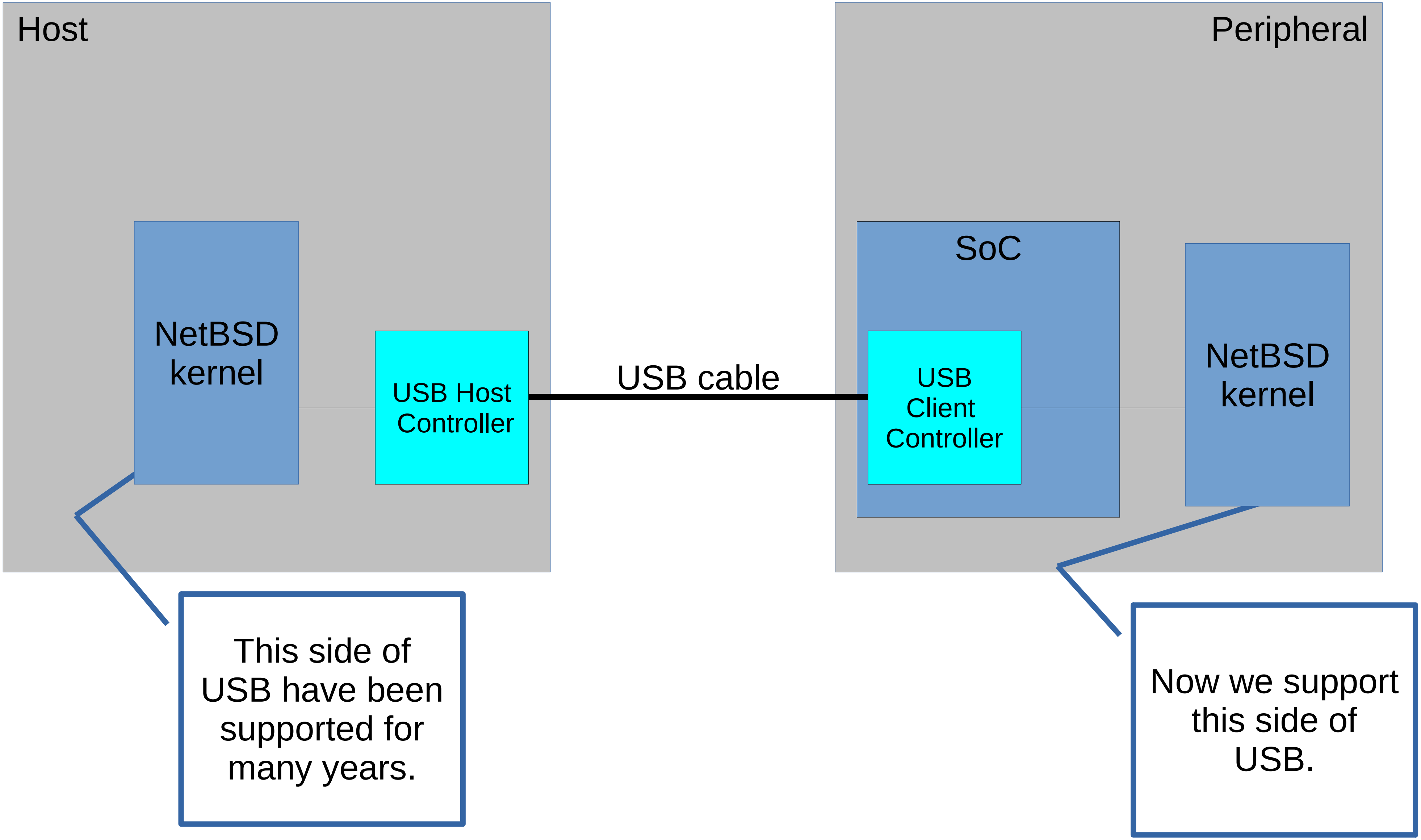
Peripheral-side USB support for NetBSD

Hiroyuki Bessho
(別所 博之)

bsh@NetBSD.org,
bessho@genetec.co.jp

What is this?

- With this framework, Platforms that run NetBSD kernel can act as USB devices, such as USB serials, mass storage, printers, Ethernet adapters, etc.



Host

NetBSD kernel

USB Host Controller

USB cable

Peripheral

SoC

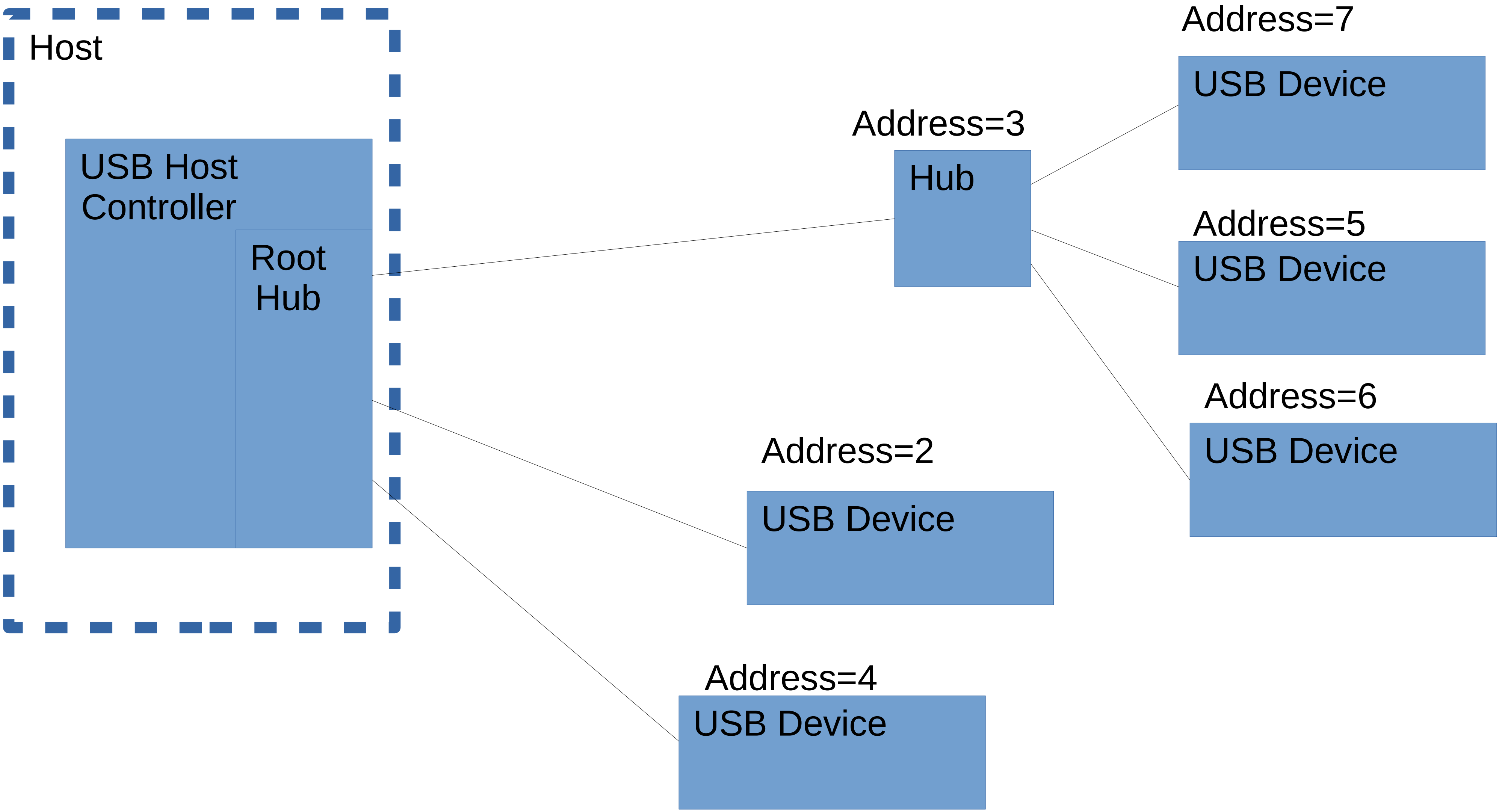
USB Client Controller

NetBSD kernel

This side of USB have been supported for many years.

Now we support this side of USB.

USB Bus



Host

USB Host Controller

Root Hub

Address=3

Hub

Address=7

USB Device

Address=5

USB Device

Address=6

USB Device

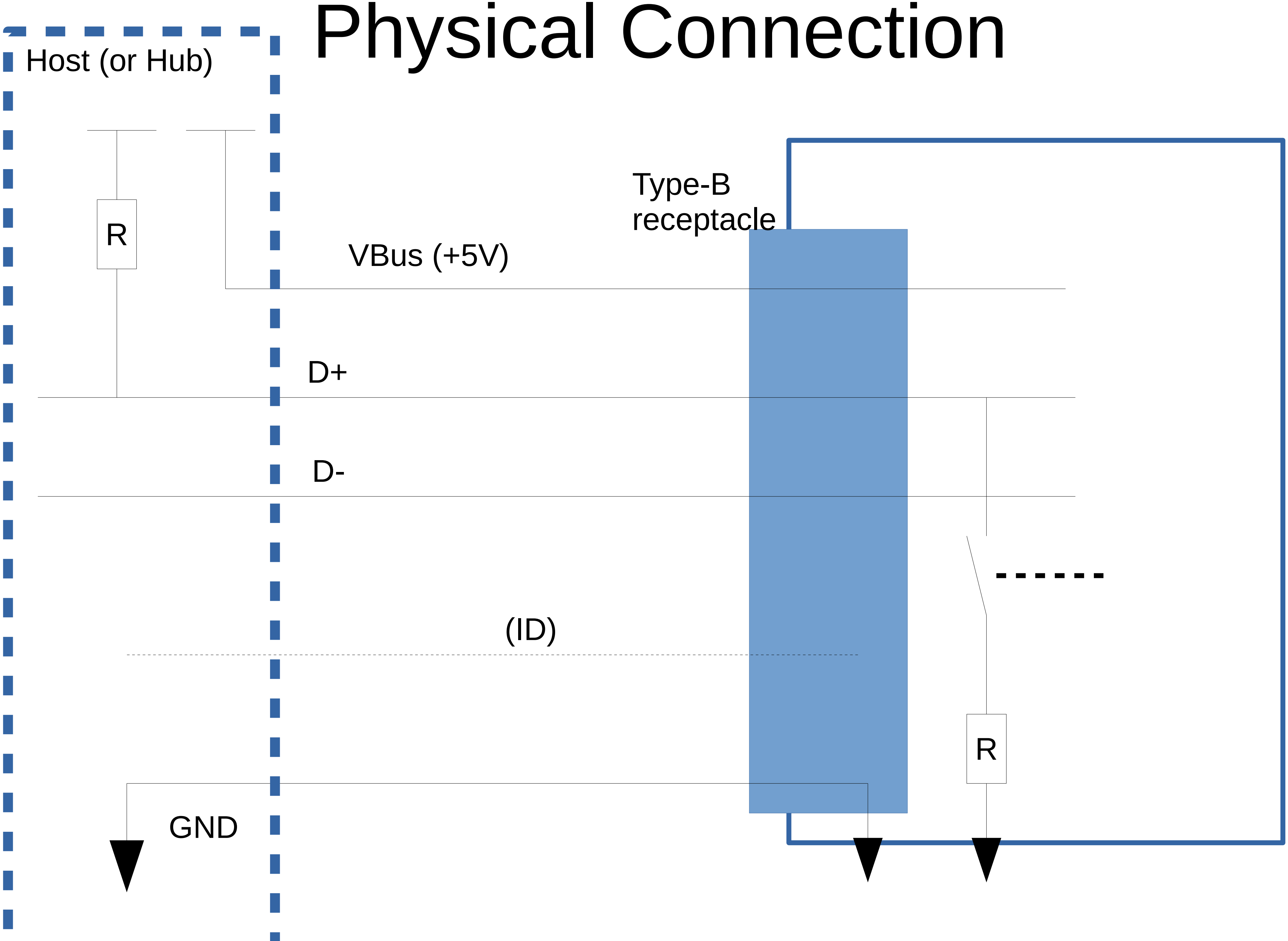
Address=2

USB Device

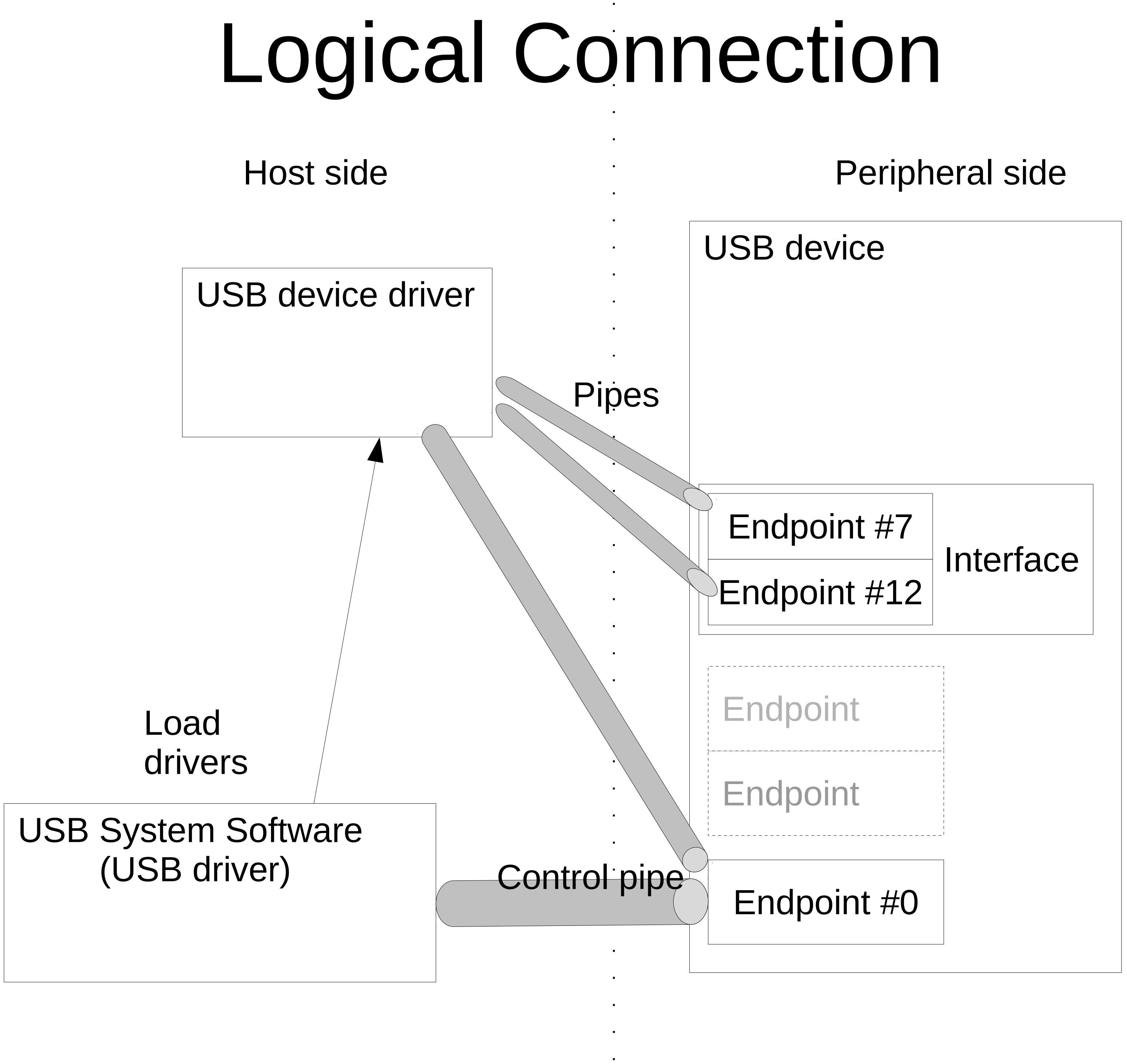
Address=4

USB Device

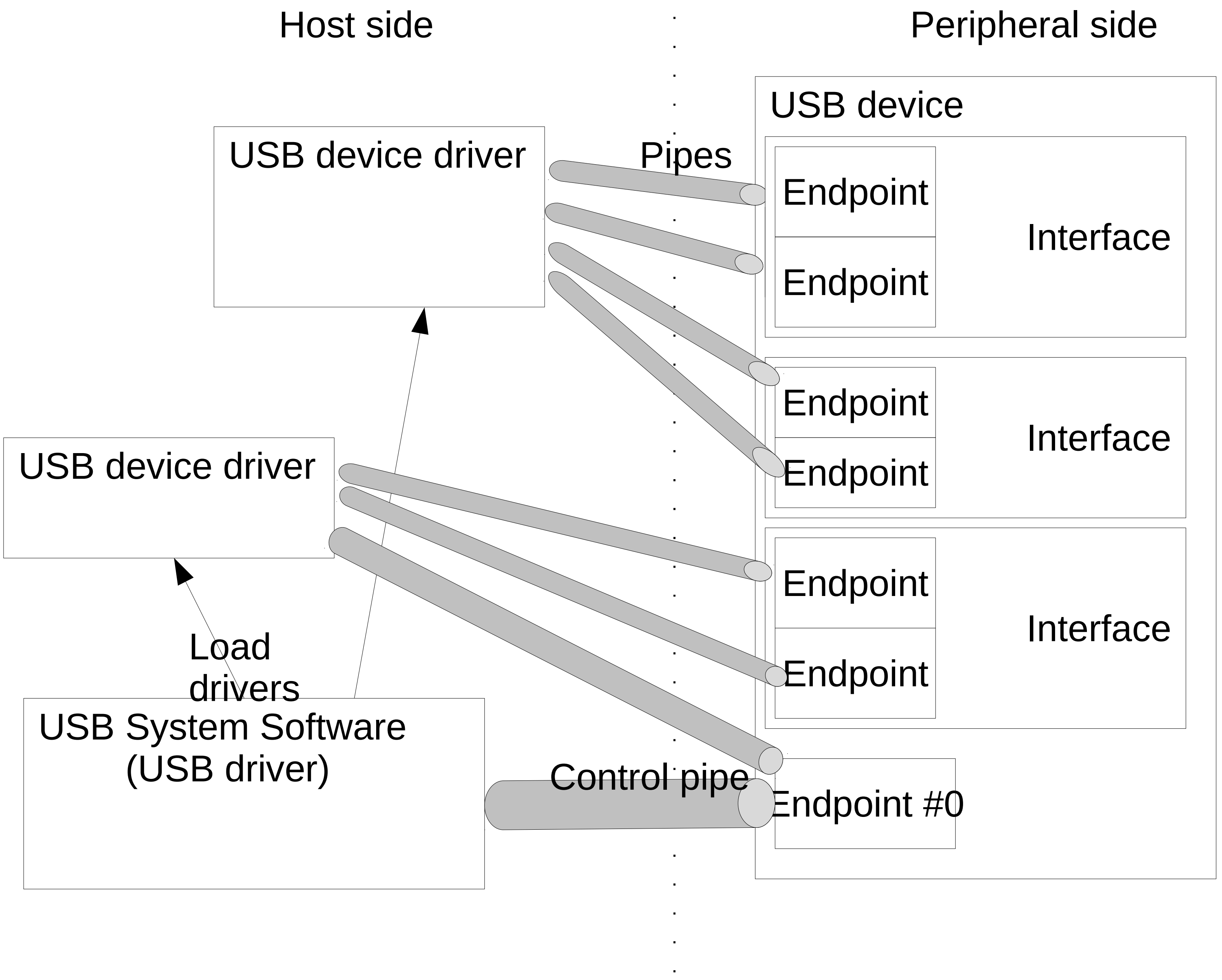
Physical Connection



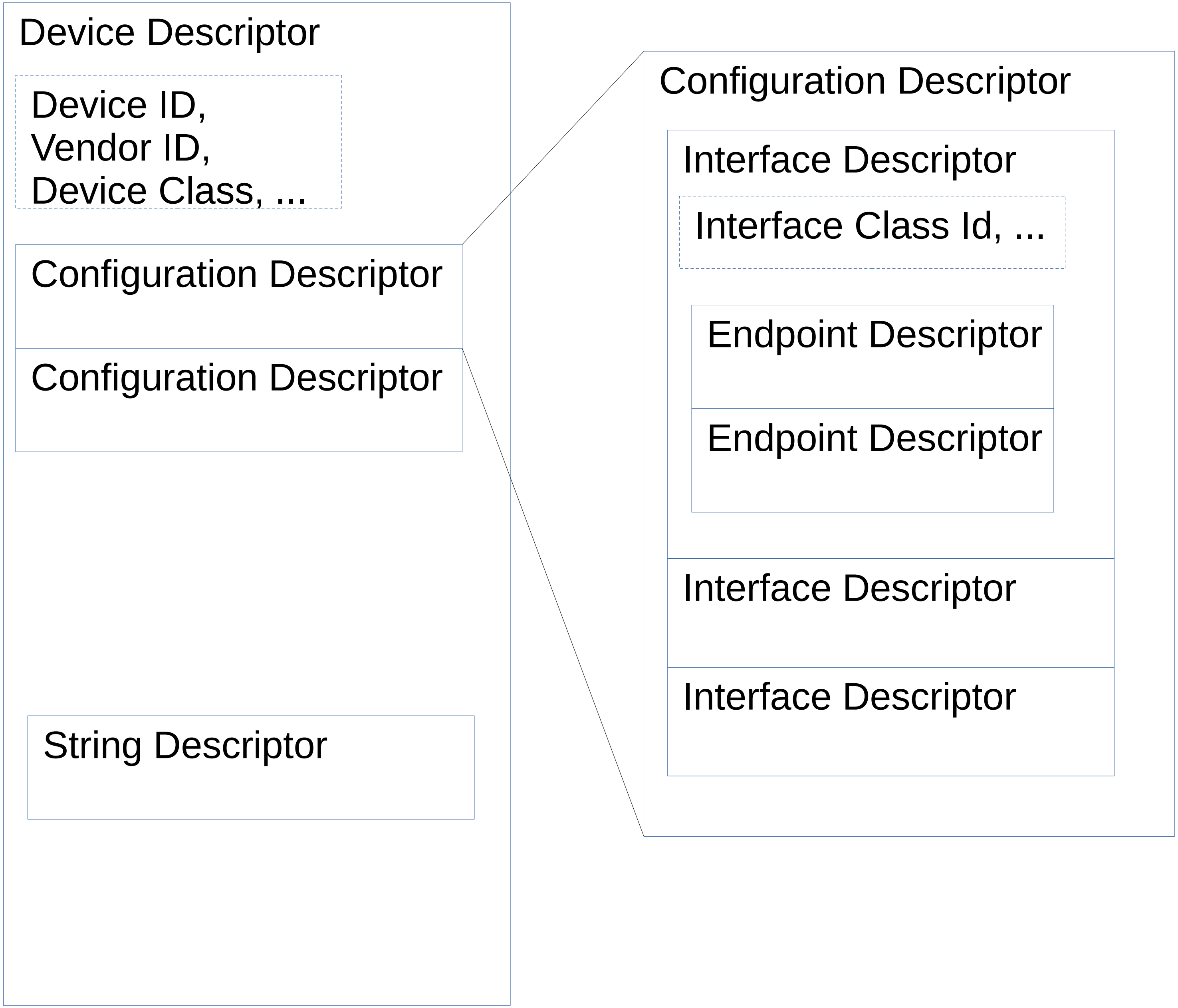
Logical Connection



Composite device



USB descriptors



Device Descriptor

Device ID,
Vendor ID,
Device Class, ...

Configuration Descriptor

Configuration Descriptor

String Descriptor

Configuration Descriptor

Interface Descriptor

Interface Class Id, ...

Endpoint Descriptor

Endpoint Descriptor

Interface Descriptor

Interface Descriptor

Standard Device Descriptor

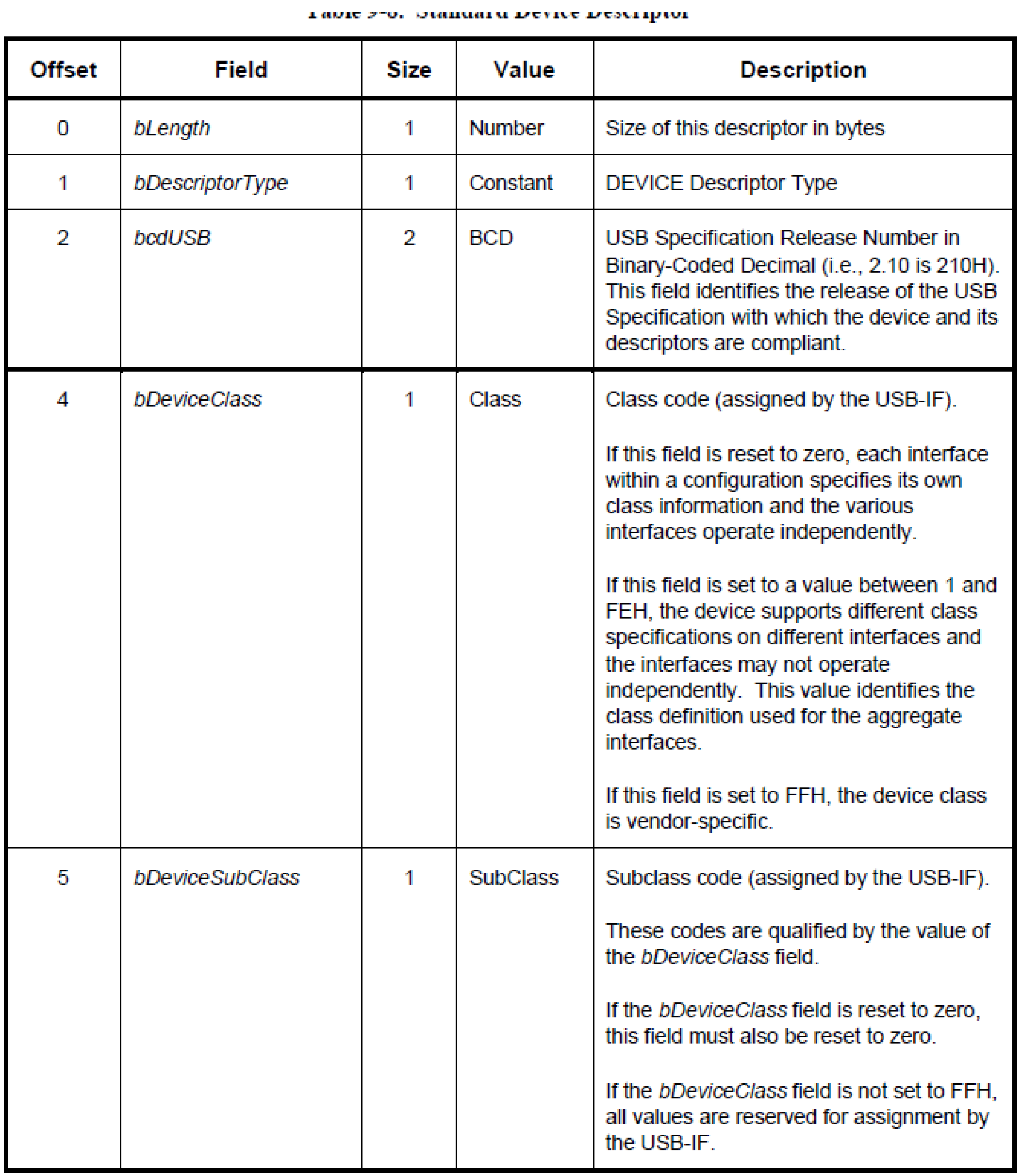
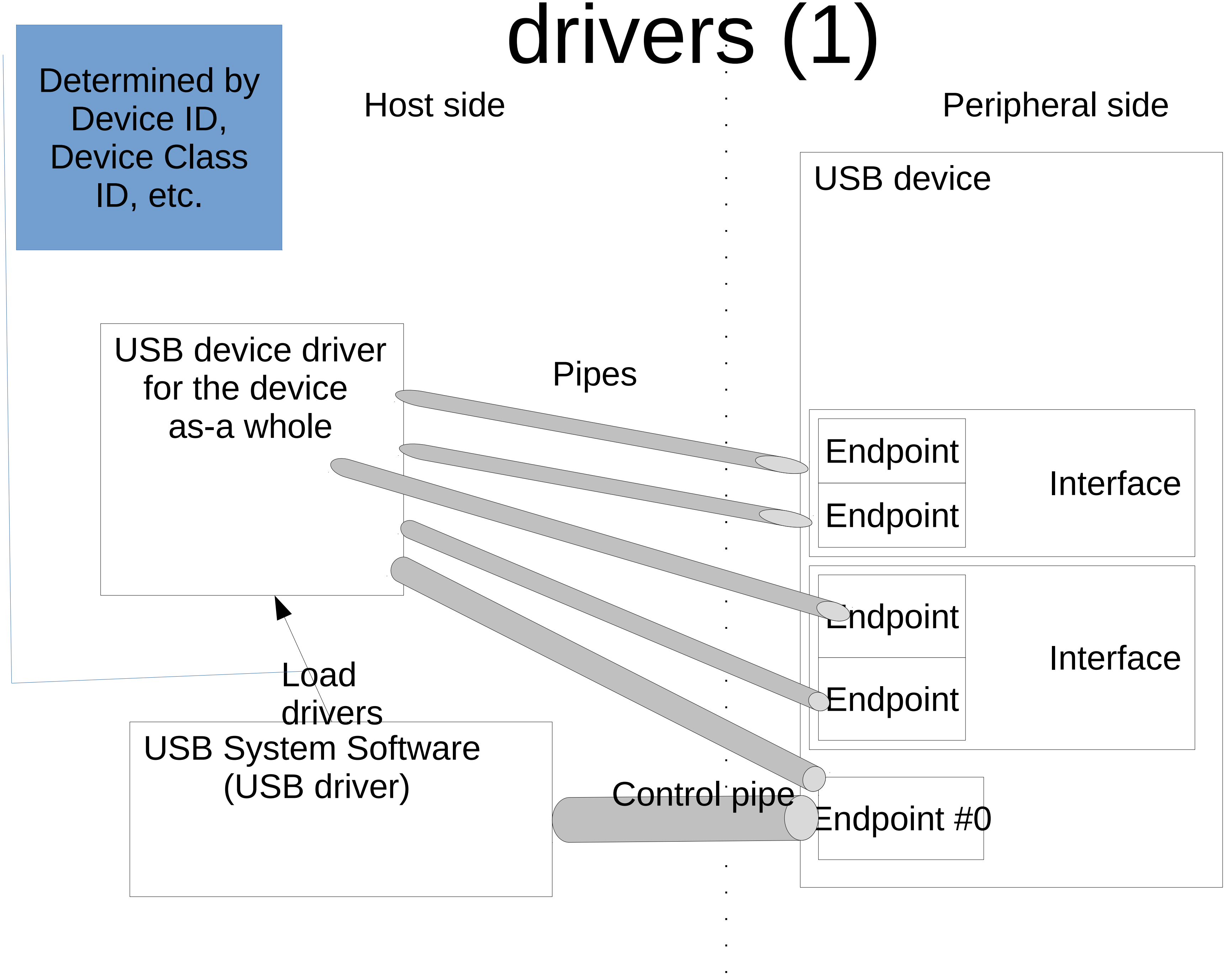


Table 9-8. Standard Device Descriptor

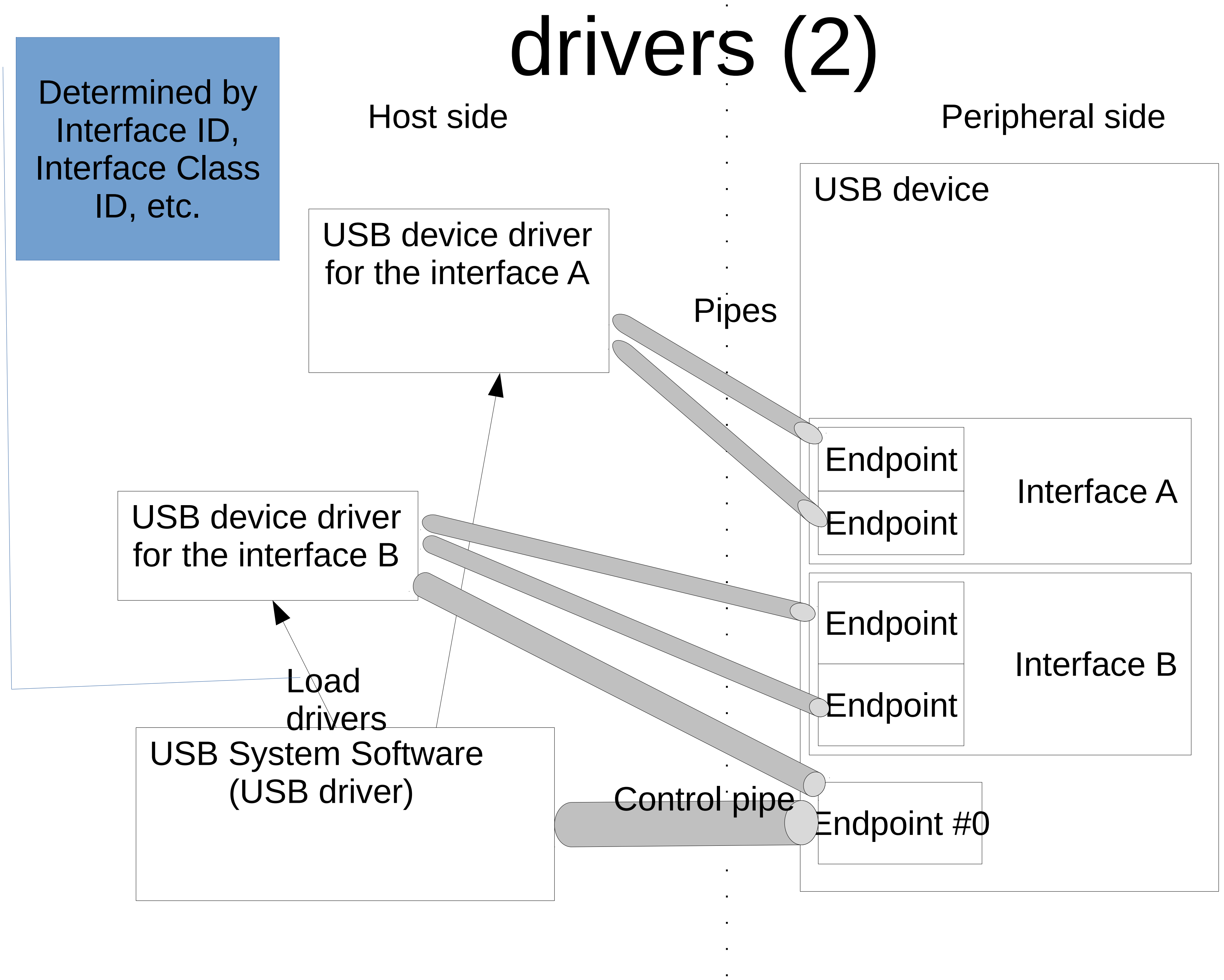
Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	DEVICE Descriptor Type
2	*bcdUSB*	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	*bDeviceClass*	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.

From USB 2.0 specification.

Composite device and host-side drivers (1)



Composite device and host-side drivers (2)


Determined by Interface ID, Interface Class ID, etc.
Host side
Peripheral side
USB device
USB device driver for the interface A
Pipes
Endpoint
Endpoint
Interface A
USB device driver for the interface B
Endpoint
Endpoint
Interface B
Load drivers
USB System Software (USB driver)
Control pipe
Endpoint #0

Design Goal of our Framework

- Common tasks among many USB devices are done in the framework.
  - Let framework users to focus on codes for functionality of interfaces they want to implement.
- The framwork and interface implementation should be independent of the characteristic of USB client controllers.

Design Goal of our Framework

- Interfaces can be implemented either as
  - in-kernel device drivers, or
  - userland programs
-

Design Goal of our Framework

- A USB device implemented using this framework can “transform” into a different USB device without rebooting into an another kernel binary.
  - Interfaces can be attached to/detached from the USB device on-the-fly.
-

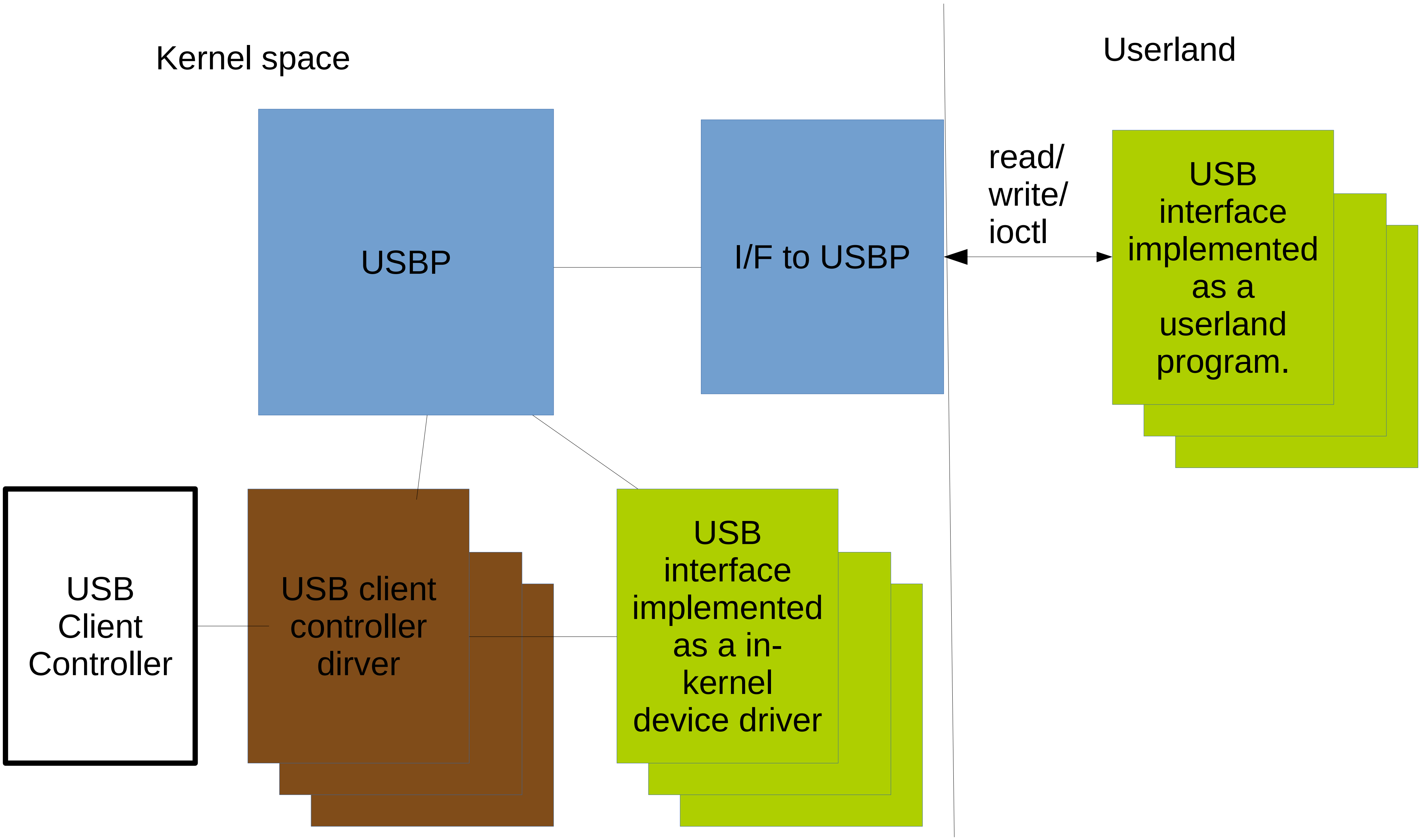
Design Goal of our Framework

- USB Interface implementation can be used to make a simple USB device, or to form a composite device combined with other interface implementations.

components in the framework

- USBP
- Client controller drivers
- USB interface drivers
- Userland interface to USBP.
- Userland programs for USB interfaces.

components in the framework


Kernel space
Userland
USBP
I/F to USBP
read/
write/
ioctl
USB interface implemented as a userland program.
USB Client Controller
USB client controller dirver
USB interface implemented as a in-kernel device driver

components in the framework

- USBP
  - logical driver for peripheral-side USB support.
  - handles USB protocol on control pipe, such as enumeration and configuration.
  - implements functionality of the USB devices
- Client controller drivers
  - there are many kind of client controllers
  - control send/receive of USB packets.
  - USBP and interface drivers access to the controller through a common interface.

components in the framework

- USB interface drivers
  - USB serial, communication device class, mass storage, human interface device, audio, video, etc.
- Userland interface to USBP.
  - in order to implement USB interface functionality as userland programs.
- Userland programs for USB interfaces.

Kernel configuration for peripheral-side support

in config(5)

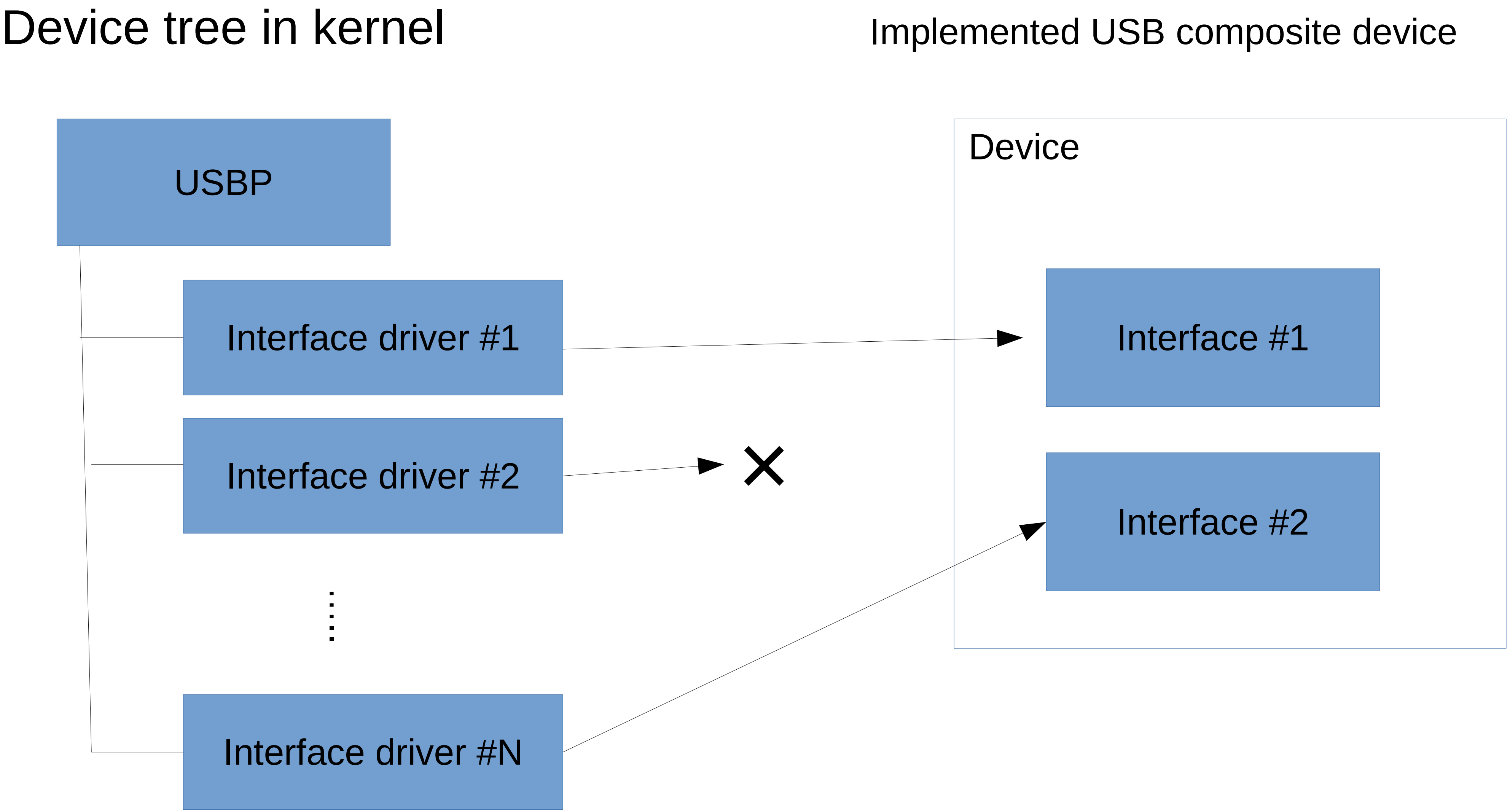
```
pxaudc0 at obio0   # Client controller driver
usbp0 at pxaudc0   # USB Peripheral-side support

cdcef0 at usbp0    # CDC Ethernet model

upftdi0 at usbp0   # FTDI USB serial emulation
ucom* at upftdi?

usbpusr* at usbp?   #  Userland gateway for USBP
```

Multiple interfaces and a composite device



API for USB interface drivers

Note: the API is now being debugged and may be modified in the future.
It is already modified from the version in my paper.

```
usbd_status usbp_add_interface(
    struct usbp_device *device,
    const struct usbp_add_iface_request *request,
    const struct usbp_interface_methods *iface_methods,
    struct usbp_interface  **interface);
```

struct usbp_add_iface_request

```
struct usbp_add_iface_request {
   struct usbp_device_info devinfo;
   struct usbp_interface_spec ispec;
   struct usbp_endpoint_request endpoints[];
};
```

used to build the device descriptor

characteristic of the interface

type and directions of required endpoints

struct usbp_device_info

```
struct usbp_device_info {
    int16_t class_id;      /* can be USBP_ID_UNSPECIFIED */
    uint8_t subclass_id;
    uint8_t protocol;
    int vendor_id;         /* can be USBP_ID_UNSPECIFIED */
    uint16_t product_id;
    uint16_t bcd_device;   /* device release number in BCD */
    const char *manufacturer_name;
    const char *product_name;
    const char *serial;         /* device's serial number */
};
```

- Used to build a device descriptor.
- If class_id is USBP_ID_UNSPECIFIED, the value from other interface or the default value is used.

struct usbp_interface_spec

```
struct usbp_interface_spec {
    uByte class_id;
    uByte subclass_id;
    uByte protocol;
    const char *description;
    enum USBP_PIPE0_USAGE {
        USBP_PIPE0_NOTUSED,
        USBP_PIPE0_SHARED,
        USBP_PIPE0_EXCLUSIVE
    } pipe0_usage;
    uint8_t num_endpoints;
};
```

Used to build an interface descriptor

- Used to build a interface descriptor, and to request endpoints to be used for the interface.

struct usbp_endpoint_request

```
struct usbp_endpoint_request {
   uint8_t dir;   /* UE_DIR_IN or UE_DIR_OUT*/
   uint8_t attributes; /* Transfer type: UE_ISOCHROMOUS,
                                         UE_BULK,
                                         UE_INTERRUPT */
   uint8_t epnum; /* endpoint number. recommended to set 0 */
   bool optional;
   u_int packetsize;
   /* need more for isochronous */
};
```

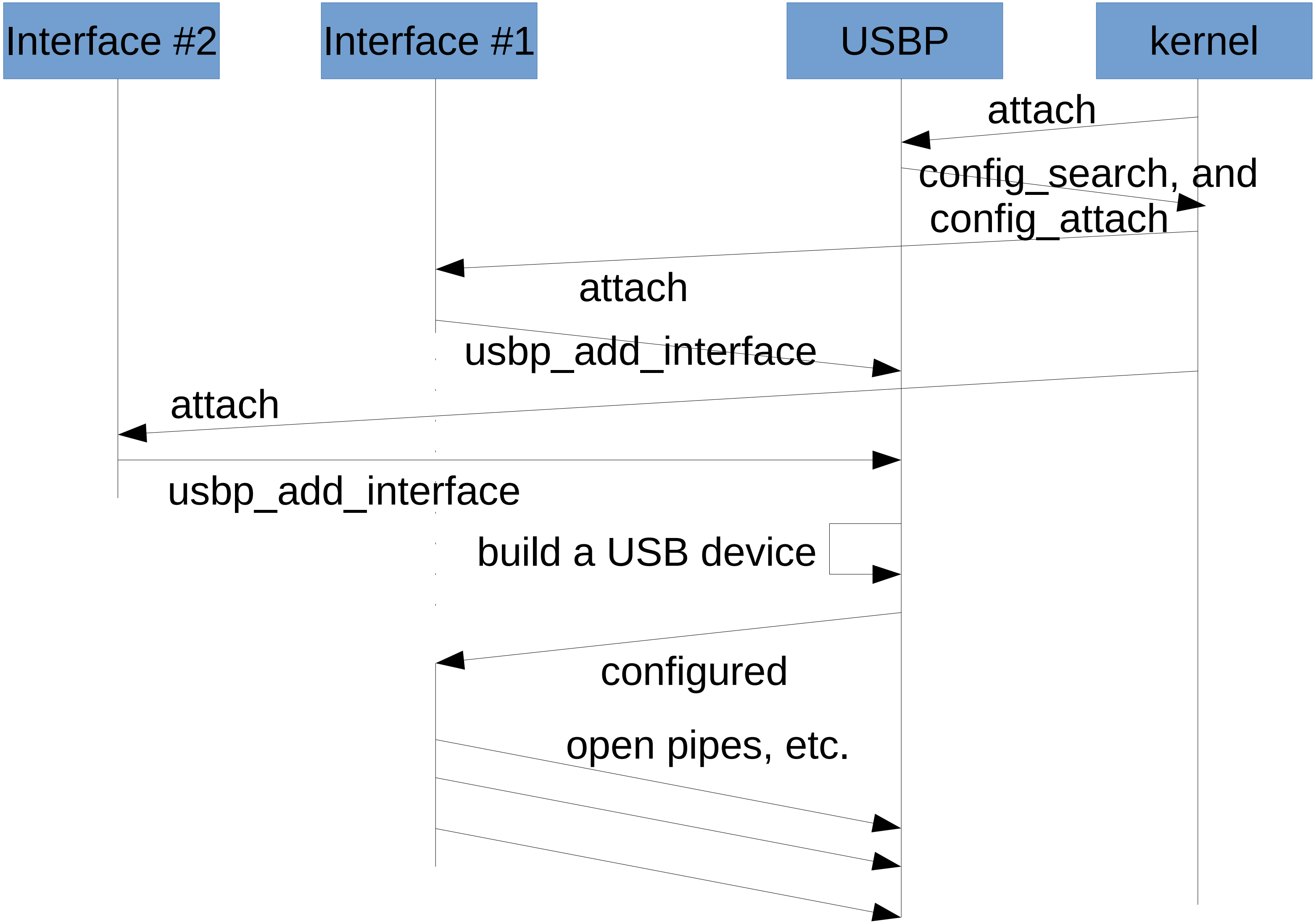
- request an endpoint for a interface.
- if the client controller can not provide the non-optional endpoint, the interface is not used.
- when epnum is 0, platform choose a suitable endpoint. You can explicitly specify endpoint number here, but it is not recommended.

struct usbp_interface_methods

```
struct usbp_interface_methods {
    usbd_status (* configured)(
                    struct usbp_interface *);
    usbd_status (* unconfigured)(
                   struct usbp_interface *);
    usbd_status (* handle_device_request)(
        struct usbp_interface *,
        usb_device_request_t *,
        void **);
    usbd_status (* fixup_idesc)(
        struct usbp_interface *,
        usb_interface_descriptor_t *);
};
```

- callback methods from USBP to interface drivers.
- “configured” method is called when the interface is actually put in the device. In this method, the interface driver starts real task.

example sequence



Other APIs

- usbp_delete_interface
- usbp_get_endpoint
- usbp_open_pipe
- usbd_alloc_xfer
- usbd_alloc_buffer
- usbd_free_xfer
- usbd_setup_xfer
- usbd_get_xfer_status
- usbd_transfer
- usbd_abort_pipe
- ….

Userland gateway

- /dev/usbp0/ctl
- /dev/usbp0/0, /dev/usbp0/1 … for endpoints.
- ioctl(USBP_IOC_ADDIFACE)
- ioctl(USBP_GETEP)
  - blocks until the interface is selected in the device.
- read/write on endpoint nodes.

Current implementation

https://github.com/bsh-git/netbsd-usb-peripheral.git

Comparison

- OpenBD
  - usbf(4)
- Linux
  - Gadget
  - http://www.linux-usb.org/gadget/

Demonstration

- FTDI USB serial adapter
- CDC Ethernet emulation (port of cdcef(4) of O)
- Umass in the userland (not working yet)

Future development

- Clean up the code , get reviewed, commit it into the codebase.
- More client controllers
- More interface drivers
- DMA support
- USB 2.0, 3.0
-

Questions?

Acknowledgement

- I started this project by porting OpenBSD's usbf(4) to NetBSD, which was written by Uwe stühler and other OpenBSD developers.
- Taylor R. Campbell, Greg Oster, and Masanobu Saitoh for reviewing my paper.
- Dan Harris and co-workers at Genetec corp. for good advice in my presentation rehearsal.