USB Gadget API for Linux

David Brownell

USB Gadget API for Linux

David Brownell

Table of Contents

Chapter 1. Introduction

This document presents a Linux-USB "Gadget" kernel mode API, for use within peripherals and other USB devices that embed Linux. It provides an overview of the API structure, and shows how that fits into a system development project. This is the first such API released on Linux to address a number of important problems, including:

- Supports USB 2.0, for high speed devices which can stream data at several dozen megabytes per second.
- Handles devices with dozens of endpoints just as well as ones with just two fixed-function ones. Gadget drivers can be written so they're easy to port to new hardware.
- Flexible enough to expose more complex USB device capabilities such as multiple configurations, multiple interfaces, composite devices, and alternate interface settings.
- USB "On-The-Go" (OTG) support, in conjunction with updates to the Linux-USB host side.
- Sharing data structures and API models with the Linux-USB host side API. This helps the OTG support, and looks forward to more-symmetric frameworks (where the same I/O model is used by both host and device side drivers).
- Minimalist, so it's easier to support new device controller hardware. I/O processing doesn't imply large demands for memory or CPU resources.

Most Linux developers will not be able to use this API, since they have USB "host" hardware in a PC, workstation, or server. Linux users with embedded systems are more likely to have USB peripheral hardware. To distinguish drivers running inside such hardware from the more familiar Linux "USB device drivers", which are host side proxies for the real USB devices, a different term is used: the drivers inside the peripherals are "USB gadget drivers". In USB protocol interactions, the device driver is the master (or "client driver") and the gadget driver is the slave (or "function driver").

The gadget API resembles the host side Linux-USB API in that both use queues of request objects to package I/O buffers, and those requests may be submitted or canceled. They share common definitions for the standard USB *Chapter 9* messages, structures, and constants. Also, both APIs bind and unbind drivers to devices. The APIs differ in detail, since the host side's current URB framework exposes a number of implementation details and assumptions that are inappropriate for a gadget API. While the model for control transfers and configuration management is necessarily different (one side is a hardware-neutral master, the other is a hardware-aware slave), the endpoint I/0 API used here should also be usable for an overhead-reduced host side API.

Chapter 2. Structure of Gadget Drivers

A system running inside a USB peripheral normally has at least three layers inside the kernel to handle USB protocol processing, and may have additional layers in user space code. The "gadget" API is used by the middle layer to interact with the lowest level (which directly handles hardware).

In Linux, from the bottom up, these layers are:

USB Controller Driver

This is the lowest software level. It is the only layer that talks to hardware, through registers, fifos, dma, irqs, and the like. The `<linux/usb/gadget.h>` API abstracts the peripheral controller endpoint hardware. That hardware is exposed through endpoint objects, which accept streams of IN/OUT buffers, and through callbacks that interact with gadget drivers. Since normal USB devices only have one upstream port, they only have one of these drivers. The controller driver can support any number of different gadget drivers, but only one of them can be used at a time.

Examples of such controller hardware include the PCI-based NetChip 2280 USB 2.0 high speed controller, the SA-11x0 or PXA-25x UDC (found within many PDAs), and a variety of other products.

Gadget Driver

The lower boundary of this driver implements hardware-neutral USB functions, using calls to the controller driver. Because such hardware varies widely in capabilities and restrictions, and is used in embedded environments where space is at a premium, the gadget driver is often configured at compile time to work with endpoints supported by one particular controller. Gadget drivers may be portable to several different controllers, using conditional compilation. (Recent kernels substantially simplify the work involved in supporting new hardware, by *autoconfiguring* endpoints automatically for many bulk-oriented drivers.) Gadget driver responsibilities include:

- handling setup requests (ep0 protocol responses) possibly including class-specific functionality
- returning configuration and string descriptors
- (re)setting configurations and interface altsettings, including enabling and configuring endpoints
- handling life cycle events, such as managing bindings to hardware, USB suspend/resume, remote wakeup, and disconnection from the USB host.
- managing IN and OUT transfers on all currently enabled endpoints

Such drivers may be modules of proprietary code, although that approach is discouraged in the Linux community.

Upper Level

Most gadget drivers have an upper boundary that connects to some Linux driver or framework in Linux. Through that boundary flows the data which the gadget driver produces and/or consumes through protocol transfers over USB. Examples include:

- user mode code, using generic (gadgetfs) or application specific files in `/dev`
- networking subsystem (for network gadgets, like the CDC Ethernet Model gadget driver)
- data capture drivers, perhaps video4Linux or a scanner driver; or test and measurement hardware.
- input subsystem (for HID gadgets)
- sound subsystem (for audio gadgets)
- file system (for PTP gadgets)
- block i/o subsystem (for usb-storage gadgets)
- ... and more

Additional Layers
: Other layers may exist. These could include kernel layers, such as network protocol stacks, as well as user mode applications building on standard POSIX system call APIs such as *open()*, *close()*, *read()* and *write()*. On newer systems, POSIX Async I/O calls may be an option. Such user mode code will not necessarily be subject to the GNU General Public License (GPL).

OTG-capable systems will also need to include a standard Linux-USB host side stack, with *usbcore*, one or more *Host Controller Drivers* (HCDs), *USB Device Drivers* to support the OTG "Targeted Peripheral List", and so forth. There will also be an *OTG Controller Driver*, which is visible to gadget and device driver developers only indirectly. That helps the host and device side USB controllers implement the two new OTG protocols (HNP and SRP). Roles switch (host to peripheral, or vice versa) using HNP during USB suspend processing, and SRP can be viewed as a more battery-friendly kind of device wakeup protocol.

Over time, reusable utilities are evolving to help make some gadget driver tasks simpler. For example, building configuration descriptors from vectors of descriptors for the configurations interfaces and endpoints is now automated, and many drivers now use autoconfiguration to choose hardware endpoints and initialize their descriptors. A potential example of particular interest is code implementing standard USB-IF protocols for HID, networking, storage, or audio classes. Some developers are interested in KDB or KGDB hooks, to let target hardware be remotely debugged. Most such USB protocol code doesn't need to be hardware-specific, any more than network protocols like X11, HTTP, or NFS are. Such gadget-side interface drivers should eventually be combined, to implement composite devices.

Chapter 3. Kernel Mode Gadget API

Gadget drivers declare themselves through a *struct usb_gadget_driver*, which is responsible for most parts of enumeration for a *struct usb_gadget*. The response to a set_configuration usually involves enabling one or more of the *struct usb_ep* objects exposed by the gadget, and submitting one or more *struct usb_request* buffers to transfer data. Understand those four data types, and their operations, and you will understand how this API works.

Incomplete Data Type Descriptions

This documentation was prepared using the standard Linux kernel `docproc` tool, which turns text and in-code comments into SGML DocBook and then into usable formats such as HTML or PDF. Other than the "Chapter 9" data types, most of the significant data types and functions are described here.

However, docproc does not understand all the C constructs that are used, so some relevant information is likely omitted from what you are reading. One example of such information is endpoint autoconfiguration. You'll have to read the header file, and use example source code (such as that for "Gadget Zero"), to fully understand the API.

The part of the API implementing some basic driver capabilities is specific to the version of the Linux kernel that's in use. The 2.6 kernel includes a *driver model* framework that has no analogue on earlier kernels; so those parts of the gadget API are not fully portable. (They are implemented on 2.4 kernels, but in a different way.) The driver model state is another part of this API that is ignored by the kerneldoc tools.

The core API does not expose every possible hardware feature, only the most widely available ones. There are significant hardware features, such as device-to-device DMA (without temporary storage in a memory buffer) that would be added using hardware-specific APIs.

This API allows drivers to use conditional compilation to handle endpoint capabilities of different hardware, but doesn't require that. Hardware tends to have arbitrary restrictions, relating to transfer types, addressing, packet sizes, buffering, and availability. As a rule, such differences only matter for "endpoint zero" logic that handles device configuration and management. The API supports limited run-time detection of capabilities, through naming conventions for endpoints. Many drivers will be able to at least partially autoconfigure themselves. In particular, driver init sections will often have endpoint autoconfiguration logic that scans the hardware's list of endpoints to find ones matching the driver requirements (relying on those conventions), to eliminate some of the most common reasons for conditional compilation.

Like the Linux-USB host side API, this API exposes the "chunky" nature of USB messages: I/O requests are in terms of one or more "packets", and packet boundaries are visible to drivers. Compared to RS-232 serial protocols, USB resembles synchronous protocols like HDLC (N bytes per frame, multipoint addressing, host as the primary station and devices as secondary stations) more than asynchronous ones (tty style: 8 data bits per frame, no parity, one stop bit). So for example the controller drivers won't buffer two single byte writes into a single two-byte USB IN packet, although gadget drivers may do so when they implement protocols where packet boundaries (and "short packets") are not significant.

Driver Life Cycle

Gadget drivers make endpoint I/O requests to hardware without needing to know many details of the hardware, but driver setup/configuration code needs to handle some differences. Use the API like this:

1. Register a driver for the particular device side usb controller hardware, such as the net2280 on PCI (USB 2.0), sa11x0 or pxa25x as found in Linux PDAs, and so on. At this point the device is logically in the USB ch9 initial state ("attached"), drawing no power and not usable (since it does not yet support enumeration). Any host should not see the device, since it's not activated the data line pullup used by the host to detect a device, even if VBUS power is available.

2. Register a gadget driver that implements some higher level device function. That will then bind() to a usb_gadget, which activates the data line pullup sometime after detecting VBUS.

3. The hardware driver can now start enumerating. The steps it handles are to accept USB power and set_address requests. Other steps are handled by the gadget driver. If the gadget driver module is unloaded before the host starts to enumerate, steps before step 7 are skipped.

4. The gadget driver's setup() call returns usb descriptors, based both on what the bus interface hardware provides and on the functionality being implemented. That can involve alternate settings or configurations, unless the hardware prevents such operation. For OTG devices, each configuration descriptor includes an OTG descriptor.

5. The gadget driver handles the last step of enumeration, when the USB host issues a set_configuration call. It enables all endpoints used in that configuration, with all interfaces in their default settings. That involves using a list of the hardware's endpoints, enabling each endpoint according to its descriptor. It may also involve using `usb_gadget_vbus_draw` to let more power be drawn from VBUS, as allowed by that configuration. For OTG devices, setting a configuration may also involve reporting HNP capabilities through a user interface.

6. Do real work and perform data transfers, possibly involving changes to interface settings or switching to new configurations, until the device is disconnect()ed from the host. Queue any number of transfer requests to each endpoint. It may be suspended and resumed several times before being disconnected. On disconnect, the drivers go back to step 3 (above).

7. When the gadget driver module is being unloaded, the driver unbind() callback is issued. That lets the controller driver be unloaded.

Drivers will normally be arranged so that just loading the gadget driver module (or statically linking it into a Linux kernel) allows the peripheral device to be enumerated, but some drivers will defer enumeration until some higher level component (like a user mode daemon) enables it. Note that at this lowest level there are no policies about how ep0 configuration logic is implemented, except that it should obey USB specifications. Such issues are in the domain of gadget drivers, including knowing about implementation constraints imposed by some USB controllers or understanding that composite devices might happen to be built by integrating reusable components.

Note that the lifecycle above can be slightly different for OTG devices. Other than providing an additional OTG descriptor in each configuration, only the HNP-related differences are particularly visible to driver code. They involve reporting requirements during the SET_CONFIGURATION request, and the option to invoke HNP during some suspend callbacks. Also, SRP changes the semantics of `usb_gadget_wakeup` slightly.

USB 2.0 Chapter 9 Types and Constants

Gadget drivers rely on common USB structures and constants defined in the `<linux/usb/ch9.h>` header file, which is standard in Linux 2.6 kernels. These are the same types and constants used by host side drivers (and usbcore).

Name

usb_speed_string — Returns human readable-name of the speed.

Synopsis

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

Arguments

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

Name

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

Synopsis

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

Arguments

dev Pointer to the given USB controller device

Description

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

Name

usb_state_string — Returns human readable name for the state.

Synopsis

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

Arguments

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

Core Objects and Methods

These are declared in `<linux/usb/gadget.h>`, and are used by gadget drivers to interact with USB peripheral controller drivers.

Name

struct usb_request — describes one i/o request

Synopsis

```
struct usb_request {
  void * buf;
  unsigned length;
  dma_addr_t dma;
  struct scatterlist * sg;
  unsigned num_sgs;
  unsigned num_mapped_sgs;
  unsigned stream_id:16;
  unsigned no_interrupt:1;
  unsigned zero:1;
  unsigned short_not_ok:1;
  void (* complete) (struct usb_ep *ep,struct usb_request *req);
  void * context;
  struct list_head list;
  int status;
  unsigned actual;
};
```

Members

buf
: Buffer used for data. Always provide this; some controllers only use PIO, or don't use DMA for some endpoints.

length
: Length of that data

dma
: DMA address corresponding to 'buf'. If you don't set this field, and the usb controller needs one, it is responsible for mapping and unmapping the buffer.

sg
: a scatterlist for SG-capable controllers.

num_sgs
: number of SG entries

num_mapped_sgs
: number of SG entries mapped to DMA (internal)

stream_id
: The stream id, when USB3.0 bulk streams are being used

no_interrupt
: If true, hints that no completion irq is needed. Helpful sometimes with deep request queues that are handled directly by DMA controllers.

zero
: If true, when writing data, makes the last packet be “short” by adding a zero length packet as needed;

short_not_ok
: When reading data, makes short packets be treated as errors (queue stops advancing till cleanup).

complete
: Function called when request completes, so this request and its buffer may be re-used. The function will always be called with interrupts disabled, and it must not sleep. Reads terminate with a short packet, or when the buffer fills, whichever comes first. When writes terminate, some data bytes will usually still be in

flight (often in a hardware fifo). Errors (for reads or writes) stop the queue from advancing until the completion function returns, so that any transfers invalidated by the error may first be dequeued.

context
For use by the completion callback

list
For use by the gadget driver.

status
Reports completion code, zero or a negative errno. Normally, faults block the transfer queue from advancing until the completion callback returns. Code “-ESHUTDOWN” indicates completion caused by device disconnect, or when the driver disabled the endpoint.

actual
Reports bytes transferred to/from the buffer. For reads (OUT transfers) this may be less than the requested length. If the short_not_ok flag is set, short reads are treated as errors even when status otherwise indicates successful completion. Note that for writes (IN transfers) some data bytes may still reside in a device-side FIFO when the request is reported as complete.

Description

These are allocated/freed through the endpoint they're used with. The hardware's driver can add extra per-request data to the memory it returns, which often avoids separate memory allocations (potential failures), later when the request is queued.

Request flags affect request handling, such as whether a zero length packet is written (the “zero” flag), whether a short read should be treated as an error (blocking request queue advance, the “short_not_ok” flag), or hinting that an interrupt is not required (the “no_interrupt” flag, for use with deep request queues).

Bulk endpoints can use any size buffers, and can also be used for interrupt transfers. interrupt-only endpoints can be much less functional.

NOTE

this is analogous to 'struct urb' on the host side, except that it's thinner and promotes more pre-allocation.

Name

struct usb_ep_caps — endpoint capabilities description

Synopsis

```
struct usb_ep_caps {
  unsigned type_control:1;
  unsigned type_iso:1;
  unsigned type_bulk:1;
  unsigned type_int:1;
  unsigned dir_in:1;
  unsigned dir_out:1;
};
```

Members

type_control
: Endpoint supports control type (reserved for ep0).

type_iso
: Endpoint supports isochronous transfers.

type_bulk
: Endpoint supports bulk transfers.

type_int
: Endpoint supports interrupt transfers.

dir_in
: Endpoint supports IN direction.

dir_out
: Endpoint supports OUT direction.

Name

struct usb_ep — device side representation of USB endpoint

Synopsis

```
struct usb_ep {
  void * driver_data;
  const char * name;
  const struct usb_ep_ops * ops;
  struct list_head ep_list;
  struct usb_ep_caps caps;
  unsigned maxpacket:16;
  unsigned maxpacket_limit:16;
  unsigned max_streams:16;
  unsigned mult:2;
  unsigned maxburst:5;
  u8 address;
  const struct usb_endpoint_descriptor * desc;
  const struct usb_ss_ep_comp_descriptor * comp_desc;
};
```

Members

driver_data
: for use by the gadget driver.

name
: identifier for the endpoint, such as “ep-a” or “ep9in-bulk”

ops
: Function pointers used to access hardware-specific operations.

ep_list
: the gadget's ep_list holds all of its endpoints

caps
: The structure describing types and directions supported by endoint.

maxpacket
: The maximum packet size used on this endpoint. The initial value can sometimes be reduced (hardware allowing), according to the endpoint descriptor used to configure the endpoint.

maxpacket_limit
: The maximum packet size value which can be handled by this endpoint. It's set once by UDC driver when endpoint is initialized, and should not be changed. Should not be confused with maxpacket.

max_streams
: The maximum number of streams supported by this EP (0 - 16, actual number is 2^n)

mult
: multiplier, 'mult' value for SS Isoc EPs

maxburst
: the maximum number of bursts supported by this EP (for usb3)

address
: used to identify the endpoint when finding descriptor that matches connection speed

desc
: endpoint descriptor. This pointer is set before the endpoint is enabled and remains valid until the endpoint is disabled.

comp_desc — In case of SuperSpeed support, this is the endpoint companion descriptor that is used to configure the endpoint

Description

the bus controller driver lists all the general purpose endpoints in gadget->ep_list. the control endpoint (gadget->ep0) is not in that list, and is accessed only in response to a driver `setup` callback.

Name

struct usb_gadget — represents a usb slave device

Synopsis

```
struct usb_gadget {
  struct work_struct work;
  struct usb_udc * udc;
  const struct usb_gadget_ops * ops;
  struct usb_ep * ep0;
  struct list_head ep_list;
  enum usb_device_speed speed;
  enum usb_device_speed max_speed;
  enum usb_device_state state;
  const char * name;
  struct device dev;
  unsigned out_epnum;
  unsigned in_epnum;
  unsigned mA;
  struct usb_otg_caps * otg_caps;
  unsigned sg_supported:1;
  unsigned is_otg:1;
  unsigned is_a_peripheral:1;
  unsigned b_hnp_enable:1;
  unsigned a_hnp_support:1;
  unsigned a_alt_hnp_support:1;
  unsigned hnp_polling_support:1;
  unsigned host_request_flag:1;
  unsigned quirk_ep_out_aligned_size:1;
  unsigned is_selfpowered:1;
  unsigned deactivated:1;
  unsigned connected:1;
};
```

Members

work
: (internal use) Workqueue to be used for `sysfs_notify`

udc
: struct usb_udc pointer for this gadget

ops
: Function pointers used to access hardware-specific operations.

ep0
: Endpoint zero, used when reading or writing responses to driver `setup` requests

ep_list
: List of other endpoints supported by the device.

speed
: Speed of current connection to USB host.

max_speed
: Maximal speed the UDC can handle. UDC must support this and all slower speeds.

state
: the state we are now (attached, suspended, configured, etc)

name	Identifies the controller hardware type. Used in diagnostics and sometimes configuration.
dev	Driver model state for this abstract device.
out_epnum	last used out ep number
in_epnum	last used in ep number
mA	last set mA value
otg_caps	OTG capabilities of this gadget.
sg_supported	true if we can handle scatter-gather
is_otg	True if the USB device port uses a Mini-AB jack, so that the gadget driver must provide a USB OTG descriptor.
is_a_peripheral	False unless is_otg, the “A” end of a USB cable is in the Mini-AB jack, and HNP has been used to switch roles so that the “A” device currently acts as A-Peripheral, not A-Host.
b_hnp_enable	OTG device feature flag, indicating that the A-Host enabled HNP support.
a_hnp_support	OTG device feature flag, indicating that the A-Host supports HNP at this port.
a_alt_hnp_support	OTG device feature flag, indicating that the A-Host only supports HNP on a different root port.
hnp_polling_support	OTG device feature flag, indicating if the OTG device in peripheral mode can support HNP polling.
host_request_flag	OTG device feature flag, indicating if A-Peripheral or B-Peripheral wants to take host role.
quirk_ep_out_aligned_size	epout requires buffer size to be aligned to MaxPacketSize.
is_selfpowered	if the gadget is self-powered.
deactivated	True if gadget is deactivated - in deactivated state it cannot be connected.
connected	True if gadget is connected.

Description

Gadgets have a mostly-portable “gadget driver” implementing device functions, handling all usb configurations and interfaces. Gadget drivers talk to hardware-specific code indirectly, through ops vectors. That insulates the gadget driver from hardware details, and packages the hardware endpoints through generic i/o queues. The “usb_gadget” and “usb_ep” interfaces provide that insulation from the hardware.

Except for the driver data, all fields in this structure are read-only to the gadget driver. That driver data is part of the “driver model” infrastructure in 2.6 (and later) kernels, and for earlier systems is grouped in a similar structure that's not known to the rest of the kernel.

Values of the three OTG device feature flags are updated before the `setup` call corresponding to USB_REQ_SET_CONFIGURATION, and before driver `suspend` calls. They are valid only when is_otg, and when the device is acting as a B-Peripheral (so is_a_peripheral is false).

Name

usb_ep_align_maybe — returns *len* aligned to ep's maxpacketsize if gadget requires quirk_ep_out_aligned_size, otherwise reguens len.

Synopsis

```
size_t usb_ep_align_maybe (struct usb_gadget * g, struct usb_ep * ep,
size_t len);
```

Arguments

g controller to check for quirk

ep the endpoint whose maxpacketsize is used to align *len*

len buffer size's length to align to *ep*'s maxpacketsize

Description

This helper is used in case it's required for any reason to check and maybe align buffer's size to an ep's maxpacketsize.

Name

gadget_is_altset_supported — return true iff the hardware supports altsettings

Synopsis

```
int gadget_is_altset_supported (struct usb_gadget * g);
```

Arguments

g
: controller to check for quirk

Name

gadget_is_stall_supported — return true iff the hardware supports stalling

Synopsis

```
int gadget_is_stall_supported (struct usb_gadget * g);
```

Arguments

g
: controller to check for quirk

Name

gadget_is_zlp_supported — return true iff the hardware supports zlp

Synopsis

```
int gadget_is_zlp_supported (struct usb_gadget * g);
```

Arguments

g controller to check for quirk

Name

gadget_is_dualspeed — return true iff the hardware handles high speed

Synopsis

```
int gadget_is_dualspeed (struct usb_gadget * g);
```

Arguments

g
: controller that might support both high and full speeds

Name

gadget_is_superspeed — return true if the hardware handles superspeed

Synopsis

```
int gadget_is_superspeed (struct usb_gadget * g);
```

Arguments

g controller that might support superspeed

Name

gadget_is_superspeed_plus — return true if the hardware handles superspeed plus

Synopsis

```
int gadget_is_superspeed_plus (struct usb_gadget * g);
```

Arguments

g controller that might support superspeed plus

Name

gadget_is_otg — return true iff the hardware is OTG-ready

Synopsis

```
int gadget_is_otg (struct usb_gadget * g);
```

Arguments

g
: controller that might have a Mini-AB connector

Description

This is a runtime test, since kernels with a USB-OTG stack sometimes run on boards which only have a Mini-B (or Mini-A) connector.

Name

struct usb_gadget_driver — driver for usb 'slave' devices

Synopsis

```
struct usb_gadget_driver {
  char * function;
  enum usb_device_speed max_speed;
  int (* bind) (struct usb_gadget *gadget,struct usb_gadget_driver *driver);
  void (* unbind) (struct usb_gadget *);
  int (* setup) (struct usb_gadget *,const struct usb_ctrlrequest *);
  void (* disconnect) (struct usb_gadget *);
  void (* suspend) (struct usb_gadget *);
  void (* resume) (struct usb_gadget *);
  void (* reset) (struct usb_gadget *);
  struct device_driver driver;
  char * udc_name;
  struct list_head pending;
  unsigned match_existing_only:1;
};
```

Members

function
: String describing the gadget's function

max_speed
: Highest speed the driver handles.

bind
: the driver's bind callback

unbind
: Invoked when the driver is unbound from a gadget, usually from rmmod (after a disconnect is reported). Called in a context that permits sleeping.

setup
: Invoked for ep0 control requests that aren't handled by the hardware level driver. Most calls must be handled by the gadget driver, including descriptor and configuration management. The 16 bit members of the setup data are in USB byte order. Called in_interrupt; this may not sleep. Driver queues a response to ep0, or returns negative to stall.

disconnect
: Invoked after all transfers have been stopped, when the host is disconnected. May be called in_interrupt; this may not sleep. Some devices can't detect disconnect, so this might not be called except as part of controller shutdown.

suspend
: Invoked on USB suspend. May be called in_interrupt.

resume
: Invoked on USB resume. May be called in_interrupt.

reset
: Invoked on USB bus reset. It is mandatory for all gadget drivers and should be called in_interrupt.

driver
: Driver model state for this driver.

udc_name
: A name of UDC this driver should be bound to. If udc_name is NULL, this driver will be bound to any available UDC.

pending	UDC core private data used for deferred probe of this driver.
match_existing_only	If udc is not found, return an error and don't add this gadget driver to list of pending driver

Description

Devices are disabled till a gadget driver successfully `bind`s, which means the driver will handle `setup` requests needed to enumerate (and meet “chapter 9” requirements) then do some useful work.

If gadget->is_otg is true, the gadget driver must provide an OTG descriptor during enumeration, or else fail the `bind` call. In such cases, no USB traffic may flow until both `bind` returns without having called `usb_gadget_disconnect`, and the USB host stack has initialized.

Drivers use hardware-specific knowledge to configure the usb hardware. endpoint addressing is only one of several hardware characteristics that are in descriptors the ep0 implementation returns from `setup` calls.

Except for ep0 implementation, most driver code shouldn't need change to run on top of different usb controllers. It'll use endpoints set up by that ep0 implementation.

The usb controller driver handles a few standard usb requests. Those include set_address, and feature flags for devices, interfaces, and endpoints (the get_status, set_feature, and clear_feature requests).

Accordingly, the driver's `setup` callback must always implement all get_descriptor requests, returning at least a device descriptor and a configuration descriptor. Drivers must make sure the endpoint descriptors match any hardware constraints. Some hardware also constrains other descriptors. (The pxa250 allows only configurations 1, 2, or 3).

The driver's `setup` callback must also implement set_configuration, and should also implement set_interface, get_configuration, and get_interface. Setting a configuration (or interface) is where endpoints should be activated or (config 0) shut down.

(Note that only the default control endpoint is supported. Neither hosts nor devices generally support control traffic except to ep0.)

Most devices will ignore USB suspend/resume operations, and so will not provide those callbacks. However, some may need to change modes when the host is not longer directing those activities. For example, local controls (buttons, dials, etc) may need to be re-enabled since the (remote) host can't do that any longer; or an error state might be cleared, to make the device behave identically whether or not power is maintained.

Name

usb_gadget_probe_driver — probe a gadget driver

Synopsis

```
int usb_gadget_probe_driver (struct usb_gadget_driver * driver);
```

Arguments

driver the driver being registered

Context

can sleep

Description

Call this in your gadget driver's module initialization function, to tell the underlying usb controller driver about your driver. The *bind*() function will be called to bind it to a gadget before this registration call returns. It's expected that the *bind*() function will be in init sections.

Name

usb_gadget_unregister_driver — unregister a gadget driver

Synopsis

```
int usb_gadget_unregister_driver (struct usb_gadget_driver * driver);
```

Arguments

driver the driver being unregistered

Context

can sleep

Description

Call this in your gadget driver's module cleanup function, to tell the underlying usb controller that your driver is going away. If the controller is connected to a USB host, it will first `disconnect`. The driver is also requested to `unbind` and clean up any device state, before this procedure finally returns. It's expected that the `unbind` functions will in in exit sections, so may not be linked in some kernels.

Name

struct usb_string — wraps a C string and its USB id

Synopsis

```
struct usb_string {
  u8 id;
  const char * s;
};
```

Members

id the (nonzero) ID for this string

s the string, in UTF-8 encoding

Description

If you're using `usb_gadget_get_string`, use this to wrap a string together with its ID.

Name

struct usb_gadget_strings — a set of USB strings in a given language

Synopsis

```
struct usb_gadget_strings {
  u16 language;
  struct usb_string * strings;
};
```

Members

language — identifies the strings' language (0x0409 for en-us)

strings — array of strings with their ids

Description

If you're using `usb_gadget_get_string`, use this to wrap all the strings for a given language.

Name

usb_free_descriptors — free descriptors returned by `usb_copy_descriptors`

Synopsis

```
void usb_free_descriptors (struct usb_descriptor_header ** v);
```

Arguments

v vector of descriptors

Optional Utilities

The core API is sufficient for writing a USB Gadget Driver, but some optional utilities are provided to simplify common tasks. These utilities include endpoint autoconfiguration.

Name

usb_gadget_get_string — fill out a string descriptor

Synopsis

```
int usb_gadget_get_string (struct usb_gadget_strings * table, int id,
u8 * buf);
```

Arguments

table of c strings encoded using UTF-8

id string id, from low byte of wValue in get string descriptor

buf at least 256 bytes, must be 16-bit aligned

Description

Finds the UTF-8 string matching the ID, and converts it into a string descriptor in utf16-le. Returns length of descriptor (always even) or negative errno

If your driver needs stings in multiple languages, you'll probably “switch (wIndex) { ... }” in your ep0 string descriptor logic, using this routine after choosing which set of UTF-8 strings to use. Note that US-ASCII is a strict subset of UTF-8; any string bytes with the eighth bit set will be multibyte UTF-8 characters, not ISO-8859/1 characters (which are also widely used in C strings).

Name

usb_descriptor_fillbuf — fill buffer with descriptors

Synopsis

```
int usb_descriptor_fillbuf (void * buf, unsigned buflen, const struct
usb_descriptor_header ** src);
```

Arguments

buf Buffer to be filled

buflen Size of buf

src Array of descriptor pointers, terminated by null pointer.

Description

Copies descriptors into the buffer, returning the length or a negative error code if they can't all be copied. Useful when assembling descriptors for an associated set of interfaces used as part of configuring a composite device; or in other cases where sets of descriptors need to be marshaled.

Name

usb_gadget_config_buf — builts a complete configuration descriptor

Synopsis

```
int usb_gadget_config_buf (const struct usb_config_descriptor * config,
void * buf, unsigned length, const struct usb_descriptor_header ** desc);
```

Arguments

config
: Header for the descriptor, including characteristics such as power requirements and number of interfaces.

buf
: Buffer for the resulting configuration descriptor.

length
: Length of buffer. If this is not big enough to hold the entire configuration descriptor, an error code will be returned.

desc
: Null-terminated vector of pointers to the descriptors (interface, endpoint, etc) defining all functions in this device configuration.

Description

This copies descriptors into the response buffer, building a descriptor for that configuration. It returns the buffer length or a negative status code. The config.wTotalLength field is set to match the length of the result, but other descriptor fields (including power usage and interface count) must be set by the caller.

Gadget drivers could use this when constructing a config descriptor in response to USB_REQ_GET_DESCRIPTOR. They will need to patch the resulting bDescriptorType value if USB_DT_OTHER_SPEED_CONFIG is needed.

Name

usb_copy_descriptors — copy a vector of USB descriptors

Synopsis

```
struct usb_descriptor_header ** usb_copy_descriptors (struct
usb_descriptor_header ** src);
```

Arguments

src null-terminated vector to copy

Context

initialization code, which may sleep

Description

This makes a copy of a vector of USB descriptors. Its primary use is to support usb_function objects which can have multiple copies, each needing different descriptors. Functions may have static tables of descriptors, which are used as templates and customized with identifiers (for interfaces, strings, endpoints, and more) as needed by a given function instance.

Composite Device Framework

The core API is sufficient for writing drivers for composite USB devices (with more than one function in a given configuration), and also multi-configuration devices (also more than one function, but not necessarily sharing a given configuration). There is however an optional framework which makes it easier to reuse and combine functions.

Devices using this framework provide a *struct usb_composite_driver*, which in turn provides one or more *struct usb_configuration* instances. Each such configuration includes at least one *struct usb_function*, which packages a user visible role such as "network link" or "mass storage device". Management functions may also exist, such as "Device Firmware Upgrade".

Name

struct usb_os_desc_ext_prop — describes one “Extended Property”

Synopsis

```
struct usb_os_desc_ext_prop {
  struct list_head entry;
  u8 type;
  int name_len;
  char * name;
  int data_len;
  char * data;
  struct config_item item;
};
```

Members

entry
: used to keep a list of extended properties

type
: Extended Property type

name_len
: Extended Property unicode name length, including terminating '\0'

name
: Extended Property name

data_len
: Length of Extended Property blob (for unicode store double len)

data
: Extended Property blob

item
: Represents this Extended Property in configfs

Name

struct usb_os_desc — describes OS descriptors associated with one interface

Synopsis

```
struct usb_os_desc {
  char * ext_compat_id;
  struct list_head ext_prop;
  int ext_prop_len;
  int ext_prop_count;
  struct mutex * opts_mutex;
  struct config_group group;
  struct module * owner;
};
```

Members

ext_compat_id
: 16 bytes of “Compatible ID” and “Subcompatible ID”

ext_prop
: Extended Properties list

ext_prop_len
: Total length of Extended Properties blobs

ext_prop_count
: Number of Extended Properties

opts_mutex
: Optional mutex protecting config data of a usb_function_instance

group
: Represents OS descriptors associated with an interface in configfs

owner
: Module associated with this OS descriptor

Name

struct usb_os_desc_table — describes OS descriptors associated with one interface of a usb_function

Synopsis

```
struct usb_os_desc_table {
  int if_id;
  struct usb_os_desc * os_desc;
};
```

Members

if_id
: Interface id

os_desc
: "Extended Compatibility ID“ and ”Extended Properties" of the interface

Description

Each interface can have at most one “Extended Compatibility ID” and a number of “Extended Properties”.

Name

struct usb_function — describes one function of a configuration

Synopsis

```
struct usb_function {
  const char * name;
  struct usb_gadget_strings ** strings;
  struct usb_descriptor_header ** fs_descriptors;
  struct usb_descriptor_header ** hs_descriptors;
  struct usb_descriptor_header ** ss_descriptors;
  struct usb_descriptor_header ** ssp_descriptors;
  struct usb_configuration * config;
  struct usb_os_desc_table * os_desc_table;
  unsigned os_desc_n;
  int (* bind) (struct usb_configuration *,struct usb_function *);
  void (* unbind) (struct usb_configuration *,struct usb_function *);
  void (* free_func) (struct usb_function *f);
  struct module * mod;
  int (* set_alt) (struct usb_function *,unsigned interface, unsigned alt);
  int (* get_alt) (struct usb_function *,unsigned interface);
  void (* disable) (struct usb_function *);
  int (* setup) (struct usb_function *,const struct usb_ctrlrequest *);
  bool (* req_match) (struct usb_function *,const struct usb_ctrlrequest *);
  void (* suspend) (struct usb_function *);
  void (* resume) (struct usb_function *);
  int (* get_status) (struct usb_function *);
  int (* func_suspend) (struct usb_function *,u8 suspend_opt);
};
```

Members

name
: For diagnostics, identifies the function.

strings
: tables of strings, keyed by identifiers assigned during `bind` and by language IDs provided in control requests

fs_descriptors
: Table of full (or low) speed descriptors, using interface and string identifiers assigned during *bind*(). If this pointer is null, the function will not be available at full speed (or at low speed).

hs_descriptors
: Table of high speed descriptors, using interface and string identifiers assigned during *bind*(). If this pointer is null, the function will not be available at high speed.

ss_descriptors
: Table of super speed descriptors, using interface and string identifiers assigned during *bind*(). If this pointer is null after initiation, the function will not be available at super speed.

ssp_descriptors
: Table of super speed plus descriptors, using interface and string identifiers assigned during *bind*(). If this pointer is null after initiation, the function will not be available at super speed plus.

config	assigned when *usb_add_function*() is called; this is the configuration with which this function is associated.
os_desc_table	Table of (interface id, os descriptors) pairs. The function can expose more than one interface. If an interface is a member of an IAD, only the first interface of IAD has its entry in the table.
os_desc_n	Number of entries in os_desc_table
bind	Before the gadget can register, all of its functions `bind` to the available resources including string and interface identifiers used in interface or class descriptors; endpoints; I/O buffers; and so on.
unbind	Reverses *bind*; called as a side effect of unregistering the driver which added this function.
free_func	free the struct usb_function.
mod	(internal) points to the module that created this structure.
set_alt	(REQUIRED) Reconfigures altsettings; function drivers may initialize usb_ep.driver data at this time (when it is used). Note that setting an interface to its current altsetting resets interface state, and that all interfaces have a disabled state.
get_alt	Returns the active altsetting. If this is not provided, then only altsetting zero is supported.
disable	(REQUIRED) Indicates the function should be disabled. Reasons include host resetting or reconfiguring the gadget, and disconnection.
setup	Used for interface-specific control requests.
req_match	Tests if a given class request can be handled by this function.
suspend	Notifies functions when the host stops sending USB traffic.
resume	Notifies functions when the host restarts USB traffic.
get_status	Returns function status as a reply to `GetStatus` request when the recipient is Interface.
func_suspend	callback to be called when SetFeature(FUNCTION_SUSPEND) is reseived

Description

A single USB function uses one or more interfaces, and should in most cases support operation at both full and high speeds. Each function is associated by *usb_add_function*() with a one configuration; that function causes *bind*() to be called so resources can be allocated as part of setting up a gadget driver. Those resources include endpoints, which should be allocated using *usb_ep_autoconfig*().

To support dual speed operation, a function driver provides descriptors for both high and full speed operation. Except in rare cases that don't involve bulk endpoints, each speed needs different endpoint descriptors.

Function drivers choose their own strategies for managing instance data. The simplest strategy just declares it "static', which means the function can only be activated once. If the function needs to be exposed in more than one configuration at a given speed, it needs to support multiple usb_function structures (one for each configuration).

A more complex strategy might encapsulate a `usb_function` structure inside a driver-specific instance structure to allows multiple activations. An example of multiple activations might be a CDC ACM function that supports two or more distinct instances within the same configuration, providing several independent logical data links to a USB host.

Name

struct usb_configuration — represents one gadget configuration

Synopsis

```
struct usb_configuration {
  const char * label;
  struct usb_gadget_strings ** strings;
  const struct usb_descriptor_header ** descriptors;
  void (* unbind) (struct usb_configuration *);
  int (* setup) (struct usb_configuration *,const struct usb_ctrlrequest *);
  u8 bConfigurationValue;
  u8 iConfiguration;
  u8 bmAttributes;
  u16 MaxPower;
  struct usb_composite_dev * cdev;
};
```

Members

label
: For diagnostics, describes the configuration.

strings
: Tables of strings, keyed by identifiers assigned during *bind*() and by language IDs provided in control requests.

descriptors
: Table of descriptors preceding all function descriptors. Examples include OTG and vendor-specific descriptors.

unbind
: Reverses *bind*; called as a side effect of unregistering the driver which added this configuration.

setup
: Used to delegate control requests that aren't handled by standard device infrastructure or directed at a specific interface.

bConfigurationValue
: Copied into configuration descriptor.

iConfiguration
: Copied into configuration descriptor.

bmAttributes
: Copied into configuration descriptor.

MaxPower
: Power consumtion in mA. Used to compute bMaxPower in the configuration descriptor after considering the bus speed.

cdev
: assigned by *usb_add_config*() before calling *bind*(); this is the device associated with this configuration.

Description

Configurations are building blocks for gadget drivers structured around function drivers. Simple USB gadgets require only one function and one configuration, and handle dual-speed hardware by always providing the same functionality. Slightly more complex gadgets may have more than one single-function configuration at a given speed; or have configurations that only work at one speed.

Composite devices are, by definition, ones with configurations which include more than one function.

The lifecycle of a usb_configuration includes allocation, initialization of the fields described above, and calling *`usb_add_config`*() to set up internal data and bind it to a specific device. The configuration's *`bind`*() method is then used to initialize all the functions and then call *`usb_add_function`*() for them.

Those functions would normally be independent of each other, but that's not mandatory. CDC WMC devices are an example where functions often depend on other functions, with some functions subsidiary to others. Such interdependency may be managed in any way, so long as all of the descriptors complete by the time the composite driver returns from its `bind` routine.

Name

struct usb_composite_driver — groups configurations into a gadget

Synopsis

```
struct usb_composite_driver {
  const char * name;
  const struct usb_device_descriptor * dev;
  struct usb_gadget_strings ** strings;
  enum usb_device_speed max_speed;
  unsigned needs_serial:1;
  int (* bind) (struct usb_composite_dev *cdev);
  int (* unbind) (struct usb_composite_dev *);
  void (* disconnect) (struct usb_composite_dev *);
  void (* suspend) (struct usb_composite_dev *);
  void (* resume) (struct usb_composite_dev *);
  struct usb_gadget_driver gadget_driver;
};
```

Members

name
: For diagnostics, identifies the driver.

dev
: Template descriptor for the device, including default device identifiers.

strings
: tables of strings, keyed by identifiers assigned during *bind* and language IDs provided in control requests. Note: The first entries are predefined. The first entry that may be used is USB_GADGET_FIRST_AVAIL_IDX

max_speed
: Highest speed the driver supports.

needs_serial
: set to 1 if the gadget needs userspace to provide a serial number. If one is not provided, warning will be printed.

bind
: (REQUIRED) Used to allocate resources that are shared across the whole device, such as string IDs, and add its configurations using *usb_add_config*(). This may fail by returning a negative errno value; it should return zero on successful initialization.

unbind
: Reverses *bind*; called as a side effect of unregistering this driver.

disconnect
: optional driver disconnect method

suspend
: Notifies when the host stops sending USB traffic, after function notifications

resume
: Notifies configuration when the host restarts USB traffic, before function notifications

gadget_driver
: Gadget driver controlling this driver

Description

Devices default to reporting self powered operation. Devices which rely on bus powered operation should report this in their *bind* method.

Before returning from *bind*, various fields in the template descriptor may be overridden. These include the idVendor/idProduct/bcdDevice values normally to bind the appropriate host side driver, and the three strings (iManufacturer, iProduct, iSerialNumber) normally used to provide user meaningful device identifiers. (The strings will not be defined unless they are defined in *dev* and *strings*.) The correct ep0 maxpacket size is also reported, as defined by the underlying controller driver.

Name

module_usb_composite_driver — Helper macro for registering a USB gadget composite driver

Synopsis

`module_usb_composite_driver` `(` *`__usb_composite_driver`*`);`

Arguments

`__usb_composite_driver` usb_composite_driver struct

Description

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

Name

struct usb_composite_dev — represents one composite usb gadget

Synopsis

```
struct usb_composite_dev {
  struct usb_gadget * gadget;
  struct usb_request * req;
  struct usb_request * os_desc_req;
  struct usb_configuration * config;
  u8 qw_sign[OS_STRING_QW_SIGN_LEN];
  u8 b_vendor_code;
  struct usb_configuration * os_desc_config;
  unsigned int use_os_string:1;
};
```

Members

gadget	read-only, abstracts the gadget's usb peripheral controller
req	used for control responses; buffer is pre-allocated
os_desc_req	used for OS descriptors responses; buffer is pre-allocated
config	the currently active configuration
qw_sign[OS_STRING_QW_SIGN_LEN]	qwSignature part of the OS string
b_vendor_code	bMS_VendorCode part of the OS string
os_desc_config	the configuration to be used with OS descriptors
use_os_string	false by default, interested gadgets set it

Description

One of these devices is allocated and initialized before the associated device driver's `bind` is called.

OPEN ISSUE: it appears that some WUSB devices will need to be built by combining a normal (wired) gadget with a wireless one. This revision of the gadget framework should probably try to make sure doing that won't hurt too much.

One notion for how to handle Wireless USB devices involves: (a) a second gadget here, discovery mechanism TBD, but likely needing separate “register/unregister WUSB gadget” calls; (b) updates to usb_gadget to include flags “is it wireless”, “is it wired”, plus (presumably in a wrapper structure) bandgroup and PHY info; (c) presumably a wireless_ep wrapping a usb_ep, and reporting wireless-specific parameters like maxburst and maxsequence; (d) configurations that are specific to wireless links; (e) function drivers that understand wireless configs and will support wireless for (additional) function instances; (f) a function to support association setup (like CBAF), not necessarily requiring a wireless adapter; (g) composite device setup that can create one or more wireless configs, including appropriate association setup support; (h) more, TBD.

Name

config_ep_by_speed — configures the given endpoint according to gadget speed.

Synopsis

```
int config_ep_by_speed (struct usb_gadget * g, struct usb_function *
f, struct usb_ep * _ep);
```

Arguments

g pointer to the gadget

f usb function

_ep the endpoint to configure

Return

error code, 0 on success

This function chooses the right descriptors for a given endpoint according to gadget speed and saves it in the endpoint desc field. If the endpoint already has a descriptor assigned to it - overwrites it with currently corresponding descriptor. The endpoint maxpacket field is updated according to the chosen descriptor.

Note

the supplied function should hold all the descriptors for supported speeds

Name

usb_add_function — add a function to a configuration

Synopsis

```
int usb_add_function (struct usb_configuration * config, struct
usb_function * function);
```

Arguments

config the configuration

function the function being added

Context

single threaded during gadget setup

Description

After initialization, each configuration must have one or more functions added to it. Adding a function involves calling its *bind*() method to allocate resources such as interface and string identifiers and endpoints.

This function returns the value of the function's `bind`, which is zero for success else a negative errno value.

Name

usb_function_deactivate — prevent function and gadget enumeration

Synopsis

```
int usb_function_deactivate (struct usb_function * function);
```

Arguments

function the function that isn't yet ready to respond

Description

Blocks response of the gadget driver to host enumeration by preventing the data line pullup from being activated. This is normally called during *bind*() processing to change from the initial “ready to respond” state, or when a required resource becomes available.

For example, drivers that serve as a passthrough to a userspace daemon can block enumeration unless that daemon (such as an OBEX, MTP, or print server) is ready to handle host requests.

Not all systems support software control of their USB peripheral data pullups.

Returns zero on success, else negative errno.

Name

usb_function_activate — allow function and gadget enumeration

Synopsis

```
int usb_function_activate (struct usb_function * function);
```

Arguments

function function on which `usb_function_activate` was called

Description

Reverses effect of `usb_function_deactivate`. If no more functions are delaying their activation, the gadget driver will respond to host enumeration procedures.

Returns zero on success, else negative errno.

Name

usb_interface_id — allocate an unused interface ID

Synopsis

```
int usb_interface_id (struct usb_configuration * config, struct
usb_function * function);
```

Arguments

config configuration associated with the interface

function function handling the interface

Context

single threaded during gadget setup

Description

`usb_interface_id` is called from usb_function.`bind` callbacks to allocate new interface IDs. The function driver will then store that ID in interface, association, CDC union, and other descriptors. It will also handle any control requests targeted at that interface, particularly changing its altsetting via `set_alt`. There may also be class-specific or vendor-specific requests to handle.

All interface identifier should be allocated using this routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier. Note that since interface identifiers are configuration-specific, functions used in more than one configuration (or more than once in a given configuration) need multiple versions of the relevant descriptors.

Returns the interface ID which was allocated; or -ENODEV if no more interface IDs can be allocated.

Name

usb_add_config — add a configuration to a device.

Synopsis

```
int usb_add_config (struct usb_composite_dev * cdev, struct
usb_configuration * config, int (*bind) (struct usb_configuration *));
```

Arguments

cdev wraps the USB gadget

config the configuration, with bConfigurationValue assigned

bind the configuration's bind function

Context

single threaded during gadget setup

Description

One of the main tasks of a composite *bind*() routine is to add each of the configurations it supports, using this routine.

This function returns the value of the configuration's *bind*(), which is zero for success else a negative errno value. Binding configurations assigns global resources including string IDs, and per-configuration resources such as interface IDs and endpoints.

Name

usb_string_id — allocate an unused string ID

Synopsis

```
int usb_string_id (struct usb_composite_dev * cdev);
```

Arguments

cdev the device whose string descriptor IDs are being allocated

Context

single threaded during gadget setup

Description

usb_string_id() is called from `bind` callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then store that ID in the appropriate descriptors and string table.

All string identifier should be allocated using this, *usb_string_ids_tab*() or *usb_string_ids_n*() routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier.

Name

usb_string_ids_tab — allocate unused string IDs in batch

Synopsis

```
int usb_string_ids_tab (struct usb_composite_dev * cdev, struct
usb_string * str);
```

Arguments

cdev the device whose string descriptor IDs are being allocated

str an array of usb_string objects to assign numbers to

Context

single threaded during gadget setup

Description

usb_string_ids() is called from `bind` callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then copy IDs from the string table to the appropriate descriptors and string table for other languages.

All string identifier should be allocated using this, *usb_string_id*() or *usb_string_ids_n*() routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier.

Name

usb_gstrings_attach — attach gadget strings to a cdev and assign ids

Synopsis

```
struct usb_string * usb_gstrings_attach (struct usb_composite_dev *
cdev, struct usb_gadget_strings ** sp, unsigned n_strings);
```

Arguments

cdev
: the device whose string descriptor IDs are being allocated and attached.

sp
: an array of usb_gadget_strings to attach.

n_strings
: number of entries in each usb_strings array (sp[]->strings)

Description

This function will create a deep copy of usb_gadget_strings and usb_string and attach it to the cdev. The actual string (usb_string.s) will not be copied but only a referenced will be made. The struct usb_gadget_strings array may contain multiple languages and should be NULL terminated. The ->language pointer of each struct usb_gadget_strings has to contain the same amount of entries. For instance: sp[0] is en-US, sp[1] is es-ES. It is expected that the first usb_string entry of es-ES contains the translation of the first usb_string entry of en-US. Therefore both entries become the same id assign.

Name

usb_string_ids_n — allocate unused string IDs in batch

Synopsis

```
int usb_string_ids_n (struct usb_composite_dev * c, unsigned n);
```

Arguments

c the device whose string descriptor IDs are being allocated

n number of string IDs to allocate

Context

single threaded during gadget setup

Description

Returns the first requested ID. This ID and next *n*-1 IDs are now valid IDs. At least provided that *n* is non-zero because if it is, returns last requested ID which is now very useful information.

usb_string_ids_n() is called from `bind` callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then store that ID in the appropriate descriptors and string table.

All string identifier should be allocated using this, *usb_string_id*() or *usb_string_ids_n*() routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier.

Name

usb_composite_probe — register a composite driver

Synopsis

```
int usb_composite_probe (struct usb_composite_driver * driver);
```

Arguments

driver the driver to register

Context

single threaded during gadget setup

Description

This function is used to register drivers using the composite driver framework. The return value is zero, or a negative errno value. Those values normally come from the driver's *bind* method, which does all the work of setting up the driver to match the hardware.

On successful return, the gadget is ready to respond to requests from the host, unless one of its components invokes `usb_gadget_disconnect` while it was binding. That would usually be done in order to wait for some userspace participation.

Name

usb_composite_unregister — unregister a composite driver

Synopsis

```
void usb_composite_unregister (struct usb_composite_driver * driver);
```

Arguments

driver the driver to unregister

Description

This function is used to unregister drivers using the composite driver framework.

Name

usb_composite_setup_continue — Continue with the control transfer

Synopsis

```
void usb_composite_setup_continue (struct usb_composite_dev * cdev);
```

Arguments

cdev the composite device who's control transfer was kept waiting

Description

This function must be called by the USB function driver to continue with the control transfer's data/status stage in case it had requested to delay the data/status stages. A USB function's setup handler (e.g. `set_alt`) can request the composite framework to delay the setup request's data/status stages by returning USB_GADGET_DELAYED_STATUS.

Composite Device Functions

At this writing, a few of the current gadget drivers have been converted to this framework. Near-term plans include converting all of them, except for "gadgetfs".

Name

drivers/usb/gadget/function/f_acm.c — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `drivers/usb/gadget/function/f_acm.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Name

drivers/usb/gadget/function/f_ecm.c — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `drivers/usb/gadget/function/f_ecm.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Name

drivers/usb/gadget/function/f_subset.c — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `drivers/usb/gadget/function/f_subset.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Name

drivers/usb/gadget/function/f_obex.c — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `drivers/usb/gadget/function/f_obex.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Name

drivers/usb/gadget/function/f_serial.c — Document generation inconsistency

Oops

Warning

The template for this document tried to insert the structured comment from the file `drivers/usb/gadget/function/f_serial.c` at this point, but none was found. This dummy section is inserted to allow generation to continue.

Chapter 4. Peripheral Controller Drivers

The first hardware supporting this API was the NetChip 2280 controller, which supports USB 2.0 high speed and is based on PCI. This is the `net2280` driver module. The driver supports Linux kernel versions 2.4 and 2.6; contact NetChip Technologies for development boards and product information.

Other hardware working in the "gadget" framework includes: Intel's PXA 25x and IXP42x series processors (`pxa2xx_udc`), Toshiba TC86c001 "Goku-S" (`goku_udc`), Renesas SH7705/7727 (`sh_udc`), MediaQ 11xx (`mq11xx_udc`), Hynix HMS30C7202 (`h7202_udc`), National 9303/4 (`n9604_udc`), Texas Instruments OMAP (`omap_udc`), Sharp LH7A40x (`lh7a40x_udc`), and more. Most of those are full speed controllers.

At this writing, there are people at work on drivers in this framework for several other USB device controllers, with plans to make many of them be widely available.

A partial USB simulator, the `dummy_hcd` driver, is available. It can act like a net2280, a pxa25x, or an sa11x0 in terms of available endpoints and device speeds; and it simulates control, bulk, and to some extent interrupt transfers. That lets you develop some parts of a gadget driver on a normal PC, without any special hardware, and perhaps with the assistance of tools such as GDB running with User Mode Linux. At least one person has expressed interest in adapting that approach, hooking it up to a simulator for a microcontroller. Such simulators can help debug subsystems where the runtime hardware is unfriendly to software development, or is not yet available.

Support for other controllers is expected to be developed and contributed over time, as this driver framework evolves.

Chapter 5. Gadget Drivers

In addition to *Gadget Zero* (used primarily for testing and development with drivers for usb controller hardware), other gadget drivers exist.

There's an *ethernet* gadget driver, which implements one of the most useful *Communications Device Class* (CDC) models. One of the standards for cable modem interoperability even specifies the use of this ethernet model as one of two mandatory options. Gadgets using this code look to a USB host as if they're an Ethernet adapter. It provides access to a network where the gadget's CPU is one host, which could easily be bridging, routing, or firewalling access to other networks. Since some hardware can't fully implement the CDC Ethernet requirements, this driver also implements a "good parts only" subset of CDC Ethernet. (That subset doesn't advertise itself as CDC Ethernet, to avoid creating problems.)

Support for Microsoft's *RNDIS* protocol has been contributed by Pengutronix and Auerswald GmbH. This is like CDC Ethernet, but it runs on more slightly USB hardware (but less than the CDC subset). However, its main claim to fame is being able to connect directly to recent versions of Windows, using drivers that Microsoft bundles and supports, making it much simpler to network with Windows.

There is also support for user mode gadget drivers, using *gadgetfs*. This provides a *User Mode API* that presents each endpoint as a single file descriptor. I/O is done using normal *read()* and *read()* calls. Familiar tools like GDB and pthreads can be used to develop and debug user mode drivers, so that once a robust controller driver is available many applications for it won't require new kernel mode software. Linux 2.6 *Async I/O (AIO)* support is available, so that user mode software can stream data with only slightly more overhead than a kernel driver.

There's a USB Mass Storage class driver, which provides a different solution for interoperability with systems such as MS-Windows and MacOS. That *Mass Storage* driver uses a file or block device as backing store for a drive, like the `loop` driver. The USB host uses the BBB, CB, or CBI versions of the mass storage class specification, using transparent SCSI commands to access the data from the backing store.

There's a "serial line" driver, useful for TTY style operation over USB. The latest version of that driver supports CDC ACM style operation, like a USB modem, and so on most hardware it can interoperate easily with MS-Windows. One interesting use of that driver is in boot firmware (like a BIOS), which can sometimes use that model with very small systems without real serial lines.

Support for other kinds of gadget is expected to be developed and contributed over time, as this driver framework evolves.

Chapter 6. USB On-The-GO (OTG)

USB OTG support on Linux 2.6 was initially developed by Texas Instruments for OMAP [http://www.omap.com] 16xx and 17xx series processors. Other OTG systems should work in similar ways, but the hardware level details could be very different.

Systems need specialized hardware support to implement OTG, notably including a special *Mini-AB* jack and associated transceiver to support *Dual-Role* operation: they can act either as a host, using the standard Linux-USB host side driver stack, or as a peripheral, using this "gadget" framework. To do that, the system software relies on small additions to those programming interfaces, and on a new internal component (here called an "OTG Controller") affecting which driver stack connects to the OTG port. In each role, the system can re-use the existing pool of hardware-neutral drivers, layered on top of the controller driver interfaces (*usb_bus* or *usb_gadget*). Such drivers need at most minor changes, and most of the calls added to support OTG can also benefit non-OTG products.

- Gadget drivers test the *is_otg* flag, and use it to determine whether or not to include an OTG descriptor in each of their configurations.
- Gadget drivers may need changes to support the two new OTG protocols, exposed in new gadget attributes such as *b_hnp_enable* flag. HNP support should be reported through a user interface (two LEDs could suffice), and is triggered in some cases when the host suspends the peripheral. SRP support can be user-initiated just like remote wakeup, probably by pressing the same button.
- On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using `usb_suspend_device()`. That also conserves battery power, which is useful even for non-OTG configurations.
- Also on the host side, a driver must support the OTG "Targeted Peripheral List". That's just a whitelist, used to reject peripherals not supported with a given Linux OTG host. *This whitelist is product-specific; each product must modify* `otg_whitelist.h` *to match its interoperability specification.*

  Non-OTG Linux hosts, like PCs and workstations, normally have some solution for adding drivers, so that peripherals that aren't recognized can eventually be supported. That approach is unreasonable for consumer products that may never have their firmware upgraded, and where it's usually unrealistic to expect traditional PC/workstation/server kinds of support model to work. For example, it's often impractical to change device firmware once the product has been distributed, so driver bugs can't normally be fixed if they're found after shipment.

Additional changes are needed below those hardware-neutral *usb_bus* and *usb_gadget* driver interfaces; those aren't discussed here in any detail. Those affect the hardware-specific code for each USB Host or Peripheral controller, and how the HCD initializes (since OTG can be active only on a single port). They also involve what may be called an *OTG Controller Driver*, managing the OTG transceiver and the OTG state machine logic as well as much of the root hub behavior for the OTG port. The OTG controller driver needs to activate and deactivate USB controllers depending on the relevant device role. Some related changes were needed inside usbcore, so that it can identify OTG-capable devices and respond appropriately to HNP or SRP protocols.