

# **USB Gadget API for Linux**

**David Brownell**

**`dbrownell@users.sourceforge.net`**

# **USB Gadget API for Linux**

by David Brownell

20 August 2004 Edition

Copyright © 2003-2004 David Brownell

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

<b>1. Introduction.....</b>	<b>1</b>
<b>2. Structure of Gadget Drivers.....</b>	<b>3</b>
<b>3. Kernel Mode Gadget API.....</b>	<b>7</b>
3.1. Driver Life Cycle .....	8
3.2. USB 2.0 Chapter 9 Types and Constants .....	9
struct usb_ctrlrequest .....	9
3.3. Core Objects and Methods.....	10
struct usb_request.....	11
struct usb_ep .....	13
usb_ep_enable.....	14
usb_ep_disable .....	15
usb_ep_alloc_request.....	16
usb_ep_free_request .....	17
usb_ep_alloc_buffer .....	18
usb_ep_free_buffer .....	19
usb_ep_queue.....	20
usb_ep_dequeue .....	22
usb_ep_set_halt.....	23
usb_ep_clear_halt .....	24
usb_ep_fifo_status.....	25
usb_ep_fifo_flush.....	26
struct usb_gadget .....	27
usb_gadget_frame_number.....	29
usb_gadget_wakeup.....	29
usb_gadget_set_selfpowered .....	30
usb_gadget_clear_selfpowered .....	31
usb_gadget_vbus_connect .....	32
usb_gadget_vbus_draw .....	33
usb_gadget_vbus_disconnect.....	34
usb_gadget_connect.....	35
usb_gadget_disconnect .....	35
struct usb_gadget_driver .....	36
usb_gadget_register_driver .....	39
usb_gadget_unregister_driver .....	39
struct usb_string .....	40
struct usb_gadget_strings.....	41
3.4. Optional Utilities.....	42
usb_gadget_get_string .....	42
usb_descriptor_fillbuf .....	43
usb_gadget_config_buf .....	44

<b>4. Peripheral Controller Drivers.....</b>	<b>47</b>
<b>5. Gadget Drivers .....</b>	<b>49</b>
<b>6. USB On-The-GO (OTG) .....</b>	<b>51</b>

# 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

## *Chapter 1. Introduction*

other is a hardware-aware slave), the endpoint I/O 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

## Chapter 2. Structure of Gadget Drivers

- 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 `kernel-doc` 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.

## 3.1. 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.

## 3.2. 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`).

### **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.

## 3.3. 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.

### struct usb\_request

#### Name

`struct usb_request` — describes one i/o request

#### Synopsis

```
struct usb_request {
    void * buf;
    unsigned length;
    dma_addr_t dma;
    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.

`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. 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.

## struct usb\_ep

### 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;
    unsigned maxpacket:16;
};
```

## Members

`driver_data`

for use by the gadget driver. all other fields are read-only to gadget drivers.

`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

`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.

## 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.

## usb\_ep\_enable

### Name

`usb_ep_enable` — configure endpoint, making it usable

## Synopsis

```
int usb_ep_enable (struct usb_ep * ep, const struct
usb_endpoint_descriptor * desc);
```

## Arguments

*ep*

the endpoint being configured. may not be the endpoint named “ep0”. drivers discover endpoints through the `ep_list` of a `usb_gadget`.

*desc*

descriptor for desired behavior. caller guarantees this pointer remains valid until the endpoint is disabled; the data byte order is little-endian (usb-standard).

## Description

when configurations are set, or when interface settings change, the driver will enable or disable the relevant endpoints. while it is enabled, an endpoint may be used for i/o until the driver receives a `disconnect` from the host or until the endpoint is disabled.

the `ep0` implementation (which calls this routine) must ensure that the hardware capabilities of each endpoint match the descriptor provided for it. for example, an endpoint named “ep2in-bulk” would be usable for interrupt transfers as well as bulk, but it likely couldn’t be used for iso transfers or for endpoint 14. some endpoints are fully configurable, with more generic names like “ep-a”. (remember that for USB, “in” means “towards the USB master”).)

returns zero, or a negative error code.

## usb\_ep\_disable

### Name

`usb_ep_disable` — endpoint is no longer usable

### Synopsis

```
int usb_ep_disable (struct usb_ep * ep);
```

### Arguments

*ep*

the endpoint being unconfigured. may not be the endpoint named “ep0”.

### Description

no other task may be using this endpoint when this is called. any pending and uncompleted requests will complete with status indicating disconnect (-ESHUTDOWN) before this call returns. gadget drivers must call `usb_ep_enable` again before queueing requests to the endpoint.

returns zero, or a negative error code.

## usb\_ep\_alloc\_request

### Name

`usb_ep_alloc_request` — allocate a request object to use with this endpoint

## Synopsis

```
struct usb_request * usb_ep_alloc_request (struct usb_ep * ep,
int gfp_flags);
```

## Arguments

*ep*

the endpoint to be used with with the request

*gfp\_flags*

GFP\_\* flags to use

## Description

Request objects must be allocated with this call, since they normally need controller-specific setup and may even need endpoint-specific resources such as allocation of DMA descriptors. Requests may be submitted with `usb_ep_queue`, and receive a single completion callback. Free requests with `usb_ep_free_request`, when they are no longer needed.

Returns the request, or null if one could not be allocated.

## usb\_ep\_free\_request

### Name

`usb_ep_free_request` — frees a request object

## Synopsis

```
void usb_ep_free_request (struct usb_ep * ep, struct  
usb_request * req);
```

## Arguments

*ep*

the endpoint associated with the request

*req*

the request being freed

## Description

Reverses the effect of `usb_ep_alloc_request`. Caller guarantees the request is not queued, and that it will no longer be requeued (or otherwise used).

# usb\_ep\_alloc\_buffer

## Name

`usb_ep_alloc_buffer` — allocate an I/O buffer

## Synopsis

```
void * usb_ep_alloc_buffer (struct usb_ep * ep, unsigned len,  
dma_addr_t * dma, int gfp_flags);
```

## Arguments

*ep*

the endpoint associated with the buffer

*len*

length of the desired buffer

*dma*

pointer to the buffer's DMA address; must be valid

*gfp\_flags*

GFP\_\* flags to use

## Description

Returns a new buffer, or null if one could not be allocated. The buffer is suitably aligned for dma, if that endpoint uses DMA, and the caller won't have to care about dma-inconsistency or any hidden “bounce buffer” mechanism. No additional per-request DMA mapping will be required for such buffers. Free it later with `usb_ep_free_buffer`.

You don't need to use this call to allocate I/O buffers unless you want to make sure drivers don't incur costs for such “bounce buffer” copies or per-request DMA mappings.

## `usb_ep_free_buffer`

### Name

`usb_ep_free_buffer` — frees an i/o buffer

## Synopsis

```
void usb_ep_free_buffer (struct usb_ep * ep, void * buf,  
dma_addr_t dma, unsigned len);
```

## Arguments

*ep*  
the endpoint associated with the buffer

*buf*  
CPU view address of the buffer

*dma*  
the buffer's DMA address

*len*  
length of the buffer

## Description

reverses the effect of `usb_ep_alloc_buffer`. caller guarantees the buffer will no longer be accessed

## usb\_ep\_queue

### Name

`usb_ep_queue` — queues (submits) an I/O request to an endpoint.



## Synopsis

```
int usb_ep_queue (struct usb_ep * ep, struct usb_request *
req, int gfp_flags);
```

## Arguments

*ep*

the endpoint associated with the request

*req*

the request being submitted

*gfp\_flags*

GFP\_\* flags to use in case the lower level driver couldn't pre-allocate all necessary memory with the request.

## Description

This tells the device controller to perform the specified request through that endpoint (reading or writing a buffer). When the request completes, including being canceled by `usb_ep_dequeue`, the request's completion routine is called to return the request to the driver. Any endpoint (except control endpoints like `ep0`) may have more than one transfer request queued; they complete in FIFO order. Once a gadget driver submits a request, that request may not be examined or modified until it is given back to that driver through the completion callback.

Each request is turned into one or more packets. The controller driver never merges adjacent requests into the same packet. OUT transfers will sometimes use data that's already buffered in the hardware. Drivers can rely on the fact that the first byte of the request's buffer always corresponds to the first byte of some USB packet, for both IN and OUT transfers.

Bulk endpoints can queue any amount of data; the transfer is packetized automatically. The last packet will be short if the request doesn't fill it out completely. Zero length packets (ZLPs) should be avoided in portable protocols since not all usb hardware can successfully handle zero length packets. (ZLPs may be explicitly written, and may be implicitly written if the request 'zero' flag is set.) Bulk endpoints may also be used for interrupt transfers; but the reverse is not true,

and some endpoints won't support every interrupt transfer. (Such as 768 byte packets.)

Interrupt-only endpoints are less functional than bulk endpoints, for example by not supporting queueing or not handling buffers that are larger than the endpoint's maxpacket size. They may also treat data toggle differently.

Control endpoints ... after getting a `setup` callback, the driver queues one response (even if it would be zero length). That enables the status ack, after transferring data as specified in the response. Setup functions may return negative error codes to generate protocol stalls. (Note that some USB device controllers disallow protocol stall responses in some cases.) When control responses are deferred (the response is written after the setup callback returns), then `usb_ep_set_halt` may be used on `ep0` to trigger protocol stalls.

For periodic endpoints, like interrupt or isochronous ones, the usb host arranges to poll once per interval, and the gadget driver usually will have queued some data to transfer at that time.

Returns zero, or a negative error code. Endpoints that are not enabled report errors; errors will also be reported when the usb peripheral is disconnected.

## usb\_ep\_dequeue

### Name

`usb_ep_dequeue` — dequeues (cancels, unlinks) an I/O request from an endpoint

### Synopsis

```
int usb_ep_dequeue (struct usb_ep * ep, struct usb_request * req);
```

## Arguments

*ep*

the endpoint associated with the request

*req*

the request being canceled

## Description

if the request is still active on the endpoint, it is dequeued and its completion routine is called (with status `-ECONNRESET`); else a negative error code is returned.

note that some hardware can't clear out write fifos (to unlink the request at the head of the queue) except as part of disconnecting from usb. such restrictions prevent drivers from supporting configuration changes, even to configuration zero (a “chapter 9” requirement).

# usb\_ep\_set\_halt

## Name

`usb_ep_set_halt` — sets the endpoint halt feature.

## Synopsis

```
int usb_ep_set_halt (struct usb_ep * ep);
```

## Arguments

*ep*

the non-isochronous endpoint being stalled

## Description

Use this to stall an endpoint, perhaps as an error report. Except for control endpoints, the endpoint stays halted (will not stream any data) until the host clears this feature; drivers may need to empty the endpoint's request queue first, to make sure no inappropriate transfers happen.

Note that while an endpoint `CLEAR_FEATURE` will be invisible to the gadget driver, a `SET_INTERFACE` will not be. To reset endpoints for the current altsetting, see `usb_ep_clear_halt`. When switching altsettings, it's simplest to use `usb_ep_enable` or `usb_ep_disable` for the endpoints.

Returns zero, or a negative error code. On success, this call sets underlying hardware state that blocks data transfers. Attempts to halt IN endpoints will fail (returning `-EAGAIN`) if any transfer requests are still queued, or if the controller hardware (usually a FIFO) still holds bytes that the host hasn't collected.

## `usb_ep_clear_halt`

### Name

`usb_ep_clear_halt` — clears endpoint halt, and resets toggle

### Synopsis

```
int usb_ep_clear_halt (struct usb_ep * ep);
```

## Arguments

*ep*

the bulk or interrupt endpoint being reset

## Description

Use this when responding to the standard usb “set interface” request, for endpoints that aren’t reconfigured, after clearing any other state in the endpoint’s i/o queue.

Returns zero, or a negative error code. On success, this call clears the underlying hardware state reflecting endpoint halt and data toggle. Note that some hardware can’t support this request (like pxa2xx\_udc), and accordingly can’t correctly implement interface altsettings.

## usb\_ep\_fifo\_status

### Name

`usb_ep_fifo_status` — returns number of bytes in fifo, or error

## Synopsis

```
int usb_ep_fifo_status (struct usb_ep * ep);
```

## Arguments

*ep*

the endpoint whose fifo status is being checked.

## Description

FIFO endpoints may have “unclaimed data” in them in certain cases, such as after aborted transfers. Hosts may not have collected all the IN data written by the gadget driver (and reported by a request completion). The gadget driver may not have collected all the data written OUT to it by the host. Drivers that need precise handling for fault reporting or recovery may need to use this call.

This returns the number of such bytes in the fifo, or a negative errno if the endpoint doesn’t use a FIFO or doesn’t support such precise handling.

## usb\_ep\_fifo\_flush

### Name

`usb_ep_fifo_flush` — flushes contents of a fifo

### Synopsis

```
void usb_ep_fifo_flush (struct usb_ep * ep);
```

### Arguments

*ep*

the endpoint whose fifo is being flushed.

### Description

This call may be used to flush the “unclaimed data” that may exist in an endpoint fifo after abnormal transaction terminations. The call must never be used except when endpoint is not being used for any protocol translation.

# struct usb\_gadget

## Name

struct usb\_gadget — represents a usb slave device

## Synopsis

```
struct usb_gadget {
    const struct usb_gadget_ops * ops;
    struct usb_ep * ep0;
    struct list_head ep_list;
    enum usb_device_speed speed;
    unsigned is_dualspeed: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;
    const char * name;
    struct device dev;
};
```

## Members

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.

`is_dualspeed`

True if the controller supports both high and full speed operation. If it does, the gadget driver must also support both.

`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.

`name`

Identifies the controller hardware type. Used in diagnostics and sometimes configuration.

`dev`

Driver model state for this abstract device.

## 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).

## usb\_gadget\_frame\_number

### Name

`usb_gadget_frame_number` — returns the current frame number

### Synopsis

```
int usb_gadget_frame_number (struct usb_gadget * gadget);
```

### Arguments

*gadget*

controller that reports the frame number

### Description

Returns the usb frame number, normally eleven bits from a SOF packet, or negative `errno` if this device doesn't support this capability.

## usb\_gadget\_wakeup

### Name

`usb_gadget_wakeup` — tries to wake up the host connected to this gadget

### Synopsis

```
int usb_gadget_wakeup (struct usb_gadget * gadget);
```

### Arguments

*gadget*

controller used to wake up the host

### Description

Returns zero on success, else negative error code if the hardware doesn't support such attempts, or its support has not been enabled by the usb host. Drivers must return device descriptors that report their ability to support this, or hosts won't enable it.

This may also try to use SRP to wake the host and start enumeration, even if OTG isn't otherwise in use. OTG devices may also start remote wakeup even when hosts don't explicitly enable it.

## usb\_gadget\_set\_selfpowered

### Name

`usb_gadget_set_selfpowered` — sets the device selfpowered feature.

## Synopsis

```
int usb_gadget_set_selfpowered (struct usb_gadget * gadget);
```

## Arguments

*gadget*

the device being declared as self-powered

## Description

this affects the device status reported by the hardware driver to reflect that it now has a local power supply.

returns zero on success, else negative errno.

# usb\_gadget\_clear\_selfpowered

## Name

`usb_gadget_clear_selfpowered` — clear the device selfpowered feature.

## Synopsis

```
int usb_gadget_clear_selfpowered (struct usb_gadget * gadget);
```

## Arguments

*gadget*

the device being declared as bus-powered

## Description

this affects the device status reported by the hardware driver. some hardware may not support bus-powered operation, in which case this feature's value can never change.

returns zero on success, else negative errno.

# usb\_gadget\_vbus\_connect

## Name

`usb_gadget_vbus_connect` — Notify controller that VBUS is powered

## Synopsis

```
int usb_gadget_vbus_connect (struct usb_gadget * gadget);
```

## Arguments

*gadget*

The device which now has VBUS power.

## Description

This call is used by a driver for an external transceiver (or GPIO) that detects a VBUS power session starting. Common responses include resuming the controller, activating the D+ (or D-) pullup to let the host detect that a USB device is attached, and starting to draw power (8mA or possibly more, especially after SET\_CONFIGURATION).

Returns zero on success, else negative errno.

## usb\_gadget\_vbus\_draw

### Name

`usb_gadget_vbus_draw` — constrain controller's VBUS power usage

### Synopsis

```
int usb_gadget_vbus_draw (struct usb_gadget * gadget, unsigned
mA);
```

### Arguments

*gadget*

The device whose VBUS usage is being described

*mA*

How much current to draw, in milliAmperes. This should be twice the value listed in the configuration descriptor bMaxPower field.

## Description

This call is used by gadget drivers during SET\_CONFIGURATION calls, reporting how much power the device may consume. For example, this could affect how quickly batteries are recharged.

Returns zero on success, else negative errno.

# usb\_gadget\_vbus\_disconnect

## Name

`usb_gadget_vbus_disconnect` — notify controller about VBUS session end

## Synopsis

```
int usb_gadget_vbus_disconnect (struct usb_gadget * gadget);
```

## Arguments

*gadget*

the device whose VBUS supply is being described

## Description

This call is used by a driver for an external transceiver (or GPIO) that detects a VBUS power session ending. Common responses include reversing everything done in `usb_gadget_vbus_connect`.

Returns zero on success, else negative errno.

# usb\_gadget\_connect

## Name

`usb_gadget_connect` — software-controlled connect to USB host

## Synopsis

```
int usb_gadget_connect (struct usb_gadget * gadget);
```

## Arguments

*gadget*

the peripheral being connected

## Description

Enables the D+ (or potentially D-) pullup. The host will start enumerating this gadget when the pullup is active and a VBUS session is active (the link is powered). This pullup is always enabled unless `usb_gadget_disconnect` has been used to disable it.

Returns zero on success, else negative errno.

# usb\_gadget\_disconnect

## Name

`usb_gadget_disconnect` — software-controlled disconnect from USB host

## Synopsis

```
int usb_gadget_disconnect (struct usb_gadget * gadget);
```

## Arguments

*gadget*

the peripheral being disconnected

## Description

Disables the D+ (or potentially D-) pullup, which the host may see as a disconnect (when a VBUS session is active). Not all systems support software pullup controls.

This routine may be used during the gadget driver `bind` call to prevent the peripheral from ever being visible to the USB host, unless later `usb_gadget_connect` is called. For example, user mode components may need to be activated before the system can talk to hosts.

Returns zero on success, else negative `errno`.

## struct usb\_gadget\_driver

### Name

`struct usb_gadget_driver` — driver for usb 'slave' devices

## Synopsis

```
struct usb_gadget_driver {  
    char * function;  
    enum usb_device_speed speed;  
    int (* bind) (struct usb_gadget *);  
    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 *);
// FIXME support safe rmmodstruct device_driver driver;
};
```

## Members

### function

String describing the gadget's function

### speed

Highest speed the driver handles.

### bind

Invoked when the driver is bound to a gadget, usually after registering the driver. At that point, ep0 is fully initialized, and ep\_list holds the currently-available endpoints. Called in a context that permits sleeping.

### 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 cpu 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.

driver

Driver model state for this driver.

## Description

Devices are disabled till a gadget driver successfully binds, 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.

## usb\_gadget\_register\_driver

### Name

`usb_gadget_register_driver` — register a gadget driver

### Synopsis

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

### Arguments

*driver*

the driver being registered

### Description

Call this in your gadget driver's module initialization function, to tell the underlying usb controller driver about your driver. The driver's `bind` function will be called to bind it to a gadget. This function must be called in a context that can sleep.

# usb\_gadget\_unregister\_driver

## 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

## 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. This function must be called in a context that can sleep.

# struct usb\_string

## 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.

# struct usb\_gadget\_strings

## 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.

## 3.4. 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.

## usb\_gadget\_get\_string

### 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

## 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 strings 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).

## usb\_descriptor\_fillbuf

### 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.

# usb\_gadget\_config\_buf

## Name

`usb_gadget_config_buf` — builds 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.



# 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 *write()* 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 *File-backed 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.