USB Gadget API for Linux

Author: David Brownell
Date: 20 August 2004

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.

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.

Kernel Mode Gadget API

Gadget drivers declare themselves through a struct :c:type:`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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst`, line 177); *backlink*

Unknown interpreted text role "c:type".

Note

Other than the "Chapter 9" data types, most of the significant data types and functions are described here.

However, 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 and upper kernel versions include 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 :c:type:`usb_gadget`, which activates the data line pullup sometime after detecting VBUS.

   System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst`, line 250); ***backlink***

   Unknown interpreted text role "c:type".

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 :ref:`linux/usb/ch9.h <usb_chapter9>` header file, which is standard in Linux 2.6+ kernels. These are the same types and constants used by host side drivers (and usbcore).

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api]

[usb]gadget.rst, line 306); *backlink*

Unknown interpreted text role "ref".

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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst, line 317)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/usb/gadget.h
   :internal:
```

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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst, line 327)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/gadget/usbstring.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst, line 330)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/gadget/config.c
   :export:
```

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 :c:type:`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".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst, line 342); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst, line 349)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/usb/composite.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api]

[usb]gadget.rst, line 352)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/usb/gadget/composite.c
   :export:
```

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`.

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.

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.

USB On-The-GO (OTG)

USB OTG support on Linux 2.6 was initially developed by Texas Instruments for OMAP 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 (:c:type:`usb_bus` or :c:type:`usb_gadget`). Such drivers need at most minor changes, and most of the calls added to support OTG can also benefit non-OTG products.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst`, **line 455**); ***backlink***

Unknown interpreted text role "c:type".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst`, **line 455**); ***backlink***

Unknown interpreted text role "c:type".

- 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 :c:type:`usb_bus` and :c:type:`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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst`, **line 500**); ***backlink***

Unknown interpreted text role "c:type".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]gadget.rst`, **line 500**); ***backlink***

Unknown interpreted text role "c:type".