EHCI driver

27-Dec-2002

The EHCI driver is used to talk to high speed USB 2.0 devices using USB 2.0-capable host controller hardware. The USB 2.0 standard is compatible with the USB 1.1 standard. It defines three transfer speeds:

- "High Speed" 480 Mbit/sec (60 MByte/sec)
- "Full Speed" 12 Mbit/sec (1.5 MByte/sec)
- "Low Speed" 1.5 Mbit/sec

USB 1.1 only addressed full speed and low speed. High speed devices can be used on USB 1.1 systems, but they slow down to USB 1.1 speeds.

USB 1.1 devices may also be used on USB 2.0 systems. When plugged into an EHCI controller, they are given to a USB 1.1 "companion" controller, which is a OHCI or UHCI controller as normally used with such devices. When USB 1.1 devices plug into USB 2.0 hubs, they interact with the EHCI controller through a "Transaction Translator" (TT) in the hub, which turns low or full speed transactions into high speed "split transactions" that don't waste transfer bandwidth.

At this writing, this driver has been seen to work with implementations of EHCI from (in alphabetical order): Intel, NEC, Philips, and VIA. Other EHCI implementations are becoming available from other vendors; you should expect this driver to work with them too.

While usb-storage devices have been available since mid-2001 (working quite speedily on the 2.4 version of this driver), hubs have only been available since late 2001, and other kinds of high speed devices appear to be on hold until more systems come with USB 2.0 built-in. Such new systems have been available since early 2002, and became much more typical in the second half of 2002.

Note that USB 2.0 support involves more than just EHCI. It requires other changes to the Linux-USB core APIs, including the hub driver, but those changes haven't needed to really change the basic "usbcore" APIs exposed to USB device drivers.

- David Brownell <dbrownell@users.sourceforge.net>

Functionality

This driver is regularly tested on x86 hardware, and has also been used on PPC hardware so big/little endianness issues should be gone. It's believed to do all the right PCI magic so that I/O works even on systems with interesting DMA mapping issues.

Transfer Types

At this writing the driver should comfortably handle all control, bulk, and interrupt transfers, including requests to USB 1.1 devices through transaction translators (TTs) in USB 2.0 hubs. But you may find bugs.

High Speed Isochronous (ISO) transfer support is also functional, but at this writing no Linux drivers have been using that support.

Full Speed Isochronous transfer support, through transaction translators, is not yet available. Note that split transaction support for ISO transfers can't share much code with the code for high speed ISO transfers, since EHCI represents these with a different data structure. So for now, most USB audio and video devices can't be connected to high speed buses.

Driver Behavior

Transfers of all types can be queued. This means that control transfers from a driver on one interface (or through usbfs) won't interfere with ones from another driver, and that interrupt transfers can use periods of one frame without risking data loss due to interrupt processing costs.

The EHCI root hub code hands off USB 1.1 devices to its companion controller. This driver doesn't need to know anything about those drivers; a OHCI or UHCI driver that works already doesn't need to change just because the EHCI driver is also present.

There are some issues with power management; suspend/resume doesn't behave quite right at the moment.

Also, some shortcuts have been taken with the scheduling periodic transactions (interrupt and isochronous transfers). These place some limits on the number of periodic transactions that can be scheduled, and prevent use of polling intervals of less than one frame.

Use by

Assuming you have an EHCI controller (on a PCI card or motherboard) and have compiled this driver as a module, load this like:

```
# modprobe ehci-hcd
```

and remove it by:

```
# rmmod ehci-hcd
```

You should also have a driver for a "companion controller", such as "ohci-hcd" or "uhci-hcd". In case of any trouble with the EHCI driver, remove its module and then the driver for that companion controller will take over (at lower speed) all the devices that were previously handled by the EHCI driver.

Module parameters (pass to "modprobe") include:

log2_irq_thresh (default 0):
: Log2 of default interrupt delay, in microframes. The default value is 0, indicating 1 microframe (125 usec). Maximum value is 6, indicating 2^6 = 64 microframes. This controls how often the EHCI controller can issue interrupts.

If you're using this driver on a 2.5 kernel, and you've enabled USB debugging support, you'll see three files in the "sysfs" directory for any EHCI controller:

"async"
: dumps the asynchronous schedule, used for control and bulk transfers. Shows each active qh and the qtds pending, usually one qtd per urb. (Look at it with usb-storage doing disk I/O; watch the request queues!)

"periodic"
: dumps the periodic schedule, used for interrupt and isochronous transfers. Doesn't show qtds.

"registers"
: show controller register state, and

The contents of those files can help identify driver problems.

Device drivers shouldn't care whether they're running over EHCI or not, but they may want to check for "usb_device->speed == USB_SPEED_HIGH". High speed devices can do things that full speed (or low speed) ones can't, such as "high bandwidth" periodic (interrupt or ISO) transfers. Also, some values in device descriptors (such as polling intervals for periodic transfers) use different encodings when operating at high speed.

However, do make a point of testing device drivers through USB 2.0 hubs. Those hubs report some failures, such as disconnections, differently when transaction translators are in use; some drivers have been seen to behave badly when they see different faults than OHCI or UHCI report.

Performance

USB 2.0 throughput is gated by two main factors: how fast the host controller can process requests, and how fast devices can respond to them. The 480 Mbit/sec "raw transfer rate" is obeyed by all devices, but aggregate throughput is also affected by issues like delays between individual high speed packets, driver intelligence, and of course the overall system load. Latency is also a performance concern.

Bulk transfers are most often used where throughput is an issue. It's good to keep in mind that bulk transfers are always in 512 byte packets, and at most 13 of those fit into one USB 2.0 microframe. Eight USB 2.0 microframes fit in a USB 1.1 frame; a microframe is 1 msec/8 = 125 usec.

So more than 50 MByte/sec is available for bulk transfers, when both hardware and device driver software allow it. Periodic transfer modes (isochronous and interrupt) allow the larger packet sizes which let you approach the quoted 480 MBit/sec transfer rate.

Hardware Performance

At this writing, individual USB 2.0 devices tend to max out at around 20 MByte/sec transfer rates. This is of course subject to change; and some devices now go faster, while others go slower.

The first NEC implementation of EHCI seems to have a hardware bottleneck at around 28 MByte/sec aggregate transfer rate. While this is clearly enough for a single device at 20 MByte/sec, putting three such devices onto one bus does not get you 60 MByte/sec. The issue appears to be that the controller hardware won't do concurrent USB and PCI access, so that it's only trying six (or maybe seven) USB transactions each microframe rather than thirteen. (Seems like a reasonable trade off for a product that beat all the others to market by over a year!)

It's expected that newer implementations will better this, throwing more silicon real estate at the problem so that new motherboard chip sets will get closer to that 60 MByte/sec target. That includes an updated implementation from NEC, as well as other vendors' silicon.

There's a minimum latency of one microframe (125 usec) for the host to receive interrupts from the EHCI controller indicating completion of requests. That latency is tunable; there's a module option. By default ehci-hcd driver uses the minimum latency, which means that if you issue a control or bulk request you can often expect to learn that it completed in less than 250 usec (depending on transfer size).

Software Performance

To get even 20 MByte/sec transfer rates, Linux-USB device drivers will need to keep the EHCI queue full. That means issuing large requests, or using bulk queuing if a series of small requests needs to be issued. When drivers don't do that, their performance results will show it.

In typical situations, a usb_bulk_msg() loop writing out 4 KB chunks is going to waste more than half the USB 2.0 bandwidth.

Delays between the I/O completion and the driver issuing the next request will take longer than the I/O. If that same loop used 16 KB chunks, it'd be better; a sequence of 128 KB chunks would waste a lot less.

But rather than depending on such large I/O buffers to make synchronous I/O be efficient, it's better to just queue up several (bulk) requests to the HC, and wait for them all to complete (or be canceled on error). Such URB queuing should work with all the USB 1.1 HC drivers too.

In the Linux 2.5 kernels, new usb_sg_*() api calls have been defined; they queue all the buffers from a scatterlist. They also use scatterlist DMA mapping (which might apply an IOMMU) and IRQ reduction, all of which will help make high speed transfers run as fast as they can.

TBD:
: Interrupt and ISO transfer performance issues. Those periodic transfers are fully scheduled, so the main issue is likely to be how to trigger "high bandwidth" modes.

TBD:
: More than standard 80% periodic bandwidth allocation is possible through sysfs uframe_periodic_max parameter. Describe that.