Peripheral-side USB support for NetBSD

HIROYUKI BESSHO

Genetec corp.
bsh@NetBSD.org

Abstract

Some hardware platforms that run NetBSD have type-B receptacles and USB client controllers, allowing them to act as USB devices.

Currently, the NetBSD kernel doesn't have generic support for the peripheral side of USB. This paper describes a new framework for NetBSD to help implement functionality as a USB device on those platforms.

1 Introduction

NetBSD has supported USB for many years, but it has been limited to host-side functionality. Many hardware platforms that can run NetBSD have some hardware components to act as USB devices, including:

- USB type-B receptacle. (or, micro-AB receptacle)
- USB client controller, and related external circuits.

These are often found on embedded platforms. The client controller is also named as 'USB device controller' or 'USB peripheral controller'. They are often found as sub-components in SoCs, and there are also specialized LSIs that can be connected to a CPU's local bus.

In order to utilize those hardware components and become a USB device, some software needs to be written. This includes:

- a driver to control USB client controller
- code to handle the USB protocol, for enumeration and configuration
- the implementation of the actual functionality as a USB device

OpenBSD introduced a framework to help implement those software components in 2007 called 'usbf'. Linux also has such a framework called 'Gadgets'.

I have implemented a framework for NetBSD to support the peripheral side of USB, partly derived from OpenBSD's.

2 Terms

The following terms are defined by the USB specification and are included here to help facilitate this discussion:

USB function A USB function transmits or receives data or control information over the bus. Typically, a function is implemented as a separate peripheral device, but a physical package may implement multiple USB functions and an embedded hub with single USB port.

From the viewpoint of system software, a USB function is same as a USB device.

USB interface A USB interface is a related set of endpoints that present a single feature of the USB function to the host.

A USB function can have multiple USB interfaces as shown in Figure 2. All USB interfaces in a USB function share single USB address.

Device endpoint An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device.

Compound device A USB device that has multiple USB functions in it. (Figure 1)

Composite device A USB device that has multiple USB interfaces that are controlled independently of each other.

3 Components in the framework

The framework has the following components:

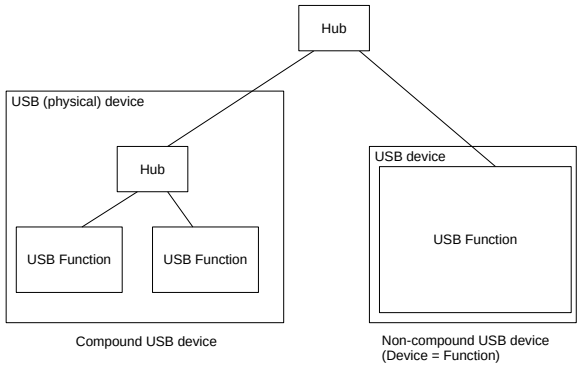

Figure 1: Compound device

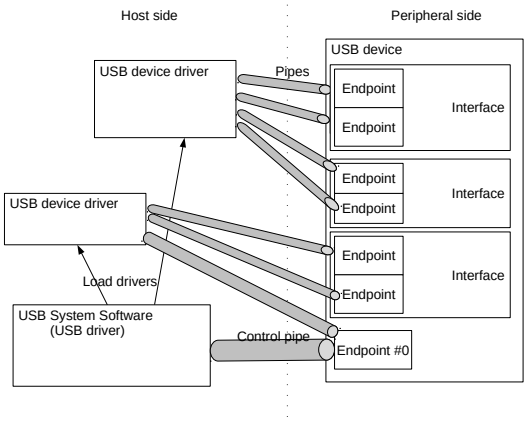

Figure 2: USB Interfaces and endpoints

USBP Driver for the peripheral side of the USB protocol. It handles the USB protocol on the control pipe. This includes, for example, enumeration and configuration.

USB interface drivers Drivers for the peripheral side of USB interfaces. These drivers implement the functionality of USB devices, such as USB serial adapters, communication-class devices, mass storage devices, human interface devices, audio, video, etc.

Client controller drivers Drivers for client controllers. These drivers send USB packets to and receive USB packets from the USB host through the USB client controller. There are many kind of USB client controllers on the market, and they vary in design.

To send/receive USB packets to/from the host, the USBP and USB interface drivers access the controller through a common interface so that they can be independent of the type of controllers.

In the kernel configuration file shown in Figure 3, a `usbp` device is attached to a client controller device, and USB interface drivers are attached to the `usbp` device.

4 USB device and interface

4.1 Choosing the USB device to become

With our framework, the platform can act as any kind of USB device. The kind of USB device can be chosen by attaching USB interface drivers to USBP. In Figure 3, the FTDI emulation interface is attached to USBP. Different interfaces can be attached to USBP as in Figure 4, and in that case, the platform provides Ethernet emulation over USB to the host.

4.2 Composite device

When two or more USB interface drivers are attached to USBP, it configures the device as a composite device as defined in the USB specification. The kernel configuration in Figure 5 will create a composite device that consists of USB mass storage and Ethernet emulation.

4.3 Limitation on multiple USB interfaces

Although any number of USB interface drivers can be attached to USBP, it has some limitations on building a multi-interface device. There are two classes of limitations:

- Limitations related to endpoints
- Limitations set by non-cooperative USB interfaces

4.3.1 Endpoint limitations

The USB specification defines that USB devices can have up to 16 endpoints. The USB host connects a pipe to a device's endpoint and use it for communication flow between the host and the device. Among the 16 endpoints, endpoint #0 is special and used for USB device configuration. A pipe to an endpoint #0 is called a control pipe. Some USB devices use control pipes for normal data transactions.

As USB pipes cannot be shared among interfaces[1], the number of total endpoints used by all interfaces in the device must be less than 15 excluding the control pipe. Also note that pipes are unidirectional.

[1] except for control pipes

```
pxaudc0 at obio0          # USB client controller on PXA250
usbp0 at pxaudc0          # peripheral-side USB support
upftdi* at usbp0          # emulates FTDI USB serial adapter
```

Figure 3: Example device tree in kernel configuration

```
cdcef* at usbp0           # CDC ECM
```

Figure 4: Another kernel configuration example

In addition, even if the USB specification allows 16 endpoints per device, USB client controllers may not have all of them implemented. Some controllers provide fewer than 16 endpoints.

Some USB client controllers set specific purposes to each endpoint. For example, PXA250's[2] client controller dedicates endpoints #1, 6, 11 to bulk IN pipes, and #4, 9, 14 to isochronous OUT pipes (see Table 1). So if interface drivers require 4 bulk IN pipes total, they can not be configured in one device on the platforms with PXA250, even though the total number of endpoints required by USB interfaces is below 16.

transfer type	direction	endpoints
control		0
bulk	IN	1, 6, 11
bulk	OUT	2, 7, 12
isochronous	IN	3, 8, 13
isochronous	OUT	4, 9, 14
interrupt	IN	5, 10, 15

Table 1: Endpoints of PXA250 client controller

4.3.2 Cooperative USB interfaces

'Cooperative USB interface' is not a term defined in the USB specification. It is introduced here for this discussion.

USB interfaces are cooperative when they:

- are identified by class codes in interface descriptors, and
- don't require exclusive use of the control pipe (pipe #0) for them.

Some USB devices, such as USB serial devices use bulk pipes for data transfer, and a control pipe for changing their mode. Thus, they are 'non-cooperative'.

USB devices that have vendor-specific class codes and are identified by vendor ID and product ID are non-cooperative. They require a specific host-side driver.

[2] ARM based SoC by Marvell, formerly Intel

Also, USB devices identified by class code in their device descriptors are non-cooperative either.

Any number of cooperative interfaces can form a composite device as long as they satisfy the endpoint limitations.

Two or more non-cooperative interfaces cannot co-exist in a single USB device, because they insist on acting as an entire USB device rather than as a single interface in a device.

One non-cooperative USB interface and some cooperative USB interfaces may be able to co-exist in one USB device, but it may require special handling in a host-side driver.

4.3.3 Attaching many USB interfaces to USBP

Any number of USB interfaces can be attached to USBP. Because of the limitations discussed above, however, USBP may not build up a USB device consisting of all USB interfaces attached to it. USBP selects as many USB interfaces as possible to construct a USB device, and ignores the remaining interfaces.

Thus, priorities among attached interfaces need to be set. In the current implementation, an interface attached later has higher priority.

5 Transform

The functionality of the USB device implemented using USBP framework is defined by USB interface drivers attached to USBP. That can be changed by attaching/detaching USB interfaces while the peripheral-side platform is running. In other words, that USB device can transform to a different device without rebooting the kernel with a different configuration.

A USB device can transform by:

- loading/unloading kernel modules for USB interface drivers.
- attaching/detaching USB interface drivers to USBP via drvctl

```
usbp0 at pxaudc? # peripheral-side USB support
upmass* at usbp0 # mass storage
cdcef* at usbp0 # CDC ECM
```

Figure 5: Composite device

5.1 Example use-case

1. The kernel starts with USBP and DFU (Device Firmware Upgrade).
   - Host sees DFU device.
2. Kernel detaches the USB interface driver.
   - As no active USB interface is attached, USBP turns off the pull-up register on the USB data line.
   - Host notices that the USB device is disconnected.
3. Kernel attaches umass interface driver to USBP.
   - USBP configures a new USB device, then turns on the pull-up.
   - Host sees an USB device connected, and finds it is a mass storage device.

   In this example, the USB device first appeared as DFU, and then transforms into mass storage device.

6 USB interfaces in userland

In addition to USB interfaces implemented by in-kernel device drivers, the framework also supports USB interfaces implemented by userland processes.

Userland programs implement the USB interface by following steps:

1. Open /dev/usbp*N*
2. Issue ioctls to give necessary information about the USB interfaces to USBP, and build up descriptors for the interface.
3. Assign endpoints using ioctl.
4. Open /dev/usbp*N*.*MM* for endpoint access, where *MM* is an endpoint number.
5. Read/write endpoints to communicate with the host.

When /dev/usbp*N* is closed by the process, the USB interfaces built are removed from the USB device.

7 USBP's interface for USB interface drivers

In this section, we list some of API functions provided by USBP. The API may change as the implementation of the framework is now under testing and debugging.

7.1 `usbp_add_interface`

A USB interface driver calls `usbp_add_interface` shown in Figure 7 to add its interface to the USB device with the following parameters:

- information to build a device descriptor (struct usbp_device_info).

  This information may be used when the USB interface is actually configured into the USB device. For a composite device, which has multiple USB interfaces, the device information from the interface with the highest priority is used.
- information about the interface. (struct usbp_interface_spec)

  This is used to build the USB interface descriptor. Types of endpoints required by the interface are also included.
- callback functions called by USBP.

7.2 `usbp_delete_interface`

A USB interface can be removed from the USB device by calling `usbp_delete_interface` shown in Figure 6.

USBP re-calculates USB descriptors and notify the host if needed.

7.3 String descriptor

The USB interface driver passes some strings to be used as human readable descriptions of the device via `usbp_add_interface`.

These strings are presented to the host through USB string descriptors. USBP manages strings provided by USB interface drivers, assigns string IDs, and send them on the host's request.

Conversion to UTF-16 is also done by USBP.

```
usbd_status usbp_delete_interface(struct usbp_device *,
                                  struct usbp_interface *);
```

Figure 6: API function `usbp_delete_interface`

8 Differences from OpenBSD implementation

I started this project by porting OpenBSD's `usbf`(4) driver, and later modified it significantly. The reasons for the significant changes were to:

1. support composite devices and add an ability to transform,
2. handle endpoint limitations imposed by client controllers, and
3. share source code between host-side and peripheral side of USB.

8.1 Sharing codes with host-side

In OpenBSD's implementation, usbf(4) device consists of many objects such as `usbf_bus`, `usbf_endpoint`, `usbf_pipe`, `usbf_xfer`, etc., and they are very similar to objects found in host-side implementation such as `usbd_bus`, `usbd_endpoint`, `usbd_pipe`, `usbd_xfer`, and so on. Figure 8 illustrates the OpenBSD's implementation. It is natural for us to want to reuse code for the host side.

Sharing host-side source code was also necessary in order to make logical drivers for USB devices run on both sides of USB. One example of such logical drivers is ucom(4). The ucom(4) driver provides common functionality for USB serial adapters, and it is designed to work with `usbd_*` objects defined in the host-side USB subsystem. So it was impossible for OpenBSD's implementation to let ucom(4) run on peripheral-side of USB.

In our implementation, we add super-class objects for both side of USB as shown in Figure 10. This made ucom(4) usable for both of host side and peripheral side.

9 Further development

9.1 Client controllers

Only one client controller is supported so far:

- Client controller in PXA250

More controllers should be supported, including ones in ARM based SoCs such as i.MX[3] and OMAP[4].

[3] ARM based SoC by Freescale

[4] Texas Instruments

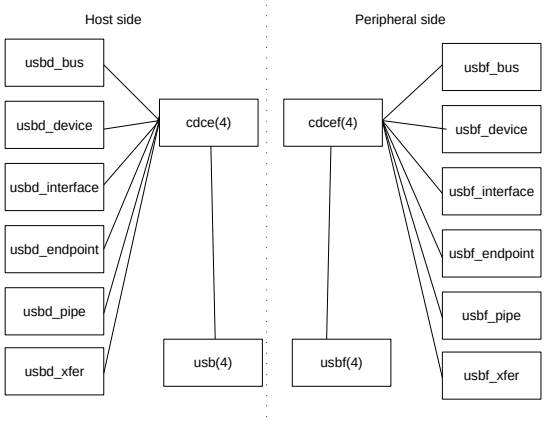

Figure 8: USB related objects in OpenBSD

9.2 USB Interfacess

Currently, the following USB interfaces have been implemented:

cdcef(4) CDC Ethernet emulation.

upftdi(4) emulates FTDI USB serial.

More USB interfaces should be implemented. The current targets are mass storage and DFU.

9.3 USB On-The-Go

USB OTG is a supplemental specification of USB that allows platforms become both host and device sharing one USB Micro-AB receptacle.

Many modern platforms have micro-AB receptacles. When two platforms with those receptacles are connected, they need to decide to take either role—host or peripheral—according to the OTG specification.

10 On going study

10.1 Comparison to Linux's Gadget

Linux has a framework for peripheral-side USB named as 'Gadget'. I haven't looked at it closely yet.

Comparison to Gadget will help improve our framework.

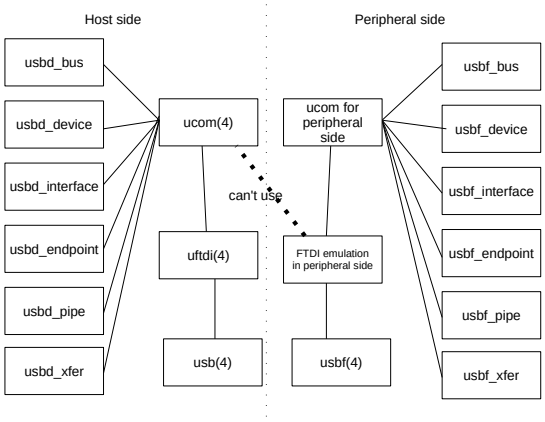

Figure 9: Difficult to use ucom(4) for both side

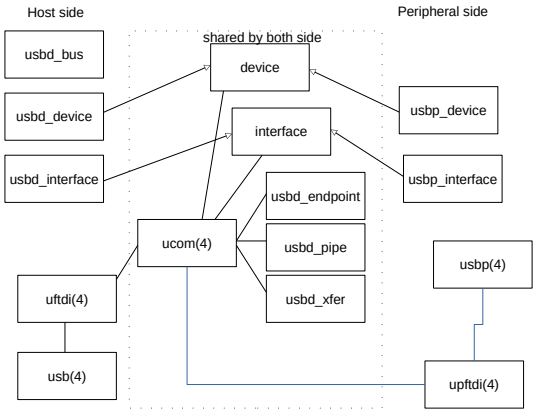

Figure 10: Sharing USB related objects and ucom(4)

11 Conclusion

This paper presented a new framework for NetBSD to support peripheral-side USB. With this framework, we can make platforms with NetBSD run as USB devices, with fairly small amount of new source code. This will be helpful for developing embedded systems.

Acknowledgment

I started this project by porting OpenBSD's `usbf`(4) and `cdcef`(4) drivers to NetBSD. They were initially written by Uwe Stühler.

I would like to thank Taylor R. Campbell, Greg Oster, and Masanobu Saitoh for reviewing this paper, suggesting improvements, and fixing many grammatical errors. If you still find broken English in this paper, those sentences were added by me after their review.

References

[1] USB Implementers Forum, Inc. (2000). Universal Serial Bus Revision 2.0 Specification

[2] USB Implementers Forum, Inc. (2010). Universal Serial Bus Class Definitions for Communications Devices, Revision 1.2 (Errata 1)

[3] USB Implementers Forum, Inc. (2007). Universal Serial Bus Micro-USB Cables and Connectors Specification Revision 1.01

```
/*
 * information used to build USB device descriptor.
 */
struct usbp_device_info {
        int16_t class_id;
        int16_t subclass_id;
        int protocol;
        int vendor_id;
        int product_id;
        int bcd_device;         /* device release number in BCD */
        const char *manufacturer_name;
        const char *product_name;
        const char *serial;     /* device's serial number */
};

#define USBP_ID_UNSPECIFIC      (-1)

/*
 * requirements of an endpoint used by the interface
 */
struct usbp_endpoint_request {
        uint8_t direction;      /* UE_DIR_IN or UE_DIR_OUT */
        uint8_t attributes;     /* Transfer type:
                                   UE_ISOCHROMOUS, UE_BULK, UE_INTERRUPT */
        u_int packetsize;
        /* need more for isochronous */
};

/*
 * information used to build an interface descriptor
 */
struct usbp_interface_spec {
        uint8_t class_id;
        uint8_t subclass_id;
        uint8_t protocol;
        enum USBP_PIPE0_USAGE {
                USBP_PIPE0_NOTUSED, /* this interface doesn't use pipe #0 */
                USBP_PIPE0_SHARED,  /* pipe#0 is shared among interfaces,
                                       by means of interface number in the packets */
                USBP_PIPE0_EXCLUSIVE /* this interface requires an
                                        exclusive use of pipe#0 */
        } pipe0_usage;
        const char *description;
        uint8_t num_endpoints;  /* the number of endpoints used by
                                   this interface excluding ep0. */
        struct usbp_endpoint_request endpoints[];
};

usbd_status usbp_add_interface(
    struct usbp_device *,           /* USB device managed by USBP */
    struct usbp_interface *,          /* USB interface to add */
    const struct usbp_device_info *,  /* information to build a device descriptor */
    const struct usbp_interface_spec *, /* information to build an interface
                                           descriptor and endpoint descriptors */
    const struct usbp_interface_methods *);  /* callback functions */
```

Figure 7: API function `usbp_add_interface`