The Linux-USB Host Side API

Introduction to USB on Linux

A Universal Serial Bus (USB) is used to connect a host, such as a PC or workstation, to a number of peripheral devices. USB uses a tree structure, with the host as the root (the system's master), hubs as interior nodes, and peripherals as leaves (and slaves). Modern PCs support several such trees of USB devices, usually a few USB 3.0 (5 GBit/s) or USB 3.1 (10 GBit/s) and some legacy USB 2.0 (480 MBit/s) busses just in case.

That master/slave asymmetry was designed-in for a number of reasons, one being ease of use. It is not physically possible to mistake upstream and downstream or it does not matter with a type C plug (or they are built into the peripheral). Also, the host software doesn't need to deal with distributed auto-configuration since the pre-designated master node manages all that.

Kernel developers added USB support to Linux early in the 2.2 kernel series and have been developing it further since then. Besides support for each new generation of USB, various host controllers gained support, new drivers for peripherals have been added and advanced features for latency measurement and improved power management introduced.

Linux can run inside USB devices as well as on the hosts that control the devices. But USB device drivers running inside those peripherals don't do the same things as the ones running inside hosts, so they've been given a different name: *gadget drivers*. This document does not cover gadget drivers.

USB Host-Side API Model

Host-side drivers for USB devices talk to the "usbcore" APIs. There are two. One is intended for *general-purpose* drivers (exposed through driver frameworks), and the other is for drivers that are *part of the core*. Such core drivers include the *hub* driver (which manages trees of USB devices) and several different kinds of *host controller drivers*, which control individual busses.

The device model seen by USB drivers is relatively complex.

- USB supports four kinds of data transfers (control, bulk, interrupt, and isochronous). Two of them (control and bulk) use bandwidth as it's available, while the other two (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.
- The device description model includes one or more "configurations" per device, only one of which is active at a time. Devices are supposed to be capable of operating at lower than their top speeds and may provide a BOS descriptor showing the lowest speed they remain fully operational at.
- From USB 3.0 on configurations have one or more "functions", which provide a common functionality and are grouped together for purposes of power management.
- Configurations or functions have one or more "interfaces", each of which may have "alternate settings". Interfaces may be standardized by USB "Class" specifications, or may be specific to a vendor or device.

  USB device drivers actually bind to interfaces, not devices. Think of them as "interface drivers", though you may not see many devices where the distinction is important. *Most USB devices are simple, with only one function, one configuration, one interface, and one alternate setting.*
- Interfaces have one or more "endpoints", each of which supports one type and direction of data transfer such as "bulk out" or "interrupt in". The entire configuration may have up to sixteen endpoints in each direction, allocated as needed among all the interfaces.
- Data transfer on USB is packetized; each endpoint has a maximum packet size. Drivers must often be aware of conventions such as flagging the end of bulk transfers using "short" (including zero length) packets.
- The Linux USB API supports synchronous calls for control and bulk messages. It also supports asynchronous calls for all kinds of data transfer, using request structures called "URBs" (USB Request Blocks).

Accordingly, the USB Core API exposed to device drivers covers quite a lot of territory. You'll probably need to consult the USB 3.0 specification, available online from www.usb.org at no cost, as well as class or device specifications.

The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the HCDs. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true, but there are still differences that crop up especially with fault handling on the less common controllers. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like unlinking an URB) isn't yet fully consistent. Device driver authors should make a point of doing disconnect testing (while the device is active) with each different host controller driver, to make sure drivers don't have bugs of their own as well as to make sure they aren't relying on some HCD-specific behavior.

USB-Standard Types

In `include/uapi/linux/usb/ch9.h` you will find the USB data types defined in chapter 9 of the USB specification. These data types are used throughout USB, and in APIs including this host side API, gadget APIs, usb character devices and debugfs interfaces. That file is itself included by `include/linux/usb/ch9.h`, which also contains declarations of a few utility routines for manipulating

these data types; the implementations are in `drivers/usb/common/common.c`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 120)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/common/common.c
   :export:
```

In addition, some functions useful for creating debugging output are defined in `drivers/usb/common/debug.c`.

Host-Side Data Types and Macros

The host side API exposes several layers to drivers, some of which are more necessary than others. These support lifecycle models for host side drivers and devices, and support passing buffers through usbcore to some HCD that performs the I/O for the device driver.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 136)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/usb.h
   :internal:
```

USB Core APIs

There are two basic I/O models in the USB API. The most elemental one is asynchronous: drivers submit requests in the form of an URB, and the URB's completion callback handles the next step. All USB transfer types support that model, although there are special cases for control URBs (which always have setup and status stages, but may not have a data stage) and isochronous URBs (which allow large packets and include per-packet fault reports). Built on top of that is synchronous API support, where a driver calls a routine that allocates one or more URBs, submits them, and waits until they complete. There are synchronous wrappers for single-buffer control and bulk transfers (which are awkward to use in some driver disconnect scenarios), and for scatterlist based streaming i/o (bulk or interrupt).

USB drivers need to provide buffers that can be used for DMA, although they don't necessarily need to provide the DMA mapping themselves. There are APIs to use used when allocating DMA buffers, which can prevent use of bounce buffers on some systems. In some cases, drivers may be able to rely on 64bit DMA to eliminate another kind of bounce buffer.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 161)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/urb.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 164)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/message.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 167)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/file.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 170)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/driver.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 173)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/usb.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 176)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/hub.c
   :export:
```

Host Controller APIs

These APIs are only for use by host controller drivers, most of which implement standard register interfaces such as XHCI, EHCI, OHCI, or UHCI. UHCI was one of the first interfaces, designed by Intel and also used by VIA; it doesn't do much in hardware. OHCI was designed later, to have the hardware do more work (bigger transfers, tracking protocol state, and so on). EHCI was designed with USB 2.0; its design has features that resemble OHCI (hardware does much more work) as well as UHCI (some parts of ISO support, TD list processing). XHCI was designed with USB 3.0. It continues to shift support for functionality into hardware.

There are host controllers other than the "big three", although most PCI based controllers (and a few non-PCI based ones) use one of those interfaces. Not all host controllers use DMA; some use PIO, and there is also a simulator and a virtual host controller to pipe USB over the network.

The same basic APIs are available to drivers for all those controllers. For historical reasons they are in two layers: :c:type:`struct usb_bus <usb_bus>` is a rather thin layer that became available in the 2.2 kernels, while :c:type:`struct usb_hcd <usb_hcd>` is a more featureful layer that lets HCDs share common code, to shrink driver size and significantly reduce hcd-specific behaviors.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 197); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 197); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 205)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/hcd.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 208)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/hcd-pci.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst, line 211)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/core/buffer.c
   :internal:
```

The USB character device nodes

This chapter presents the Linux character device nodes. You may prefer to avoid writing new kernel code for your USB driver. User mode device drivers are usually packaged as applications or libraries, and may use character devices through some programming library that wraps it. Such libraries include:

- libusb for C/C++, and
- jUSB for Java.

Some old information about it can be seen at the "USB Device Filesystem" section of the USB Guide. The latest copy of the USB Guide can be found at http://www.linux-usb.org/

Note

- They were used to be implemented via *usbfs*, but this is not part of the sysfs debug interface.
    - This particular documentation is incomplete, especially with respect to the asynchronous mode. As of kernel 2.5.66 the code and this (new) documentation need to be cross-reviewed.

What files are in "devtmpfs"?

Conventionally mounted at `/dev/bus/usb/`, usbfs features include:

- `/dev/bus/usb/BBB/DDD` ... magic files exposing the each device's configuration descriptors, and supporting a series of ioctls for making device requests, including I/O to devices. (Purely for access by programs.)

Each bus is given a number (`BBB`) based on when it was enumerated; within each bus, each device is given a similar number (`DDD`). Those `BBB/DDD` paths are not "stable" identifiers; expect them to change even if you always leave the devices plugged in to the same hub port. *Don't even think of saving these in application configuration files.* Stable identifiers are available, for user mode applications that want to use them. HID and networking devices expose these stable IDs, so that for example you can be sure that you told the right UPS to power down its second server. Pleast note that it doesn't (yet) expose those IDs.

/dev/bus/usb/BBB/DDD

Use these files in one of these basic ways:

- *They can be read,* producing first the device descriptor (18 bytes) and then the descriptors for the current configuration. See the USB 2.0 spec for details about those binary data formats. You'll need to convert most multibyte values from little endian format to your native host byte order, although a few of the fields in the device descriptor (both of the BCD-encoded fields, and the vendor and product IDs) will be byteswapped for you. Note that configuration descriptors include descriptors for interfaces, altsettings, endpoints, and maybe additional class descriptors.
- *Perform USB operations* using *ioctl()* requests to make endpoint I/O requests (synchronously or asynchronously) or manage the device. These requests need the `CAP_SYS_RAWIO` capability, as well as filesystem access permissions. Only one ioctl request can be made on one of these device files at a time. This means that if you are synchronously reading an endpoint from one thread, you won't be able to write to a different endpoint from another thread until the read completes. This works for *half duplex* protocols, but otherwise you'd use asynchronous i/o requests.

Each connected USB device has one file. The `BBB` indicates the bus number. The `DDD` indicates the device address on that bus. Both

of these numbers are assigned sequentially, and can be reused, so you can't rely on them for stable access to devices. For example, it's relatively common for devices to re-enumerate while they are still connected (perhaps someone jostled their power supply, hub, or USB cable), so a device might be `002/027` when you first connect it and `002/048` sometime later.

These files can be read as binary data. The binary data consists of first the device descriptor, then the descriptors for each configuration of the device. Multi-byte fields in the device descriptor are converted to host endianness by the kernel. The configuration descriptors are in bus endian format! The configuration descriptor are wTotalLength bytes apart. If a device returns less configuration descriptor data than indicated by wTotalLength there will be a hole in the file for the missing bytes. This information is also shown in text form by the `/sys/kernel/debug/usb/devices` file, described later.

These files may also be used to write user-level drivers for the USB devices. You would open the `/dev/bus/usb/BBB/DDD` file read/write, read its descriptors to make sure it's the device you expect, and then bind to an interface (or perhaps several) using an ioctl call. You would issue more ioctls to the device to communicate to it using control, bulk, or other kinds of USB transfers. The IOCTLs are listed in the `<linux/usbdevice_fs.h>` file, and at this writing the source code (`linux/drivers/usb/core/devio.c`) is the primary reference for how to access devices through those files.

Note that since by default these `BBB/DDD` files are writable only by root, only root can write such user mode drivers. You can selectively grant read/write permissions to other users by using `chmod`. Also, usbfs mount options such as `devmode=0666` may be helpful.

Life Cycle of User Mode Drivers

Such a driver first needs to find a device file for a device it knows how to handle. Maybe it was told about it because a `/sbin/hotplug` event handling agent chose that driver to handle the new device. Or maybe it's an application that scans all the `/dev/bus/usb` device files, and ignores most devices. In either case, it should :c:func:`read()` all the descriptors from the device file, and check them against what it knows how to handle. It might just reject everything except a particular vendor and product ID, or need a more complex policy.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst`, line 322); *backlink***
>
> Unknown interpreted text role "c:func".

Never assume there will only be one such device on the system at a time! If your code can't handle more than one device at a time, at least detect when there's more than one, and have your users choose which device to use.

Once your user mode driver knows what device to use, it interacts with it in either of two styles. The simple style is to make only control requests; some devices don't need more complex interactions than those. (An example might be software using vendor-specific control requests for some initialization or configuration tasks, with a kernel driver for the rest.)

More likely, you need a more complex style driver: one using non-control endpoints, reading or writing data and claiming exclusive use of an interface. *Bulk* transfers are easiest to use, but only their sibling *interrupt* transfers work with low speed devices. Both interrupt and *isochronous* transfers offer service guarantees because their bandwidth is reserved. Such "periodic" transfers are awkward to use through usbfs, unless you're using the asynchronous calls. However, interrupt transfers can also be used in a synchronous "one shot" style.

Your user-mode driver should never need to worry about cleaning up request state when the device is disconnected, although it should close its open file descriptors as soon as it starts seeing the ENODEV errors.

The ioctl() Requests

To use these ioctls, you need to include the following headers in your userspace program:

```
#include <linux/usb.h>
#include <linux/usbdevice_fs.h>
#include <asm/byteorder.h>
```

The standard USB device model requests, from "Chapter 9" of the USB 2.0 specification, are automatically included from the `<linux/usb/ch9.h>` header.

Unless noted otherwise, the ioctl requests described here will update the modification time on the usbfs file to which they are applied (unless they fail). A return of zero indicates success; otherwise, a standard USB error code is returned (These are documented in :ref:`usb-error-codes`).

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst`, line 370); *backlink***
>
> Unknown interpreted text role "ref".

Each of these files multiplexes access to several I/O streams, one per endpoint. Each device has one control endpoint (endpoint zero)

which supports a limited RPC style RPC access. Devices are configured by hub_wq (in the kernel) setting a device-wide *configuration* that affects things like power consumption and basic functionality. The endpoints are part of USB *interfaces*, which may have *altsettings* affecting things like which endpoints are available. Many devices only have a single configuration and interface, so drivers for them will ignore configurations and altsettings.

Management/Status Requests

A number of usbfs requests don't deal very directly with device I/O. They mostly relate to device management and status. These are all synchronous requests.

USBDEVFS_CLAIMINTERFACE

This is used to force usbfs to claim a specific interface, which has not previously been claimed by usbfs or any other kernel driver. The ioctl parameter is an integer holding the number of the interface (bInterfaceNumber from descriptor).

Note that if your driver doesn't claim an interface before trying to use one of its endpoints, and no other driver has bound to it, then the interface is automatically claimed by usbfs.

This claim will be released by a RELEASEINTERFACE ioctl, or by closing the file descriptor. File modification time is not updated by this request.

USBDEVFS_CONNECTINFO

Says whether the device is lowspeed. The ioctl parameter points to a structure like this:

```
struct usbdevfs_connectinfo {
        unsigned int   devnum;
        unsigned char  slow;
};
```

File modification time is not updated by this request.

You can't tell whether a "not slow" device is connected at high speed (480 MBit/sec) or just full speed (12 MBit/sec). You should know the devnum value already, it's the DDD value of the device file name.

USBDEVFS_GETDRIVER

Returns the name of the kernel driver bound to a given interface (a string). Parameter is a pointer to this structure, which is modified:

```
struct usbdevfs_getdriver {
        unsigned int  interface;
        char          driver[USBDEVFS_MAXDRIVERNAME + 1];
};
```

File modification time is not updated by this request.

USBDEVFS_IOCTL

Passes a request from userspace through to a kernel driver that has an ioctl entry in the *struct usb_driver* it registered:

```
struct usbdevfs_ioctl {
        int     ifno;
        int     ioctl_code;
        void    *data;
};

/* user mode call looks like this.
 * 'request' becomes the driver->ioctl() 'code' parameter.
 * the size of 'param' is encoded in 'request', and that data
 * is copied to or from the driver->ioctl() 'buf' parameter.
 */
static int
usbdev_ioctl (int fd, int ifno, unsigned request, void *param)
{
        struct usbdevfs_ioctl   wrapper;

        wrapper.ifno = ifno;
        wrapper.ioctl_code = request;
        wrapper.data = param;

        return ioctl (fd, USBDEVFS_IOCTL, &wrapper);
}
```

File modification time is not updated by this request.

This request lets kernel drivers talk to user mode code through filesystem operations even when they don't create a character or block special device. It's also been used to do things like ask devices what device special file should be used. Two pre-defined ioctls are used to disconnect and reconnect kernel drivers, so that user mode code can completely manage binding and configuration of devices.

USBDEVFS_RELEASEINTERFACE

This is used to release the claim usbfs made on interface, either implicitly or because of a USBDEVFS_CLAIMINTERFACE call, before the file descriptor is closed. The ioctl parameter is an integer holding the number of the interface (bInterfaceNumber from descriptor); File modification time is not updated by this request.

> **Warning**
>
> *No security check is made to ensure that the task which made the claim is the one which is releasing it. This means that user mode driver may interfere other ones.*

USBDEVFS_RESETEP

Resets the data toggle value for an endpoint (bulk or interrupt) to DATA0. The ioctl parameter is an integer endpoint number (1 to 15, as identified in the endpoint descriptor), with USB_DIR_IN added if the device's endpoint sends data to the host.

> **Warning**
>
> *Avoid using this request. It should probably be removed.* Using it typically means the device and driver will lose toggle synchronization. If you really lost synchronization, you likely need to completely handshake with the device, using a request like CLEAR_HALT or SET_INTERFACE.

USBDEVFS_DROP_PRIVILEGES

This is used to relinquish the ability to do certain operations which are considered to be privileged on a usbfs file descriptor. This includes claiming arbitrary interfaces, resetting a device on which there are currently claimed interfaces from other users, and issuing USBDEVFS_IOCTL calls. The ioctl parameter is a 32 bit mask of interfaces the user is allowed to claim on this file descriptor. You may issue this ioctl more than one time to narrow said mask.

Synchronous I/O Support

Synchronous requests involve the kernel blocking until the user mode request completes, either by finishing successfully or by reporting an error. In most cases this is the simplest way to use usbfs, although as noted above it does prevent performing I/O to more than one endpoint at a time.

USBDEVFS_BULK

Issues a bulk read or write request to the device. The ioctl parameter is a pointer to this structure:

```
struct usbdevfs_bulktransfer {
        unsigned int  ep;
        unsigned int  len;
        unsigned int  timeout; /* in milliseconds */
        void          *data;
};
```

The `ep` value identifies a bulk endpoint number (1 to 15, as identified in an endpoint descriptor), masked with USB_DIR_IN when referring to an endpoint which sends data to the host from the device. The length of the data buffer is identified by `len`; Recent kernels support requests up to about 128KBytes. *FIXME say how read length is returned, and how short reads are handled.*.

USBDEVFS_CLEAR_HALT

Clears endpoint halt (stall) and resets the endpoint toggle. This is only meaningful for bulk or interrupt endpoints. The ioctl parameter is an integer endpoint number (1 to 15, as identified in an endpoint descriptor), masked with USB_DIR_IN when referring to an endpoint which sends data to the host from the device.

Use this on bulk or interrupt endpoints which have stalled, returning `-EPIPE` status to a data transfer request. Do not issue the control request directly, since that could invalidate the host's record of the data toggle.

USBDEVFS_CONTROL

Issues a control request to the device. The ioctl parameter points to a structure like this:

```
struct usbdevfs_ctrltransfer {
        __u8   bRequestType;
        __u8   bRequest;
        __u16  wValue;
        __u16  wIndex;
        __u16  wLength;
        __u32  timeout;  /* in milliseconds */
        void   *data;
};
```

The first eight bytes of this structure are the contents of the SETUP packet to be sent to the device; see the USB 2.0

specification for details. The bRequestType value is composed by combining a `USB_TYPE_*` value, a `USB_DIR_*` value, and a `USB_RECIP_*` value (from `linux/usb.h`). If wLength is nonzero, it describes the length of the data buffer, which is either written to the device (USB_DIR_OUT) or read from the device (USB_DIR_IN).

At this writing, you can't transfer more than 4 KBytes of data to or from a device; usbfs has a limit, and some host controller drivers have a limit. (That's not usually a problem.) *Also* there's no way to say it's not OK to get a short read back from the device.

USBDEVFS_RESET

Does a USB level device reset. The ioctl parameter is ignored. After the reset, this rebinds all device interfaces. File modification time is not updated by this request.

> **Warning**
>
> *Avoid using this call* until some usbcore bugs get fixed, since it does not fully synchronize device, interface, and driver (not just usbfs) state.

USBDEVFS_SETINTERFACE

Sets the alternate setting for an interface. The ioctl parameter is a pointer to a structure like this:

```
struct usbdevfs_setinterface {
        unsigned int  interface;
        unsigned int  altsetting;
};
```

File modification time is not updated by this request.

Those struct members are from some interface descriptor applying to the current configuration. The interface number is the bInterfaceNumber value, and the altsetting number is the bAlternateSetting value. (This resets each endpoint in the interface.)

USBDEVFS_SETCONFIGURATION

Issues the :c:func:`usb_set_configuration()` call for the device. The parameter is an integer holding the number of a configuration (bConfigurationValue from descriptor). File modification time is not updated by this request.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]usb.rst`, line 603); *backlink***
>
> Unknown interpreted text role "c:func".

> **Warning**
>
> *Avoid using this call* until some usbcore bugs get fixed, since it does not fully synchronize device, interface, and driver (not just usbfs) state.

Asynchronous I/O Support

As mentioned above, there are situations where it may be important to initiate concurrent operations from user mode code. This is particularly important for periodic transfers (interrupt and isochronous), but it can be used for other kinds of USB requests too. In such cases, the asynchronous requests described here are essential. Rather than submitting one request and having the kernel block until it completes, the blocking is separate.

These requests are packaged into a structure that resembles the URB used by kernel device drivers. (No POSIX Async I/O support here, sorry.) It identifies the endpoint type (`USBDEVFS_URB_TYPE_*`), endpoint (number, masked with USB_DIR_IN as appropriate), buffer and length, and a user "context" value serving to uniquely identify each request. (It's usually a pointer to per-request data.) Flags can modify requests (not as many as supported for kernel drivers).

Each request can specify a realtime signal number (between SIGRTMIN and SIGRTMAX, inclusive) to request a signal be sent when the request completes.

When usbfs returns these urbs, the status value is updated, and the buffer may have been modified. Except for isochronous transfers, the actual_length is updated to say how many bytes were transferred; if the USBDEVFS_URB_DISABLE_SPD flag is set ("short packets are not OK"), if fewer bytes were read than were requested then you get an error report:

```
struct usbdevfs_iso_packet_desc {
        unsigned int                    length;
        unsigned int                    actual_length;
        unsigned int                    status;
};

struct usbdevfs_urb {
```

```
        unsigned char                   type;
        unsigned char                   endpoint;
        int                             status;
        unsigned int                    flags;
        void                            *buffer;
        int                             buffer_length;
        int                             actual_length;
        int                             start_frame;
        int                             number_of_packets;
        int                             error_count;
        unsigned int                    signr;
        void                            *usercontext;
        struct usbdevfs_iso_packet_desc  iso_frame_desc[];
};
```

For these asynchronous requests, the file modification time reflects when the request was initiated. This contrasts with their use with the synchronous requests, where it reflects when requests complete.

USBDEVFS_DISCARDURB
: *TBS* File modification time is not updated by this request.

USBDEVFS_DISCSIGNAL
: *TBS* File modification time is not updated by this request.

USBDEVFS_REAPURB
: *TBS* File modification time is not updated by this request.

USBDEVFS_REAPURBNDELAY
: *TBS* File modification time is not updated by this request.

USBDEVFS_SUBMITURB
: *TBS*

The USB devices

The USB devices are now exported via debugfs:

- `/sys/kernel/debug/usb/devices` ... a text file showing each of the USB devices on known to the kernel, and their configuration descriptors. You can also poll() this to learn about new devices.

/sys/kernel/debug/usb/devices

This file is handy for status viewing tools in user mode, which can scan the text format and ignore most of it. More detailed device status (including class and vendor status) is available from device-specific files. For information about the current format of this file, see below.

This file, in combination with the poll() system call, can also be used to detect when devices are added or removed:

```
int fd;
struct pollfd pfd;

fd = open("/sys/kernel/debug/usb/devices", O_RDONLY);
pfd = { fd, POLLIN, 0 };
for (;;) {
    /* The first time through, this call will return immediately. */
    poll(&pfd, 1, -1);

    /* To see what's changed, compare the file's previous and current
       contents or scan the filesystem.  (Scanning is more precise.) */
}
```

Note that this behavior is intended to be used for informational and debug purposes. It would be more appropriate to use programs such as udev or HAL to initialize a device or start a user-mode helper program, for instance.

In this file, each device's output has multiple lines of ASCII output.

I made it ASCII instead of binary on purpose, so that someone can obtain some useful data from it without the use of an auxiliary program. However, with an auxiliary program, the numbers in the first 4 columns of each `T:` line (topology info: Lev, Prnt, Port, Cnt) can be used to build a USB topology diagram.

Each line is tagged with a one-character ID for that line:

```
T = Topology (etc.)
B = Bandwidth (applies only to USB host controllers, which are
virtualized as root hubs)
D = Device descriptor info.
P = Product ID info. (from Device descriptor, but they won't fit
together on one line)
S = String descriptors.
C = Configuration descriptor info. (* = active configuration)
I = Interface descriptor info.
```

```
E = Endpoint descriptor info.
```

/sys/kernel/debug/usb/devices output format

Legend::

d = decimal number (may have leading spaces or 0's) x = hexadecimal number (may have leading spaces or 0's) s = string

Topology info

```
T:  Bus=dd Lev=dd Prnt=dd Port=dd Cnt=dd Dev#=ddd Spd=dddd MxCh=dd
|   |      |      |       |       |      |       |       |__MaxChildren
|   |      |      |       |       |      |       |__Device Speed in Mbps
|   |      |      |       |       |      |__DeviceNumber
|   |      |      |       |       |__Count of devices at this level
|   |      |      |       |__Connector/Port on Parent for this device
|   |      |      |__Parent DeviceNumber
|   |      |__Level in topology for this bus
|   |__Bus number
|__Topology info tag
```

Speed may be:

1.5	Mbit/s for low speed USB
12	Mbit/s for full speed USB
480	Mbit/s for high speed USB (added for USB 2.0); also used for Wireless USB, which has no fixed speed
5000	Mbit/s for SuperSpeed USB (added for USB 3.0)

For reasons lost in the mists of time, the Port number is always too low by 1. For example, a device plugged into port 4 will show up with `Port=03`.

Bandwidth info

```
B:  Alloc=ddd/ddd us (xx%), #Int=ddd, #Iso=ddd
|   |                       |         |__Number of isochronous requests
|   |                       |__Number of interrupt requests
|   |__Total Bandwidth allocated to this bus
|__Bandwidth info tag
```

Bandwidth allocation is an approximation of how much of one frame (millisecond) is in use. It reflects only periodic transfers, which are the only transfers that reserve bandwidth. Control and bulk transfers use all other bandwidth, including reserved bandwidth that is not used for transfers (such as for short packets).

The percentage is how much of the "reserved" bandwidth is scheduled by those transfers. For a low or full speed bus (loosely, "USB 1.1"), 90% of the bus bandwidth is reserved. For a high speed bus (loosely, "USB 2.0") 80% is reserved.

Device descriptor info & Product ID info

```
D:  Ver=x.xx Cls=xx(s) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
P:  Vendor=xxxx ProdID=xxxx Rev=xx.xx
```

where:

```
D:  Ver=x.xx Cls=xx(sssss) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
|   |        |             |      |       |       |__NumberConfigurations
|   |        |             |      |       |__MaxPacketSize of Default Endpoint
|   |        |             |      |__DeviceProtocol
|   |        |             |__DeviceSubClass
|   |        |__DeviceClass
|   |__Device USB version
|__Device info tag #1
```

where:

```
P:  Vendor=xxxx ProdID=xxxx Rev=xx.xx
|   |           |           |__Product revision number
|   |           |__Product ID code
|   |__Vendor ID code
|__Device info tag #2
```

String descriptor info

```
S:  Manufacturer=ssss
|   |__Manufacturer of this device as read from the device.
|      For USB host controller drivers (virtual root hubs) this may
|      be omitted, or (for newer drivers) will identify the kernel
|      version and the driver which provides this hub emulation.
|__String info tag
```

```
S:  Product=ssss
|   |__Product description of this device as read from the device.
|      For older USB host controller drivers (virtual root hubs) this
|      indicates the driver; for newer ones, it's a product (and vendor)
|      description that often comes from the kernel's PCI ID database.
|__String info tag

S:  SerialNumber=ssss
|   |__Serial Number of this device as read from the device.
|      For USB host controller drivers (virtual root hubs) this is
|      some unique ID, normally a bus ID (address or slot name) that
|      can't be shared with any other device.
|__String info tag
```

Configuration descriptor info

```
C:* #Ifs=dd Cfg#=dd Atr=xx MPwr=dddmA
| | |       |       |      |__MaxPower in mA
| | |       |       |__Attributes
| | |       |__ConfigurationNumber
| | |__NumberOfInterfaces
| |__ "*" indicates the active configuration (others are " ")
|__Config info tag
```

USB devices may have multiple configurations, each of which act rather differently. For example, a bus-powered configuration might be much less capable than one that is self-powered. Only one device configuration can be active at a time; most devices have only one configuration.

Each configuration consists of one or more interfaces. Each interface serves a distinct "function", which is typically bound to a different USB device driver. One common example is a USB speaker with an audio interface for playback, and a HID interface for use with software volume control.

Interface descriptor info (can be multiple per Config)

```
I:* If#=dd Alt=dd #EPs=dd Cls=xx(sssss) Sub=xx Prot=xx Driver=ssss
| | |      |      |       |              |      |       |__Driver name
| | |      |      |       |              |      |          or "(none)"
| | |      |      |       |              |      |__InterfaceProtocol
| | |      |      |       |              |__InterfaceSubClass
| | |      |      |       |__InterfaceClass
| | |      |      |__NumberOfEndpoints
| | |      |__AlternateSettingNumber
| | |__InterfaceNumber
| |__ "*" indicates the active altsetting (others are " ")
|__Interface info tag
```

A given interface may have one or more "alternate" settings. For example, default settings may not use more than a small amount of periodic bandwidth. To use significant fractions of bus bandwidth, drivers must select a non-default altsetting.

Only one setting for an interface may be active at a time, and only one driver may bind to an interface at a time. Most devices have only one alternate setting per interface.

Endpoint descriptor info (can be multiple per Interface)

```
E:  Ad=xx(s) Atr=xx(ssss) MxPS=dddd Ivl=dddss
|   |        |            |         |__Interval (max) between transfers
|   |        |            |__EndpointMaxPacketSize
|   |        |__Attributes(EndpointType)
|   |__EndpointAddress(I=In,O=Out)
|__Endpoint info tag
```

The interval is nonzero for all periodic (interrupt or isochronous) endpoints. For high speed endpoints the transfer interval may be measured in microseconds rather than milliseconds.

For high speed periodic endpoints, the `EndpointMaxPacketSize` reflects the per-microframe data transfer size. For "high bandwidth" endpoints, that can reflect two or three packets (for up to 3KBytes every 125 usec) per endpoint.

With the Linux-USB stack, periodic bandwidth reservations use the transfer intervals and sizes provided by URBs, which can be less than those found in endpoint descriptor.

Usage examples

If a user or script is interested only in Topology info, for example, use something like `grep ^T: /sys/kernel/debug/usb/devices` for only the Topology lines. A command like `grep -i ^[tdp]: /sys/kernel/debug/usb/devices` can be used to list only the lines that begin with the characters in square brackets, where the valid characters are TDPCIE. With a slightly more able script, it can display any selected lines (for example, only T, D, and P lines) and change their output format. (The `procusb` Perl script is the beginning of this idea. It will list only selected lines [selected from

TBDPSCIE] or "All" lines from `/sys/kernel/debug/usb/devices`.)

The Topology lines can be used to generate a graphic/pictorial of the USB devices on a system's root hub. (See more below on how to do this.)

The Interface lines can be used to determine what driver is being used for each device, and which altsetting it activated.

The Configuration lines could be used to list maximum power (in milliamps) that a system's USB devices are using. For example, `grep ^C: /sys/kernel/debug/usb/devices`.

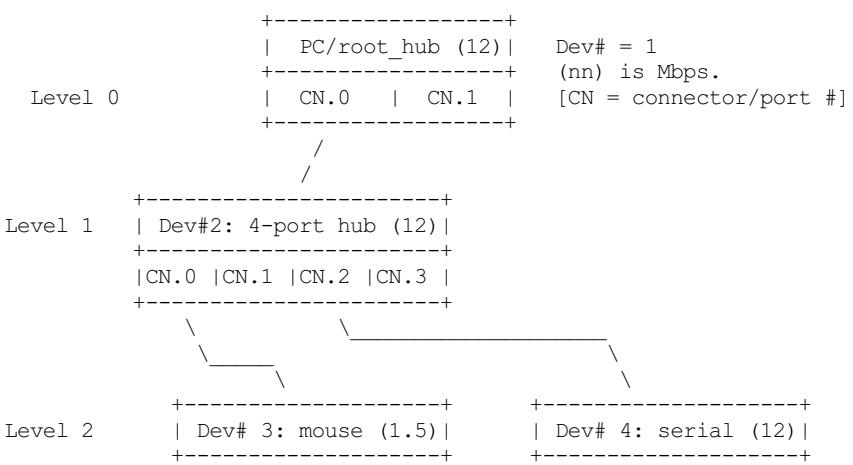
Here's an example, from a system which has a UHCI root hub, an external hub connected to the root hub, and a mouse and a serial converter connected to the external hub.

```
T:  Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12   MxCh= 2
B:  Alloc= 28/900 us ( 3%), #Int=  2, #Iso=  0
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 0.00
S:  Product=USB UHCI Root Hub
S:  SerialNumber=dce0
C:* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=255ms

T:  Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12   MxCh= 4
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0451 ProdID=1446 Rev= 1.00
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=255ms

T:  Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#=  3 Spd=1.5  MxCh= 0
D:  Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=04b4 ProdID=0001 Rev= 0.00
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=mouse
E:  Ad=81(I) Atr=03(Int.) MxPS=   3 Ivl= 10ms

T:  Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#=  4 Spd=12   MxCh= 0
D:  Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0565 ProdID=0001 Rev= 1.08
S:  Manufacturer=Peracom Networks, Inc.
S:  Product=Peracom USB to Serial Converter
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial
E:  Ad=81(I) Atr=02(Bulk) MxPS=  64 Ivl= 16ms
E:  Ad=01(O) Atr=02(Bulk) MxPS=  16 Ivl= 16ms
E:  Ad=82(I) Atr=03(Int.) MxPS=   8 Ivl=  8ms
```

Selecting only the `T:` and `I:` lines from this (for example, by using `procusb ti`), we have

```
T:  Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12   MxCh= 2
T:  Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12   MxCh= 4
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
T:  Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#=  3 Spd=1.5  MxCh= 0
I:  If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=mouse
T:  Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#=  4 Spd=12   MxCh= 0
I:  If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial
```

Physically this looks like (or could be converted to):



Or, in a more tree-like structure (ports [Connectors] without connections could be omitted):

```
PC:  Dev# 1, root hub, 2 ports, 12 Mbps
```

```
|_ CN.0:  Dev# 2, hub, 4 ports, 12 Mbps
     |_ CN.0:  Dev #3, mouse, 1.5 Mbps
     |_ CN.1:
     |_ CN.2:  Dev #4, serial, 12 Mbps
     |_ CN.3:
|_ CN.1:
```