way to export a service and register it with the name service more than once is a problem to be solved.

7.3.8 Starting USB Client Drivers and SKB

The USB client drivers should be started by the device manager Kaluga. That way Kaluga is aware of the running drivers in the system. In cooperation with the SKB Kaluga can find the most suitable driver by executing a query on it. There has to be a record format specified similar to the one of the PCI devices, which is used as a basis for device driver lookups.

Bibliography

[1] ETH Zurich Andrew Baumann, Systems Group. *Inter-Dispatcher Communication in Barrelfish, Barrelfish Technical Note 001*, May 2011.

[2] Andrew Baumann; Simon Peter; Adrian Schüpbach; Akhilesh Singhania; Timothy Roscoe; Paul Barham; and Rebecca Isaacs. Your computer is already a distributed system. why isn't your os? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, May 2009.

[3] National Semiconductor Compaq, Microsoft. *Open Host Controller Interface Specification for USB, Release 1.0a.* Compaq, Microsoft, National Semiconductor, September 1999.

[4] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2013. Chapter 10.

[5] Microsoft Corp. Usb driver stack architecture. `http://msdn.microsoft.com/en-us/library/windows/hardware/hh406256(v=vs.85).aspx`, June 2013. [Online; accessed 2013-07-16].

[6] USB Implementers Forum. *Device Class Definition for Human Interface Devices (HID), Version 1.11.* USB, June 2001.

[7] USB Implementors Forum. Superspeed usb (usb 3.0) performance to double with new capabilities. `http://www.usb.org/press/USB-IF_Press_Releases/SuperSpeed_10Gbps_USBIF_Final.pdf`, January 2013. [Online; accessed 2013-06-30].

[8] FreeBSD. Freebsd documentation. `http://www.freebsd.org/doc/en/books/arch-handbook/usb.html`, June 2013. [Online; accessed 2013-06-30].

[9] Raphael Fuchs. A session control interface for a multikernel. Bachelor's thesis, ETH Zurich, August 2012.

[10] Takahiro Hirofuchi; Eiji Kawai; Kazutoshi Fujikawa; and Hideki Sunahara. Usb/ip—a peripheral bus extension for device sharing over ip network. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 47–60, 2005.

[11] Gerd Griessbach. Usb for drops. Master's thesis, TU-Dresden, Chair of Operating Systems, March 2003.

[12] Intel. *Universal Host Controller Interface Design Guide, Revision 1.1.* Intel Corporation, March 1996.

[13] Intel. *Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0.* Intel Corporation, March 2002.

[14] Intel. *eXtensible Host Controller Interface for USB, Revision 1.0.* Intel Corporation, May 2010.

[15] Inc Linux Kernel Organization. The linux kernel archives. `https://www.kernel.org/`, July 2013. [Online; accessed 2013-07-18].

[16] OMAPpedia.org. Pandaboard usbboot. `http://omapedia.org/wiki/PandaBoard_USBBOOT`, April 2012. [Online; accessed 2013-07-07].

[17] TU Dresden Operating Systems Group. The l4 u-kernel family. `http://os.inf.tu-dresden.de/L4/bib.html`, Sept 2005. [Online; accessed 2013-07-18].

[18] Pandaboard.org. *OMAP4460 Pandaboard ES System Reference Manual, Revision 0.1.* Pandaboard.org, doc-21054 edition, September 2011.

[19] Compac; Hewlett-Packard; Intel; Lucent; Microsoft; Nec; Philips. *Universal Serial Bus Specification, Revision 2.0*, April 2000.

[20] Andrew Baumann; Paul Barham; Pierre-Evariste Dagand; Tim Harris; Rebecca Isaacs; Simon Peter; Timothy Roscoe; Adrian Schüpbach; and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on OS Principles*, Big Sky, MT, USA, October 2009.

[21] Akhilesh Singhania and ETH Zurich Ihor Kuz, Systems Group. *Capability Management in Barrelfish, Barrelfish Technical Note 013*, March 2011.

[22] SMSC. *Highly Integrated Full Featured Hi-Speed USB 2.0 ULPI Transceiver (USB3320)*, July 2009.

[23] HP; Intel; LSI; Microsoft; Nec; Samsung; ST-Ericsson. *Wireless Universal Bus Specification 1.1.* usb.org, September 2010.

[24] ETH Zurich Systems Group. *Mackerel User Guide, Barrelfish Technical Note 002*, March 2013.

[25] Systems Group ETH Zurich Team Barrelfish. *Barrelfish OS Services, Barrelfish Technical Note 012*, August 2010.

[26] TI. *OMAP4460 Multimedia Device Silocon Errata, Revision A.* Texas Instruments, public version edition, September 2011.

[27] TI. *OMAP4460 Multimedia Device Silocon Revision 1.x, Version Y.* Texas Instruments, technical reference manual edition, March 2013.

[28] Animesh Trivedi. Hotplug in a multikernel operating system. Master's thesis, ETH Zurich, 2009.

[29] usb.org. Usb class codes. `http://www.usb.org/developers/defined_class`, December 2011. [Online; accessed 2013-07-16].

[30] Dirk Vogt. Usb for the l4 environment. Master's thesis, TU Dresden, September 2008.

[31] Gerd Zellweger. Unifying synchronization and events in a multicore operating system. Master's thesis, ETH Zurich, 2012.