Linux USB - Past, Present and Future

Brad Hards

Abstract

The major focus for many Linux developers has been in those parts of the kernel that support server type applications. However a rapidly expanding area of Linux development has been in the desktop applications area. A key aspect of this has been in the use of USB peripherals - especially keyboards, mice and mass storage devices. The hot-plug nature of USB devices has caused some of the underlying assumptions about Unix type devices to be re-assessed, and new or modified approaches to be adopted

This paper presents a rambling discourse on what USB is, how Linux came to support it, what Linux USB can currently do, and where the Linux USB project might go.

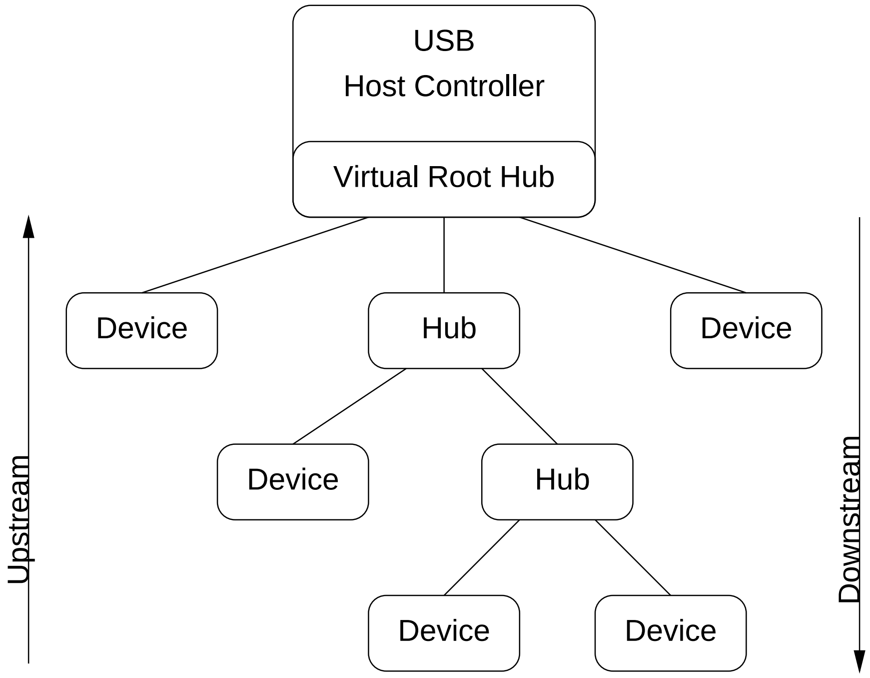
About USB

USB is an acronymn, standing for Universal Serial Bus. The name indicates the intended application - a single bus design, using serial transfers (one bit after another), with universal application. Like all things dreamed up by marketeers, the truth is a little different.

USB is basically a desktop bus, intended for connecting peripheral devices to a single computer. It optimises ease of use over performance, and cost over flexibility.

All transfers over USB are initiated by the host (almost always the single computer). Hubs are used to connect the host to the peripheral devices. There is a virtual hub built into the host device. This can be seen in the example (from a UHCI root hub) below.

USB devices end up being attached in a tree structure, where branches on the tree are created by hubs, and the nodes on the end of the tree are the devices. This is shown in the figure below.



Universal Serial Bus topology, showing the tree structure

There are four types of transfers:

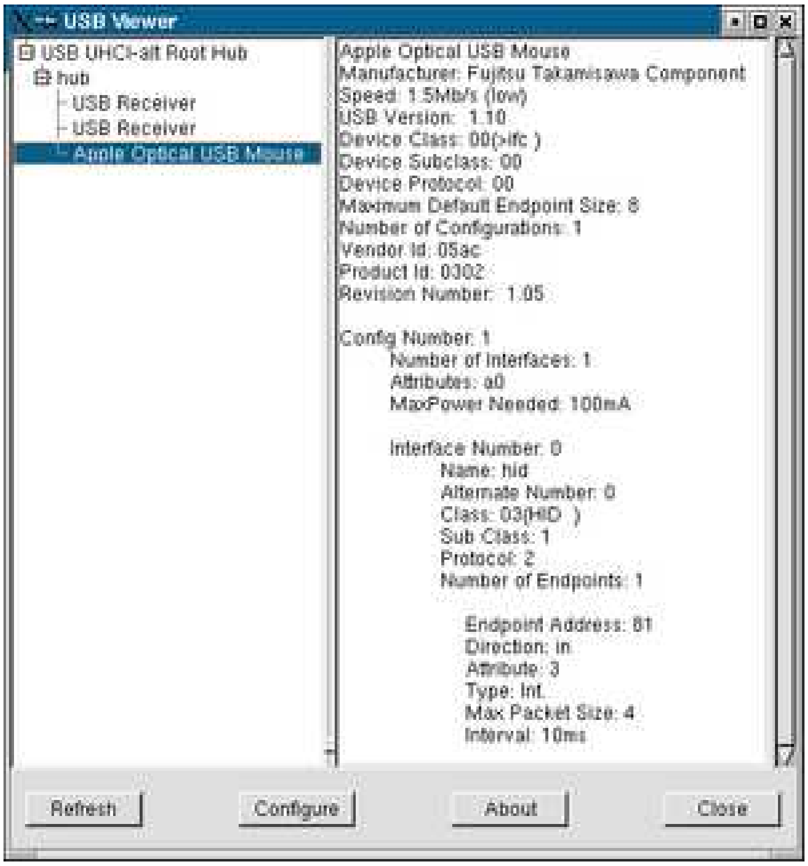
- Control
- Interrupt (polled)
- Bulk (not on low speed devices)
- Isochronous (not on low speed devices)

Devices can exchange information at a range of speeds (1.5Mbps, 12Mpbs and 480Mbps are the raw speeds, never reached in real applications), known as low-speed, full-speed and high-speed. The high-speed variation was introduced in version 2.0 of the USB specification, but a device does not have to work at 480Mbps to meet USB 2.0 requirements.

Device example 1 - USB mouse.

A mouse is probably the most common of USB devices, and mice are very simple, so they make a reasonable introduction. Ignoring the physical implementation of rollers, switches and so on, we basically have a source of data. But remember that the bus is universal. So the obvious question is "how does the host know it is a mouse, and not a keyboard, a joystick, a zip drive, or a pair of speakers?"; The answer is that USB devices are *self-describing*. There is a standard way for a device to notify the upstream hub that it is attached (by pulling down one of the two data lines to ground), and there is a standard way for the host to get description information from a device (using control transfers to the default endpoint).

The descriptors for a USB device (in this case, a mouse) are shown in the figure below.

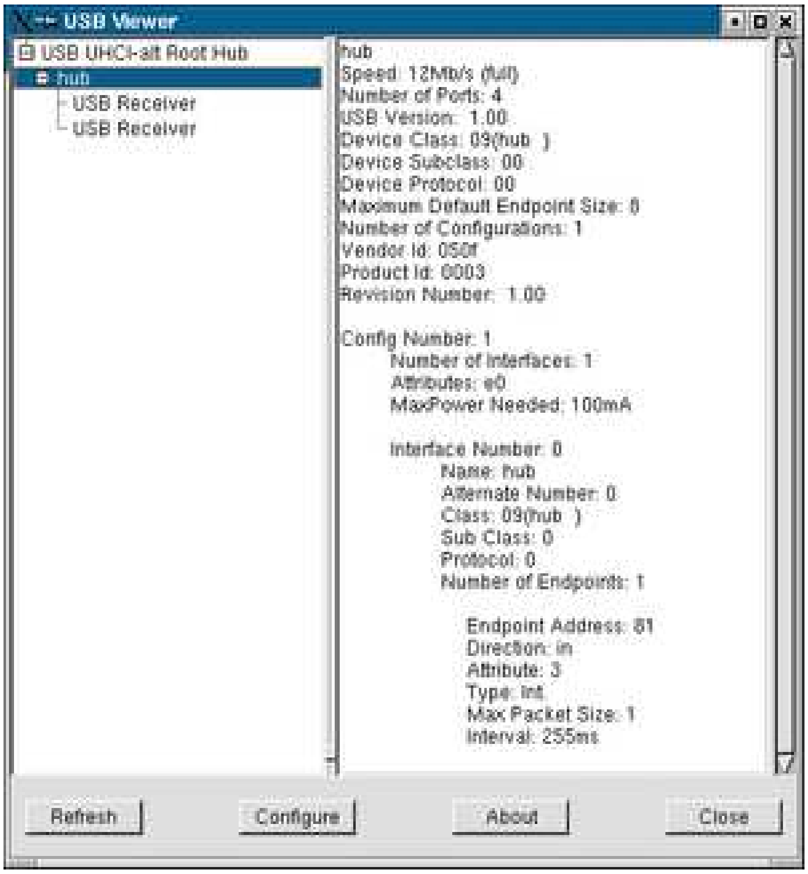


Universal Serial Bus mouse

A USB mouse should comply with the Human Interface Device (HID) class specification. Class specifications are drawn up by the USB Implementers Forum (in reality, by representatives from major peripheral manufacturers, and industry heavyweights like Intel and Microsoft), and show how a device should work. The HID specification provides a way for the device to identify itself as having a certain number of buttons, a certain number of axes, and so on. It also specifies the way in which reports are returned over the interrupt endpoint. So when a button is clicked on the mouse, that information gets sent to the host when the interrupt endpoint is next polled. They way that the kernel gets those bits to the X server (or other user space application) will be discussed later.

Device example 2 - USB hub

USB hubs are so central to the USB architecture that the class specification for hubs is actually included in the main USB specification (Chapter 11). Changes to the USB specification (from 1.0 to 1.1 and then to 2.0) are basically trivial, except for hub functionality. USB hubs basically provide physical connection fan-out. However because the host needs to know when things have changed on downstream ports, the hub has to indicate the change, again using an interrupt endpoint.

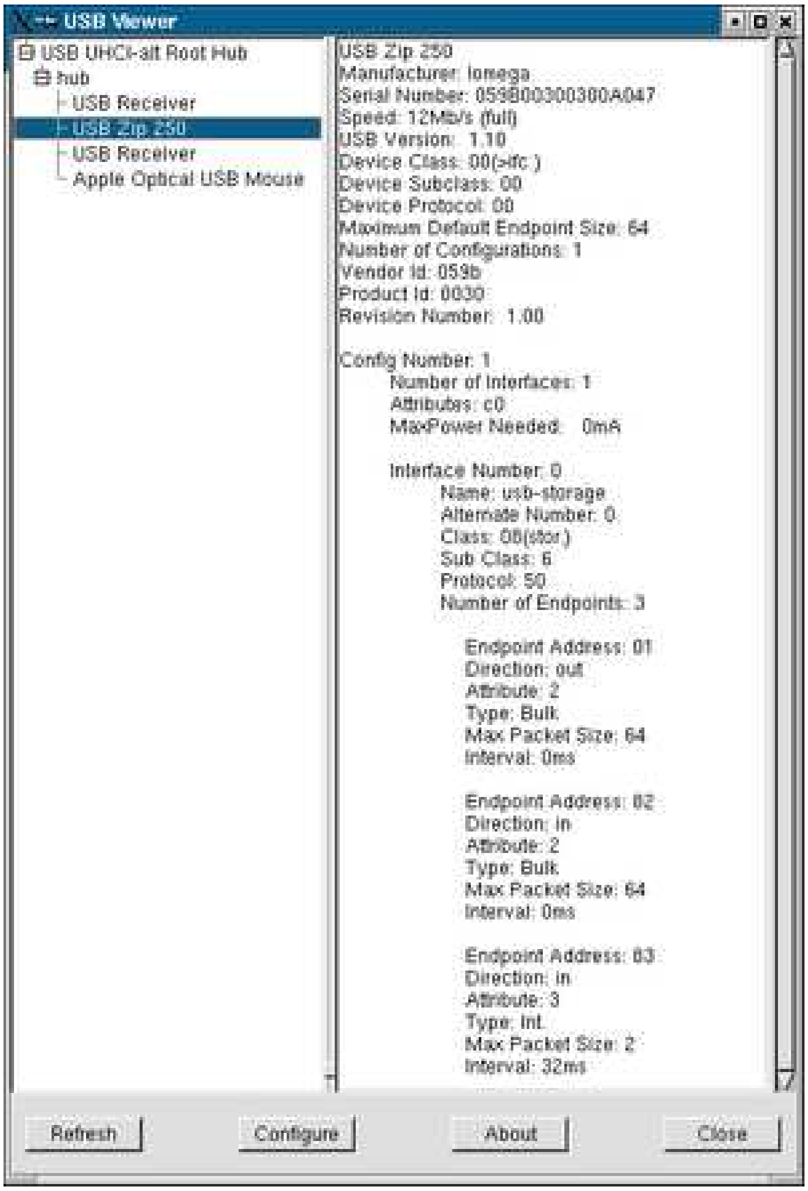


Universal Serial Bus hub

Use of interrupt endpoints to indicate status changes is pretty standard.

Device example 3 - Zip 250

A Zip drive conforms to the Mass Storage Device Class. The Mass Storage class specification has a fair number of variations, but the Zip drive uses the Control-Bulk-Interrupt (CBI) transport protocol. The main data flow to and from the disk is over a pair of bulk endpoints (one in each direction), while the interrupt endpoint reports status.

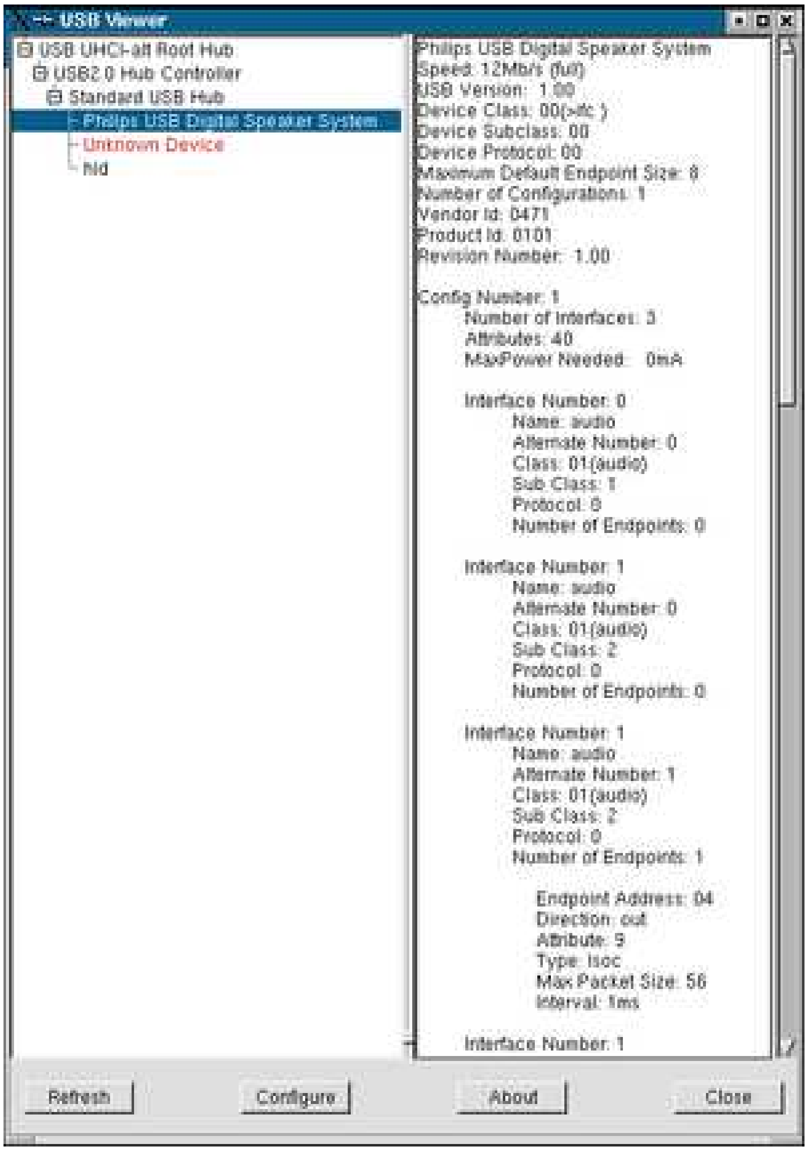


Universal Serial Bus removable drive

Despite the number of variations available in the (freely available) specification, a number of manufacturers managed to create vendor specific extensions (or limitations, such as crashing the device microcontroller when you try to probe additional LUNs), or build broken descriptors (like the Casio QV cameras that claim to use CBI, but actually use the Control-Bulk protocol). These are relatively easy to fix. Some others, such as devices that return broken checksums, are doomed never to be fully supported.

Device example 4 - USB Speakers

USB Speakers basically include sound-card and speaker functionality. The data flows out on isochronous endpoints, to ensure that the data arrives on time. Speakers can typically vary sampling rates to tune for the available bandwidth (with associated variation in quality).



Universal Serial Bus Speakers

Linux USB History

The major focus for many Linux developers has been in those parts of the kernel that support server type applications. However a rapidly expanding area of Linux development has been in the desktop applications area. A key aspect of this has been in the use of USB peripherals - especially keyboards, mice and mass storage devices. The hot-plug nature of USB devices has caused some of the underlying assumptions about Unix type devices to be re-assessed, and new or modified approaches to be adopted.

Linux USB development was originally done as part of the UUSBD (Universal USB Daemon) by Inaky Perez Gonzalez. Inaky had a master plan for support, and had a lot of infrastructure done. However he only had OHCI support and mouse support working (to a useable state) when Linus came in:

> Subject: [linux-usb] Alternate USB support - test code
>
> From: Linus Torvalds <torvalds@transmeta.com>
>
> Date: 1999-03-30 19:29:12
>
> I've been using some of my copious spare time (hah!) to explore USB the last few weeks, and I have a driver that is set up fairly differently from the standard Linux-USB driver, but that shows some promise. It only works with UHCI controllers right now, although that is simply because I didn't want to bother with a USB controller that I don't have access to anyway.
>
> If somebody wants to look at a simple debug-only USB mouse driver, feel free to look at ftp.kernel.org/pub/linux/kernel/testing. It's designed to be more direct-to-the-hardware than uusbd, because I found the uusbd code hard to follow. So it's kind of simplistic, but simple is good.
>
> And it's called 0.01 for a very good reason: don't expect very much from it. I'm mainly sending out this email to just get some comments.
>
> Linus

Inaky's code was pretty hard to understand, and a lot of developers switched to Linus' very simple setup. People added a keyboard driver, hub support, and a few other drivers, before a major rewrite was done:

> Subject: [linux-usb] UHCI/usb-core rewrite project
>
> From: Georg Acher <usb@atbode95.informatik.tu-muenchen.de>
>
> Date: 1999-11-08 18:13:27
>
> UHCI/usb-core rewrite project
>
> Motivation
>
> Both a group at TU Munich (Georg Acher, Deti Fliegl) and myself have done USB hardware designs with the EZUSB microcontroller. Both their (a Digital Audio Broadcast receiver) and my project heavily use isochronous transfers, as does USB audio, a pet project of myself.
>
> However, we soon noticed that the UHCI driver is very unstable when doing iso transfers. So we wanted to fix this. Reading through the source code convinced us that large parts of the driver

needed to be rewritten. Two weeks ago I drove to Munich and we first discussed implementation issues and then started coding.

Interface to client drivers

The current interface that the HCDs expose to client drivers is very heterogenous. This makes it rather difficult to use. So we decided to design a new interface. The new interface is completely asynchronous. Wrapper provide the old synchronous control and bulk interface. The interface is centered around an asynchronous request structure and two routines.

(See http://marc.theaimsgroup.com/?l=linux-usb&m=96207400316872&w=2 for the rest of the message)

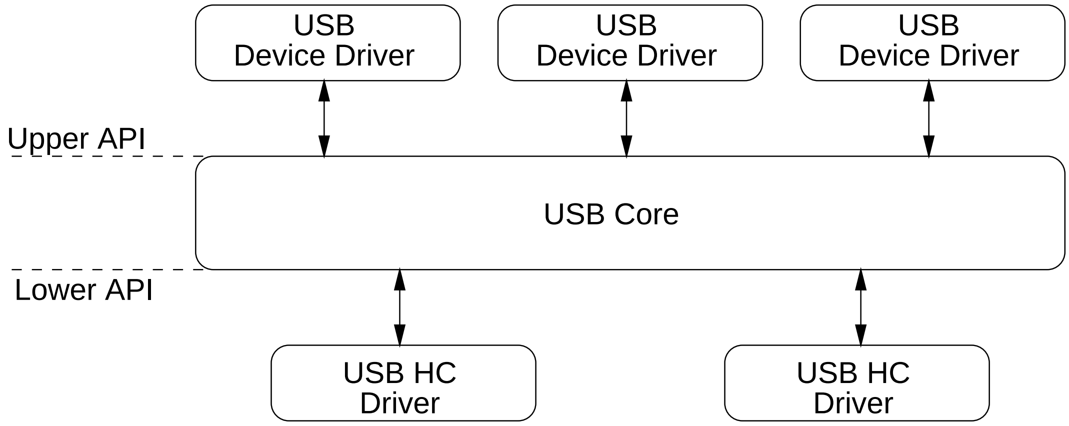
Since then, a lot of other changes have been implemented, include first use of the new input layer (see later) and changes to reflect changes in kernel infrastructure (for example, networking and PCI changes).

Where we are now

Introduction

This section will run through what Linux USB currently (early August 2002) can do, in either 2.4 or 2.5 kernels. There is no USB support in 2.0 kernels, and 2.2 series kernels typically lag well behind the 2.4 kernels. In addition, some drivers don't exist in 2.2, or are not supported, as discussed below.

The interaction between components can be represented in different ways, although the diagram below is a useful simplification.



Universal Serial Bus Driver interaction

Host Controllers

The host controller is the bit of your computer that provides the connection to the USB “Type A” sockets typically mounted on the back (sometimes the front, on the side of an iMac and some laptops) of the case. There are three main host controller designs - Universal Host Controller Interface (UHCI, used by Intel and Via), Open Host Controller Interface (OHCI, used by Apple, Compaq and most other chipset manufacturers) and Enhanced Host Controller Interface (EHCI, used on high speed plug-in cards). Host controllers driver basically provide the link between the bottom of the common USB core code, and the underlying transport (typically PCI).

UHCI currently has two drivers (in 2.4), for historical reasons relating the URB rewrite project. UHCI is a pretty minimal host controller design, and the drivers seem to have timing problems. Some drivers and devices have been reported to work with one of the host controller drivers, and not work with the other. One of the UHCI drivers was dropped as part of the host controller redesign in 2.5.

OHCI is generally better thought out than UHCI, and the OHCI driver tends to provide better support for things like queuing of requests, and to be generally more stable. OHCI is also significantly better on big-endian machines than either of the UHCI drivers.

EHCI is only compatible with high speed devices, and performs a hand-off to a full speed / low speed controller (almost always OHCI, although it could be UHCI, or some other design) if the device is not high speed compatible.

Human Interface Device

Human Interface Device is the class specification for a whole range of different devices, ranging from USB keyboards and mice, joysticks, tablets. All of these are supported (in various ways) by the HID driver and the Input sub-system. There are also HID devices that are not strictly for human interface, but have similar communication needs - for example, power supplies (especially UPS) and USB monitor controls

Most USB users are pretty familiar with keyboards and mice, and yet they involve pretty sophisticated drivers and higher level interfaces.

The historical USB keyboard and mouse drivers are called “usbkbd” and “usbmouse” respectively. These drivers support the simplified "Boot Protocol" version of the HID specification. To get proper HID support, you need the “hid” driver, which uses the Input subsystem. Using Boot Protocol drivers is inherently broken, and should *never* be done.

The Input subsystem is not part of the USB project. Instead, it is a wide ranging project to refactor the various ways in which Linux handles user inputs. Instead of PS2 keyboards, serial mice, ADB joysticks and USB tablets all being handled by quite distinct routines, Input uses a standardised event based system, and a set of standard event handlers that generate both the existing interfaces (such as a char device called `/dev/input/mice`, that is basically the amalgam of all of the different mice, touchpads and trackballs attached, in standard ImPS/2 mouse format) and also a new event interface that can be used to provide support for more capable devices that cannot be fully described by the limited existing interfaces (such as a X driver that handles mice with more than five logical buttons).

So while the hid driver can generate suitable events for the Input subsystem, other drivers (such as the Apple ADB driver) can also generate suitable events. Linux USB actually uses this for some Wacom tablets that do not conform to the HID specification (despite descriptors claiming that they do), with the “wacom” driver. In the current 2.5 kernels, the Input subsystem is almost fully integrated.

For devices such as Uninterruptable Power Supplies and monitors, that make no sense to the Input subsystem, there is the HID device interface, a char interface on `/dev/usb/hiddevX`. The hiddev API uses a read() interface, and a set of ioctl() calls. The ioctl()s are used for setting up the read() interface, and also for setting parameters in the HID device. There are also ioctl() calls for interrogating specific fields.

Mass Storage

USB Mass storage support is very strong, being based on the Mass Storage Device class, and well supported by strong maintenance team, including some good manufacturer support. Most devices are close enough to the Mass Storage class to just work (or can be made to work with a little effort), although certain devices (eg HP8200e CD-RW, ISD-200 based hard drives, and a few SmartMedia and Compact Flash readers) need some additional driver help. A very few devices are so badly broken that they are unsupported.

USB Mass Storage basically fits into the bottom of the SCSI stack, and relies on the higher level SCSI drivers (“sd”, “scd”, “st” and “sg”) to interface to the world. This means that a USB floppy drive (for example), when connected to a machine with no other SCSI controllers, will appear as /dev/sda. Similarly, a CD in an external drive will typically appear as /dev/scd0.

In addition to the main USB storage driver, there is also reverse engineered support for a couple of external USB hard drive designs that do not comply with the USB mass storage class specification, and do not have publically available specifications. These are not included in the kernel, and given the likelihood of things going badly wrong with the data on the drive, including them in the kernel in the near future seems unlikely.

Printers

USB printers basically use two bulk pipes to send data and control information (per the IEEE1284 specification), and the “printer” driver is among the simplest of the USB drivers to understand. The interface to user-space is a char device (/dev/usb/lp).

Some USB printers require special "initialisation" strings to make them listen on the USB interface. This is normally pretty easy to handle as part of the printing system.

Networking and Data Communications

USB to Ethernet dongles, and USB to USB networking cables, basically fit between the USB core code, and the normal networking stack.

There are four basic USB to Ethernet dongle designs, each based on a different chipset. Those designs are packaged by dozens of different manufacturers, under various names and model numbers. However it is usually fairly easy to sort out which chipset your device is using, and that tells you which driver you need.

If the device claims 10/100Mbps support, it is almost certainly based on the ADMtek Pegasus chipset. This was the first of the USB networking devices to be supported ("pegasus" driver), and has had a lot of changes incorporated. Very new device designs supporting 10/100Mbps may use a Realtek design, and require the "rt8150" driver.

The most common (aka cheapest) of the USB networking chips is the KLSI KL5KUSB101 10Mbps device. Almost all devices not claiming 10Mbps support use this chip, which is supported by the "kaweth" driver. The device is cheap to make because it has no permanent storage, and needs firmware to be downloaded by the driver before the device will work. This is currently handled by the kernel driver, and although this was going to be moved to userspace, there might be some issues with userspace needing memory to load the image file after a suspend/resume cycle, and having the swap device on the other end of the networking link.

The least common of the chips is the CATC Netmate design, which is only used by smartBridges and Belkin. However the chip design is pretty good and performance is pretty good, especially with an OHCI controller. The driver is called "catc".

With the removal of serial and parallel ports, the old standby file transfer mechanisms using null modem cables is becoming more difficult. However Linux now supports a wide range of USB to USB networking cables. These cables cannot be simple wiring (as was possible for serial and parallel) for a whole host of electrical and protocol reasons. Instead, they can be considered two simple USB devices, each on a separate bus, joined back-to-back. Each host then just talks to the device on its bus, and the devices exchange data. There are several manufacturers, and Linux support for all of them is included in a single driver called "usbnet". This driver also provides a simple way to connect devices running Linux (eg an iPaq or an embedded application) to a workstation, and a matching device side driver has been developed for the Strong ARM SA1100 family.

The most recent of the USB networking drivers is the only one that is described by a class specification. The Communication Device Class has a range of different "models". Linux currently supports two of the models - Abstract Control Model (used for convential POTS and ISDN modems, see below) and Ethernet model (mainly used for Cable modems). The Ethernet model basically provides a standard way of communicating anything a network interface could need (such as setting a multicast address list, getting MAC address, broadcasting and so on), but is not easy to implement. The Sharp Zaurus also provides device side support for this interface (albeit without some of the usual stuff like an interrupt or bulk endpoint to provide notifications), and both the Zaurus and a range of USB cable modems (seemly all based on two designs) are supported by the “CDCEther” driver (“cdc-ether” in 2.5). Future USB to Ethernet dongles would do well to support this model, however there is but a slim chance.

Multimedia

Multimedia means different things to different people, but this section deals with USB audio, USB webcams, USB cameras, and USB radios.

The major USB audio application is USB speakers. Conceptually, USB speakers are sound card and speakers integrated together. The audio driver interfaces up to the normal Linux sound card driver stack (CONFIG_SOUND) and works through the normal `/dev/mixer` and `/dev/dsp` interfaces, just like any other sound card. USB speakers often have push buttons on the front. Sometimes these do internal things to the speakers, but sometimes the buttons are actually a HID interface (and are consequentially handled by the HID drivers). In this case, volume controls on USB speakers have the same effect (and are handled in an identical way) to the volume controls on a USB multimedia keyboard. Note that, in a similar way to soundcards also providing line inputs, some USB audio devices can also do audio input. Both audio input and output are handled by the Linux USB “audio” driver, per the USB Audio Class specification.

USB webcams come in many different forms, and all use vendor specific commands. There is support for a wide range of webcams, which use the Video4Linux API to various degrees of success. OV511 based cameras (“ov511” driver) and Philips cameras (“pwc” driver, full functionality requires a binary only module not provided in the kernel distribution) seem the best bets, although a lot of reverse engineering work has been done with the USB Snoopy tool, which dumps transfers under Windows for subsequent analysis, and most webcams have either working support, or someone actively investigating support.

Some of the higher end cameras (say >1M pixels) have USB ports that can be used to obtain the images from the camera without removing the storage media. Most cameras have taken the sensible option of just presenting the storage media as Mass Storage class, which allows you to mount the filesystem directly, and apply the full power of Linux file utilties. For devices without such support (and for those who feel that a GUI is useful for handling images), the user-space GPhoto (and GPhoto2) package is definately worth investigation.

There is one USB radio known - the D-Link DSB-R100. It is really only USB tuning (the audio comes out as analog, ready to feed into speakers). Linux provides an interface between the device and the Video4Linux API using the “dsbr100” driver

Modems

As mentioned previously, Linux supports two of the many Communication Device Class models. Both POTS and ISDN modems can use the Abstract Control Model in the “acm” driver (“cdc-acm” in 2.5). This basically provides a character device interface to the modem, generally using the Hayes (AT) command set.

However it should be noted that many manufacturers have chosen not to use the Abstract Control Model, and instead have build a USB soft-modem. This is especially common in POTS modems - ISDN seems to be more standardised. It is generally not possible to guess whether a particular modem is going to work with Linux without looking at the descriptors. However a modem that says it will not work with MacOS is normally only good for paperweight use with Linux as well.

Legacy interfaces (serial / parallel)

USB is gradually taking over the existing uses of serial and parallel ports, and some manufacturers (eg Apple and Compaq) have stopped providing these ports on at least some of their products. However there are still a large number of serial and parallel port devices in use, and redesign to use the latest features of USB is often uneconomical. These legacy devices can be used with USB-only computers (or computers with too few ports) with USB to serial or USB to parallel adapters. These adapters can be either external (as a "magic cable"); or mounted internally so that the device looks like it is USB only, but the existing hardware design and user-space programs can be used.

USB to parallel adapters can either present a printer (`/dev/lp`) type interface, or a parallel port interface (`/dev/parport`), depending on cable capabilities and the mode of the device. Linux supports most cables as printers (see above), however the Lucent USS720 chip is supported in parallel port mode by the “uss720” driver, which interfaces up to the standard parallel port code.

A very wide range of USB to serial adapters are now supported in Linux, ranging from small PDA adapters to high end, quad-port serial monstronities, with a range of different drivers. These drivers present a normal serial type interface (like `/dev/tty`, except on `/dev/usb/usbTTY`).

Hubs

USB hubs are the hidden magic in the whole system. They are remarkably complex (especially with USB 2.0), and yet the user just sees them as a box that allows you to plug in more stuff. The hub driver is built into the USB core code, and each of the host controllers implements a virtual hub (called the root hub), so every port in the system is treated identical.

USB 1.1 hub support is pretty solid, and the current hub code in 2.4 and 2.5 supports USB 2.0 hubs, although some of the advanced functionality like the transaction translator (TT) support needs more work.

Linux USB Development

> **About this section:** This section is possibly apocryphal. Some of the events depicted in this section may be based on real events, and may have occurred to in inexperienced developer...

So you work on a driver for a while, get it so it mostly works, and you send it Alan Cox for the -ac patches - always a good spot to get some more testing before you find that brown paper bag bug. Eventually Alan partly merges it to Linus. You wait for a while longer, and nothing seems to be happening.

So you throw together a quick patch to contain all the missing parts (Config.in, Configure.help - important to keep ESR off your back at that stage, even if no-one ever reads it, Makefile, MAINTAINERS), and send it off to Linus, CC: the USB maintainer and the USB developers mailing list, and this other maintainer, not because there is any real issue, but you did make a really minor change to some of their code:

> This patch (against 2.4.10-pre15) enables the USB CDC Ethernet driver, which is included in the current kernels, but is not "enabled" with Configure.help, Config.in and Makefile entries.
>
> In addition to enabling the driver selection, this patch also modifies the way device probing is done for USB CDC support (both abstract control model and ethernet model), by switching to probing on interface, rather than device, descriptors. This should make probing more efficient.

Now you've read Rusty's saga, and you know it takes at least eight submissions to get Linus to accept anything, but someone on the mailing list will probably give it a try, and just in case you get really lucky, you prepare yourself for some problems to be discovered during on the next test pre-patch kernel, because that is what pre-patches are for, right?

So you get an email from the USB maintainer:

> Any reason they acm driver shouldn't use the USB_DEVICE_ID_MATCH_INT_INFO() macro instead of specifying just USB_DEVICE_ID_MATCH_INT_CLASS and USB_DEVICE_ID_MATCH_INT_SUBCLASS like you did?

and you propose a solution, throw in something extra to prove you didn't go out of your way to avoid using a convenience macro, and submit it.

You get another email:

> Yeah, a cleaner patch, removing that check would be great.
>
> And make it against 2.4.10, since your patch made it into there, going around the maintainer's back, not nice :)

A a little more checking shows that Linus decided to apply this one, where the Changelog for 2.4.10 reads

> final:
>
> Andrew Grover: ACPI update
>
> Al Viro: block devices..
>
> Andrea Arcangeli: fix list manipulation bogosity
>
> Trond Myklebust: 64-bit file locking fixes
>
> Brad Hards: USB CDC ethernet
>
> Chris Mason: reiserfs speedup
>
> Robert Love: re-merge AMD 761 GART support that was lost in -ac merge
>
> Adam Richter: check pci_module_init() return value

... and then it all comes unravelled.

You get another email, from the USB modem maintainer:

> I just received an e-mail that some modems do not work in 2.4.10

Oops.

And then you spend a couple of hours figuring out why (oh, the joys of alternate configurations, and hardware with vendor specific attributes), and come up with a couple of options.

Naturally you end up with reverting back to the old working code.

Future

Predicting the future is always dangerous, so I won't try to get into too much detail, but some of the changes that we can expect in the next couple of kernel releases (ie. 2.6 and the next stable series) are:

1. Full USB 2.0 hub support. Implementation of the Transaction Translator support. Making the little lights flash.
2. Refactoring of the USB webcam support. There is already a usbvideo library, but most drivers don't use it.
3. Monitor class support. There is some improvements required in the hiddev support for more sophisticated devices. The rest of the problem can be solved in user space, including some nice GUI applications.

4. Power class support. There is already some support in apcupsd and Network UPS Tools (NUT), but more support seems likely.
5. USB 2.0 devices and consequences. A few manufacturers have announced plans for USB 2.0 devices that will probably need higher performance / lower latency drivers. As an example, consider a high speed scanner or webcams that can deliver 200Mbps. We already know that we can't sustain more than about 20% of the theoretical limit on current USB 2.0 hard drives without doing some magic to assure contiguous buffers. USB 2.0 networking is probably going to need careful work to get decent performance.
6. Power management support. The new driverfs work (which has been lead with USB and PCI) fits well with USB, and this can likely solve many of the power management problems that people have been suffering. USB remote-wakeup will likely be able to be fully implemented.
7. Generally keeping up with changes inflicted upon us.
8. Various refactoring, including converting the CDC ACM modem to be a serial driver. This will free up a couple of major numbers.
9. Changing the way in which minor numbers get allocated to a fully dynamic arrangment (where you can get minor numbers as required (no prior allocation of sixteen minors to a single driver, irrespective of the number of devices required). This means that the drivers can support any number of devices. This is an alternative model, and if successful will likely be extended to the rest of the kernel, effectively obsoleting devfs.
10. User space improvements and improving hotplug.Our current hotplug system is pretty basic, yet relatively powerful. What is needs is more sophisticated arrangements, such as those for doing zero-configuration networking. This will probably need some more user space tools, and perhaps some better communication channels for advising of status.

    There will also be userspace changes required to support the fully dynamic device allocation.

USBOnTheGo?

USB on the Go ("OTG") is a low performance, low cost way for a device to switch to being a host. The concept is that a device like a PDA would normally be used with a workstation, but it might be useful to plug in a keyboard or mouse occasionally. To do this, the PDA needs to be able to act as both a host and a device. However its host requirements aren't really that great, so it might make sense to provide a cut-down form of the host controller.

The OTG stuff is pretty new, and there aren't any real applications yet. Notwithstanding, Linux already has host controller support for one of the chipsets...

Tips for USB under Linux

- Don't use bus-powered hubs. Use hubs with power supplies attached.
- Search archives before posting user questions to linux-usb-users@lists.sourceforge.net
- Don't post user questions to the developer mailing list (or cross-post).
- Don't post USB questions just to the linux-kernel list.
- Don't use mass storage devices with 2.2 vintage kernels.
- >Boot Protocol drivers for mouse and keyboard are very limited, and should never be used. Even if you are sure it is right, you shouldn't use this. Everyone should use the input drivers.
- Mass storage device support feeds into the SCSI layer. You need to have the right kind of SCSI drivers installed (eg. "scd" and "sg" for USB CD-RW).