

# **The Linux-USB Host Side API**

---

# The Linux-USB Host Side API

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

---







---

# 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 "kernel doc" 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 3. USB-Standard Types

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

`usb_speed_string` — Returns human readable-name of the speed.

## Synopsis

```
const char * usb_speed_string (enum usb_device_speed speed);
```

## Arguments

*speed* The speed to return human-readable name for. If it's not any of the speeds defined in `usb_device_speed` enum, string for `USB_SPEED_UNKNOWN` will be returned.

## Name

`usb_get_maximum_speed` — Get maximum requested speed for a given USB controller.

## Synopsis

```
enum usb_device_speed usb_get_maximum_speed (struct device * dev);
```

## Arguments

*dev*    Pointer to the given USB controller device

## Description

The function gets the maximum speed string from property “maximum-speed”, and returns the corresponding enum `usb_device_speed`.

## Name

`usb_state_string` — Returns human readable name for the state.

## Synopsis

```
const char * usb_state_string (enum usb_device_state state);
```

## Arguments

*state* The state to return a human-readable name for. If it's not any of the states devices in `usb_device_state_string` enum, the string UNKNOWN will be returned.

---

# Chapter 4. Host-Side Data Types and Macros

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







## 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.



desires (expressed through userspace tools). However, drivers can call `usb_reset_configuration` to reinitialize the current configuration and all its interfaces.







slot_id	Slot ID assigned by xHCI
removable	Device can be physically removed from this port
l1_params	best effort service latency for USB2 L1 LPM state, and L1 timeout.
u1_params	exit latencies for USB3 U1 LPM state, and hub-initiated timeout.
u2_params	exit latencies for USB3 U2 LPM state, and hub-initiated timeout.
lpm_disable_count	Ref count used by <code>usb_disable_lpm</code> and <code>usb_enable_lpm</code> to keep track of the number of functions that require USB 3.0 Link Power Management to be disabled for this <code>usb_device</code> . This count should only be manipulated by those functions, with the <code>bandwidth_mutex</code> is held.

## Notes

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

## Name

`usb_hub_for_each_child` — iterate over all child devices on the hub

## Synopsis

```
usb_hub_for_each_child ( hdev, port1, child );
```

## Arguments

*hdev*     USB device belonging to the usb hub

*port1*   portnum associated with child device

*child*   child device pointer





## Name

USB\_DEVICE — macro used to describe a specific usb device

## Synopsis

```
USB_DEVICE ( 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 );
```

## 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_CLASS` — describe a usb device with a specific interface class

## Synopsis

```
USB_DEVICE_INTERFACE_CLASS ( vend, prod, cl );
```

## Arguments

*vend* the 16 bit USB Vendor ID

*prod* the 16 bit USB Product ID

*cl* bInterfaceClass value

## Description

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

## Name

USB\_DEVICE\_INTERFACE\_PROTOCOL — describe a usb device with a specific interface protocol

## Synopsis

```
USB_DEVICE_INTERFACE_PROTOCOL ( 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\_INTERFACE\_NUMBER — describe a usb device with a specific interface number

## Synopsis

```
USB_DEVICE_INTERFACE_NUMBER ( vend, prod, num );
```

## Arguments

*vend* the 16 bit USB Vendor ID

*prod* the 16 bit USB Product ID

*num* bInterfaceNumber value

## Description

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

## Name

USB\_DEVICE\_INFO — macro used to describe a class of usb devices

## Synopsis

```
USB_DEVICE_INFO ( 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 );
```

## 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 );
```

## 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

`USB_VENDOR_AND_INTERFACE_INFO` — describe a specific usb vendor with a class of usb interfaces

## Synopsis

```
USB_VENDOR_AND_INTERFACE_INFO ( vend, cl, sc, pr );
```

## Arguments

*vend* the 16 bit USB Vendor 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 vendor 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\_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, umode_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

`module_usb_driver` — Helper macro for registering a USB driver

## Synopsis

```
module_usb_driver ( __usb_driver );
```

## Arguments

*\_\_usb\_driver*    `usb_driver` struct

## Description

Helper macro for USB drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init` and `module_exit`





















---

## Chapter 5. USB Core APIs

There are two basic I/O models in the USB API. The most elemental one is asynchronous: drivers submit requests in the form of an URB, and the URB's completion callback 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 (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_free_urb` — frees the memory used by a urb when all users of it are finished

## Synopsis

```
void usb_free_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 (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.

## Return

A pointer to the urb with the incremented reference counter.

## Name

`usb_anchor_urb` — anchors an URB while it is processed

## Synopsis

```
void usb_anchor_urb (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 (struct urb * urb);
```

## Arguments

*urb* pointer to the urb to anchor

## Description

Call this to stop the system keeping track of this URB





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



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_block_urb` — reliably prevent further use of an URB

## Synopsis

```
void usb_block_urb (struct urb * urb);
```

## Arguments

*urb* pointer to URB to be blocked, may be NULL

## Description

After the routine has run, 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.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

## Name

`usb_kill_anchored_urbs` — cancel transfer requests en masse

## Synopsis

```
void usb_kill_anchored_urbs (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 (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 (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 (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 triggered. 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_anchor_suspend_wakeups` —

## Synopsis

```
void usb_anchor_suspend_wakeups (struct usb_anchor * anchor);
```

## Arguments

*anchor* the anchor you want to suspend wakeups on

## Description

Call this to stop the last urb being unanchored from waking up any `usb_wait_anchor_empty_timeout` waiters. This is used in the hcd urb give- back path to delay waking up until after the completion handler has run.

## Name

`usb_anchor_resume_wakeups` —

## Synopsis

```
void usb_anchor_resume_wakeups (struct usb_anchor * anchor);
```

## Arguments

*anchor* the anchor you want to resume wakeups on

## Description

Allow `usb_wait_anchor_empty_timeout` waiters to be woken up again, and wake up any current waiters if the anchor is empty.

## Name

`usb_wait_anchor_empty_timeout` — wait for an anchor to be unused

## Synopsis

```
int usb_wait_anchor_empty_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 if you want to be sure all an anchor's URBs have finished

## Return

Non-zero if the anchor became unused. Zero on timeout.

## Name

`usb_get_from_anchor` — get an anchor's oldest urb

## Synopsis

```
struct urb * usb_get_from_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

## Return

The oldest urb from *anchor*, or NULL if *anchor* has no urbs associated with it.

## Name

`usb_scuttle_anchored_urbs` — unanchor all an anchor's urbs

## Synopsis

```
void usb_scuttle_anchored_urbs (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 (struct usb_anchor * anchor);
```

## Arguments

*anchor* the anchor you want to query

## Return

1 if the anchor has no urbs associated with it.











## Name

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

## Synopsis

```
void usb_sg_cancel (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_reset_endpoint` — Reset an endpoint's state.

## Synopsis

```
void usb_reset_endpoint (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_deregister_device_driver` — unregister a USB device (not interface) driver

## Synopsis

```
void usb_deregister_device_driver (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 (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.

## Return

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 (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_enable_autosuspend` — allow a USB device to be autosuspended

## Synopsis

```
void usb_enable_autosuspend (struct usb_device * udev);
```

## Arguments

*udev* the USB device which may be autosuspended

## Description

This routine allows *udev* to be autosuspended. An autosuspend won't take place until the `autosuspend_delay` has elapsed and all the other necessary conditions are satisfied.

The caller must hold *udev*'s device lock.

## Name

`usb_disable_autosuspend` — prevent a USB device from being autosuspended

## Synopsis

```
void usb_disable_autosuspend (struct usb_device * udev);
```

## Arguments

*udev* the USB device which may not be autosuspended

## Description

This routine prevents *udev* from being autosuspended and wakes it up if it is already autosuspended.

The caller must hold *udev*'s device lock.

## Name

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

## Synopsis

```
void usb_autopm_put_interface (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 attempted. The attempt may fail (see `autosuspend_check`).

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 (struct usb_interface * intf);
```

## Arguments

*intf* the `usb_interface` whose counter should be decremented

## Description

This routine does much the same thing as `usb_autopm_put_interface`: It decrements *intf*'s usage counter and schedules a delayed autosuspend request if the counter is  $\leq 0$ . The difference is that it does not perform any synchronization; callers should hold a private lock and 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_put_interface_no_suspend` — decrement a USB interface's PM-usage counter

## Synopsis

```
void usb_autopm_put_interface_no_suspend (struct usb_interface * intf);
```

## Arguments

*intf* the `usb_interface` whose counter should be decremented

## Description

This routine decrements *intf*'s usage counter but does not carry out an autosuspend.

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 (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.

*intf*'s usage counter is incremented to prevent subsequent autosuspends. However if the autoresume fails then the counter is re-decremented.

This routine can run only in process context.

## Return

0 on success.

## Name

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

## Synopsis

```
int usb_autopm_get_interface_async (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 device is suspended. The differences are that it does not perform any synchronization (callers should hold a private lock and handle all synchronization issues themselves), and it does not autoresume the device directly (it only queues a request). After a successful call, the device may not yet be resumed.

This routine can run in atomic context.

## Return

0 on success. A negative error code otherwise.

## Name

`usb_autopm_get_interface_no_resume` — increment a USB interface's PM-usage counter

## Synopsis

```
void usb_autopm_get_interface_no_resume (struct usb_interface * intf);
```

## Arguments

*intf* the `usb_interface` whose counter should be incremented

## Description

This routine increments *intf*'s usage counter but does not carry out an autoresume.

This routine can run in atomic context.

## Name

`usb_find_alt_setting` — Given a configuration, find the alternate setting for the given interface.

## Synopsis

```
struct usb_host_interface * usb_find_alt_setting (struct usb_host_con-  
fig * config, unsigned int iface_num, unsigned int alt_num);
```

## Arguments

*config*        the configuration to search (not necessarily the current config).

*iface\_num*    interface number to search in

*alt\_num*       alternate interface setting number to search for.

## Description

Search the configuration's interface cache for the given alt setting.

## Return

The alternate setting, if found. NULL otherwise.

## Name

`usb_ifnum_to_if` — get the interface object with a given interface number

## Synopsis

```
struct usb_interface * usb_ifnum_to_if (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 to find the interface object with the particular interface number.

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!

## Return

A pointer to the interface that has *ifnum* as interface number, if found. NULL otherwise.

## Name

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

## Synopsis

```
struct usb_host_interface * usb_altnum_to_altsetting (const struct usb-  
b_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.

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!

## Return

A pointer to the entry of the altsetting array of *intf* that has *altnum* as the alternate setting number. NULL if not found.

## Name

`usb_find_interface` — find `usb_interface` pointer for driver and device

## Synopsis

```
struct usb_interface * usb_find_interface (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 bus device list and returns a pointer to the interface with the matching minor and driver. Note, this only works for devices that share the USB major number.

## Return

A pointer to the interface with the matching major and *minor*.

## Name

`usb_for_each_dev` — iterate over all USB devices in the system

## Synopsis

```
int usb_for_each_dev (void * data, int (*fn) (struct usb_device *, void *));
```

## Arguments

*data*    data pointer that will be handed to the callback function

*fn*      callback function to be called for each USB device

## Description

Iterate over all USB devices and call *fn* for each, passing it *data*. If it returns anything other than 0, we break the iteration prematurely and return that value.

## Name

`usb_alloc_dev` — usb device constructor (usbcore-internal)

## Synopsis

```
struct usb_device * usb_alloc_dev (struct usb_device * parent, struct  
usb_bus * bus, unsigned port1);
```

## Arguments

*parent*    hub to which device is connected; null to allocate a root hub

*bus*       bus used to access the device

*port1*     one-based index of port; ignored for root hubs

## Context

`!in_interrupt`

## Description

Only hub drivers (including virtual root hub drivers for host controllers) should ever call this.

This call may not be used in a non-sleeping context.

## Return

On success, a pointer to the allocated usb device. NULL on failure.

## Name

`usb_get_dev` — increments the reference count of the usb device structure

## Synopsis

```
struct usb_device * usb_get_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.

## Return

A pointer to the device with the incremented reference counter.

## Name

`usb_put_dev` — release a use of the usb device structure

## Synopsis

```
void usb_put_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 (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.

## Return

A pointer to the interface with the incremented reference counter.

## Name

`usb_put_intf` — release a use of the usb interface structure

## Synopsis

```
void usb_put_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 (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.

## Return

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 (struct usb_device * dev);
```

## Arguments

*dev* the device whose bus is being queried

## Return

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

## Note

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_alloc_coherent` — allocate dma-consistent buffer for URB\_NO\_XXX\_DMA\_MAP

## Synopsis

```
void * usb_alloc_coherent (struct usb_device * dev, size_t size, gfp_t  
mem_flags, dma_addr_t * dma);
```

## Arguments

<i>dev</i>	device the buffer will be used with
<i>size</i>	requested buffer size
<i>mem_flags</i>	affect whether allocation may block
<i>dma</i>	used to return DMA address of buffer

## Return

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).

## Note

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_free_coherent`.

## Name

`usb_free_coherent` — free memory allocated with `usb_alloc_coherent`

## Synopsis

```
void usb_free_coherent (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_alloc_coherent`, 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 (struct urb * urb);
```

## Arguments

*urb* urb whose transfer\_buffer/setup\_packet will be mapped

## Description

URB\_NO\_TRANSFER\_DMA\_MAP is 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_dma_sync` calls to synchronize memory and dma state.

Reverse the effect of this call with `usb_buffer_unmap`.

## Return

Either NULL (indicating no buffer could be mapped), or *urb*.

## Name

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

## Synopsis

```
void usb_buffer_dmasync (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 (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 (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

## Return

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

## Note

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

## Arguments

<i>dev</i>	device to which the scatterlist will be mapped
<i>is_in</i>	mapping transfer direction
<i>sg</i>	the scatterlist to synchronize
<i>n_hw_ents</i>	the positive return value from <code>usb_buffer_map_sg</code>

## 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 (const struct usb_device * dev, int is_in,  
struct scatterlist * sg, int n_hw_ents);
```

## Arguments

<i>dev</i>	device to which the scatterlist will be mapped
<i>is_in</i>	mapping transfer direction
<i>sg</i>	the scatterlist to unmap
<i>n_hw_ents</i>	the positive return value from <code>usb_buffer_map_sg</code>

## 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 (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.

## Return

0 if successful. A negative error code otherwise.

## Name

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

## Synopsis

```
void usb_set_device_state (struct usb_device * udev, enum usb_device_s-  
tate 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 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 (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 (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

The same as for `usb_reset_and_verify_device`.

## Note

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 (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 the reset is delayed so long that the interface is unbound from its driver, the reset will be skipped.
- This function can be called during `.probe`. It can also be called during `.disconnect`, but doing so is pointless because the reset will not occur. If you really want to reset the device during `.disconnect`, call `usb_reset_device` directly -- but watch out for nested unbinding issues!

## Name

`usb_hub_find_child` — Get the pointer of child device attached to the port which is specified by *port1*.

## Synopsis

```
struct usb_device * usb_hub_find_child (struct usb_device * hdev, int  
port1);
```

## Arguments

*hdev*     USB device belonging to the usb hub

*port1*   port num to indicate which port the child device is attached to.

## Description

USB drivers call this function to get hub's child device pointer.

## Return

NULL if input param is invalid and child's `usb_device` pointer if non-NULL.

---

# Chapter 6. Host Controller APIs

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 (int speed, int is_input, int isoc, int byte-  
count);
```

## Arguments

<i>speed</i>	from <code>dev-&gt;speed</code> ; <code>USB_SPEED_{LOW,FULL,HIGH}</code>
<i>is_input</i>	true iff the transaction sends data to the host
<i>isoc</i>	true for isochronous transactions, false for interrupt ones
<i>bytecount</i>	how many bytes in the transaction.

## Return

Approximate bus time in nanoseconds for a periodic transaction.

## Note

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 (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`.

## Return

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 (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.

## Return

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 (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 (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_alloc_streams` — allocate bulk endpoint stream IDs.

## Synopsis

```
int usb_alloc_streams (struct usb_interface * interface, struct usb_host_endpoint ** eps, unsigned int num_eps, unsigned int num_streams, gfp_t mem_flags);
```

## Arguments

<i>interface</i>	alternate setting that includes all endpoints.
<i>eps</i>	array of endpoints that need streams.
<i>num_eps</i>	number of endpoints in the array.
<i>num_streams</i>	number of streams to allocate.
<i>mem_flags</i>	flags hcd should use to allocate memory.

## Description

Sets up a group of bulk endpoints to have *num\_streams* stream IDs available. Drivers may queue multiple transfers to different stream IDs, which may complete in a different order than they were queued.

## Return

On success, the number of allocated streams. On failure, a negative error code.

## Name

`usb_free_streams` — free bulk endpoint stream IDs.

## Synopsis

```
int usb_free_streams (struct usb_interface * interface, struct usb_host_endpoint ** eps, unsigned int num_eps, gfp_t mem_flags);
```

## Arguments

*interface*    alternate setting that includes all endpoints.

*eps*                array of endpoints to remove streams from.

*num\_eps*        number of endpoints in the array.

*mem\_flags*    flags hcd should use to allocate memory.

## Description

Reverts a group of bulk endpoints back to not using stream IDs. Can fail if we are given bad arguments, or HCD is broken.

## Return

0 on success. On failure, a negative error code.

## Name

`usb_hcd_resume_root_hub` — called by HCD to resume its root hub

## Synopsis

```
void usb_hcd_resume_root_hub (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 (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 `hub_wq` 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.

## Return

0 if successful.

## Name

`usb_hcd_irq` — hook IRQs to HCD framework (bus glue)

## Synopsis

```
irqreturn_t usb_hcd_irq (int irq, void * __hcd);
```

## Arguments

*irq*      the IRQ being raised

*\_\_hcd*    pointer to the HCD whose IRQ is being signaled

## Description

If the controller isn't HALTed, calls the driver's irq handler. Checks whether the controller is now dead.

## Return

`IRQ_HANDLED` if the IRQ was handled. `IRQ_NONE` otherwise.

## Name

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

## Synopsis

```
void usb_hc_died (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.

Only call this function with the primary HCD.

## Name

`usb_create_shared_hcd` — create and initialize an HCD structure

## Synopsis

```
struct usb_hcd * usb_create_shared_hcd (const struct hc_driver * driver,  
struct device * dev, const char * bus_name, struct usb_hcd * primary_hcd);
```

## Arguments

<i>driver</i>	HC driver that will use this hcd
<i>dev</i>	device for this HC, stored in <code>hcd-&gt;self.controller</code>
<i>bus_name</i>	value to store in <code>hcd-&gt;self.bus_name</code>
<i>primary_hcd</i>	a pointer to the <code>usb_hcd</code> structure that is sharing the PCI device. Only allocate certain resources for the primary HCD

## 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.

## Return

On success, a pointer to the created and initialized HCD structure. On failure (e.g. if memory is unavailable), `NULL`.

## Name

`usb_create_hcd` — create and initialize an HCD structure

## Synopsis

```
struct usb_hcd * usb_create_hcd (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.

## Return

On success, a pointer to the created and initialized HCD structure. On failure (e.g. if memory is unavailable), `NULL`.

## Name

`usb_add_hcd` — finish generic HCD structure initialization and register

## Synopsis

```
int usb_add_hcd (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 (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 (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`.

## Return

0 if successful.

## Name

`usb_hcd_pci_remove` — shutdown processing for PCI-based HCDs

## Synopsis

```
void usb_hcd_pci_remove (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 (struct pci_dev * dev);
```

## Arguments

*dev*    USB Host Controller being shutdown

## Name

`hcd_buffer_create` — initialize buffer pools

## Synopsis

```
int hcd_buffer_create (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.

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

## Return

0 if successful. A negative `errno` value otherwise.

## Name

`hcd_buffer_destroy` — deallocate buffer pools

## Synopsis

```
void hcd_buffer_destroy (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)

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* [<http://libusb.sourceforge.net>] for C/C++, and *jUSB* [<http://jUSB.sourceforge.net>] 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 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 each device's configuration descriptors, and supporting a series of `ioctl`s 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 maybe 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)

<i>busmode=MMM</i>	Controls the file mode used for the /proc/bus/usb/BBB directories. (Default: 0555)
<i>busuid=NNNNN</i>	Controls the UID used for the /proc/bus/usb/BBB directories. (Default: 0)
<i>devgid=NNNNN</i>	Controls the GID used for the /proc/bus/usb/BBB/DDD files. (Default: 0)
<i>devmode=MMM</i>	Controls the file mode used for the /proc/bus/usb/BBB/DDD files. (Default: 0644)
<i>devuid=NNNNN</i>	Controls the UID used for the /proc/bus/usb/BBB/DDD files. (Default: 0)
<i>listgid=NNNNN</i>	Controls the GID used for the /proc/bus/usb/devices and drivers files. (Default: 0)
<i>listmode=MMM</i>	Controls the file mode used for the /proc/bus/usb/devices and drivers files. (Default: 0444)
<i>listuid=NNNNN</i>	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 `hub_wq` (in the kernel) setting a device-wide *configuration* that affects things like power consumption and basic functionality. The endpoints are part of USB *interfaces*, which may have *altsettings* affecting things like which endpoints are available. Many devices only have a single configuration and interface, so drivers for them will ignore configurations and altsettings.

## Management/Status Requests

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

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

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

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

**USBDEVFS\_CONNECTINFO** Says whether the device is `lowspeed`. The `ioctl` parameter points to a structure like this:

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

File modification time is not updated by this request.

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

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

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

File modification time is not updated by this request.

**USBDEVFS\_IOCTL** Passes a request from userspace through to a kernel driver that has an `ioctl` entry in the *struct usb\_driver* it registered.

```

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

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

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

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

```

File modification time is not updated by this request.

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

#### USBDEVFS\_RELEASEINTERFACE

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

### Warning

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

#### USBDEVFS\_RESETEP

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

### Warning

*Avoid using this request. It should probably be removed. Using it typically means the device and driver will lose toggle synchronization. If you really lost synchronization,*

you likely need to completely handshake with the device, using a request like `CLEAR_HALT` or `SET_INTERFACE`.

## 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\_\*), end-

point (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*