HelenOS USB 3.x Stack

Ondřej Hlavatý, Jan Hrach, Jaroslav Jindrák,
Petr Mánek, Michal Staruch

March 18, 2018

Contents

List of Figures

List of Listings

Chapter 1

Introduction

You are reading the main documentation of the HelUSB3 project. The main goal of the project was to extend the existing USB support in HelenOS with support for USB 3, namely implementing the driver for host controllers compliant with the eXtensible Host Controller Interface (xHCI), extending the USB device driver framework to USB work at USB 3 speeds, xHCI-only devices and overall refactoring and cleanup of the USB subsystem in HelenOS. The previous support for USB was implemented as a software project back in 2011, then was maintained and extended by HelenOS developers (mostly by one team member of the project).

1.1 Structure of This Document

In the first chapter, we try to build context needed to understand the following chapters. You can get an idea of what HelenOS is and how drivers work there. Then, we try to introduce the reader to idioms and paradigms used in the USB world, as the rest of the documentation assumes the reader has this basic knowledge. All of this information is included only for the sake of completeness, we do not aim to replace (nor duplicate) the documentation of HelenOS nor USB specifications. Please consider these sections as informative and supportive only, and follow on topics in their respective documentations.

In the second chapter, we focus on the xHCI driver, which is a major part of the project. We describe its overall architecture and discuss implementation decisions. This chapter, together with in-code documentation and knowledge of the xHCI specification, shall provide reader enough information to modify and extend our implementation without much trouble.

The third chapter is dedicated to modifications to the existing USB subsystem. We do not discuss the obvious ones (like support for USB 3 hubs), but those that would be avoidable if we took the assignment to the word. Since we like to leave the table clean, we had to change some design decisions of the previous authors. Some of the changes introduce more systematic approach, some were needed to implement additional parts of the assignment without using a duct tape. Some of them are just optimizations towards better performance or usability. Also, the documentation of the previous project was obsoletes by recent developments in HelenOS, so we try to document the current state of the USB subsystem.

To support the implementation process and to allow us to test it in a semi-automatic way, we have developed a standalone testing mechanism that tests the USB stack end-to-end by using modified virtual environment. This is the topic of Chapter 4.

The last chapter concludes the documentation with a brief summary of the accomplished project goals, instructions on how to access the materials stored online and on the attached CD. In addition, it outlines possible future development of the USB framework following the project end.

As the USB subsystem is mostly useless without drivers for actual USB devices, we would like to lessen the initial barrier that stands in the way of implementing new USB device drivers. The Appendix A is written as a complete guide to writing device drivers, hiding implementation details of the stack and focusing on how to use it only. It still requires the reader to understand the USB architecture and mechanics though, as well as understanding the device they are writing the driver for.

The Appendix B includes the official project specification (in Czech) and a concise timeline of the project realization throughout the years 2017 and 2018.

1.2 About HelenOS

HelenOS is a microkernel operating system. Contrary to operating systems with monolithic kernels, key operating system functionality (including device drivers, filesystems and networking) is implemented in user-space processes (or servers) that interact via message passing interface. The rationale of this decision is that the failure of one component, e.g. a faulty driver, does not result in the crash of the entire system. It also allows a more modular system design.

HelenOS was started in 2004 as a software project at the Faculty of Mathematics and Physics at Charles University and currently is being developed by students, former students and staff of this university, among with a number of contributors around the world. It has participated in Google Summer of Code several times. HelenOS is used as a research operating system and as a platform for bachelor and master theses and student software projects.

HelenOS supports several architectures (among them ia32, amd64, 32-bit ARM and big endian MIPS) and is released under the BSD license.

1.2.1 xHCI Support in Operating Systems

Extensible Host Controller Interface was first drafted in 2008 and the final version of the specification was released in 2010. Since then, xHCI support arrived in all mainstream operating systems like Windows, Linux, BSDs and Mac OS X.

Regarding special-purpose, microkernel and research OSes, xHCI support is not yet widespread. Most notably, xHCI support is included in Google's microkernel-based Fuchsia and in bare-metal iPXE. Haiku Project is currently developing xHCI driver for their OS.

In HelenOS, the support for USB was started by the HelUSB project defended in 2011. In that time, the USB driver framework was designed, and delivered with a few USB drivers. From the Host Controller side, UHCI and OHCI were supported almost completely. As for the device drivers, only a generic HID driver was provided to demonstrate the functionality of the framework. Since the project was delivered, the USB stack evolved a little and new drivers were implemented. The EHCI support was rather rudimentary and there was no support for xHCI.

1.3 Drivers in HelenOS

Drivers in HelenOS are separate userspace tasks. There is a common framework for writing device drivers, the *Device Driver Framework*, abbreviated as DDF. The DDF consists of two major parts: the `devman` service taking care about starting drivers, attaching newly found devices to drivers, and many more; and a `libdrv` library creating a platform to write DDF-compatible drivers on.

A driver, once started, is essentially a server task. It fills a callback table with driver operations, and waits for being called by the library. Once the devman wants the driver to take care of a new *device*, its `dev_add` callback is called. The driver is then supposed to do whatever it needs to make the device operating, eventually creating *functions* – nodes that are passed back to devman to be taken care of. The function can be either an *exposed* function, being a leaf node usable directly by the user (keyboard, disk partition, ...), or an *inner* function, essentially being a *device* for another driver.

Drivers then communicate with each other using *interfaces*. Every function is provided with a set of interfaces it provides to the child driver. The driver itself then serves as a mediator or translator of operations called on its functions to yet another operations called on its device. For example, the *PCI* driver exposes a function for every card physically attached to the bus. One of its functions is an xHC, which is given as a device to the *xHCI* driver. It then creates a function for every gadget connected through USB. USB functions are provided with an interface to issue USB transfers. USB device drivers use this interface to actually drive the USB devices. The xHCI driver translates the transfer requests to writes into PIO space offered by the PCI driver as a part of the PCI function interface.

In the very end, the device tree exposes only its leaves. Every leaf node (exposed function) can be added into several categories. Categories being a well-known classes of devices (e.g. a disk partition) provide an interface of a service. For the user, the whole DDF ecosystem looks like a modular implementation of the service interface, allowing them, for example, to mount the partition regardless of where it is located and how it is connected to the system.

1.4 Briefly About USB

Disclaimer: This chapter is heavily copied from the documentation of the previous project *USB subsystem in HelenOS*[6] and filled in with the new features and specifics of USB 3.

This chapter provides a brief overview of the USB architecture. It does not aim to be a drop-in replacement for Universal Serial Bus specifications. It can be also viewed as a summary of what the reader of this document must already know. If you are new to USB, you can use this chapter as a starting point for further reading.

1.4.1 The 'Big' Picture

USB is a technology used for connecting peripherals to host computer. Its main advantages are simplicity of usage for end users, pretty good scalability (in terms of number of simultaneously attached devices) and a choice of possible transfer types that accommodate requests of diametrically different devices. The USB defines the following aspects (most of them will be mentioned later in this chapter):

- bus physical topology
- physical interface — both mechanical (plugs) and electrical
- power management
- configuration of attached devices
- querying attached devices for supported functions
- low-level data protocol used on the bus
- high-level data protocol — transfer types

The following sections will describe parts that are most important for programmers of USB-related software, such as device drivers or host controllers. They will not include description of low-level aspects, such as electrical current settings or sizes of mechanical plugs. The acronym USB is used in two slightly different meanings in the following text. First, as a common prefix (e.g. USB device), second in its original meaning, describing the bus itself (when the word 'bus' is used, it will always refer to the Universal Serial Bus unless explicitly stated otherwise).

1.4.2 Bus Topology

USB uses a tree-like physical bus topology. In the root of the tree there is the host computer with a hardware chip — host controller — controlling all communication over the bus. All communication requests of device drivers are directed to the driver of the host controller that schedules them and then sends them to the USB 'wire'. The host controller provides several *ports* to which devices are connected.

USB devices come in two flavors. First, there are *functions* — these provide actual functionality and create leaf nodes in the bus topology. An example of an USB function is a printer or a keyboard. The other kind are *hubs* that create branches on the bus and to which the other devices might be connected.

Hubs provide means to attach more devices to the same bus. The attachment place is called a port (typical hubs for PCs has about four ports). Each hub has two different kinds of ports. First, there is the upstream port through which the hub is connected to a parent device (i.e. the predecessor in the tree) and then there are several downstream ports through which the other devices are connected. The host controller must always provide at least one port to allow device attachment. Typically, the ports provided by the host controller are provided by its own hub, called *root hub*.

All USB versions must be backwards compatible in a sense that if you connect the latest-version USB device to a most recent USB host controller through USB 1.0 hub, the communication will use the protocol as specified by USB 1.0. For the drivers, the precise protocol version used usually does not mean much, differences are described in terms of different *speeds*. Knowledge of the speed the device operates on is important for proper scheduling.

- **USB 1** defines *low* and *full* speeds (OHCI, UHCI, xHCI)
- **USB 2** defines *high* speed (EHCI, xHCI)

- **USB 3** defines *super* speed (xHCI only)

Looking at the list of controllers supporting individual speeds you can see that EHCI is missing for low and full speeds. EHCI supports only high-speed devices on its root hub ports. There must be either a companion controller sharing physical plugs or an embedded high-speed hub to support low- and full-speed devices too.

The correlation between speeds and USB protocol version resulted in unclear distinction between them. Unless explicitly stated, a “high-speed device” refers to a device compatible with USB 2.0 that is able to communicate at high, full, and low speeds. Counterintuitively, “super-speed device” is a device that operates using the USB 3 protocol. This is due to the dual-bus topology explained in section 1.4.8.

Although physically, the topology is a tree rooted in the host controller, the topology is abstracted by the HC. In USB 1, hubs are just transparent splitters that add no logic to the bus. USB 2 adds high-speed hubs, which can isolate environments of different speeds. USB 3 with its dual-bus topology is even more complex – every USB 3 hub behaves as two almost-separate hubs, a regular high-speed one and another super-speed only. When a device is connected, a decision is made on which one it will show up. Furthermore, USB 3 allows to control separate bus instances for arbitrary subsets of roothub ports. Also, every link is now separate full-duplex link, so that hubs now must be smarter and employ a Store and Forward paradigm. The difference is similar to the difference between Ethernet hubs and switches – a virtual bus topology has changed to a routed star topology.

Each device connected to a bus instance receives a unique address (a positive integer) and reacts only to communication that is targeted to it. Before the device receives this address, it listens on the *default address* (number zero). To prevent situation when more devices would listen on the default address simultaneously, each hub must provide means to disable communication forwarding to certain ports (USB terminology says that hub does not *signal* on given downstream port). USB 3 with separated instances uses a different mechanism, a *Route String*, which contains the number of downstream port on every tier to route the packet to its destination.

1.4.3 Device Configurations, Interfaces and Endpoints

USB was designed to be as flexible regarding device configurations as possible. Because of this, each device can provide several *configurations*. Each configuration may provide completely different sets of features but in reality practically almost all common USB devices have only a single configuration.

Each configuration can provide one or more *interfaces*. The interface provides the actual functionality of the device. For example, an USB printer provides the printing as an interface. Some devices may provide multiple interfaces. For example, a digital camera may provide an interface for direct communication with the camera (usually via vendor-specific drivers) and, as a fallback, a mass-storage interface that can be used to download photos when the specialized drivers are not available.

Each interface can have several *alternate settings* that change device capabilities at the interface level. For example, network card may offer several alternate settings using different sizes of data packets, or cameras offering an alternate setting for each supported resolution.

The actual communication with the device performed through *endpoints*. Each endpoint belongs to a single interface and represents a communication ‘gate’ to the device — USB uses the term *pipe* to describe an abstract connection between the device driver on one side and the device on the other one. Endpoints are usually unidirectional (exceptions are control endpoints) and they have several attributes that must be obeyed by the driver.

A special endpoint — *control endpoint zero* (also called default control endpoint) — is present on each device and is used for configuration purposes and for standard requests (e.g. for querying the device for generic capabilities).

1.4.4 Bus Protocol and Transfer Types

The USB defines how data that are supposed to be sent to (or received from) a device shall be encoded and treated prior to their sending over the wire. This overview skips the physical aspect of the communication and starts with an abstraction, that the host controller has, which is a way to send a stream of bytes using the wire.

Communication over the bus uses data *packets* which encapsulate the actual payload and some service data. The service data include target address, endpoint number, packet type and data needed for packet integrity checks.

Before moving on to describing packet types, it is necessary to mention transfer types. In order to allow devices with different transfer requirements (e.g. web camera that supplies a continuous stream of data vs. keyboard with a few bytes several times a minute) the transfers (and therefore endpoints) are divided into four different types.

The first type of transfers are *control transfers*. These are used for the device configuration and for manipulating the state of the device. Every device has at least the default control endpoint but it is possible for the device to have multiple control endpoints. The control transfers are bidirectional.

The second type are *interrupt transfers*. These are intended for small and infrequent data where minimizing the latency is the primary goal. These are considered periodic transfers and the host controller reserves a part of the bus bandwidth for them. The typical devices using this transfer type are all HID devices.

The third type of transfers are *bulk transfers*. Bulk transfers are used to send large chunks of data over the bus when the requirements are to have correct data, but there are no strict requirements on latency. Typical devices include printers, scanners or external disks. USB 3 also introduces streams as an another layer of abstraction over bulk transfers.

The last type of transfers are *isochronous transfers*. Unlike bulk transfers, where data integrity is a priority, isochronous transfers ignore data integrity and all priority is put on quick and periodic delivery (policy is that delayed data are worse than damaged ones). Multimedia devices such as web-cameras or wireless controllers use this transfer type.

When a device driver wants to send data to the device (the device cannot initiate the transfer), it gives the payload to the host controller together with information about target device (address) and endpoint. It must also provide information about transfer type and data direction. For incoming transfers, it instead reserves the data buffers to which the device can send the data. The host controller then encapsulates these information and issues USB packets on the bus. This is already part of the abstraction xHCI provides, the xHCI driver only instructs the hardware to do so. Former HC interfaces had to schedule the individual transfers into USB *frames* (a time unit on which USB is synchronized).

The protocol is roughly the same for bulk, interrupt and isochronous transfers. Control transfers are a bit different because they have a special preamble — a setup packet. The setup packet contains commands that might change the device state and this packet might be optionally followed by a data phase. The control transfers are the only ones that are actually bidirectional (because the setup packet is always outgoing, while the data phase could be either in or out).

1.4.5 Device Descriptors and Device Classes

In previous sections querying USB device for its capabilities was mentioned. This section will describe this querying in more detail. Each USB device must provide several data blocks, called *descriptors*, describing its configuration and capabilities. Some descriptors are device-dependent while others are generic for any USB device.

The roughest description of a device is provided in a *device descriptor*. It holds information about device vendor, device class (see below), number of possible configurations and several other details.

Each configuration is then described by a *configuration descriptor*. This descriptor also contains the number of interfaces the configuration provides. Each interface is then described by an interface descriptor. The interface descriptor specifies which class the interface belongs to and how many endpoints it specifies.

Unsurprisingly, each endpoint is described by an *endpoint descriptor*. This descriptor defines attributes of the endpoint — data direction, transfer type and endpoint number.

In case of USB 3, an endpoint descriptor is followed by a *superspeed endpoint companion descriptor*. This defines USB 3 specific features such as bulk streams.

The descriptors can be viewed as a tree, where the device descriptor is the root node that has several descendants — configurations. Each configuration then groups interfaces and endpoint descriptors are leaves. Device may provide its own descriptors (e.g. vendor-specific ones) that may stand aside of this tree or be part of it (then they are typically inserted somewhere between interfaces or endpoints).

Although logically the descriptors form a tree, the device usually returns them in a serialized manner. For example, interface and endpoint descriptors can be retrieved only as a part of a configuration descriptor they belong to. The ordering in the serialized form implies the logical tree it represents.

USB was designed with flexibility of offered functionality in mind but to prevent total chaos, devices (or rather their functionality) were divided by common attributes into so called *device classes*. Each class deals with devices of certain kind, e.g. HID devices, printers, scanners, audio or a special vendor class for any device that would not fit into any other one. Some classes define *subclasses* for fine-grained resolution (e.g. HID class offers subclasses for keyboards and mice).

The class is reported as a part of device and interface descriptor. That is because a single physical device may provide functionality of several classes simultaneously, depending which interface the software communicates with. In such cases the class reported through the device descriptor is a special value with meaning 'no class, see interface'.

1.4.6 Bus Enumeration

USB was designed with possibility of hot-plugging and thus it has to have well-defined actions when a new device is added. The following is a simplified sequence of what all the involved drivers and devices must go through. The setup is that new device is attached to some hub that is already initialized and configured.

1. The hub (the hardware) detects change on one of its ports.
2. The hub waits some time to allow electrical current stabilization.
3. The hub informs the driver that a change occurred.
4. The hub driver queries the hub for exact kind of change (whether device was added or removed, etc.).
5. The hub driver requests reservation of default address (to prevent having more devices listening on the same address).
6. The hub driver tells the hub to enable the port.
7. The port is enabled (i. e., signalling is open), the driver then must wait again for stabilization of currents.
8. The driver queries the device for its device descriptor and assigns a new address to it.
9. The driver can release the reservation of the default address.
10. Based on the values of device descriptor, proper driver for the new device is found and started.

The new driver is then told the address of the device and it is up to it to configure and finish the initialization of the device. As stated above, the sequence is simplified and due to errors in devices (not all USB devices obey the specification exactly) some steps (e.g. setting new address) have to be tried several times or in a specific order.

1.4.7 USB Communication

Packets, Transactions, Transfers

The communication on USB bus is *packet* oriented. Every piece of information sent to device or host is translated into packet form before it reaches the bus.

However, packet level is not accessible to the software and drivers are not able to send separate packets. The smallest communication entity available is called a *transaction.* These differ between various USB versions and therefore won't be described here. See USB specifications for different transaction types of each USB version.

Requests handled by host controller drivers are called *transfers.* One transfer usually corresponds to one command to the device. Transfers may consists of several transactions. USB specification distinguishes the four types of transfers already described above: control transfers, interrupt transfers, bulk transfers and isochronous transfers.

These transfers have their specifics which are not exposed by the HC. Therefore, the only transfer type requiring special behavior is control transfer type.

Control transfers consist of three stages:

- **SETUP stage**: uses setup transaction and identifies command to the device
- **DATA stage**: optional, the direction (IN/OUT) and its presence depends on the command sent in setup stage

- **STATUS stage**: direction used in STATUS stage is opposite to that in DATA stage. If data stage was not present direction of STATUS stage is IN.

Control transfers must complete all stages without being interrupted by another control transfer to the same endpoint. If two or more control transfers interleave, it will result in failure of the interrupted transfers.

Bus Access Constraints

Different transfer types have different bus access limitations. Periodic transfers (isochronous and interrupt) are limited to 90% of the available bandwidth. Interrupt and isochronous communication happens regularly (hence periodic) and maximum requested bandwidth has to be reserved in advance. Failure to reserve the bandwidth results in an error. The driver may then choose another interface setting or it can report an initialization failure.

At least 10% of bandwidth has to be available for control transfers. It might be less if there are not enough control transfers pending.

Bandwidth that is left unused after periodic and control transfers requests were satisfied is used by bulk transfers. Bulk transfers work in best effort mode and if there is no available bandwidth they are postponed.

In USB version 2 and lower, It is the role of host controller driver scheduler to guarantee bus access constraints.

1.4.8 USB 3 Features

Bus Duality

The USB 3 bus and USB 3 devices are backwards compatible with USB 2. As the protocol is completely different this was done by splitting the bus into two parts. The physical implementation is best shown in Figure 1.1.

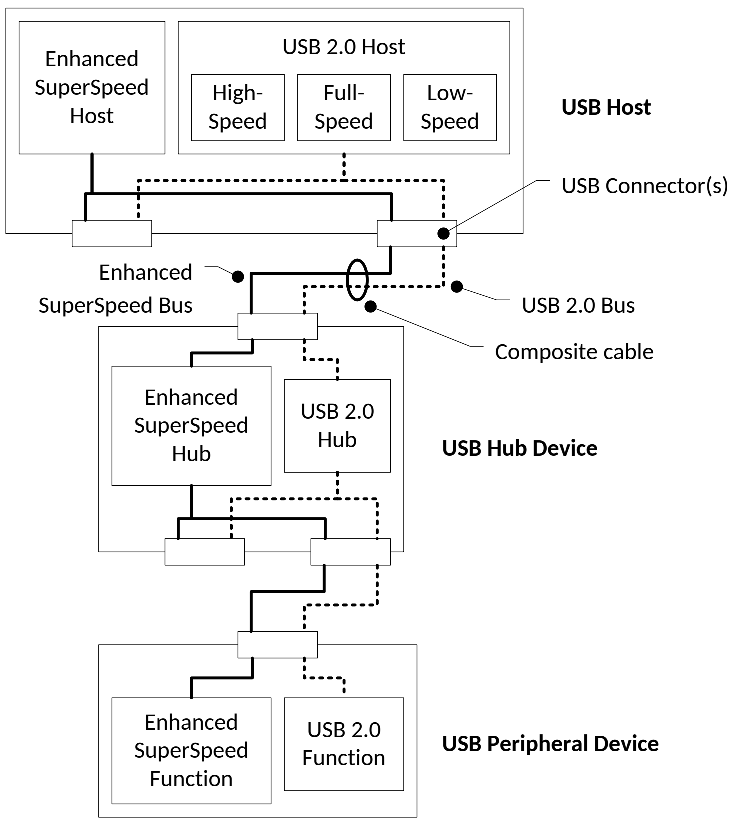

Figure 1.1: Dual bus architecture, (Figure 3-1 from [**usb3**])

There are three cases of a pair of devices connecting. The simplest one is a USB 3 device/hub connecting to a USB 3 hub. Both devices handshake and use only the Enhanced Superspeed Bus. When a USB 2 device is plugged into USB 3 hub, the hub will recognize it, and use only the USB 2 bus. The last case, USB 2 device connecting to USB 2 hub, follows the specification of USB 2.

The key idea is that every USB 3 device is essentially two devices with a shared port – an Enhanced Superspeed device and a USB 2 device, of which only one is active at the same time. The specification doesn't require the device to have the same function in both USB 2 and USB 3 modes. For example, the USB 2 function may be used to only describe that the device can only operate in USB 3 mode.

Hubs are a notable exception from the single-mode rule as they are actually two independent hubs in one box. The inner USB 2 hub connects as an USB 2 device and the USB 3 hub connects as a USB 3 device. When a downstream device is connected physically it is decided to which inner hub it will be connected based on the device actual speed.

You can see that USB 3 sort of breaks the rule of backwards compatibility. The protocol is not backwards compatible on its own, but the specification forces all USB 3-compliant devices to be USB 2-compliant as well. The majority of compatibility breaks are on the hardware level though, the protocols are very similar from the perspective of drivers.

Differences from USB 2

USB 3 introduces several new features used to speed up the communication and offer more bandwidth. Most of these features are hidden to the device drivers. They will be described in this section.

One of the new features USB 3 introduces is called bursting. An endpoint that supports bursting states its maximum data burst size in its SuperSpeed Endpoint Companion descriptor. When a transfer is initiated the sender may send multiple packets in a row before it has to wait for an explicit handshake. This eliminates the wait time and improves the communication efficiency.

Unlike USB 2 and previous versions where the bus was half-duplex, the bus in USB 3 is full duplex. This means that a device using USB 3 may use both IN and OUT transactions concurrently.

The devices on USB 2 bus communicate using broadcast. For OUT transactions, every hub broadcasts the packet to all enabled downstream ports and only the targeted device should accept the packet. In USB 3, the communication is routed directly to the receiver. This introduces the need for route strings, which describe the ports of the hubs on the way to the device.

In USB 2, the controller is required to poll the device to offer it an opportunity to finish an interrupt transfer. Using the full-duplex links, USB 3 introduces asynchronous notifications to reduce the overhead of polling.

At last but not least, USB 3 introduces more complex power management features. For example, an USB 3 device may initiate link power state change at the device while in USB 2, this could be only initiated by the host.

1.5 Existing USB Subsystem

As mentioned in subsection 1.2.1, HelenOS already supported UHCI, OHCI and to some extent EHCI.

The original documentation of the HelUSB project described the state of affairs as the project ended. Since then, several improvements in both USB support and in HelenOS driver framework in general were made, for example new USB drivers like mass-storage were added. Unfortunately, these changes are pretty much not documented anywhere. In this section, we will focus on the things that changed since the documentation was written. We do not aim to replace the original documentation of HelUSB project.

By all means, information in this chapter is written with regard to the state before our *project was implemented. Thus, great part of the information given in this section is already obsolete, but it's needed to assess the damage we're personally responsible for.*

The USB framework defines two classes of drivers – the host controller drivers and USB function drivers. For the first class, there is a library called `libusbhost` that aids in providing the unified interface of the host controller to USB function drivers, and also implements common HC functionality. There are four HC drivers at the moment:

- **VHC**, Virtual Host Controller. Implemented in the early phase of HelUSB project, served probably as a dummy backend to allow better work parallelization and debugging.

- **UHCI**, Universal Host Controller Interface driver. The earliest interface supporting speeds of USB 1.0: Low- and Full-speed devices. Important for running HelenOS under QEMU, as it's the interface of the default HC QEMU emulates. Apart from isochronous transfers, the driver covers all functionality UHCI provides.
- **OHCI**, Open Host Controller Interface driver. Somewhat complete, yet a bit simplified, especially in terms of transfer scheduling. Does not care about the polling interval, but schedules all interrupt transfers on every frame. Isochronous transfers not supported.
- **EHCI**, Enhanced Host Controller Interface driver. Mostly a copy of OHCI driver, as it uses similar structures. Shall support USB 2 speeds, but the support is very limited – the driver cannot use High-speed hubs to communicate with Full- and Low-speed devices, as the support for Transaction Translation is completely broken. Also, the bandwidth accounting is not implemented for High speed. Neither this controller supports isochronous transfers.

The HC driver is no longer split in half (as the project documentation states), but all HC drivers emulate a virtual hub that is driven by a regular `usbhub` driver. The tree physical topology of USB is kept only inside the HC driver, and is presented flat to the Device Driver Framework – all USB devices are child functions of the HC driver. They communicate with each other through the DDF driver interface called `usb_iface`, which contains all methods various drivers use.

When the driver enumerates a USB function, it is usually taken care of by the `usbmid` driver. This driver scans the device descriptor for provided interfaces, and creates child functions for them. These functions are then driven by class drivers. Notable exception being the `usbhub` driver, taking care of the device directly, as hubs are not allowed to have multiple interfaces.

The USB function drivers are well supported by the `libusbdev` library. This library builds an abstraction layer above `libdrv`, used by other drivers directly, to better suit the needs of USB devices. It does a complete initialization of the USB device, including initiating a separate IPC connection to the HC driver directly – to avoid bouncing all operations in the `usbmid` driver. For this purpose, the `usb_iface` contains two methods: `get_my_iface` and `get_my_interface_handle`. The first one is answered by the `usbmid` driver with the number of interface driven, or with a value of -1 by the HC driver if the driver serves the whole device. The `get_my_interface_handle` call is recursive, until it reaches the HC driver – which answers it with devman handle of the USB device function. The device driver then uses it to initiate connection to the HC driver, the same way as the usbmid driver does.

Although this is sort-of hacky solution (the devman handle is supposed to be private), it is currently the only one. Ideally, the drivers would use a special method to let new connection forward to the HC driver, but for complicated reasons, this does not work as expected. We discussed this with the current HelenOS developers, and they confirmed us that the issue is still not solved.

The `usbhub` driver uses another four methods defined by the `usb_iface`. The interface methods `reserve_default_address` and `release_default_address` ensure synchronization across multiple hubs (possibly across multiple hub drivers), as the software must ensure that only one device is listening on the default address at the same time. Then, `device_enumerate` and `device_remove` announce that a device is connected to (detached from) the hub, to be enumerated (removed) by the HC driver. The interesting part is that the hub driver has no access to the created device, as the logical topology presented to the DDF is flat.

All USB device drivers specify the endpoints they expect from the device in a form of a static description, which they pass to the `libusbdev` library during driver initialization. Once a new device is added, the library fetches the device descriptor and matches available endpoints against the specification provided by the driver. Then the library creates *pipes* – an abstraction of endpoints based on their properties, not their exact numbers, which are usually implementation defined. Pipes are then used by the driver as, well, pipes to push data through and read data from.

The pipe creation process and their usage define the last four methods of which the `usb_iface` is comprised of: `(un)register_endpoint`, `read` and `write`. The endpoint (un)registration informs the HC driver about a pipe creation/disposal, and `read`/`write` methods are used to actually transmit packets. Note that the interface is unified regardless of the transfer type used by the endpoint.

As for the drivers available, there is a solid support for USB HID devices, implementing keyboards, mice and multimedia keys. Also, a driver for USB Mass storage exists and somehow works, despite several warnings and errors printed to the log. Also, a fallback driver is provided to handle any USB device, to enable the device examination for devices without their own driver, mainly for debugging purposes.

Not to forget, there are two userspace utilities related to the USB stack. One of them, `mkbd`, is not so important, as it is used only to demonstrate functionality of multimedia keys HID driver. The other one, `usbinfo`, can be used to list available USB devices:

```
/ # usbinfo -l
Bus 37: /hw/pci0/00:04.0/ctl
        Device 61: /hw/pci0/00:04.0/usb1-fs
        Device 65: /hw/pci0/00:04.0/usb2-ls
```

Other use-cases for this utility include descriptor dumping and examination of device status.

It needs to be said that whole structure of the USB framework (and also the DDF in general) expects the drivers to behave correctly and does not implement any countermeasures against malicious behaviour of drivers. For example, the `usbinfo` utility connects directly to the HC in the same way as the device driver does, and fetches the device descriptor. In fact, any other task can communicate directly with any USB device. Or, any driver can call the interface methods designed for hubs only – for example, it can reserve the default address and never release it. Due to the experimental and in-development nature of HelenOS, this is not an important problem. Yet, it is an obstacle to solve before HelenOS will be ready for "normal" users, and it will be a tough one.

Another thing related to the whole USB stack is that support for device removal is in fact non-existent. At the time of HelUSB project, there was no support for device removal in the Device Driver Framework, so it's not surprising that the USB framework inherited this. There are attempts to terminate interaction and release resources in case of repeated communication errors.

Chapter 2

xHCI Stack Implementation

The structure of the xHCI driver is quite straightforward, as it tries to fit into the scheme of how hardware and the rest of HelenOS works. We decided to use the existing library `libusbhost` to reduce code duplication with other HC drivers. It came out that this library needs a lot of changes to support us in this goal, but that's for chapter 3.

The USB host controller driver using `libusbhost`, xHCI included, serves as a connecting layer between the hardware and the library, and exposes its bus interface.

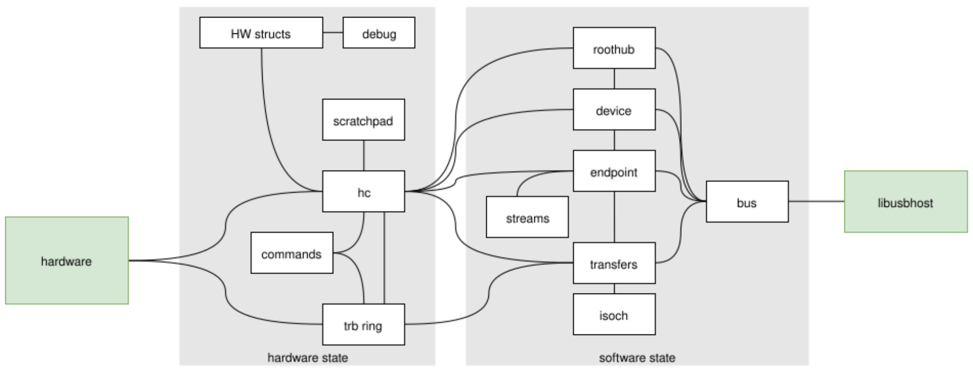

Figure 2.1: The modules of xHCI driver

The scheme is not at all strict, we're in a C world, there are dependencies almost everywhere – take it as an informal overview to get an idea. The whole driver can be split into two parts. The left one takes care of the hardware perception of what's going on, the right one is about managing the software structures and memory. We start with describing the modules in the hardware part, as their functionality is clear. Their order follows the order in which they were implemented.

2.1 Hardware Structures

The `hw_struct` directory contains C structures that represent registers and control structures used by the xHC. The memory layout of the structures is defined by the xHCI specification[xHCI] and thus the files in this directory should be mostly treated as read-only.

2.1.1 Registers

The register structures (defined in `regs.h`) represent hardware registers presented by the xHC to system software implemented as Memory-Mapped I/O space. They can be divided into four categories as follows:

Capability registers These specify read-only limits, restrictions, and capabilities of the specific xHC implementation used.

Runtime and Operational registers These specify the current xHC configuration and runtime modifiable state.

Extended capabilities These specify optional features of the particular controller.

Doorbell array An array of up to 256 doorbell registers, which supports up to 255 USB devices or hubs. Each doorbell register provides the software with a mechanism for notifying the xHC if it has slot or endpoint-related work to perform.

In our implementation, all of these can be accessed through the `xhci_hc_t` structure and can be modified through it using the register handling macros defined in `regs.h`. Note that not all register bits can be manipulated freely by the system, some impose restrictions:

RO, Read-only Register bits are read-only and cannot be altered by system software. An example is the *CAPLENGTH* register (see [xHCI], Section 5.3.1).

RW, Read-Write Register bits are read-write and can be altered by system software. An example is the *USBCMD* register (see [xHCI], Section 5.4.1), in which some bits or bit ranges are RW (and some are read-only).

RW1C, Write-1-to-clear Register bits indicate status when read, a set bit can be cleared by writing a '1', writing a '0' to such register bit has no effect. An example is the *Event Interrupt* bit in the *USBSTS* register (see [xHCI], Section 5.4.2) which is set to '1' by the xHC when an interrupt occurs and can be cleared by the driver by writing '1' to it once the event is scheduled for handling.

RW1S, Write-1-to-set Register bits indicate status when read, a clear bit can be set by writing a '1', writing a '0' to such register bit has no effect. Examples are the *Command Stop* and *Command Abort* bits of the *CRCR* register (see [xHCI], Section 5.4.5).

This is not an exhaustive list of access attributes, for the entire list, see [xHCI], Section 5.1.1.

Register Access Macros

Registers are accessed very often in all hardware-related modules. We felt that there is a need to centralize the information defined in the specification, especially the subdivision of registers to individual fields. There are several common solutions to this problem.

Probably the most common one, which we also considered at first, is defining two constants for every such field: mask and shift. When one needs to read a field, they read the value of the register, use the mask to select bits, and then shift the value according to the shift constant. To write a field, one reads the value of the whole register, uses the bitwise negation of the mask to copy surrounding bits, then shift the value to be written into its place. This solution is simple to understand, yet hard to use correctly. There's a lot of repetition, the more if you consider endianity (HelenOS is targeted also on Big Endian platforms, while the USB world is Little Endian).

Another possibility is to define macros for reading and writing every single field. That idea was discarded in its very beginning. We wanted a solution that requires only one definition per field, cannot be used in a wrong way and is sensibly short to write. So we came up with the register access macros. The best introduction is probably by an artificial example:

```
#define XHCI_SOME_FIELD             usbreg, 32, RANGE, 13, 7

unsigned field = XHCI_REG_RD(hc->op_regs, XHCI_SOME_FIELD);
XHCI_REG_WR(hc->op_regs, XHCI_SOME_FIELD, 42);
```

Listing 2.1: On the first line, we read bits 13 to 7 of the field `hc->op_regs->usbreg` to a variable, and then change the same bits in the register to a value 42.

All the definitions of macros like `XHCI_SOME_FIELD` relevant for xHCI are contained in the header file `hw_struct/regs.h`. The definition contains all the information necessary to access the field. It says that the register field is contained in a field `usbreg` of an operational register structure (the one `hc->op_regs` points to), the structure field is a 32-bit wide dword, and that the register field is contained in bits 13 to 7 of it.

The primary principle used to implement these preprocessor macros is the specific order of macro expansion in C. In the example, the register definition macro is used as an argument to the `XHCI_REG_RD` macro. Both macros are expanded in a breadth-first fashion, producing just another preprocessor macro `XHCI_REG_RD_INNER(hc->op_regs, usbreg, 32, RANGE, 13, 7)`. You can see that one argument is expanded to several arguments for the inner macro. What happens next is pretty simple. The `RANGE` argument token is glued to a prefix, producing a name for another macro, which selects between implementations for whole fields, bit ranges and individual bit flags. The `XHCI_REG_RD_RANGE` then extracts the specified bits read from the whole field. The size argument is needed to properly handle endianity. All other top-level macros (`XHCI_REG_WR`, `XHCI_REG_SET`, `XHCI_REG_CLR`, `XHCI_REG_MASK` and `XHCI_REG_SHIFT`) operate on the same principle.

We think we have achieved our goal. These macros are a bit hard to understand but very easy to use and require just one line of definition per field. Looking back though, the work was probably not worth it – the registers are not used that much to justify the existence of register definition of every single register field. But it is already done and shall there be a need to access more registers, it's easily accessible without thinking how to select the proper bits and ensure the correct endianness. And even if there wasn't, it serves as a nice showcase of what are preprocessor macros capable of.

2.1.2 Contexts

Contexts (defined in `hw_struct/context.h`) are control structures that represent devices and their configuration as well as the parameters of the communication between the xHC and system software. The `xhci_hc_t` structure contains the *Device Context Base Address Array* (DCBAA), which holds up to 255 pointers to device contexts at indices 1 through 255 and a pointer to the scratchpad array at index 0 (see 2.4). Each device context contains a slot context (used to describe the device as a whole, represented by `xhci_slot_ctx_t`) and 31 endpoint contexts (represented by `xhci_ep_ctx_t`).

Most of these contexts will be described in more detail in the following sections.

2.2 Debug

Since both the internal state of the xHC and its capabilities are often described by a handful of bits located in a packed 32-bit register, we needed a way to monitor these values in a human-readable form. The `debug.h` and `debug.c` files contain a set of register dumping functions that use the register reading macros described in the previous section to print the values of all the bit sets contained in a register to the driver's log. They also contain auxiliary functions that are used to convert numeric codes to their meaning in a string form and functions that dump the contents of a hardware structure (such as `xhci_endpoint_ctx_t`).

These functions have proved to be of great use and should any future maintainer of the HelenOS xHC stack find themselves in a bug-ridden situation, putting these functions to the areas of code they suspect of mischievous deeds might be a good starting point.

2.3 TRB Ring

One of the primary means of communication with the HC, apart from the register interface, are the TRB rings. Their management is fully contained in the TRB ring module. Before we dive into the implementation details, let us first describe how these rings work.

2.3.1 Architectural Overview

Transaction Request Block, everywhere else abbreviated as TRB, is a fixed-size structure with contents of variable type. The most notable exception being a *TRB Type* field. Each of the 64 TRB types defines the other fields that are contained inside a TRB. Most commonly, it just contains a pointer to another buffer together with the size of the buffer, or a result of an operation.

TRB Ring is then a circular queue comprised of individual TRBs. The main difference from older HCIs is that TRBs are required to be contiguous in memory, forming a *TRB Ring Segment*. It is, of course, needed to link the two ends of the segment together, one must use a special TRB type called *Link TRB*, which serves as a glue between the end and the beginning of a segment. It is not required from the ring to be composed of only one segment, but then special precautions must be taken.

TRB Ring is used in many instances during the lifetime of the Host Controller. There are two special instances though: the *Command Ring* and the *Event Ring.* The command ring serves as a channel to command the HC, and the command interface is described in more depth in section 2.5. The event ring is asynchronously read to retrieve results of commands, completed transactions and so on, as the primary information channel from HC to host. Actually, there can be more than one instance of Event Ring, but that requires the host to be able to recognize multiple interrupters, which is not supported in the current version of HelenOS. The event subsystem is further discussed in the section 2.6.2. The third type of a ring, a *Transfer Ring*, is used for every pipe endpoint on the bus - either Endpoint or Stream. These are used by the Transfer module, described in section 2.9. For now, let's discover the mechanics of the rings themselves, regardless of their type.

As already said, TRB ring is a circular queue. It always has one producer and one consumer of TRBs. Every ring implicitly defines two pointers: the enqueue pointer and the dequeue pointer. The producer uses the enqueue pointer to enqueue TRBs and the consumer uses the dequeue pointer to dequeue them. The pointers are not shared between the host and the Host Controller. Instead, every TRB has one bit with special semantics, the *Cycle Bit.* The transition between values of TRB Cycle Bit of individual TRBs defines the value of Enqueue pointer. An example situation is shown in the figure 2.2.

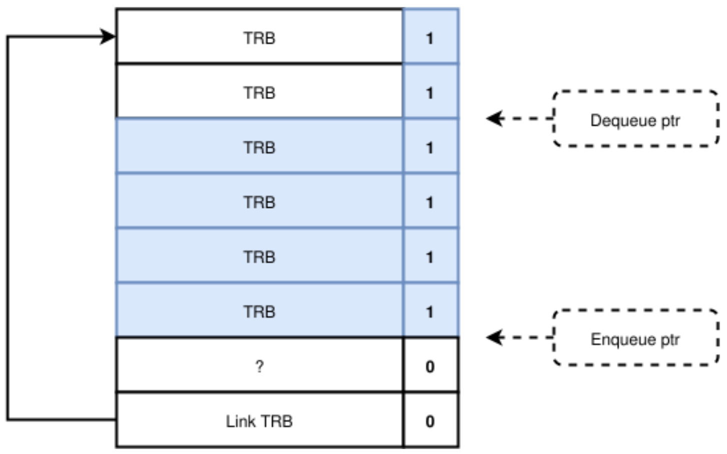

Figure 2.2: A simple example of a TRB ring.

It is important to understand that the consumer cannot write to the ring and thus cannot change the value of the Cycle Bit after it processes a TRB. Therefore, the position of the dequeue pointer must be signalized by other means. Furthermore, there has to be a mechanism to stop the consumer from considering the TRBs in the beginning of the segment as enqueued, after it wraps around the segment boundary. The enqueue pointer is not defined by a transition from one to zero, but more generally by a transition from *Producer Cycle State*, `PCS`, to `~PCS`. This bit is kept in the memory of the producer. Similarly, there exists a *Consumer Cycle State*, which has the same value as `PCS` had in the moment when the value of the enqueue pointer was the same as the current value of the dequeue pointer.

The Link TRB contains a *Toggle Cycle* flag, which if set, instructs both producer and consumer to toggle its cycle state when this TRB is enqueued/dequeued (Link TRBs are not necessarily fixed in place, they are queue members as any other TRB). There shall always be an odd number of Link TRBs with the Toggle Cycle flag set on the ring.

We haven't yet covered how the producer can be aware of the dequeue pointer value. In case of Command and Transfer rings, the producer is the Host Controller. Every finished command or transfer is denoted by an Event TRB placed on the Event Ring. This Event TRB usually contains the physical address of the TRB on the respective Command or Transfer ring. This way the producer knows that the dequeue pointer value is not less than the reported value. Similarly, when the host processes an Event TRB on the Event Ring, it shall report the physical address of the TRB processed by writing the value into a dedicated register.

Also, the Event ring differs a bit in the usage of multiple segments. As the Host Controller is obviously unable to allocate ring segments and map them into virtual address space of the driver, and it's even not possible to allow the host to write Link TRBs onto the ring, which is otherwise read-only for the host, a different approach has to be used. Instead of the segments being linked together, the host preallocates a number of segments and an *Event Ring Segment Table*, which is another structure defined by the specification. The segments' addresses and sizes are written into the table, and the address of the table is written to a dedicated register. At the time of the write, the ownership of all the buffers is transferred to the Host Controller.

2.3.2 Implementation

As the requirements imposed by the specifications are quite strict, there's not much freedom in the implementation. However, there were some decisions to be discussed.

First, we decided not to restrict ourselves to single-segment rings as other existing implementations do, though we haven't implemented runtime scaling yet. Second, we implement and use three types of TRB rings; the third one being a simplified TRB ring used by the host from both sides. Its purpose is better described in the section 2.6.2. The three implementations are somehow separate, but because they share principles, they reside in a common module.

The containing structures are called `xhci_trb_ring_t`, `xhci_event_ring_t` and `xhci_sw_ring_t` for Command/Transfer rings, Event ring, and host-only rings, respectively. Despite their name, these structures only represent the ring in host memory and are not the structures given to the hardware. Let's walk through them one by one.

Command and Transfer Rings

These rings are prepared to be multi-segment. They contain a list of `trb_segment_t` structures, which represent one ring segment. The segment itself is allocated as a memory accessible for DMA of size `PAGE_SIZE`. The space for TRBs is in the beginning to ensure alignment. At the end, there is a small footer keeping the physical address of the segment and a link to the segment list. There are helper functions to manage segments.

The main runtime interface to rings of this kind is the function `xhci_trb_ring_enqueue_multiple`, used to enqueue a contiguous array of TRBs (referred to as Transfer Descriptor, TD) onto the ring. As this function must not block to be usable in atomic contexts, it first performs a dry run to check if there is enough space on the ring, then rollbacks and actually emplaces the TRBs. Because it might be needed to interleave the Transfer Descriptor with multiple Link TRBs, it is not possible to simply calculate the free space in advance.

Because the caller of this function needs to know the physical address of the TRB enqueued to pair it with the completion event, the address is assigned to an output argument. If there are multiple TRBs to be enqueued, the one with *Interrupt On Completion* flag is reported – because that's the one that will generate the event.

The second crucial part of the interface is the function `xhci_trb_ring_update_dequeue`, by which the dequeue pointer advancement is announced to the ring. As the rings need to be configured to the hardware somehow, there is also a function called `xhci_trb_ring_reset_dequeue_state`, which is used to reset the ring and also return a value used to configure the ring in the HC – a physical pointer mixed with the initial Consumer Cycle State.

Event Ring

The implementation of the event ring is fairly simple, with the main interaction point being the function `xhci_event_ring_dequeue`. This function, symmetrically to the TRB ring enqueue, does not block if the ring is empty, but returns `ENOENT` instead. This structure also contains the Event Ring Segment Table.

Software Ring

This structure is to be used as a buffer for exchanging information between different fibrils of the driver. It is an ordinary implementation of a circular blocking queue made for TRBs. It uses the Cycle bit inside TRBs to indicate active entries, because both sides are allowed write to the ring when the guard is locked. Also, this ring is not allocated using DMA buffers, because it is not expected to be accessed by hardware.

2.4 Scratchpad

Scratchpads are buffers that an xHC implementation can request from the system software for its internal needs. The size of these buffers is specified in the *PAGESIZE* register found in the operational register set defined in xhci_op_regs_t.

The amount of buffers requested by the xHC is specified in the *Max Scratchpad Bufs Hi* and *Max Scratchpad Bufs Lo* registers of the second set of structural parameters (*HCSPARAMS2*, see [xHCI], Section 5.3.4) defined in xhci_cap_regs_t.

The allocation of these buffers takes place as part of the host controller initialization, specifically in hc_init_memory, which calls xhci_scratchpad_alloc. The pointers to these buffers are then passed to the xHC in the *Scratchpad Buffer Array*, pointer to which occupies the first index of the *Device Context Base Address Array* (dcbaa field of xhci_hc_t).

Our implementation originally implemented these as a standalone structure xhci_scratchpad_t, which served mainly for the purposes of resource management by keeping both the physical addresses for the xHC and the virtual addresses for deallocation. This was, however, later refactored to use dma_buffer_t, which is part of libusb and was created for the same purpose.

Once the xHCI driver finishes its execution, the scratchpad buffers are deallocated by a call to xhci_scratchpad_free as part of hc_fini.

2.5 Commands

The xHC offers an independent command interface. During operation, the xHCI driver uses this interface to manipulate device slots, devices and endpoints by executing various commands provided by a unified command subsystem. This section provides details on the structure and implementation of this subsystem.

2.5.1 Execution Workflow

The xHC command interface consists of a TRB command ring and a command doorbell register. Commands are executed by placing various command TRBs onto this ring, forming a *Command Descriptor*.

After placing the respective TRBs and writing to the xHC command doorbell register, the descriptors on the command ring are sequentially processed by the xHC, resulting in either failure or successful completion. The result of every command descriptor is reported back to the xHCI driver in the form of a *Command Completion Event* placed onto the primary event TRB ring.

After writing into the xHC command doorbell register and before receiving the respective command completion event, the xHCI driver can attempt to abort the issued command. Such action might be of use for instance if the command completion event does not arrive within a set time period.

Note that this section intentionally omits hardware technical details, which are not instrumental in understanding the command subsystem. For further hardware documentation of the xHC command interface, refer to [xHCI], Section 4.6.

2.5.2 Structure

The xHCI driver command subsystem instance is represented by the xhci_cmd_ring_t structure which exists throughout the entire duration of the xHCI driver's lifecycle. The purpose of this structure is to maintain and manage the command TRB ring and to keep track of enqueued command descriptors.

Individual command descriptors are represented by the xhci_cmd_t structure. In this structure are stored high-level parameters of the command as well as the command TRB, which is placed onto the command ring when the command is executed. While the high-level command parameters are kept directly in this structure, the hardware-related internals are kept in a substructure, which is commonly referred to as *command header*. The purpose of this separation is to stress that the header contents are to be exclusively accessible to the command subsystem, while the rest of the structure remains accessible to the entire xHCI driver.

Besides data structures, the command subsystem offers a centralized command completion event handler function – xhci_handle_command_completion – which is called by the event subsystem in case a *Command Completion Event* is encountered.

The last major component of the command subsystem are functions used to generate and schedule commands on the xHC. These functions produce valid instances of the xhci_cmd_t and place their respective TRBs onto the command ring managed by the xhci_cmd_ring_t structure, requesting either blocking or non-blocking semantics for waiting on their completion. These functions are described in detail in the next section.

2.5.3 Command Lifecycle

Issuing Commands

By design, the internal logic of the command subsystem is kept opaque with respect to other components of the driver. A notable example of this is the mechanism for command scheduling.

Commands are usually issued by the HC component of the xHCI driver. At the time of issuing, the information required can be broken down into three groups:

Command Type One of the 15 commands supported by the xHC command interface.

Command-specific Parameters The number, type and semantics of these parameters depend on the specified command type.

Completion Semantics This is the desired behavior of command execution.

In the *blocking mode*, the call to issue the command will block the calling fibril until the completion event is received.

On the other hand, in the *non-blocking mode*, the fibril will only be blocked until the command is issued – the command subsystem will take the ownership of the command and deallocate it when the completion event arrives. This mode is meant for commands which do not require completion event handling and involves more complicated memory management since the command subsystem is responsible for freeing the command after the completion occurs.

Depending on the desired completion semantics of the issued command, the HC component calls either the `xhci_cmd_sync` or `xhci_cmd_async` function and passes it a configured instance of the `xhci_cmd_t` structure. Upon such call, the command subsystem will execute an internal command handler function, which copies the high-level command-specific parameters configured by the issuer, and use their values to construct a command descriptor consisting of TRBs.

At this point, the `xhci_cmd_ring_t` structure is modified and the new command descriptor is placed onto the TRB ring. The corresponding instance of `xhci_cmd_t` is added to the active command list and depending on the completion semantics, the issuing fibril is either suspended or continues execution regardless of the completion event.

Handling Completion

When a command is completed, a *Command Completion Event* is generated by the xHC. This event is detected by the event subsystem, which in such case triggers the `xhci_handle_command_completion` function.

This function extracts the address of the completed command descriptor and uses it to find a matching instance of the `xhci_cmd_t` structure in the active command list.

Depending on the desired completion semantics, the command subsystem either wakes the sleeping issuer fibril, or discards the non-blocking command from the memory.

Aborting Commands

The command subsystem defined by xHCI contains a mechanism to trigger early command abortion. It is not guaranteed to be effective for all commands, just for commands control of which is out of HC's hands. A good example might be the `SET_ADDRESS` command, which issues the USB control request *SetAddress*, and waits for the device's response. Such command blocks until the response is received. If the software, for any reason, wants the command to be aborted, it can write 1 to the `CA` bit of the *Command Ring Control Register.*

When the HC is triggered by a command abortion, it places a *Command Ring Stopped* event on the primary event ring and halts command processing. The ring can be started again by ringing its doorbell. If a command was actually aborted, a corresponding *Command Completion Event* with status *Command Aborted* is placed first.

We decided to set a fixed timeout for every command, regardless of its possibility to block, arbitrarily chosen as 10 seconds. That gives a generous amount of time for the HC to finish a command. When this timeout is over, a command is aborted.

Here comes the catch: the driver is not able to choose which command is to be aborted – it is always the one currently processed. Usually, it's the one that blocks the pipe, but it means that the driver cannot simply place a timed constraint on a command execution. To reflect the hardware semantics, all fibrils that expire the timeout behave the same: they trigger a command abortion, wait for the command ring to be stopped, then start it again.

To orchestrate several fibrils trying to schedule new commands, waiting for commands to be completed and also the one handling an interrupt, a simple state machine with synchronization is incorporated. In case the command ring is being restarted, newly issued commands are delayed to prevent the doorbell from being rung unexpectedly. When there are multiple fibrils with an expired timeout while the abort is already triggered but not yet finished, they retract and wait for the full timeout period again.

That common execution path is enclosed within a function called `try_abort_current_command` (static for the command subsystem). As it's hard to force a command to stall, we haven't had many opportunities to test the behavior. Usually, the mechanism was triggered by a deadlock in our code, which didn't magically disappear when a command was aborted. Also, the commands always finish on time in a virtual environment. At the time of writing this documentation, there has been only one observed legitimate occurrence of a command stall on real hardware. The abort mechanism did its job well that day.

2.5.4 Usage Examples

Since the command subsystem is used at a multitude of places in the HC component, it has been the goal of the authors to make its usage elegant and effortless. For that reason, a dedicated inline notation syntax powered by preprocessor macros has been devised and implemented. This is demonstrated in Listing 2.2.

```c
xhci_hc_t * const hc;

/* Issue a "Set TR Dequeue Pointer" command synchronously. */
const int result = xhci_cmd_sync_inline(hc, SET_TR_DEQUEUE_POINTER,
    /* Command-specific arguments use struct initializer. */
    .slot_id = slot_id,
    .endpoint_id = dci,
    .stream_id = stream_id,
    .dequeue_ptr = addr,
);

/* At this point, the command is completed with ‘result‘. */
```

Listing 2.2: Usage example of the xHCI driver command subsystem inline syntax. This snippet issues a *Set TR Dequeue Pointer* command to the HC in blocking mode. Note that the command initialization, configuration and finalization is handled by the inline macro syntax.

2.6 Host Controller Module

This module contains a mixture of things that didn't fit anywhere else or didn't deserve their own submodule. We tried to move the “xHCI specification quirks” here, so that other modules rely only on the semantics and overview of the xHCI, and don't care that much about the exact technical details. Of course, it's not strictly possible, because the whole driver is about technical details of xHCI.

2.6.1 Initialization

A substantial part of this module handles the initialization of the controller. When the xHC device is added, several steps need to be done. Let's walk through them in order. The DDF callback `dev_add` is handled by the `libusbhost` library, so the story begins there.

1. At the very beginning, the supplementary structures are allocated in the DDF device node. These structures accompany the entire lifetime of a driven HC.
2. The DDF control function is created. Through it, the user may command the HC driver.

3. The hardware resources such as MMIO space or IRQ number are obtained from the parent device (usually PCI driver).

   Now, the structure and hardware resources are handed out to the xHCI driver for the first time, to do its initialization.

4. The MMIO range needs to be mapped, and the proper register areas found. xHCI specifies several areas of MMIO registers, with variable offsets between them. We cache the pointers to the areas inside the `xhci_hc_t` structure to make the access convenient.

5. The roothub structures are initialized. The number of ports is obtained and memory for state machines is allocated.

6. Extended capabilities need to be parsed soon enough, as they contain some crucial information. Namely the information about legacy support, but also the protocol versions supported on individual roothub ports.

7. A lot of memory structures are to be initialized now. The *DCBA* array, event ring, scratchpads, command ring, the device-keeping array and also the event worker fibril.

   After all of that, the basic driver initialization is completed, and the execution returns to the library.

8. The interrupts need to be enabled if they are available. The driver is given an opportunity to generate the bottom-half IRQ handling code, and then the code is registered in the kernel. Shall any of these steps fail, the failure is not critical and the interrupts are just marked unavailable.

9. If the HC is being controlled by the BIOS (denoted by the extended capability), it needs to be claimed.

10. The HC is started. In case of xHCI, an initialization sequence as described in the specification is performed. The HC is reset to transition into a known state. The addresses of the memory structures are configured. If the interrupts are available, the interrupter 0 is enabled. Finally, the HC is started by setting a bit in the operational registers.

11. Before the control is returned to the library, all roothub ports are checked for a change, because the reset changed their connection state, but the Port Change Events had no ring to be written on.

12. If the interrupts are not available, a replacement polling fibril is started.

13. The roothub is to be set up. Other HCs using virthub must undergo an enumeration process, so this must be done after the initialization is fully completed. In the case of xHCI, this step is skipped.

A symmetric reverse sequence is performed to make the HC stopped again. This functionality is however not tested, because neither QEMU nor hardware that was available during development of our project supports PCIe hotplugging.

2.6.2 Events

Another functionality provided by the HC module is the first line of handling events. Events are the primary information channel from the device to the host. Every synchronous operation needs to be finished by waiting for an event.

The tricky part in handling events is, again, the synchronization. Event handlers must not issue operations which would wait for other events. The problem is that this event dependency is inevitable in some cases. Let us describe two scenarios which will explain the complexity of the final solution.

First, there are the Port State Changed Events. Handling these events usually involves device enumeration – that is however handled in a separate fibril. It's even not the reset completion which would create a deadlock – if it is expected, all the locks are unlocked. The main problem is a device disconnection while the enumeration is still in progress – when it happens, the event handling fibril must wait until the enumeration fibril terminates. But the enumeration might be in the later phase, in which it issues commands. In order to terminate, a command must be completed. That introduces a dependency, which cannot be simply removed. Because of it, Port Change Detected Events and Command Completion Events must be handled independently on each other.

Similarly, the enumeration process involves fetching descriptors from the device. That introduces another dependency, this time between Transfer Events and Port Change Detected Events. Neither these two can be handled sequentially.

The last dependency is between Command Completion Events and Transfer Events, forming a dependency triangle. This one might not be that obvious, but it is there and cannot be avoided easily. The handler of the Transfer Event needs to obtain a reference to an endpoint. To ensure coherency, the reference must be created while the endpoint is still registered, i.e. in a critical section. The same critical section that protects the endpoint unregistration – which needs to abort currently running transfers.

In order to avoid a deadlock, all three types of events need to be processed separately. We achieve that by processing only so-called fast events (Command Completion, MFINDEX Wrap) in the interrupt handler, and route the Port Change Events and Transfer Events to two Software TRB Rings (see section 2.3.2), from which two other worker fibrils dequeue and handle the events independently.

2.6.3 Commands Abstraction

Although the command interface has a module on its own, there are situations where the specific commands used are a technical detail hiding the actual semantics of the operation. To give an example, to inform the HC about dropping an endpoint, one must issue the *Configure Endpoint* command with a flag set. Also, some commands require an input context with a proper content, some don't.

The rest of the Host controller module tries to cover these implementation details and offer a more intuitive interface. Although the device and endpoint modules do fill the contexts on their own (being also a technical detail), the decision whether it's needed to fill an input context is left to the HC module.

2.7 Root Hub

The purpose of this module is very simple – take care of the root hub. Before we explain how our roothub works, let's have a look at why and how other HC drivers handle it.

The main problem a hub driver faces is synchronization one. When a new device is detected, it needs to be enumerated. Enumeration process in the case of USB 2.0 devices requires the hub to reset the port the device is attached to in order to move it to the *Default* state when it's listening on the default address 0. When the port reset is triggered, the hub must wait until the port reset is complete. During the whole process the device can be disconnected and the port reset will never be completed. Furthermore, the completion of the port reset is indicated by the same means as port connection, resulting in a deadlock in the naïve solution. The proper solution therefore involves spawning new fibrils and nontrivial synchronization.

All four HC drivers (UHCI, OHCI, EHCI, and VHC) are using a virtual roothub. In principle that means that they create a virtual USB device, which is listening on the default address in the beginning, and triggers an enumeration process. Then there is a little branch in transfer scheduling, that takes transfers directed to the same address as the virtual device's, and delivers them by calling a function instead. The virtual device behaves exactly like a real USB hub would – it has its standard descriptors, replies to setup requests and so on. So it enumerates like a USB hub would and creates a DDF device, which is taken by the `usbhub` driver. The driver then handles the virtual roothub like any other hub – by sending USB control transfers. The virthub module translates USB transfers to callbacks, which are implemented by the individual HC drivers to read and modify register space of the Host Controller.

This solution is very clean in design, regarding that root hub functionality is exactly the same as any other hub's, but just controlled by MMIO mapped registers instead of USB packets. It is even recommended by the USB 2.0 specification [USB 2.0] to embody this solution. But that's pretty much the only advantage this solution has. One has to write a lot of code to even implement the callbacks, not mentioning the virthub module itself. And a lot of code always comes with bugs. Also, while having no real performance impact, it requires several context switches, IPC calls, bouncing memory buffers and a lot of unnecessary allocations to clear one bit in the register space (USB hubs operate by setting and clearing so-called Features, indicating e.g. that a connection on a port changed). But it needs to be stressed more that this performance impact is mitigated by the fact that real hubs use real USB transactions above that and that hub interaction is very sparse.

We searched for a solution that would keep the cleanness in terms of shared functionality between usbhub driver and roothub and decided to move the port state machine and related fibril synchronization into the USB library. That cleanly separates the hardest part of handling hub port changes from the code that is actually handling them and enumerating the devices. This state machine is used by both the xHCI roothub and also by the rewritten usbhub. More information about this new module can be found in section 3.4.

One more thing related to the xHCI roothub, which we crossed while debugging on real hardware: the USB hub has a bit dedicated to indicate that a port is enabled (PED), and the port can be disabled using it. Counterintuitively, the port is disabled by writing a 1 to it. Even worse, this bit is in the same field as the change bits with RW1C semantics are – which means that the standard approach of reading and writing back the value read fails hard. This took us several hours to discover because the port reset was completed successfully, but right after that the device was inaccessible, even when we didn't do anything with it yet. To make matters more complicated, QEMU ignores this bit completely so the code worked fine in a virtual environment.

2.8 Bus Module

2.8.1 Device

Devices in the xHCI are represented by structures called device contexts, these are managed by the xHC and used to report device configuration and current state. A device context consists of a slot context, which represents the device as a whole (e.g. contains information about addressing or power management) and is implemented in the `xhci_slot_ctx_t` structure, and 31 endpoint contexts (which are described in the following section), one for each of the device's possible endpoints (though most of them may never be used).

Note that one will not find a specific structure used to represent a device context. We had one at first, but later discovered that according to a note in section 6.2.1 of the specification, the *Context Size* field in the *HCCPARAMS1* register determines whether a device context contains 32 or 64-byte members (the size of the individual members does not change, but the 64-byte version adds 32 bytes wide padding to each of them). Because of this, the device context structure cannot be represented statically and instead had to be implemented as a DMA buffer (see section 3.6) and its members offsets need to be computed at runtime with macros.

A device is initialized during a process called device enumeration, which begins when the system detects a new device and calls `xhci_device_enumerate` defined in `device.c`. This function then initializes the device by requesting its slot (index to the *Device Context Base Address Array* field in `xhci_hc_t`), enabling and configuring its control endpoint and requesting an address for it.

2.8.2 Endpoint

Every device can have up to 31 different endpoints. Every endpoint is represented by a structure called `xhci_endpoint_ctx_t` which is a part of `xhci_device_ctx_t`. Every endpoint has its number and direction and this information is used together as an index to the endpoint context array. The structure is initialized by the xHCI driver during endpoint initialization and most of its contents are set accordingly to the contents of its endpoint descriptor reported by the device. This structure is used to communicate the endpoint settings to the xHC. The xHC can also report the current endpoint state by setting the fields in endpoint context.

The xHCI driver uses `xhci_endpoint_t` for the actual representation of the endpoint. This structure contains all information acquired from the endpoint descriptors with some additional information required by the implementation such as TRB ring or mutex. The stored data is needed to fill the endpoint context because the xHCI driver does not have access to endpoint descriptors. The endpoint structure also contains a zero-sized array called `isoch` and the rationale behind it is described in section 2.9.

The endpoint initialization is initiated by the device driver. This sets up the endpoint structures and initializes its TRB ring. This is not true for default control endpoint which is initialized automatically during device initialization. After the initialization the endpoint can be immediately used by the driver by putting TRBs on the endpoint's TRB ring.

2.9 Transfers

The xHC uses transfers to abstract USB communication. Every call to `usbhc_transfer` is accomplished by creating a transfer, setting it up, handing it to the xHC and waiting for transfer completion. This section describes the code structure and implementation details of the xHCI transfer subsystem.

2.9.1 Executing Transfers

The xHC is responsible for communicating with USB devices using USB packets. These USB packets are created from the data given to the xHC through the transfer rings by the software. The software isn't required to split the transfer into USB packets, the xHC takes care of that.

Each transfer is formed by a *Transfer Descriptor*, which is a sequence of TRBs with a chain bit set. This allows the software to schedule a transfer containing larger data, or data not stored in contiguous memory. Each TRB points to a single contiguous buffer of data (in physical memory) and specifies the buffer's size. This buffer must be accessible to the hardware and its physical address needs to be set to the TRB.

Every endpoint uses a single transfer ring (with the exception of streams) initialized during the endpoint configuration. This transfer ring is passed to the xHC by the endpoint context. The software is not allowed to modify the transfer ring except when it wants to enqueue more transfer descriptors. If the software requires to modify the ring, it has to first stop the endpoint using the command interface, and after the modification is done it has to use the evaluate context command to communicate the changes to the xHC.

After the software puts a transfer descriptor on the transfer ring, it rings the doorbell associated with the device and the endpoint. This notifies the xHC and causes it to schedule the endpoint and process the transfer.

After the transfer descriptor finishes processing, the xHC can optionally notify the software of the completion by generating a *Transfer Event* on the event ring with the completion code set to reflect the result. By default, this only happens if the TRB processing ends with an error. The software can set the *IOC (Interrupt on Completion)* bit if it needs to be notified regardless of an error happening.

2.9.2 Lifecycle

Every transfer is represented by a single instance of the `xhci_transfer_t` structure. This structure encapsulates the `usb_transfer_batch_t` structure, which is used by `libusbhost` to schedule USB batches. Its lifetime is managed by the library calling the bus operations.

After the library allocates and fills up the `usb_transfer_batch_t`, the transfer is scheduled. At this point the xHCI takes control over the transfer. The transfer is processed in `xhci_transfer_schedule`, function where it is divided into individual TRBs, the correct bits are set depending on the transfer type, TRBs are enqueued on the transfer ring and the doorbell is rung.

The fibril responsible for the transfer is then suspended, waiting for a transfer event signalling its completion. For that reason, the IOC bit is set for every transfer descriptor, so that the software gets notified at all times.

After the transfer event is received, the event loop calls the `xhci_handle_transfer_event` function. There, the TRB which triggered the event is processed and the transfer, that scheduled it, is identified. For IN endpoints, the received data is then copied to previously prepared buffers. Lastly, an error code is set in the batch and the batch is finalized, waking up the suspended fibril.

2.9.3 Isochronous Transfers

Due to the nature of isochronous transfers, their implementation needs to avoid the common steps used by other transfers. This section describes the requirements and details of the isochronous transfer support.

The xHC has an internal timer which measures time in *microframes*, where 1 microframe is exactly 125 microseconds. The current time in microframes can be read at any time from the `MFINDEX` register. This register has only 14 bits and therefore it overflows every 2.048 seconds. The xHC can notify the software about the overflow by generating a *MFINDEX Wrap Event* on the event ring. This feature is disabled by default and activated at the xHCI driver startup.

Every isochronous transfer descriptor must have a set schedule time. This time is calculated by the xHCI driver and depends on the isochronous endpoint interval. Two following transfer descriptors must be always exactly the interval apart. Since isochronous endpoints can have up to 4-second interval, we need to track the *MFINDEX Wrap Events* to correctly determine the schedule.

The xHC gives its driver strict deadlines when an isochronous transfer descriptor may be present on the transfer ring for the transfer to execute successfully. This interval is partially HC-specific and it is based on *Isochronous scheduling interval* (IST). The transfer must be present on the ring at least *IST* microframes and at most 895 milliseconds before it is to be executed.

The isochronous endpoint also reserves some bandwidth during its setup and has only its reserved bandwidth available. For that reason the xHC never permits using more bandwidth. This sets a hard cap on the size of the data transferred in a single transfer descriptor. Scheduling a transfer descriptor with larger data causes the xHC to refuse to transfer the data, generating an error in the process.

If the xHC accesses an isochronous transfer ring to retrieve a transfer descriptor and the transfer ring is empty, the xHC generates a *Transfer Event* with the completion code set to *Ring Overrun* for IN endpoints, and *Ring Underrun* for OUT endpoints. This also removes the endpoint from the schedule and lets xHCI driver clean the structure and report the error to the device driver, if needed. To reschedule the endpoint, the xHCI driver needs to ring a doorbell.

Implementation

When we were implementing isochronous transfers, we had to consider all the requirements described above. We have tried not to introduce a new API specific to isochronous transfers to prevent cluttering the USB interface.

Following that intent, we have decided to slightly alter the semantics of calling `usb_transfer` on isochronous endpoints. The `xhci_endpoint_t` structure includes a substructure with fields specific to isochronous endpoints in `xhci_isoch_t`. There is a small trick present to avoid inflating the `xhci_endpoint_t` for every type of endpoint. The last field of the structure is a zero-length array of `xhci_isoch_t`. This doesn't take any space, but the pointer to this field is always valid and points behind the structure. We then allocate the structure with `calloc(1, sizeof(xhci_endpoint_t) + (type == USB_TRANSFER_ISOCHRONOUS) * sizeof(*ep->isoch))`. If the endpoint is isochronous, the memory allocated is larger to accommodate the additional space for `xhci_isoch_t` and we use the pointer mentioned previously to access it.

The `xhci_isoch_t` contains a dynamic array of the `xhci_isoch_transfer_t` structure. Instances of this structure represent isochronous transfers and maintain permanently allocated data buffers. Size of the dynamic array depends on the IST and the endpoint interval as we need to make sure we put the transfer descriptors on the transfer ring in advance but not schedule data, which is too old.

IN Endpoints

For IN endpoints, the semantics of `usbhc_transfer` is changed so that instead of triggering a read transfer, the call to `usb_pipe_read` retrieves previously read data and returns them to the caller instantly without blocking on the I/O if possible.. This means that we have to enqueue read transfers in advance and the caller is expected to withdraw them fast enough. Only if there is no prepared data, the call blocks, waiting for their delivery instead of returning an error.

To keep the transfer ring filled with transfer descriptors, we prepare a transfer descriptor for every transfer buffer when the first read is called, and enqueue as many of them as possible to the transfer ring before ringing the doorbell for the first time. This means that not all descriptors may be enqueued because of the strict deadlines imposed by the xHC. If this happens, we start a timer, which takes care of putting the transfer descriptors on the ring when the time is right. After every call to `usbhc_transfer` and withdrawing the received data, the processed buffer is recycled and scheduled back on the transfer ring.

OUT Endpoints

For OUT endpoints, the `usbhc_transfer` is modified, so that it doesn't ensure that the transfer finishes. Instead, it only copies the data to the internal buffer and attempts to put the buffer on the ring, if possible. If there is no free buffer yet (so all the buffers are on the ring), the call blocks until at least one transfer is finished and its buffer is freed.

Unlike with IN endpoints, we need to track which buffers are filled and which ones are empty, and schedule only the former. Other than that, the scheduling is the same. When scheduling the buffers at the start, we also need to make sure some of those are already present on the ring, so we postpone the execution of the first transfer descriptor by a short delay.

Error Handling

Transfer errors in isochronous transfers are very often considered not to be fatal and thus are skipped. Therefore, we have decided not to deliver them to device drivers immediately. Instead, errors are returned with the next subsequent `usbhc_transfer` call in the `error` field of `usb_transfer_batch_t`.

On the other hand, *Ring Overrun* and *Ring Underrun* errors are considered fatal as there is no simple way to recover from them. For IN endpoints and therefore for *Ring Overrun*, the error tells us that the driver either cannot keep up and read the data fast enough, or it has stopped reading completely. To not preserve old data, we reset the state of our internal buffers. If the driver attempts to read again, the transfers are started again as if the endpoint was accessed for the first time.

For OUT endpoints, *Ring Underrun* means that the device driver cannot supply data fast enough or has stopped sending them. We reset the endpoint again and restart if the driver writes to the endpoint again.

Unfortunately, some host controllers (or QEMU, at least) don't generate *Ring Overrun* and *Ring Underrun* events. We have therefore implemented a mechanism to detect the issues these events convey ourselves. There is a *reset timer* for each isochronous endpoint that is started every time a buffer is added to the ring, or a transfer event is received. The timer is set to expire after the endpoint interval passes plus a small constant. If this timer expires, we know that there was no event for at least the endpoint interval and therefore the endpoint isn't working properly, so we consider this as the error and reset the endpoint.

2.9.4 Streams

Streams are a new feature that was introduced with USB 3. They are used to create another abstraction layer over bulk endpoints in order to give driver an easy way to schedule multiple unrelated transfers in parallel using a single endpoint. Every bulk endpoint must either work in a mode without streams or work with stream support. It is not possible for any endpoint to work in both modes at the same time.

Every device with an endpoint that supports streams declares the support in the superspeed endpoint companion descriptor. The device driver then configures the endpoint with the stream support to enable streams. A single bulk endpoint may support up to 65533 streams.

At the host side, every stream has its own transfer ring. When the device driver communicates through the USB pipe, it can choose a different stream for every `usbhc_transfer`.

Streams are scheduled cooperatively over a single bulk pipe. The current stream selection can be initiated by both host controller and the device. The opposite side may refuse the selection if for example it doesn't have data buffers prepared. It is not possible for the system software to select the current stream manually.

According to the USB3 specification, one possible way to use streams may be supporting out-of-order data transfers required for mass storage device command queuing. During the development of xHCI support for HelenOS, we have not encountered any USB mass storage device declaring stream support. Therefore, the current xHCI stream support is completely untested and its API is not available to the device drivers. The current implementation is described in the next few paragraphs in case there is a need to write a driver using bulk streams and debug their support.

Stream Contexts

Every stream has its own transfer ring, which must be made accessible to the xHC. For regular endpoints, the dequeue pointer of the transfer ring is passed by the endpoint context. With streams, there are many transfer rings that need to be passed to the xHC. To accomplish that, the xHCI driver needs to use stream contexts.

A stream context is a simple structure used by the xHC to store transfer ring dequeue pointers. There are two possible approaches to setting up stream contexts. The first approach is called a linear stream array. In this case, every stream context directly contains a transfer ring dequeue pointer. The disadvantage of this approach is a need for a large continuous memory buffer as there may be up to 65536 streams. When selecting a stream, the stream ID set by system software is used directly as an index to the stream context array.

The second approach uses a hierarchical structure where each primary stream context may either contain a transfer ring dequeue pointer, or it may point to a secondary stream context array. In this case, the maximum size of primary stream context array is 256 stream contexts. The secondary stream context array can again have up to 256 stream contexts. This allows the software to use multiple smaller memory buffers instead of a single large continuous buffer. The secondary stream context support is not mandatory and not every host controller may provide such support. With this approach, the stream ID set by system software is split into two parts. The first few low order bits (the exact amount depends on the number of primary streams used) is used as an index to primary stream context array and the high order bits are used as an index into secondary stream context array.

In our code, every stream context has an associated instance of `xhci_stream_data_t`. This structure is used to bookkeep either the secondary stream context array buffer or the associated transfer ring.

To enable stream support, `libusbhost` should call either `xhci_endpoint_request_primary_streams` or `xhci_endpoint_request_secondary_streams`. This initializes the stream contexts and transitions the endpoint to the mode with stream support. The first function initializes a linear stream context array. The second function initializes a hierarchical array and its parameter `sizes` defines the sizes of individual secondary stream context arrays. These functions assume that there are no TRBs currently scheduled on the transfer ring of the affected endpoint. To stop using streams, `libusbhost` should call `xhci_endpoint_remove_streams`.

Handling Transfer Events

To finish a transfer and report the result to the device driver, we need to process transfer events which tell us when a transfer finishes. This works well for any endpoint which does not use streams. Nevertheless, if an endpoint uses streams, we encounter a problem. There is no way to determine the stream, which generated the transfer event, solely from the dequeued TRB.

Luckily, the xHC gives us a way to pass custom data in the transfer event. A xHCI driver may create a TRB with the TRB Type set to *Event Data TRB*, which is chained to the previous TRB it is associated with. This TRB has the IOC bit set instead of the previous TRB and it can bear any custom 64-bit value. The previous TRB also needs to have *Evaluate Next TRB* bit set which forces the xHC to always process the *Event Data TRB* immediately before another stream is selected.

Our xHCI driver uses *Event Data TRBs* to store the pointer of the transfer structure creating this transfer. This allows us to access the transfer structure in the transfer event handler and determine the selected stream.

Chapter 3

USB Subsystem Modifications

Before we start explaining individual modifications to the USB stack, we think it is good to explain the overall architecture of it. Please see the figure 3.1.

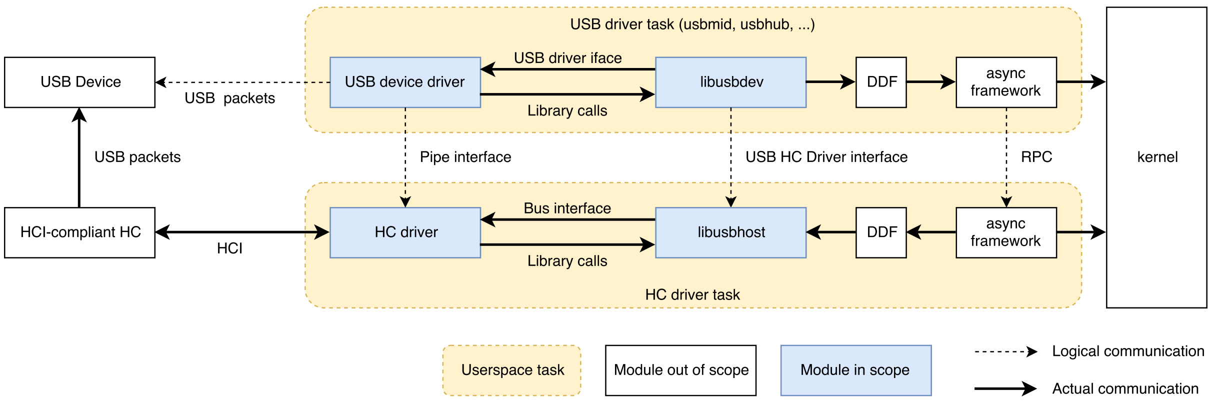

Figure 3.1: The runtime architecture of the USB stack, showing the communication paths

At the highest logical level, the USB device driver communicates with the device using USB packets. The packets are sent through pipes, illusion of which is created by the `libusbdev` library. The library communicates with the HC driver using the `usbhc` driver interface. The drivers actually communicate using the Device Driver Framework, which in the end uses the synchronous RPC built upon asynchronous IPC mechanisms in kernel. The scope of USB stack is limited to the USB device driver tasks, and to the USB Host Controller driver tasks. Because the USB Device Framework hasn't changed much in USB 3, our primary focus related to implementing USB 3 support was the HC driver. The xHCI driver itself was exhaustively explained in the previous chapter, this chapter explains the modifications needed to implement it. Furthermore, it contains the modifications needed or supporting the implementation of other key requirements of the project.

3.1 Bus Interface

Let us start with the most notable change we introduced to the `libusbhost` library. Yet before we do, we would like to describe the previous way how `libusbhost` supported USB HC drivers.

3.1.1 Former State

The library support was mediated by a single structure, `ddf_hc_driver_t`. This structure contained callbacks implementing routines to initialize internal structures, start the HC, handle interrupt and so on. Once the driver's DDF callback `dev_add` was called, the driver passed the call to the library along with the `ddf_hc_driver_t` instance, which described how the HC is to be initialized. In one of the callbacks, the driver called a function `hcd_set_implementation`, by which it configured runtime operations.

From the perspective of data structures, the `libubshost` defined a structure for an instance of host controller, `hcd_t`, and for endpoint, `endpoint_t`. Both structures had a field for storing driver private data. The library also managed a device tree, with the device structures stored inside DDF function nodes. The device structures were however private for the library, and apart from two callbacks ep_add_hook and ep_remove_hook the driver had no information about the devices. Also, every endpoint could have two callbacks assigned by the driver, which were called to alter maintenance of the Toggle bit (toggle_get/set).

The main work to be done by the driver is transfer management. For this purpose, the library defined a structure usb_transfer_batch_t, which was created for every transfer and then passed to the driver's schedule operation. The batch carried all the information about a USB transfer: target address, pointer to the data buffer and its size, direction of the transfer and so on. Along with this data, it carried one of two callbacks (in/out) to be called after the transfer is finished. This particular callback was set by the library, depending on the context from which a transfer was scheduled. Transfer could be initiated either by the HC itself (e.g. reading the device descriptors to determine match IDs for the DDF), or by the driver of the device via the IPC interface methods (usb_read or usb_write).

In the first case, the driver called hcd_send_batch_sync, which created a simple structure with completion flag, and the callback was set to a function which filled the structure and toggled the completion flag. Then the issuing fibril polled the completion flag until the transfer was completed. In a case of a bus transaction issued by the device driver, the callback answered the IPC call. The asynchronicity of HelenOS IPC fits this scheme perfectly.

Sometimes, if the request modified the internal state of the device (e.g. USB ResetEndpoint request), the callback was wrapped in another callback, which reset the endpoint's toggle after completion, and called the original callback.

The overall architecture of using callbacks passed as function arguments and stored all around was really flexible, but very hard to read and understand. Instead of simply tracing function calls, we needed to carefully study the lifetime of the structure to track origins and modifications of these callbacks.

Also, we felt that keeping driver-private data in a void pointer (with the burden of maintaining the allocated memory) doesn't fit well for a library of such a specific purpose. Instead, we decided to start a little revolution here.

3.1.2 Current State

The most notable change we introduced is a clear separation of the callbacks, forming the interface of the HC driver against the libusbhost library, named the "Bus interface" in the figure 3.1. All the runtime callbacks the library calls are contained in a single structure, bus_ops_t, which is defined by the driver. It is possible (and recommended) that the instance of this structure is static const, and a pointer to it is shared between all HC's the driver controls. The names of the callbacks are "namespaced" by their first word, to clearly distinct which entity the operation works with (device_enumerate, endpoint_register, batch_schedule, . . .).

Moreover, we defined a bunch of structures the library shares with the driver. All of them use pointers and arrays to build a tree of entities the driver works with. The lifetime of all entities is managed by the library, but the driver is (with some exceptions) responsible for their allocation and destruction. That gives the driver great flexibility in where the memory for the structures will be allocated, and also how large the memory is. None of them contains a void pointer to store private data, as its not necessary – the driver is supposed to allocate memory for a bigger structure if it needs it.

The C standard defines that pointer to a structure is equivalent to a pointer to its first field. This promise is heavily relied on to implement "inheritance" of a C structures. To give an example, suppose that the driver needs to keep more data about an endpoint. It then creates a structure for it:

```c
typedef struct hc_endpoint {
        endpoint_t base;

        bool some_flag;
        unsigned more_data;
} hc_endpoint_t;
```

Then, in the endpoint_create bus operation, it is supposed to create the endpoint_t structure. It allocates a space for hc_endpoint_t instead, and returns the pointer to its base field (which we know is the same, but of different type). Whenever it then receives an endpoint_t * argument to a bus

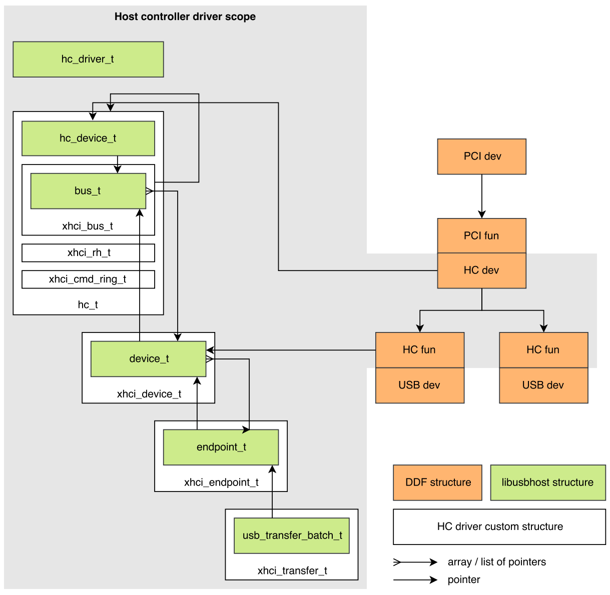

Figure 3.2: Bus structures demonstrated on the example of how xHCI driver uses them.

operation, it knows it is safe to typecast the pointer to `hc_endpoint_t *`, and work with the extended structure containing more data.

All structures and their relations can be seen on figure 3.1.2. You can see that all of them have pointers to their parent entities. That allows us to reduce number of arguments passed to the bus operations, while retaining the flexibility needed to implement the operations in different drivers. Lets walk through the entities one by one.

Structures Shared With the Driver

`hc_driver_t` Represents the driver instance. Created at the task startup (usually as a static instance), passed to the `hc_driver_main`. Contains operations to initialize a HC device.

`hc_device_t` Created by the library for every HC controlled. Contains pointers to the DDF device, control function, interrupt-replacement fibril. Shall be treated as opaque by the driver.

`bus_t` Created by the driver for every HC it controls. Manages the reservation of default address, keeps the pointer to bus operations. Strictly speaking, `hc_device_t` and `bus_t` represent the same entity – the HC instance. They are separated to isolate responsibilities.

`device_t` Allocated inside the DDF function node for every USB device managed by the bus. Contains an array of endpoints registered for that device. Also, it contains a list of children `device_t` structures, in case this device represents a hub. Because the topology presented to the DDF is flattened, this is the only place where the USB tree topology is stored.

`endpoint_t` Created dynamically at the time of registering an endpoint by the device driver. Contains information needed for scheduling and synchronization of scheduled transfers. Reference counted, as its lifetime can extend the lifetime of a device.

`usb_transfer_batch_t` Created dynamically at the time of initiating a transfer. Ownership travels across more entities, more information to be found in section 3.2.3.

Now that we know the entities, we can introduce the operations expected to be implemented by the HC driver.

Bus Operations

`interrupt` Process a hardware IRQ. Is passed a bus instance and a 32-bit status retrieved from the top half.

`status` When the library fails to enable the IRQ, it starts a polling fibril instead. The polling fibril calls the status method to retrieve the status to be fed to the IRQ handler.

`device_enumerate` When a (root)hub finds a new device, the library creates the DDF function node, and allocates a `device_t` structure. This operation is then called to address the device and create the DDF match ids for it.

`device_gone` Called when the hub signalizes a device is gone. Not required.

`device_offline` **and** `device_online` The library calls these when the user requested the device to be brought offline/online. More information about this mechanism is in section 3.2.2.

`endpoint_create` Called when the library needs to materialize an endpoint from the endpoint descriptor fetched from the device.

`endpoint_register` When the device driver maps an endpoint descriptor, a pipe is created. On the HC side, respective endpoint is registered. The registration is performed by the library itself, this is just to let the driver do the HC-specific work.

`endpoint_unregister` When the pipe is closed, the endpoint is unregistered. The HC is supposed to terminate all transfers on that endpoint before it returns from this operation.

`endpoint_destroy` A function to free an endpoint and all allocated information. Called once the reference count drops to zero. If not specified, the structure is freed by `free`.

`batch_create` Called by the library when a transfer needs to be initiated. If not specified, the batch is allocated by `calloc`.

`batch_schedule` Called once the batch is ready to be scheduled.

`batch_destroy` Once the batch is finalized and its completion callback is called, this function is supposed to free the memory. If not specified, the batch is freed by `free`.

Some operations do have generic implementation, so that the driver is not required to implement them, a common behavior is implied. Others are required only to support a specific functionality – if not implemented, the functionality is unavailable. Of course, functionality like the batch scheduling is optional, but without it the driver is kind of useless.

The only callback that remains separated from the bus is the batch completion callback. In order to support both synchronous batches from HC itself and asynchronous batches from IPC, there needs to be a distinction. However, the callback is no longer abused to implement additional functionality when the transfer is finished.

To avoid duplicating functionality that was previously in the library and now is supposed to be implemented by the driver, we created a module `<usb/host/usb2_bus.h>`, which contains an implementation of some operations relevant for USB2 and below. It implements address allocation, as only xHC assigns addresses in hardware, older HCs leaves this responsibility on software. The same holds for bandwidth management, which is further moved to a separate module `<usb/host/bandwidth.h>`.

Development of the Bus Interface Throughout the Project

At first revisions, the bus interface was a bit richer. Also, it contained more of the USB2 specifics hardwired – like the bandwidth management. The impulse to create the bus interface was the lack of systematic approach in the previous solution, so it was a bit hard to understand. So we first did a refactoring process, which moved all callbacks to the bus. For a long time, there was a kind of runtime virtual binding of functions – the older HCs used `usb2_bus_t` as their bus implementation, and overrode some of its functions. As time passed by, we were able to identify and isolate parts that could be separated cleanly and reduce the bus interface to its current state.

The result is not that much different from the former state – only extended by a few device-related callbacks, as xHC needs to know about devices, former HCs need not. But the fact we needed to do the refactoring just to understand what is going on proves the fact this interface must have been cleaned up and defined more clearly. Also, the sharing of structures reduced the amount of bookkeeping needed to be done by the drivers. This claim is supported by the fact that even though we implemented new features and made performance optimizations, the codebase of former HC drivers shrunk a few kilobytes.

3.2 Explicit Device Removal

One of the project goals is to alter the USB subsystem to allow support for explicit device removal. Such feature can be found in most modern operating systems and is often used to ensure that devices are left in a consistent state after a physical port detachment occurs.

The explicit device removal feature usually provides a frontend interface in the operating system, through which users can observe currently connected devices and, if needed, issue a signal to the operating system that their physical detachment is imminent. Following that, the system is expected to promptly terminate all ongoing communications with the device and signal the user back. After receiving the confirmation, user can then safely unplug the device from the system bus without any risk of interrupting communications, which could otherwise result in undefined state of the device.

3.2.1 Considerations

In the USB protocol, communications between the host and the device take place in the form of *transfers*. Depending on its version, the host controller may have various roles in the realization of these transfers. For that reason, version-specific modifications are carried out separately in host controller drivers, whereas common functionality is implemented in the bus module, which is a part of `libusbhost`.

The disconnection routine for explicit device removal is implemented as follows:

1. The user signals the intention to disconnect a USB device.
2. The respective device drivers are notified to end their business (e.g. flush buffers or close files), possibly scheduling a multitude of transfers to the device.
3. The HC driver disables the capability to schedule new transfers to the device.
4. The HC driver aborts all leftover active transfers to the device.
5. The device configuration is dropped, leaving it in the *Addressed* state, in which it is considered safe to be physically removed from the bus.

If the user requests that this routine is rolled back, the steps of the disconnection routine are just executed in reverse order. The following can therefore be labeled as a reconnection routine:

1. The user signals the intention to resume communications with a USB device, on which the disconnection routine has been previously performed.
2. The HC driver configures the device.
3. The HC driver enables the capability to schedule new transfers to the device.
4. The operating system is notified that the device is reachable and matches it to appropriate drivers, which initiate communications with it.

The specialization of both routines is performed in the same way as other bus module interactions. All HC drivers hand off their DDF and device callbacks to the bus module, which then calls them back to perform low-level commands related to specific devices, endpoints and transfers. This way, the high-level logic contained by the bus module essentially follows the listed descriptions above and the version-specific extensions are resolved in the respective HC driver implementations.

3.2.2 Offline and Online DDF Signal

The HelenOS Device Driver Framework includes two user-initiated signals relevant to the implementation of this feature.

Offline Signal This signal informs a driver attached to a DDF node that its managed device may be removed in the near future. The driver is expected to immediately cease all user operations on the device and unbind its child DDF functions, possibly sending a *Device Remove* signal to all their attached drivers in the process.

Online Signal This signal is a logical counterpart to the previous signal. It informs a driver attached to a DDF node that its managed device will not be removed in the near future. The driver is expected to expose all child DDF functions related to the device, possibly sending a *Device Add* signal to all their matched drivers in the process.

These signals can be easily issued by the user from the system shell by means of the `devctl` application. See Listing 3.1 for invocation example.

```
# Prepare the unplug high speed device at address 2.
devctl offline /hw/pci0/00:04.0/usb2-hs

# We changed our mind. Bring the device back online.
devctl online /hw/pci0/00:04.0/usb2-hs
```

Listing 3.1: Example usage of the `devctl` application to issue offline and online signal to a USB high speed device at address 2. The host controller PCI address is `00:04.0`.

It follows that these signals can be used for the implementation of the explicit device removal at the level of USB host controller drivers. For that reason, `libusbhost` has been extended to handle appropriate DDF callbacks for functions corresponding to HC's child devices. Their handling is forwarded to the bus module, which executes the disconnection or the reconnection routine for the *offline* and *online* signal respectively. In addition, the transfer scheduling mechanism of the bus module has been extended to permit scheduling new transfers only to devices which are currently online.

The general scheme of stopping communication with a device breaks down to unregistering all its registered endpoints. The biggest challenge the driver faces is to abort all currently running transfers on an endpoint that is being unregistered. The majority of transfers (Bulk, Control, Isochronous, Interrupt-out) wouldn't pose a problem – we could just wait the short while until they are completed, either successfully or not. But then there are Interrupt-in transfers, which, especially in case of gone device, may not complete in a timely manner.

3.2.3 Aborting Active Transfers

It is not possible to "abort a transfer" in a generic way, mainly because of synchronization issues. Before we explain how can a transfer be properly aborted in various Host Controllers, let us describe the lifecycle of a transfer batch, a structure representing a transfer in HC drivers.

Currently, USB stack in HelenOS only supports synchronous interface to interact with pipes. The two functions are called `usb_pipe_read` and `usb_pipe_write`. Driver calls these functions on pipes, and provides a buffer – either filled with data, or to be filled. Once the call crosses the IPC barrier, it is joined to a call to `bus_device_send_batch`. This function finds the target endpoint structure, and passes control to `endpoint_send_batch`.

There, an instance of `usb_transfer_batch_t` structure is created and filled with parameters of the transfer. It is then passed to the driver implementation to be scheduled. The driver typically copies the data to a buffer suitable for the device, prepares some supporting structures, and finally, schedules the transfer to the hardware.

Since then, an interrupt may come and finish the transfer in a different fibril. A transfer is finished by copying the data out from the hardware buffer to the batch buffer, setting the error code and calling a completion callback. This callbacks answers the original incoming IPC call, causing the `usb_pipe_read/write` function to return. After that, the transfer batch is destroyed.

But that's not the only scenario that may happen. From the moment a transfer is created, a pointer to it must not be forgotten, otherwise the caller would never return. But on the other side, once the pointer to batch is stored somewhere, the transfer might be aborted at any time. Furthermore, once the transfer is scheduled to the hardware, the buffers must not be deallocated until the driver is sure that the hardware won't use them anymore.

This synchronization problem might be resolved by locking the batch and reference counting, but then different problems would arise (e.g. a transfer could be finished after the endpoint was successfully unregistered, just because we cannot know if there's any). So, we decided to take a different approach.

As the interface is synchronous (and it doesn't make much sense to make it asynchronous, unless under special conditions), and the endpoint is assumed to be available for one driver only, there's no point in having more than one transfer active at a time. So, we store the pointer to a batch inside the endpoint structure, in the field `active_batch`. This field shall not be accessed, unless the endpoint guard is locked.

Speaking about the endpoint guard leads us to one of the strange design decisions we made. Endpoints do not have their own guard, they inherit one while being put into the online state. That is, when the endpoint is being registered, the HC driver calls `endpoint_set_online` and passes a pointer to a mutex which will be used to synchronize transfers on that endpoint. The endpoint itself never locks this mutex, it only uses it to ensure correctness (whether the mutex is locked in named functions) and to wait on a condition variable.

Sooner or later in the scheduling of a transfer batch, there is a need to access shared structures. Once there is, the HC driver locks the mutex, and in a single critical section it calls the function `endpoint_activate_locked`. If this function successfully returns, the endpoint is reserved for a given batch. By calling the function, the batch ownership is given to the endpoint – once the critical section ends, the batch must be considered already finished.

In the meantime, there are two possible execution flows that are related. First of all, there is the interrupt handler finishing transfers. Once it decides a transfer is to be finished, it is supposed to have the mutex already locked (to avoid racing with the scheduling fibril), and calls function named `endpoint_deactivate_locked`. This function allows another batch to use the endpoint, and transfers the ownership of the batch to the caller.

Second, there can be a fibril trying to unregister the endpoint. To do so, it locks the mutex, and calls `endpoint_set_offline`. This blocks access for further transfers, and also wakes fibrils waiting inside `endpoint_activate_locked` from the sleep, returning an error value. The unregistering fibril then may decide to wait a while to finish already running transfers, and then do HC-specific steps to remove the running transfer from the hardware. Eventually, it shall call `endpoint_deactivate_locked`, which takes the ownership of the batch, and gives the unregistering fibril a sole right to finish the transfer with an error code.

This whole mechanism is completely opt-in, and the driver can avoid using it (like xHCI do for stream-enabled endpoints). The HC-specific part of the implementation is discussed in the next sections.

3.2.4 UHCI, OHCI and EHCI Specifics

All three HCI's that were supported prior to our project have similar structure with regard to what is required to implement transfer aborting. Let us first describe very briefly how these controllers handle transfers.

Generally, all three host controllers require the driver to create a system of linked structures in memory (for UHCI and OHCI, restricted to the lower 4 GBs of addressable space). The names and guts differ, the structures however describe a linked chain of queues. Queues are then filled by transfer descriptors, which describe USB transfers to be done. Once the transfer is done, its descriptor is flagged, removed from the queue and if the descriptor is marked, the host is interrupted. More specific information can be found either in respective specifications, or in the documentation of the HelUSB project.

At the time of receiving an interrupt, the host does not know which transfer was finished, so it has to check all pending transfer descriptors for the completion flag. In case all the transfer descriptors of a transfer are completed, the driver may finish the transfer.

When aborting a transfer, the driver must make sure the controller is not using any of the allocated buffers. It is allowed to modify the queues while the controller is scanning them, the modifications must however follow an order in which the consistency of the structure is guaranteed at any time. After it does, it must notify the driver that the structure has changed, in order to force the controller to clear its caches (EHCI only).

Because the interrupt handler is polling the transfers for completion, it is enough for the driver to remove the transfer from the list of pending transfers to be sure no other fibril will ever complete it. After that, we can finish it ourselves with an error. Note that in this case it is unknown to driver whether the transfer was completed or not – but since the driver is unregistering an endpoint, the device must already be in a state in which it expects removal.

The nature of handling finished transfers defines the weird semantics of the inherited mutex. Let's consider a situation, in which two locks were involved: a lock protecting the HC structures (lists, interrupt handling, . . .) called H and a lock protecting the endpoint E. When a transfer is scheduled, it must first wait until the endpoint is available. To avoid a deadlock between finishing the current transfer and waiting for it to finish, the mutex E must be taken first. Also, E cannot be released before taking H, because after E is released the transfer can be aborted immediately. On the other hand, when an interrupt comes, there is no way how to get a pointer to the endpoint without taking H first. Neither H can be released before taking E, because we cannot access the batch unless holding E, and even transferring a pointer to an endpoint does not help – in between the critical sections the current transfer transfer can be aborted and a new one scheduled, resulting in completing a wrong transfer. To avoid ABBA deadlock eventually, we just have to avoid using two locks for transfer synchronization.

From the further perspective, these controllers do not have an internal state for individual devices and endpoints, so the deconfiguration and its rollback is an operation on software-state only. As such it is already done completely by the `libusbhost` library.

3.2.5 xHCI Specifics

Since xHCI is the latest HC interface implementation, a lot more is done by the hardware for the HC driver in comparison with previous versions. The concept of xHCI command ring leads to very elegant implementation of the required functionality on the HC driver part.

For the purpose of aborting active transfers, the xHCI features an explicit *Stop Endpoint* command, which instructs the HC to abort all transfers to a specific device endpoint. This command is issued by the HC driver for all removed device endpoints, which are active at the moment of the request. Furthermore, device configuration is dropped along with all remaining endpoints by issuing a *Configure Endpoint* command with the DC (deconfigure) flag.

Reconnection is quite straightforward and requires only that the HC driver issues a regular *Configure Endpoint* command in order to transition the device from the *Addressed* state back to the *Configured* state.

3.2.6 Driver Support

Since the existing USB drivers were quite incomplete, their implementation has been extended to add support for explicit device removal. This mostly lead to novel approaches to deallocation of device-related memory structures, which is often performed in the `device_remove` and `device_gone` driver callbacks.

For instance, a number of drivers required that an implementation of `device_remove` is created in the first place. Quite so often, the existing implementation of `device_gone` has provided a good starting point, given that both functions have to deal with USB device's demise. The fundamental difference was that while `device_gone` merely dealt with the fallout of unexpectedly unplugged device, `device_remove` had the opportunity to tie up all lose ends prior to the physical detachment of the device.

This posed a problem, especially in HID and hub drivers, which heavily relied on polling of interrupt endpoints. Since polling was a synchronous operation from the device driver's point of view, a polling fibril has been created when the driver started managing the device. When the device was unplugged, the polling operation failed with an error, waking up and effectively terminating the fibril in the process. Since no explicit joining mechanism was available, the implementation of `device_gone` merely spinned a limited number of times, waiting for the polling fibril to die on repeated error on pipe read. This mechanism was however unsuitable for `device_remove`, since the fibril would not be awaken due to an error caused by the physical disconnect, which has not happened yet.

This lead to complete refactoring and extension of the USB device polling mechanism, which is described in detail in Section 3.3. In summation, the new version of the mechanism allows device drivers to join their polling fibrils and consistently wake them up in order to avoid deadlocks.

To actually stop the polling in the `device_remove` callback, the device driver has to trigger a transfer abort inside the HC. There's no sensible way of how to allow a driver abort its own transfer, which wouldn't introduce potential memory leaks inside HC or synchronization problems. So we decided that the driver will have to unregister the endpoint as the only way to abort a currently running transfer (and also disable scheduling of new ones). The device's endpoints (or, at this layer already called pipes) are completely managed by the `libusbdev` library. Previously, their lifetime was strictly tied to the lifetime of the handled device. There are only two possible ordering of these two operations:

1. Close the pipes after the `device_remove` callback returns. After that, the polling fibrils will end and destroy their structures. The library USB device structure will have to be reference-counted and the counter will be managed by the driver to account for polling fibrils.

2. Close the pipes prior to calling `device_remove`, effectively destroying the only reason why this callback exists, and make the expected removal equivalent to the unexpected one.

It is pretty obvious that we had to choose a completely different approach. So we came up with three more options:

3. Introduce a new callback, e.g. `device_removed`, to be called after the pipes are closed. The removal would be split into two phases: first, in `device_remove` the loose ends are closed and the device is brought to a state expecting removal, then in `device_removed`, the polling fibrils are joined and structures are destroyed.

4. After calling `device_remove` and closing the pipes, call `device_gone`.

5. Allow the driver to close individual pipes imperatively.

At first, we decided to go with option 3). But it was yet another callback, we didn't come up with a name that wasn't so similar and yet was clear, and finally, the `device_removed` callback implementations were very similar to `device_gone`.

The option 4) seems reasonable at the first look, but it would create inconsistency between DDF drivers and USB drivers, which would be very confusing for developers. [1]

In the end, the only option left was letting the driver close its own pipes. So we introduced a new library call `usb_device_unmap_ep`, which does exactly that. The new library polling mechanism closes its pipe automatically, but this functionality is usable in any driver for any endpoint.

The implementation of explicit device removal support for the rest of the drivers was a trivial extension of their previous functionality.

3.3 Polling Mechanism Improvements

As hinted by the previous section, the USB device endpoint polling mechanism has been completely refactored.

The polling mechanism was originally introduced to simplify repeated scenario in USB device drivers: handling interrupt IN endpoints. Instead of them interrupting the flow of the device driver, the driver has to poll them – which includes sleeping the vast majority of time waiting for the transfer to finish (signalling an interrupt from the device). In order not to block the fibril handling connection from `devman`, the driver must start a new fibril just for the polling. To avoid repetition, the polling subsystem offers the driver a more interrupt-like interface.

3.3.1 Before State

In the original state, USB device drivers had to configure polling using a configuration structure named `usb_device_auto_polling_t`, then pass this structure to one of four functions:

- `usb_device_auto_poll`,
- `usb_device_auto_poll_desc`,
- `usb_device_auto_polling`,
- `usb_device_auto_polling_desc`.

These functions copied the configuration and started an automated polling fibril, which interacted with the device driver using callbacks specified in the configuration structure. At the end of the polling, the fibril deallocated all of the resources and terminated.

[1]Actually, some team members think this behavior shall be present in DDF too, but we didn't open a discussion with HelenOS developers.

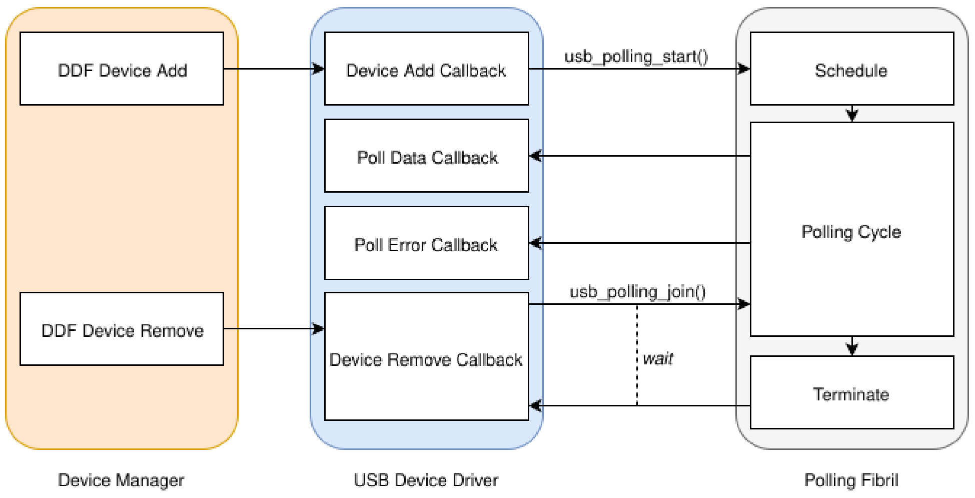

Figure 3.3: A diagram of interactions between the Device Manager, USB device driver and one of its polling fibrils during its lifecycle.

3.3.2 Motivation

There was a number of problems with the status quo:

- The semantics of functions used to initiate polling was not clearly distinguished by their name (and neither their documentation).
- In addition, the polling functions had a high number of arguments, which opened possible room to device driver errors due to their misinterpretation (or further API changes in the future).
- The polling fibril was detached at the moment of polling start and could outlive the device in the driver's memory, then fail later when accessing memory, which was already freed in `device_remove` or `device_gone`. Some drivers bypassed this by spinning in these callbacks, waiting for the fibril to terminate. However, if the fibril was still polling even after a number of attempts, a non-zero error code was returned, rendering the entire DDF function (along with its subtree) in a defunct state.
- The driver was unable to inspect the state of the polling fibril directly, so often a flag had to be created and maintained by polling callbacks.
- A distinct subset of polling parameters were not configurable and were hard-assigned their default values inside function implementation. If the driver wanted to change any (and not necessarily all) of such parameters, it had to specify all the values by itself. Because the constants were hard-coded in the implementation, the driver then usually had to copy the values.

3.3.3 Modifications

The configuration structure `usb_device_auto_polling_t` has been renamed for simplicity sake to `usb_polling_t`. Instead of serving as a one-time configuration structure during polling initiation, its role changed to represent the entire instance of the polling process throughout its lifetime.

Introducing standard functions such as `usb_polling_init` and `usb_polling_fini`, the device driver is now fully responsible for the ownership of the structure. This is convenient, since drivers often have their own structures for device data, where `usb_polling_t` can be placed as a field, dropping the need for additional calls to `malloc` and `free`. In addition, this resolves the problem with default values of various configuration parameters, since in `usb_polling_init` all parameters are assigned their default values and device driver can override only those desired.

All four of the original polling initialization functions were unified into a single function called `usb_polling_start`. Since there is now a clear structure, which represents the polling instance, the arguments of the original four functions were moved to `usb_polling_t`, where they are clearly named and documented, preventing any possible errors from their misinterpretation. Suffice it to say, that the original four functions mostly fulfilled the role of syntax sugar, which is now rendered unnecessary, given the fact that default values of configuration parameters are pre-filled in the polling structure.

Lastly, the API was extended with the `usb_polling_join` function, which closes the polling pipe and consistently waits until the polling fibril terminates. This function addresses the problem of spinning in driver's `device_remove` or `device_gone` callbacks, or possible negligence, which may result in the polling fibril outliving the device and then accessing invalid memory. Calling this function in this context will result in the immediate and synchronous termination of the polling mechanism prior to deallocation (as depicted in Figure 3.3).

Furthermore, the exposure of internal polling parameters now gives device drivers more creativity in their approach to polling. For instance, drivers can now inquire about the state of the polling fibril without the need to have a private flag maintained by their polling callbacks. The drivers can also change polling parameters such as request size or polling delay mid-flight, which is a more flexible approach than to stop polling, change parameters and then start polling again (note that stopping polling at will was not supported by the previous implementation without generating actual errors from the hardware device).

A nice minimalist example of the new polling mechanism usage can be found in Listing 3.2

```c
static usb_polling_t polling;
static uint8_t buffer[13];

static bool callback(usb_device_t *dev, uint8_t *buffer, size_t size, void *arg)
{
	printf("Have data!/n");

	// Return true if we wish to continue polling.
	return true;
}

static void demo()
{
	// Initialize.
	usb_polling_init(&polling);

	// Configure.
	polling.device = /* some usb_device_t here */;
	polling.ep_mapping = /* some interrupt(in) endpoint of the device */;
	polling.buffer = buffer;
	polling.request_size = sizeof(buffer);
	polling.on_data = callback;

	// Start polling.
	usb_polling_start(&polling);

	// Sleep synchronously for a while.
	async_usleep(10000);

	// End polling and clean up.
	usb_polling_join(&polling);
	usb_polling_fini(&polling);
}
```

Listing 3.2: Minimal usage example of the new USB device polling mechanism.

3.4 A Library Module for USB Hubs

We introduced a new module to support writing hub drivers: `usb/port.h`. It solves the problem of hub drivers that events are announced through a single channel, even though they need to wait for each other. The implementation of this module was motivated not only by the need of refactoring, nor because we wanted to share the functionality with the xHCI driver, but because the previous implementation of `usbhub` driver synchronized the fibrils wrong. There might have been situations in which two fibrils were spawned and enumerated the same device.

To solve the issue a state information is managed for every port, represented by the structure `usb_port_t`. It contains the current port state, one of the following:

Disabled There has been no activity on the port yet, or it is already over. Initial state.

Connecting A connected event came, the enumerating fibril was started. It hasn't finished yet, so the device is not enumerated yet. Also, the device is still connected, no error event came while the connecting is in progress.

Enumerated The device was successfully enumerated, so it has to be removed after it will be disconnected.

Disconnecting A disconnected event came after the device was enumerated, so a removing fibril was started.

Error An event about the device disconnection state came while the enumeration was in progress, so the enumeration fibril has to be stopped. This is achieved by returning `EINTR` from the blocking functions.

The basic synchronization invariant is that the state never changes unless the guard is locked. It is expected that the guard is not held for a long time, so the fibril generating events shall not be blocked indefinitely. The only exception being the final phase of enumeration (announcing the device to the bus), which once started, cannot be easily interrupted – but this operation on the bus is also expected to run for a limited time only.

The scheme of states and allowed transitions can be seen on figure 3.4.

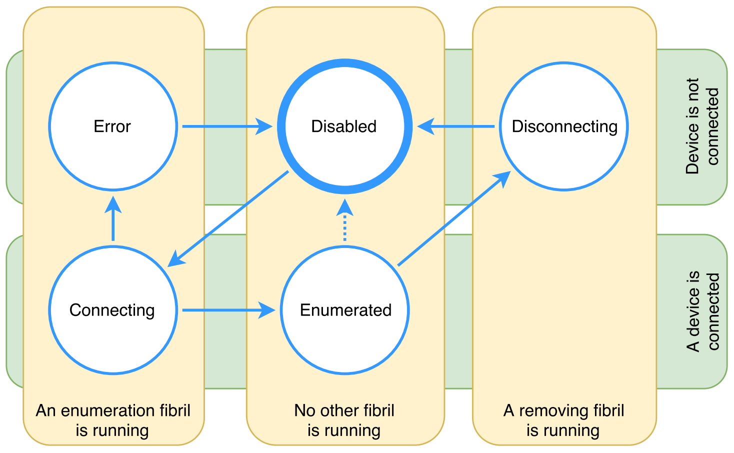

Figure 3.4: A graph of port states and their transitions.

The transitions are triggered by delivering events to the module. This is done by calling following functions:

- `usb_port_connected`
- `usb_port_disabled`
- `usb_port_enabled`

The `connected` and `disabled` functions take a callback as an argument. If the respective state transition is triggered, this callback is run in a separate fibril. These functions can block to obtain the port guard, and the `disabled` event can furthermore block while waiting for the enumeration fibril to terminate. But before it does, it makes sure the worker will be notified as soon as possible.

To make it work, the callbacks must not block while holding the guard. They can, however, wait for the enabled event, signalling the completion of port reset, using a function `usb_port_wait_for_enabled`. The caller is required to check the return value of it – it can either finish with `EOK`, timeout with `ETIMEOUT` or be interrupted with `EINTR`. The enabled event is sent by calling the function `usb_port_enabled`.

When this fibril management is separated, an implementation of USB hub driver or root hub driver is very simple. The hub driver just initializes the `usb_port_t` structure for every port it manages, then waits for events from the hardware (either by poling the *Status Change Endpoint*, or by waiting for an interrupt), and forwards the events to this module, providing an implementation of the enumeration and removal process.

One last thing to note is what to do when the structure is finalized, but a device is still connected. It might happen that there is a subtree of devices and hubs being removed because the subtree root R has been unexpectedly removed. In that case, a port disconnect is delivered to the parenting hub, which triggers the device removal process. The hub R then receives the DDF signal `device_offline`, and starts removal of its own children. But there's a catch: USB hierarchy is presented as a flat one to the DDF framework. That means that the devman has no clue about hubs (it sees them as ordinary USB devices), and considers that all of them are connected to the HC directly. Device removal is prepared for the situation that device will remove also its *children* devices, but serializes removal of *sibling* devices. Therefore, recursive removal of hubs creates a deadlock in devman.

Fortunately, the HC driver must be prepared for badly written drivers, so it cleans up after the device function is unbound. This cleanup includes also removal of its former children devices – so the workaround for this problem is simply leaving those devices connected, as the HC will remove them itself. This state transition is denoted by dotted line in the graph, and is triggered by finalizing the port structure while a device is enumerated.

3.5 USB Tablet Driver

This modification is very standalone and seemingly simple, yet very useful and appreciated. We extended the HID driver to support absolutely positioned devices. That means, one can now connect a USB tablet and it will work in HelenOS. If you are still wondering what this could be useful for (people using USB tablets are usually graphic designers or photographers, not microkernel developers), try running QEMU with an emulated one:

```
$ qemu-system-x86_64 -enable-kvm -usb -device usb-tablet -boot d -cdrom image.iso
```

When using mouse with relative positioning (PS/2, USB mouse), one has to first click inside the window of QEMU to let it grab input. To release it again, a special key combination (for current QEMU Ctrl+Alt+G) must be pressed. When using an emulated USB tablet instead, the mouse is not "locked" inside the window, but it can freely move in and out and still be registered by the guest OS.

3.6 DMA buffers

A simple but repeated scenario gave rise to another new submodule of `libusbhost`. It started as a thin abstraction, and its purpose grew to a major performance optimization targeting the whole system. Therefore, we plan to discuss with HelenOS core developers to incorporate it in other drivers as well.

3.6.1 The original purpose

A common task for drivers is allocating memory for buffers, that are accessible for DMA. Often there are some restrictions given by the hardware driven – the buffer must be placed in the lower 32bit addressable space, it has to be aligned, physically contiguous, or possibly not crossing a page boundary.

Even if the buffer is intended for the hardware use only (like xHCI scratchpads), the driver must keep the pointer to where the buffer is mapped inside its virtual address space in order to release the memory once it is not needed anymore. Hardware devices do not share the virtual address space with the task though, so the driver must always obtain also the physical address of the buffer. To satisfy all of these requirements, the HelenOS kernel offers a specialized API. The driver just needs to call a function `dma_map_anonymous`, and is given both the virtual and physical address. It is able to specify its requirements using flags. This API is powerful, but complex and inconvenient to use.

The authors of the former USB stack addressed the complexity by creating utility functions in `<usb/host/utils/malloc32.h>`. These functions offer a familiar interface of `malloc` and `free`. The `malloc32` function intentionally discards the physical address provided by `dma_map_anonymous`, in order to be simple to use. To retrieve the physical address later, one can use another utility function, `addr_to_phys`. This one is a simple wrapper for a syscall.

Even previous authors of the USB subsystem were aware of it being a syscall, and tried to cache the physical address where it would be used unnecessarily multiple times. The usage scheme of these functions then grew wild: the memory was allocated by `malloc32`, just to be translated by `addr_to_phys` on the next line. These two pointers were then stored inside the same structure.

Being aware of this use-case, we created a different submodule for allocation of DMA buffers. Instead of a plain pointer, the caller uses a structure called `dma_buffer_t`. When the buffer is no longer needed, it is freed by calling `dma_buffer_free`. As the translation to physical address is usually needed in a sequential manner shortly after allocation, we cache the physical address of the base of a contiguous chunk and compute the resulting physical address ourselves. Therefore, `dma_buffer_phys` might be used to efficiently translate a virtual pointer to a physical one.

When the buffer is contiguous in memory, it's easy to calculate the physical address ourselves, without a help of the kernel. To obtain a physical address for a concrete position in the buffer, one just needs to call `dma_buffer_phys`, which translates a virtual address pointing inside a buffer to a corresponding physical address.

One more reason to drop the previous `malloc32` helpers completely is that developers that are beginners to HelenOS might see `malloc32` as a weird name for an ordinary `malloc`, and start using it to allocate ordinary memory, which is very wasteful and expensive.

Even authors of the previous implementation started to overlook fact, that `malloc32` actually allocates a whole page, and implemented some operations very prodigally. The most notable occurrence was in transfer handling – every transfer descriptor and queue head was allocated in a separate memory page. From the performance point of view, allocation and clearing of several memory pages just to issue one short transfer is an extreme overhead. Furthermore, a TD can describe at most 8KB transfer, so a linear number of TDs are used. On the other hand, it is also waste of memory to allocate the memory in advance. So we at least merged all the needed structures to one allocated buffer to reduce the overhead to one page only.

3.6.2 Policies

To accomplish a goal described in the following chapter, we needed a way how to express constraints put on DMA buffers by individual drivers. After evaluating the individual needs, it came out that there are just two of them: a requirement of using 32-bit addresses, and a requirement of physical contiguity. However, the contiguity is not just binary (either fully contiguous or not). For example, xHCI-compatible controller is able to do bytewise scatter-gather. That means it can handle also non-contiguous buffers, translated as a scatter-gather on pages. On the other hand, every TRB can describe a 64KB block, so it can benefit from the buffer being contiguous by using less TRBs per transfer.

We used our benchmarking system to measure performance of issuing transfers. The scenario contains a QEMU virtual device (tmon) receiving our data and a software driver of this device issuing transfer. The most important aspect of the testing is that neither of these two entities actually care about the data being transmitted – the driver just allocates a buffer of garbage and sends it. The receiving side just counts the size of the received data. It is a completely artificial scenario, but allows us to measure the overhead induced by handling transfers, thus limiting the possible performance in real scenarios.

We measured the maximum performance of bulk transfers of a 64KB buffer. That is the maximum size of data transferred over IPC in HelenOS. When we relied on the buffer being contiguous, we could always issue single TRB per transfer. When not, we had to split the transfer to 16 TRBs. In the first case, the average performance was around 300 MB/s, in the second one, it dropped to 60 MB/s. That shows that the overhead is significant and we should use as large TRBs as possible.

It seems that the best solution would be to allocate chunks of 16 pages and split the data accordingly. However, using a buffer vector would introduce yet another API which would drivers have to use instead of a standard one. And with the experience with IO vectors from QEMU, we decided this is not a good option. The more when the same task can be solved by a slightly smarter kernel allocator, mapping the chunks into contiguous virtual memory. But anyway, we needed a way to express how big the chunks have to be.

Let us introduce another scenario, which could've worked if we had the power. Imagine a VFS server trying to copy 16 megabytes of data from hard drive to a flash drive. Let's say that the hard drive driver is capable of arbitrary scatter-gather operations, but handles 32-bit addresses only. On the other side, there is an xHC, which can handle 64-bit addresses, but it works best with 64KB-contiguous chunks. However, in between there sits a USB mass storage driver, which (just to demonstrate the idea) can issue only 32KB transfers, so it splits bigger ones to multiple calls. In an ideal world, the VFS driver would allocate a 16 MB buffer that is in the lower 4 gigabytes of memory, and every 32KB chunk of it is physically contiguous. Then it would share the buffer using IPC memory sharing capabilities, so that the hard drive driver loads the data in it. Then, without any copying, the VFS server would share the buffer all the way down to the xHCI driver, which would issue 32 KB transfers as ordered by the mass storage driver with the data directly, again without any further copying.

This level of copy-free efficiency is thought to be reasonably possible only in unikernel systems. But for an unknown reason we were too enthusiastic to let it be and decided to do a little experiment, designing a system which would allow this scenario to happen even in HelenOS.

The answer to all these issues is simple. We use a pointer-sized bitmap called `dma_policy_t` to carry flags. The flag on bit i denotes that the buffer may be split into 2^{i+1}B-sized chunks that are physically contiguous. As the minimal granularity is `PAGE_SIZE`, the lower bits can be used to carry different flags. Currently, we use just one to denote the 32-bit restriction.

The policy is a part of the pipe description, which is passed from HC driver to the USB device driver. It may than allocate a buffer according to this policy, and be sure that the HC driver will be able to use it directly. The policy a buffer is allocated with is passed along, for the HC driver to check. Because we generally trust the driver, there is no further checking. We do not trust the human writing it though so the policy is not automatically assumed but must be passed explicitly with the buffer pointer. This has another benefit – in case the policy is not exactly satisfied (e.g. the xHCI driver is given a non-contiguous buffer), it may decide that instead of bouncing it, which is expensive, it just uses a page-sized chunks for scatter-gather.

If this mechanism would be adopted by the rest of the system, the scenario described above would start with the VFS driver fetching policies from the lower-layer drivers. The HC driver would respond with a policy requiring 64KB chunks, but it would be modified by the mass storage driver to loosen the contiguity requirements, because it knows that the limit is actually 32KB. This can be done by a simple bitwise AND operation. From the other side, the hard drive driver would respond with a policy requiring 32-bit addresses. The VFS driver then computes a bitwise OR, creating a policy that satisfies both sides.

3.7 Memory Sharing

As promised in the previous section, we modified the interface of host controller driver. Instead of sending large data through IPC, which requires kernel to copy data from one task to another, we now use IPC memory sharing. While it might be actually slower for small transfers (latency benchmarks shows the difference is not relevant), it makes a big difference for large transfers. Utilizing the DMA policies described in the previous section, we achieved a copy-free path from USB device driver to the xHC.

By switching to memory sharing, we released the size limit of one transfer – the shared memory area might be arbitrarily large, as opposed to the 64KB artificial limit imposed by the kernel on data transfers. That introduced the need to actually split transfers to multiple TRBs if needed, so the benchmarked performance of previously described scenario dropped to 200 MB/s. However, by using transfers of size 16 MB, we can reach a measured bandwidth of around 1.3 GB/s. As the theoretical limit of the USB 3.1 (gen. 1) bus is 671 MB/s, we can say that the bottleneck is no longer in the xHCI driver.

As for the other drivers, the benchmarks on EHCI shown an improvement at 64KB blocks from 20 MB/s to 25 MB/s, but by using much bigger blocks (larger than 32 MB) we were able to reach 360 MB/s. Again, this is much bigger than the theoretical limit of 60 MB/s for the bus due to the test nature and virtual environment.

Chapter 4

Benchmarks and Testing

To demonstrate and benchmark performance of the xHCI stack, a proprietary subsystem has been implemented in HelenOS. The primary function of this subsystem is to communicate with a custom QEMU USB device over arbitrary endpoints and provide statistical information related to the communication. This way, it can be used to verify the correctness of message transmission, experiment with synchronization and to measure performance indicators.

In addition, since the subsystem is built on top of the USB device driver framework and has no xHCI-specific requirements, it can be also used to compare parameters of the xHCI stack with its predecessors.

The subsystem is composed of three parts:

QEMU fork with proprietary diagnostic device (usb-tmon) This implementation of QEMU contains a virtual USB device, which carries the diagnostic device class descriptor.

USB Diagnostic Device Driver (usbdiag) The usbdiag driver matches with the QEMU diagnostic device and facilitates all communication with it. It also exposes a remote interface for all HelenOS applications.

User Frontend Program (tmon) The tmon program is the primary user frontend in shell. It can use the interface exposed by usbdiag to perform various tests with diagnostic devices and return human-readable results.

4.1 QEMU Device

The `usb-tmon` virtual device is a diagnostic class USB device created to allow us to test all of the different transfer types on one device, gather data about the throughput and speed of the communication and validate the contents of the USB packets sent between HelenOS and a device.

In order to support communications of all types, the device contains an endpoint for each direction of each transfer type – i.e. interrupt, bulk and isochronous. Since we want to both check the speed of the driver and the correctness of the data sent, each endpoint is duplicated creating two sets of endpoints – first set, which does not validate the transferred data, and a second set, which checks that each four bytes of the data equal to a predefined macro `CHECK`. Each of these endpoints has an associated macro that contains its endpoint number which has the form of `EP_<type>_<direction>` for the first set and `CHECKED_EP_<type>_<direction>` for the second set, where `<type>` can be `INT` for interrupt transfers, `BULK` for bulk transfers or `ISOC` for isochronous transfers and `<direction>` can be either `IN` or `OUT`.

4.1.1 Monitoring

Regardless of which of the aforementioned endpoint sets is used, the `usb-tmon` device prints information about the transfers it handles to QEMU's standard output:

Interrupt Time since last `IN` or `OUT` interrupt request in microseconds.

Bulk Amount of bytes transferred in the last second for every second of an `IN` or `OUT` bulk transfer.

Isochronous Notification about receiving the request.

4.1.2 Implementation

The implementation of this device is located in the file `dev-tmon.c` in the helenos-xhci-team/qemu fork of the official QEMU repository. It contains several key structures and functions:

USBTmonState Structure that represents the current state of a `usb-tmon` device.

desc_tmon, desc_device_tmon, desc_iface_tmon These three structures form the descriptor of the device and contain information about the device's class, protocol, endpoints etc.

usb_tmon_class_init Called when QEMU starts, so it is used as a constructor function for the virtual device and all relevant data.

usb_tmon_realize Called when an instance of the `usb-tmon` device gets created and is used to initialize a specific instance of `USBTmonState`.

usb_tmon_handle_attach Called when an instance of the `usb-tmon` device gets attached to the guest OS.

usb_tmon_handle_control Called when the device receives a control request, in its current implementation simply forwards the request to QEMU via `usb_desc_handle_control`.

usb_tmon_handle_data Called when the device receives an interrupt, a bulk or an isochronous data request, determines the receiving endpoint, stores information about handled data and if needed, sends or validates a USB packet.

usb_tmon_{int|bulk|isoc}_{in|out} Called on specific kinds of transfers and track sent/received data.

These are the structures and functions one needs to modify in order to modify the behavior of the device. Additionally, the source code contains helper functions (e.g. time measurement with `get_now_sec` and `get_now_usec`) and QEMU debugging/informational functions and structures (e.g. `desc_strings`, `vmstate_usb_tmon`, `usb_tmon_info` and `usb_tmon_register_types`). These should seldom require modification.

For the purposes of modifying or debugging usb-tmon's source code, the header `usb.h` contains most of the structure definitions and function declarations that might be needed.

4.1.3 Usage

To attach an instance of the `usb-tmon` device to a running QEMU, one can type *"device_add usb-tmon"* into QEMU's monitor (which can be accessed via the Ctrl-Alt-2 key combination or by redirecting the monitor to QEMU's standard IO by adding *"-monitor stdio"* to QEMU's startup command). To have QEMU start with `usb-tmon` attached, they may add `"-device usb-tmon"` to their QEMU startup command.

4.2 Driver

The diagnostic device driver (known as `usbdiag`) is a proprietary device driver. Although it matches with all devices of the *Diagnostic* class, its main purpose is to communicate with the `usb-tmon` virtual device described in the previous section through a simple protocol.

To the rest of the system, the driver offers a simple interface, which allows to schedule tests to various endpoints of the diagnostic device. The output of such tests is a performance measurement (number of transfers performed and duration of all transfers) and optionally, a verification signalling that the data has not been malformed by the communication channel.

4.2.1 Structure

The `usbdiag` driver consists of the following source files:

`main.c` Main entry point, hooks for the USB device driver interface.

`device.h`, `device.c` Diagnostic device descendant of the `usb_device_t` structure, endpoint mapping.

`tests.h`, `tests.c` IN and OUT diagnostic tests, data verification, hooks for RPC.

4.2.2 Usage

Although the driver interface is open to all system applications, it is expected to be used only from the frontend application described in the next section. The exposed API is in detail described in `usbdiag_iface.h` and consists of four functions:

`usbdiag_connect` Open an IPC session with a diagnostic device.

`usbdiag_disconnect` Close IPC session with a diagnostic device.

`usbdiag_test_in` Initiate a synchronous test on an IN endpoint.

`usbdiag_test_out` Initiate a synchronous test on an OUT endpoint.

Quite obviously, the API follows a simple communication session paradigm. After a communication session is initiated, the user application can initiate multiple tests on the diagnostic device before closing the session and terminating the communication channel.

Since in order to initiate a session, the caller must already have a specific diagnostic device in mind, the `usbdiag` driver defines a `USBDIAG_CATEGORY`, which marks all DDF functions exposing the corresponding interface. This category can also be used to query the list of device functions using the `loc_category_get_svcs` function provided by the system *Location Service*.

4.3 Frontend

4.3.1 Structure

The `tmon` application consists of the following source files:

`main.c` Main entry point, command selection and usage string.

`commands.h` Executable commands.

`list.c` Implementation of the *"list"* command.

`tf.h`, `tf.c` Testing framework, common code for all "`test-*`" commands.

`resolve.h`, `resolve.c` Resolving DDF device from string using devman's IPC interface.

`tests.c` Implementation of IN and OUT test calls. Output formatting.

4.3.2 Usage

```
tmon: benchmark USB diagnostic device

Usage: tmon command [device] [options]

      list - Print a list of connected diagnostic devices.
      test-intr-in - Read from interrupt endpoint as fast as possible.
      test-intr-out - Write to interrupt endpoint as fast as possible.
      test-bulk-in - Read from bulk endpoint as fast as possible.
      test-bulk-out - Write to bulk endpoint as fast as possible.
      test-isoch-in - Read from isochronous endpoint as fast as possible.
      test-isoch-out - Write to isochronous endpoint as fast as possible.

      -t --duration
            Set the minimum test duration (in seconds).
      -s --size
            Set the data size (in bytes) transferred in a single cycle.
      -v --validate
            Validate the correctness of transferred data (impacts performance).

If no device is specified, the first device is used provided that it is the
only one connected. Otherwise, the command fails.
```

4.4 Massive Surprise Removal Testing

As a simple tool to test multiple scenarios regarding device removal, we wrote a shell script. It helped us find a lot of synchronization issues in hub driver and HC drivers. The setup is very simple – just run HelenOS in QEMU with a management socket located at `$SOCKFILE`:

```
$ qemu-system-x86_64 -qmp unix:$SOCKFILE,server,nowait \
        -device nec-usb-xhci -boot d -cdrom image.iso
```

Then, assuming a QEMU build in a directory `$QEMU_ROOT`, run the following snippet:

```
: ${repeats:=1} ${count:=1} ${driver:=usb-hub}
: ${in_delay:=5} ${out_delay:=$in_delay}

for rep in $(seq 1 $repeats); do
        for i in $(seq 1 $count);do
                echo "device_add driver=$driver id=burst-$i"
        done

        sleep $in_delay

        for i in $(seq 1 $count);do
                echo "device_del id=burst-$i"
        done

        sleep $out_delay
done | python2 "$QEMU_ROOT/scripts/qmp/qmp-shell" "$SOCKFILE" >/dev/null
```

The first two lines set default values for various parameters. There are several combinations which we consider interesting.

4.4.1 Fast Attach-Detach Test

In this test, we used a ridiculously big value of `repeats`, and very short value of `in_delay`, something like a tenth of a second. Running the script then simulates e.g. a bad cable, which stays connected only for a short while.

This test does not work correctly with xHCI, because of a bug in QEMU that cannot be worked around retaining correctness on real HW. QEMU correctly aborts transfers in a case of device removal, but any transfer scheduled later is just ignored. That's fine, the xHCI implementation issues a Stop Endpoint command to force HC release the ring and allow it to deallocate abort the transfer itself. Except that QEMU for some reason doesn't allow issuing a Stop Endpoint command for the default control endpoint. So, in case of bad timing, we can issue a transfer on EP 0 (like reading the device descriptor as a part of enumeration) after QEMU knows that the device is removed, but we don't know it yet. After that, we issue a Stop Endpoint command, which fails. Because we cannot know whether the endpoint is used or not after the command fails, we cannot simply abort the transfer. And because the disconnection handler fibril must make sure the enumeration fibril is not stopped, this results in an infinite waiting for QEMU response.

Note that this scenario simulates a real-world scenario, which does not happen in virtual environment. So we consider it is acceptable that this test fails because of QEMU.

Other HC drivers handled this test fairly well, because they do not maintain a soft state in the hardware, they just poll transfers for completion. Thus they know when it's safe to abort the transfer manually.

4.4.2 Balloon Test

We used a scenario with a large number of `repeats`, large `count`, and long enough delays (like 15 seconds). We simulate a lot of devices connecting, waiting for matters to stabilize, then disconnecting the devices again.

The main aspect this test is testing is resource leakage. When the resource usage grows still after the first few rounds, something must be wrong. This way, we found a leaking reference in `libdrv`, which resulted in devices never being dropped. That means that we were the first ever to actually remove a device from HelenOS.

Also, we discovered that limits of HelenOS are much bigger than those of QEMU. First, QEMU allows at maximum 64 slots per xHC to be enabled. Even though there are 127 valid USB addresses available, you cannot connect more than 64 devices at once. Even worse, after this limit is reached, QEMU will not report an error – it will simply enable a slot that is already enabled, resulting in correctly failed assertions in our code.

The second controller that is failing under QEMU is OHCI, which has a hard limit of 32 hops through the periodic queue. Because every hub (and HID device too) has an interrupt endpoint, it is added to the periodic queue. QEMU makes OHCI fail with an unrecoverable error after it reaches this limit of hops. Because of it, you can connect at most 30 devices to OHCI under QEMU.

4.4.3 Burst Test

The last parameter combination we tried was a bursting one. Using `repeats = 1`, `count = 60`, and `in_delay = out_delay = 4`. This combination connects a subtree of 60 hubs, and allows an arbitrary subtree of them to enumerate. The disconnection comes during the enumeration, which is a tough test for the hub driver. The more considering that the disconnection comes for all devices at once.

Running this test started the hub refactoring described in section 3.4, because neither the hub driver, nor our roothub at that time was prepared for such situations, and failed hard multiple times. Also, using this test we discovered the fundamental problem of flat hierarchy presented to DDF, which is explained in the section 3.4 as well.

4.5 Automated Removal Testing

For automated testing of the explicit device removal feature (further described in Section 3.2), a robust testing framework has been developed. Similarly to what was described in the previous section, this framework operates mainly by means of shell scripts, sequentially executing instructions over the QEMU Monitoring Protocol and dissecting the kernel logs generated by the OS.

The need for this tool is self-evident. With a multitude of HCI devices and a multitude of device drivers, it is humanly infeasible to thoroughly test explicit device removal in all possible configurations. Instead, the testing framework can execute simple test scenarios automatically, record the system logs and look for keywords indicating crashes and other failures. Humans can later only analyze logs with suspected failures, which constitute a much smaller portion of the entire whole.

The testsuite is available in the tests repository.

4.5.1 Unexpected Detachment Test

In this test, the following sequence of events is simulated:

1. HelenOS is booted up.
2. The tested device is attached.
3. The tested device is detached.

The aim of this test is to simulate a common user behavior, in which USB devices are removed without any prior hints to the system. In such a case, the HC driver is expected to recognize that the device is no longer connected, tear down its corresponding data structures and notify the *Device Manager*, which should remove its subtree from the DDF device tree.

In addition, the USB device driver is expected to handle `device_gone` callback gracefully, terminating all scheduled communications and notifying other userspace applications of the device demise.

4.5.2 Proper Detachment Test

In this test, the following sequence of events is simulated:

1. HelenOS is booted up.
2. The tested device is attached.
3. The DDF function corresponding to the device is brought offline.
4. The tested device is detached.

This test scenario simulates the correct user behavior for device detachment. The only difference with respect to that described in the previous section is that prior to physically disconnecting the device, the offline signal is issued, giving the HC and device drivers an opportunity to set their integral affairs in order prior to terminating communications.

The purpose of this test is to check whether both the HC and the device driver handle the offline signal and the `device_remove` callback properly and without deadlocks.

4.5.3 Suspend Test with Improper Detachment

In this test, the following sequence of events is simulated:

1. HelenOS is booted up.
2. The tested device is attached.
3. The DDF function corresponding to the device is brought offline.
4. The DDF function corresponding to the device is brought online.
5. The tested device is detached.

Again, this test scenario builds on the previous one, adding an extraneous online signal to the event sequence. This aims to simulate an unexpected device removal (not different from that in the Improper Detachment Test) following a temporary device suspension.

The purpose of this test is to verify whether the offline signal is fully reversible both in the HC and the device driver, and whether both drivers handle currently not only the offline signal, but the online signal as well.

Chapter 5

Conclusion

This chapter concludes the body of the project documentation. Following up to this point, its purpose has been to outline in detail all basic facts relevant to the HelUSB3 project. While the bulk of this text has been mainly concerned with the means of accomplishing the set project goals and implementation details leading up to it, the subsequent paragraphs only briefly list the accomplishments themselves for a quick and concise project summary.

In the introduction, the main goal of the project is stated as to bring xHCI and USB 3 support to HelenOS, and to generally refactor, fix, extend and clean up the existing USB framework implemented in HelenOS since 2011. In the official project specification (see Appendix B), these fairly general tasks are broken down into more specific requirements.

Throughout the duration of this project, a new host controller driver has been implemented to control various xHC models (among others, NEC Renesas uPD720200). The driver is capable of initiating and maintaining proper communication with the xHC in full compliance with [xHCI]. Its features include responding to port events, enumerating new devices, and scheduling data transfers at a multitude of speed levels ranging from USB 1.0 to USB 3.1. USB 3.2 released in September 2017 (during the project) is supported only partially as we had no opportunities to test it yet.

As explained in Section 3.1, the developed driver has been successfully integrated with the existing USB framework, allowing it to utilize a whole range of USB device drivers previously implemented in the system (for instance, HID such as mice, keyboards or tablets, mass storage devices and others). In order to maintain compatibility with previous HC versions as well as USB version abstraction from the point of device drivers, the USB framework has been heavily refactored. During this process, a large number of race conditions and other bugs were fixed not only in the USB framework, but also in older HC and USB device driver implementations. Consequently, the HelenOS USB framework is now up-to-date with USB 3, providing very similar interfaces to both device and HC drivers in the spirit of the original design.

Furthermore, the presented project has accomplished to implement the isochronous data transfer mode for time-critical devices and the explicit device removal feature, which allows device drivers to consistently terminate communication upon user signal before actual device detachment occurs. This feature can also be used to temporarily suspend operation of any connected USB device for an arbitrary time period without blocking the corresponding device driver. Since both of these features had no prior demonstrators, the existing USB device drivers have been modified to enable device removal and a new diagnostic device has been implemented to schedule isochronous transfers (along with its device driver and an application to control it). Lastly, bandwidth counting has been extended to track utilization of EHCI-controlled buses at High speeds.

The entire project implementation has been tested with success in a simulated environment as well as on real-world computers, achieving results comparable with other operating systems. In addition, critical components of the system have been thoroughly tested for high volume of incoming events and common user behaviors. The introduced and refactored source code is documented in in-code comments, documentation strings next to function declarations and this very document (and its referenced literature).

The accomplishments listed above satisfy all primary and some of the optional goals of the HelUSB3 project, as defined by its specification. The following sections describe how to access the attached CD, and discuss the possible future development following the end of this project.

5.1 Attached Materials

Apart from the attached CD, our work can be retrieved from a public Git repositories hosted at GitHub[1].

- the `helenos` repository contains the main product of our work, sources of the operating system itself
- the `qemu` repository contains Qemu 2.10 with the `usb-tmon` virtual device
- the `tests` repository contains automated tests described in sec. 4.5.
- the `docs` repository contains sources of this document
- the **attached CD** contains a ZIP archive with snapshots of `helenos`, `qemu` and `tests` repositories. It is also a bootable CD with ready-to-run HelenOS build with xHCI support. You can also run the system in QEMU (preferably version 2.10 or newer) with

  ```
  qemu-system-x86_64 -enable-kvm -device nec-usb-xhci,id=xhci -device
   ↪  usb-tablet -boot d -cdrom /dev/cdrom -m 1024
  ```

  The HelenOS image contains the sources of itself.

5.2 Try It Yourself

This section provides a brief crashcourse on how to get our work running.

After cloning our repository, HelenOS should be compiled as usual according to the official guide. Namely, the following two commands should suffice given that necessary build dependencies are installed:

```
$ ./tools/toolchain.sh amd64
$ make
```

The `make` command prompts with a config window. Select *amd64* default profile and turn off SMP because SMP support is currently broken in HelenOS [2] [3] [4]. Then simply submit it.

5.2.1 Running in QEMU

A reasonably recent (version 2.10 or newer) QEMU is required.

The script `ew.py`, which is the official way to launch QEMU with correct parameters, has been modified to include both an xHCI controller and a USB tablet device (as introduced in 3.5) by default. Therefore, one should just run:

```
$ ./tools/ew.py
```

After the system starts, the tablet device should function correctly. The log of the HelenOS xHCI driver can be viewed with

```
$ edit /log/xhci
```

Other related logfiles are `/log/usbhub`, `/log/usbhid` and `/log/usbmid`. You might want to raise the verbosity of log messages for the individual components, which is usually done by changing a log-level constant in `/uspace/drv/bus/*/main.c`.

[1]https://github.com/helenos-xhci-team
[2]http://www.helenos.org/ticket/461
[3]http://www.helenos.org/ticket/387
[4]http://www.helenos.org/ticket/388

5.2.2 Testing with TMon

`usb-tmon` (described in 4.1) is a custom USB device emulated by QEMU. `tmon` allows us to read and write data to and from endpoints types and provides some diagnostics.

To use `usb-tmon`, first compile our version of QEMU that contains the `usb-tmon` driver:

```
$ cd qemu
$ mkdir build; cd build
$ ../configure --target-list=x86_64-softmmu
$ make
```

Then start HelenOS in your freshly-compiled QEMU:

```
$ ./tools/ew.py -qemu_path ../qemu/build/x86_64-softmmu/
```

Then switch to the QEMU monitor (Ctrl+Alt+2) and enter `device_add usb-tmon`. Then switch back to HelenOS (Ctrl+Alt+1) and enter `tmon list` to the console. A new device should appear. Now you can start the test, for example with:

```
$ tmon test-bulk-in
```

5.2.3 Running on Real Hardware

Our work has been tested on a desktop with Teratrend μDP720202 Rev. 1.0 PCIe card and on a ThinkPad x240 laptop with Intel 8 series xHC (PCI ID 8086:9C31). Regarding peripherals, several no-name and branded USB2 and USB3 hubs, mice, keyboards, multi-interface keyboards (with multimedia keys) and flash drives have been tested.

It is possible to boot `image.iso` on the target computer, however booting over the network proved to be much more convenient during development. For netboot, we used GRUB instead of the more widespread pxelinux, as HelenOS uses `multiboot` protocol and pxelinux does not support it.

The following commands can be used to generate a PXE GRUB image:

```
$ grub-mkimage --format=i386-pc --output=core.img --prefix="(pxe)" pxe tftp
$ cat /usr/lib/grub/i386-pc/pxeboot.img core.img > grub2pxe
```

Then `/usr/lib/grub/i386-pc`, all files from `helenos/boot/distroot/boot/` and the following `grub.cfg` are placed to the TFTP server root and `grub2pxe` is booted.

```
set timeout=1
insmod multiboot
menuentry "helenos" {
  set root=(pxe)
  multiboot /kernel.bin
  module    /ns /boot/ns
  module    /loader /boot/loader
  module    /init /boot/init
  module    /locsrv /boot/locsrv
  module    /rd /boot/rd
  module    /vfs /boot/vfs
  module    /logger /boot/logger
  module    /ext4fs /boot/ext4fs
  module    /initrd.img /boot/initrd.img
}
```

5.3 Future Development

This chapter outlines features, which were considered optional extensions of the project but were not realized due to time, complexity or other constraints. These features provide good starting points for the future development of the USB stack.

5.3.1 Power Management

The xHCI Specification focuses a lot on power management. Devices have multiple states corresponding to the amount of power used, and links between devices do so as well. Of course, to fully utilize these features, there must have been a system-wide mechanism to declare intentions about power management. We are not there yet.

Despite that, there are a few power switches we could toggle just now. Given the small number of drivers implemented, we could for example make the USB fallback driver suspend the device.

We consider power management a topic that needs to be addressed on the system level, and as such we did not pay much attention to it.

5.3.2 Asynchronous I/O

The aspect we care about though is performance. Although the primary goal of this project was to allow users to use USB on machines equipped only with xHC, the performance benefit of USB 3 is not negligible, and could be well worth it.

Every transfer type of USB focuses on different characteristics of the transfer. Interrupt pipes strive for the lowest latency possible – in this field, the synchronous semantics works very well. The latency between receiving interrupts and delivering them is as fast as an IPC reply to a call.

Considering isochronous endpoints, it is simply not possible to satisfy them with a synchronous API. For that reason, we decided to change the semantics to asynchronous one. The downside of the current approach is that the HC driver is forced to copy the data out from the shared buffer, because its ownership is temporary. But since isochronous transfers in xHCI are just a proof-of-concept and there are no drivers for real devices yet, we have decided not to optimize prematurely.

The bulk transfer type can however be utilized both synchronously and asynchronously, and both use cases have their advantages as well as disadvantages. While synchronous API is easier to work with, asynchronous would offer much better performance though. However, as there is just a single driver using bulk endpoints, we have decided not to complicate matters and stay with synchronous-only API for bulk pipes.

5.3.3 Isochronous Transfers in UHCI+OHCI+EHCI

The isochronous module implemented in xHCI can be thought of as a generic scheduling framework for isochronous transfers. The scheduler behaves like a leaky bucket, scheduling the transfers to xHC at a constant rate while throttling the device driver. This is the most complicated part of handling isochronous transfers, and implementing an asynchronous API for drivers is a good opportunity to generify it for other drivers as well.

Implementing the support for isochronous transfers for older HCs is a bit harder task, though. Those drivers use proprietary data structures for isochronous transfers, and require the software to schedule transfers so all time constraints are satisfied. (The xHC does it in hardware, and software is just responsible for issuing transfers fast enough.) Also, because of a heavily simplified implementation of periodic scheduling in the former HC drivers, substantial refactoring is inevitable. That however requires a deeper understanding of inner workings of individual HCIs.

Referenced Literature

This chapter lists all documents and websites relevant to or cited by this documentation. If interested, the reader is referred to these for more detailed explanation of mechanisms, architectures, paradigms and technologies referenced in this report.

Universal Serial Bus

[EHCI] Intel Corporation. *Enhanced Host Controller Interface Specification for Universal Serial Bus*. Version 1.0. 2002.
URL: `https://www.intel.com/content/www/us/en/io/universal-serial-bus/ehci-specification-for-usb.html`.

[xHCI] Intel Corporation. *eXtensible Host Controller Interface for Universal Serial Bus*. Version 1.1. 2013.
URL: `https://www.intel.com/content/www/us/en/io/universal-serial-bus/extensible-host-controler-interface-usb-xhci.html`.

[USB 2.0] *Universal Serial Bus 2.0 Specification*. 2000.
URL: `http://www.usb.org/developers/docs/usb20_docs/`.

[USB 3.2] *Universal Serial Bus 3.2 Specification*. 2017.
URL: `http://www.usb.org/developers/docs/`.

HelenOS

[1] *About HelenOS*. 2017.
URL: `http://www.helenos.org/`.

[2] *Compiling HelenOS From Source*. 2018.
URL: `http://www.helenos.org/wiki/UsersGuide/CompilingFromSource`.

[3] *DeviceDrivers*. 2017.
URL: `http://www.helenos.org/wiki/DeviceDrivers`.

[4] *Knowledge Articles on HelenOS Wiki*.
URL: `http://www.helenos.org/wiki/KnowledgeArticles`.

[5] Lenka Trochtová. "Rozhraní pro ovladače zařízení v HelenOS". 2010.
URL: `https://is.cuni.cz/webapps/zzp/detail/47852/`.

[6] *USB subsystem in HelenOS*.
URL: `http://helenos-usb.sourceforge.net/`.

Appendix A

Writing USB Drivers

This appendix provides documentation and general guidelines for usage of the refactored USB device driver interface. As such, it is meant for developers of HelenOS device drivers, who can use it as a reference guide. By intention, this section abstracts the reader from all modifications to the stack, and focuses only on the latest state of the interface. All changes to the interface are described in detail in Chapter 3.

The reader should note that there exists a similar section in the initial USB stack documentation. The text in this appendix could be considered an updated version or extension of that text.

A.1 Basics

This section gives details on the basic structure of USB device drivers, their role in the system and components they usually interact with.

A.1.1 Framework

USB device drivers use the generic *Device Driver Framework* available in HelenOS. Because all USB drivers have similar initialization routines, a thin layer – specific to USB devices – was added above the generic one. This layer mainly serves as a middleware for easy communication with USB host controller drivers, performing USB specific resource management and enabling device drivers to initialize endpoint pipes and interact with USB devices. For those reasons, USB device drivers are recommended to be linked with `libdrv` and `libusbdev`, which contain both aforementioned layers respectively.

It is expected that USB device drivers specify in advance not only their relevant match identifiers, which are used by the Device Manager to pair new devices with available drivers, but also all endpoints, which shall be present on the device through a USB driver structure. Later when a new device is found, a specialized device structure is prepared and pipe abstractions are initialized.

Device drivers live the same life cycle as any other drivers controlled by the *Device Manager*. A quick summary follows:

1. The driver is started at the convenience of the Device Manager if and when a compatible device is found. At startup, the driver registers with the USB framework, which in turn registers also with the Device Driver Framework.

2. During its lifetime, the driver receives callbacks from the USB Framework, informing it about relevant device events. On the basis of these events, the driver then communicates with the device and exposes various interfaces to other tasks in the system. At this point, the controlled device usually becomes visible and useful to the user.

3. When there is no more need for the driver to run (i.e. no devices to control), the Device Manager may terminate the driver to save resources.

In the Device Driver Framework, drivers are the consumers of *devices* and providers of *functions*. This paradigm allows them to expose an unlimited number of nodes (representing logical or physical units) for every device they control. The same basic principle holds for USB drivers as well.

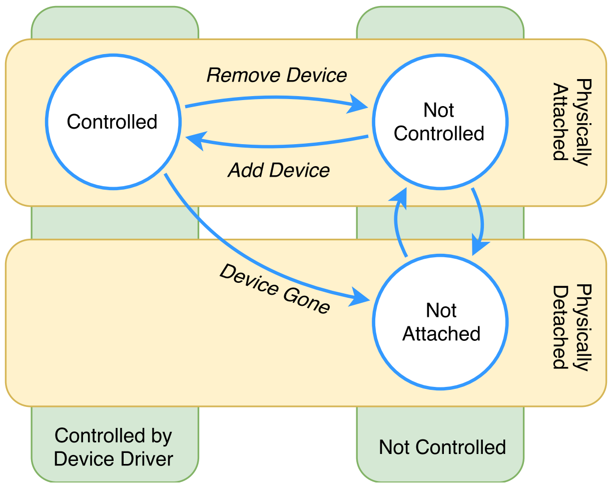

Figure A.1: Diagram of USB device states and transition events, as perceived by the device driver. Note that these states do *not* correspond to USB bus device states (e.g. *Configured*, *Addressed*, etc.) in any way.

A.1.2 Device Callbacks

As explained in the previous section, USB device drivers are informed about relevant device events by asynchronous callbacks from the USB framework. To simplify usage, these callbacks are identical to those of the Device Driver Framework:

Add Device This event notifies the driver that a new device has been discovered and matched to it. From this point on, the driver is allowed to communicate with the device in order to configure it and expose its functions to the rest of the system. Further communication with the device will likely depend on remote calls originating from other system tasks utilizing the exposed interface.

Remove Device This event instructs the driver to immediately disallow new user operations on a device, terminate all currently running operations in a timely manner, and hand device control back to the system, as the device will likely be physically removed from the bus in the foreseeable future.

Device Gone This event informs the driver that a device has been physically disconnected from the system without a prior *Remove Device* event. Since the device is no longer reachable, the driver is to force interrupt all user operations, which were running at the time of receiving the event and report failure to the callers.

Offline Function By receiving this event, the driver is asked by the system to explicitly transition a specific function exposed by one of its controlled devices into the *Offline* state. The meaning of such transition might depend on the interpretation of the function. For more information, see the Device Driver Framework Documentation[3].

Online Function By receiving this event, the driver is asked by the system to explicitly transition a specific function exposed by one of its controlled devices into the *Online* state. Again, the meaning of such transition might depend on the interpretation of the function. For more information, see the Device Driver Framework Documentation.

The listed events are delivered to device drivers through function calls to event handlers specified in the main USB driver operations structure (see Listing A.1 for a minimal example). The order of event delivery models the physical lifecycle of the device within the system. For instance, it is not possible for the *Add Device* event to occur more than once for the same device in a row. For reference, the exact device states and transitions between them (corresponding to the events with respective names) are shown in Figure A.1.

Furthermore, since there are no guarantees on the synchronization between calls, the driver is explicitly responsible for synchronizing accesses to its private data structures as well as device communication.

```c
static int device_add(usb_device_t *dev)
{
        usb_log_info("Device '%s' added.", usb_device_get_name(dev));
        return EOK;
}

static int device_remove(usb_device_t *dev)
{
        usb_log_info("Device '%s' removed.", usb_device_get_name(dev));
        return EOK;
}

static int device_gone(usb_device_t *dev)
{
        usb_log_info("Device '%s' gone.", usb_device_get_name(dev));
        return EOK;
}

static const usb_driver_ops_t driver_ops = {
        .device_add = device_add,
        .device_remove = device_remove,
        .device_gone = device_gone,
};
```

Listing A.1: Main USB device driver operations structure with corresponding event handlers. Note that events *Offline Function* and *Online Function* do not require a handler and can therefore be omitted in the minimal example.

A.2 Device Communication

In USB, *a pipe* is an abstraction primitive for a communication channel between a device and a host computer. For the sake of simplicity, the provided framework relies on a mechanism based on the same abstraction in order to facilitate communication between device drivers and devices.

A.2.1 Endpoint Description

In order to use pipes, device drivers must first define at least one *endpoint description*. The purpose of such definition is to specify all device endpoints, which will be used for communication by the device driver throughout its lifecycle. For that reason, all descriptions ought to be specified in advance and referenced in the main device driver structure (for example of possible endpoint description definition, refer to Listing A.2).

Endpoint description contains information which can be matched to USB endpoint descriptors in very much the same way as match identifiers are used to pair devices with device drivers in HelenOS. Following this scheme, the presented USB framework does most of the heavy lifting for device drivers. When a new device is added, prior to delivery of the *Add Device* event, all endpoint descriptions provided by the driver are matched to device endpoints, resulting in two possible outcomes for every description:

1. A device endpoint matching the provided description is found and *an endpoint mapping* is created.
2. No device endpoint matches the provided description, hence no mapping is created.

Device drivers can later query the result of the matching, and if successful, retrieve the created endpoint mapping containing a fully initialized pipe instance. An example of this is shown in Listing A.3.

A.2.2 Pipe I/O

Provided that an endpoint description has been defined, successfully matched and an endpoint mapping has been retrieved along with a pipe instance, a device driver can schedule data transfers to a device.

```
static const usb_endpoint_description_t bulk_in_ep = {
        .transfer_type = USB_TRANSFER_BULK,
        .direction = USB_DIRECTION_IN,
        .interface_class = USB_CLASS_MASS_STORAGE,
        .interface_subclass = USB_MASSSTOR_SUBCLASS_SCSI,
        .interface_protocol = USB_MASSSTOR_PROTOCOL_BBB,
        .flags = 0
};

static const usb_endpoint_description_t bulk_out_ep = {
        .transfer_type = USB_TRANSFER_BULK,
        .direction = USB_DIRECTION_OUT,
        .interface_class = USB_CLASS_MASS_STORAGE,
        .interface_subclass = USB_MASSSTOR_SUBCLASS_SCSI,
        .interface_protocol = USB_MASSSTOR_PROTOCOL_BBB,
        .flags = 0
};

static const usb_endpoint_description_t *endpoints[] = {
        &bulk_in_ep, &bulk_out_ep, NULL
};
```

Listing A.2: Main USB device driver endpoint description array. This particular example shows two *Bulk* endpoints for SCSI mass storage data transfers in both the *In* and *Out* directions.

Pipes offer a very simple synchronous command interface similar to that of an open file descriptor, namely `usb_pipe_read` and `usb_pipe_write` for reading from IN endpoints and writing to OUT endpoints respectively. While semantics of these functions depends on endpoint types (consult USB specification for details), their basic usage is as follows:

1. Device driver allocates a buffer of appropriate size (and fills it with data if writing).
2. Either the `usb_pipe_read` or `usb_pipe_write` function is called, scheduling transfers to the device. The function receives a pipe pointer, pointer to the user-allocated buffer and data size.
3. The calling fibril is blocked until transfers succeed or fail (see error code). If reading, the number of read bytes is returned as well.
4. Device driver recycles or disposes of the allocated buffer.

Note that while the write function blocks the calling fibril until the entire buffer is transferred to the device, the read function might return successfully with a lower number of bytes read than the actual data size requested, and may thus have to be called multiple times.

In addition, due to the multiplatform nature of HelenOS, it is not advisable to assume endianity of the host system. Instead, the `uint{16|32}_host2usb` and `uint{16|32}_usb2host` macros serve to convert host system endianity to and from the transfer endianity defined in the USB Protocol Specification at driver's convenience. A complete example of pipe write is shown in Listing A.4.

A.2.3 Automated Polling

In USB devices which often interact with a human user, it is important to wait for user input by spinning (or polling) on some *Interrupt IN* endpoint. In the abstraction used, this translates to calling `usb_pipe_read` in a perpetual loop and responding to incoming data or errors. Since this behavior is common to a significant class of drivers, a reusable unified implementation is provided by the USB framework.

The implementation is represented by the `usb_polling_t` structure and its associated functions. Similarly to device event handlers, device drivers are expected to configure this structure with polling

```c
static int device_add(usb_device_t *dev)
{
        /* Find mapping for the endpoint description. */
        usb_endpoint_mapping_t *mapping = usb_device_get_mapped_ep_desc(dev,
        ↪  &bulk_out_ep);

        /* Determine if the mapping was successful. */
        if (!mapping || !mapping->present) {
                usb_log_error("Endpoint mapping failed!");
                return ENOENT;
        }

        usb_pipe_t *pipe = &mapping->pipe;

        /* Now we can write to ‘pipe‘. */
        return write_data(pipe);
}
```

Listing A.3: Sample implementation of the *Device Add* event handler, obtaining a mapping and a USB pipe from one endpoint description defined in Listing A.2.

handler functions. Later when polling should be initiated, a call to the usb_polling_start function will spawn a separate fibril, spinning on the pipe and calling handlers when data arrive or usb_pipe_read fails with error.

When polling is to be ceased, a call to the usb_polling_join function will spontaneously wake up the polling fibril and block the caller fibril until its termination. If polling is started, a call to this function must *always* occur no later than in the *Remove Device* or *Device Gone* event handler in order to prevent the fibril from outlasting the device lifecycle within driver private data structures..

Furthermore, note that once any device driver starts polling on a pipe, it transfers the pipe ownership to the polling fibril, relinquishing all control over it. That prohibits subsequent calls to usb_pipe_read or other relevant functions. It is also not feasible to poll only for a limited time period, then join and reuse the pipe for arbitrary data reads. This is due to the fact that once usb_polling_join is called, the pipe is left closed for all communication, destroying the underlying endpoint mapping.

An example usage of the pipe polling interface is shown in Listing 3.2.

A.3 Miscellaneous

This section features general suggestions and recommendations for USB device driver implementation.

A.3.1 Device-specific Data Structures

Device drivers often need to maintain stateful and descriptive information related to their controlled devices. A common practice is to store this kind of information in device-specific data structures. As their name suggests, such structures usually correspond to a single controlled device, and can thus closely follow its lifecycle, being allocated and configured when the device is added and being freed upon its removal.

The presented USB framework offers device drivers a straightforward mechanism to associate any of their data structures with USB devices by specifying custom *device data* in the usb_device_t structure, which is present in all device-related events. Device data is allocated at most once for every device and can have any driver-specified, albeit constant size. When the device is later removed from the driver, device data is automatically freed if present.

Allocation of device data is performed by a call to the usb_device_data_alloc function. This function has the same return value and semantics as the calloc function. In addition, the pointer returned by this function is stored within the usb_device_t structure and can be retrieved from that

```c
static int write_data(usb_pipe_t *pipe)
{
        int rc = EOK;
        /* Allocate memory. */
        static const size_t size = 64;
        char *buffer = (char *) malloc(size);
        if (!buffer) {
                rc = ENOMEM;
                goto err;
        }

        /* Fill `buffer` with arithmetic sequence of bytes. */
        for (int i = 0; i < size; ++i) buffer[i] = (char) i;

        /* Write data. */
        rc = usb_pipe_write(pipe, buffer, size);
        if (rc != EOK) {
                usb_log_error("Write failed with error: %s", str_error(rc));
                goto err_buffer;
        }

        /* Clean up. */
err_buffer:
        free(buffer);
err:
        return rc;
}
```

Listing A.4: Example write to a USB pipe. This is a possible implementation of the `write_data` function referenced in Listing A.3.

point onward until the device is removed by a call to the `usb_device_data_get` function. Example usage of this mechanism can be seen in Listing A.5.

A.3.2 Logging

Another useful feature of the USB device driver framework is a set of logging macros. These macros copy the scheme common to the HelenOS logging framework, offering various log levels a format string syntax for a variable number of arguments consistent with the `printf` function syntax. With decreasing severity, the presented logging macros are named as follows:

`usb_log_fatal` for fatal errors,

`usb_log_error` for recoverable errors,

`usb_log_warning` for warnings,

`usb_log_info` for informational messages (produces a log message of the *Note* level),

`usb_log_debug` for debugging messages,

`usb_log_debug2` for verbose debugging messages.

Prior to printing the first log message, the `log_init` function must be called. During runtime, the default HelenOS log level can be adjusted by a call to the `logctl_set_log_level` function. Keep in mind that in the current implementation of HelenOS, every call to a logging macro results in an IPC call. For that reason, it is not advisable to perform logging in performance-sensitive parts of the code, even if the log level is sufficiently high.

For more information about logging in HelenOS, consult the HelenOS Documentation.

```c
typedef struct my_data {
        /* Any device-specific data here. */
        usb_polling_t polling;
} my_data_t;

static int device_add(usb_device_t *dev)
{
        /* Allocate device data. */
        my_data_t *data = (my_data_t *) usb_device_data_alloc(dev,
         ↪  sizeof(my_data_t));
        if (!data) {
                return ENOMEM;
        }

        usb_polling_init(&data->polling);
        /* ... configure polling ... */
        usb_polling_start(&data->polling);

        return EOK;
}

static int device_remove(usb_device_t *dev)
{
        /* Retrieve device data. */
        my_data_t *data = (my_data_t *) usb_device_data_get(dev);

        /* Stop polling. */
        usb_polling_join(&data->polling);
        usb_polling_fini(&data->polling);

        /* After returning, device data is freed automatically. */
        return EOK;
}
```

Listing A.5: Example device data allocation and usage. In this case, the `my_data_t` device-specific data structure is used to keep track of automated pipe polling. Note that some error conditions have been intentionally ignored for the sake of brevity.

A.3.3 Exposing an Interface

In order for controlled hardware to become usable to system tasks, HelenOS device drivers can expose IPC interfaces to abstract hardware-specific features and provide a set of well-defined logical operations for device interaction.

In HelenOS IPC, one of the preferred ways to achieve this is by interacting with *services* – system tasks, which run as daemons, and whose purpose is to connect user tasks with device drivers. Since services usually keep track of devices of the same category, it is necessary to first identify the appropriate service for a driver and consult its documentation.

While the specifics of exposing IPC interfaces may vary for different device categories, the general scheme remains the same:

1. For a DDF function, fill a specific structure with configuration and call handlers Then register them with the respective system service.

2. Respond to service-specific calls in compliance with service specification. Handling of these calls will likely result in communication with the device.

3. When the device is removed, cease handling service calls and unregister the device from the service.

A.3.4 Descriptor Parser

While they are sometimes stored in a serialized form, USB descriptors can be viewed as a tree. For instance, configuration descriptor is followed by interface and endpoint descriptors, which are not directly accessible. To aid with problems such as this, a simple descriptor parser is provided by the presented framework.

The parser has been designed to be as generic as possible and its interface might thus be somewhat terse. It operates on a byte array and offers functions for finding the first nested descriptor and its next sibling (i.e. descriptor on the same depth of nesting). The parser expects that the input is an array of bytes where the descriptors are sorted in *prefix tree traversal* order. Next, it expects that each descriptor has its length stored in the first byte and the descriptor type in the second byte. That corresponds to standard descriptor layout. The parser determines the nesting from a list of parent-child pairs that is given to it during parser initialization. This list is terminated with a pair where both parent and child are set to -1.

The parser uses the `usb_dp_parser_t` structure, which contains array with possible descriptor nesting (`usb_dp_descriptor_nesting_t`). The data for the parser are stored in the `usb_dp_parser_data_t`, the arg field is intended for custom data.

For processing the actual data, two functions are available. The `usb_dp_get_nested_descriptor` function takes a parser, parser data and a pointer to parent as parameters. For retrieving the sibling descriptor (i.e. descriptor at the same depth) one can use `usb_dp_get_sibling_descriptor`. Whereas this function takes the same arguments, it also requires two extra arguments — pointer to the parent descriptor (i.e. parent of both nested ones) and a pointer to the preceding descriptor. Such pointer must always point to the first byte of the descriptor (i.e. to the length of the descriptor, not to its type).

Additionally, there exists a simple iterator over the descriptors. The function `usb_dp_walk_simple` takes a callback as a parameter, which is executed for each found descriptor and receives thee current depth (starting with 0) and a pointer to current descriptor.

Appendix B

Project Specification and Timeline

This section contains the original project specification, as it was approved by the project committee at Faculty of Mathematics and Physics, Charles University. The actual project realization timeline is also included.

B.1 Project Specification (in Czech)

B.1.1 Základní informace

Jméno projektu HelenOS USB 3.x Stack

Zkratka HelUSB3

Vedoucí Martin Děcký `decky@d3s.mff.cuni.cz`

Konzultanti –

Anotace Rozšíření podpory sběrnice USB a USB zařízení v mikrojádrovém multiserverovém operačním systému HelenOS o revizi 3.0 (resp. 3.1), sjednocení této podpory s dříve implementovanou funkcionalitou pro USB 1.0, 1.1 a 2.0 a další vylepšení.

B.1.2 Motivace

Podpora sběrnice USB v revizi 1.0 a 1.1, která byla naimplementována v rámci Softwarového projektu v roce 2011 (HelUSB), výrazně rozšířila praktickou použitelnost operačního systému HelenOS v souvislosti s periferními zařízeními. Na tuto funkcionalitu bylo později navázáno podporou USB 2.0. V současné době je poslední specifikovaná revize USB 3.0 (z hlediska hardwarového transportu potom 3.1) a začínají se objevovat počítače, které implementují pouze tuto revizi (tedy z pohledu operačního systému nejsou zpětně kompatibilní se staršími revizemi). Proto je vhodné, aby byl operační systém HelenOS doplněn o nativní podporu pro USB 3.0 (resp. 3.1). Tento projekt je pochopitelně také vhodnou příležitostí pro provedení sjednocení s předchozími implementacemi a provedení dalších souvisejících vylepšení.

B.1.3 Popis projektu

Primárním předmětem projektu je rozšířit generický framework pro použití USB sběrnic a implementaci ovladačů USB zařízení v systému HelenOS o podporu USB revize 3.0 (resp. 3.1), při zachování plné podpory předchozích revizí 1.0, 1.1 a 2.0. Cílem je, aby bylo možné na běžně dostupném hardwaru využít nejvyšší možnou revizi USB s přihlédnutím k možnostem daných periferních zařízení. Součástí projektu je také implementace základního demonstrátoru – ovladače konkrétního fyzického USB host controlleru a ověření funkcionality konkrétních koncových USB zařízení.

- Implementace ovladače pro USB host controller podle normy xHCI.
  - Podpora přenosových režimů a rychlostí odpovídajících USB 1.0, 1.1, 2.0 a 3.0.

    - Enumerace zařízení na sběrnici USB 3.0 a spouštění příslušných ovladačů koncových USB zařízení (s využitím existujícího Device Driver Frameworku), jsou-li k dispozici.
        * Zachování kompatibility s dříve naimplementovanými ovladači host controllerů podle norem OHCI, UHCI a EHCI.
        * Ideálně zachování možnosti implementovat ovladače koncových USB zařízení nezávisle na použité variantě host controlleru.
    - Demonstrátor: Ovladač pro xHCI host controller NEC Renesas uPD720200
- Implementace explicitního mechanismu odpojování USB zařízení (očekávaného i neočekávaného).
    - Podpora přerušení USB komunikace (USB communication abort) na hardwarové úrovni bez zablokování ovladače USB zařízení.
- Implementace podpory isochronního režimu komunikace USB zařízení.
- Volitelná část zadání: Rozšíření USB frameworku o dosud nepodporované vlastnosti (např. správa napájení), o podporu specifických xHCI host controllerů (např. Intel Sunrise Point-H a/nebo contollery integrované na vývojových deskách či SoC čipech typu BeagleBoard, BeagleBone, Raspberry Pi) či jiná vylepšení (např. implementace nových ovladačů koncových USB zařízení jako je DisplayLink, dokončení implementace správy přenosového pásma a výkonu, odladění ovladačů controllerů na jiných platformách).

B.1.4 Platforma, technologie

- Framework a ovladače budou odladěny v simulátoru QEMU a na běžném fyzickém PC (obojí pro platformy x86 a x86-64) vybaveném výše uvedeným USB controllerem a případně kombinací již dříve podporovaných USB controllerů s použitím již existujících ovladačů koncových USB zařízení.
- Framework a ovladače budou implementovány takovým způsobem, který nebude principielně bránit jejich budoucímu využití na jiných platformách než x86 a x86-64.
- Framework a ovladače budou implementovány takovým způsobem, aby zachovávaly celkovou softwarovou architekturu mikrojádrového multiserverového operačního systemu HelenOS a aby nedošlo k omezení již dříve naimplementované a odladěné funkcionality (tj. podpora OHCI, UHCI atd.).

B.1.5 Odhad náročnosti

Na základě zkušeností z dřívějšího Softwarového projektu HelUSB (implementace podpory OHCI, UHCI) lze říci, že zadání je řešitelským týmem o 5 – 6 studentech zvládnutelné ve standardní době. Projekt klade na řešitele především následující nároky, ze kterých přirozeně plyne přibližný harmonogram prací:

- Nastudování specifikace sběrnice USB, specifikace xHCI, specifikace controlleru NEC Renesas uPD720200, volitelně studium implementací v jiných operačních systémech. [1 měsíc]
- Nastudování softwarové architektury, relevantních mechanismů a existující implementace systému HelenOS (async framework, Device Driver Framework, podpora OHCI, UHCI, EHCI). [1 měsíc]
- Implementace podpory xHCI a controlleru NEC Renesas uPD720200, integrace s existující implementací, refactoring. [2 měsíce]
- Implementace explicitního mechanismu odpojování USB zařízení, přerušení USB komunikace a isochronního režimu komunikace. [1,5 měsíce]
- Implementace vhodné podmnožiny volitelných částí zadání. [2,5 měsíce]
- Důkladné odladění výsledné implementace, sepsání dokumentace. [1 měsíc]

Aktuální stav systému HelenOS poskytuje dostatečně stabilní základ pro úspěšnou realizaci projektu, riziko nedokončení projektu je malé. Zadání projektu záměrně klade důraz na elegantní integraci výsledného řešení s existujícím kódem včetně nutného refactoringu. Nelze pochopitelně zcela vyloučit nutnost opravovat drobné chyby v existující implementaci, které mohou být v průběhu práce na tomto projektu odhaleny, nemělo by se však jednat o zásadní překážky.

Protože časovou náročnost implementace povinných bodů zadání není možné předem zcela spolehlivě odhadnout, počítá zadání také s volitelnými částmi, které by posloužily jednak pro zlepšení celkové funkcionality USB v systému HelenOS a jednak umožnily dosáhnout za všech okolností obvyklého objemu implementačních prací v rámci Softwarového projektu. V případě, že povinné části zadání zaberou odhadovaný čas, bude možné implementovat vhodnou podmnožinu volitelných částí. Ukáže-li se naopak, že odhad časové náročnosti povinných částí byl podhodnocen, resp. nadhodnocen, potom bude možné omezit, resp. rozšířit, podíl implementace volitelných částí.

B.1.6 Vymezení projektu

Projekt je zaměřen na následující oblasti:

- softwarové inženýrství,
- vývoj software,
- systémové programování,
- spolehlivé systémy.

B.2 Project Timeline

This section contains the full timeline of project realization throughout the years 2017 and 2018.

2017	**April**	**18**	Project specification finalized.
	May	**1**	Project officially started.
	May		Studying USB, xHCI specifications, HelenOS wiki. Hello Worlds.
	June	**8**	First commit in the project branch.
	June		Skeleton of the xHCI driver structure. Supportive structures.
	July		Command interface, event handling.
	August		Root hub. First signs of dialog with the HC.
	October		Transfers, debugging. The great usbhost refactoring.
	October	**17**	USB mouse driver transmitting movement data over the xHCI stack.
	November		Debugging, refactoring.
	November	**12**	HelenOS 0.7.1 was released. Changed VCS from Bazaar to GIT.
	December		Transfer Monitor, testing, benchmarking.
2018	January		Device removal, isochronous transfers, USB 3 extensions.
			Testing, debugging, writing documentation.
	January	**30**	Project considered delivered by the supervisor.
	February		Merging into mainline, finishing touches.
	February	**28**	Project submitted for defense.

Table B.1: Project timeline.