The Linux-USB Host Side API

Table of Contents

Chapter 1. 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 one USB 2.0 tree (480 Mbit/sec each) with a few USB 1.1 trees (12 Mbit/sec each) that are used when you connect a USB 1.1 device directly to the machine's "root hub".

That master/slave asymmetry was designed-in for a number of reasons, one being ease of use. It is not physically possible to assemble (legal) USB cables incorrectly: all upstream "to the host" connectors are the rectangular type (matching the sockets on root hubs), and all downstream connectors are the squarish type (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. And finally, at the electrical level, bus protocol overhead is reduced by eliminating arbitration and moving scheduling into the host software.

USB 1.0 was announced in January 1996 and was revised as USB 1.1 (with improvements in hub specification and support for interrupt-out transfers) in September 1998. USB 2.0 was released in April 2000, adding high-speed transfers and transaction-translating hubs (used for USB 1.1 and 1.0 backward compatibility).

Kernel developers added USB support to Linux early in the 2.2 kernel series, shortly before 2.3 development forked. Updates from 2.3 were regularly folded back into 2.2 releases, which improved reliability and brought `/sbin/hotplug` support as well more drivers. Such improvements were continued in the 2.5 kernel series, where they added USB 2.0 support, improved performance, and made the host controller drivers (HCDs) more consistent. They also simplified the API (to make bugs less likely) and added internal "kerneldoc" documentation.

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.

Chapter 2. 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 that are capable of high-speed operation must also support full-speed configurations, along with a way to ask about the "other speed" configurations which might be used.

- Configurations 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 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 asynchnous 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 2.0 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 on the 2.5 kernels, but there are still differences that crop up especially with fault handling. 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. (You will need external USB 1.1 and/or USB 2.0

hubs to perform all those tests.)

Chapter 3. USB-Standard Types

Table of Contents

In `<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, and usbfs.

Name

struct usb_ctrlrequest — SETUP data for a USB device control request

Synopsis

```
struct usb_ctrlrequest {
  __u8 bRequestType;
  __u8 bRequest;
  __le16 wValue;
  __le16 wIndex;
  __le16 wLength;
};
```

Members

bRequestType

matches the USB bmRequestType field

bRequest

matches the USB bRequest field

wValue

matches the USB wValue field (le16 byte order)

wIndex

matches the USB wIndex field (le16 byte order)

wLength

matches the USB wLength field (le16 byte order)

Description

This structure is used to send control requests to a USB device. It matches the different fields of the USB 2.0 Spec section 9.3, table 9-2. See the USB spec for a fuller description of the different fields, and what they are used for.

Note that the driver for any interface can issue control requests. For most devices, interfaces don't coordinate with each other, so such requests may be made at any time.

Name

usb_endpoint_num — get the endpoint's number

Synopsis

```
int usb_endpoint_num (epd);

const struct usb_endpoint_descriptor * epd;
```

Arguments

epd

endpoint to be checked

Description

Returns *epd*'s number: 0 to 15.

Name

usb_endpoint_type — get the endpoint's transfer type

Synopsis

```
int usb_endpoint_type (epd);

const struct usb_endpoint_descriptor * epd;
```

Arguments

epd

endpoint to be checked

Description

Returns one of USB_ENDPOINT_XFER_{CONTROL, ISOC, BULK, INT} according to *epd*'s transfer type.

Name

usb_endpoint_dir_in — check if the endpoint has IN direction

Synopsis

```
int usb_endpoint_dir_in (epd);

const struct usb_endpoint_descriptor * epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint is of type IN, otherwise it returns false.

Name

usb_endpoint_dir_out — check if the endpoint has OUT direction

Synopsis

`int usb_endpoint_dir_out (epd);`

`const struct usb_endpoint_descriptor * epd;`

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint is of type OUT, otherwise it returns false.

Name

usb_endpoint_xfer_bulk — check if the endpoint has bulk transfer type

Synopsis

`int usb_endpoint_xfer_bulk (epd);`

`const struct usb_endpoint_descriptor * epd;`

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint is of type bulk, otherwise it returns false.

Name

usb_endpoint_xfer_control — check if the endpoint has control transfer type

Synopsis

`int usb_endpoint_xfer_control (epd);`

`const struct usb_endpoint_descriptor * epd;`

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint is of type control, otherwise it returns false.

Name

usb_endpoint_xfer_int — check if the endpoint has interrupt transfer type

Synopsis

```
int usb_endpoint_xfer_int (epd);
```

```
const struct usb_endpoint_descriptor *  epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint is of type interrupt, otherwise it returns false.

Name

usb_endpoint_xfer_isoc — check if the endpoint has isochronous transfer type

Synopsis

```
int usb_endpoint_xfer_isoc (epd);
```

```
const struct usb_endpoint_descriptor *  epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint is of type isochronous, otherwise it returns false.

Name

usb_endpoint_is_bulk_in — check if the endpoint is bulk IN

Synopsis

```
int usb_endpoint_is_bulk_in (epd);
```

```
const struct usb_endpoint_descriptor *  epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint has bulk transfer type and IN direction, otherwise it returns false.

Name

usb_endpoint_is_bulk_out — check if the endpoint is bulk OUT

Synopsis

```
int usb_endpoint_is_bulk_out (epd);
```

```
const struct usb_endpoint_descriptor *  epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint has bulk transfer type and OUT direction, otherwise it returns false.

Name

usb_endpoint_is_int_in — check if the endpoint is interrupt IN

Synopsis

`int` **`usb_endpoint_is_int_in`** `(`*epd*`);`

`const struct usb_endpoint_descriptor *` *`epd`*`;`

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint has interrupt transfer type and IN direction, otherwise it returns false.

Name

usb_endpoint_is_int_out — check if the endpoint is interrupt OUT

Synopsis

`int` **`usb_endpoint_is_int_out`** `(`*epd*`);`

`const struct usb_endpoint_descriptor *` *`epd`*`;`

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint has interrupt transfer type and OUT direction, otherwise it returns false.

Name

usb_endpoint_is_isoc_in — check if the endpoint is isochronous IN

Synopsis

```
int usb_endpoint_is_isoc_in (epd);

const struct usb_endpoint_descriptor * epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint has isochronous transfer type and IN direction, otherwise it returns false.

Name

usb_endpoint_is_isoc_out — check if the endpoint is isochronous OUT

Synopsis

```
int usb_endpoint_is_isoc_out (epd);

const struct usb_endpoint_descriptor * epd;
```

Arguments

epd

endpoint to be checked

Description

Returns true if the endpoint has isochronous transfer type and OUT direction, otherwise it returns false.

Chapter 4. Host-Side Data Types and Macros

Table of Contents

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.

Name

struct usb_host_ss_ep_comp — Valid for SuperSpeed devices only

Synopsis

```
struct usb_host_ss_ep_comp {
  struct usb_ss_ep_comp_descriptor desc;
  unsigned char * extra;
  int extralen;
};
```

Members

desc

endpoint companion descriptor, wMaxPacketSize in native byteorder

extra

descriptors following this endpoint companion descriptor

extralen

how many bytes of “extra” are valid

Name

struct usb_host_endpoint — host-side endpoint descriptor and queue

Synopsis

```
struct usb_host_endpoint {
  struct usb_endpoint_descriptor desc;
  struct list_head urb_list;
  void * hcpriv;
  struct ep_device * ep_dev;
  struct usb_host_ss_ep_comp * ss_ep_comp;
  unsigned char * extra;
  int extralen;
  int enabled;
};
```

Members

desc

descriptor for this endpoint, wMaxPacketSize in native byteorder

urb_list

urbs queued to this endpoint; maintained by usbcore

hcpriv

for use by HCD; typically holds hardware dma queue head (QH) with one or more transfer descriptors (TDs) per urb

ep_dev

ep_device for sysfs info

ss_ep_comp

companion descriptor information for this endpoint

extra

descriptors following this endpoint in the configuration

extralen

how many bytes of “extra” are valid

enabled

URBs may be submitted to this endpoint

Description

USB requests are always queued to a given endpoint, identified by a descriptor within an active interface in a given USB configuration.

Name

struct usb_interface — what usb device drivers talk to

Synopsis

```
struct usb_interface {
  struct usb_host_interface * altsetting;
  struct usb_host_interface * cur_altsetting;
  unsigned num_altsetting;
  struct usb_interface_assoc_descriptor * intf_assoc;
  int minor;
  enum usb_interface_condition condition;
  unsigned is_active:1;
  unsigned sysfs_files_created:1;
  unsigned ep_devs_created:1;
  unsigned unregistering:1;
  unsigned needs_remote_wakeup:1;
  unsigned needs_altsetting0:1;
  unsigned needs_binding:1;
  unsigned reset_running:1;
  struct device dev;
  struct device * usb_dev;
  atomic_t pm_usage_cnt;
  struct work_struct reset_ws;
};
```

Members

altsetting

array of interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order.

cur_altsetting

the current altsetting.

num_altsetting

number of altsettings defined.

intf_assoc

interface association descriptor

minor

the minor number assigned to this interface, if this interface is bound to a driver that uses the USB major number. If this interface does not use the USB major, this field should be unused. The driver should set this value in the `probe` function of the driver, after it has been assigned a minor number from the USB core by calling `usb_register_dev`.

condition

binding state of the interface: not bound, binding (in `probe`), bound to a driver, or unbinding (in `disconnect`)

is_active

flag set when the interface is bound and not suspended.

sysfs_files_created

sysfs attributes exist

ep_devs_created

endpoint child pseudo-devices exist

unregistering

flag set when the interface is being unregistered

needs_remote_wakeup

flag set when the driver requires remote-wakeup capability during autosuspend.

needs_altsetting0

flag set when a set-interface request for altsetting 0 has been deferred.

needs_binding

flag set when the driver should be re-probed or unbound following a reset or suspend operation it doesn't support.

reset_running

set to 1 if the interface is currently running a queued reset so that `usb_cancel_queued_reset` doesn't try to remove from the workqueue when running inside the worker thread. See `__usb_queue_reset_device`.

dev

driver model's view of this device

usb_dev

if an interface is bound to the USB major, this will point to the sysfs representation for that device.

pm_usage_cnt

PM usage counter for this interface; autosuspend is not allowed unless the counter is 0.

reset_ws

Used for scheduling resets from atomic context.

Description

USB device drivers attach to interfaces on a physical device. Each interface encapsulates a single high level function, such as feeding an audio stream to a speaker or reporting a change in a volume control. Many USB devices only have one interface. The protocol used to talk to an interface's endpoints can be defined in a usb “class” specification, or by a product's vendor. The (default) control endpoint is part of every interface, but is never listed among the interface's descriptors.

The driver that is bound to the interface can use standard driver model calls such as `dev_get_drvdata` on the dev member of this structure.

Each interface may have alternate settings. The initial configuration of a device sets altsetting 0, but the device driver can change that setting using `usb_set_interface`. Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-conformant USB devices that use isochronous endpoints will use them in non-default settings.

The USB specification says that alternate setting numbers must run from 0 to one less than the total number of alternate settings. But some devices manage to mess this up, and the structures aren't necessarily stored in numerical order anyhow. Use `usb_altnum_to_altsetting` to look up an alternate setting in the altsetting array based on its number.

Name

struct usb_interface_cache — long-term representation of a device interface

Synopsis

```
struct usb_interface_cache {
  unsigned num_altsetting;
  struct kref ref;
  struct usb_host_interface altsetting[0];
};
```

Members

num_altsetting

number of altsettings defined.

ref

reference counter.

altsetting[0]

variable-length array of interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order.

Description

These structures persist for the lifetime of a usb_device, unlike struct usb_interface (which persists only as long as its configuration is installed). The altsetting arrays can be accessed through these structures at any time, permitting comparison of configurations and providing support for the /proc/bus/usb/devices pseudo-file.

Name

struct usb_host_config — representation of a device's configuration

Synopsis

```
struct usb_host_config {
  struct usb_config_descriptor desc;
  char * string;
  struct usb_interface_assoc_descriptor * intf_assoc[USB_MAXIADS];
  struct usb_interface * interface[USB_MAXINTERFACES];
  struct usb_interface_cache * intf_cache[USB_MAXINTERFACES];
  unsigned char * extra;
  int extralen;
};
```

Members

desc

the device's configuration descriptor.

string

pointer to the cached version of the iConfiguration string, if present for this configuration.

intf_assoc[USB_MAXIADS]

list of any interface association descriptors in this config

interface[USB_MAXINTERFACES]

array of pointers to usb_interface structures, one for each interface in the configuration. The number of interfaces is stored in desc.bNumInterfaces. These pointers are valid only while the the configuration is active.

intf_cache[USB_MAXINTERFACES]

array of pointers to usb_interface_cache structures, one for each interface in the configuration. These structures exist for the entire life of the device.

extra

pointer to buffer containing all extra descriptors associated with this configuration (those preceding the first interface descriptor).

extralen

length of the extra descriptors buffer.

Description

USB devices may have multiple configurations, but only one can be active at any time. Each encapsulates a different operational environment; for example, a dual-speed device would have separate configurations for full-speed and high-speed operation. The number of configurations available is stored in the device descriptor as bNumConfigurations.

A configuration can contain multiple interfaces. Each corresponds to a different function of the USB device, and all are available whenever the configuration is active. The USB standard says that interfaces are supposed to be numbered from 0 to desc.bNumInterfaces-1, but a lot of devices get this wrong. In addition, the interface array is not guaranteed to be sorted in numerical order. Use `usb_ifnum_to_if` to look up an interface entry based on its number.

Device drivers should not attempt to activate configurations. The choice of which configuration to install is a policy decision based on such considerations as available power, functionality provided, and the user's desires (expressed through userspace tools). However, drivers can call `usb_reset_configuration` to reinitialize the current configuration and all its interfaces.

Name

struct usb_device — kernel's representation of a USB device

Synopsis

```
struct usb_device {
  int devnum;
  char devpath[16];
```

```
  u32 route;
  enum usb_device_state state;
  enum usb_device_speed speed;
  struct usb_tt * tt;
  int ttport;
  unsigned int toggle[2];
  struct usb_device * parent;
  struct usb_bus * bus;
  struct usb_host_endpoint ep0;
  struct device dev;
  struct usb_device_descriptor descriptor;
  struct usb_host_config * config;
  struct usb_host_config * actconfig;
  struct usb_host_endpoint * ep_in[16];
  struct usb_host_endpoint * ep_out[16];
  char ** rawdescriptors;
  unsigned short bus_mA;
  u8 portnum;
  u8 level;
  unsigned can_submit:1;
  unsigned discon_suspended:1;
  unsigned persist_enabled:1;
  unsigned have_langid:1;
  unsigned authorized:1;
  unsigned authenticated:1;
  unsigned wusb:1;
  int string_langid;
  char * product;
  char * manufacturer;
  char * serial;
  struct list_head filelist;
#ifdef CONFIG_USB_DEVICE_CLASS
  struct device * usb_classdev;
#endif
#ifdef CONFIG_USB_DEVICEFS
  struct dentry * usbfs_dentry;
#endif
  int maxchild;
  struct usb_device * children[USB_MAXCHILDREN];
  int pm_usage_cnt;
  u32 quirks;
  atomic_t urbnum;
  unsigned long active_duration;
#ifdef CONFIG_PM
  struct delayed_work autosuspend;
  struct work_struct autoresume;
  struct mutex pm_mutex;
  unsigned long last_busy;
  int autosuspend_delay;
  unsigned long connect_time;
  unsigned auto_pm:1;
  unsigned do_remote_wakeup:1;
  unsigned reset_resume:1;
  unsigned autosuspend_disabled:1;
  unsigned autoresume_disabled:1;
  unsigned skip_sys_resume:1;
#endif
  struct wusb_dev * wusb_dev;
  int slot_id;
};
```

Members

devnum

: device number; address on a USB bus

devpath[16]

: device ID string for use in messages (e.g., /port/...)

route

: tree topology hex string for use with xHCI

state

: device state: configured, not attached, etc.

speed

: device speed: high/full/low (or error)

tt

: Transaction Translator info; used with low/full speed dev, highspeed hub

ttport

: device port on that tt hub

toggle[2]

: one bit for each endpoint, with ([0] = IN, [1] = OUT) endpoints

parent

: our hub, unless we're the root

bus

: bus we're part of

ep0

: endpoint 0 data (default control pipe)

dev

: generic device interface

descriptor

: USB device descriptor

config

all of the device's configs

actconfig

the active configuration

ep_in[16]

array of IN endpoints

ep_out[16]

array of OUT endpoints

rawdescriptors

raw descriptors for each config

bus_mA

Current available from the bus

portnum

parent port number (origin 1)

level

number of USB hub ancestors

can_submit

URBs may be submitted

discon_suspended

disconnected while suspended

persist_enabled

USB_PERSIST enabled for this device

have_langid

whether string_langid is valid

authorized

policy has said we can use it; (user space) policy determines if we authorize this device to be used or not. By default, wired USB devices are authorized. WUSB devices are not, until we authorize them from user space. FIXME -- complete doc authenticated

Crypto authentication passed

wusb

device is Wireless USB

string_langid

language ID for strings

product

iProduct string, if present (static)

manufacturer

iManufacturer string, if present (static)

serial

iSerialNumber string, if present (static)

filelist

usbfs files that are open to this device

usb_classdev

USB class device that was created for usbfs device access from userspace

usbfs_dentry

usbfs dentry entry for the device

maxchild

number of ports if hub

children[USB_MAXCHILDREN]

child devices - USB devices that are attached to this hub

pm_usage_cnt

usage counter for autosuspend

quirks

quirks of the whole device

urbnum

number of URBs submitted for the whole device

active_duration

total time device is not suspended

autosuspend

for delayed autosuspends

autoresume

for autoresumes requested while in_interrupt

pm_mutex

protects PM operations

last_busy

time of last use

autosuspend_delay

in jiffies

connect_time

time device was first connected

auto_pm

autosuspend/resume in progress

do_remote_wakeup

remote wakeup should be enabled

reset_resume

needs reset instead of resume

autosuspend_disabled

autosuspend disabled by the user

autoresume_disabled

autoresume disabled by the user

skip_sys_resume

skip the next system resume

wusb_dev

if this is a Wireless USB device, link to the WUSB specific data for the device.

slot_id

Slot ID assigned by xHCI

Notes

Usbcore drivers should not set usbdev->state directly. Instead use `usb_set_device_state`.

Name

usb_interface_claimed — returns true iff an interface is claimed

Synopsis

```
int usb_interface_claimed (iface);
struct usb_interface * iface;
```

Arguments

iface

the interface being checked

Description

Returns true (nonzero) iff the interface is claimed, else false (zero). Callers must own the driver model's usb bus readlock. So driver `probe` entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

Name

usb_make_path — returns stable device path in the usb tree

Synopsis

```
int usb_make_path (dev,
                   buf,
                   size);
```

```
struct usb_device *  dev;
char *               buf;
size_t               size;
```

Arguments

dev

: the device whose path is being constructed

buf

: where to put the string

size

: how big is “buf”?

Description

Returns length of the string (> 0) or negative if size was too small.

This identifier is intended to be “stable”, reflecting physical paths in hardware such as physical bus addresses for host controllers or ports on USB hubs. That makes it stay the same until systems are physically reconfigured, by re-cabling a tree of USB devices or by moving USB host controllers. Adding and removing devices, including virtual root hubs in host controller driver modules, does not change these path identifers; neither does rebooting or re-enumerating. These are more useful identifiers than changeable (“unstable”) ones like bus numbers or device addresses.

With a partial exception for devices connected to USB 2.0 root hubs, these identifiers are also predictable. So long as the device tree isn't changed, plugging any USB device into a given hub port always gives it the same path. Because of the use of “companion” controllers, devices connected to ports on USB 2.0 root hubs (EHCI host controllers) will get one path ID if they are high speed, and a different one if they are full or low speed.

Name

USB_DEVICE — macro used to describe a specific usb device

Synopsis

```
USB_DEVICE (vend,
            prod);
```

```
vend;
prod;
```

Arguments

vend

the 16 bit USB Vendor ID

prod

the 16 bit USB Product ID

Description

This macro is used to create a struct usb_device_id that matches a specific device.

Name

USB_DEVICE_VER — describe a specific usb device with a version range

Synopsis

```
USB_DEVICE_VER (vend,
                prod,
                lo,
                hi);

vend;
prod;
lo;
hi;
```

Arguments

vend

the 16 bit USB Vendor ID

prod

the 16 bit USB Product ID

lo

the bcdDevice_lo value

hi

the bcdDevice_hi value

Description

This macro is used to create a struct usb_device_id that matches a specific device, with a version range.

Name

USB_DEVICE_INTERFACE_PROTOCOL — describe a usb device with a specific interface protocol

Synopsis

```
USB_DEVICE_INTERFACE_PROTOCOL (vend,
                               prod,
                               pr);

 vend;
 prod;
 pr;
```

Arguments

vend

    the 16 bit USB Vendor ID

prod

    the 16 bit USB Product ID

pr

    bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific interface protocol of devices.

Name

USB_DEVICE_INFO — macro used to describe a class of usb devices

Synopsis

```
USB_DEVICE_INFO (cl,
                 sc,
                 pr);

 cl;
 sc;
```

pr;

Arguments

cl

bDeviceClass value

sc

bDeviceSubClass value

pr

bDeviceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific class of devices.

Name

USB_INTERFACE_INFO — macro used to describe a class of usb interfaces

Synopsis

```
USB_INTERFACE_INFO (cl,
                    sc,
                    pr);
```

cl;
sc;
pr;

Arguments

cl

bInterfaceClass value

sc

bInterfaceSubClass value

pr

bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific class of interfaces.

Name

USB_DEVICE_AND_INTERFACE_INFO — describe a specific usb device with a class of usb interfaces

Synopsis

```
USB_DEVICE_AND_INTERFACE_INFO (vend,
                               prod,
                               cl,
                               sc,
                               pr);
```

```
 vend;
 prod;
 cl;
 sc;
 pr;
```

Arguments

vend

the 16 bit USB Vendor ID

prod

the 16 bit USB Product ID

cl

bInterfaceClass value

sc

bInterfaceSubClass value

pr

bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific device with a specific class of interfaces.

This is especially useful when explicitly matching devices that have vendor specific bDeviceClass values, but standards-compliant interfaces.

Name

struct usbdrv_wrap — wrapper for driver-model structure

Synopsis

```
struct usbdrv_wrap {
  struct device_driver driver;
  int for_devices;
};
```

Members

driver

    The driver-model core driver structure.

for_devices

    Non-zero for device drivers, 0 for interface drivers.

Name

struct usb_driver — identifies USB interface driver to usbcore

Synopsis

```
struct usb_driver {
  const char * name;
  int (* probe) (struct usb_interface *intf,const struct usb_device_id *id);
  void (* disconnect) (struct usb_interface *intf);
  int (* ioctl) (struct usb_interface *intf, unsigned int code,void *buf);
  int (* suspend) (struct usb_interface *intf, pm_message_t message);
  int (* resume) (struct usb_interface *intf);
  int (* reset_resume) (struct usb_interface *intf);
  int (* pre_reset) (struct usb_interface *intf);
  int (* post_reset) (struct usb_interface *intf);
  const struct usb_device_id * id_table;
  struct usb_dynids dynids;
  struct usbdrv_wrap drvwrap;
  unsigned int no_dynamic_id:1;
  unsigned int supports_autosuspend:1;
  unsigned int soft_unbind:1;
};
```

Members

name

The driver name should be unique among USB drivers, and should normally be the same as the module name.

probe

Called to see if the driver is willing to manage a particular interface on a device. If it is, probe returns zero and uses `usb_set_intfdata` to associate driver-specific data with the interface. It may also use `usb_set_interface` to specify the appropriate altsetting. If unwilling to manage the interface, return -ENODEV, if genuine IO errors occured, an appropriate negative errno value.

disconnect

Called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded.

ioctl

Used for drivers that want to talk to userspace through the "usbfs" filesystem. This lets devices provide ways to expose information to user space regardless of where they do (or don't) show up otherwise in the filesystem.

suspend

Called when the device is going to be suspended by the system.

resume

Called when the device is being resumed by the system.

reset_resume

Called when the suspended device has been reset instead of being resumed.

pre_reset

Called by `usb_reset_device` when the device is about to be reset.

post_reset

Called by `usb_reset_device` after the device has been reset

id_table

USB drivers use ID table to support hotplugging. Export this with MODULE_DEVICE_TABLE(usb,...). This must be set or your driver's probe function will never get called.

dynids

used internally to hold the list of dynamically added device ids for this driver.

drvwrap

Driver-model core structure wrapper.

no_dynamic_id

if set to 1, the USB core will not allow dynamic ids to be added to this driver by preventing the sysfs file from being created.

supports_autosuspend

if set to 0, the USB core will not allow autosuspend for interfaces bound to this driver.

soft_unbind

if set to 1, the USB core will not kill URBs and disable endpoints before calling the driver's disconnect method.

Description

USB interface drivers must provide a name, `probe` and `disconnect` methods, and an id_table. Other driver fields are optional.

The id_table is used in hotplugging. It holds a set of descriptors, and specialized data may be associated with each entry. That table is used by both user and kernel mode hotplugging support.

The `probe` and `disconnect` methods are called in a context where they can sleep, but they should avoid abusing the privilege. Most work to connect to a device should be done when the device is opened, and undone at the last close. The disconnect code needs to address concurrency issues with respect to `open` and `close` methods, as well as forcing all pending I/O requests to complete (by unlinking them as necessary, and blocking until the unlinks complete).

Name

struct usb_device_driver — identifies USB device driver to usbcore

Synopsis

```
struct usb_device_driver {
  const char * name;
  int (* probe) (struct usb_device *udev);
  void (* disconnect) (struct usb_device *udev);
  int (* suspend) (struct usb_device *udev, pm_message_t message);
  int (* resume) (struct usb_device *udev, pm_message_t message);
  struct usbdrv_wrap drvwrap;
  unsigned int supports_autosuspend:1;
};
```

Members

name

The driver name should be unique among USB drivers, and should normally be the same as the module name.

probe

Called to see if the driver is willing to manage a particular device. If it is, probe returns zero and uses `dev_set_drvdata` to associate driver-specific data with the device. If unwilling to manage the device, return a negative errno value.

disconnect

Called when the device is no longer accessible, usually because it has been (or is being) disconnected or the driver's module is being unloaded.

suspend

Called when the device is going to be suspended by the system.

resume

Called when the device is being resumed by the system.

drvwrap

Driver-model core structure wrapper.

supports_autosuspend

if set to 0, the USB core will not allow autosuspend for devices bound to this driver.

Description

USB drivers must provide all the fields listed above except drvwrap.

Name

struct usb_class_driver — identifies a USB driver that wants to use the USB major number

Synopsis

```
struct usb_class_driver {
  char * name;
  char *(* devnode) (struct device *dev, mode_t *mode);
  const struct file_operations * fops;
  int minor_base;
```

```
};
```

Members

name

the usb class device name for this driver. Will show up in sysfs.

devnode

Callback to provide a naming hint for a possible device node to create.

fops

pointer to the struct file_operations of this driver.

minor_base

the start of the minor range for this driver.

Description

This structure is used for the `usb_register_dev` and `usb_unregister_dev` functions, to consolidate a number of the parameters used for them.

Name

struct urb — USB Request Block

Synopsis

```
struct urb {
  struct list_head urb_list;
  struct list_head anchor_list;
  struct usb_anchor * anchor;
  struct usb_device * dev;
  struct usb_host_endpoint * ep;
  unsigned int pipe;
  int status;
  unsigned int transfer_flags;
  void * transfer_buffer;
  dma_addr_t transfer_dma;
  struct usb_sg_request * sg;
  int num_sgs;
  u32 transfer_buffer_length;
  u32 actual_length;
  unsigned char * setup_packet;
  dma_addr_t setup_dma;
  int start_frame;
  int number_of_packets;
  int interval;
```

```
  int error_count;
  void * context;
  usb_complete_t complete;
  struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

Members

urb_list

For use by current owner of the URB.

anchor_list

membership in the list of an anchor

anchor

to anchor URBs to a common mooring

dev

Identifies the USB device to perform the request.

ep

Points to the endpoint's data structure. Will eventually replace *`pipe`*.

pipe

Holds endpoint number, direction, type, and more. Create these values with the eight macros available; usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is “ctrl” (control), “bulk”, “int” (interrupt), or “iso” (isochronous). For example `usb_sndbulkpipe` or `usb_rcvintpipe`. Endpoint numbers range from zero to fifteen. Note that “in” endpoint two is a different endpoint (and pipe) from “out” endpoint two. The current configuration controls the existence, type, and maximum packet size of any given endpoint.

status

This is read in non-iso completion functions to get the status of the particular request. ISO requests only use it to tell whether the URB was unlinked; detailed status for each frame is in the fields of the iso_frame-desc.

transfer_flags

A variety of flags may be used to affect how URB submission, unlinking, or operation are handled. Different kinds of URB can use different flags.

transfer_buffer

This identifies the buffer to (or from) which the I/O request will be performed unless URB_NO_TRANSFER_DMA_MAP is set (however, do not leave garbage in transfer_buffer even then). This buffer must be suitable for DMA; allocate it with `kmalloc` or equivalent. For transfers to “in” endpoints, contents of this buffer will be modified. This buffer is used for the data stage of control transfers.

transfer_dma

When transfer_flags includes URB_NO_TRANSFER_DMA_MAP, the device driver is saying that it provided this DMA address, which the host controller driver should use in preference to the transfer_buffer.

sg

scatter gather buffer list

num_sgs

number of entries in the sg list

transfer_buffer_length

How big is transfer_buffer. The transfer may be broken up into chunks according to the current maximum packet size for the endpoint, which is a function of the configuration and is encoded in the pipe. When the length is zero, neither transfer_buffer nor transfer_dma is used.

actual_length

This is read in non-iso completion functions, and it tells how many bytes (out of transfer_buffer_length) were transferred. It will normally be the same as requested, unless either an error was reported or a short read was performed. The URB_SHORT_NOT_OK transfer flag may be used to make such short reads be reported as errors.

setup_packet

Only used for control transfers, this points to eight bytes of setup data. Control transfers always start by sending this data to the device. Then transfer_buffer is read or written, if needed.

setup_dma

For control transfers with URB_NO_SETUP_DMA_MAP set, the device driver has provided this DMA address for the setup packet. The host controller driver should use this in preference to setup_packet, but the HCD may chose to ignore the address if it must copy the setup packet into internal structures. Therefore, setup_packet must always point to a valid buffer.

start_frame

Returns the initial frame for isochronous transfers.

number_of_packets

Lists the number of ISO transfer buffers.

interval

Specifies the polling interval for interrupt or isochronous transfers. The units are frames (milliseconds) for full and low speed devices, and microframes (1/8 millisecond) for highspeed ones.

error_count

Returns the number of ISO transfers that reported errors.

context

For use in completion functions. This normally points to request-specific driver context.

complete

Completion handler. This URB is passed as the parameter to the completion function. The completion function may then do what it likes with the URB, including resubmitting or freeing it.

iso_frame_desc[0]

Used to provide arrays of ISO transfer buffers and to collect the transfer status for each buffer.

Description

This structure identifies USB transfer requests. URBs must be allocated by calling `usb_alloc_urb` and freed with a call to `usb_free_urb`. Initialization may be done using various usb_fill_*_`urb` functions. URBs are submitted using `usb_submit_urb`, and pending requests may be canceled using `usb_unlink_urb` or `usb_kill_urb`.

Data Transfer Buffers

Normally drivers provide I/O buffers allocated with `kmalloc` or otherwise taken from the general page pool. That is provided by transfer_buffer (control requests also use setup_packet), and host controller drivers perform a dma mapping (and unmapping) for each buffer transferred. Those mapping operations can be expensive on some platforms (perhaps using a dma bounce buffer or talking to an IOMMU), although they're cheap on commodity x86 and ppc hardware.

Alternatively, drivers may pass the URB_NO_xxx_DMA_MAP transfer flags, which tell the host controller driver that no such mapping is needed since the device driver is DMA-aware. For example, a device driver might allocate a DMA buffer with `usb_buffer_alloc` or call `usb_buffer_map`. When these transfer flags are provided, host controller drivers will attempt to use the dma addresses found in the transfer_dma and/or setup_dma fields rather than determining a dma address themselves.

Note that transfer_buffer must still be set if the controller does not support DMA (as indicated by bus.uses_dma) and when talking to root hub. If you have to trasfer between highmem zone and the device on such controller, create a bounce buffer or bail out with an error. If transfer_buffer cannot be set (is in highmem) and the controller is DMA capable, assign NULL to it, so that usbmon knows not to use the value. The setup_packet must always be set, so it cannot be located in highmem.

Initialization

All URBs submitted must initialize the dev, pipe, transfer_flags (may be zero), and complete fields. All

URBs must also initialize transfer_buffer and transfer_buffer_length. They may provide the URB_SHORT_NOT_OK transfer flag, indicating that short reads are to be treated as errors; that flag is invalid for write requests.

Bulk URBs may use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers should always terminate with a short packet, even if it means adding an extra zero length packet.

Control URBs must provide a setup_packet. The setup_packet and transfer_buffer may each be mapped for DMA or not, independently of the other. The transfer_flags bits URB_NO_TRANSFER_DMA_MAP and URB_NO_SETUP_DMA_MAP indicate which buffers have already been mapped. URB_NO_SETUP_DMA_MAP is ignored for non-control URBs.

Interrupt URBs must provide an interval, saying how often (in milliseconds or, for highspeed devices, 125 microsecond units) to poll for transfers. After the URB has been submitted, the interval field reflects how the transfer was actually scheduled. The polling interval may be more frequent than requested. For example, some controllers have a maximum interval of 32 milliseconds, while others support intervals of up to 1024 milliseconds. Isochronous URBs also have transfer intervals. (Note that for isochronous endpoints, as well as high speed interrupt endpoints, the encoding of the transfer interval in the endpoint descriptor is logarithmic. Device drivers must convert that value to linear units themselves.)

Isochronous URBs normally use the URB_ISO_ASAP transfer flag, telling the host controller to schedule the transfer as soon as bandwidth utilization allows, and then set start_frame to reflect the actual frame selected during submission. Otherwise drivers must specify the start_frame and handle the case where the transfer can't begin then. However, drivers won't know how bandwidth is currently allocated, and while they can find the current frame using usb_get_current_frame_number () they can't know the range for that frame number. (Ranges for frame counter values are HC-specific, and can go from 256 to 65536 frames from “now”.)

Isochronous URBs have a different data transfer model, in part because the quality of service is only “best effort”. Callers provide specially allocated URBs, with number_of_packets worth of iso_frame_desc structures at the end. Each such packet is an individual ISO transfer. Isochronous URBs are normally queued, submitted by drivers to arrange that transfers are at least double buffered, and then explicitly resubmitted in completion handlers, so that data (such as audio or video) streams at as constant a rate as the host controller scheduler can support.

Completion Callbacks

The completion callback is made `in_interrupt`, and one of the first things that a completion handler should do is check the status field. The status field is provided for all URBs. It is used to report unlinked URBs, and status for all non-ISO transfers. It should not be examined before the URB is returned to the completion handler.

The context field is normally used to link URBs back to the relevant driver or request state.

When the completion callback is invoked for non-isochronous URBs, the actual_length field tells how many bytes were transferred. This field is updated even when the URB terminated with an error or was unlinked.

ISO transfer status is reported in the status and actual_length fields of the iso_frame_desc array, and the number of errors is reported in error_count. Completion callbacks for ISO transfers will normally (re)submit URBs to ensure a constant transfer rate.

Note that even fields marked “public” should not be touched by the driver when the urb is owned by the hcd, that is, since the call to `usb_submit_urb` till the entry into the completion routine.

Name

usb_fill_control_urb — initializes a control urb

Synopsis

```
void usb_fill_control_urb (urb,
                           dev,
                           pipe,
                           setup_packet,
                           transfer_buffer,
                           buffer_length,
                           complete_fn,
                           context);

struct urb *          urb;
struct usb_device *   dev;
unsigned int          pipe;
unsigned char *       setup_packet;
void *                transfer_buffer;
int                   buffer_length;
usb_complete_t        complete_fn;
void *                context;
```

Arguments

urb

pointer to the urb to initialize.

dev

pointer to the struct usb_device for this urb.

pipe

the endpoint pipe

setup_packet

pointer to the setup_packet buffer

transfer_buffer

pointer to the transfer buffer

buffer_length

length of the transfer buffer

complete_fn

pointer to the usb_complete_t function

context

what to set the urb context to.

Description

Initializes a control urb with the proper information needed to submit it to a device.

Name

usb_fill_bulk_urb — macro to help initialize a bulk urb

Synopsis

```
void usb_fill_bulk_urb (urb,
                        dev,
                        pipe,
                        transfer_buffer,
                        buffer_length,
                        complete_fn,
                        context);

struct urb *          urb;
struct usb_device *   dev;
unsigned int          pipe;
void *                transfer_buffer;
int                   buffer_length;
usb_complete_t        complete_fn;
void *                context;
```

Arguments

urb

pointer to the urb to initialize.

dev

pointer to the struct usb_device for this urb.

pipe

the endpoint pipe

transfer_buffer

pointer to the transfer buffer

buffer_length

length of the transfer buffer

complete_fn

pointer to the usb_complete_t function

context

what to set the urb context to.

Description

Initializes a bulk urb with the proper information needed to submit it to a device.

Name

usb_fill_int_urb — macro to help initialize a interrupt urb

Synopsis

```
void usb_fill_int_urb (urb,
                       dev,
                       pipe,
                       transfer_buffer,
                       buffer_length,
                       complete_fn,
                       context,
                       interval);

struct urb *         urb;
struct usb_device *  dev;
unsigned int         pipe;
void *               transfer_buffer;
int                  buffer_length;
usb_complete_t       complete_fn;
void *               context;
int                  interval;
```

Arguments

urb

pointer to the urb to initialize.

dev

pointer to the struct usb_device for this urb.

pipe

the endpoint pipe

transfer_buffer

pointer to the transfer buffer

buffer_length

length of the transfer buffer

complete_fn

pointer to the usb_complete_t function

context

what to set the urb context to.

interval

what to set the urb interval to, encoded like the endpoint descriptor's bInterval value.

Description

Initializes a interrupt urb with the proper information needed to submit it to a device. Note that high speed interrupt endpoints use a logarithmic encoding of the endpoint interval, and express polling intervals in microframes (eight per millisecond) rather than in frames (one per millisecond).

Name

usb_urb_dir_in — check if an URB describes an IN transfer

Synopsis

```
int usb_urb_dir_in (urb);

struct urb * urb;
```

Arguments

urb

URB to be checked

Description

Returns 1 if *urb* describes an IN transfer (device-to-host), otherwise 0.

Name

usb_urb_dir_out — check if an URB describes an OUT transfer

Synopsis

```
int usb_urb_dir_out (urb);

struct urb * urb;
```

Arguments

urb

URB to be checked

Description

Returns 1 if *urb* describes an OUT transfer (host-to-device), otherwise 0.

Name

struct usb_sg_request — support for scatter/gather I/O

Synopsis

```
struct usb_sg_request {
  int status;
  size_t bytes;
};
```

Members

status

zero indicates success, else negative errno bytes

counts bytes transferred.

Description

These requests are initialized using `usb_sg_init`, and then are used as request handles passed to `usb_sg_wait` or `usb_sg_cancel`. Most members of the request object aren't for driver access.

The status and bytecount values are valid only after `usb_sg_wait` returns. If the status is zero, then the bytecount matches the total from the request.

After an error completion, drivers may need to clear a halt condition on the endpoint.

Chapter 5. USB Core APIs

Table of Contents

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 handle 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.

Name

usb_init_urb — initializes a urb so that it can be used by a USB driver

Synopsis

```
void usb_init_urb (urb);

struct urb *  urb;
```

Arguments

urb

pointer to the urb to initialize

Description

Initializes a urb so that the USB subsystem can use it properly.

If a urb is created with a call to `usb_alloc_urb` it is not necessary to call this function. Only use this if you allocate the space for a struct urb on your own. If you call this function, be careful when freeing the memory for your urb that it is no longer in use by the USB core.

Only use this function if you _really_ understand what you are doing.

Name

usb_alloc_urb — creates a new urb for a USB driver to use

Synopsis

```
struct urb * usb_alloc_urb (iso_packets,
                            mem_flags);

int    iso_packets;
gfp_t  mem_flags;
```

Arguments

iso_packets

number of iso packets for this urb

mem_flags

the type of memory to allocate, see `kmalloc` for a list of valid options for this.

Description

Creates an urb for the USB driver to use, initializes a few internal structures, incrementes the usage counter, and returns a pointer to it.

If no memory is available, NULL is returned.

If the driver want to use this urb for interrupt, control, or bulk endpoints, pass '0' as the number of iso packets.

The driver must call `usb_free_urb` when it is finished with the urb.

Name

usb_free_urb — frees the memory used by a urb when all users of it are finished

Synopsis

```
void usb_free_urb (urb);

struct urb *  urb;
```

Arguments

urb

pointer to the urb to free, may be NULL

Description

Must be called when a user of a urb is finished with it. When the last user of the urb calls this function, the memory of the urb is freed.

Note

The transfer buffer associated with the urb is not freed unless the URB_FREE_BUFFER transfer flag is set.

Name

usb_get_urb — increments the reference count of the urb

Synopsis

```
struct urb * usb_get_urb (urb);

struct urb *  urb;
```

Arguments

urb

pointer to the urb to modify, may be NULL

Description

This must be called whenever a urb is transferred from a device driver to a host controller driver. This allows proper reference counting to happen for urbs.

A pointer to the urb with the incremented reference counter is returned.

Name

usb_anchor_urb — anchors an URB while it is processed

Synopsis

```
void usb_anchor_urb (urb,
                     anchor);

struct urb *          urb;
struct usb_anchor *   anchor;
```

Arguments

urb

pointer to the urb to anchor

anchor

pointer to the anchor

Description

This can be called to have access to URBs which are to be executed without bothering to track them

---

Name

usb_unanchor_urb — unanchors an URB

Synopsis

```
void usb_unanchor_urb (urb);

struct urb *  urb;
```

Arguments

urb

pointer to the urb to anchor

Description

Call this to stop the system keeping track of this URB

Name

usb_submit_urb — issue an asynchronous transfer request for an endpoint

Synopsis

```
int usb_submit_urb (urb,
                    mem_flags);

struct urb *  urb;
gfp_t         mem_flags;
```

Arguments

urb

pointer to the urb describing the request

mem_flags

the type of memory to allocate, see `kmalloc` for a list of valid options for this.

Description

This submits a transfer request, and transfers control of the URB describing that request to the USB subsystem. Request completion will be indicated later, asynchronously, by calling the completion handler. The three types of completion are success, error, and unlink (a software-induced fault, also called “request cancellation”).

URBs may be submitted in interrupt context.

The caller must have correctly initialized the URB before submitting it. Functions such as `usb_fill_bulk_urb` and `usb_fill_control_urb` are available to ensure that most fields are correctly initialized, for the particular kind of transfer, although they will not initialize any transfer flags.

Successful submissions return 0; otherwise this routine returns a negative error number. If the submission is successful, the `complete` callback from the URB will be called exactly once, when the USB core and Host Controller Driver (HCD) are finished with the URB. When the completion function is called, control of the URB is returned to the device driver which issued the request. The completion handler may then immediately free or reuse that URB.

With few exceptions, USB device drivers should never access URB fields provided by usbcore or the HCD until its `complete` is called. The exceptions relate to periodic transfer scheduling. For both interrupt and isochronous urbs, as part of successful URB submission urb->interval is modified to reflect the actual transfer period used (normally some power of two units). And for isochronous urbs, urb->start_frame is modified to reflect when the URB's transfers were scheduled to start. Not all isochronous transfer scheduling policies will work, but most host controller drivers should easily handle ISO queues going from now until 10-200 msec into the future.

For control endpoints, the synchronous `usb_control_msg` call is often used (in non-interrupt context) instead of this call. That is often used through convenience wrappers, for the requests that are standardized in the USB 2.0 specification. For bulk endpoints, a synchronous `usb_bulk_msg` call is available.

Request Queuing

URBs may be submitted to endpoints before previous ones complete, to minimize the impact of interrupt latencies and system overhead on data throughput. With that queuing policy, an endpoint's queue would never be empty. This is required for continuous isochronous data streams, and may also be required for some kinds of interrupt transfers. Such queuing also maximizes bandwidth utilization by letting USB controllers start work on later requests before driver software has finished the completion processing for earlier (successful) requests.

As of Linux 2.6, all USB endpoint transfer queues support depths greater than one. This was previously a HCD-specific behavior, except for ISO transfers. Non-isochronous endpoint queues are inactive during cleanup after faults (transfer errors or cancellation).

Reserved Bandwidth Transfers

Periodic transfers (interrupt or isochronous) are performed repeatedly, using the interval specified in the urb. Submitting the first urb to the endpoint reserves the bandwidth necessary to make those transfers. If the USB subsystem can't allocate sufficient bandwidth to perform the periodic request, submitting such a periodic request should fail.

For devices under xHCI, the bandwidth is reserved at configuration time, or when the alt setting is selected. If there is not enough bus bandwidth, the configuration/alt setting request will fail. Therefore, submissions to periodic endpoints on devices under xHCI should never fail due to bandwidth constraints.

Device drivers must explicitly request that repetition, by ensuring that some URB is always on the endpoint's queue (except possibly for short periods during completion callacks). When there is no longer an urb queued, the endpoint's bandwidth reservation is canceled. This means drivers can use their completion handlers to ensure they keep bandwidth they need, by reinitializing and resubmitting the just-completed urb until the driver longer needs that periodic bandwidth.

Memory Flags

The general rules for how to decide which mem_flags to use are the same as for kmalloc. There are four different possible values; GFP_KERNEL, GFP_NOFS, GFP_NOIO and GFP_ATOMIC.

GFP_NOFS is not ever used, as it has not been implemented yet.

GFP_ATOMIC is used when (a) you are inside a completion handler, an interrupt, bottom half, tasklet or timer, or (b) you are holding a spinlock or rwlock (does not apply to semaphores), or (c) current->state != TASK_RUNNING, this is the case only after you've changed it.

GFP_NOIO is used in the block io path and error handling of storage devices.

All other situations use GFP_KERNEL.

Some more specific rules for mem_flags can be inferred, such as (1) start_xmit, timeout, and receive methods of network drivers must use GFP_ATOMIC (they are called with a spinlock held); (2) queuecommand methods of scsi drivers must use GFP_ATOMIC (also called with a spinlock held); (3) If you use a kernel thread with a network driver you must use GFP_NOIO, unless (b) or (c) apply; (4) after you have done a `down` you can use GFP_KERNEL, unless (b) or (c) apply or your are in a storage driver's block io path; (5) USB probe and disconnect can use GFP_KERNEL unless (b) or (c) apply; and (6) changing firmware on a running storage or net device uses GFP_NOIO, unless b) or c) apply

---

Name

usb_unlink_urb — abort/cancel a transfer request for an endpoint

Synopsis

```
int usb_unlink_urb (urb);

struct urb * urb;
```

Arguments

urb

pointer to urb describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. URBs complete only once per submission, and may be canceled only once per submission. Successful cancellation means termination of *urb* will be expedited and the completion handler will be called with a status code indicating that the request has been canceled (rather than any other code).

Drivers should not call this routine or related routines, such as `usb_kill_urb` or `usb_unlink_anchored_urbs`, after their disconnect method has returned. The disconnect function should synchronize with a driver's I/O routines to insure that all URB-related activity has completed before it returns.

This request is always asynchronous. Success is indicated by returning -EINPROGRESS, at which time the URB will probably not yet have been given back to the device driver. When it is eventually called, the completion function will see *urb*->status == -ECONNRESET. Failure is indicated by `usb_unlink_urb` returning any other value. Unlinking will fail when *urb* is not currently "linked" (i.e., it was never submitted, or it was unlinked before, or the hardware is already finished with it), even if the completion handler has not yet run.

Unlinking and Endpoint Queues

[The behaviors and guarantees described below do not apply to virtual root hubs but only to endpoint queues for physical USB devices.]

Host Controller Drivers (HCDs) place all the URBs for a particular endpoint in a queue. Normally the queue advances as the controller hardware processes each request. But when an URB terminates with an error its queue generally stops (see below), at least until that URB's completion routine returns. It is guaranteed that a stopped queue will not restart until all its unlinked URBs have been fully retired, with their completion routines run, even if that's not until some time after the original completion handler returns. The same behavior and guarantee apply when an URB terminates because it was unlinked.

Bulk and interrupt endpoint queues are guaranteed to stop whenever an URB terminates with any sort of error, including -ECONNRESET, -ENOENT, and -EREMOTEIO. Control endpoint queues behave the same way except that they are not guaranteed to stop for -EREMOTEIO errors. Queues for isochronous endpoints are treated differently, because they must advance at fixed rates. Such queues do not stop when an URB encounters an error or is unlinked. An unlinked isochronous URB may leave a gap in the stream of packets; it is undefined whether such gaps can be filled in.

Note that early termination of an URB because a short packet was received will generate a -EREMOTEIO error if and only if the URB_SHORT_NOT_OK flag is set. By setting this flag, USB device drivers can build deep queues for large or complex bulk transfers and clean them up reliably after any sort of aborted transfer by unlinking all pending URBs at the first fault.

When a control URB terminates with an error other than -EREMOTEIO, it is quite likely that the status stage of the transfer will not take place.

Name

usb_kill_urb — cancel a transfer request and wait for it to finish

Synopsis

`void usb_kill_urb (urb);`

`struct urb * urb;`

Arguments

urb

pointer to URB describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and available for reuse. These features make this an ideal way to stop I/O in a `disconnect` callback or `close` function. If the request has not already finished or been unlinked the completion handler will see urb->status == -ENOENT.

While the routine is running, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

This routine may not be used in an interrupt context (such as a bottom half or a completion handler), or when holding a spinlock, or in other situations where the caller can't `schedule`.

This routine should not be called by a driver after its disconnect method has returned.

Name

usb_poison_urb — reliably kill a transfer and prevent further use of an URB

Synopsis

`void usb_poison_urb (urb);`

`struct urb * urb;`

Arguments

urb

pointer to URB describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and cannot be reused. These features make this an ideal way to stop I/O in a `disconnect` callback. If the request has not already finished or been unlinked the completion handler will see urb->status == -ENOENT.

After and while the routine runs, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

This routine may not be used in an interrupt context (such as a bottom half or a completion handler), or when holding a spinlock, or in other situations where the caller can't `schedule`.

This routine should not be called by a driver after its disconnect method has returned.

Name

usb_kill_anchored_urbs — cancel transfer requests en masse

Synopsis

```
void usb_kill_anchored_urbs (anchor);

struct usb_anchor * anchor;
```

Arguments

anchor

: anchor the requests are bound to

Description

this allows all outstanding URBs to be killed starting from the back of the queue

This routine should not be called by a driver after its disconnect method has returned.

Name

usb_poison_anchored_urbs — cease all traffic from an anchor

Synopsis

```
void usb_poison_anchored_urbs (anchor);

struct usb_anchor * anchor;
```

Arguments

anchor

anchor the requests are bound to

Description

this allows all outstanding URBs to be poisoned starting from the back of the queue. Newly added URBs will also be poisoned

This routine should not be called by a driver after its disconnect method has returned.

Name

usb_unpoison_anchored_urbs — let an anchor be used successfully again

Synopsis

`void` **`usb_unpoison_anchored_urbs`** `(`*anchor*`);`

`struct usb_anchor *` *`anchor`*`;`

Arguments

anchor

anchor the requests are bound to

Description

Reverses the effect of usb_poison_anchored_urbs the anchor can be used normally after it returns

---

Name

usb_unlink_anchored_urbs — asynchronously cancel transfer requests en masse

Synopsis

`void` **`usb_unlink_anchored_urbs`** `(`*anchor*`);`

`struct usb_anchor *` *`anchor`*`;`

Arguments

anchor

anchor the requests are bound to

Description

this allows all outstanding URBs to be unlinked starting from the back of the queue. This function is asynchronous. The unlinking is just tiggered. It may happen after this function has returned.

This routine should not be called by a driver after its disconnect method has returned.

Name

usb_wait_anchor_empty_timeout — wait for an anchor to be unused

Synopsis

```
int usb_wait_anchor_empty_timeout (anchor,
                                   timeout);

struct usb_anchor *  anchor;
unsigned int         timeout;
```

Arguments

anchor

the anchor you want to become unused

timeout

how long you are willing to wait in milliseconds

Description

Call this is you want to be sure all an anchor's URBs have finished

---

Name

usb_get_from_anchor — get an anchor's oldest urb

Synopsis

`struct urb * usb_get_from_anchor (anchor);`

`struct usb_anchor * anchor;`

Arguments

anchor

the anchor whose urb you want

Description

this will take the oldest urb from an anchor, unanchor and return it

---

Name

usb_scuttle_anchored_urbs — unanchor all an anchor's urbs

Synopsis

`void usb_scuttle_anchored_urbs (anchor);`

`struct usb_anchor * anchor;`

Arguments

anchor

the anchor whose urbs you want to unanchor

Description

use this to get rid of all an anchor's urbs

Name

usb_anchor_empty — is an anchor empty

Synopsis

`int usb_anchor_empty (anchor);`

`struct usb_anchor * anchor;`

Arguments

anchor

the anchor you want to query

Description

returns 1 if the anchor has no urbs associated with it

Name

usb_control_msg — Builds a control urb, sends it off and waits for completion

Synopsis

```
int usb_control_msg (dev,
                     pipe,
                     request,
                     requesttype,
                     value,
                     index,
                     data,
                     size,
                     timeout);

struct usb_device *  dev;
unsigned int         pipe;
__u8                 request;
__u8                 requesttype;
__u16                value;
__u16                index;
void *               data;
__u16                size;
int                  timeout;
```

Arguments

dev

pointer to the usb device to send the message to

pipe

endpoint “pipe” to send the message to

request

USB message request value

requesttype

USB message request type value

value

USB message value

index

USB message index value

data

pointer to the data to send

size

length in bytes of the data to send

timeout

time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple control message to a specified endpoint and waits for the message to complete, or timeout.

If successful, it returns the number of bytes transferred, otherwise a negative error number.

Don't use this function from within an interrupt context, like a bottom half handler. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb`. If a thread in your driver uses this call, make sure your `disconnect` method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Name

usb_interrupt_msg — Builds an interrupt urb, sends it off and waits for completion

Synopsis

`int` **`usb_interrupt_msg`** `(`*usb_dev*`,`

pipe,
data,
len,
actual_length,
timeout);

```
struct usb_device *  usb_dev;
unsigned int         pipe;
void *               data;
int                  len;
int *                actual_length;
int                  timeout;
```

Arguments

usb_dev

pointer to the usb device to send the message to

pipe

endpoint “pipe” to send the message to

data

pointer to the data to send

len

length in bytes of the data to send

actual_length

pointer to a location to put the actual length transferred in bytes

timeout

time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple interrupt message to a specified endpoint and waits for the message to complete, or timeout.

If successful, it returns 0, otherwise a negative error number. The number of actual bytes transferred will be stored in the actual_length paramater.

Don't use this function from within an interrupt context, like a bottom half handler. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb` If a thread in your driver uses this call, make sure your `disconnect` method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Name

usb_bulk_msg — Builds a bulk urb, sends it off and waits for completion

Synopsis

```
int usb_bulk_msg (usb_dev,
                  pipe,
                  data,
                  len,
                  actual_length,
                  timeout);

struct usb_device *  usb_dev;
unsigned int         pipe;
void *               data;
int                  len;
int *                actual_length;
int                  timeout;
```

Arguments

usb_dev

pointer to the usb device to send the message to

pipe

endpoint “pipe” to send the message to

data

pointer to the data to send

len

length in bytes of the data to send

actual_length

pointer to a location to put the actual length transferred in bytes

timeout

time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple bulk message to a specified endpoint and waits for the message to complete, or timeout.

If successful, it returns 0, otherwise a negative error number. The number of actual bytes transferred will be stored in the actual_length paramater.

Don't use this function from within an interrupt context, like a bottom half handler. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb` If a thread in your driver uses this call, make sure your `disconnect` method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Because there is no `usb_interrupt_msg` and no USBDEVFS_INTERRUPT ioctl, users are forced to abuse this routine by using it to submit URBs for interrupt endpoints. We will take the liberty of creating an interrupt URB (with the default interval) if the target is an interrupt endpoint.

Name

usb_sg_init — initializes scatterlist-based bulk/interrupt I/O request

Synopsis

```
int usb_sg_init (io,
                 dev,
                 pipe,
                 period,
                 sg,
                 nents,
                 length,
                 mem_flags);

struct usb_sg_request *  io;
struct usb_device *      dev;
unsigned                 pipe;
unsigned                 period;
struct scatterlist *     sg;
int                      nents;
size_t                   length;
gfp_t                    mem_flags;
```

Arguments

io

request block being initialized. until `usb_sg_wait` returns, treat this as a pointer to an opaque block of memory,

dev

the usb device that will send or receive the data

pipe

endpoint “pipe” used to transfer the data

period

polling rate for interrupt endpoints, in frames or (for high speed endpoints) microframes; ignored for bulk

sg

scatterlist entries

nents

how many entries in the scatterlist

length

how many bytes to send from the scatterlist, or zero to send every byte identified in the list.

mem_flags

SLAB_* flags affecting memory allocations in this call

Description

Returns zero for success, else a negative errno value. This initializes a scatter/gather request, allocating resources such as I/O mappings and urb memory (except maybe memory used by USB controller drivers).

The request must be issued using `usb_sg_wait`, which waits for the I/O to complete (or to be canceled) and then cleans up all resources allocated by `usb_sg_init`.

The request may be canceled with `usb_sg_cancel`, either before or after `usb_sg_wait` is called.

Name

usb_sg_wait — synchronously execute scatter/gather request

Synopsis

void **usb_sg_wait** (*io*);

struct usb_sg_request * *io*;

Arguments

io

request block handle, as initialized with `usb_sg_init`. some fields become accessible when this call returns.

Context

!in_interrupt ()

Description

This function blocks until the specified I/O operation completes. It leverages the grouping of the related I/O requests to get good transfer rates, by queueing the requests. At higher speeds, such queuing can significantly improve USB throughput.

There are three kinds of completion for this function. (1) success, where io->status is zero. The number of io->bytes transferred is as requested. (2) error, where io->status is a negative errno value. The number of io->bytes transferred before the error is usually less than requested, and can be nonzero. (3) cancellation, a type of error with status -ECONNRESET that is initiated by `usb_sg_cancel`.

When this function returns, all memory allocated through `usb_sg_init` or this call will have been freed. The request block parameter may still be passed to `usb_sg_cancel`, or it may be freed. It could also be reinitialized and then reused.

Data Transfer Rates

Bulk transfers are valid for full or high speed endpoints. The best full speed data rate is 19 packets of 64 bytes each per frame, or 1216 bytes per millisecond. The best high speed data rate is 13 packets of 512 bytes each per microframe, or 52 KBytes per millisecond.

The reason to use interrupt transfers through this API would most likely be to reserve high speed bandwidth, where up to 24 KBytes per millisecond could be transferred. That capability is less useful for low or full speed interrupt endpoints, which allow at most one packet per millisecond, of at most 8 or 64 bytes (respectively).

It is not necessary to call this function to reserve bandwidth for devices under an xHCI host controller, as the bandwidth is reserved when the configuration or interface alt setting is selected.

Name

usb_sg_cancel — stop scatter/gather i/o issued by `usb_sg_wait`

Synopsis

```
void usb_sg_cancel (io);

struct usb_sg_request *  io;
```

Arguments

io

request block, initialized with `usb_sg_init`

Description

This stops a request after it has been started by `usb_sg_wait`. It can also prevents one initialized by `usb_sg_init` from starting, so that call just frees resources allocated to the request.

Name

usb_get_descriptor — issues a generic GET_DESCRIPTOR request

Synopsis

```
int usb_get_descriptor (dev,
                        type,
                        index,
                        buf,
                        size);

struct usb_device *  dev;
unsigned char        type;
unsigned char        index;
void *               buf;
int                  size;
```

Arguments

dev

the device whose descriptor is being retrieved

type

the descriptor type (USB_DT_*)

index

the number of the descriptor

buf

where to put the descriptor

size

how big is “buf”?

Context

!in_interrupt ()

Description

Gets a USB descriptor. Convenience functions exist to simplify getting some types of descriptors. Use `usb_get_string` or `usb_string` for USB_DT_STRING. Device (USB_DT_DEVICE) and configuration descriptors (USB_DT_CONFIG) are part of the device structure. In addition to a number of USB-standard descriptors, some devices also use class-specific or vendor-specific descriptors.

This call is synchronous, and may not be used in an interrupt context.

Returns the number of bytes received on success, or else the status code returned by the underlying `usb_control_msg` call.

Name

usb_string — returns UTF-8 version of a string descriptor

Synopsis

```
int usb_string (dev,
                index,
                buf,
                size);

struct usb_device *  dev;
int                  index;
char *               buf;
size_t               size;
```

Arguments

dev

the device whose string descriptor is being retrieved *index*

    the number of the descriptor

buf

    where to put the string

size

    how big is “buf”?

Context

!in_interrupt ()

Description

This converts the UTF-16LE encoded strings returned by devices, from `usb_get_string_descriptor`, to null-terminated UTF-8 encoded ones that are more usable in most kernel contexts. Note that this function chooses strings in the first language supported by the device.

This call is synchronous, and may not be used in an interrupt context.

Returns length of the string (>= 0) or usb_control_msg status (< 0).

Name

usb_get_status — issues a GET_STATUS call

Synopsis

```
int usb_get_status (dev,
                    type,
                    target,
                    data);

struct usb_device *  dev;
int                  type;
int                  target;
void *               data;
```

Arguments

dev

    the device whose status is being checked *type*

USB_RECIP_*; for device, interface, or endpoint

target

zero (for device), else interface or endpoint number

data

pointer to two bytes of bitmap data

Context

!in_interrupt ()

Description

Returns device, interface, or endpoint status. Normally only of interest to see if the device is self powered, or has enabled the remote wakeup facility; or whether a bulk or interrupt endpoint is halted ("stalled").

Bits in these status bitmaps are set using the SET_FEATURE request, and cleared using the CLEAR_FEATURE request. The `usb_clear_halt` function should be used to clear halt ("stall") status.

This call is synchronous, and may not be used in an interrupt context.

Returns the number of bytes received on success, or else the status code returned by the underlying `usb_control_msg` call.

Name

usb_clear_halt — tells device to clear endpoint halt/stall condition

Synopsis

```
int usb_clear_halt (dev,
                    pipe);

struct usb_device *  dev;
int                  pipe;
```

Arguments

dev

device whose endpoint is halted

pipe

endpoint “pipe” being cleared

Context

!in_interrupt ()

Description

This is used to clear halt conditions for bulk and interrupt endpoints, as reported by URB completion status. Endpoints that are halted are sometimes referred to as being “stalled”. Such endpoints are unable to transmit or receive data until the halt status is cleared. Any URBs queued for such an endpoint should normally be unlinked by the driver before clearing the halt condition, as described in sections 5.7.5 and 5.8.5 of the USB 2.0 spec.

Note that control and isochronous endpoints don't halt, although control endpoints report “protocol stall” (for unsupported requests) using the same status code used to report a true stall.

This call is synchronous, and may not be used in an interrupt context.

Returns zero on success, or else the status code returned by the underlying `usb_control_msg` call.

Name

usb_reset_endpoint — Reset an endpoint's state.

Synopsis

```
void usb_reset_endpoint (dev,
                         epaddr);

struct usb_device *  dev;
unsigned int         epaddr;
```

Arguments

dev

the device whose endpoint is to be reset

epaddr

the endpoint's address. Endpoint number for output, endpoint number + USB_DIR_IN for input

Description

Resets any host-side endpoint state such as the toggle bit, sequence number or current window.

Name

usb_set_interface — Makes a particular alternate setting be current

Synopsis

```
int usb_set_interface (dev,
                       interface,
                       alternate);

struct usb_device *  dev;
int                  interface;
int                  alternate;
```

Arguments

dev

    the device whose interface is being updated

interface

    the interface being updated

alternate

    the setting being chosen.

Context

!in_interrupt ()

Description

This is used to enable data transfers on interfaces that may not be enabled by default. Not all devices support such configurability. Only the driver bound to an interface may change its setting.

Within any given configuration, each interface may have several alternative settings. These are often used to control levels of bandwidth consumption. For example, the default setting for a high speed interrupt endpoint may not send more than 64 bytes per microframe, while interrupt transfers of up to 3KBytes per microframe are legal. Also, isochronous endpoints may never be part of an interface's default setting. To access such bandwidth, alternate interface settings must be made current.

Note that in the Linux USB subsystem, bandwidth associated with an endpoint in a given alternate setting is not reserved until an URB is submitted that needs that bandwidth. Some other operating systems allocate bandwidth early, when a configuration is chosen.

This call is synchronous, and may not be used in an interrupt context. Also, drivers must not change altsettings while urbs are scheduled for endpoints in that interface; all such urbs must first be completed (perhaps forced by unlinking).

Returns zero on success, or else the status code returned by the underlying `usb_control_msg` call.

Name

usb_reset_configuration — lightweight device reset

Synopsis

```
int usb_reset_configuration (dev);

struct usb_device *  dev;
```

Arguments

dev

: the device whose configuration is being reset

Description

This issues a standard SET_CONFIGURATION request to the device using the current configuration. The effect is to reset most USB-related state in the device, including interface altsettings (reset to zero), endpoint halts (cleared), and endpoint state (only for bulk and interrupt endpoints). Other usbcore state is unchanged, including bindings of usb device drivers to interfaces.

Because this affects multiple interfaces, avoid using this with composite (multi-interface) devices. Instead, the driver for each interface may use `usb_set_interface` on the interfaces it claims. Be careful though; some devices don't support the SET_INTERFACE request, and others won't reset all the interface state (notably endpoint state). Resetting the whole configuration would affect other drivers' interfaces.

The caller must own the device lock.

Returns zero on success, else a negative error code.

Name

usb_driver_set_configuration — Provide a way for drivers to change device configurations

Synopsis

```
int usb_driver_set_configuration (udev,
                                  config);

struct usb_device *  udev;
int                  config;
```

Arguments

udev

the device whose configuration is being updated

config

the configuration being chosen.

Context

In process context, must be able to sleep

Description

Device interface drivers are not allowed to change device configurations. This is because changing configurations will destroy the interface the driver is bound to and create new ones; it would be like a floppy-disk driver telling the computer to replace the floppy-disk drive with a tape drive!

Still, in certain specialized circumstances the need may arise. This routine gets around the normal restrictions by using a work thread to submit the change-config request.

Returns 0 if the request was succesfully queued, error code otherwise. The caller has no way to know whether the queued request will eventually succeed.

Name

usb_register_dev — register a USB device, and ask for a minor number

Synopsis

```
int usb_register_dev (intf,
                      class_driver);

struct usb_interface *     intf;
struct usb_class_driver *  class_driver;
```

Arguments

intf

pointer to the usb_interface that is being registered

class_driver

pointer to the usb_class_driver for this device

Description

This should be called by all USB drivers that use the USB major number. If CONFIG_USB_DYNAMIC_MINORS is enabled, the minor number will be dynamically allocated out of the list of available ones. If it is not enabled, the minor number will be based on the next available free minor, starting at the class_driver->minor_base.

This function also creates a usb class device in the sysfs tree.

`usb_deregister_dev` must be called when the driver is done with the minor numbers given out by this function.

Returns -EINVAL if something bad happens with trying to register a device, and 0 on success.

Name

usb_deregister_dev — deregister a USB device's dynamic minor.

Synopsis

```
void usb_deregister_dev (intf,
                         class_driver);

struct usb_interface *     intf;
struct usb_class_driver *  class_driver;
```

Arguments

intf

pointer to the usb_interface that is being deregistered

class_driver

pointer to the usb_class_driver for this device

Description

Used in conjunction with `usb_register_dev`. This function is called when the USB driver is finished with the minor numbers gotten from a call to `usb_register_dev` (usually when the device is disconnected from the system.)

This function also removes the usb class device from the sysfs tree.

This should be called by all drivers that use the USB major number.

Name

usb_driver_claim_interface — bind a driver to an interface

Synopsis

```
int usb_driver_claim_interface (driver,
                                iface,
                                priv);

struct usb_driver *     driver;
struct usb_interface *  iface;
void *                  priv;
```

Arguments

driver

: the driver to be bound

iface

: the interface to which it will be bound; must be in the usb device's active configuration

priv

: driver data associated with that interface

Description

This is used by usb device drivers that need to claim more than one interface on a device when probing (audio and acm are current examples). No device driver should directly modify internal usb_interface or usb_device structure members.

Few drivers should need to use this routine, since the most natural way to bind to an interface is to return the private data from the driver's `probe` method.

Callers must own the device lock, so driver `probe` entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

Name

usb_driver_release_interface — unbind a driver from an interface

Synopsis

```
void usb_driver_release_interface (driver,
                                   iface);

struct usb_driver *    driver;
struct usb_interface * iface;
```

Arguments

driver

the driver to be unbound

iface

the interface from which it will be unbound

Description

This can be used by drivers to release an interface without waiting for their `disconnect` methods to be called. In typical cases this also causes the driver `disconnect` method to be called.

This call is synchronous, and may not be used in an interrupt context. Callers must own the device lock, so driver `disconnect` entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

Name

usb_match_id — find first usb_device_id matching device or interface

Synopsis

```
const struct usb_device_id * usb_match_id (interface,
                                           id);

struct usb_interface *       interface;
const struct usb_device_id * id;
```

Arguments

interface

the interface of interest

id

array of usb_device_id structures, terminated by zero entry

Description

usb_match_id searches an array of usb_device_id's and returns the first one matching the device or interface, or null. This is used when binding (or rebinding) a driver to an interface. Most USB device drivers will use this indirectly, through the usb core, but some layered driver frameworks use it directly. These device tables are exported with MODULE_DEVICE_TABLE, through modutils, to support the driver loading functionality of USB hotplugging.

What Matches

The “match_flags” element in a usb_device_id controls which members are used. If the corresponding bit is set, the value in the device_id must match its corresponding member in the device or interface descriptor, or else the device_id does not match.

“driver_info” is normally used only by device drivers, but you can create a wildcard “matches anything” usb_device_id as a driver's “modules.usbmap” entry if you provide an id with only a nonzero “driver_info” field. If you do this, the USB device driver's `probe` routine should use additional intelligence to decide whether to bind to the specified interface.

What Makes Good usb_device_id Tables

The match algorithm is very simple, so that intelligence in driver selection must come from smart driver id records. Unless you have good reasons to use another selection policy, provide match elements only in related groups, and order match specifiers from specific to general. Use the macros provided for that purpose if you can.

The most specific match specifiers use device descriptor data. These are commonly used with product-specific matches; the USB_DEVICE macro lets you provide vendor and product IDs, and you can also match against ranges of product revisions. These are widely used for devices with application or vendor specific bDeviceClass values.

Matches based on device class/subclass/protocol specifications are slightly more general; use the USB_DEVICE_INFO macro, or its siblings. These are used with single-function devices where bDeviceClass doesn't specify that each interface has its own class.

Matches based on interface class/subclass/protocol are the most general; they let drivers bind to any interface on a multiple-function device. Use the USB_INTERFACE_INFO macro, or its siblings, to match class-per-interface style devices (as recorded in bInterfaceClass).

Note that an entry created by USB_INTERFACE_INFO won't match any interface if the device class is set to Vendor-Specific. This is deliberate; according to the USB spec the meanings of the interface class/subclass/protocol for these devices are also vendor-specific, and hence matching against a standard product class wouldn't work anyway. If you really want to use an interface-based match for such a device, create a match record that also specifies the vendor ID. (Unforunately there isn't a standard macro for creating records like this.)

Within those groups, remember that not all combinations are meaningful. For example, don't give a product version range without vendor and product IDs; or specify a protocol without its associated class and subclass.

Name

usb_register_device_driver — register a USB device (not interface) driver

Synopsis

```
int usb_register_device_driver (new_udriver,
                                owner);

struct usb_device_driver *  new_udriver;
struct module *             owner;
```

Arguments

new_udriver

USB operations for the device driver

owner

module owner of this driver.

Description

Registers a USB device driver with the USB core. The list of unattached devices will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized devices. Returns a negative error code on failure and 0 on success.

Name

usb_deregister_device_driver — unregister a USB device (not interface) driver

Synopsis

```
void usb_deregister_device_driver (udriver);

struct usb_device_driver *  udriver;
```

Arguments

udriver

USB operations of the device driver to unregister

Context

must be able to sleep

Description

Unlinks the specified driver from the internal USB driver list.

Name

usb_register_driver — register a USB interface driver

Synopsis

```
int usb_register_driver (new_driver,
                         owner,
                         mod_name);

struct usb_driver *  new_driver;
struct module *      owner;
const char *         mod_name;
```

Arguments

new_driver

USB operations for the interface driver

owner

module owner of this driver.

mod_name

module name string

Description

Registers a USB interface driver with the USB core. The list of unattached interfaces will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized interfaces. Returns a negative error code on failure and 0 on success.

NOTE

if you want your driver to use the USB major number, you must call `usb_register_dev` to enable that functionality. This function no longer takes care of that.

Name

usb_deregister — unregister a USB interface driver

Synopsis

`void` **`usb_deregister`** `(`*driver*`);`

`struct usb_driver *` *`driver`*`;`

Arguments

`driver`

USB operations of the interface driver to unregister

Context

must be able to sleep

Description

Unlinks the specified driver from the internal USB driver list.

NOTE

If you called `usb_register_dev`, you still need to call `usb_deregister_dev` to clean up your driver's allocated minor numbers, this * call will no longer do it for you.

Name

usb_autopm_put_interface — decrement a USB interface's PM-usage counter

Synopsis

`void` **`usb_autopm_put_interface`** `(`*intf*`);`

`struct usb_interface *` *`intf`*`;`

Arguments

intf

the usb_interface whose counter should be decremented

Description

This routine should be called by an interface driver when it is finished using *intf* and wants to allow it to autosuspend. A typical example would be a character-device driver when its device file is closed.

The routine decrements *intf*'s usage counter. When the counter reaches 0, a delayed autosuspend request for *intf*'s device is queued. When the delay expires, if *intf*->pm_usage_cnt is still <= 0 along with all the other usage counters for the sibling interfaces and *intf*'s usb_device, the device and all its interfaces will be autosuspended.

Note that *intf*->pm_usage_cnt is owned by the interface driver. The core will not change its value other than the increment and decrement in usb_autopm_get_interface and usb_autopm_put_interface. The driver may use this simple counter-oriented discipline or may set the value any way it likes.

If the driver has set *intf*->needs_remote_wakeup then autosuspend will take place only if the device's remote-wakeup facility is enabled.

Suspend method calls queued by this routine can arrive at any time while *intf* is resumed and its usage counter is equal to 0. They are not protected by the usb_device's lock but only by its pm_mutex. Drivers must provide their own synchronization.

This routine can run only in process context.

Name

usb_autopm_put_interface_async — decrement a USB interface's PM-usage counter

Synopsis

```
void usb_autopm_put_interface_async (intf);

struct usb_interface * intf;
```

Arguments

intf

the usb_interface whose counter should be decremented

Description

This routine does essentially the same thing as `usb_autopm_put_interface`: it decrements *intf*'s usage counter and queues a delayed autosuspend request if the counter is <= 0. The difference is that it does not acquire the device's pm_mutex; callers must handle all synchronization issues themselves.

Typically a driver would call this routine during an URB's completion handler, if no more URBs were pending.

This routine can run in atomic context.

Name

usb_autopm_get_interface — increment a USB interface's PM-usage counter

Synopsis

`int` **`usb_autopm_get_interface`** `(`*intf*`);`

`struct usb_interface *` *`intf`*`;`

Arguments

`intf`

> the usb_interface whose counter should be incremented

Description

This routine should be called by an interface driver when it wants to use *`intf`* and needs to guarantee that it is not suspended. In addition, the routine prevents *`intf`* from being autosuspended subsequently. (Note that this will not prevent suspend events originating in the PM core.) This prevention will persist until `usb_autopm_put_interface` is called or *`intf`* is unbound. A typical example would be a character-device driver when its device file is opened.

The routine increments *`intf`*'s usage counter. (However if the autoresume fails then the counter is re-decremented.) So long as the counter is greater than 0, autosuspend will not be allowed for *`intf`* or its usb_device. When the driver is finished using *`intf`* it should call `usb_autopm_put_interface` to decrement the usage counter and queue a delayed autosuspend request (if the counter is <= 0).

Note that *`intf`*->pm_usage_cnt is owned by the interface driver. The core will not change its value other than the increment and decrement in usb_autopm_get_interface and usb_autopm_put_interface. The driver may use this simple counter-oriented discipline or may set the value any way it likes.

Resume method calls generated by this routine can arrive at any time while *`intf`* is suspended. They are not protected by the usb_device's lock but only by its pm_mutex. Drivers must provide their own synchronization.

This routine can run only in process context.

Name

usb_autopm_get_interface_async — increment a USB interface's PM-usage counter

Synopsis

```
int usb_autopm_get_interface_async (intf);

struct usb_interface *  intf;
```

Arguments

intf

the usb_interface whose counter should be incremented

Description

This routine does much the same thing as `usb_autopm_get_interface`: it increments *intf*'s usage counter and queues an autoresume request if the result is > 0. The differences are that it does not acquire the device's pm_mutex (callers must handle all synchronization issues themselves), and it does not autoresume the device directly (it only queues a request). After a successful call, the device will generally not yet be resumed.

This routine can run in atomic context.

Name

usb_autopm_set_interface — set a USB interface's autosuspend state

Synopsis

```
int usb_autopm_set_interface (intf);

struct usb_interface *  intf;
```

Arguments

intf

the usb_interface whose state should be set

Description

This routine sets the autosuspend state of *intf*'s device according to *intf*'s usage counter, which the caller must have set previously. If the counter is <= 0, the device is autosuspended (if it isn't already suspended and if nothing else prevents the autosuspend). If the counter is > 0, the device is autoresumed (if it isn't already awake).

Name

usb_ifnum_to_if — get the interface object with a given interface number

Synopsis

```
struct usb_interface * usb_ifnum_to_if (dev,
                                        ifnum);

const struct usb_device *  dev;
unsigned                   ifnum;
```

Arguments

dev

the device whose current configuration is considered

ifnum

the desired interface

Description

This walks the device descriptor for the currently active configuration and returns a pointer to the interface with that particular interface number, or null.

Note that configuration descriptors are not required to assign interface numbers sequentially, so that it would be incorrect to assume that the first interface in that descriptor corresponds to interface zero. This routine helps device drivers avoid such mistakes. However, you should make sure that you do the right thing with any alternate settings available for this interfaces.

Don't call this function unless you are bound to one of the interfaces on this device or you have locked the device!

Name

usb_altnum_to_altsetting — get the altsetting structure with a given alternate setting number.

Synopsis

```
struct usb_host_interface * usb_altnum_to_altsetting (intf,
```

```
                                                             altnum);

const struct usb_interface *  intf;
unsigned int                  altnum;
```

Arguments

intf

the interface containing the altsetting in question

altnum

the desired alternate setting number

Description

This searches the altsetting array of the specified interface for an entry with the correct bAlternateSetting value and returns a pointer to that entry, or null.

Note that altsettings need not be stored sequentially by number, so it would be incorrect to assume that the first altsetting entry in the array corresponds to altsetting zero. This routine helps device drivers avoid such mistakes.

Don't call this function unless you are bound to the intf interface or you have locked the device!

Name

usb_find_interface — find usb_interface pointer for driver and device

Synopsis

```
struct usb_interface * usb_find_interface (drv,
                                            minor);

struct usb_driver *  drv;
int                  minor;
```

Arguments

drv

the driver whose current configuration is considered

minor

the minor number of the desired device

Description

This walks the driver device list and returns a pointer to the interface with the matching minor. Note, this only works for devices that share the USB major number.

Name

usb_get_dev — increments the reference count of the usb device structure

Synopsis

`struct usb_device * usb_get_dev (dev);`

`struct usb_device * dev;`

Arguments

dev

the device being referenced

Description

Each live reference to a device should be refcounted.

Drivers for USB interfaces should normally record such references in their `probe` methods, when they bind to an interface, and release them by calling `usb_put_dev`, in their `disconnect` methods.

A pointer to the device with the incremented reference counter is returned.

Name

usb_put_dev — release a use of the usb device structure

Synopsis

`void usb_put_dev (dev);`

`struct usb_device * dev;`

Arguments

dev

device that's been disconnected

Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

Name

usb_get_intf — increments the reference count of the usb interface structure

Synopsis

```
struct usb_interface * usb_get_intf (intf);

struct usb_interface *  intf;
```

Arguments

intf

the interface being referenced

Description

Each live reference to a interface must be refcounted.

Drivers for USB interfaces should normally record such references in their `probe` methods, when they bind to an interface, and release them by calling `usb_put_intf`, in their `disconnect` methods.

A pointer to the interface with the incremented reference counter is returned.

Name

usb_put_intf — release a use of the usb interface structure

Synopsis

```
void usb_put_intf (intf);

struct usb_interface *  intf;
```

Arguments

intf

interface that's been decremented

Description

Must be called when a user of an interface is finished with it. When the last user of the interface calls this function, the memory of the interface is freed.

Name

usb_lock_device_for_reset — cautiously acquire the lock for a usb device structure

Synopsis

```
int usb_lock_device_for_reset (udev,
                               iface);

struct usb_device *           udev;
const struct usb_interface *  iface;
```

Arguments

udev

device that's being locked

iface

interface bound to the driver making the request (optional)

Description

Attempts to acquire the device lock, but fails if the device is NOTATTACHED or SUSPENDED, or if iface is specified and the interface is neither BINDING nor BOUND. Rather than sleeping to wait for the lock, the routine polls repeatedly. This is to prevent deadlock with disconnect; in some drivers (such as usb-storage) the `disconnect` or `suspend` method will block waiting for a device reset to complete.

Returns a negative error code for failure, otherwise 0.

Name

usb_get_current_frame_number — return current bus frame number

Synopsis

```
int usb_get_current_frame_number (dev);

struct usb_device * dev;
```

Arguments

dev

the device whose bus is being queried

Description

Returns the current frame number for the USB host controller used with the given USB device. This can be used when scheduling isochronous requests.

Note that different kinds of host controller have different “scheduling horizons”. While one type might support scheduling only 32 frames into the future, others could support scheduling up to 1024 frames into the future.

Name

usb_buffer_alloc — allocate dma-consistent buffer for URB_NO_xxx_DMA_MAP

Synopsis

```
void * usb_buffer_alloc (dev,
                         size,
                         mem_flags,
                         dma);

struct usb_device * dev;
size_t              size;
gfp_t               mem_flags;
dma_addr_t *        dma;
```

Arguments

dev

device the buffer will be used with

size

requested buffer size

mem_flags

affect whether allocation may block

dma

used to return DMA address of buffer

Description

Return value is either null (indicating no buffer could be allocated), or the cpu-space pointer to a buffer that may be used to perform DMA to the specified device. Such cpu-space buffers are returned along with the DMA address (through the pointer provided).

These buffers are used with URB_NO_xxx_DMA_MAP set in urb->transfer_flags to avoid behaviors like using “DMA bounce buffers”, or thrashing IOMMU hardware during URB completion/resubmit. The implementation varies between platforms, depending on details of how DMA will work to this device. Using these buffers also eliminates cacheline sharing problems on architectures where CPU caches are not DMA-coherent. On systems without bus-snooping caches, these buffers are uncached.

When the buffer is no longer used, free it with `usb_buffer_free`.

Name

usb_buffer_free — free memory allocated with `usb_buffer_alloc`

Synopsis

```
void usb_buffer_free (dev,
                      size,
                      addr,
                      dma);

struct usb_device *  dev;
size_t               size;
void *               addr;
dma_addr_t           dma;
```

Arguments

dev

device the buffer was used with

size

requested buffer size

addr

CPU address of buffer

dma

DMA address of buffer

Description

This reclaims an I/O buffer, letting it be reused. The memory must have been allocated using `usb_buffer_alloc`, and the parameters must match those provided in that allocation request.

Name

usb_buffer_map — create DMA mapping(s) for an urb

Synopsis

```
struct urb * usb_buffer_map (urb);

struct urb *  urb;
```

Arguments

urb

urb whose transfer_buffer/setup_packet will be mapped

Description

Return value is either null (indicating no buffer could be mapped), or the parameter. URB_NO_TRANSFER_DMA_MAP and URB_NO_SETUP_DMA_MAP are added to urb->transfer_flags if the operation succeeds. If the device is connected to this system through a non-DMA controller, this operation always succeeds.

This call would normally be used for an urb which is reused, perhaps as the target of a large periodic transfer, with `usb_buffer_dmasync` calls to synchronize memory and dma state.

Reverse the effect of this call with `usb_buffer_unmap`.

Name

usb_buffer_dmasync — synchronize DMA and CPU view of buffer(s)

Synopsis

```
void usb_buffer_dmasync (urb);
```

`struct urb * urb;`

Arguments

urb

urb whose transfer_buffer/setup_packet will be synchronized

Name

usb_buffer_unmap — free DMA mapping(s) for an urb

Synopsis

```
void usb_buffer_unmap (urb);

struct urb * urb;
```

Arguments

urb

urb whose transfer_buffer will be unmapped

Description

Reverses the effect of `usb_buffer_map`.

Name

usb_buffer_map_sg — create scatterlist DMA mapping(s) for an endpoint

Synopsis

```
int usb_buffer_map_sg (dev,
                       is_in,
                       sg,
                       nents);

const struct usb_device *  dev;
int                        is_in;
struct scatterlist *       sg;
int                        nents;
```

Arguments

dev

device to which the scatterlist will be mapped

is_in

mapping transfer direction

sg

the scatterlist to map

nents

the number of entries in the scatterlist

Description

Return value is either < 0 (indicating no buffers could be mapped), or the number of DMA mapping array entries in the scatterlist.

The caller is responsible for placing the resulting DMA addresses from the scatterlist into URB transfer buffer pointers, and for setting the URB_NO_TRANSFER_DMA_MAP transfer flag in each of those URBs.

Top I/O rates come from queuing URBs, instead of waiting for each one to complete before starting the next I/O. This is particularly easy to do with scatterlists. Just allocate and submit one URB for each DMA mapping entry returned, stopping on the first error or when all succeed. Better yet, use the usb_sg_*() calls, which do that (and more) for you.

This call would normally be used when translating scatterlist requests, rather than `usb_buffer_map`, since on some hardware (with IOMMUs) it may be able to coalesce mappings for improved I/O efficiency.

Reverse the effect of this call with `usb_buffer_unmap_sg`.

Name

usb_buffer_dmasync_sg — synchronize DMA and CPU view of scatterlist buffer(s)

Synopsis

```
void usb_buffer_dmasync_sg (dev,
                            is_in,
                            sg,
                            n_hw_ents);
```

```
const struct usb_device *  dev;
int                        is_in;
struct scatterlist *       sg;
int                        n_hw_ents;
```

Arguments

dev

device to which the scatterlist will be mapped

is_in

mapping transfer direction

sg

the scatterlist to synchronize

n_hw_ents

the positive return value from usb_buffer_map_sg

Description

Use this when you are re-using a scatterlist's data buffers for another USB request.

Name

usb_buffer_unmap_sg — free DMA mapping(s) for a scatterlist

Synopsis

```
void usb_buffer_unmap_sg (dev,
                          is_in,
                          sg,
                          n_hw_ents);

const struct usb_device *  dev;
int                        is_in;
struct scatterlist *       sg;
int                        n_hw_ents;
```

Arguments

dev

device to which the scatterlist will be mapped *is_in*

mapping transfer direction

sg

the scatterlist to unmap

n_hw_ents

the positive return value from usb_buffer_map_sg

Description

Reverses the effect of `usb_buffer_map_sg`.

Name

usb_hub_clear_tt_buffer — clear control/bulk TT state in high speed hub

Synopsis

```
int usb_hub_clear_tt_buffer (urb);

struct urb *  urb;
```

Arguments

urb

an URB associated with the failed or incomplete split transaction

Description

High speed HCDs use this to tell the hub driver that some split control or bulk transaction failed in a way that requires clearing internal state of a transaction translator. This is normally detected (and reported) from interrupt context.

It may not be possible for that hub to handle additional full (or low) speed transactions until that state is fully cleared out.

Name

usb_set_device_state — change a device's current state (usbcore, hcds)

Synopsis

```
void usb_set_device_state (udev,
                           new_state);

struct usb_device *    udev;
enum usb_device_state  new_state;
```

Arguments

udev

pointer to device whose state should be changed

new_state

new state value to be stored

Description

udev->state is _not_ fully protected by the device lock. Although most transitions are made only while holding the lock, the state can can change to USB_STATE_NOTATTACHED at almost any time. This is so that devices can be marked as disconnected as soon as possible, without having to wait for any semaphores to be released. As a result, all changes to any device's state must be protected by the device_state_lock spinlock.

Once a device has been added to the device tree, all changes to its state should be made using this routine. The state should _not_ be set directly.

If udev->state is already USB_STATE_NOTATTACHED then no change is made. Otherwise udev->state is set to new_state, and if new_state is USB_STATE_NOTATTACHED then all of udev's descendants' states are also set to USB_STATE_NOTATTACHED.

Name

usb_root_hub_lost_power — called by HCD if the root hub lost Vbus power

Synopsis

```
void usb_root_hub_lost_power (rhdev);

struct usb_device *  rhdev;
```

Arguments

rhdev

struct usb_device for the root hub

Description

The USB host controller driver calls this function when its root hub is resumed and Vbus power has been interrupted or the controller has been reset. The routine marks *rhdev* as having lost power. When the hub driver is resumed it will take notice and carry out power-session recovery for all the “USB-PERSIST”-enabled child devices; the others will be disconnected.

Name

usb_reset_device — warn interface drivers and perform a USB port reset

Synopsis

`int` **`usb_reset_device`** `(`*udev*`);`

`struct usb_device *` *`udev`*`;`

Arguments

udev

device to reset (not in SUSPENDED or NOTATTACHED state)

Description

Warns all drivers bound to registered interfaces (using their pre_reset method), performs the port reset, and then lets the drivers know that the reset is over (using their post_reset method).

Return value is the same as for `usb_reset_and_verify_device`.

The caller must own the device lock. For example, it's safe to use this from a driver `probe` routine after downloading new firmware. For calls that might not occur during `probe`, drivers should lock the device using `usb_lock_device_for_reset`.

If an interface is currently being probed or disconnected, we assume its driver knows how to handle resets. For all other interfaces, if the driver doesn't have pre_reset and post_reset methods then we attempt to unbind it and rebind afterward.

Name

usb_queue_reset_device — Reset a USB device from an atomic context

Synopsis

```
void usb_queue_reset_device (iface);

struct usb_interface * iface;
```

Arguments

iface

USB interface belonging to the device to reset

Description

This function can be used to reset a USB device from an atomic context, where `usb_reset_device` won't work (as it blocks).

Doing a reset via this method is functionally equivalent to calling `usb_reset_device`, except for the fact that it is delayed to a workqueue. This means that any drivers bound to other interfaces might be unbound, as well as users from usbfs in user space.

Corner cases

- Scheduling two resets at the same time from two different drivers attached to two different interfaces of the same device is possible; depending on how the driver attached to each interface handles ->`pre_reset`, the second reset might happen or not.

- If a driver is unbound and it had a pending reset, the reset will be cancelled.

- This function can be called during `.probe` or `.disconnect` times. On return from `.disconnect`, any pending resets will be cancelled.

There is no no need to lock/unlock the *`reset_ws`* as `schedule_work` does its own.

NOTE

We don't do any reference count tracking because it is not needed. The lifecycle of the work_struct is tied to the usb_interface. Before destroying the interface we cancel the work_struct, so the fact that work_struct is queued and or running means the interface (and thus, the device) exist and are referenced.

Chapter 6. Host Controller APIs

Table of Contents

These APIs are only for use by host controller drivers, most of which implement standard register interfaces such as 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).

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.

The same basic APIs are available to drivers for all those controllers. For historical reasons they are in two layers: struct usb_bus is a rather thin layer that became available in the 2.2 kernels, while struct usb_hcd is a more featureful layer (available in later 2.4 kernels and in 2.5) that lets HCDs share common code, to shrink driver size and significantly reduce hcd-specific behaviors.

Name

usb_calc_bus_time — approximate periodic transaction time in nanoseconds

Synopsis

```
long usb_calc_bus_time (speed,
                        is_input,
                        isoc,
                        bytecount);
```

```
int  speed;
int  is_input;
int  isoc;
int  bytecount;
```

Arguments

speed

from dev->speed; USB_SPEED_{LOW,FULL,HIGH} *is_input*

true iff the transaction sends data to the host

isoc

true for isochronous transactions, false for interrupt ones

bytecount

how many bytes in the transaction.

Description

Returns approximate bus time in nanoseconds for a periodic transaction. See USB 2.0 spec section 5.11.3; only periodic transfers need to be scheduled in software, this function is only used for such scheduling.

Name

usb_hcd_link_urb_to_ep — add an URB to its endpoint queue

Synopsis

```
int usb_hcd_link_urb_to_ep (hcd,
                            urb);

struct usb_hcd *  hcd;
struct urb *      urb;
```

Arguments

hcd

host controller to which `urb` was submitted

urb

URB being submitted

Description

Host controller drivers should call this routine in their `enqueue` method. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for URB submission, as well as for endpoint shutdown and for usb_kill_urb.

Returns 0 for no error, otherwise a negative error code (in which case the `enqueue` method must fail). If no error occurs but `enqueue` fails anyway, it must call `usb_hcd_unlink_urb_from_ep` before releasing the private spinlock and returning.

Name

usb_hcd_check_unlink_urb — check whether an URB may be unlinked

Synopsis

```
int usb_hcd_check_unlink_urb (hcd,
                              urb,
                              status);

struct usb_hcd *  hcd;
struct urb *      urb;
int               status;
```

Arguments

hcd

host controller to which *urb* was submitted

urb

URB being checked for unlinkability

status

error code to store in *urb* if the unlink succeeds

Description

Host controller drivers should call this routine in their `dequeue` method. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for making sure than an unlink is valid.

Returns 0 for no error, otherwise a negative error code (in which case the `dequeue` method must fail). The possible error codes are:

-EIDRM: *urb* was not submitted or has already completed. The completion function may not have been called yet.

-EBUSY: *urb* has already been unlinked.

Name

usb_hcd_unlink_urb_from_ep — remove an URB from its endpoint queue

Synopsis

```
void usb_hcd_unlink_urb_from_ep (hcd,
                                 urb);

struct usb_hcd *  hcd;
struct urb *      urb;
```

Arguments

hcd

host controller to which *urb* was submitted

urb

URB being unlinked

Description

Host controller drivers should call this routine before calling `usb_hcd_giveback_urb`. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for URB completion.

Name

usb_hcd_giveback_urb — return URB from HCD to device driver

Synopsis

```
void usb_hcd_giveback_urb (hcd,
                           urb,
                           status);

struct usb_hcd *  hcd;
struct urb *      urb;
int               status;
```

Arguments

hcd

host controller returning the URB

urb

urb being returned to the USB device driver.

status

completion status code for the URB.

Context

`in_interrupt`

Description

This hands the URB from HCD to its USB device driver, using its completion function. The HCD has freed all per-urb resources (and is done using urb->hcpriv). It also released all HCD locks; the device driver won't cause problems if it frees, modifies, or resubmits this URB.

If *urb* was unlinked, the value of *status* will be overridden by *urb*->unlinked. Erroneous short transfers are detected in case the HCD hasn't checked for them.

Name

usb_hcd_resume_root_hub — called by HCD to resume its root hub

Synopsis

`void` **`usb_hcd_resume_root_hub`** `(`*hcd*`);`

`struct usb_hcd *` *`hcd`*`;`

Arguments

hcd

host controller for this root hub

Description

The USB host controller calls this function when its root hub is suspended (with the remote wakeup feature enabled) and a remote wakeup request is received. The routine submits a workqueue request to resume the root hub (that is, manage its downstream ports again).

Name

usb_bus_start_enum — start immediate enumeration (for OTG)

Synopsis

```
int usb_bus_start_enum (bus,
                        port_num);

struct usb_bus *  bus;
unsigned          port_num;
```

Arguments

bus

the bus (must use hcd framework)

port_num

1-based number of port; usually bus->otg_port

Context

`in_interrupt`

Description

Starts enumeration, with an immediate reset followed later by khubd identifying and possibly configuring the device. This is needed by OTG controller drivers, where it helps meet HNP protocol timing requirements for starting a port reset.

Name

usb_hc_died — report abnormal shutdown of a host controller (bus glue)

Synopsis

```
void usb_hc_died (hcd);

struct usb_hcd *  hcd;
```

Arguments

hcd

pointer to the HCD representing the controller

Description

This is called by bus glue to report a USB host controller that died while operations may still have been pending. It's called automatically by the PCI glue, so only glue for non-PCI busses should need to call it.

Name

usb_create_hcd — create and initialize an HCD structure

Synopsis

```
struct usb_hcd * usb_create_hcd (driver,
                                 dev,
                                 bus_name);

const struct hc_driver *  driver;
struct device *           dev;
const char *              bus_name;
```

Arguments

driver

HC driver that will use this hcd

dev

device for this HC, stored in hcd->self.controller

bus_name

value to store in hcd->self.bus_name

Context

`!in_interrupt`

Description

Allocate a struct usb_hcd, with extra space at the end for the HC driver's private data. Initialize the generic members of the hcd structure.

If memory is unavailable, returns NULL.

Name

usb_add_hcd — finish generic HCD structure initialization and register

Synopsis

```
int usb_add_hcd (hcd,
                 irqnum,
                 irqflags);

struct usb_hcd *  hcd;
unsigned int      irqnum;
unsigned long     irqflags;
```

Arguments

hcd

the usb_hcd structure to initialize

irqnum

Interrupt line to allocate

irqflags

Interrupt type flags

Finish the remaining parts of generic HCD initialization

allocate the buffers of consistent memory, register the bus, request the IRQ line, and call the driver's `reset` and `start` routines.

Name

usb_remove_hcd — shutdown processing for generic HCDs

Synopsis

```
void usb_remove_hcd (hcd);

struct usb_hcd *  hcd;
```

Arguments

hcd

the usb_hcd structure to remove

Context

`!in_interrupt`

Description

Disconnects the root hub, then reverses the effects of `usb_add_hcd`, invoking the HCD's `stop` method.

Name

usb_hcd_pci_probe — initialize PCI-based HCDs

Synopsis

```
int usb_hcd_pci_probe (dev,
                       id);

struct pci_dev *              dev;
const struct pci_device_id *  id;
```

Arguments

dev

USB Host Controller being probed

id

pci hotplug id connecting controller to HCD framework

Context

`!in_interrupt`

Description

Allocates basic PCI resources for this USB host controller, and then invokes the `start` method for the HCD associated with it through the hotplug entry's driver_data.

Store this function in the HCD's struct pci_driver as `probe`.

Name

usb_hcd_pci_remove — shutdown processing for PCI-based HCDs

Synopsis

void **usb_hcd_pci_remove** (*dev*);

struct pci_dev * *dev*;

Arguments

dev

USB Host Controller being removed

Context

!in_interrupt

Description

Reverses the effect of `usb_hcd_pci_probe`, first invoking the HCD's `stop` method. It is always called from a thread context, normally “rmmod”, “apmd”, or something similar.

Store this function in the HCD's struct pci_driver as `remove`.

Name

usb_hcd_pci_shutdown — shutdown host controller

Synopsis

void **usb_hcd_pci_shutdown** (*dev*);

struct pci_dev * *dev*;

Arguments

dev

USB Host Controller being shutdown

Name

hcd_buffer_create — initialize buffer pools

Synopsis

int **hcd_buffer_create** (*hcd*);

`struct usb_hcd * ` *`hcd`*`;`

Arguments

`hcd`

    the bus whose buffer pools are to be initialized

Context

`!in_interrupt`

Description

Call this as part of initializing a host controller that uses the dma memory allocators. It initializes some pools of dma-coherent memory that will be shared by all drivers using that controller, or returns a negative errno value on error.

Call `hcd_buffer_destroy` to clean up after using those pools.

Name

hcd_buffer_destroy — deallocate buffer pools

Synopsis

`void` **`hcd_buffer_destroy`** `(`*hcd*`);`

`struct usb_hcd * ` *`hcd`*`;`

Arguments

`hcd`

    the bus whose buffer pools are to be destroyed

Context

`!in_interrupt`

Description

This frees the buffer pools created by `hcd_buffer_create`.

Chapter 7. The USB Filesystem (usbfs)

Table of Contents

This chapter presents the Linux *usbfs*. You may prefer to avoid writing new kernel code for your USB driver; that's the problem that usbfs set out to solve. User mode device drivers are usually packaged as applications or libraries, and may use usbfs through some programming library that wraps it. Such libraries include libusb for C/C++, and jUSB for Java.

Unfinished

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.

Configure usbfs into Linux kernels by enabling the *USB filesystem* option (CONFIG_USB_DEVICEFS), and you get basic support for user mode USB device drivers. Until relatively recently it was often (confusingly) called *usbdevfs* although it wasn't solving what *devfs* was. Every USB device will appear in usbfs, regardless of whether or not it has a kernel driver.

What files are in "usbfs"?

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

- `/proc/bus/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.
- `/proc/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. "usbfs" doesn't (yet) expose those IDs.

Mounting and Access Control

There are a number of mount options for usbfs, which will be of most interest to you if you need to override the default access control policy. That policy is that only root may read or write device files (`/proc/bus/BBB/DDD`) although anyone may read the `devices` or `drivers` files. I/O requests to the device also need the CAP_SYS_RAWIO capability,

The significance of that is that by default, all user mode device drivers need super-user privileges. You can change modes or ownership in a driver setup when the device hotplugs, or maye just start the driver right then, as a privileged server (or some activity within one). That's the most secure approach for multi-user systems, but for single user systems ("trusted" by that user) it's more convenient just to grant everyone all access (using the *devmode=0666* option) so the driver can start whenever it's needed.

The mount options for usbfs, usable in /etc/fstab or in command line invocations of *mount*, are:

busgid=NNNNN

> Controls the GID used for the /proc/bus/usb/BBB directories. (Default: 0)

busmode=MMM

> Controls the file mode used for the /proc/bus/usb/BBB directories. (Default: 0555)

busuid=NNNNN

> Controls the UID used for the /proc/bus/usb/BBB directories. (Default: 0)

devgid=NNNNN

> Controls the GID used for the /proc/bus/usb/BBB/DDD files. (Default: 0)

devmode=MMM

> Controls the file mode used for the /proc/bus/usb/BBB/DDD files. (Default: 0644)

devuid=NNNNN

> Controls the UID used for the /proc/bus/usb/BBB/DDD files. (Default: 0)

listgid=NNNNN

> Controls the GID used for the /proc/bus/usb/devices and drivers files. (Default: 0)

listmode=MMM

> Controls the file mode used for the /proc/bus/usb/devices and drivers files. (Default: 0444)

listuid=NNNNN

> Controls the UID used for the /proc/bus/usb/devices and drivers files. (Default: 0)

Note that many Linux distributions hard-wire the mount options for usbfs in their init scripts, such as

/etc/rc.d/rc.sysinit, rather than making it easy to set this per-system policy in /etc/fstab.

/proc/bus/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 the Documentation/usb/proc_usb_info.txt file in your Linux kernel sources.

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("/proc/bus/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.

/proc/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.

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 /proc/bus/usb device files, and ignores most devices. In either case, it should `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.

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 `Documentation/usb/error-codes.txt` in your kernel sources.)

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 khubd (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 charactor 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.

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 `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.

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