USB subsystem in HelenOS

USB subsystem in HelenOS

Matúš Dekánek
Vojtěch Horký
Matěj Klonfar
Ľuboš Slovák
Ján Veselý

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction to HelenOS USB subsystem

This manual describes support for USB devices in HelenOS operating system. The manual is intended primarily for developers of device drivers and host controller drivers. However, even end users of HelenOS or authors of end-user applications interacting with USB driver can find interesting information here.

HelenOS is a microkernel operating system where most of the functionality is provided by userspace tasks rather than system calls. This includes file system service, networking and also device drivers.

USB is a standard for communication between host computers and computer peripherals. The advantage of USB over other kinds of connection between a computer and a peripheral is its flexibility. USB supports plugging and unplugging of devices without need to turn off any of the components. The data flow model is designed to accommodate together peripherals with different transfer characteristics.

Adding support for USB devices to HelenOS shall improve the usability of the system in general even if the support concerns only USB 1.1 and the only peripheral drivers are for keyboards and mice.

Also, the support can be thought of as a proof of concept. Proof that it is possible to write drivers completely in userspace with only minimal support of kernel. Actually, changes introduced by the USB team to kernel were only cosmetic and no huge changes were needed. It also proves that userspace drivers can be divided into relatively high number of standalone tasks (processes) that are relatively simple, thus making the system more robust. Technically, the support can be also viewed as a demonstration that the new framework for writing device drivers is well designed and works. The framework was added to HelenOS recently (state in the time of writing this manual — first half of 2011).

1.1 USB support in other operating systems

Major operating systems like MS Windows, Linux distributions, UNIXes, BSDs and Solaris include full support for USB 1.1 and 2.0 as well as drivers for various host controllers. This is not surprising as some OS vendors contributed to USB development, and the specification has been available for some time.

Among experimental and academic OSes like Haiku, GNU Hurd and Minix, only Haiku includes full USB support. Neither of the two micro-kernel based systems (Hurd, Minix) include USB support, this makes HelenOS's USB implementation a trailblazer.

Chapter 2

HelenOS overview

This chapter is a brief overview of HelenOS operating system. It cannot and does not want to replace HelenOS design documentation[1] and information spread on community wiki[2] but it provides a summary of what the reader shall know about HelenOS prior reading this documentation. For some chapters, no or little knowledge of HelenOS is required but if you plan to write your own drivers, you will need to know something about HelenOS.

2.1 In lieu of introduction

HelenOS was started as a student software project in 2004 with the aim to build a portable microkernel operating system. The main goal was to create lightweight kernel supporting several different architectures with clean design and implementation.

Since then HelenOS has been used as a base platform for several bachelor and diploma theses. Apart from its microkernel nature the reason for using HelenOS for further development and research is its clean design and small size. At the beginning of 2011, over 20 students were participating on HelenOS as a part of their theses or other projects.

From the beginning, HelenOS was developed with portability in mind. Although the number of supported architectures is high (AMD64/EM64T (x86-64), ARM, IA-32, IA-64 (Itanium), 32-bit MIPS, 32-bit PowerPC and SPARC V9) the kernel is still relatively small and userspace contains only minimum of architecture specific code. USB was tested and developed on IA-32 and AMD64 architectures but the userspace library and kernel layer hides the architecture dependent ‘stuff’ and porting USB support to other architectures shall not create any unexpected problems.

The microkernel nature of HelenOS means that vital services the operating system provides are not part of kernel but are rather provided by userspace tasks. There are separate tasks for handling file system service, for networking or for drivers management. Most of these services are used transparently. For example, the fact that call to `printf` is actually translated to a call to virtual file system service from where it goes to console service that actually prints the string is hidden from the programmer and the semantics of `printf` does not differ at all from a monolithic OS. Knowledge of these services is not needed for understanding and developing USB drivers. There are exceptions, though.

First one is **`devmap`** — device mapper — service that creates representation of hardware and virtual devices in a file system. It takes care of the `/dev` directory that is somehow similar to the directory found in Unix systems.

[1] http://www.helenos.org/documentation
[2] http://trac.helenos.org/

The other one is **`devman`** — device manager — service that controls startup and communication between device drivers. Understanding this service is vital for writing device drivers for HelenOS and will be described in following section in more detail.

However, before describing it, several notes have to be made about inter process communication in HelenOS in general.

2.2 HelenOS IPC: phones and fibrils

Communication between tasks is an important part of any general purpose operating system. And it is a vital part of a microkernel design. This section will be an overview of inter-process communication in HelenOS. Please note, that the term inter-*process* communication — IPC — is used even in microkernel environment where processes are usually called tasks.

HelenOS is based on asynchronous IPC. That means that when an application sends a message, the kernel takes care of the delivery but does not block the sender and the sender retrieves the answer by another request to the kernel.

The advantage of asynchronous communication is that the task can do something useful when waiting for the answer instead of being blocked in the kernel during some IPC system call. Also, it is easier to create an illusion of synchronous processing when asynchronous API is available than vice versa.

The disadvantage is that programming in an asynchronous way is more complicated. For servers accepting connections from more clients, asynchronous processing usually ends with registering of callbacks when some message is delivered and the code gets complicated.

In HelenOS, this disadvantage was bypassed by introduction of *fibrils* and asynchronous framework.

A *fibril* is a userspace cooperatively scheduled thread. Kernel has no knowledge of a fibril and the fibrils are backed by a real (i.e. kernel) thread. The scheduling of fibrils is cooperative and happens usually when asynchronous IPC operation take place.

The asynchronous framework is a set of C functions (part of HelenOS `libc`) that makes an illusion of synchronous IPC in order to allow simpler implementations of services. The main idea is simple: IPC will be asynchronous and each client connected to the service will be handled in a separate fibril where the communication would look like a synchronous one. The synchronous illusion is created by the framework. The framework keeps track of the fibrils and does the callback registration for them. When a fibril executes a pseudo-synchronous operation, the framework uses the asynchronous version of the operation and then schedules another fibril. When the reply arrives, the framework schedules the original fibril and for the programmer it looks like a return from a function.

This way the programming of services looks more natural and the asynchronous advantages are not lost. It also makes the kernel simpler. The same thing can be achieved using real (kernel backed) threads but then the kernel would need to keep track to which thread the message belongs. Now, the message is addressed to the whole task and the framework ensures that it is delivered to the right fibril.

So far, IPC was described in general terms such as message or response. HelenOS uses its own terms for some actions and it is necessary to know them in order to understand API documentation.

Probably the most confusing name is *phone*. A phone represents an open connection to some other task. Unlike in real life, these phones are unidirectional and only one side can initiate a message sending.

The message sent over the phone is called a *call*. The receiving party of the call eventually *answers it* by sending back a reply. A typical call consists of 5 integer arguments where the first argument is a *method* number describing action to be taken by the receiving party. An answer call also takes such arguments but the first argument is treated as a status code whether the action was successful. There are several special calls that are intercepted and preprocessed by the kernel. One of them is a *hangup* call that terminates the connection. Another group of such calls are data transfer calls that are used to transfer larger amount of data. Using a special call it is possible to create a *callback connection* that reverses the direction of the phone. This call reverses the normal connection scheme ‘I want to connect to you’ to ‘Do you want to connect to me?’.

Last two specials calls are a *duplication* request and a *forwarding* one. The first one duplicates already opened connection which is useful for example to allow parallel IPC between two tasks. The second one forwards an incoming call to another task. This functionality is crucial for naming service to forward connections to respective services.

When a task starts, it has only one opened phone connected to a *naming service*. This server remembers connection to other services and by duplicating and forwarding the connection allows other tasks to connect to them.

This scheme means that kernel itself has to know only about the naming service. Each task will got the phone to it and services will make a callback connection to register themselves for other tasks while clients will ask the naming service to be forwarded to other services.

2.3 Device driver framework in HelenOS

Drivers in HelenOS are implemented as standalone tasks in userspace that require only minimal kernel intervention. This is in great contrast to monolithic kernels where all drivers are part of kernel and are living in the same address space. In HelenOS, each driver is a separate task that communicates with others through IPC. It seeks kernel assistance only for privileged operations, such as interrupt handling.

Although the idea of having *all* drivers in userspace exists since the beginning of HelenOS some drivers are actually part of the kernel. Some are there because they are needed by kernel. For example, a driver for AT keyboard is needed for kernel console. Some are vital for the boot phase and their move into userspace would be too complicated (e.g. ACPI reading). And finally, it is still possible to find traces from the first versions of HelenOS when total separation of drivers (i.e. their move into userspace) was not possible (e.g. because it would complicate development too much).

The framework itself consists of two parts. The first part is a the device manager service that is responsible for starting the drivers and for controlling their life-cycle. The other part is a `libdrv` library that is used by the drivers and that encapsulates communication with device manager.

Each driver is a standalone executable program that controls all instances of a certain device type. For example, when there are two USB keyboards attached to the computer, they will be both controlled by the same task (i.e. the same running instance of an executable file).

The device is attached to the driver by the device manager when some bus driver detects new device and **`devman`** decides that the driver is suitable for controlling the new device.

1. When the device is detected, the bus driver queries it to get as accurate description of the device as possible. The description should typically involve manufacturer identification and device kind (e.g. keyboard, VGA-compatible card or printer). From this description the bus driver creates *match ids*.
   A match id is a string with description and a positive integer — score — telling how accurate is the description. For example, match id with manufacturer would have higher score than match id with generic device kind because the former is more accurate.
   Format of a match id is not strictly set but most of the time, URI-like format is used. For example, PCI-connected devices uses format `pci/ven=106b&dev=003f` where `106b` is vendor id and `003f` is device id.

2. These match ids are sent to **`devman`** together with message that new device has been discovered.

3. **`devman`** then goes through the list of all of its drivers and decides which is the most suitable one. Each driver also has a list of match ids describing which devices it is capable of controlling. The match ids are then compared for equality and highest score determines which driver will control the device.

4. Once **devman** knows which driver will take care of the device, it informs the driver about it. If the driver was not yet launched, **devman** takes care of starting it.

5. At the driver side, the incoming message is translated into a callback defined by the driver author. This callback is usually called `add_device`. In this function, the driver shall initialize the device and check that the device is usable. After initialization, it informs **devman** that it accepted the device.

The description so far missed any information on how applications communicate with the drivers or vice versa. This is done through so called *interfaces*. For an application, the interface of the driver describes its capabilities and way how to query the device. From the driver point of view, interface is a set of callbacks the driver must implement.

Obviously, there must be IPC behind these interfaces. The application talks with the driver through IPC that is wrapped on both sides into higher-level functions. For applications, these functions are sometimes part of `libc` or specialized libraries. On the other side of the IPC, drivers use `libdrv` that wraps the call reception into callbacks to C functions. Nevertheless, if you are planning to add you own interface, you will need to implement the IPC layer by yourself because HelenOS misses any automated stub generation.

2.4 Where to go next

This chapter was a mere overview or a refreshment course if you wish. If you feel uncertain in some aspects it is recommended to learn more about HelenOS prior trying to implement USB drivers.

Below is a list of recommended sources

- HelenOS wiki[3] is full of materials
- HelenOS sources[4]
- master thesis of Lenka Trochtová about device driver framework[5] (the API evolved since the defense of the thesis but the principles remained the same)

[3]http://trac.helenos.org
[4]http://www.helenos.org/sources
[5]http://www.helenos.org/doc/theses/lt-thesis.pdf

Chapter 3

USB overview

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.

3.1 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 authors of device drivers. 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).

3.2 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. Example of a USB function is a printer or a keyboard. The other kind are *hubs* that create branches on the bus and to which 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). Actually, 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 other devices are connected. The host controller must always provide at least one port to allow device attachment. Typically, the host controller provides its own hub, called *root hub*.

Although physically, the topology is a tree rooted in the host controller, addressing of the devices uses a flat structure (see figure below [p. 8]) and under certain abstraction, hubs can be viewed as transparent splitters that add no logic to the bus.

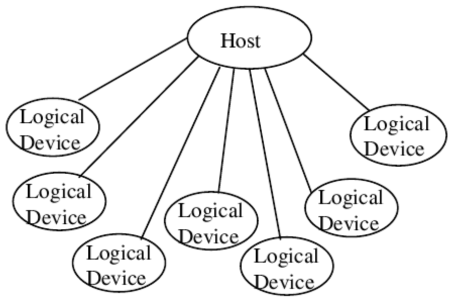

Figure 3.1: Logical USB topology

Each device connected to a host controller 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).

3.3 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 practically almost all common USB devices have only single configuration.

Each configuration can provide one or more *interfaces*. The interface provides the actual functionality of the device. For example, USB printer provides the printing as an interface. Some devices may provide more than one interface. For example, a digital camera may provide an interface for direct communication with the camera (usually via vendor 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* than change device capabilities at interface level. For example, network card may offer several alternate settings using different sizes of data packets.

The actual communication with the device happens 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 device driver on one side and the device on the other one. Endpoints are unidirectional (either in or out) and have several attributes that must be obeyed by the driver.

A special endpoint — *control endpoint zero* (sometimes *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).

3.4 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 ‘electrical’ aspect of the communication and starts with abstraction that the host controller has means to send a stream of bytes to the wire.

Communication over the bus uses data *packets* that encapsulates 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 few bytes several times minute) transfers over the bus happen in four different types.

Control transfers They are used for device configuration and for manipulating the state of the device.

Interrupt transfers These are intended for small and infrequent data where minimizing latency is the primary goal. Typical devices using such transfer type are all HID devices.

Bulk transfers Bulk transfers are used for sending large chunks of data over the bus when the requirements are to have undamaged data and even a bigger delay is acceptable. Typical devices include printers or scanners.

Isochronous transfers Unlike bulk transfers, where data integrity is a priority, isochronous transfers ignore data checking and priorities are put on quick 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 or accept some data from the device (the device cannot initiate the transfer under any conditions), 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. The host controller then encapsulates these information into two packets. The first packet informs about data direction while the other transfers the actual data. For incoming transfers, it actually reserves bandwidth to which the device can send the data.

This protocol is roughly the same for bulk, interrupt and isochronous transfers. Control transfers are a bit different because they have a special preambule — 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. Notice that 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).

3.5 Device descriptors and device classes

In previous sections querying USB device for its capabilities was mentioned. This sections 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 to which class the interface belongs 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[1].

The descriptors can be viewed as a tree, where 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. The USB descriptor tree [p. 10] figure shows the logical descriptor tree, circled numbers next to items of configuration subtree denote ordering in the serialized descriptor.

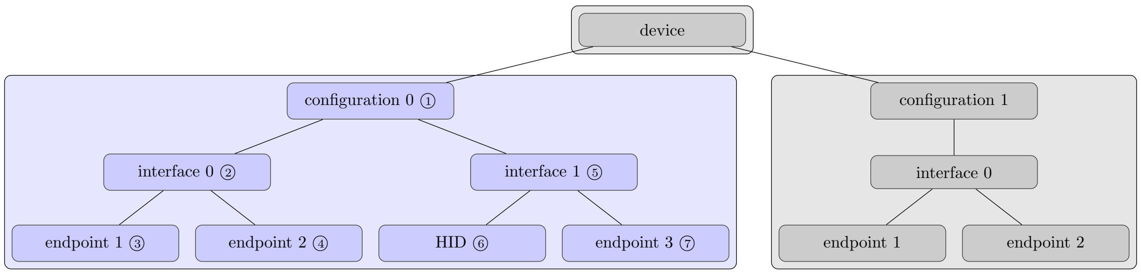

Figure 3.2: USB descriptor tree

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 subclass for keyboards and mice).

The class is reported as a part of device and interface descriptor. That is because 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'.

3.6 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.

[1]When you write a device driver, you ought to know what endpoints the device shall have. The endpoint descriptor is then used to merely map driver expectations to correct endpoint number.

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. signaling 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 specific ordering.

3.7 USB communication

3.7.1 Packets, transactions, transfers

3.7.1.1 Packets

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. Packet formats are described in USB Specification[2], part 8.4.

3.7.1.2 Transactions

However, packet level is not accessible to the software and drivers are not able to sent separate packets. The smallest communication entity available is called a *transaction*. There are five (more precisely seven) types of transactions available:

- setup transaction
- data IN transaction (DATA0 or DATA1)
- data OUT transaction (DATA0 or DATA1)
- isochronous data IN transaction
- isochronous data OUT transaction

[2] http://esd.cs.ucr.edu/webres/usb11.pdf

Every transaction consists of several packets. Usually it is a token packet, a data packet (might be DATA0 or DATA1), and a handshake packet. Isochronous transactions do not support handshake an thus consist of only two packets.

HC driver should alternate DATA0/1 transactions to provide some level of data sequence synchronization. This so called *Data toggle protocol* is described in USB specification, part 8.6.

Isochronous transactions do not support toggle synchronization and USB host should only use DATA0 packets, but it must accept both DATA0 and DATA1 in isochronous data IN transactions.

3.7.1.3 Transfers

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 four types of transfers:

- *Control transfers*. These consist of three stages:
  - SETUP stage: uses setup transactions, identifies command to the device, data packet is always DATA0 and 8 bytes long.
  - DATA stage(optional): uses alternating data IN (DATA1/0) or alternating data OUT (DATA1/0) transactions. Stage direction (IN/OUT) and its presence depends on the command sent in setup stage
  - STATUS stage: uses one data IN or data OUT transaction (always DATA1). 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.
- *Bulk transfers*. Bulk transfers use data IN or data OUT transactions to move data. Toggle synchronization is maintained by alternating DATA0/1.
- *Interrupt transfers*. Interrupt transfers are similar to bulk transfers but usually carry smaller amounts of data. They are considered periodic transfers and can specify time interval of repetitions.
- *Isochronous transfers*. Isochronous transfers use isochronous data IN and isochronous data OUT transactions. These are also considered periodic transfers but the time interval must be 1ms.

3.7.1.4 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 required bandwidth has to be reserved in advance. Failure to reserve bandwidth should result in driver/device 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, can be used by bulk transfers. Bulk transfer work in best effort mode and if there is no available bandwidth they are postponed.

It is the role of host controller driver scheduler to guarantee bus access constraints.

Part II

USB subsystem in HelenOS

Chapter 4

USB subsystem architecture

The goal of the USB subsystem is to support USB devices in HelenOS. This goal consists of the following issues:

1. Implement drivers for host controllers.
2. Implement drivers for USB devices.
3. Implement a mechanism for starting device driver when new device is plugged in.
4. Allow client applications to use the devices through the drivers.
5. Allow drivers to communicate with each other.

Fortunately, HelenOS already offers means for solving the last three issues. The device driver framework (see Device driver framework in HelenOS [p. 5]) that can be used as the backing framework for USB related drivers.

The generic framework — usually referred to as DDF — already offers the following:

- Starting drivers automatically (e.g. when a device is attached).
- Generic layer for driver to driver communication.
- Generic layer for exposing device interfaces to client applications.
- Software representation of physical connection of the devices.

Although DDF currently lacks some features, such as support for device unplugging, it was decided that the concept is right and implementation stable enough to allow its usage as a backend for USB device drivers.

One of the greatest advantages of DDF is the pattern that each driver is a standalone task that communicates with other drivers (or client applications) solely through IPC. Drivers then could be smaller which shall positively affect the number of bugs present. The USB drivers follow this pattern and for more complicated devices there are even several drivers that are cooperating.

This cooperation comes in two flavors. The first case is, for example, the UHCI driver (driver for host controller). To make the driver simpler in terms of code, it was decided to split the driver into separate driver for the host controller part and for root hub. These two drivers are tightly coupled and cannot run without each other (i.e. failure of one of the drivers will render the other one unusable as well). Yet this separation leads to cleaner code and easier maintenance.

The second case are multi-interface devices. For example, a digital camera may offer two means to access the photographs. The first one is through some vendor specific interface (for which you usually obtain the drivers on a CD accompanying the camera itself). The second one serves as a fallback and the camera is able to pretend that it is nothing more than a standard mass storage device. Coupling vendor interface with a standard one serving as a fallback is quite common pattern among USB devices. To maximize the possibility of using the fallback as easily as possible, a special driver for such devices was created. This driver is started for any multi-interface device and its only role is to scan the device for interfaces it offers and to launch drivers for each interface.

This multi-interface devices driver (referred to as USB MID driver) of course does not prevent from creating vendor specific drivers. These drivers could simply override the usage of the provided MID driver by offering higher score (suitability) for a specific device. Also, the overriding can be done on both levels. Either the driver can take over the whole device or over one of its interfaces only. The authors believe that such solution is versatile enough to satisfy needs of any USB device.

Figure below [p. 15] shows an example of USB physical topology.

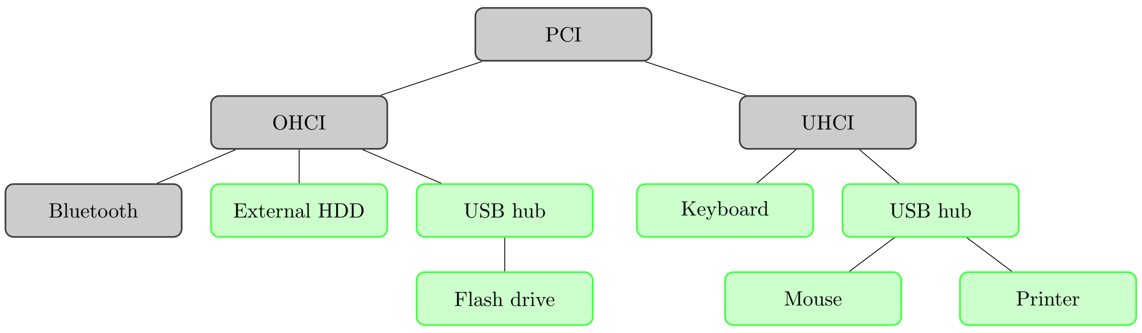

Figure 4.1: Physical USB topology

Although having both OHCI and UHCI chips on the same bus is something very rare there is principally nothing to prevent that. The tree nodes are in 1:1 mapping to the hardware devices. The grey nodes refer to internal devices that cannot be unplugged by normal means. That includes both host controllers and a Bluetooth controller that is often connected via USB bus. Remaining devices create a common subset of USB devices used today.

The next picture describes which drivers are actually started for each of these physical devices. To make the picture simpler, only the UHCI branch is displayed.

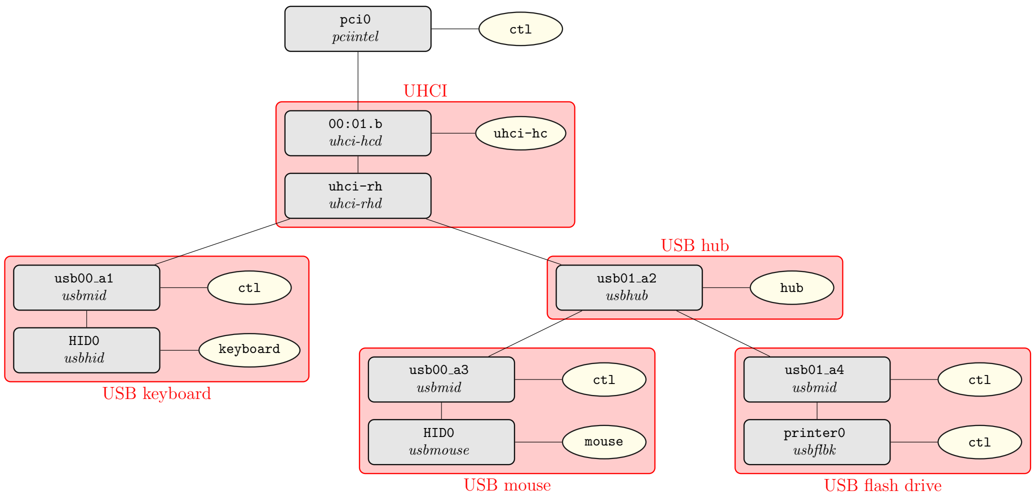

Figure 4.2: Mapping USB drivers to physical USB devices

The root node in the picture is the device node for PCI bus. As this node is not used directly, no detailed information about it will be provided here.

To the PCI, the UHCI controller is connected. As mentioned before, UHCI driver consists of two separate drivers controlling the same device. Notice that only the host controller driver provides an external function `uhci-hc` for clients. The only tasks that may want to communicate with root hub are drivers controlling the devices directly attached to it and these will use the parent function.

Another hub is connected to the one of the root hub ports. In this case, it is a normal external hub. Since the hub is a simple device, it is driven by single driver. The provided function `hub` has no special functionality (except to create entry in the devices tree).

All other devices are multi-interface ones. Because of that, the first node in the physical device is driven by the MID driver. The driver creates a `ctl` function that is currently used only to display the presence of the device in the device listing.

For each interface of these MID devices a new driver is started. For keyboard driver, the generic HID driver is started and this driver creates one external function `keyboard`. To this function, the console connects itself in order to receive key events. USB mouse driver is started for the mouse device and similarly to HID driver creates a `mouse` function used by the console.

Because printer devices are not supported, a special driver was started for the printer interface. This fallback driver is used for any device where no better driver exists. The purpose of the fallback driver `usbflbk` is to create corresponding entries in the device tree and thus allow querying of the device from within HelenOS.

If a digital camera with two interfaces — vendor and mass storage — was connected to the system, the MID driver would cause that each of the interfaces would be driven by the fallback driver as two separate devices.

Before explaining how the protocol used by device drivers for scheduling transfers to be issued by the host controller looks like, a special shortcut used by the drivers must be explained.

Although the physical wiring of USB devices through hubs to the host computer creates a tree with possibly many layers of nodes, from the point of the host controller the hierarchy is flat. The host controller targets the devices by their assigned addresses and virtually does not care where in the physical topology they are located.

Ideally, each driver could communicate only with the immediate parent and with its children (i.e. with drivers controlling devices that are physically attached to it / it is physically attached to). Transformed into

DDF, that would mean that the mouse from the figure above [p. 16] would send the request for transfer scheduling first to the MID driver, which would forward it to the hub, then onto the root hub and finally to the host controller driver itself.

The advantage of this parent-only communication is that it follows the physical topology closely. But there are several disadvantages. Firstly, it is slower. This impact may not be of vital importance but it is there. Next, it complicates the drivers along the way. They have to provide other functionality and for USB it is really only a forwarding.

Because of this, and supported by the flat view by the host controller itself, authors decided that device drivers would directly communicate with the host controller driver.

The following figure displays how the end device driver actually connects to the host controller for the first time.

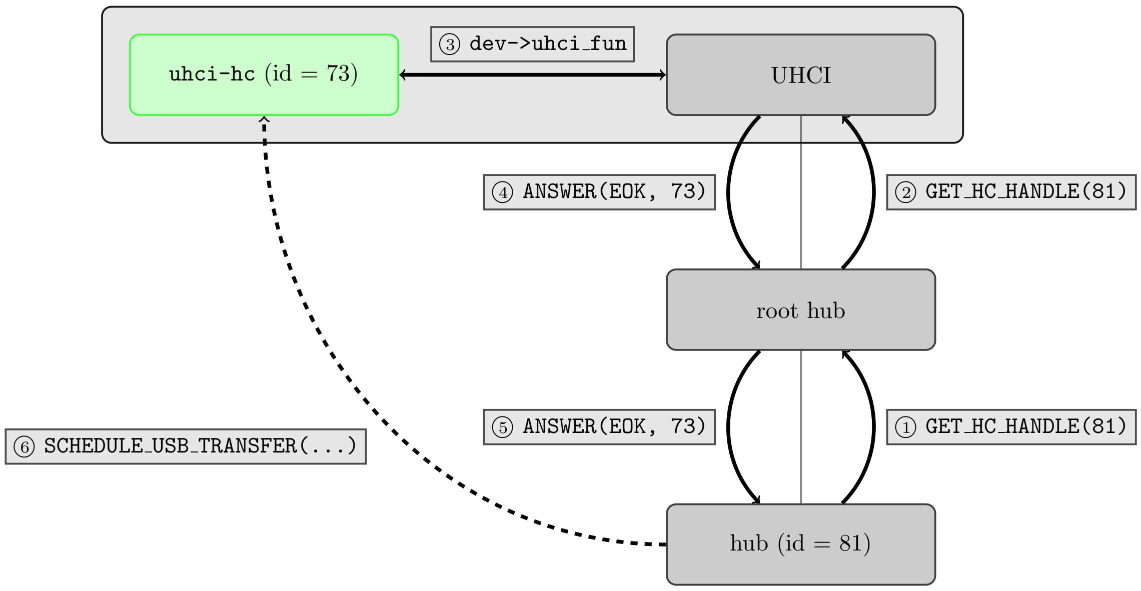

Figure 4.3: Connecting to host controller

Obviously, that has to be done by hopping over parents but this is done once when the device is added and the handle (identifier) of the host controller is then kept for future use. Notice that also the answer is hopping back because each request is a new IPC message.

For sending the actual transfer requests, a new connection directly to the host controller is opened.

Obviously, this recursive retrieval can be improved because the drivers along the way can cache the value to be returned because that is something not expected to change.

The actual protocol used for communication between the drivers is using DDF interfaces and is built above them. For example, host controller offers a USBHC interface that must implement callbacks for scheduling transfers and for USB address management (needed by hubs to allow them assign new address to newly connected devices). Similarly, USB devices offer USB interface where one of the most important function is function for retrieving the handle of the host controller. This is used in the recursive search for host controller.

The method details are described in following chapters together with notes for implementers. For device drivers, a thin layer above DDF was created to simplify writing the drivers. This layer takes care of initialization of the communication with host controller driver and with the device itself. The IPC is then hidden under wrapper functions almost completely. Depending of the device, the driver may provide additional interfaces (e.g. a keyboard interface) and implement its methods. The USB framework ensures that the

USB communication could be written without need to bother with low level problems such as making IPC connection etc.

As was stated in the USB overview, a device can support more configurations and each interface may offer alternate settings. The possibility to have more configurations is used very rarely. As a matter of fact, during development no device providing more than one configuration was encountered. Because of that, the authors decided that multiple configurations would not be supported at all. See chapter about future development [p. 77] for a few notes what changes would have to be made to support this feature.

Also the alternate interfaces are very rare and they are almost non-existent within the HID class. Thus the support for them is very limited and not tested.

Chapter 5

USB subsystem libraries & API

This chapter provides a brief overview of USB-related libraries available in HelenOS. Details for each of the libraries can be found in subsequent chapters.

5.1 Common USB library

Definitions common for any USB device or host controller are in `libusb`. This includes structures for device descriptors and basic constants.

This library is used by all other USB related libraries and thus you have to link with this library.

More details can be found in Common USB library [p. 46].

5.2 USB device drivers library

All USB device drivers are built above a common layer that provides abstraction over physical communication with a USB device. This layer is part of `libusbdev` and gives the programmer access to objects using the same level of abstraction as many USB specifications.

For example, the programmer uses object called `usb_pipe_t` to communicate with the device and do not have to care about actual USB transfers and transactions that are created on the bus.

Part of this library are also wrappers for standard device requests, simple parser of serialized descriptors or helpers for device polling.

This library is described in more detail in Writing USB device drivers [p. 57].

5.3 USB library for host controllers

According to USB specification, a host controller must create so called bus interface to allow unified communication from device drivers. The interface itself is described in the generic driver library but the implementation is up to the drivers. However, the functionality is similar for any host controller and thus a common library `libusbhost` exists.

This library provides helper structures and functions for keeping information about connected devices, about their endpoints or keeping track of free bandwidth.

For more information see Writing USB host controller drivers [p. 70].

5.4 USB library for human input devices

USB device for human input is self-descriptive in a way that each device describes which input controls it offers and how these controls are set. The parsing of such description is not a trivial task and because HIDs can be very specific, the whole parser and other helper functions were put into a separate library called `libusbhid`.

This library is expected to be used by any HID driver and more information about the parser and other functions can be found in Writing HID drivers and subdrivers [p. 66].

Chapter 6

Changes to HelenOS not related with USB

This chapter describes changes to HelenOS that are not directly related to USB subsystem. Some of the changes can remain in the HelenOS, some are rather temporary workarounds that shall be removed once HelenOS supports certain feature.

The repository used by USB team was regularly updated with latest changes to the mainline repository. The interval of such merges was less than fortnight and thus merging the USB back to the mainline shall not produce many conflicts.

6.1 Kernel space

Changes to kernel include temporal workaround for missing physical memory allocator and a short sleep system call.

Current version of HelenOS has no means to allow userspace tasks allocate physical memory with given attributes. A task can create new address space area (continuous block of pages) but has no means to specify where from the physical frames shall be allocated. That could lead to troubles with devices reading or writing data from a memory. Such devices may use only 32-bit addressing and it is not possible to use frames with addresses above 4GB.

Such allocator is currently being developed by another Software project team (NIC framework) and the authors decided that it is not necessary to duplicate this work. Thus only a simple system call for retrieving physical address was added. Obviously, this could fail on machines with more than 4GB memory but that would mean that someone was able to use all 4GB of memory which is hard to imagine with available end user software in HelenOS.

Memory caching was also disabled to prevent inconsistencies in communication between a device and a driver.

The granularity of fibril sleep in userspace was not sufficient when executing operations on I/O space registers. These operations must be separated by a very short delay that was not possible to ensure with existing libraries. The kernel does not export the real time precisely enough and thus a simple actively waiting system call was added.

The question whether to use a system call or some userspace routine is opened for further discussion among HelenOS developers.

6.2 Device manager

During development, several problems in **`devman`** were found. These were reported upstream and repaired in the mainline.

Several minor improvements were applied only in USB repository. These might be either cherry-picked or merged together with the whole USB subsystem.

6.3 The console server

The **`console`** server is responsible for switching virtual terminals and forwarding keyboard events to them. **`console`** was extended to allow automatic detection of hot pluggable devices, such as USB keyboards.

The detection of hot-plug HIDs is done in following manner. The respective device drivers shall detect the device, initialize it and create exposed DDF function for it. Keyboard device drivers shall add this function to `keyboard` device class while mouse devices to the `mouse` one.

The **`console`** periodically checks directory with these classes and tries to connect to any new device that appears there.

A special workaround was added to allow proper connection destruction when the driver stops sending new events (typical situation for device unplugging).

It is expected that **`console`** would undergo a major rewrite in near future that would simplify its communication with device drivers and would probably lead to creating a special service for processing of events from human input devices.

Such server might take care of some functionality currently handled in device drivers. For example, automatic key repeat is handled in the device driver. The design of the driver would be cleaner if it would only need to send ‘key press’ and ‘key release’ events and leave repetition on the server. The server may also take care of layout switching, another functionality that is now duplicated in drivers.

Writing the HID server would probably require rather drastic changes to several components of HelenOS and was beyond scope of the USB project.

6.4 PCI driver

The existing PCI driver was extended to allow interrupt enabling and to allow drivers of attached devices write to PCI configuration space.

Unfortunately, similar changes were done by the NICF team with almost identical API. However, it is expected that resolving the conflicts would be trivial.

Part III

USB drivers in HelenOS

Chapter 7

Host controller driver task

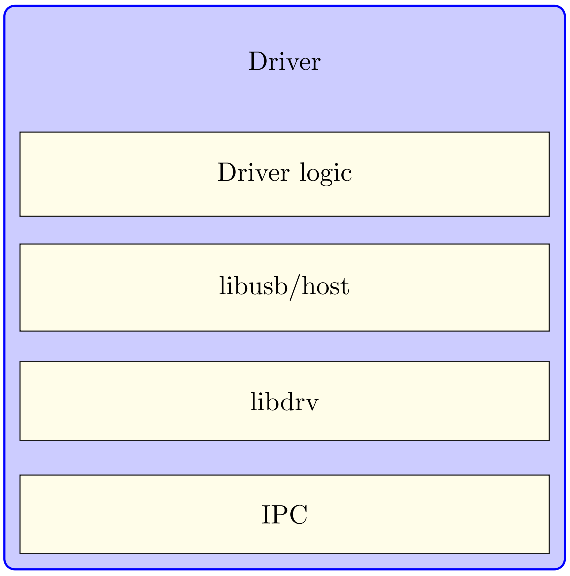

Figure 7.1: Host controller driver structure

7.1 Role

The role of the host controller driver task is to provide interface for USB device drivers. The interface is independent from different host controller implementations. Moreover, it contains functionality that is shared or not specified by host controller specifications.

The host controller driver task interface is specified in `<usbhc_iface.h>` provided by `libdrv`. The interface provides two sets of functions, the first set manages state of USB bus and it is a part of USB bus driver, the other set provides access to communication capabilities of the underlying host controller. While it is possible to use this interface directly in applications and device drivers, the suggested way is to use USB bus driver frontend implemented in the `libusb` family of libraries.

7.2 USB bus driver functionality provided by host controller driver task

7.2.1 USB address management

USB specification dictates that every connected device in order to get to the addressed state needs to have a 7-bit address. Address 0 is considered default and should not be permanently assigned to any device. It is the role of USB bus driver to keep track of assigned addresses and refuse to initialize a new device if there is not a free address available.

Structure that keeps track of free and assigned addresses is called `usb_device_keeper_t` and can be found in `<usb/host/device_keeper.h>` in `libusbhost`. Apart from maintaining the list of used and free addresses it provides a mapping from USB address to DDF device handle and vice versa.

The last information that is kept in this structure is device's speed. USB 1.1 allows device to communicate in *full speed* or *low speed*. Moreover, devices using different speed can coexists on one bus, thus this information needs to be stored per device.

New addresses are requested using `device_keeper_get_free_address()`, the returned address is marked as occupied and device's speed is stored. Once a new DDF instance has been created its handle is bound to the USB address using `device_keeper_bind()`.

To access information stored within this structure there is a set of functions. `usb_device_keeper-_find()` returns USB address used by the device. `usb_device_keeper_find_by_address()` provides access to address to DDF handle mapping. Finally, `usb_device_keeper_get_speed()` returns speed information associated with USB address.

7.2.2 Communication pipe backend

Endpoints represent devices' end of USB communication pipes. While device drivers communicate via pipes, these pipes are software constructs and store information about the pipe's hardware target. Host controllers accept communication requests identified by address, endpoint number, direction, speed, transfer type, and maximum size of data packet. Complete information is provided during pipe creation.

The structure is called `endpoint_t` and it is defined in `<usb/host/endpoint.h>`, provided by `l-ibusbhost`. In addition to providing data storage, this structure allows host controller hooks and ability to store host controller driver's private data. It also serializes communication to prevent interruptions and failures.

Pipe backends can be shared. It is possible for several processes to communicate with one device using one registered endpoint. Attempts to register multiple endpoints targeting the same destination are not supported. Endpoint sharing feature is used by USB utilities to access USB devices that are connected to the system.

There is one special role `endpoint_t` structures are used for, the toggle protocol synchronization. Toggle protocol is a simple sequence checking mechanism, it only alternates one bit. The problem is that this sequence can be reset by communication with different endpoint on the same device (usually endpoint 0). Thus, all control traffic needs to intercepted and checked for commands that reset toggle bit sequence.

Registered endpoints are stored in a structure called `usb_endpoint_manager_t`, defined in `<usb/-host/usb_endpoint_manager.h>` in `libusbhost`. In addition to endpoint manipulation it provides `usb_endpoint_manager_reset_if_need()` that checks USB command for signs of toggle sequence reset and `bandwidth_count_usb11()` that calculates required bandwidth in full speed bit-times (100% bandwidth = 12000000 full-speed bit times).

7.2.3 USB bandwidth allocation

Capacity of the USB bus is limited and USB devices have different data transport requirements. Thus, every transfer type is handled differently. Isochronous and interrupt transfers (sometimes called periodic) are used for regular transmissions of relatively small data, these transfers consist of one transaction complete within one frame (1ms). On the other hand, control and bulk transfers happen relatively infrequently, consist of several transactions and may need several frames to complete.

While every host controller provides a schedule to organize transfers in one frame, it is the role of USB bus driver to make sure that transfers in that schedule are organized according to USB specification.

The bandwidth reservation happens during endpoint registration. Periodic transfers' bandwidth requirements are calculated and if there is not enough bandwidth available, endpoint's registration fails. Up to 90% of USB bandwidth can be reserved using this method. The remaining 10% is used by both control and bulk transfers. The requirement to reserve 10% for control transfers is achieved using priorities within host controller schedulers.

7.2.4 Linking requests and endpoints

Communication requests are bound to endpoints and the structure that represents this bond is called `usb_transfer_batch_t`. It is defined in `<usb/host/batch.h>`. Apart from storing the designated endpoint it is the place where request buffers, callback pointer, and callback parameters are stored. These information are necessary for any host controller and using a common structure prevents code duplication. The structure provides a hook for host controller driver to store its specific memory representation of the request.

7.3 UHCI Host controller driver

UHCI design guide mentions that UHCI was designed with hardware simplicity in mind. It means that great deal of functionality is left to be implemented by the driver software. On the other hand, it also means that there is a considerable freedom for OS developers in the driver's implementation.

7.3.1 Legacy support, PCI control

UHCI hardware may integrate support for legacy use of mice and keyboards. Note that the implementation described in the design guide[1] is voluntary and it is not the only way to provide legacy support. However, host controllers that do not follow UHCI design guide will need separate drivers and may fail to work with generic HelenOS UHCI driver.

Legacy support, if implemented, is controlling PCI configuration space of the UHCI hardware. It has to be turned off *before* any further actions are taken. This routine, called `pci_disable_legacy()`, is implemented in `pci.c`.

Two more PCI specific routines, are implemented in `pci.c`:

- `pci_get_my_registers()` queries the PCI driver for hardware resources (I/O registers, IRQ number).
- `pci_enable_irq()` asks the PIC driver to enable IRQ assigned to the UHCI device.

[1] http://download.intel.com/technology/usb/UHCI11D.pdf

7.3.2 HelenOS DDF device setup

In the DDF framework's hierarchy, UHCI consists of two functions, a host controller device, and a root-hub device. The former is controlled directly by the UHCI host controller driver task as a so called *external function*. The latter is initialized as an *internal function* and its controller is handed over to the separate UHCI root hub driver task. A minimal support that provides information about root hub's resources is present within the host controller driver task, this support structures implement root hub's USB interface.

DDF device and host controller function's initialization is implemented by `device_setup_uhci()`, found in `uhci.c`. Root hub DDF function is initialized by `rh_init()` in `root_hub.c`.

7.3.3 Memory structures

Memory structures used by the UHCI hardware are described in chapter 3 of UHCI design guide. There are several types of hardware accessible memory structures:

- **Link Pointer** (LP) — Link pointer is not a separate entity. It is either in a group of 1024 pointers, called *frame list*, or it is a part of more complex memory structures described below.
- **Queue Head** (QH) — Queue head indicates a start of a separate group of transactions. UHCI does not support QH stacking. While it is legal to have QH within a queue (any depth), only the last QH is kept during schedule execution and the execution is stopped if that QH does not link to any further entity.
- **Transfer Descriptor** (TD) — Despite the name, TDs in fact describe transactions (as defined on page 9 of USB specification). TDs abstract sending of token packet, sending/receiving data packet (DATA0 or DATA1) and a handshake packet (if applicable). While one TD might represent entire transfer, this does not have to be the case with more complicated (i.e. control) transfers.
- **Data buffer** — Data buffer does not follow any special form and it provides raw data to be sent or that have been received from the device. However, the memory it occupies has to be continuous for host controller access (data to be sent during one transaction cannot cross page boundaries). It is the role of UHCI hardware to provide actual encoding before the data reach the wire.

Due to the limitations of the LP type all structures, except the data buffer, need to be 16byte aligned. UHCI hardware works with physical memory addresses and it is limited by the size of pointers (32bit). Thus the driver needs to be careful about placement of memory structures. A simple memory allocator wrapper is present in `<utils/malloc32.h>`.

All memory structures' definitions are stored in the `hw_struct` in their respective files.

7.3.4 Device startup and initialization

After the legacy support has been turned off, the driver assumes control over the UHCI device. Device initialization is divided into two phases. The first phase, called memory initialization prepares all the structures that are to be used by the device and forms a schedule skeleton, the other assigns these structures to the device and starts the hardware processing. If hardware interrupts can be used driver setups events that cause interrupts. Note that polling mode is experimental and only available until more PIC drivers for HelenOS are available, using it might result in performance degradation and/or other problems.

7.3.4.1 Schedule

UHCI design guide suggests using one queue for all transfer types (except for isochronous). HelenOS UHCI driver uses a separate queue for every transfer. This has the advantage of easier scheduling, and it also prevents one faulty transaction from halting the entire transfer queue.

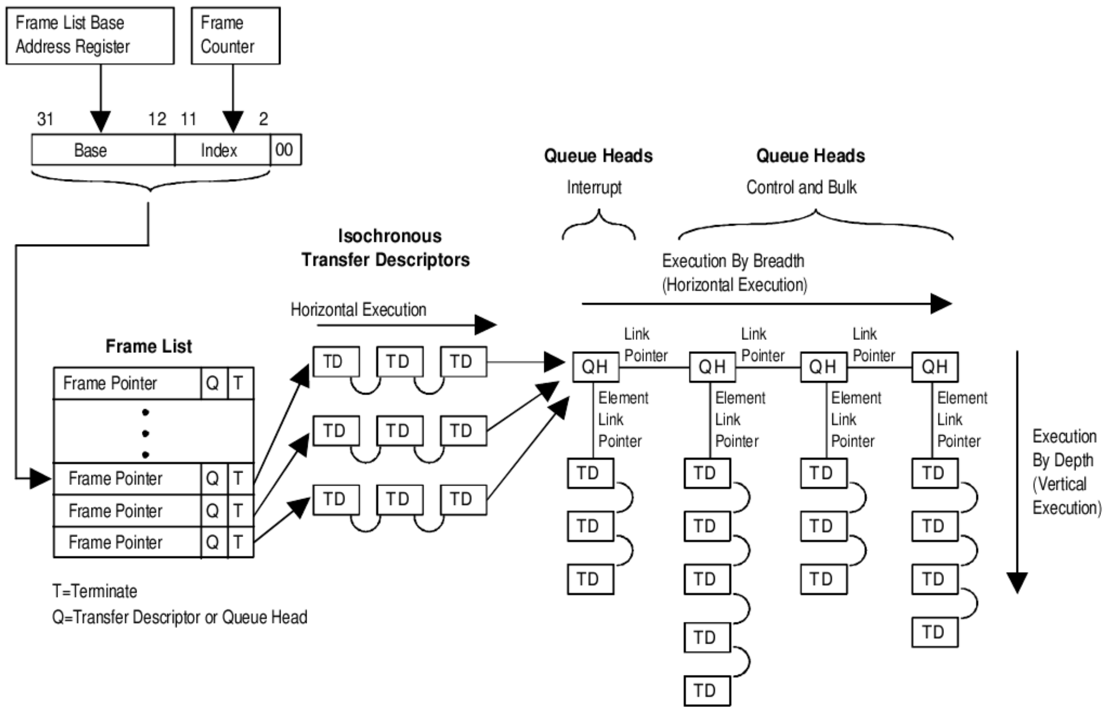
Frame List Base Address Register
Frame Counter
31
12
11
2
Base
Index
00
Frame List
Frame Pointer
Q
T
T=Terminate
Q=Transfer Descriptor or Queue Head
Isochronous Transfer Descriptors
Horizontal Execution
TD
Queue Heads
Interrupt
Queue Heads
Control and Bulk
Execution By Breadth (Horizontal Execution)
Link Pointer
QH
Element Link Pointer
Execution By Depth (Vertical Execution)

Figure 7.2: UHCI scheduling queue

HelenOS UHCI driver schedule uses one list of QHs just like the simple schedule suggested by the UHCI design guide. However, the QHs representing transfer types are never used, and a new transfer adds its own queue to the queue list. The exact position is between the last QH of the same transfer type, and the dummy QH of the next transfer type. This addition is done using one atomic memory write that updates the dependent pointer. Queue removal is done using the same pointer update technique.

Transfer queues are linked in a way to satisfy USB bus access constraints. The order of precedence: *periodic transfers* (interrupt) $\rightarrow$ *low speed control transfers* $\rightarrow$ *full speed control transfers* $\rightarrow$ *bulk transfers* (only full speed). The amount of periodic transfers is limited, thanks to USB bus driver, to 90% of bus bandwidth, thus the remaining 10% can all be used by control transfers and only after these transfer types are satisfied a bulk transfer can take place.

One might notice the separate queues for low and full speed control transfers. UHCI hardware supports a feature called *full speed bandwidth reclamation*. This feature relies on the fact that there are two modes in which queues can be executed — either all TDs in the queue are executed before moving to the next queue (vertical mode), or just the first TD from the queue is executed. Switching full speed control and bulk queues to the non-vertical mode and pointing the last bulk queue back to the first (dummy) control queue may achieve better distribution of bandwidth among connected devices, and offer a better chance for bulk transfers to access the bus. On the other hand, it may violate the USB bus access constraints and this feature was reported buggy on certain hardware. Thus it is not used or enabled in the HelenOS UHCI driver, yet the support is there for any future developer willing to explore the benefits of this technique.

The driver uses `transfer_list_t` to provide queue abstraction. Apart from providing access to dummy QHs, this structure keeps a software list of all enqueued transfers. Pointers used in the UHCI structures use physical memory address space and are not of much use. `transfer_list_set_next()` is used during initialization to create the proper structure (described above). `transfer_list_add_batch()` is used to enqueue a transfer, represented by the batch structure. `transfer_list_remove_finished()` checks for completed transfers and removes them from the list for further processing. These two functions implement addition and removal routines described above. In case of fatal hardware error all pending transfers need to be aborted and `transfer_list_abort_all()` does just that.

It is possible to construct a special scheduling structure using 1024 different frame schedule startup points in the *frame list*. This structure, a binary or any other tree, might be used for advanced scheduling of interrupt transfers. Interrupt transfers do not need to be scheduled in every frame and specify an interval value of a minimal rate. While host controller driver is free to assign any lower value and it does not break the specification to schedule these transfers on every frame, it might be beneficial to bandwidth sharing and utilization. This advanced feature is not implemented.

7.3.5 Operation

The driver operates on events, if hardware interrupts are not available a single fibril is used to check UHCI status and emulate interrupts.

There are two kinds of events the driver responds to, the first is user/software generated, the other is hardware generated. User generated events are limited by the USBHC interface and include transfer requests and USB bus driver requests. USB bus driver requests are handled by structures described in chapter USB bus driver functionality provided by host controller driver task [p. 25].

Three types of transfers are supported: *interrupt*, *control* and *bulk*. Support for *isochronous* transfers was postponed until any kind of multimedia framework is added to HelenOS. A `usb_transfer_batch_t` instance is created. As most of the functionality is provided by `libusbhc`, only UHCI specific routines are implemented. These routines create a memory structure made of one QH and several TDs:

- `batch_control_write()`
- `batch_control_read()`
- `batch_interrupt_in()`
- `batch_interrupt_out()`
- `batch_bulk_in()`
- `batch_bulk_out()`

After the memory structures are prepared, the batch is inserted into the right schedule.

Hardware generated events are interrupts. Interrupts might indicate a change in transaction's status or device errors. Status of `usb_transfer_batch_t` is checked using `batch_is_complete()`. This routine walks all TDs until the last TD is complete without an error or a transaction error is found. Status is recorded in the `usb_transfer_batch_t` and reported to the process that initiated the transfer.

Device errors indicate a severe condition, all pending transfers are aborted and hardware restart is attempted. Up to three hardware restarts may happen until the driver gives up and reports malfunction.

7.3.6 UHCI Root Hub

USB specification defines USB root hub as a standard USB hub device. However, it allows parts of hub functionality to be implemented in software. It means that some functionality needs to be emulated (SET_ADDRESS), or not supported (per port power settings).

7.3.6.1 HelenOS implementation

It was decided that rather than implementing a hub emulation layer, that would get UHCI root hub capabilities to match those of regular hubs, a separate driver would be developed that accesses root hub hardware directly. This decision has a disadvantage of having a separate driver for root hub devices and it is not directly supported (it's not forbidden either) by USB specification. On the other hand, it offers several advantages. There is no hardware emulation, no hub descriptors, no parsing of USB requests, no construction of USB responses, no address assignment, no filtering of communication to the root hub. This means that UHCI root hub driver is really simple, in fact sources in `uspace/drv/uhci-rhd` contain only few hundred lines of code, including licenses and comments.

7.3.6.2 Root hub hardware device

UHCI specification mentions that every UHCI device must have at least two root hub ports. It allows more ports to be present but does not specify a way to get the number of ports present. Thus only two ports are considered and controlled by the driver.

7.3.6.3 Startup

The first part of root hub driver is implemented within the UHCI host controller driver, namely retrieving the I/O address of registers controlling the ports (`uhci_rh_init()`). These can be found at offset 0xC and occupy 2 bytes of IO space per port (`uhci_init()`). It is logical to implement this part within the host controller driver (`uhci.c` and `uhci_rh.c`) as both host controller and its root hub are the same PCI device. Once the address space is identified, control is handed over to the root hub driver itself.

7.3.6.4 Operation

The root hub driver starts a separate *fibril* for every port. This fibril periodically checks for changes on the port by reading the port status register in `uhci_port_check()`. These changes include connection of a new device and device removal.

When a new device is detected a device initialization routine `uhci_port_new_device()` is started. The driver uses function `usb_hc_new_device_wrapper()` provided by `libusb`. This function is provided a callback to port reset and enable routine (`uhci_port_reset_enable()`). This routine is device specific and could not be abstracted to the library. Root hub driver stores a device handle to the newly added device, it will be used during device removal, when such mechanism in HelenOS becomes available.

If a connection change is detected on a port that had a device assigned, it is assumed that device was disconnected and a disconnect routine `uhci_port_remove_device()` is started. Device removal is not yet supported by HelenOS.

7.3.7 Features not implemented

It was decided that implementing certain features is not possible/too low priority in the given time frame and the state of the OS.

7.3.7.1 Host controller

Support for full speed bandwidth reclamation has been dropped due to hardware issues and a limited benefit this feature offers.

Support for isochronous transfers is not implemented as there was no way to test this feature and its implementation will be dependent on HelenOS media framework that does not exist at the moment.

There might be hardware quirks required for certain UHCI hardware implementations, these were not considered and will require extensive testing.

7.3.7.2 Root Hub

There are no signs of power management whatsoever. No work has been done in HelenOS in this direction, and the port suspend/resume functionality is not used.

Although UHCI design guide does not specify a way to get the number of present root hub ports, some OSes try to figure this out by reading the IO space after the second port control register.

Ability to detect over-current condition is not part of the original UHCI design guide and was added in Intel chip set specification. However, implementation of this feature vary in quality, some devices does not report over-current condition at all, others report constant false positive. Enabling this feature would need per device quirks.

7.4 OHCI Host controller driver

OHCI host controller is a rather complicated device and `the design goal has been to balance between the complexity of the hardware and software'[2]. It supports more functionality in hardware, but also provides less freedom for system developers.

7.4.1 Legacy control, pre-OS driver

OHCI provides a way to access USB devices even before an OS is loaded. This enables computers to support features such as USB mass storage boot, and availability of basic HID device to systems that do not support USB. Thankfully, OHCI provides a rather detailed information about hands-off routines in various pre-OS configurations in chapter 5.1.1.3 Take control of host controller. HelenOS OHCI driver's implementation is in `hc_gain_control()` in `hc.c`.

7.4.2 HelenOS DDF device setup

OHCI DDF setup copies that of UHCI driver. It uses one internal and one external function (for root hub and host controller). The only difference is in the driver used to control OHCI root hub. OHCI root hub shares some resources (like interrupt) with the host controller, that prevent the driver separation. A decision was made to follow USB specification and provide an emulation layer for standard USB hub device interface.

Implementation of DDF specific code can be found in `ohci.c`. Root hub registration to devman is implemented in `hc_register_hub()` in `hc.c`.

7.4.3 Memory structures

OHCI host controller uses several rather complicated memory structures described in chapter 4 of OHCI specification:

[2]OHCI specification, ftp://ftp.compaq.com/pub/supportinformation/papers/hcir1_0a.pdf, p. 1

- **Endpoint Descriptor** (ED) — ED is USB device's endpoint representation. This memory structure is kept between transfers and maintains endpoint status toggle protocol integrity. Hooks and private data place in `usb_enpoint_manager_t` are utilized for quick access to this structure for every endpoint. EDs require that they have at least one TD assigned and will not execute if there is only one TD in their list. This requirement makes it easier to enqueue more TDs as the dummy TD becomes the first TD of a new transfer. `ed_t` is defined in `<hw_struct/endpoint_descriptor.h>`.
- **Transfer Descriptor**
  - *General Transfer Descriptor* (TD) — TD may represent entire transfer if it's not a control transfer. It is capable of transferring of up to 8kB of data and maintains its state across frames. TDs are assigned to EDs and form a linked list. It may use its own toggle protocol values or use the one stored in parent ED. `td_t` is defined in `<hw_struct/transfer_descriptor.h>`
  - *Isochronous Transfer Descriptor* (ITD) — ITDs are not currently used as the support for streaming media is not present. These structures may represent up to 8 isochronous data transactions. `id_t` is defined in `<hw_struct/iso_transfer_descriptor.h>`
- **Host Controller Communication Area** (HCCA) — HCCA is a multi-purpose structure. It contains starting EDs for periodic schedule (similar to UHCI *frame list*). Host controller regularly update a frame number in one of its fields and a list of completed TDs is maintained here. `hcca_t` is defined in `<hw_struct/hcca.h>`.
- **Data Buffer** — Memory structures described above support up to 8kB buffers. It means that processing that structure can automatically advance offset pointer and even cross page boundaries (once).

OHCI hardware works in physical memory address space and its pointer type is limited (32bit). Thus the driver needs to be careful about placement of memory structures. A simple memory allocator wrapper is present in `<utils/malloc32.h>`.

7.4.4 Device startup and initialization

OHCI initialization is done in several phases. The first stage prepares memory structures and constructs a skeleton schedule. The second stage initializes root hub emulation. Only after these preparations complete successfully can the control be gained from any BIOS/pre-OS driver. Once the driver is in control, it can reset and setup the device to use the prepared structures. Note that immediately after OHCI reset an interrupt will be triggered if there are USB devices connected, thus it is crucial that it can be processed in a sound environment.

7.4.4.1 Schedule

OHCI uses several schedules for accessing the bus. The first schedule is for periodic transfers and uses 32 startup points in the HCCA. While OHCI suggests using an advanced structure for interrupt transfer scheduling, it was decided that for the sake of simplicity only one interval value would be used. This does not break the USB specification and makes the driver much less complicated. USB bus driver guarantees that all scheduled transfers in this schedule will complete in one frame.

The second schedule is used for control transfers. It has a separate startup point and the driver must indicate that there is a traffic present to the host controller. This schedule operates in a round-robin fashion as the last ED is remembered across frames. The third schedule operates in the same way as the second and it is used for bulk transfers.

The first 10% of every frame is reserved for non-periodic (control and bulk) transfers. Host controller alternates between control and bulk schedules according to the pre set ration (1:1, 2:1, 4:1). While this does not obey the USB specification to the letter, it enables some bulk throughput even on very busy schedules. After the required 10% has been served, the priority is given to the periodic schedule. All transfers in periodic schedule must complete and if there is any time remaining it is allocated to the non-periodic schedules.

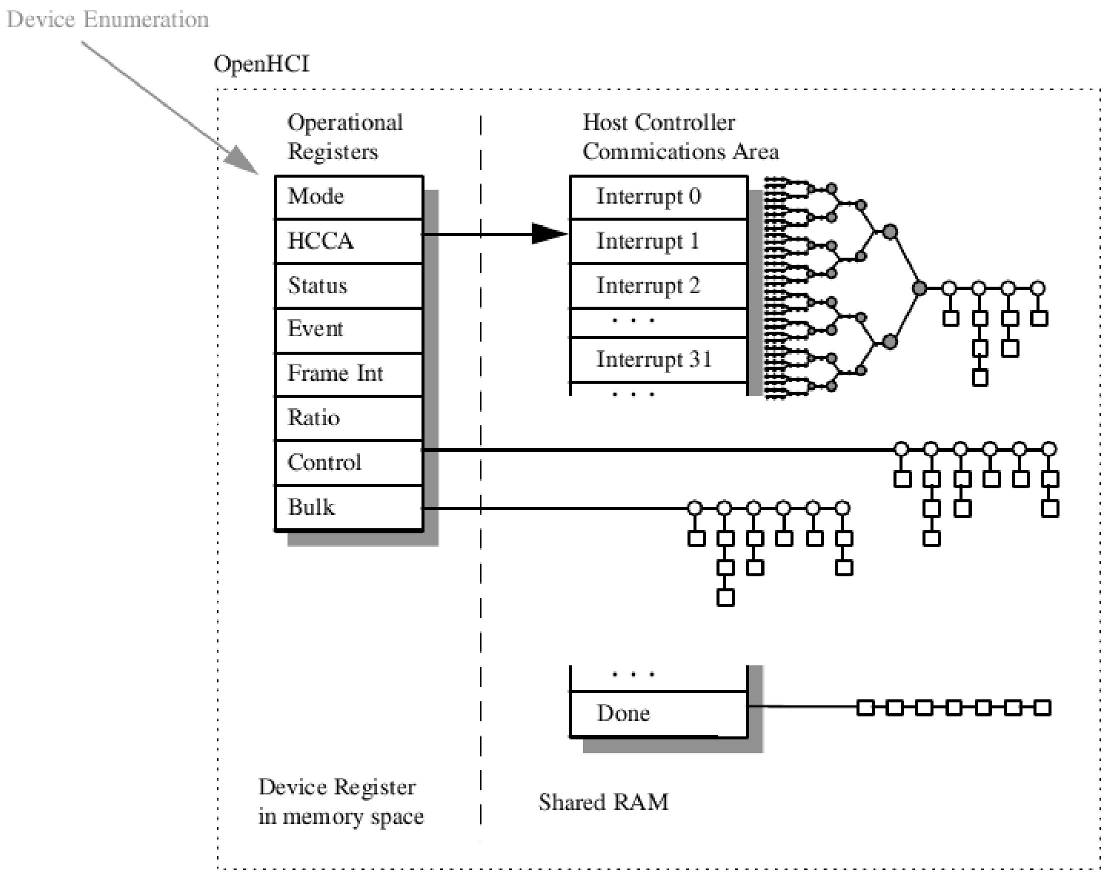

Figure 7.3: OHCI scheduling queues

for schedule abstraction. It uses dummy first ED to avoid translating physical addresses and register access. Every endpoint is enqueued during its registration and dequeued when it's no longer necessary. As all the work is done by hardware, it is the only thing left to the driver. `endpoint_list_add_ep()` and `endpoint_list_remove_ep()` are implemented in `endpoint_list.c`

7.4.5 Operation

Unless interrupts are disabled, OHCI hcd operation is event-driven. USB bus driver requests are handled using the provided structures. The sole exception is endpoint registration/unregistration. The fact that OHCI knows about and provides structure for endpoints (ED) makes it necessary to use hooks and private data storage provided by the `usb_endpoint_manager_t` to store endpoint's ED. Moreover, one dummy TD needs to be assigned to every new ED. As there is no place in `ed_t` to store virtual/linear address of this dummy TD, a separate structure is used to link EDs to their respective TDs, called `hcd_endpoint_t`.

While endpoint registration is a bit more complicated, transfer requests are rather simple. There is a `usb_-transfer_batch_t` instance created to store the transfer's data and link to the endpoint. OHCI specific routines are implemented to setup memory structures:

- `batch_control_write()`

- `batch_control_read()`
- `batch_interrupt_in()`
- `batch_interrupt_out()`
- `batch_bulk_in()`
- `batch_bulk_out()`

The easy part comes in when the batch is about to be scheduled. There is no need to search for appropriate queue or list, it was done during endpoint registration. Thus, only a pointer update in ED is necessary to enqueue the transfer. In case of control and bulk transfer the host controller needs not to be notified to consider appropriate schedules during execution. Requests for the root hub are redirected to the emulation software.

OHCI relies on interrupts to provide the driver with event information. Interrupts may indicate a transfer completion or error, root hub event, or a host controller error. Root hub status events are handled by the OHCI root hub emulation software and host controller is restarted on serious errors.

OHCI host controller maintains two lists of retired TDs. The first is accessible via a MMIO register and is actively used by the host controller to store retired TDs. The other is stored in HCCA and is updated at the end of frame if this update is allowed. This update can be accompanied by an interrupt. As both of these lists use physical addresses, they are completely useless. However, the interrupt indicating update also indicates a transfer completion or partial completion. This interrupt is used to check pending `usb_t-ransfer_batch_t` structures for completion and the outcome is reported to the process that requested the transfer.

7.5 OHCI root hub driver

OHCI root hub and its workflow is much similar to the normal USB hub (see USB hub driver [p. 36]). Therefore only a simple layer between the host controller driver and the hub driver is needed. This layer processes hub drivers' requests and 'emulates' function of the normal USB hub. It is embedded into the OHCI driver. This implementation is recommended by the OHCI specification.

7.5.1 Registers

For communication with the root hub device the driver uses four types of memory mapped registers initialized by the host controller driver (HCD). Two of these are registers with root hub information. Third is a root hub status register and the fourth is the port status register. Further information on these registers can be found in OHCI specification.

7.5.2 Initialization

Initialization takes place in function `rh_init()`. The root hub driver initializes the structure representing the hub. The interrupt buffer is preallocated, so that it doesn't need to be allocated for each single interrupt request. Also, the hub descriptor and the full configuration descriptors are created as they are not to be changed for a particular root hub during its lifetime.

The HCD then starts the standard hub driver and 'remembers' root hub USB address so that all requests to this address are redirected to the root hub driver.

7.5.3 Operation

The driver receives two types of events: requests from the USB hub driver and interrupts from HCD. Hub driver requests are handled in `rh_request()`, HCD interrupts in `rh_interrupt()`.

7.5.3.1 Hub driver requests

These requests are either control requests or interrupt requests.

Control transfers

Control requests are handled in `process_ctrl_request()`. These might be commands to get hub status, descriptor of certain type or configuration, set or clear a feature, set address, configuration or descriptor. More information on these commands is available in USB hub specification (USB specification[3], chapter 11).

For device descriptor request, interface descriptor request and endpoint descriptor request answers, static instances of these descriptors are used — these are the same for all OHCI root hubs and do not need to be allocated for each instance.

Data for status requests are in status registers. These registers have mostly the same format as the normal hub driver expects, thus they are only copied into the response.

`Set address request' returns `ENOTSUP` error — the root hub should have fixed address according to specification. Set descriptor request is not supported and returns `ENOTSUP` error. Set configuration request is ignored. These changes are ignored or not supported, because have no meaning for the root hub.

Interrupt transfers

When the root hub receives an interrupt request, it checks whether there are any changes, using hub and port status registers. If there is a change, it fills the status change bitmask (preallocated in the hub structure) and finalizes the transfer (`process_interrupt_mask_in_instance()`). If there is no change, the driver stores the request pointer and waits for next event.

Upon receiving an interrupt (in interrupt callback function `rh_interrupt()` the driver will fill the status change bitmask and finalize the stored transfer. Only one interrupt transfer can be stored per root hub instance — more interrupt requests should not occur (the hub driver is expected to be waiting for the result of the previous interrupt transfer).

7.5.3.2 Interrupts

When an interrupt is received, the driver checks whether there is a waiting interrupt transfer and if there is one, it finalizes it. Otherwise the interrupt is ignored — the root hub driver cannot initiate transfers to the hub driver.

[3] http://esd.cs.ucr.edu/webres/usb11.pdf

Chapter 8

USB hub driver

Hubs are devices used to extend the number of available ports for host controller so that more devices can be plugged to it. This section describes non-root hubs. A typical non-root hub has one upstream and several downstream ports.

8.1 Specification

8.1.1 Responsibilities of the hub driver

The hub driver is responsible mainly for detecting a new device plugged into the hub and detecting when a device has been unplugged [1]. In the former case, the driver must enable the corresponding port, acquire and set address of the new device and start/notice the appropriate driver, if available. In the latter case, the driver should free the unplugged devices' address and should notice devices' driver that the device has been unplugged. However, this cannot implemented, because DDF does not support disconnecting devices and terminating their drivers [2].

Another task of the driver is to handle the over-current conditions. The driver powers down ports that indicate over-current condition. This is related to overcurrent protection of USB devices.

The driver should also manage the power available to the devices in the USB tree. This is not implemented, because all the hubs seem to indicate that they are self-powered and thus do not need any bus-power management.

8.1.2 HelenOS specific tasks

The USB hub driver, as any other USB driver, communicates with device endpoints trough the host controller driver. That means it must connect to this driver and initialize communication with the proper host controller driver instance (there might be more than one USB host controller in the system). For their initialization and administration the *pipe API* is used (see Writing USB device drivers — The API [p. 58]). This automatically initializes the hub endpoint communication channels before entering hub driver code.

Besides the connections to the devices' endpoints (pipes) the hub driver also needs direct connection to the host controller driver. The hub driver needs this connection to reserve the default address on HC and to requst a free address from HCD for newly plugged devices. This connection must be initialized by hand and open whenever hub driver needs to reserve the default address or to request a new address. After that it must be closed again.

[1] By plugging a device we understand physically attaching devices USB connector to a hub port. By unplugging a device we understand detaching USB connector from hub port

[2] see Support for device unplugging [p. **??**] in Future development [p. 77]

8.2 Implementation

Hub driver implementation is in `uspace/drv/usbhub`. The driver first initializes structures for USB device drivers When a new hub is registered, the driver initializes its structures and starts its fibril. This polls the hub on the interrupt pipe and handles any detected changes.

8.2.1 Startup and initialization

Initialization of a new hub instance first enters code of the *pipe API* framework, which initializes communication links to the hub devices' endpoints. Hub has only two endpoints — default control endpoint and interrupt endpoint. The interrupt ednpoint is defined by `hub_status_change_endpoint_description` in `main.c`. Then the driver enters `usb_hub_add_device()` and loads basic information about the hub, such as port count and list of ports (`usb_hub_process_hub_specific_info`). It also initializes connection to the host controller. All this information is stored in a hub-representing structure `usb_hub_info_t`.

After obtaining the hub information and creating the hub instance, the hub ports are powered up. After that a fibril with hub-control loop is started by calling `usb_device_auto_poll()` with `hub_port_changes_callback()` and a pointer to hub-representing structure as its parameters. Thus each properly initialized hub has its own fibril. This way an error on one hub should not cause problems on other hubs. It also prevents some deadlock scenarios, such as two hubs reserving default address on the same HC.

8.2.2 Operation

Hub driver works in a control loop that checks changes on 'its' hub and then waits a specified time period. In the release version, the time period is set to 100 ms. It can be, however, set to much longer time for debug purposes.

If an error occurs while waiting for an interrupt from the hub, `usb_hub_polling_terminated_callback()` is called, gracefully terminating the hub instance.

If there is a change in hub status, the hub control loop invokes `hub_port_changes_callback()`. The hub driver should receive at least one change in each loop. The status change can occur either on the hub or on one or more of its ports. All indicated changes are then handled one by one.

8.2.2.1 Hub status changes

After detecting a global hub status change hub driver asks hub for hub status bitmap containing status and status change flags. Status changes occurring on a hub are *local power status change* and *over-current condition change*. They are handled in `usb_hub_process_global_interrupt()`.

Local power change is ignored. It seems that all hubs indicate that they are self-powered making any attempt on hub power management almost futile, including response to hub local power change.

On the *over-current condition* the driver shuts down power on all hub ports. After this condition is over, power on all ports is restored.

8.2.2.2 Port status changes

Handling port status changes is the most important responsibility of the hub. They are handled in `usb_hub_process_port_interrupt()`. This function first asks for port status bitmap containing port status and port status change flags.

Hub indicates port change whenever a new device is plugged or unplugged. It also indicates the *over-current condition*, *port reset* and also enable/suspend change.

Port enabled and *port suspended* changes are ignored — their flags are only cleared so that they are not indicated again. These changes can be generated only by hub driver, which currently does not do this. They are related to power management functions which are not implemented (see Future development [p. 77]).

8.2.2.3 Adding a new device

When a device is plugged into a USB hub it causes the hub to indicate *port connection change*. In such a case the hub driver starts a new fibril that will reset the port, wait until it is enabled, assign it a new address received from HCD and finally start/notice its driver. Reseting the port and requesting new address is done in `libusb` function `usb_hc_new_device_wrapper()`.

If any of these operations fails, the default address is released and the hub driver does not continue enabling the device. However if the device triggers connection change again, the whole process is restarted.

8.2.2.4 Removing a device

As DDF does not support device removal, the hub driver only clears the change status bit and continues in such case. However, should the device removal feature be added into DDF, there is commented code snippet for devices' address release in the routine for the connection status change `usb_hub_removed_device()`.

The host controller driver should ignore requests from the USB drivers that are directed to unplugged devices and return an error to it. This way the driver of the unplugged device should notice that the device was unplugged and stop.

8.2.2.5 Enable/suspend change

These changes are ignored by the driver. Hub driver does not enable/suspend ports, therefore these changes should not occur. However, if such change a occurs, the change indicating flag is cleared.

8.2.3 Unplugging a hub

When a hub is unplugged, the driver control loop detects a communication error and calls `usb_hub_polling_terminated_callback()`. This function releases the hub structures, effectively stopping the hub instance. This is generally the recommended way to unplug any USB device.

Chapter 9

HID class driver

9.1 Introduction

The HID class comprises devices that allow users to control computer systems. Typical examples are keyboards, mouses, joysticks, various buttons, switches, data gloves, etc. Some of these devices also provide some sort of feedback to the user (displays, audio feedback, etc.).

The USB HID device class definition[1] specifies how such devices should be implemented and how a HID class driver should communicate with them. As various HID devices may have very different requirements, this document defines a generic and extensible protocol called *Report protocol* (see Report parser [p. 41]).

9.1.1 HID device characteristics

Some of HID devices, namely keyboards and mouses, may require BIOS support (so that they may be used even before the OS and its USB drivers are loaded). These are called *Boot devices*. As the Report protocol may be significantly complex, there is a pre-defined *Boot protocol* for keyboards and mouses. Support for this protocol is declared by defining an interface to have *Boot Interface Subclass* (see Device descriptors and device classes [p. 9]).

Each HID device has at least two pipes (i.e. endpoints) — a *Control* pipe (default and present in all USB devices) and an *Interrupt In* pipe. Optionally, an additional *Interrupt Out* pipe may be present (for detailed description of these pipes, see the USB HID class definition, part 4.4).

HID devices are characterized by HID class-specific descriptors — *HID descriptor*, *Report descriptor* and *Physical descriptor(s)*. The HID descriptor mainly characterizes the following Report and Physical descriptors. There is at least one Report descriptor per interface and any number of Physical descriptors. Report descriptor characterizes the Reports used by the device to communicate with the host. It must be parsed in order for the driver to be able to parse (and interpret) incoming Reports. (For details on Report descriptor, see the USB HID class definition, part 6.2.2.) There are three types of reports — *Input* (used for data from the device), *Output* (data to the device) and *Feature* (describes configuration information that can be sent to the device).

Physical descriptor provides information about the part of human body used to activate some control (for details, see USB HID class definition, part 6.2.3). This is in fact a quite advanced feature and is not supported in the HelenOS USB HID driver. (Even if it was supported, the system could make no use of it anyway.)

[1] http://www.usb.org/developers/devclass_docs/HID1_11.pdf

9.2 HID driver

All USB HID devices share the same way of communicating with host (using the Interrupt In pipe) and the same format of data (Reports). Thus only the data interpretation is what makes the devices distinct from application point of view.

This allows separation of handling the HID device on the lower level from the data interpretation and processing on the higher level. There are two ways to achieve this:

1. Separate the main HID driver (communication with the device) and the device-specific drivers (data interpretation and processing) into independent processes. These specific drivers would be launched on-demand, based on the characteristics of the device.

2. Retain whole HID driver in one process, but separate the interpretation logic into modules with clearly defined API.

Both approaches have their pros and cons which will be discussed in the following paragraphs.

The first approach (separate processes) is more clean from the design point of view — different tasks are done by different processes — all in the spirit of a true microkernel OS. Adding a new driver for some special device would be also easier, as the existing code would not have to be changed. On the other hand, there would be some non-trivial obstructions to overcome. Firstly, the report parsing could not be done inside the main HID driver as there would be no way of sending the parsed report over to the child driver processes. It would have to be serialized, but then it would be easier (and with less IPC involved) to just send unparsed reports over to the child drivers and let them do the parsing. However, in such case the idea of separating common HID logic from device-specific would not be fully accomplished. Another obstacle is the distinction between various types of devices and the need to associate drivers with devices. With DDF (see Device driver framework in HelenOS [p. 5]) in place, it would be necessary to create some sort of match IDs for each device, based on the contents of the Report descriptor. As this may be fairly complex, it might be quite complicated to create universally functional match IDs.

The second approach (one process handling all kinds of HID devices) would make it harder, or at least less straightforward, to add drivers for new devices as it would change the code of the whole driver which would then need to be recompiled and restarted. However, with well-defined API for the ‘subdriver’ modules it would not be very hard to create such code and it would be also quite easy to communicate directly with the device in case of need. The Report parsing may be done in the main part of the driver, so that the subdrivers receive parsed data and can do only the required interpretation. Another advantage is the special handling of Boot devices — e.g. falling back to the Boot protocol in case there is no subdriver that understands the report.

Therefore, the autors decided for the second approach, creating one complex HID driver which handles all kinds of devices but has a clean subdriver API.

9.2.1 HID driver core

The core of the HID driver is responsible for several tasks:

- Creating and initializing an internal structure that holds information about the device (`usb_hid_dev_t`).
- Parsing the Report descriptor and initializing the structure for parsing Reports.
- Periodic polling for data, parsing and passing the parsed data to the subdrivers.

In the initialization phase (`usb_hid_init()`), the driver first checks if the device declares support of either keyboard or mouse boot protocol. Then it initializes the report parser by retrieving and parsing the Report descriptor (`usb_hid_process_report_descriptor()`). Afterwards it searches for any suitable subdrivers, comparing their requirements with the device characteristics (`usb_hid_find_sub-drivers()`). If none are found or if the Report descriptor could not be parsed, the driver falls back to the boot protocol, if it is supported by the device. Otherwise it initializes all subdrivers matched to the device.

During normal operation, the driver polls the device for new data and any time the data arrive, it parses the incoming Report and passes it to the subdrivers' callbacks. For polling, it uses the facilities provided by `libusb` (see Automatic polling [p. 58]). Functions used as callbacks for automatic polling are `usb_hi-d_polling_callback()` and `usb_hid_polling_ended_callback()`.

The code of the driver core can be found in `/uspace/drv/usbhid/`.

9.2.2 Subdriver API

Each subdriver must specify properties of the device it wants to control (so that it may be matched to the device) and several callbacks for various situation encountered when controlling the device. These situations are:

- There is a new device of the desired type and a new `usb_hid_dev_t` structure is being initialized.
- There are new data from the device.
- Polling of the device ended (probably due to some error).
- The `usb_hid_dev_t` structure representing the device is being destroyed.

By defining callbacks for these situation, the driver may initialize its own data structure when a new device has been added, process the incoming data and do any required cleanup. The core of the HID driver receives events from the autopolling loop (new data, polling was terminated) and executes the callbacks of all initialized subdrivers sequentially. The callback for initialization is called in the `usb_hid_init()` function after the HID device structure has been initialized. The callback for deinitialization is executed in the `usb_hid_free()` function, which is called when the driver is releasing control of the device.

For detailed information about writing HID subdrivers, see Writing HID subdrivers [p. 66].

9.2.3 Report parser

As it was already mentioned (see HID device characteristics [p. 39]), HID devices communicate with the host in form of so-called *Reports*. *Reports* are byte arrays in format specified by Report Descriptor. Report parser reads both parsed Report Descriptor and Reports and translate data from logical units, used by devices, and physical units, used by host. For basic introduction to HID Report parser API see section HID Report parser [p. 68] of the Writing HID drivers and subdrivers chapter.

The Report parser is responsible for making out the meaning of data sent between HID device and the host. As it was discussed above, data are sent in form of reports. Their structure is defined by the Report Descriptor. The Report Descriptor is a flat byte array where predefined tags can be recognized (see USB HID Class Definition, part 6.2.2) followed by the data of particular tag. There are three level of tags:

- **Main** — which define every single report item (or data field). Attributes of item are defined by current parsing state table and by tag's data
- **Global** — make presistent change of the current parsing state table.
- **Local** — change the current parsing state table with effect only to the first next Main tag

Always when Main tag is discovered, an appropriate count of report items (defined by the parsing state table) is created in the Report Structure.

Here it is needed to clarify the Report Structure (`usb_hid_report_t`). It contains all information from the Report Descriptor about all Reports (`usb_hid_report_description_t`) and its fields (`usb_-hid_report_field_t`). As the HID driver changed in time, the Report Structure also went through changes.

The first approach was having structure which understands not only the HID data, but also the needs of HID driver. The Report Structure was flat, meaning that all report items of all reports were in one long linked list. It was very comfortable, but also inefficient. HID driver could register callbacks for some types of HID devices or more precisely, for some specifical data sent by device. In fact there were two specific callbacks — one for keyboard and one for mouse. This approach is very limited and for example subdriver matching is impossible to make working.

The second approach was inspired by the LUFA[2] library. LUFA is an open-source USB library for the USB-enabled AVR microcontrollers, released under the MIT license. It is designed to provide an easy to use, feature rich framework for the development of USB peripherals and hosts. In this approach, Report Parser does not register any callbacks. Data are filled directly to the Report Structure, where they can be found by HID driver or subdrivers. The Report Parser always parses the whole report and after the parsing is done, whole Report Structure is sent to the particular driver which can decide which data to process and which not. That was for Input Reports, but for Output Reports it is very similar. Driver fills all data fields in the Report Structure he wants to fill and lets the Report Parser to fill the output report buffer. For detailed view on how the Report Structure could be traversed see HID Report parser [p. 68]. Also, due to the structured organization of the Report Structure the processing of report is much more effective.

9.2.3.1 Report Parser Limitations

As mentioned in section HID Report parser [p. 68], the Feature reports are not supported. On the Report Descriptor level they are well parsed, but the API for HID driver is missing. Feature Reports are very rare and we did not come across any device using them during development.

According to the USB HID Class Definition suggestion, the Delimiters are supported only in the basic form. Delimiters allow to define alternative Usages for one Report Item. Specification allows to reduce this set of Usages and make only the first one, the most preffered, accessible. For details see USB HID Class Definition, part 6.2.2.8.

The Report parser also omits the Designator Index values, which determines the body part used for a control (index points to a designator in the Physical descriptor) and String Index values, which allows a string to be associated with a particular item or control (index points to a String Descriptor).

9.3 Keyboard subdriver

The USB keyboard subdriver provides means to communicate with and use any standard-compliant USB keyboard. It takes care of device initialization, polling the device for Input reports and relaying parsed data to applications (actually only to the console), maintaining the state of locks (Caps Lock, Num Lock and Scroll Lock), synchronization of LEDs with the locks and auto-repeating of keys.

This subdriver is used either if the Report contains Collection with Usage Keyboard from Usage page Generic Desktop (for details about Collections and Usages see section Report and report descriptor parsing [p. 66]; detailed information are also in the USB HID class definition, part 6.2.2.6, Usage pages are listed in HID Usage Tables[3]), or Boot class keyboard for which there was no other subdriver matched.

[2]http://www.fourwalledcubicle.com/files/LUFA/Doc/091122/html/main.html
[3]http://www.usb.org/developers/devclass_docs/Hut1_12.pdf

The subdriver is located in /uspace/drv/usbhid/kbd/. The main structure, representing keyboard is usbhid_kbd_t, which uses the generic HID device structure (see above) and holds keyboard-specific information, such as currently pressed keys, active locks or information about auto-repeating of keys (see <usb/hid/kbd/kbddev.h>).

When the control over device is handled to the subdriver (by calling usb_kbd_init() from the HID core initialization function usb_hid_init()), it creates a new usbhid_kbd_t structure and initializes it. The initialization consists mainly of setting some default values, preparing buffers, creating a fibril for key autorepeat (see <usb/hid/kbd/kbdrepeat.h>) and creating a DDF function for the keyboard. It also sets the LEDs on the keyboard to the default value (Num Lock turned on, other turned off) and sets the Idle rate[4] to 0 (i.e. infinity — the keyboard will only report when a new event is available).

After receiving a report (via callback usb_kbd_polling_callback()) and parsing it, the subdriver evaluates which keys have been pressed or released, updates the state of modifiers and lock keys and notifies applications connected to the driver. In fact, currently the driver only communicates with the console to which it sends new key events. In order for the console to be able to connect to the HID driver controlling the keyboard device, the keyboard subdriver creates a DDF function (named keyboard) and adds it to the keyboard class. The console periodically checks this class for new functions and tries to create a connection to such function (i.e. driver). The connection from the console is handled in the \func({default_connection_handler()}) function, defined in kbddev.c.

9.4 Mouse subdriver

The mouse subdriver (located in /uspace/drv/usbhid/mouse/) takes care of HID devices providing mouse functionality. It is used either if the Report contains Collection with Usage Mouse from Usage page Generic Desktop, or as a fallback for any device declaring support of mouse Boot protocol.

In the initialization routine (usb_mouse_init()) it creates the main mouse structure (usb_mouse_-t) and initializes it by creating buffer for pressed buttons and creating DDF function (and adds it to class mouse so that the console will try to connect to this function as well).

When a new event occurs (usb_mouse_polling_callback()), the subdriver retrieves values for X and Y axes, for the wheel and for buttons. The movement in X and Y axes together with the buttons are sent to the console (if there is one connected) as mouse events. The wheel scrolling is treated as pressing arrow keys (up or down; actually one move of the wheel is interpreted as 3 presses of the corresponding arrow key).

9.5 Multimedia keys subdriver

Multimedia keyboards often have separate interface for multimedia keys. Though this is not specified by any standard, most of vendors use such separate interface and report the keys as Usages from the Consumer Usage page. This allows for a fairly generic way to handle these keys. Therefore, a specialized subdriver for these keys was added to the USB HID driver in HelenOS (it is located in /uspace/drv/usbhid/multimedia/).

This subdriver is initialized for any interface on which the device's Reports contain Usages from page Consumer. Currently, there is no use for these keys in HelenOS, but in the future it should be easy to add mapping of these Usages to any HelenOS keycodes (by altering the usb_hid_keymap_consumer static array in keymap.c) or to process them in other way (by modifying the usb_multimedia_polling_callback() function).

[4]Rate at which the device would periodically report events even if there are none new events ready.

9.6 Generic HID subdriver

In order to allow simple and straightforward extensibility of HID device support, a generic HID subdriver was implemented. This subdriver is initialized as a fallback for each device (more precisely for each interface) not supported by other subdrivers and additionaly for all supported devices (interfaces). Its main function is to provide interface for third-party applications to receive reports from the device.

It creates a node in the device tree (named `hid`) to which the applications may connect. The client interface is available in `<usb/hid/iface.h>` of `libusbhid`. It consists of four functions:

- `usbhid_dev_get_report_descriptor_length()` which returns length of the Report descriptor in bytes. This is needed to parse the Report descriptor in the application.
- `usbhid_dev_get_report_descriptor()` which retrieves the Report descriptor of the device.
- `usbhid_dev_get_event_length()` returns the **maximum** size of the incoming Report in bytes. (Useful to prepare a buffer for the Reports.)
- `usbhid_dev_get_event()` which retrieves the last Report reported by the device, together with its number (to distinguish already processed Reports).

This implies that the application receives raw data from the device and must do all the parsing by itself. It was designed this way because of problems with sending structured data over IPC. They would have to be serialized but then it is easier just to send the raw data from the device. This is also the reason for the first two interface functions — getting the Report descriptor and its length — as these are needed for initializing the HID parser. The HID parser is part of `libusbhid` and is located in `<usb/hid/hidparser.h>` and `<usb/hid/hiddescriptor.h>`.

Part IV

Writing USB drivers for HelenOS

Chapter 10

Introduction to writing USB drivers

This chapter is an introduction for writing drivers of USB devices and USB host controllers. It provides general guidelines for their writing and is followed by separate chapters describing implementations of USB device drivers and drivers of host controllers.

Following sections will describe features available to the developer of USB drivers and used conventions.

10.1 Common USB library

Because all USB drivers share some functionality (such as communication with their host controller), a common USB library exists. The library is located in `/uspace/lib/usb`, the actual library file is `/uspace/lib/usb/libusb.a` and the headers are in the `include/` subdirectory. All functions, types and macros are prefixed with `usb_` (or `USB_`) to prevent name clashes with other libraries.

When using the library, add `/uspace/lib/usb/include` to include path of your compiler. That way, you would be able to include `libusb` headers easily. Do not be tempted to add also the `usb` subdirectory directly because some files bear very generic names which could lead to confusion with headers from other libraries. Below is an example of `Makefile` settings when linking with `libusb` and part of C source that includes the base USB types and base definitions of HID class.

```
LIBS = $(LIBUSB_PREFIX)/libusb.a
EXTRA_CFLAGS += -I$(LIBUSB_PREFIX)/include
```

```
#include <usb/usb.h>
#include <usb/classes/hid.h>
```

Individual functions will be described in chapters regarding writing USB device drivers and writing host controller drivers. However, for detailed explanation, it is recommended to consult reference documentation (or, ultimately, the source codes). Below is an overview of functionality available in `libusb`.

Base USB types Type definitions (mostly aliases through `typedef` construct) for basic types such as USB address or endpoint number are present. Although C does not force strong type checking, using aliases is recommended for purposes of static analysis etc.

They are located in `<usb/usb.h>`.

Descriptor structures Standard descriptors (device, interface etc.) have their counterpart in `libusb`.

Descriptors' structures are located in `<usb/descriptor.h>` and class specific in appropriate files in the `classes/` subdirectory.

Actual communication with the device For communication with the physical device, drivers are using so called 'pipe API' that reflects the pipe concept (i.e. data channel with the device) described in USB specification.

The pipes structures are located in `<usb/pipes.h>`.

Wrappers for standard device requests Standard device requests, such as retrieval of descriptors, are implemented as wrappers over low level communication functions. Support for class-specific requests is limited but is expected to grow as more drivers are being added to HelenOS.

These wrappers are in class-specific headers and in `<usb/request.h>`.

Logging functions The library offers unified way for reporting errors and for dumping debugging information. See below for more details.

10.2 Logging and debugging

Currently, HelenOS lacks any central mechanism for logging system events (such as Unix **`syslog`**). To ensure some kind of forward compatibility, it was decided that all USB related drivers will use the same mechanism for reporting errors and for logging device events. The whole mechanism is actually only a set of functions for logging events backed by writing these messages to screen and to a disk file.

The idea is that all events the driver wants to record (and inform the user about) shall be reported via these functions. That ensures that all drivers use the same format which shall simplify debugging. The logging itself is done through a single function that takes logging level as a first parameter (see below for levels description) and remaining parameters are virtually passed as-are to a `printf` function for actual printing.

There are four levels for basic reporting. They are mostly designed for reporting errors with different severity levels. They are followed by several levels for debugging messages. The default setting for all drivers is expected to behave in following way: the four basic levels are always printed to the screen as they contain vital information and *all* levels are written to log file in the `/log` directory. Below is a list of logging levels with examples for their typical usage.

fatal Highest severity level used for reporting unrecoverable errors of the driver itself. Such messages are expected to be printed before total failure of the driver.

error Messages with this level reports serious problem with the controlled device, such as inability to control it. This level shall be used when the driver itself is healthy but is not able to control the physical device. As a matter of fact, this is the level that is most used in existing drivers because it is used for reporting error states that would rarely occur, e.g. no memory available or failure during IPC.

warning Level for reporting issues the driver is able to recover from gracefully. This level could be used for situations when the driver detects unexpected behavior of the device but is still able to operate it normally.

info Informational messages reporting (more or less) some success. Typical usage is informing that new device was attached and is already configured and initialized. This is the last level that is usually printed to the screen and thus shall be used modestly.

debug , debug2 These levels are used for debugging purposes. There is no strict definition for which actions each level shall be used. The purpose of more levels is to have the ability to limit the number of messages printed in some hierarchical way. For actual usage, common sense should prevail.

10.2.1 Logging API

All function used for logging are part of libusb library, declarations are in <usb/debug.h> header.

Logging is initialized with usb_log_enable() function, first parameter is maximum log level used for printing to screen (more precisely, to standard output), second is driver name. The name will be used as a file name for log copy on the disk and as a prefix of messages printed to screen. For example, generic USB printer driver would be typically initialized like this:

```c
int main(int argc, char *argv)
{
        usb_log_enable(USB_LOG_LEVEL_INFO, "usbprinter");

        /* Other initializations. */

        return ddf_driver_main(&printer_driver);
}
```

The actual logging is then done using the usb_log_printf(). The first parameter is logging level (see usb_log_level_t enum). Second argument is a formatting string followed by parameters used in the formatting string. As a shortcut, macros for individual levels exists (e.g. usb_log_warning(fmt, ...), usb_log_info(fmt, ...)) and their usage is preferred.

```c
        /* ... */

        void *buffer = malloc(buffer_size);
        if (buffer == NULL) {

                /* Notice that \n was inserted at the end. */
                usb_log_error("Out of memory (requested %zuB).\n",
                    buffer_size);

                /*
                 * Following command would have exactly the same effect
                 * but the former is easier to read.
                 */
                usb_log_printf(USB_LOG_LEVEL_ERROR,
                    "Out of memory (requested %zuB).\n",
                    buffer_size);
                return;
        }

        /* ... */
```

10.2.2 Dumping data buffers

At early stages of developing drivers, the developer often needs to dump data retrieved from the device in some raw format. For such situation the usb_debug_str_buffer() function is prepared that can be easily used together with the logging API and that dumps data buffer as a list of hexadecimal numbers.

The function has three parameters — pointer to the data buffer, buffer size in bytes and number of bytes to actually print (setting 0 as the last parameter is equal for setting the actual buffer size). Internally, the function is limited to output of only about 200 characters (that is about 60 bytes). If you need more, go into libusb/src/debug.c and change the value of BUFFER_DUMP_LEN constant.

This is an example of typical usage (however, the buffer would typically contain some dynamic data):

```
uint8_t the_data[] = { 1, 2, 15, 32, 60, 0, 17 };
usb_log_debug("Received data: %s.\n",
    usb_debug_str_buffer(the_data, 7, 0));
/* Will print "Received data: 01 02 0f 20 3c 00 11." */
```

General problem with functions dumping variable-length data in C is memory allocation. Both solutions — string buffer must be prepared by caller or the function will allocate the memory and the caller will deallocate it — are annoying. Especially when the string is only printed and then forgotten. With `usb_debug_str_buffer()` we decided to sacrifice safety (to some extent) in exchange for simplicity of usage.

Basically, the function uses static variable for the string and thus consequent calls destroy the previous data. However, this static string is actually fibril local and thus it is completely safe to use this function even in multi-fibril programs without any need for explicit locking. Internally, the function uses two buffers and thus it is possible to call the function twice in the same logging output.

10.3 Conventions on USB match ids

This section describes conventions used by existing drivers for creating device match ids. Match id consists of two parts — match score and match string. Only match strings will be discussed here because the score is a matter of each driver — how suitable it is for individual devices.

USB match strings uses formatting similar to query part of URLs. Each match string starts with `usb` prefix to separate match strings of USB devices from devices of other kinds. Individual (logical) parts are separated by ampersands `&` and for concrete values (e.g. driver for concrete version of a specific device) the format `attribute=value` is used.

The match strings are generated in `usb_device_create_match_ids_from_device_descriptor()` and `usb_device_create_match_ids_from_interface()` functions (sources in `recognise.c`). Below are several examples but the best way for creating driver match strings is to see the actual output from **`usbinfo`** application of a concrete device.

Below is a dump of match strings generated for mass storage interface (hence the `interface` part) of Kingston flash disk. The match strings are sorted by their score in decreasing order. Note that almost all numbers are printed in hexadecimal but always with standard `0x` prefix for emphasis.

```
# The most concrete specification includes vendor, product and release.
usb&vendor=0x0951&product=0x1614&release=1.10&interface&class=mass-storage& ↩
    subclass=0x06&protocol=0x50
# First, generic attributes are dropped (the vendor+product identifies the
# device uniquely anyway)...
usb&vendor=0x0951&product=0x1614&release=1.10&interface&class=mass-storage& ↩
    subclass=0x06
usb&vendor=0x0951&product=0x1614&release=1.10&interface&class=mass-storage
usb&vendor=0x0951&product=0x1614&interface&class=mass-storage&subclass=0x06& ↩
    protocol=0x50
usb&vendor=0x0951&product=0x1614&interface&class=mass-storage&subclass=0x06
usb&vendor=0x0951&product=0x1614&interface&class=mass-storage
# ...then only class identification with vendor id is left...
usb&vendor=0x0951&interface&class=mass-storage&subclass=0x06&protocol=0x50
usb&vendor=0x0951&interface&class=mass-storage&subclass=0x06
usb&vendor=0x0951&interface&class=mass-storage
# ...and the lowest score has generic mass storage device.
usb&interface&class=mass-storage&subclass=0x06&protocol=0x50
usb&interface&class=mass-storage&subclass=0x06
usb&interface&class=mass-storage
```

```
# The fallback match string will force that all devices (interfaces) will
# have a driver attached (otherwise they will not be listed in /dev/devices)
usb&interface&fallback
```

10.4 Device driver framework interfaces used in USB subsystem

This section describes implementation details in interfaces used for communication between USB drivers. The communication is done purely using interfaces provided by the Device driver framework. There are two interfaces for USB drivers: one for host controllers and one for function devices. Their description is followed by explanation of how the USB device finds the host controller (driver), how it is connected to the host controller and how it learns its own USB address.

10.4.1 Host controller interface

USB device drivers talk to the physical devices by submitting transfer requests to host controller. This communication is done through the host controller interface. The interface is used for following kinds of actions.

- Scheduling transfers on the bus.
- Registering endpoints.
- USB address management.

The transfer scheduling allows USB device to send and receive data from the physical devices.

Each endpoint the driver wants to communicate with must be registered with the host controller. That is needed to allow host controller to keep track of existing transfers, check bandwidth requirements and it might be used in the future for more strict permission checking. The endpoint registration is done automatically by the USB framework. The driver lists of expected endpoints and the framework matches them with the endpoints provided by the device and registers them. Host controllers can use prepared generic data structures for storing information about registered endpoints.

To allow unique address assignment to attached devices, a single register of used devices must be kept for each host controller. The host controller then provides means to register a new address or to query whether the address is in use (this is needed by **`lsusb`**).

To allow USB device drivers to determine USB addresses of the devices they control, the host controller keeps track to which (**`devman`**) device handle is each address assigned. This is called *address binding* and is done by the hub driver. That introduces a few problems that are described later.

The interface structure with callbacks that is supposed to be used by host controllers is in `<usbhc_iface.h>` (in `libdrv`) together with IPC protocol details. The actual callbacks are described in more detail later in chapter about host controllers.

The functions for client side — the device drivers — are in `libusbdev` and are part of the pipe API.

Note that reservation of default USB address is done by registering an endpoint on it.

10.4.2 USB device interface

The USB interface is currently needed only for device initialization. It must be implemented by all hub drivers, host controller drivers (see below why) and by drivers functionally similar to the multi-interface driver.

The interface is used by each USB device driver to get its assigned USB address, USB interface number and handle of the host controller driver.

The interface callback structure is in `<usb_iface.h>` (in `libdrv`) together with IPC protocol details.

Client side is hidden inside functions initializing the USB framework (`usb_device_create()`).

How the interface is actually used is described in next section.

10.4.3 Finding host controller and own address

When a driver for a USB device is created, it needs to learn what is its parent (which host controller) and what is its USB address. However the only driver it can connect to is its immediate parent, what is typically a hub. Thus the request needs to be forwarded and in some cases changed a bit to end with correct response.

The simplest problem is getting **`devman`** handle of the parent host controller. The device connects to the parent device driver and asks for it. The parent driver either has it cached or forwards the requests to its parent. This ends in the host controller driver that also must implement this method where it returns its own handle (more precisely, the handle of the DDF function implementing the USBHC interface).

Getting the assigned USB address is more problematic. As it was already mentioned above, the hub driver binds the USB address with a device handle assigned by **`devman`**. However, this handle is an ID of internal node existing inside the hub driver. The actual device node (controlled by the driver in question) has a different handle. That means that a driver cannot directly ask a host controller and give it its own handle. Such handle would not be found and the request would end with error. Thus the driver asks its *immediate* parent for it. The parent then inserts the correct value (it is the handle of the function to which the request came) and forwards the request.

Such request could be forwarded several times in which case the handle would be overwritten in each hub driver and the address of the topmost hub would be returned always. A special value zero is used to prevent that. If the interface method receives non zero value, it is supposed to forward it without any changes. For zero it shall substitute it with its own handle and forward it.

The multi-interface driver needs to look like another hub for each of the interface drivers and forward the requests to the real hub driver. Thus it is an exception to the ‘zero substitute’ rule as its forwards the zero because the MID device node handle is not the correct one. The correct one is substituted by its parent driver.

Most of the USB devices define their functionality at interface level. The MID driver takes care of that by creating a new device node and starting the appropriate driver for each interface. These drivers then operate on single interface only. Because single device can have several interfaces of the same kind, thus all controlled by the same driver, it is necessary to ensure that the driver would control only its own interface. This interface number is returned by the MID driver by a special interface method.

When developing a hub driver, one can use the prepared implementations of the interface methods from `<usb/ddfiface.h>` in `libusb`. The client side is part of the pipe API.

Chapter 11

Virtual USB devices layer

To speed-up development of device drivers at the beginning of the USB project a program simulating USB keyboard was written. The program was accompanied with virtual host controller and was originally intended as a temporary hack that would be removed when drivers for real hardware are ready. However the virtual USB keyboard can be used for testing the HID driver and thus the virtual host controller remained and this chapter describes its design and way how to write simulated USB devices.

The subsystem for virtual USB devices consists of two parts. First, it is the virtual host controller simulating a hardware host controller and next it contains applications simulating the devices. In order to make both parts smaller and thus more easily maintainable, the virtual devices are all standalone applications communicating through IPC with the virtual host controller.

The interface for communication between (simulated) host controller and device is described below.

11.1 Virtual host controller

The virtual host controller is a userspace application that provides two different interfaces. The first interface is towards USB device drivers where the program offers the same functionality as must be provided by any USB host controller driver in HelenOS. The second interface is for virtual devices and can be thought of as a simulation of the bus. The application can be also logically divided into two parts — hardware simulator (more or less) and a driver for the simulator.

Unlike real (i.e. hardware) host controllers, the virtual one does not work with transactions but only with transfers. Although originally the host controller split the transfers into individual SETUP, OUT and IN transactions, the idea was later abandoned because it complicated both the virtual host controller as well as the virtual devices. This can be thought of as a major difference but this difference is in no way propagated to the device drivers and simplifies the implementation a lot.

Part of the host controller is also a root hub that is driven by the hub driver [p. 36]. The simulated devices then appear as being plugged into the root hub.

11.2 Virtual devices

Virtual USB devices are userspace applications that simulate a life of a real USB device. When they are started, they connect to the virtual host controller and announce their presence. The root hub then simulates a change on the port and it is up to the hub driver to inform the device manager that new device was discovered. Later, virtual host controller would send transfers to the virtual device that shall answer them.

The idea is that the application simulating the real device would contain some state automaton describing the status of the device. The status could be changed by two ways: by itself or by the driver. For example, human interface devices would change by themselves, thus simulating human interaction and the driver only asks for current status. But simulator for mass storage would keep list of blocks and would change their content by commands from the driver and would not create any activity by its own.

The actual communication is done through IPC. The virtual device initiates the connection by connecting (via **devman**) to the virtual host controller that is expected to have always the same device path (`/virt/usbhc/hc`). The device then asks for a callback phone to allow host controller to initiate transfers. This way, the master-slave nature of USB is preserved even on the virtual layer.

11.2.1 Writing your own simulated device

If you want to write your own simulated USB device, this section will give you an overview what the `libusbvirt` offers and how to create such device. In this section it is expected that you know the USB architecture and terms well and the focus is on writing the actual device.

All functions and types needed for virtual USB are in `libusbvirt` library, to use it in your application do not forget to add following directives to your Makefile:

```
LIBS = $(LIBUSBVIRT_PREFIX)/libusbvirt.a
EXTRA_CFLAGS = -I$(LIBUSBVIRT_PREFIX)/include
```

The headers then appear in `usbvirt` directory.

The virtual device is represented by `usbvirt_device_t` structure. This structures contains device name (used for debugging messages), pointers to the 'ops' structure (`usbvirt_device_ops_t`) and pointer to device descriptors.

The device operations are callbacks that are executed when there are data on the virtual bus belonging to this device. The callbacks for default control endpoint are initialized differently than callbacks for normal data endpoints. The callback for control transfer — represented by `usbvirt_control_request_handler_t` — specifies the nature of the request (e.g. whether it is a standard or a class one) and the actual callback. There is an array of these callbacks, the last item must have handler set to NULL (and thus is ignored when looking for suitable handlers). Data callbacks are much simpler: for both in and out transfers there exists an array indexed by endpoint number where you register the callback.

The library itself implements several standard control handlers that take care of returning descriptors (see below) and setting device address. They can serve as an example of how the callbacks are specified:

```
usbvirt_control_request_handler_t library_handlers[] = {
        {
                .req_direction = USB_DIRECTION_OUT,
                .req_recipient = USB_REQUEST_RECIPIENT_DEVICE,
                .req_type = USB_REQUEST_TYPE_STANDARD,
                .request = USB_DEVREQ_SET_ADDRESS,
                .name = "SetAddress",
                .callback = req_set_address
        },
        {
                .req_direction = USB_DIRECTION_IN,
                .req_recipient = USB_REQUEST_RECIPIENT_DEVICE,
                .req_type = USB_REQUEST_TYPE_STANDARD,
                .request = USB_DEVREQ_GET_DESCRIPTOR,
                .name = "GetDescriptor",
                .callback = req_get_descriptor
        },
        {
                .req_direction = USB_DIRECTION_OUT,
                .req_recipient = USB_REQUEST_RECIPIENT_DEVICE,
                .req_type = USB_REQUEST_TYPE_STANDARD,
                .request = USB_DEVREQ_SET_CONFIGURATION,
                .name = "SetConfiguration",
                .callback = req_set_configuration
        },

        { .callback = NULL }
};
```

The data buffers passed to the callbacks are always allocated and deallocated by the caller and their data size is always provided. For transfers from device to host, you are expected to write the relevant data to these buffers and write how many bytes you actually wrote. For control read transfers, a simple helper exists `usbvirt_control_reply_helper()` that takes care of buffer size checking.

Each USB device is identified by contents of its descriptors. These descriptors are in virtual USB represented by `usbvirt_descriptors_t` structure that contains the standard device descriptor (the structure is defined in `libusb`) and pointer to configuration. The configuration contains the actual configuration descriptor and a list of other descriptors that shall be returned together with it. Typically, you will put interface and endpoint descriptors into this array. The ordering in the array is respected when data are returned to the driver.

Below is an example from virtual root hub of how the descriptors could be prepared. The actual descriptors are not shown, see `uspace/drv/vhc/hub/virthub.c` for complete source code.

```
static usbvirt_device_configuration_extras_t extra_descriptors[] = {
        {
                /* usb_standard_interface_descriptor_t */
                .data = (uint8_t *) &std_interface_descriptor,
                .length = sizeof(std_interface_descriptor)
        },
        {
                /* hub_descriptor_t (private in virtual host controller) */
                .data = (uint8_t *) &hub_descriptor,
                .length = sizeof(hub_descriptor)
        },
        {
                /* usb_standard_endpoint_descriptor_t */
                .data = (uint8_t *) &endpoint_descriptor,
                .length = sizeof(endpoint_descriptor)
        }
};

static usbvirt_device_configuration_t configuration = {
        /* usb_standard_configuration_descriptor_t */
        .descriptor = &std_configuration_descriptor,
        .extra = extra_descriptors,
        .extra_count = sizeof(extra_descriptors)/sizeof(extra_descriptors ↩
            [0])
};

static usbvirt_descriptors_t descriptors = {
        /* usb_standard_device_descriptor_t */
        .device = &std_device_descriptor,
        .configuration = &configuration,
        .configuration_count = 1,
};
```

Typically, all described structures will be prepared statically and in the `main()` function, the device will only register itself with the virtual host controller. That is done using the `usbvirt_device_plug()`

function that simulates attachment of a new device. Please notice that it is not possible to control more virtual USB devices from single application.

After the simulated device is plugged in, the respective driver will try to control the device and the callbacks would be fired. Meanwhile, the application shall simulate the life-cycle of the device, possible independently on the driver.

For example, a simulated human interface device can spawn a fibril that would change list of pressed keys with specified delay and the actual callback for data retrieval will simply return current state of the keyboard buffer. The actual code of the callback then could look like this (we omit locking as not much harm can be done — after all it is only about reading).

```
#define INPUT_SIZE 8
/* Array of pressed key events (schedule) in groups of INPUT_SIZE items */
static uint8_t in_data[] = {
        ...
};
/* Number of events. */
static size_t in_data_count = sizeof(in_data)/INPUT_SIZE;
/* Start reading at in_data_position * INPUT_SIZE. */
static size_t in_data_position = 0;

/* The actual callback. */
static int on_data_in(usbvirt_device_t *dev,
    usb_endpoint_t endpoint, usb_transfer_type_t tr_type,
    void *buffer, size_t buffer_size, size_t *actual_size)
{
        /* We shall verify that endpoint and tr_type has correct values. */

        static size_t last_pos = (size_t) -1;
        size_t pos = in_data_position;
        if (pos >= in_data_count) {
                /* Announce that the keyboard is gone. */
                return EBADCHECKSUM;
        }

        /* No new data, NAK the request. */
        if (last_pos == pos) {
                return ENAK;
        }

        if (buffer_size > INPUT_SIZE) {
                buffer_size = INPUT_SIZE;
        }
        if (act_buffer_size != NULL) {
                *act_buffer_size = buffer_size;
        }

        /* Simply copy the data to the buffer and update last position. */
        memcpy(buffer, in_data + pos * INPUT_SIZE, buffer_size);
        last_pos = pos;

        return EOK;
}
```

11.3 Virtual HID

The virtual HID is currently the only application that simulates a USB device (except for the root hub in the virtual host controller). Its main purpose is thorough testing of the HID parser that has to deal with complex Report descriptors.

The application was designed to be easily extensible. It is easy to add a new USB interface with a special report descriptor. Each interface then defines callbacks for data setting and retrieval and a special callback for simulating actions.

The application is located in `uspace/app/vuhid` and is launched with **`vuh`** command, parameters being names of interfaces the new device shall have.

If you want to extend the virtual HID with your own interface, probably the easiest way is to copy the existing interface and change it to your own needs. The simplest one is the one for boot protocol keyboard that is located in `uspace/app/vuhid/hids/bootkbd.c`. The interface is represented by `vuhid_-interface_t` structure, the individual attributes are described in the reference documentation. Once you prepare this interface, you add it to the list in `ifaces.h` and in `ifaces.c`.

You can then launch your new interface by specifying its name on the command line when starting the **`vuh`** application.

Chapter 12

Writing USB device drivers

This chapter will guide you through the process of writing driver for a USB device. It describes how to use the USB framework and functions that are available as part of the USB library.

12.1 Quick overview

This section and the next one will provide a high-level overview of the framework for writing device drivers. The following section will then guide you through the low-level details using simple mouse driver as an example.

12.1.1 The framework

The USB device drivers are using 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 extends the generic one with possibility to initialize endpoint pipes that will be needed by the driver. The author of the driver specifies which endpoints shall be present on the device through a USB driver structure and when new device is found, a specialized device structure is prepared and the pipes abstractions are initialized.

The driver itself lives the same life-cycle as any other driver controlled by **`devman`**. The generic framework does not support device unplugging yet. Although the USB hub drivers are able to detect device unplug, no special action is taken on the side of the drivers of the removed device because the backing framework is not able to handle such situation. Because of this, device drivers shall use some simple heuristics to stop controlling a device once there were several consecutive failed attempts to communicate with it.

12.1.2 Using the pipes

In USB, pipe is a basic abstraction for communication channel between a device and a host computer. Such channels have their representation in HelenOS as well. When using the provided framework, such pipes will be initialized for you and you can start using them right away.

Although the USB library is trying to hide low-level details of the IPC communication among tasks, there are areas when that is not completely possible. The pipes are used for transferring data from device to host (or vice versa) and thus they must be first transferred to (or from) the driver of the host controller. Such transfers involve IPC. The IPC is initiated through kernel and each open connection must be backed by some kernel structure. Because of this, inter-task connection shall be considered expensive and thus pipes must open this connection before transferring the data and close it after the transfer is complete. This is done by using sessions on the pipes.

Session is an open connection to the host controller driver and must be started before transferring data and shall be closed once the transfer is complete. Only when a session on a pipe is opened, it is possible to use the pipe for data transfer.

The pipe functions for data transfers always operate on data buffers provided by the user and they never allocate memory on their own.

Last thing that needs to be mentioned is the asynchronous nature of the transfer functions. For the programmer, all operations on pipes are blocking: execution will not return from the function before the requested operation is completed. However, drivers are using the asynchronous framework and are using fibrils. Thus, the function is blocking but may cause a fibril context switch and thus it is necessary to use fibril synchronization primitives when accessing shared data structures.

12.1.3 Automatic polling

Most input device drivers work in a similar manner. They keep polling the device for data and process the data when there are some. Translated into the code, it means writing an endless loop that opens a session over some interrupt in pipe, requests the in data transfer, closes the session and processes the data. For such situations, a wrapper function was created that takes care of the common code.

12.1.4 Adding child devices

Each device driver in device driver framework may create a child device that could be controlled by another driver. In USB, such child spawning comes in two flavors. First is a ‘hub case’ when hub or similar device detects new physical device. Second case is the MID driver one where the same physical device is run by several drivers (e.g. interface drivers).

The second case is not USB specific and functions provided by the `libdrv` shall be used. Namely `ddf_fun_create()` and `ddf_fun_bind()`.

For hubs, the device adding routine is part of bus enumeration [p. 10]. The enumeration is quite complex process and a special wrapper `usb_hc_new_device_wrapper()` exists in `libusbdev`. The function has a lot of parameters that are described in the reference manual in more detail. Most of them are rather auxiliary ones needed for proper functioning of the device driver framework.

One of the most important parameter is a callback for enabling a port. This callback must ensure that after its execution, the new device is accessible via default USB address. It is important to notice that this function is blocking (that is the reason why hub driver spawns a new fibril and uses a condition variable to return from the ‘enable port’ callback).

This wrapper is used both by the hub driver and by the UHCI root hub driver.

12.2 The API

This section will describe the API that was described in previous sections. The initialization is described below through an example for USB mouse because it is more understandable that way.

12.2.1 Pipes

The pipe is represented by `usb_pipe_t` structure and is initialized for you by the framework. However, if you want to initialize it manually, you first need to prepare a backing connection to the device — `usb_device_connection_t` — that represents the USB wire. Once you initialize the connection (typically by `usb_device_connection_initialize_from_device()`), you can initialize the pipe: for the default control pipe, there is a special function `usb_pipe_initialize_default_control()`, for other pipes you would use `usb_pipe_initialize()`.

To transfer the data over the pipe, a session on it must be started. That is done through the `usb_pipe_start_session()` function. The session is then closed by `usb_pipe_end_session()`. Both of these functions return error code but under normal circumstances, they shall succeed. Unless the pipe is not properly initialized, the function for starting a session will fail only if the host controller driver task died (or refuses connection) or the driver used all kernel resources dedicated for maintaining IPC. The first error is fatal but when something similar happens, the system is going down anyway and you cannot do much about it. Remedy for second kind of error is to close sessions as soon as possible.

The data transfer is done using `usb_pipe_read()` for in transfers (from device to host) and `usb_pipe_write()` for out transfers. Both of these functions expects a pipe with started session and a data buffer as parameters. The data buffer for out transfers must be already in USB endianness. Beware, some functions accept data in native endianness. The expected endianness is always stated in the reference documentation, next to parameter explanation.

Control transfers have their own functions for issuing transfers because these transfers could be bidirectional. For control read transfer, function `usb_pipe_control_read()` is ready, for write transfers there is `usb_pipe_control_write()`.

Functions described above are all part of the `<usb/pipes.h>`.

12.2.2 Automatic polling

For automatic polling, function `usb_device_auto_poll()` is available (in `<usb/devdrv.h>`). The function has quite a lot of parameters to allow as generic usage as possible. The obvious parameters are the pipe that is used for polling and the device to which the pipe belongs.

The next parameter is a callback that is issued when the transfer is complete. Parameters for the callback is the device that returned the data, data buffer and size of this buffer. The callback shall not try to destroy (e.g. deallocate) the buffer, that is responsibility of the callback. Since the generic routine does not do any parsing, the buffer is passed in USB endianness. The callback returns a boolean type determining whether the polling shall continue. Returning `false` will immediately terminate the polling loop.

The function has a second callback that is issued when the polling loop is terminated. The first argument is again the device followed by boolean specifying whether the termination was due to user (i. e. `false` returned from the first callback) or due to errors when communicating with the device. This callback can be set to `NULL` when the driver does not need to perform any action after termination of the polling.

The function accepts one more argument — a generic pointer that is passed as-is to both callbacks. This argument is useful for any user data and can be `NULL`.

The function spawns a new fibril and then returns, the return value indicates success. It is important to know that the new fibril — the polling one — may start execution even before the `usb_device_auto_poll()` returns. Thus, it is necessary to prepare all data structures and locking primitives prior to the call to this function.

12.3 Guide for creating simple USB mouse driver

This section will guide you through the process of creating a very simple, yet functional, mouse driver. Because the focus of the chapter is the USB part of the driver, we will omit how the retrieved data of the mouse would be used to actually move some pointer around the screen.

The mouse driver is part of the HelenOS distribution and used file paths refers to its actual placement within HelenOS source distribution.

12.3.1 Preparing the driver

The first steps when creating a driver are to create separate directory and to add this directory to HelenOS build scripts.

We will use `/uspace/drv/usbmouse` directory and we need to add corresponding entries to `/uspace/Makefile` (variable `DIRS`) and boot Makefiles (`/boot/Makefile.common` and architecture specific files `/boot/arch/*/Makefile.inc`.

Next, we need to create a file with match ids for the driver. Because we want to have the driver as simple as possible, we will focus only on devices supporting boot protocol. Such devices (device interfaces) must specify that the subclass is of a boot type and a mouse protocol. The match id created thus must have the following value (it needs to be stored in `/uspace/drv/usbmouse/usbmouse.ma`):

```
100 usb&interface&class=HID&subclass=0x01&protocol=0x02
```

You can also use the **`usbinfo`** application to print match ids of a certain device. Unless you are using some special mouse, this line (with different score) will be part of the output.

12.3.2 Driver initialization

The `main()` function of a USB driver is usually a very short routine that could consist of a single function call (`usb_driver_main()`) giving the control to the underlying framework to handle the rest. However, the argument to this function describes the driver and the devices it will control and its preparation requires explanation.

The argument passed is an instance of `usb_driver_t` which has two major items: driver operations and description of pipes the driver will use. The operations is a `usb_driver_ops_t` structure with callback that is executed when new device is about to be controlled by the driver. This function will be described later. The second part, pipes description, specifies which pipes shall be initialized by the framework.

The actual initialization is done by reading various device descriptors and then mapping them onto the driver expectations. That is because the device may actually have more pipes etc. (e.g. for some vendor specific functions).

The pipes are then available through the `usb_device_t` structure that is passed as an argument to the callback when new device is added. This structure will be described later.

Below is an excerpt from the mouse driver that specifies that the mouse must have an Interrupt In pipe that must be part of HID interface describing a mouse with boot capabilities.

```
#include <usb/usb.h>
#include <usb/pipes.h>
#include <usb/classes/classes.h>
#include <usb/classes/hid.h>

usb_endpoint_description_t poll_endpoint_description = {
        .transfer_type = USB_TRANSFER_INTERRUPT,
        .direction = USB_DIRECTION_IN,
        .interface_class = USB_CLASS_HID,
        .interface_subclass = USB_HID_SUBCLASS_BOOT,
        .interface_protocol = USB_HID_PROTOCOL_MOUSE,
        .flags = 0
};
```

When you will be writing your own device driver, you will read the pipes specification from the data sheet of the actual device or from generic specification of a USB class.

Preparing the driver structure is then straightforward. The implementation of the `usbmouse_add_device()` will be described in next section.

```
#include <usb/usb.h>
#include <usb/devdrv.h>

/* Driver name (the same as the directory where it resides).
 * Using NAME is only a convention but it is widely used in many HelenOS
 * services. */
#define NAME "usbmouse"

/* Callback when new device is to be controlled by this driver. */
static int usbmouse_add_device(usb_device_t *);

/* Currently, the framework supports only device adding. Once the framework
 * supports unplug, more callbacks will be added. */
static usb_driver_ops_t mouse_driver_ops = {
        .add_device = usbmouse_add_device,
};

/* Array of endpoints expected on the device, NULL terminated. */
static usb_endpoint_description_t *endpoints[] = {
        &poll_endpoint_description,
        NULL
};

/* The driver itself. */
static usb_driver_t mouse_driver = {
        .name = NAME,
        .ops = &mouse_driver_ops,
        .endpoints = endpoints
};

/* Main task routine. */
int main(int argc, char *argv[])
{
        /* Initialize the logging to some high value for debugging purposes. ↩
             */
        usb_log_enable(USB_LOG_LEVEL_DEBUG, NAME);

        return usb_driver_main(&mouse_driver);
}
```

12.3.3 Device adding routine

The framework calls the `usb_driver_ops_t.add_device()` when new device was plugged to the machine and by match ids it seems that the device shall be controlled by the driver.

The function has a single argument — `usb_device_t` structure representing the new device. This structure contains the already initialized pipes as they were described in the driver structure.

This routine is the place to create device functions and to bind interfaces with the device.

For the mouse, it can be actually very simple:

```
#include <usb/dev/driver.h>
```

```
#include <usb/hid/request.h>
#include <usb/debug.h>

static int usbmouse_add_device(usb_device_t *dev)
{
        /* Create the function exposed under /dev/devices. */
        ddf_fun_t *mouse_fun = ddf_fun_create(dev->ddf_dev,
            fun_exposed, "mouse");
        if (mouse_fun == NULL) {
                return ENOMEM;
        }
        int rc = ddf_fun_bind(mouse_fun);
        if (rc != EOK) {
                ddf_fun_destroy(mouse_fun);
        }

        /* Add the function to mouse class (ignore errors here). */
        ddf_fun_add_to_class(mouse_fun, "mouse");

        /* Set the boot protocol. */
        rc = usbhid_req_set_protocol(&dev->ctrl_pipe, dev->interface_no,
            USB_HID_PROTOCOL_BOOT);
        if (rc != EOK) {
                /* This is probably not crucial. If the mouse does not
                 * know how to handle this request, it is probably
                 * a boot-only mouse and we are safe anyway. */
                usb_log_warn("Mouse refuse to switch to boot mode.\n");
        }

        /* Start automated polling function.
         * This will create a separate fibril that will query the device
         * for the data continuously */.
        rc = usb_device_auto_poll(dev,
            /* Index of the polling pipe. */
            0,
            /* Callback when data arrives. */
            usb_mouse_polling_callback,
            /* How much data to request. */
            dev->pipes[0].pipe->max_packet_size,
            /* Callback when the polling ends. */
            usb_mouse_polling_ended_callback,
            /* Custom argument. */
            NULL);

        if (rc != EOK) {
                usb_log_error("Failed to start polling fibril: %s.\n",
                    str_error(rc));
                ddf_fun_destroy(mouse_fun);
                return rc;
        }

        usb_log_info("controlling new mouse (handle %llu).\n",
            dev->ddf_dev->handle);

        return EOK;
}
```

As you can see, everything is quite simple — the pipes are prepared as a `pipes` array using the same order as in the endpoint description and for mouse we only started the automatic polling.

12.3.4 Bypassing the framework

The USB-specific routine for adding USB devices has one disadvantage. There is no way you can possibly combine two different frameworks that use such customization. Thus the USB framework allows bypassing of this mechanism.

That is done in following way. You use the standard device driver framework with its `add_device()` callback and you create the USB device — represented by `usb_device_t` — manually. The callback for adding new device could look like this:

```
int my_add_device(ddf_dev_t *dev)
{
        int rc;

        /* Initialize the USB device. */
        usb_device_t *usb_dev = NULL;
        const char *err_msg = NULL;
        rc = usb_device_create(dev, endpoint_description, &usb_dev, &err_msg ↩
            );
        if (rc != EOK) {
                usb_log_error("Failed to create USB device `%s' (%s): %s.\n" ↩
                    ,
                    gen_dev->name, err_msg, str_error(rc));
                return rc;
        }

        /* Initialize the other framework. */
        ...

        /* Tell devman the driver accepts this device. */
        return EOK;
}
```

The `libusbdev` library allows even more fine-grained customization or bypassing of the framework.

For example, it is possible to initialize the pipes without creating container USB device or to retrieve standard descriptors manually. The library allows to change interface setting to an alternative one using the `usb_device_select_interface()` function. However as stated before, alternate interfaces were so far never tested and the implementation might contain minor bugs.

For details, consult the reference documentation. Such functions are located in `<usb/dev/driver.h>`.

12.3.5 Processing the data

This tutorial does not cover sending events to the system about pointer movements, we will shorten the callback to merely printing data it received.

Then, it is extremely simple:

```
/* Returning false means "stop the polling". */
bool usb_mouse_polling_callback(usb_device_t *dev,
    uint8_t *buffer, size_t buffer_size, void *arg)
{
        usb_log_debug("got buffer: %s.\n",
            usb_debug_str_buffer(buffer, buffer_size, 0));

        /* Print which buttons are pressed. */
        uint8_t butt = buffer[0];
        char str_buttons[4] = {
                butt & 1 ? '#' : '.',
                butt & 2 ? '#' : '.',
                butt & 4 ? '#' : '.',
                0
        };

        /* Count pointer movement. */
        int shift_x = ((int) buffer[1]) - 127;
        int shift_y = ((int) buffer[2]) - 127;
        int wheel = ((int) buffer[3]) - 127;

        /* Handle special cases. */
        if (buffer[1] == 0) {
                shift_x = 0;
        }
        if (buffer[2] == 0) {
                shift_y = 0;
        }
        if (buffer[3] == 0) {
                wheel = 0;
        }


        usb_log_info("buttons=%s  dX=%+3d  dY=%+3d  wheel=%+3d\n",
            str_buttons, shift_x, shift_y, wheel);

        /* We do not want to poll the device continuously.
         * Sleep for 1ms to allow the device some rest ;-). */
        async_usleep(1000);

        /* Tell the routine to continue polling. */
        return true;
}
```

12.4 Standard requests

The USB specification describes several control requests each device must respond to. These requests are used to query status of the device or for retrieval of device descriptors. These requests are very common and wrappers for them were created. The wrappers have declarations in `<usb/request.h>`, for parameters description consult the reference manual.

12.5 Standard descriptors

For most of standard USB descriptors there exist C struct counterparts in `libusb`. Generic structures (used by all devices) are stored in `<usb/descriptor.h>`. For class-specific descriptors, see corresponding files in `classes` subdirectory.

If you will create your own descriptors (e.g. vendor specific), do not forget that compiler will typically try to align the structure and adding `__attribute__(packed)` is necessary (although non-standard, this attribute is supported by most compilers).

12.6 Parser of USB descriptors

It was already mentioned [p. 9] that USB descriptors can be viewed as a tree but they are sometimes stored in a serialized manner. For example, configuration descriptor is followed by interface and endpoint descriptors and there is no way to retrieve these directly. A simple parser was created for deserialization purposes.

The parser tries to be as generic as possible and thus the interface it provides is somewhat terse. It operates on byte array and offers only functions for finding first nested descriptor and for finding 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 (see descriptor tree figure [p. 10]). 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 descriptors 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 pair where both parent and child are set to `-1`.

The parser structures and functions are stored in `<usb/dp.h>`. The parser itself uses `usb_dp_parser_t` structure. Currently it only contains array with possible descriptor nesting (`usb_dp_descriptor_nesting_t`). The data for the parser are then stored in `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()` takes parser, parser data and pointer to parent as parameters. For retrieving sibling descriptor (i.e. descriptor at the same depth) one can use `usb_dp_get_sibling_descriptor()`. This function also takes parser and parser data as parameters. But it requires two extra arguments — pointer to the parent descriptor (i.e. parent of both nested ones) and pointer to the preceding descriptor. The mentioned pointer must always point to the first byte of the descriptor (i.e. to the length of the descriptor, not to its type).

There is also a simple iterator over the descriptors. The function `usb_dp_walk_simple()` takes a callback as a parameter. This callback is then executed for each found descriptor and it receives current depth (starting with 0) and pointer to current descriptor. For other parameters, see reference manual.

Chapter 13

Writing HID drivers and subdrivers

13.1 USB HID library

To ease writing of HID drivers and subdrivers, all the common functionality was separated into the `libusbhid` library, located in `/uspace/lib/usbhid/`. Its funcions may be divided into few logical units:

Class-specific requests Functions for all HID class-specific requests are available, i.e. *Get/Set Report*, *Get/Set Protocol* and *Get/Set Idle*. For details about these requests, see the HID Specification[1], part 7.2. These functions can be found in `<usb/hid/request.h>`.

Report and report descriptor parsing This unit is more thoroughly described below [p. 66].

Usages' definitions and other constants These include constants for various Usage pages and usages from them (only those needed for the development; extending them is straightforward), located in the `usages/` subdirectory as well as other constants from the HID specification (located in `<usb/hid/hid.h>`).

HID interface for third-party applications This is the interface third-party applications can use to retrieve data from the HID driver. More detailed description can be found in Generic HID subdriver [p. 44].

13.1.1 Report and report descriptor parsing

HID devices communicate with the host in form of so-called *Reports*. Definitions of these reports are read from the Report Descriptor (see Standard USB Descriptors [p. 65]). HID parser is divided into two parts. At first, the report descriptor is parsed in **HID Report Descriptor Parser**. Later, particular reports are parsed using **HID Report Parser**. Both parts share one report structure for information interchange.

13.1.1.1 HID Report Descriptor Parser

HID Report Descriptor Parser reads and parses the Report Descriptor. For detailed information about Report Descriptor's format see the USB HID Class definition, part 6.2.2.

USB HID devices recognize three types of reports:

- **Input Reports** — host receives data from device

[1] http://www.usb.org/developers/devclass_docs/HID1_11.pdf

- **Output Reports** — host sends data to device
- **Feature Reports** — host sends configuration information to the device. These reports are not supported. They are correctly parsed from report descriptor but API for using them is missing.

Device could have more than one report of each type. In such case, *Report ID* must be set to identify the report. Using report IDs is mandatory for all reports if there are more reports for any type or if there is any report using report ID. In other cases, they could be ommited. Zero is not a valid report ID value and means that report ID is not used.

HID Report Descriptor Parser can be found in `usb/hid/hiddescriptor.h`. There is the main function of the parser `usb_hid_parse_report_descriptor()`. It takes the raw descriptor data (byte array) and fills the opaque report structure (`usb_hid_report_t`). This structure is then used for processing of all reports for the device.

Except this main function, there are also simple iterators over reports and their data fields. Function `usb-_hid_get_next_report_id` returns next report ID in the report structure to the given one. For listing all data fields of one particular report, there is `usb_hid_get_sibling()` function, which takes Usage path and pointer to the data field as parameter (or NULL) and returns the next one (or first one if NULL is given). Unlike `usb_dp_walk_simple()` for standard USB descriptors, there is no callback. Both these function can be found in `usb/hid/hidparser.h`. For other parameters, see reference manual.

13.1.1.2 Usage paths

In report descriptor, all data fields are organized into a tree structure of *Collections*, which are in fact only logical units packing some data fields together. Data fields can be only at the leaf level of the tree structure. Each collection (as well as data fields) has a Usage and Usage page assigned. Unlike the data fields, this pair of Usage and Usage page cannot be changed.

Pair of Usage and Usage page defines the meaning of field's data (e.g. mouse X axis movement). All of these Usages from root of the report descriptor up to the leaves create *Usage path* and each data field has one Usage path assigned. Drivers can use these Usage paths retrieve only specific data. There are five modes of Usage paths matching:

- **Strict** mode — Usage paths must be identical.
- **Begin** mode — Usage path specified by the driver must be prefix of the path provided by the device.
- **End** mode — Like **Begin**, must path must be suffix.
- **Usage page only** mode — only Usage pages in Usage paths are compared. Must be combined with either Begin or End mode.
- **Anywhere** mode — is limited to length of searched path to 1. Match if searching path contains the searched one at any level.

It is very recomended to use **Begin** mode, optionaly combined with the **Usage page only** one. The others are harder to use. If there are Array items in report, Usage paths related to these items end by pair of null Usage and null Usage page, because they depend on particular reports and they are filled after the report is parsed. These null Usages match all others (because of the cost of testing all possible Usages). In **Begin** mode driver can search only some particular **Collections** on upper levels.

The report ID can also be specified in the Usage path. Then the path can be matched only to a Usage path belonging to appropriate report. Report IDs are not considered when zero report ID is given. All functions working with Usage paths can be found in `usb/hid/hidpath.h`.

13.1.1.3 HID Report parser

HID Report parser uses the previously filled *Report structure* and makes value translation between logical units, used by the device, and physical units, used by the host. Also fills in (in case of Input reports) the correct Usage and Usage page for data fields if needed (only Array items change the Usage and Usage page according to the processed data).

The `usb_hid_parse_report()` takes report structure and raw report as parameters and fills the found values to the report structure. Driver can pick up the data using the iterator.

On the other hand, function `usb_hid_report_output_translate()` picks up data from the report structure and fills the output report buffer.

In Reports, two types of data fields can be recognized:

- **Variable items** — values of these fields represent value of some physical control. For example, a mouse movement in X axis. These fields have its Usage and Usage page fixed.
- **Array items** — values of these fields represent index to the array of possible Usages. Array items can be used only for detecting presence of some event — pressing a key for example. Its Usage and Usage page can (and will) be changed as another report comes.

Each data field has also flag if it is **constant** or **variable**. Value of Constant data fields cannot be changed neither by host nor by device. HID Report parser skips constant fields in `usb_hid_get_sibling()`.

For details about Usages and Usage pages see Usage Table specification [2].

13.2 Writing HID subdrivers

The easiest way to improve support of HID devices in HelenOS would be to utilize the available facilities provided by the HID driver and the HID framework (see above) and to write a subdriver for handling a particular type of device. This comprises two steps:

1. Specifying the type of device, or type of data it must provide in order to be controlled by this subdriver. This is done by filling in the `usb_hid_subdriver_mapping_t` structure (see `<usb/hid/subdrivers.h>`) and adding it to the `usb_hid_subdrivers` array in `subdrivers.c`.
2. Implementing callbacks for several situations the HID driver may encounter. (Note that any of the callbacks may be set to NULL if no special handling of such situation is required.)
   a. Initialization (`usb_hid_driver_init_t`) — this callback will be executed after the HID device structure (`usb_hid_driver_poll`) is initialized. The subdriver may do any initialization of its own structures, or of the device (with which it can communicate using the `usb_device_t` structure stored inside the HID device structure.
   b. New data from device (`usb_hid_driver_poll_t`) — this function will be called when there are new data ready at the device.
   c. Polling ended (`usb_hid_driver_poll_ended_t`) — the polling of the device was terminated.
   d. Deinitialization (`usb_hid_driver_deinit_t`) — called when the driver is destroying the HID device structure.

   These callbacks are grouped into a structure representing the subdriver (`usb_hid_subdriver_t`), together with a data pointer to store any subdriver-specific data. This structure is also part of the mapping mentioned above, so that the mapping maps the callbacks to the particular device.

[2] http://www.usb.org/developers/devclass_docs/Hut1_12.pdf

The subdriver source files must be then added to the HID driver `Makefile` in order to be compiled into the driver. When this is accomplished, the HID driver will try to match any device it is supposed to control, to this subdriver. All the callbacks will receive the HID device structure as the first parameter, i.e. they will have direct access to the data about the device, as well as the incoming reports (stored in the structure as well).

13.3 Writing HID drivers

In case the options provided by the subdriver system are for any reasons not satisfactory, or if the implementor does not wish to leave the polling to the `libusbdev` library, it is also possible to write a whole separate driver for any kind of HID device, with the help of the USB HID library [p. 66]. In such case, the driver must specify the match IDs (see conventions [p. 49]) appropriately, so that it will be launched by the DDF.

A simple example is the mouse driver mentioned in the previous chapter [p. 59].

Chapter 14

Writing USB host controller drivers

The most common USB 1.1 host controllers follow UHCI or OHCI specification and are supported by the implemented drivers. However, it is not forbidden to use different host controllers that do not comply (partially or at all) with those specifications, these would require a specific host controller drivers. This chapter presents a simple how-to manual and uses EHCI host controller as an example.

14.1 EHCI

EHCI brings support for high speed signalling on USB bus. If the ability to mix USB 2 and USB 1.1 devices on one bus is not required, it is fully possible to use USB 2 devices in the current state of USB support for HelenOS, once EHCI driver is implemented.

14.2 DDF and initialization

A host controller device is usually a PCI device, thus it is necessary to know its assigned vendor and product id. This device provides functionality for both host controller and root hub. EHCI shares status register and interrupts between host controller and root hub, thus it will need to emulate USB hub and handle USB requests using device register access. EHCI driver will need to initialize USB bus driver structures (`usb_device_keeper_t` and `usb_endpoint_manager_t`) as well as its local structures that can be found in EHCI specification in chapter 3.

14.3 Root hub

Root hub function will be at the top of USB device hierarchy and it needs to implement `usb_iface_t` interface. This interface is defined in `<usb_iface.h>` in `libdrv`. It consists of three functions:

- `get_address` is used to map device handle to assigned USB address, the root hub implementation will have to access host controller structures directly or forward this call to the host controller function.
- `get_interface` is used by multi-interface devices and does not have to be implemented.
- `get_hc_handle` is used to gain direct access to the host controller. Root hub's implementation should return handle of its sibling function.

14.4 USB bus driver back-end

Host controller interface is defined in `<usbhc_iface.c>` in `libdrv`. Functions in this interface can be divided into two categories, the first one can be called USB bus driver back-end functions: * `reque-st_address` is called when a new address for a new device is needed. * `bind_address` binds used USB address to devman device handle. This connection is used by the driver to get its USB address and by usb utilities to query devices based on drivers. * `find_by_address` provides information about device address to driver mapping. * `release_address` returns address requested by `request_address`. It may be caused by device initialization failure, or device unplugging.

All functions that manipulate addresses should be handled by `usb_device_keeper_t` in the EHCI driver.

- `register_endpoint` registers pipe back-end.
- `unregister_endpoint` unregisters pipe back-end.

These two functions should be handled by `usb_endpoint_manager_t`. As there is no per endpoint EHCI specific data structure, the driver does not need to add any special functionality to this calls.

14.5 Host controller data transfers

The other part of `usbhc_iface_t` interface handles data transfers:

- `interrupt_out`
- `interrupt_in`
- `bulk_out`
- `bulk_in`
- `control_write`
- `control_read`

All these functions are similar. They need to setup internal memory representation of the requested transfer and enqueue this representation into internal schedule.

14.6 Functionality

Using EHCI driver implementation mentioned in this chapter it will be possible to use high speed devices. It will not be possible use USB 1.1 devices connected to high speed hubs. This feature requires utilization of Transaction Translation features and more extensive changes to the USB framework.

Part V

Appendices

Appendix A

User documentation

HelenOS USB subsystem brings support for USB devices into HelenOS, a micro-kernel multi-platform operating system. This manual describes its capabilities and additional tools for the HelenOS user.

A.1 Supported platforms

Currently the HelenOS USB subsystem was tested on IA-32 and AMD-64 platforms. Any current PC should be able to use it, unless its USB chip fails to meet USB requirements specified by USB specification.

A.2 How to get HelenOS

HelenOS can be downloaded either in form of source files or binaries. The easiest way to get HelenOS run is to download binary for your platform from official HelenOS web site[1] (in most cases in form of bootable CD image) and depending on the form of the binary either use it as a machine image for target platform emulator or write it on a compact disk and boot the system from the CD[2].

A.3 Get the newest USB-development version

To ensure you have HelenOS with the latest version of USB subsystem, you may want to download source files from repository of the project and compile them by yourself. It is recommended that you build the project on Linux system, as many tools used to build the project are much harder to use in other systems' environments.

First you need to download the sources from repository. For that you need to install a client for Bazaar, an open-source distributed version control system. Then just type in the console

```
bzr branch bzr://helenos-usb.bzr.sourceforge.net/bzrroot/helenos-usb/ ↩
    mainline helenos_usb
```

for downloading branch of the HelenOS USB development team. This will download source files for whole HelenOS into `helenos_usb`. After that, switch to the new directory and run **`make`**.

[1] http://www.helenos.org/

[2] Note, that at the time of writing this documentation the USB subsystem has not been merged into main HelenOS repository yet.

```
cd helenos_usb
make
```

A menu with build options will appear. Select preconfigured defaults for `ia32` or `amd64`, disable `SMP support` and confirm to start the build. After the build the `helenos_usb` will contain ISO image of bootable CD with HelenOS `image.iso`.

A.4 Subsystem capabilities

Current version of HelenOS USB subsystem supports any USB 1.1 compliant device given that there is the right device driver in the system and it does not require isochronous data transfers (those are used mostly for multimedia devices, which are not supported in HelenOS). This includes USB hubs and human interface devices such as mice or keyboards. Subsystem is also able to recognize most other USB devices. However, without appropriate drivers for them it is not able to use them.

USB devices can be plugged at any time during the system run or even before. Most of them should be able to gracefully recover from unplugging.

A.5 Tools

This chapter describes USB related command-line tools available for users of HelenOS.

A.5.1 List of attached devices and host controllers — `lsusb`

The only purpose of this tool is to list host controllers found on the current machine and list of USB devices currently attached to them.

Unlike the original **`lsusb`** from Linux, this one does not aim to provide detailed information about attached devices. You are supposed to use **`usbinfo [p. 74]`** instead.

Below is an example output from QEMU when three mass storage devices were attached to it.

```
Bus 01: /hw/pci0/00:01.2/uhci-hc
  Device 01: /hw/pci0/00:01.2/uhci-rh/usb00_a1
  Device 02: /hw/pci0/00:01.2/uhci-rh/usb01_a2
  Device 03: /hw/pci0/00:01.2/uhci-rh/usb01_a2/usb00_a3
  Device 04: /hw/pci0/00:01.2/uhci-rh/usb01_a2/usb01_a4
```

A.5.2 USB device information — `usbinfo`

This section describes how to use the **`usbinfo`** application available in HelenOS. This tool can be used to query USB devices currently connected to the computer and was originally inspired by Linux **`lsusb`** command.

The **`usbinfo`** is a tool primary for developers of USB device drivers. It prints rather low level information about the device that is usually of little importance for end user. Below is a list of features provided by **`usbinfo`**.

- report status of the devices
- print standard descriptors provided by the device

- print all string descriptors
- list match ids describing the device

usbinfo is a command line tool and required actions are specified as switches.

A.5.2.1 Specifying the device

There are two options how to specify the device. First one is numerical, using bus number and device address The second one uses the device's hardware path.

If you want to use the numerical variant, you ought to know which bus number was assigned to the host controller the device is attached to and what is the device address. You can get these from output of **lsusb [p. 74]**. You then specify the device as host controller number, followed by period or colon and by device address.

The hardware paths are listed in the `/dev/devices` directory. The only difference is that file entries in the `/dev/devices` use backslashes as hardware components delimiter, while **usbinfo** expects normal forward slashes. Actually, the backslashes under `/dev/devices` are there only to bypass current limitations of **devmap**.

For example, when HelenOS is run under QEMU with option `-usbdevice mouse`, following entries will appear after issuing **ls /dev/devices**:

```
\hw\pci0\ctl
\hw\pci0\00:01.0\ctl
\hw\pci0\00:01.0\com1\a
\hw\pci0\00:01.2\uhci-hc
\hw\pci0\00:01.2\uhci-rh\usb00_a1\ctl
\hw\pci0\00:01.2\uhci-rh\usb00_a1\HID0\mouse
```

The last two entries represent the mouse and both can be used to retrieve the information about it.

More devices can be specified at the command line. The device specification (either through numbers or through full path) is a 'file' option and thus is not preceded by any switch.

A.5.2.2 Specifying the actions

By default, **usbinfo** will print short device identification, printing code of the vendor and code of the device.

This behavior can be altered via switches described in following paragraphs.

Short help can be printed with `--help` switch. This switch stops any further processing and no device is queried at all.

The default **device identification** mentioned in the first paragraph can be forced with `--identification` switch.

To see what **match strings** will be generated by hub when the device is connected, you shall use the '--match-ids` switch. Although the creation of match ids is described formally elsewhere [p. 49], sometimes it is easier to plug the device in and see `in natura' what match ids will be generated.

Standard descriptor can be printed with `--descriptor-tree` switch. For detailed descriptor dump, use `--descriptor-tree-full`.

To print **string descriptors** use `--strings` switch. The string dumping is very naive because it tries to obtain first few strings regardless whether they are actually referenced. This could lead to retrieval of invalid data. That is not an error of the device but rather rough approach of the **usbinfo** application.

The **status of the device** could be printed with `--status` switch.

Below is a short summary of available options together with their short variants.

Long option	Short option	Option description
`--help`	`-h`	This summary.
`--identification`	`-i`	Short device identification (default).
`--match-ids`	`-m`	Match ids.
`--descriptor-tree`	`-t`	Brief descriptor tree.
`--descriptor-tree-full`	`-T`	Detailed descriptor tree.
`--strings`	`-s`	String descriptors.
`--status`	`-S`	Device status.

Table A.1: Summary of **`usbinfo`** switches

A.5.3 Print out pressed multimedia keys — `mkbd`

This is a proof-of-concept application to demonstrate the capability of the HID driver to support multimedia keys on keyboards and the interface the HID driver provides for other applications.

The **`mkbd`** application can communicate only with devices providing certain interface. The HelenOS USB HID driver creates device nodes named `hid` for each HID device present in the system. Only these device nodes can be used with **`mkbd`**. Moreover, only nodes created for the multimedia keys device will actually report something.

It is a command line tool and takes exactly one argument — the path to the device you wish to monitor. The path may be in format of the whole path to the device node under `/dev/devices` or in shortened form: `<bus_number>:<device_number>/rest/of/path`.

After successful startup, the application will print out the Usages assigned to the pressed multimedia keys.

Appendix B

Future development

Implementation of USB subsystem for HelenOS is a good base for further extension of HelenOS capabilities. One of these capabilities may be the support for USB 2 host controllers and devices. Its possible implementation is discussed in section USB 2.0 [p. 78] below. Other fields of future development include additional drivers, support for isochronous transfers (multimedia devices), power management or support for alternate configuration and interfaces.

B.1 Support for device unplugging

The device driver framework does not provide functionality for handling device unplugging. The USB drivers are almost ready when such support would be added.

The drivers use a heuristics that several consecutive failed attempts for communication with them means the device was removed by a user. The driver then tries to free all allocated resources.

Obviously, changes to the drivers would need to be made once the unplugging support is available. For most of the drivers, the changes would be rather trivial. Host controllers are not expected to be hot-pluggable devices and support for device unplug is not necessary.

B.2 New device drivers and support for isochronous transfers

There are many drivers that can be implemented such as driver for mass storage devices. It was the goal of the project — to create a robust platform for writing USB device drivers.

USB subsystem currently lacks support for isochronous transfers. These are multimedia transfers and there is no multimedia support in HelenOS. Thus the feature was not implemented as there was no suitable way to test it.

Implementation of this feature should include

1. Framework for shared memory buffers between tasks with focus on multimedia data.
2. At least one driver that would use isochronous transfers and the multimedia framework for data transfers
3. A simple application that would be able to use data flow from or to a device using isochronous transfers

The implementation would require changes to the current USB transfer scheduler. Current schedulers are not ready for time critical transfers and such needs are ignored.

B.3 Power management

Another field of future development is implementation of power management. HelenOS currently lacks any power management features. If it is implemented, it will suspend or resume particular USB devices or a whole USB device tree at once.

Suspending a device from power management task would require it to obtain device's parent handle in the DDF tree and ask the parent to suspend/resume the device. Hub driver and UHCI root hub driver would implement a call to suspend/resume a specified device. They would also handle 'suspended status' port change — this is not implemented as currently there is no way a device could be suspended.

HCD is responsible for suspending entire bus and all connected devices. Implementation would require a suspend and resume call for HCD as well as implementation of these commands in HCD.

B.4 Alternate configurations and interfaces settings

Support for more device configurations is not present at all (because hardware vendors do not use this feature much) but adding it is possible. The following changes would have to be done:

1. Support for device unplugging. It is necessary that the drivers can give up the control of the device because the other configuration may provide completely different set of interfaces.
2. Adding tools to allow user to select alternate configuration.
3. Changing the hub driver (and possibly MID driver, too) to support configuration switch.

Support for alternate interfaces is implemented but was not tested. The framework always chooses the first setting and support for interface switching would involve following changes:

1. Possibly add logic to switch interface automatically to the respective drivers according to properties in each of the settings.
2. Adding tool to switch the interface. This tool would need to work at interface level.
3. Minimal support for device unplugging because interface change might involve destruction of existing DDF functions and creating new ones.
4. Fix possible problems in existing implementation of interface switching.

B.5 USB 2.0

This section focuses on USB 2 and its influence on USB 1.1 drivers implementation. It discusses features of USB 2 enabled hosts that need to be considered by OS drivers developers.

B.5.1 Backwards compatibility

Both USB 2 hosts and devices are designed to be backwards compatible with USB 1.1. High speed devices (USB 2) must be able to communicate at full speed and full speed mode is the default. Special handshake is used during reset stage of device enumeration to determine whether a device is high speed capable. From the host's perspective, USB 2 controller (defined by Enhanced Host Controller Interface — EHCI) provides a separate bus and backwards compatibility is achieved by sharing root hub ports with USB 1.1 controllers, so called companion controllers (using USB 1.1 devices connected to other than root hub, uses transaction translation capabilities of high speed hubs and is beyond the scope of this text).

If an EHCI controller is configured, all ports are controlled by its driver. In the case of low speed devices, or full speed devices that fail to respond to high speed identification sequence, the port control is released to one of the companion controllers.

B.5.2 BIOS and OS control

USB 1.1 defined a legacy support for USB mice and keyboards and USB 2 extended this idea to so called Pre-OS USB control. It is used to support USB devices before an OS starts, like booting from USB dongle. EHCI specifies semaphores and a routine for graceful handover of hardware control from BIOS/firmware to the OS. If the OS is not aware of USB 2 functionality this control stays with BIOS and may cause problems (there may be a BIOS setting controlling this feature).

Therefore, a STUB driver in `uspace/drv/ehci-hcd` for EHCI controllers is provided. This driver recognizes EHCI host controllers and gains control from any pre-OS software in `pci_disable_legacy`. It turns off any USB 2 host controller found, releasing control of its root hub ports to companion USB 1.1 host controllers.

Appendix C

Project timeline

This appendix sums up the realization of the project in a chronological order.

August 2010	Getting to know HelenOS. First experiments with writing code for HelenOS. Setting up tools for project (Trac, Bazaar repository, mailing-list, calendar).
September — October 2010	Study of USB and USB HID specifications. Virtual host controller driver implemented. First draft of project architecture.
11th November 2010	Official start of the project.
November 2010	Modifying architecture in order to use the new Device Driver Framework. First draft of USB framework.
December 2010	First drafts of HC driver, hub driver and keyboard driver.
January 2011	Hub driver implemented. First draft of HID report parser. basic UHCI driver and UHCI root hub driver implemented.
4th February 2011	Prototype of the project completed, only basic features supported.
February 2011	New version of the USB driver API, porting the existing code to it Modifying code to reflect changes in DDF. Driver for multi-interface devices. Completed support of standard keyboards.
March 2011	Testing with real hardware. Started documenting the project. Basic HID descriptor parser and basic generic HID report parser Support for generic (non-boot) reports in HID driver.
April 2011	OHCI driver implemented. Improving HID parser. Mass storage driver stub. EHCI driver stub. Subdrivers in HID driver. Testing with real hardware, debugging.
May 2011	Reorganization of libraries. Testing with real hardware, debugging. Tools implemented (**`lsusb`**, **`usbinfo`**, **`mkbd`**). Finished documentation. Presentation for other HelenOS developers.
2nd June 2011	Project delivered.

Table C.1: Project timeline