

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.

Table of Contents

1. Introduction to USB on Linux	1
2. USB Host-Side API Model	3
3. USB-Standard Types	5
struct usb_ctrlrequest	5
4. Host-Side Data Types and Macros	7
struct usb_host_endpoint	7
struct usb_interface	8
struct usb_interface_cache	10
struct usb_host_config	11
usb_interface_claimed	12
usb_make_path	13
USB_DEVICE	14
USB_DEVICE_VER	15
USB_DEVICE_INFO	16
USB_INTERFACE_INFO	17
struct usb_driver	18
struct usb_class_driver	20
struct urb	21
usb_fill_control_urb	26
usb_fill_bulk_urb	27
usb_fill_int_urb	28
struct usb_sg_request	30
5. USB Core APIs	33
usb_init_urb	33
usb_alloc_urb	34
usb_free_urb	35
usb_get_urb	36
usb_submit_urb	36
usb_unlink_urb	39
usb_kill_urb	41
usb_control_msg	42
usb_bulk_msg	43
usb_sg_init	45
usb_sg_wait	46
usb_sg_cancel	48
usb_get_descriptor	49
usb_get_string	50
usb_string	51
usb_get_status	53
usb_clear_halt	54

usb_set_interface.....	55
usb_reset_configuration	56
usb_register_dev	57
usb_deregister_dev.....	59
usb_register	59
usb_deregister	60
usb_ifnum_to_if.....	61
usb_altnum_to_altsetting	62
usb_driver_claim_interface.....	64
usb_driver_release_interface	65
usb_match_id.....	66
usb_find_interface.....	68
usb_alloc_dev	69
usb_get_dev	70
usb_put_dev	70
usb_lock_device.....	71
usb_trylock_device	72
usb_lock_device_for_reset.....	73
usb_unlock_device.....	74
usb_find_device	75
usb_get_current_frame_number	76
usb_buffer_alloc.....	77
usb_buffer_free	78
usb_buffer_map	79
usb_buffer_dmasync	80
usb_buffer_unmap	80
usb_buffer_map_sg.....	81
usb_buffer_dmasync_sg.....	82
usb_buffer_unmap_sg.....	83
usb_hub_tt_clear_buffer	84
usb_set_device_state.....	85
usb_disconnect.....	86
usb_suspend_device.....	87
usb_resume_device	88
usb_reset_device	89
6. Host Controller APIs	91
usb_bus_init	91
usb_alloc_bus.....	92
usb_register_bus.....	93
usb_deregister_bus.....	93
usb_register_root_hub.....	94
usb_calc_bus_time	95
usb_claim_bandwidth	96

usb_release_bandwidth.....	97
usb_bus_start_enum.....	98
usb_hcd_giveback_urb.....	99
usb_hcd_irq.....	100
usb_create_hcd.....	101
usb_hcd_pci_probe	102
usb_hcd_pci_remove	103
usb_hcd_pci_suspend	104
usb_hcd_pci_resume.....	105
hcd_buffer_create.....	105
hcd_buffer_destroy	106
7. The USB Filesystem (usbfs).....	109
7.1. What files are in "usbfs"?.....	109
7.2. Mounting and Access Control	110
7.3. /proc/bus/usb/devices	111
7.4. /proc/bus/usb/BBB/DDD	111
7.5. Life Cycle of User Mode Drivers	112
7.6. The ioctl() Requests	112
7.6.1. Management/Status Requests	113
7.6.2. Synchronous I/O Support.....	115
7.6.3. Asynchronous I/O Support	118

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 at the root (the system's master), hubs as interior nodes, and peripheral devices 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 part for 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 the downstream type are the squarish type (or they are built in to the peripheral). Software doesn't need to deal with distributed autoconfiguration since the pre-designated master node manages all that. At the electrical level, bus protocol overhead is reduced by eliminating arbitration and moving scheduling into 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, including high speed transfers and transaction translating hubs (used for USB 1.1 and 1.0 backward compatibility).

USB support was added to Linux early in the 2.2 kernel series shortly before the 2.3 development forked off. Updates from 2.3 were regularly folded back into 2.2 releases, bringing new features such as `/sbin/hotplug` support, more drivers, and more robustness. The 2.5 kernel series continued such improvements, and also worked on USB 2.0 support, higher performance, better consistency between host controller drivers, API simplification (to make bugs less likely), and providing internal "kerneldoc" documentation.

Linux can run inside USB devices as well as on the hosts that control the devices. Because the Linux 2.x USB support evolved to support mass market platforms such as Apple Macintosh or PC-compatible systems, it didn't address design concerns for those types of USB systems. So it can't be used inside mass-market PDAs, or other peripherals. USB device drivers running inside those Linux peripherals don't do the same things as the ones running inside hosts, and so they've been given a different name: they're called *gadget drivers*. This document does not present gadget drivers.

Chapter 2. USB Host-Side API Model

Within the kernel, host-side drivers for USB devices talk to the "usbcore" APIs. There are two types of public "usbcore" APIs, targetted at two different layers of USB driver. Those are *general purpose* drivers, exposed through driver frameworks such as block, character, or network devices; and drivers that are *part of the core*, which are involved in managing a USB bus. Such core drivers include the *hub* driver, which manages trees of USB devices, and several different kinds of *host controller driver (HCD)*, which control individual busses.

The device model seen by USB drivers is relatively complex.

- USB supports four kinds of data transfer (control, bulk, interrupt, and isochronous). Two transfer types use bandwidth as it's available (control and bulk), while the other two types of transfer (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 that might be used.
- Configurations have one or more "interface", 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 messaging. It also supports asynchronous calls for all kinds of data transfer, using request structures called "URBs" (USB Request Blocks).

Accordingly, the USB Core API exposed to device drivers covers quite a lot of territory. You'll probably need to consult the USB 2.0 specification, available online from www.usb.org at no cost, as well as class or device specifications.

Chapter 2. USB Host-Side API Model

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

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.

struct usb_ctrlrequest

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.

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.

struct usb_host_endpoint

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;
    unsigned char * extra;
    int extralen;
};
```

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

extra

descriptors following this endpoint in the configuration

extralen

how many bytes of “extra” are valid

Description

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

struct usb_interface

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;
    int minor;
    enum usb_interface_condition condition;
    struct device dev;
    struct class_device * class_dev;
};
```

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.

`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`)

`dev`

driver model's view of this device

`class_dev`

driver model's class view of this device.

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 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.

struct usb_interface_cache

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.

struct usb_host_config

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 * 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.

`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 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 hotplug scripts). However, drivers can call `usb_reset_configuration` to reinitialize the current configuration and all its interfaces.

usb_interface_claimed

Name

`usb_interface_claimed` — returns true iff an interface is claimed

Synopsis

```
int usb_interface_claimed (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.

usb_make_path

Name

`usb_make_path` — returns stable device path in the usb tree

Synopsis

```
int usb_make_path (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 identifiers; 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.

USB_DEVICE

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.

USB_DEVICE_VER

Name

USB_DEVICE_VER — macro used to 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.

USB_DEVICE_INFO

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.

USB_INTERFACE_INFO

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.

struct usb_driver

Name

`struct usb_driver` — identifies USB driver to `usbcore`

Synopsis

```
struct usb_driver {
    struct module * owner;
    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, u32 state);
    int (* resume) (struct usb_interface *intf);
    const struct usb_device_id * id_table;
    struct device_driver driver;
};
```


Members

owner

Pointer to the module owner of this driver; initialize it using `THIS_MODULE`.

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 `dev_set_drvdata` 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 a 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.

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.

driver

the driver model core driver structure.

Description

USB 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).

struct usb_class_driver

Name

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

Synopsis

```
struct usb_class_driver {
    char * name;
    struct file_operations * fops;
    mode_t mode;
    int minor_base;
};
```

Members

`name`

devfs name for this driver. Will also be used by the driver class code to create a usb class device.

fops

pointer to the struct file_operations of this driver.

mode

the mode for the devfs file to be created for 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.

struct urb

Name

`struct urb` — USB Request Block

Synopsis

```
struct urb {
    struct list_head urb_list;
    struct usb_device * dev;
    unsigned int pipe;
    int status;
    unsigned int transfer_flags;
    void * transfer_buffer;
    dma_addr_t transfer_dma;
    int transfer_buffer_length;
    int 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.

dev

Identifies the USB device to perform the request.

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). 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`.

`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`.

`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` and `setup_packet` must still be set because not all host controllers use DMA, nor do virtual root hubs).

Initialization

All URBs submitted must initialize the `dev`, `pipe`, `transfer_flags` (may be zero), and complete fields. The `URB_ASYNC_UNLINK` transfer flag affects later invocations of the `usb_unlink_urb` routine. Note: Failure to set `URB_ASYNC_UNLINK` with `usb_unlink_urb` is deprecated. For synchronous unlinks use `usb_kill_urb` instead.

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.

usb_fill_control_urb

Name

`usb_fill_control_urb` — initializes a control urb

Synopsis

```
void usb_fill_control_urb (struct urb * urb, struct usb_device  
* dev, unsigned int pipe, unsigned char * setup_packet, void *  
transfer_buffer, int buffer_length, usb_complete_t complete,  
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

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.

`usb_fill_bulk_urb`

Name

`usb_fill_bulk_urb` — macro to help initialize a bulk urb

Synopsis

```
void usb_fill_bulk_urb (struct urb * urb, struct usb_device *  
dev, unsigned int pipe, void * transfer_buffer, int  
buffer_length, usb_complete_t complete, 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

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.

usb_fill_int_urb

Name

`usb_fill_int_urb` — macro to help initialize a interrupt urb

Synopsis

```
void usb_fill_int_urb (struct urb * urb, struct usb_device *  
dev, unsigned int pipe, void * transfer_buffer, int  
buffer_length, usb_complete_t complete, 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

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).

struct usb_sg_request

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

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.

usb_init_urb

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.

usb_alloc_urb

Name

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

Synopsis

```
struct urb * usb_alloc_urb (int iso_packets, int 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.

usb_free_urb

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, that must be done elsewhere.

usb_get_urb

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.

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

usb_submit_urb

Name

`usb_submit_urb` — issue an asynchronous transfer request for an endpoint

Synopsis

```
int usb_submit_urb (struct urb * urb, int 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 cancelation”).

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 cancelation).

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.

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 you 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

usb_unlink_urb

Name

`usb_unlink_urb` — abort/cancel a transfer request for an endpoint

Synopsis

```
int usb_unlink_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 cancelation means the request's completion handler will be called with a status code indicating that the request has been canceled (rather than any other code) and will quickly be removed from host controller data structures.

In the past, clearing the `URB_ASYNC_UNLINK` transfer flag for the URB indicated that the request was synchronous. This usage is now deprecated; if the flag is clear the call will be forwarded to `usb_kill_urb` and the return value will be 0. In the future, drivers should call `usb_kill_urb` directly for synchronous unlinking.

When the `URB_ASYNC_UNLINK` transfer flag for the URB is set, this request is asynchronous. Success is indicated by returning `-EINPROGRESS`, at which time the URB will normally have been unlinked but not yet given back to the device driver. When it is called, the completion function will see `urb->status == -ECONNRESET`. Failure is indicated by any other return value. Unlinking will fail when the 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

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 any fault (such as an error, or being unlinked) its queue stops, at least until that URB's completion routine returns. It is guaranteed that the 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.

This means that USB device drivers can safely build deep queues for large or complex transfers, and clean them up reliably after any sort of aborted transfer by unlinking all pending URBs at the first fault.

Note that an URB terminating early because a short packet was received will count as an error if and only if the `URB_SHORT_NOT_OK` flag is set. Also, that all unlinks performed in any URB completion handler must be asynchronous.

Queues for isochronous endpoints are treated differently, because they advance at fixed rates. Such queues do not stop when an URB is unlinked. An unlinked URB may leave a gap in the stream of packets. It is undefined whether such gaps can be filled in.

When a control URB terminates with an error, it is likely that the status stage of the transfer will not take place, even if it is merely a soft error resulting from a short-packet with `URB_SHORT_NOT_OK` set.

usb_kill_urb

Name

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

Synopsis

```
void usb_kill_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`.

usb_control_msg

Name

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

Synopsis

```
int usb_control_msg (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 jiffies 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.

usb_bulk_msg

Name

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

Synopsis

```
int usb_bulk_msg (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 jiffies 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` parameter.

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.

usb_sg_init

Name

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

Synopsis

```
int usb_sg_init (struct usb_sg_request * io, struct usb_device
* dev, unsigned pipe, unsigned period, struct scatterlist *
sg, int nents, size_t length, int 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.

usb_sg_wait

Name

`usb_sg_wait` — synchronously execute scatter/gather request

Synopsis

```
void usb_sg_wait (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) cancelation, 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).

usb_sg_cancel

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.

usb_get_descriptor

Name

`usb_get_descriptor` — issues a generic GET_DESCRIPTOR request

Synopsis

```
int usb_get_descriptor (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.

usb_get_string

Name

`usb_get_string` — gets a string descriptor

Synopsis

```
int usb_get_string (struct usb_device * dev, unsigned short  
langid, unsigned char index, void * buf, int size);
```

Arguments

dev

the device whose string descriptor is being retrieved

langid

code for language chosen (from string descriptor zero)

index

the number of the descriptor

buf

where to put the string

size

how big is “buf”?

Context

`!in_interrupt ()`

Description

Retrieves a string, encoded using UTF-16LE (Unicode, 16 bits per character, in little-endian byte order). The `usb_string` function will often be a convenient way to turn these strings into kernel-printable form.

Strings may be referenced in device, configuration, interface, or other descriptors, and could also be used in vendor-specific ways.

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.

usb_string

Name

`usb_string` — returns ISO 8859-1 version of a string descriptor

Synopsis

```
int usb_string (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

!lin_interrupt ()

Description

This converts the UTF-16LE encoded strings returned by devices, from `usb_get_string_descriptor`, to null-terminated ISO-8859-1 encoded ones that are more usable in most kernel contexts. Note that all characters in the chosen descriptor that can’t be encoded using ISO-8859-1 are converted to the question mark (“?”) character, and this function chooses strings in the first language supported by the device.

The ASCII (or, redundantly, “US-ASCII”) character set is the seven-bit subset of ISO 8859-1. ISO-8859-1 is the eight-bit subset of Unicode, and is appropriate for use many uses of English and several other Western European languages. (But it doesn’t include the “Euro” symbol.)

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).

usb_get_status

Name

`usb_get_status` — issues a GET_STATUS call

Synopsis

```
int usb_get_status (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.

usb_clear_halt

Name

`usb_clear_halt` — tells device to clear endpoint halt/stall condition

Synopsis

```
int usb_clear_halt (struct usb_device * dev, int pipe);
```

Arguments

dev

device whose endpoint is halted

pipe

endpoint “pipe” being cleared

Context

`!lin_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.

usb_set_interface

Name

`usb_set_interface` — Makes a particular alternate setting be current

Synopsis

```
int usb_set_interface (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

!lin_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.

usb_reset_configuration

Name

`usb_reset_configuration` — lightweight device reset

Synopsis

```
int usb_reset_configuration (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 data toggle (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. 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.

usb_register_dev

Name

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

Synopsis

```
int usb_register_dev (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 the `devfs` file for the usb device, if `devfs` is enabled, and 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.

usb_deregister_dev

Name

`usb_deregister_dev` — deregister a USB device's dynamic minor.

Synopsis

```
void usb_deregister_dev (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 cleans up the `devfs` file for the usb device, if `devfs` is enabled, and removes the usb class device from the `sysfs` tree.

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

usb_register

Name

`usb_register` — register a USB driver

Synopsis

```
int usb_register (struct usb_driver * new_driver);
```

Arguments

new_driver

USB operations for the driver

Description

Registers a USB 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 devices. 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.

usb_deregister

Name

`usb_deregister` — unregister a USB driver

Synopsis

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

Arguments

driver

USB operations of the 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.

usb_ifnum_to_if

Name

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

Synopsis

```
struct usb_interface * usb_ifnum_to_if (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!

usb_altnum_to_altsetting

Name

`usb_altnum_to_altsetting` — get the altsetting structure with a given

Synopsis

```
struct usb_host_interface * usb_altnum_to_altsetting (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!

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!

usb_driver_claim_interface

Name

`usb_driver_claim_interface` — bind a driver to an interface

Synopsis

```
int usb_driver_claim_interface (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 and the driver model's `usb_bus_type.subsys` writelock. So driver `probe` entries don't need extra locking, but other call contexts may need to explicitly claim those locks.

usb_driver_release_interface

Name

`usb_driver_release_interface` — unbind a driver from an interface

Synopsis

```
void usb_driver_release_interface (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 and the driver model's `usb_bus_type.subsys.writelock`. So driver `disconnect` entries don't need extra locking, but other call contexts may need to explicitly claim those locks.

usb_match_id

Name

`usb_match_id` — find first `usb_device_id` matching device or interface

Synopsis

```
const struct usb_device_id * usb_match_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` and “`modules.usbmap`”, 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 bDeviceClass).

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.

usb_find_interface

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 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.

usb_alloc_dev

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.

usb_get_dev

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.

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

usb_put_dev

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.

usb_lock_device

Name

`usb_lock_device` — acquire the lock for a usb device structure

Synopsis

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

Arguments

udev

device that's being locked

Description

Use this routine when you don't hold any other device locks; to acquire nested inner locks call `down(&udev->serialize)` directly. This is necessary for proper interaction with `usb_lock_all_devices`.

usb_trylock_device

Name

`usb_trylock_device` — attempt to acquire the lock for a usb device structure

Synopsis

```
int usb_trylock_device (struct usb_device * udev);
```

Arguments

udev

device that's being locked

Description

Don't use this routine if you already hold a device lock; use `down_trylock(&udev->serialize)` instead. This is necessary for proper interaction with `usb_lock_all_devices`.

Returns 1 if successful, 0 if contention.

usb_lock_device_for_reset

Name

`usb_lock_device_for_reset` — cautiously acquire the lock for a

Synopsis

```
int usb_lock_device_for_reset (struct usb_device * udev,
                                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` callback will block waiting for a device reset to complete.

Returns a negative error code for failure, otherwise 1 or 0 to indicate that the device will or will not have to be unlocked. (0 can be returned when an interface is given and is BINDING, because in that case the driver already owns the device lock.)

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` callback will block waiting for a device reset to complete.

Returns a negative error code for failure, otherwise 1 or 0 to indicate that the device will or will not have to be unlocked. (0 can be returned when an interface is given and is BINDING, because in that case the driver already owns the device lock.)

usb_unlock_device

Name

`usb_unlock_device` — release the lock for a usb device structure

Synopsis

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


Arguments

udev

device that's being unlocked

Description

Use this routine when releasing the only device lock you hold; to release inner nested locks call `up(&udev->serialize)` directly. This is necessary for proper interaction with `usb_lock_all_devices`.

usb_find_device

Name

`usb_find_device` — find a specific usb device in the system

Synopsis

```
struct usb_device * usb_find_device (u16 vendor_id, u16
product_id);
```

Arguments

vendor_id

the vendor id of the device to find

product_id

the product id of the device to find

Description

Returns a pointer to a struct `usb_device` if such a specified usb device is present in the system currently. The usage count of the device will be incremented if a device is found. Make sure to call `usb_put_dev` when the caller is finished with the device.

If a device with the specified vendor and product id is not found, NULL is returned.

usb_get_current_frame_number

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

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.

usb_buffer_alloc

Name

`usb_buffer_alloc` — allocate dma-consistent buffer for URB_NO_XXX_DMA_MAP

Synopsis

```
void * usb_buffer_alloc (struct usb_device * dev, size_t size,  
int 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 tying down I/O mapping hardware for long idle periods. The implementation varies between platforms, depending on details of how DMA will work to this device. Using these buffers also helps prevent cacheline sharing problems on architectures where CPU caches are not DMA-coherent.

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

usb_buffer_free

Name

`usb_buffer_free` — free memory allocated with `usb_buffer_alloc`

Synopsis

```
void usb_buffer_free (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.

usb_buffer_map

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

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`.

usb_buffer_dmasync

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

usb_buffer_unmap

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`.

usb_buffer_map_sg

Name

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

Synopsis

```
int usb_buffer_map_sg (struct usb_device * dev, unsigned pipe,
struct scatterlist * sg, int nents);
```

Arguments

dev

device to which the scatterlist will be mapped

pipe

endpoint defining the mapping 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`.

usb_buffer_dmasync_sg

Name

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

Synopsis

```
void usb_buffer_dmasync_sg (struct usb_device * dev, unsigned  
pipe, struct scatterlist * sg, int n_hw_ents);
```


Arguments

dev

device to which the scatterlist will be mapped

pipe

endpoint defining the mapping 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.

usb_buffer_unmap_sg

Name

`usb_buffer_unmap_sg` — free DMA mapping(s) for a scatterlist

Synopsis

```
void usb_buffer_unmap_sg (struct usb_device * dev, unsigned
pipe, struct scatterlist * sg, int n_hw_ents);
```

Arguments

dev

device to which the scatterlist will be mapped

pipe

endpoint defining the mapping 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`.

usb_hub_tt_clear_buffer

Name

`usb_hub_tt_clear_buffer` — clear control/bulk TT state in high speed hub

Synopsis

```
void usb_hub_tt_clear_buffer (struct usb_device * udev, int  
pipe);
```

Arguments

udev

-- undescribed --

pipe

identifies the endpoint of the failed 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.

usb_set_device_state

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_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 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`.

usb_disconnect

Name

`usb_disconnect` — disconnect a device (usbcore-internal)

Synopsis

```
void usb_disconnect (struct usb_device ** pdev);
```

Arguments

pdev

pointer to device being disconnected

Context

!in_interrupt ()

Description

Something got disconnected. Get rid of it and all of its children.

If *pdev is a normal device then the parent hub must already be locked. If *pdev is a root hub then this routine will acquire the usb_bus_list_lock on behalf of the caller.

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

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

usb_suspend_device

Name

usb_suspend_device — suspend a usb device

Synopsis

```
int usb_suspend_device (struct usb_device * udev, u32 state);
```

Arguments

udev

device that's no longer in active use

state

PM_SUSPEND_MEM to suspend

Context

must be able to sleep; device not locked

Description

Suspends a USB device that isn't in active use, conserving power. Devices may wake out of a suspend, if anything important happens, using the remote wakeup mechanism. They may also be taken out of suspend by the host, using `usb_resume_device`. It's also routine to disconnect devices while they are suspended.

Suspending OTG devices may trigger HNP, if that's been enabled between a pair of dual-role devices. That will change roles, such as from A-Host to A-Peripheral or from B-Host back to B-Peripheral.

Returns 0 on success, else negative errno.

usb_resume_device

Name

`usb_resume_device` — re-activate a suspended usb device

Synopsis

```
int usb_resume_device (struct usb_device * udev);
```

Arguments

udev

device to re-activate

Context

must be able to sleep; device not locked

Description

This will re-activate the suspended device, increasing power usage while letting drivers communicate again with its endpoints. USB resume explicitly guarantees that the power session between the host and the device is the same as it was when the device suspended.

Returns 0 on success, else negative errno.

usb_reset_device

Name

`usb_reset_device` — perform a USB port reset to reinitialize a device

Synopsis

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

Arguments

udev

device to reset (not in SUSPENDED or NOTATTACHED state)

Description

WARNING - don't reset any device unless drivers for all of its interfaces are expecting that reset! Maybe some driver->reset method should eventually help ensure sufficient cooperation.

Do a port reset, reassign the device's address, and establish its former operating configuration. If the reset fails, or the device's descriptors change from their values before the reset, or the original configuration and altsettings cannot be restored, a flag will be set telling khubd to pretend the device has been disconnected and then re-connected. All drivers will be unbound, and the device will be re-enumerated and probed all over again.

Returns 0 if the reset succeeded, -ENODEV if the device has been flagged for logical disconnection, or some other negative error code if the reset wasn't even attempted.

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`.

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.

usb_bus_init

Name

`usb_bus_init` — shared initialization code

Synopsis

```
void usb_bus_init (struct usb_bus * bus);
```

Arguments

bus

the bus structure being initialized

Description

This code is used to initialize a `usb_bus` structure, memory for which is separately managed.

usb_alloc_bus

Name

`usb_alloc_bus` — creates a new USB host controller structure

Synopsis

```
struct usb_bus * usb_alloc_bus (struct usb_operations * op);
```

Arguments

op

pointer to a struct `usb_operations` that this bus structure should use

Context

`!in_interrupt`

Description

Creates a USB host controller bus structure with the specified `usb_operations` and initializes all the necessary internal objects.

If no memory is available, `NULL` is returned.

The caller should call `usb_put_bus` when it is finished with the structure.

usb_register_bus

Name

`usb_register_bus` — registers the USB host controller with the usb core

Synopsis

```
int usb_register_bus (struct usb_bus * bus);
```

Arguments

bus

pointer to the bus to register

Context

`!in_interrupt`

Description

Assigns a bus number, and links the controller into usbcore data structures so that it can be seen by scanning the bus list.

usb_deregister_bus

Name

`usb_deregister_bus` — deregisters the USB host controller

Synopsis

```
void usb_deregister_bus (struct usb_bus * bus);
```

Arguments

bus

pointer to the bus to deregister

Context

!in_interrupt

Description

Recycles the bus number, and unlinks the controller from usbcore data structures so that it won't be seen by scanning the bus list.

usb_register_root_hub

Name

`usb_register_root_hub` — called by HCD to register its root hub

Synopsis

```
int usb_register_root_hub (struct usb_device * usb_dev, struct
device * parent_dev);
```

Arguments

usb_dev

the usb root hub device to be registered.

parent_dev

the parent device of this root hub.

Description

The USB host controller calls this function to register the root hub properly with the USB subsystem. It sets up the device properly in the device tree and stores the `root_hub` pointer in the bus structure, then calls `usb_new_device` to register the usb device. It also assigns the root hub's USB address (always 1).

usb_calc_bus_time

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
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.

usb_claim_bandwidth

Name

`usb_claim_bandwidth` — records bandwidth for a periodic transfer

Synopsis

```
void usb_claim_bandwidth (struct usb_device * dev, struct urb  
* urb, int bustime, int isoc);
```

Arguments

dev

source/target of request

urb

request (`urb->dev == dev`)

bustime

bandwidth consumed, in (average) microseconds per frame

isoc

true iff the request is isochronous

Description

Bus bandwidth reservations are recorded purely for diagnostic purposes. HCDs are expected not to overcommit periodic bandwidth, and to record such reservations whenever endpoints are added to the periodic schedule.

FIXME averaging per-frame is suboptimal. Better to sum over the HCD's entire periodic schedule ... 32 frames for OHCI, 1024 for UHCI, settable for EHCI (256/512/1024 frames, default 1024) and have the bus expose how large its periodic schedule is.

usb_release_bandwidth

Name

`usb_release_bandwidth` — reverses effect of `usb_claim_bandwidth`

Synopsis

```
void usb_release_bandwidth (struct usb_device * dev, struct  
urb * urb, int isoc);
```

Arguments

dev

source/target of request

urb

request (urb->dev == dev)

isoc

true iff the request is isochronous

Description

This records that previously allocated bandwidth has been released. Bandwidth is released when endpoints are removed from the host controller's periodic schedule.

usb_bus_start_enum

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

-- undescribed --

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.

usb_hcd_giveback_urb

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, struct pt_regs * regs);
```

Arguments

hcd

host controller returning the URB

urb

urb being returned to the USB device driver.

regs

pt_regs, passed down to the URB completion handler

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.

usb_hcd_irq

Name

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

Synopsis

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

Arguments

irq

the IRQ being raised

__hcd

pointer to the HCD whose IRQ is beinng signaled

r

saved hardware registers

Description

When registering a USB bus through the HCD framework code, use this to handle interrupts. The PCI glue layer does so automatically; only bus glue for non-PCI system busses will need to use this.

usb_create_hcd

Name

`usb_create_hcd` — create and initialize an HCD structure

Synopsis

```
struct usb_hcd * usb_create_hcd (const struct hc_driver *
driver);
```

Arguments

driver

HC driver that will use this 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.

If memory is unavailable, returns `NULL`.

usb_hcd_pci_probe

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`.

usb_hcd_pci_remove

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`.

usb_hcd_pci_suspend

Name

`usb_hcd_pci_suspend` — power management suspend of a PCI-based HCD

Synopsis

```
int usb_hcd_pci_suspend (struct pci_dev * dev, u32 state);
```

Arguments

dev

USB Host Controller being suspended

state

state that the controller is going into

Description

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

usb_hcd_pci_resume

Name

`usb_hcd_pci_resume` — power management resume of a PCI-based HCD

Synopsis

```
int usb_hcd_pci_resume (struct pci_dev * dev);
```

Arguments

dev

USB Host Controller being resumed

Description

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

hcd_buffer_create

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, or returns a negative errno value on error.

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

hcd_buffer_destroy

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; but only devices with kernel drivers show up in *devfs*.

7.1. 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 `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.

7.2. 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)

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`.

7.3. `/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.

Otherwise the main use for this file from programs is to `poll()` it to get notifications of usb devices as they're plugged or unplugged. To see what changed, you'd need to read the file and compare "before" and "after" contents, scan the filesystem, or see its hotplug event.

7.4. `/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.

7.5. 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.

7.6. 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 `khudb` (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.

7.6.1. Management/Status Requests

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

USBDEVFS_CLAIMINTERFACE

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

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

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

USBDEVFS_CONNECTINFO

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

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

File modification time is not updated by this request.

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

USBDEVFS_GETDRIVER

Returns the name of the kernel driver bound to a given interface (a string).

Parameter is a pointer to this structure, which is modified:

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

File modification time is not updated by this request.

USBDEVFS_IOCTL

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

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

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

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



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

File modification time is not updated by this request.

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

USBDEVFS_RELEASEINTERFACE

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

Warning

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

USBDEVFS_RESETEP

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

Warning

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

7.6.2. 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 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.

7.6.3. 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

