Linux Kernel Module Programming

Anandkumar

July 11, 2021

USB Driver

What is USB?

- USB stands for Universal Serial Bus
- Provides an expandable, fast, bi-directional, low cost, hot pluggable Plug and Play serial hardware interface
- Allows users to connect a wide variety of peripherals to a computer and have them automatically configured and ready to use
- Implemented to provide a replacement for legacy ports to make the addition of peripheral devices quick and easy for the end user

Pre-Releases of USB

- **USB 0.7:** Released in November 1994.
- **USB 0.8:** Released in December 1994.
- **USB 0.9:** Released in April 1995.
- **USB 0.99:** Released in August 1995.
- **USB 1.0:** Released in November 1995

History of USB

- There have been three versions released prior to 3.0

  - USB 1.0 in January 1996 – data rates of 1.5 Mbps up to 12 Mbps

  - USB 1.1 in September 1998 – first widely used version of USB

  - USB 2.0 in April 2000

    Major feature revision was the addition of a high speed transfer rate of 480 Mbps

USB 3.0 Now

- On Nov 17,2008 It was Developed
- It is called as “SUPER SPEED” Technology
- Transfer Mode of Up to 4.8 Gbps

Key Features

- Single connector type
  - Replaces all different legacy connectors with one well-defined standardized USB connector for all USB peripheral devices
- Hot swappable
  - Devices can be safely plugged and unplugged as needed while the computer is running (no need to reboot)
- Plug and Play
  - OS software automatically identifies, configures, and loads the appropriate driver when connection is made

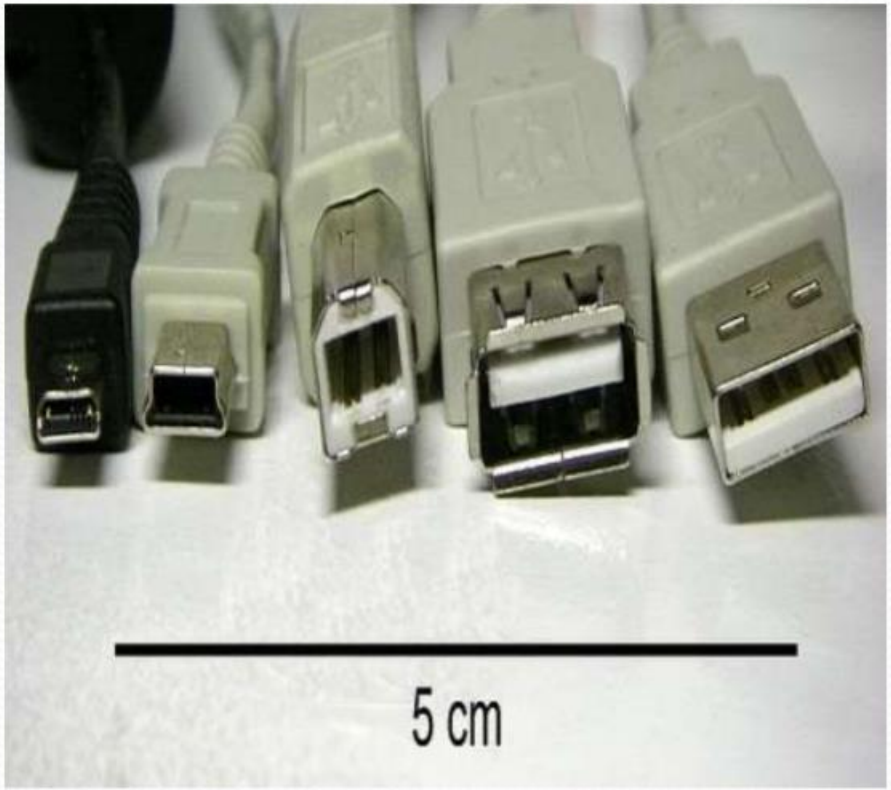
Key Features

- High performance
  - USB offers data transfer speeds at up to 4.8 Gbps
- Expandability
  - Up to 127 different peripheral devices may theoretically be connected to a single bus at one time
- Bus-supplied power
  - USB distributes the power to all connected devices, eliminating the need for an external power source for low power devices (flash drives, memory cards, Bluetooth)

Connector properties

- Availability
  - Consumer Products are expected to become available in 2010
- Usability
  - Most connectors cannot be plugged in upside down
- Durability
  - The standard connectors were designed to be robust
- Compatibility
  - Two-way communication is also possible. In USB 3.0, full-duplex communications are done when using SuperSpeed (USB 3.0) transfer

Connector Types



- • male micro USB
  • male mini USB B-type
  • male B-type
  • female A-type
  • male A-type

Pin	Name	Cable Color	Description
1	VCC	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	GND	Black	Ground

Maximum Useful Distance

- **USB 1.1** maximum cable length is 3 metres (9.8 ft)

- **USB 2.0** maximum cable length is 5 metres (16 ft)

- **USB 3.0** cable assembly may be of any length

USB 2.0 & USB 3.0

APPLICATIONS

- USB implements connections to storage devices using a set of standards called the USB mass storage device class.
- USB 3.0 can also support portable hard disk drives. The earlier versions of USBs were not supporting the 3.5 inch hard disk drives.
- These external drives usually contain a translating device that interfaces a drive of conventional technology (IDE, PATA, SATA, ATAPI, or even SCSI) to a USB port.

Linux USB drivers

Linux USB basics

Linux USB drivers

USB drivers (1)

USB core drivers

- Architecture independent kernel subsystem.
  Implements the USB bus specification.
  Outside the scope of this training.

USB host drivers

- Different drivers for each USB control hardware.
  Usually available in the Board Support Package.
  Architecture and platform dependent.
  Not covered yet by this training.

USB drivers (2)

<u>USB device drivers</u>

- Drivers for devices on the USB bus.
  The main focus of this course!
- Platform independent: when you use Linux on an embedded platform, you can use any USB device supported by Linux
  (cameras, keyboards, video capture, wi-fi dongles...).

<u>USB device controller drivers</u>

- For Linux systems with just a USB device controller (frequent in embedded systems).
  Not covered yet by this course.

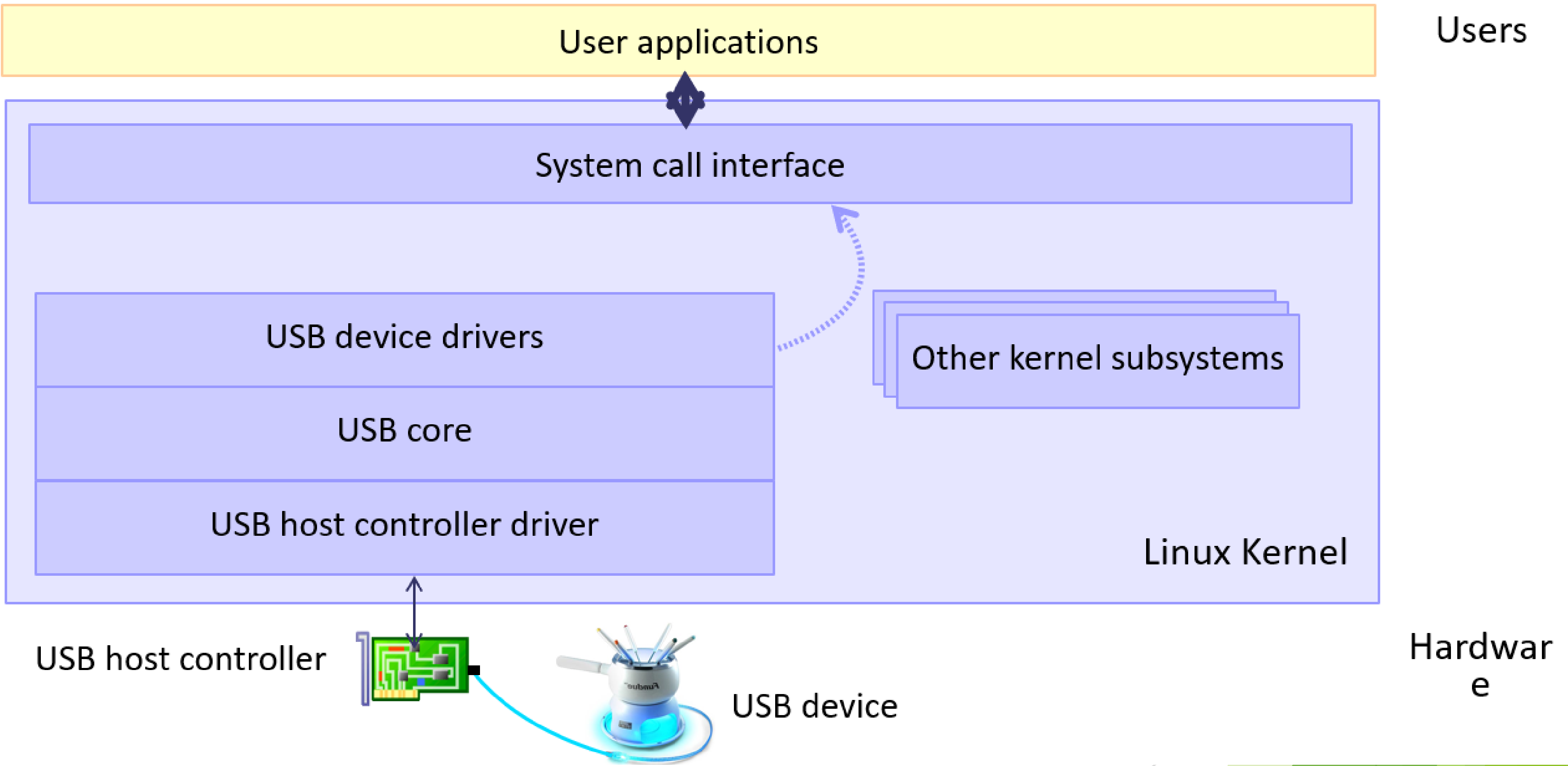
USB gadget drivers

Drivers for Linux systems with a USB device controller

- Typical example: digital cameras.
  You connect the device to a PC and see the camera as a USB storage device.
- USB device controller driver:
  Platform dependent. Supports the chip connecting to the USB bus.
- USB gadget drivers, platform independent. Examples:
  Ethernet gadget: implements networking through USB
  Storage gadget: makes the host see a USB storage device
  Serial gadget: for terminal-type of communication.

See Documentation/DocBook/gadget/ in kernel sources.

Linux USB support overview



USB host controllers - OHCI and UHCI

2 competing Host Control Device (HCD) interfaces

- OHCI - Open Host Controller Interface
  Compaq's implementation adopted as a standard for USB 1.0 and 1.1
  by the USB Implementers Forum (USB-IF).
  Also used for Firewire devices.
- UHCI - Universal Host Controller Interface.
  Created by Intel, insisting that other implementers use it and pay royalties for it. Only VIA licensed UHCI, and others stuck to OHCI.

This competition required to test devices for both host controller standards!

For USB 2.0, the USB-IF insisted on having only one standard.

USB host controllers - EHCI

EHCI - Extended Host Controller Interface.

- For USB 2.0. The only one to support high-speed transfers.
- Each EHCI controller contains four virtual HCD implementations to support Full Speed and Low Speed devices.
- On Intel and VIA chipsets, virtual HCDs are UHCI. Other chipset makers have OHCI virtual HCDs.

USB transfer speed

- Low-Speed: up to 1.5 Mbps
  Since USB 1.0
- Full-Speed: up to 12 Mbps
  Since USB 1.1
- Hi-Speed: up to 480 Mbps
  Since USB 2.0

Linux USB drivers

Linux USB basics

USB devices

USB descriptors

Operating system independent. Described in the USB specification

- Device - Represent the devices connected to the USB bus. Example: USB speaker with volume control buttons.
- Configurations - Represent the state of the device. Examples: Active, Standby, Initialization
- Interfaces - Logical devices. Examples: speaker, volume control buttons.
- Endpoints - Unidirectional communication pipes. Either `IN` (device to computer) or `OUT` (computer to device).

Control endpoints

- Used to configure the device, get information about it, send commands to it, retrieve status information.
- Simple, small data transfers.
- Every device has a control endpoint (endpoint 0), used to configure the device at insertion time.
- The USB protocol guarantees that the corresponding data transfers will always have enough (reserved) bandwidth.

Interrupt endpoints

- Transfer small amounts of data at a fixed rate each time the hosts asks the device for data.
- Guaranteed, reserved bandwidth.
- For devices requiring guaranteed response time, such as USB mice and keyboards.
- Note: different than hardware interrupts. Require constant polling from the host.

Bulk endpoints

- Large sporadic data transfers
  using all remaining available bandwidth.
- No guarantee on bandwidth or latency.
- Guarantee that no data is lost.
- Typically used for printers, storage or network devices.

Isochronous endpoints

- Also for large amounts of data.
- Guaranteed speed
  (often but not necessarily as fast as possible).
- No guarantee that all data makes it through.
- Used by real-time data transfers (typically audio and video).

The `usb_endpoint_descriptor` structure (1)

The `usb_endpoint_descriptor` structure contains all the USB-specific data announced by the device itself. Here are useful fields for driver writers:

- `__u8 bEndpointAddress`:
  USB address of the endpoint.
  It also includes the direction of the endpoint. You can use the `USB_ENDPOINT_DIR_MASK` bitmask to tell whether this is a `USB_DIR_IN` or `USB_DIR_OUT` endpoint.
  Example:

```
if ((endpoint->desc.bEndpointAddress &
 USB_ENDPOINT_DIR_MASK) == USB_DIR_IN)
```

The `usb_endpoint_descriptor` structure (2)

- `__u8 bmAttributes`:
  The type of the endpoint. You can use the `USB_ENDPOINT_XFERTYPE_MASK` bitmask to tell whether the type is `USB_ENDPOINT_XFER_ISOC`, `USB_ENDPOINT_XFER_BULK`, `USB_ENDPOINT_XFER_INT` or `USB_ENDPOINT_XFER_CONTROL`.
- `__u8 wMaxPacketSize`:
  Maximum size in bytes that the endpoint can handle. Note that if greater sizes are used, data will be split in `wMaxPacketSize` chunks.
- `__u8 bInterval`:
  For interrupt endpoints, device polling interval (in milliseconds).

Note that the above names do not follow Linux coding standards. The Linux USB implementation kept the original name from the USB specification (http://www.usb.org/developers/docs/).

Interfaces

- Each interface encapsulates a single high-level function (USB logical connection). Example (USB webcam): video stream, audio stream, keyboard (control buttons).
- One driver is needed for each interface!
- Alternate settings: each USB interface may have different parameter settings. Example: different bandwidth settings for an audio interface. The initial state is in the first setting, (number `0`).
- Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-compliant USB devices that use isochronous endpoints will use them in non-default settings.

The `usb_interface` structure (1)

USB interfaces are represented by the `usb_interface` structure. It is what the USB core passes to USB drivers.

- `struct usb_host_interface *altsetting;`
  List of alternate settings that may be selected for this interface, in no particular order.
  The `usb_host_interface` structure for each alternate setting allows to access the `usb_endpoint_descriptor` structure for each of its endpoints:
  `interface->alsetting[i]->endpoint[j]->desc`
- `unsigned int num_altsetting;`
  The number of alternate settings.

The `usb_interface` structure (2)

- `struct usb_host_interface *cur_altsetting;`
  The currently active alternate setting.
- `int minor;`
  Minor number this interface is bound to.
  (for drivers using `usb_register_dev()`, described later).

Other fields in the structure shouldn't be needed by USB drivers.

Configurations

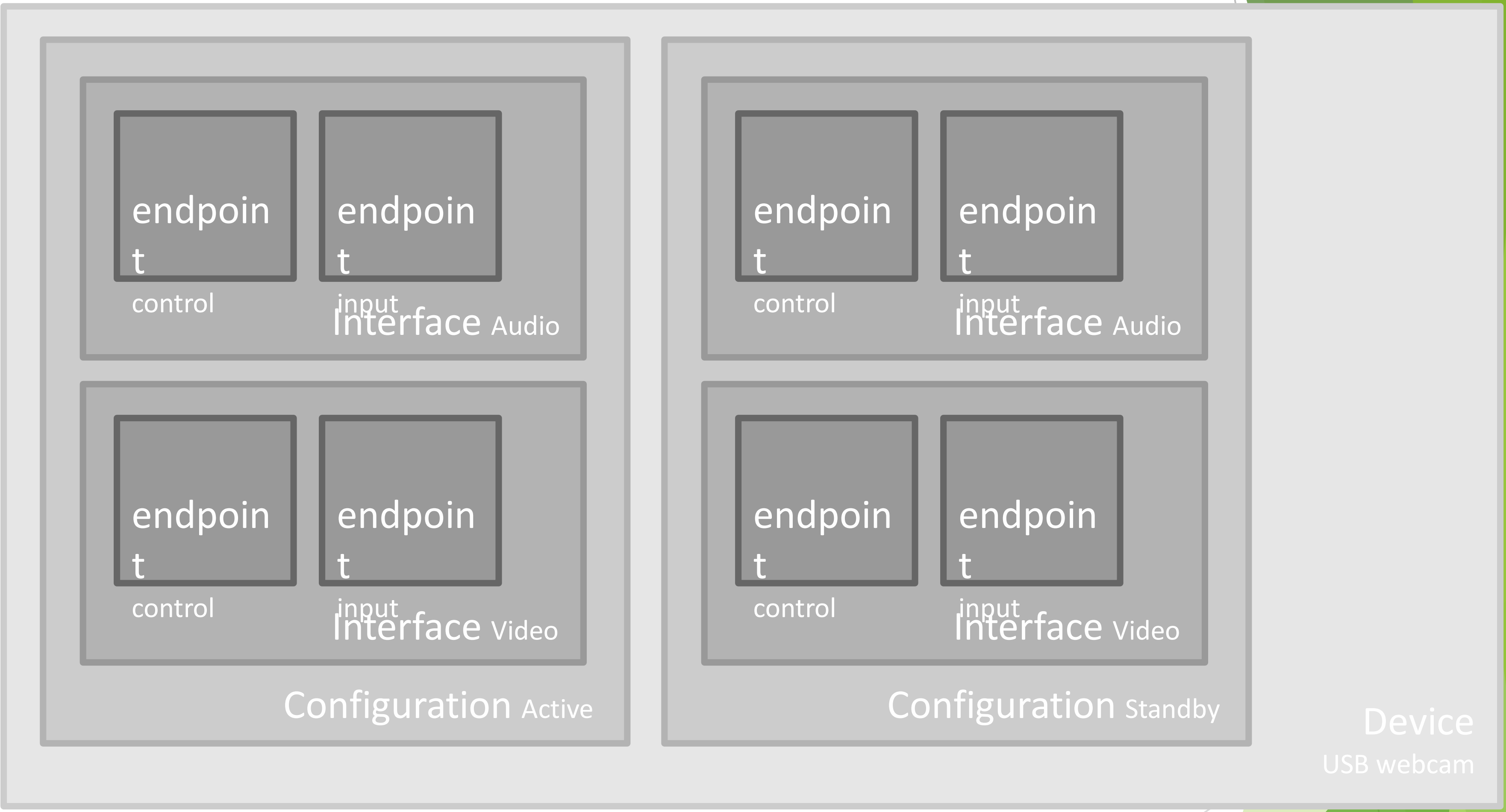
Interfaces are bundled into configurations.

- Configurations represent the state of the device.
  Examples: Active, Standby, Initialization
- Configurations are described
  with the `usb_host_config` structure.
- However, drivers do not need to access this structure.

Devices

- Devices are represented by the `usb_device` structure.
- We will see later that several USB API functions need such a structure.
- Many drivers use the `interface_to_usbdev()` function to access their `usb_device` structure from the `usb_interface` structure they are given by the USB core.

USB device overview



USB devices - Summary

- Hierarchy: device configurations interfaces endpoints
- 4 different types of endpoints
  - control: device control, accessing information, small transfers. Guaranteed bandwidth.
  - interrupt (keyboards, mice...): data transfer at a fixed rate. Guaranteed bandwidth.
  - bulk (storage, network, printers...): use all remaining bandwidth. No bandwidth or latency guarantee.
  - isochronous (audio, video...): guaranteed speed. Possible data loss.

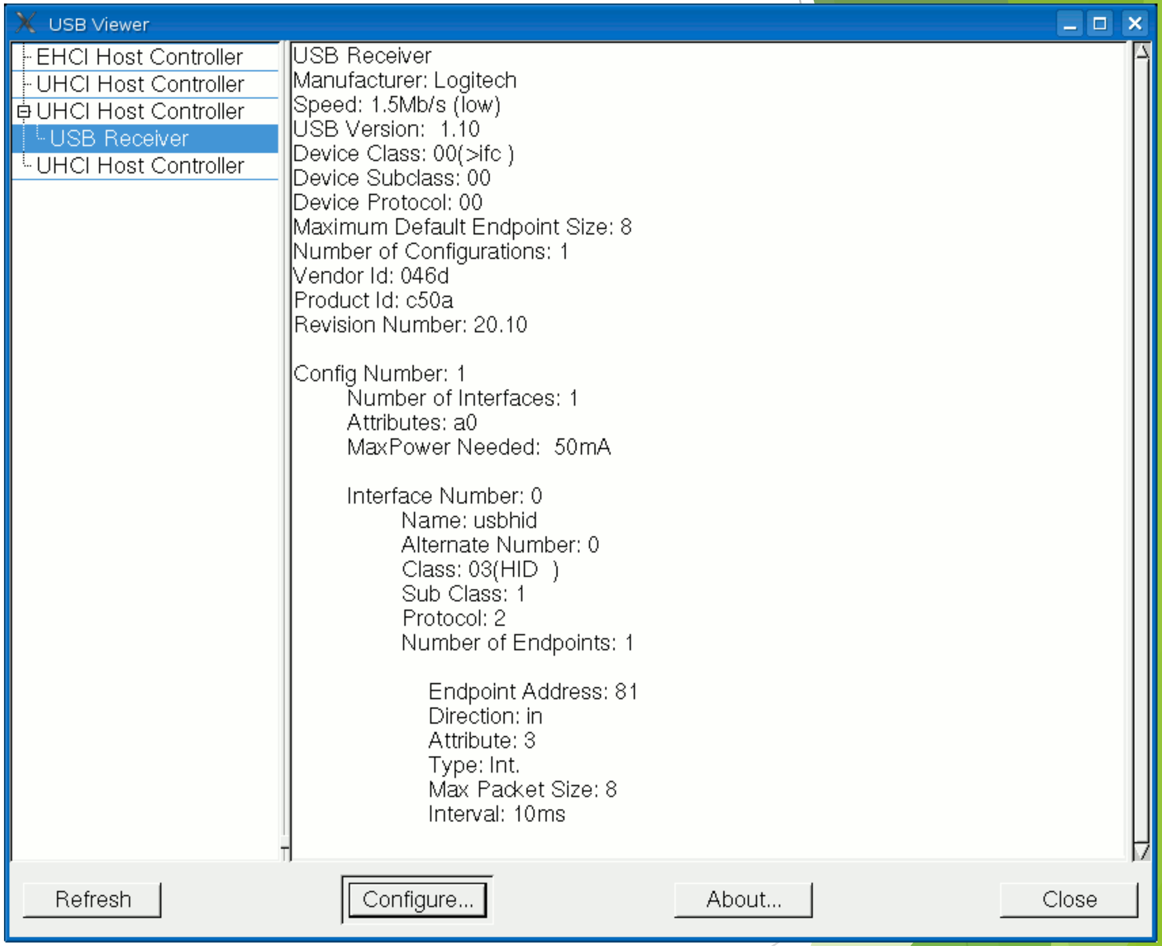
Linux USB drivers

Linux USB basics

User-space representation

usbview

http://usbview.sourceforge.net
Graphical display of the contents of `/proc/bus/usb/devices`.



usbtree

http://www.linux-usb.org/usbtree

Also displays information from `/proc/bus/usb/devices`:

```
> usbtree
 \/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/6p, 
 480M
 /:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
 /:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
     |__ Port 1: Dev 7, If 0, Class=HID, Driver=usbhid, 1.5M
 /:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
```

Linux USB drivers

Linux USB communication

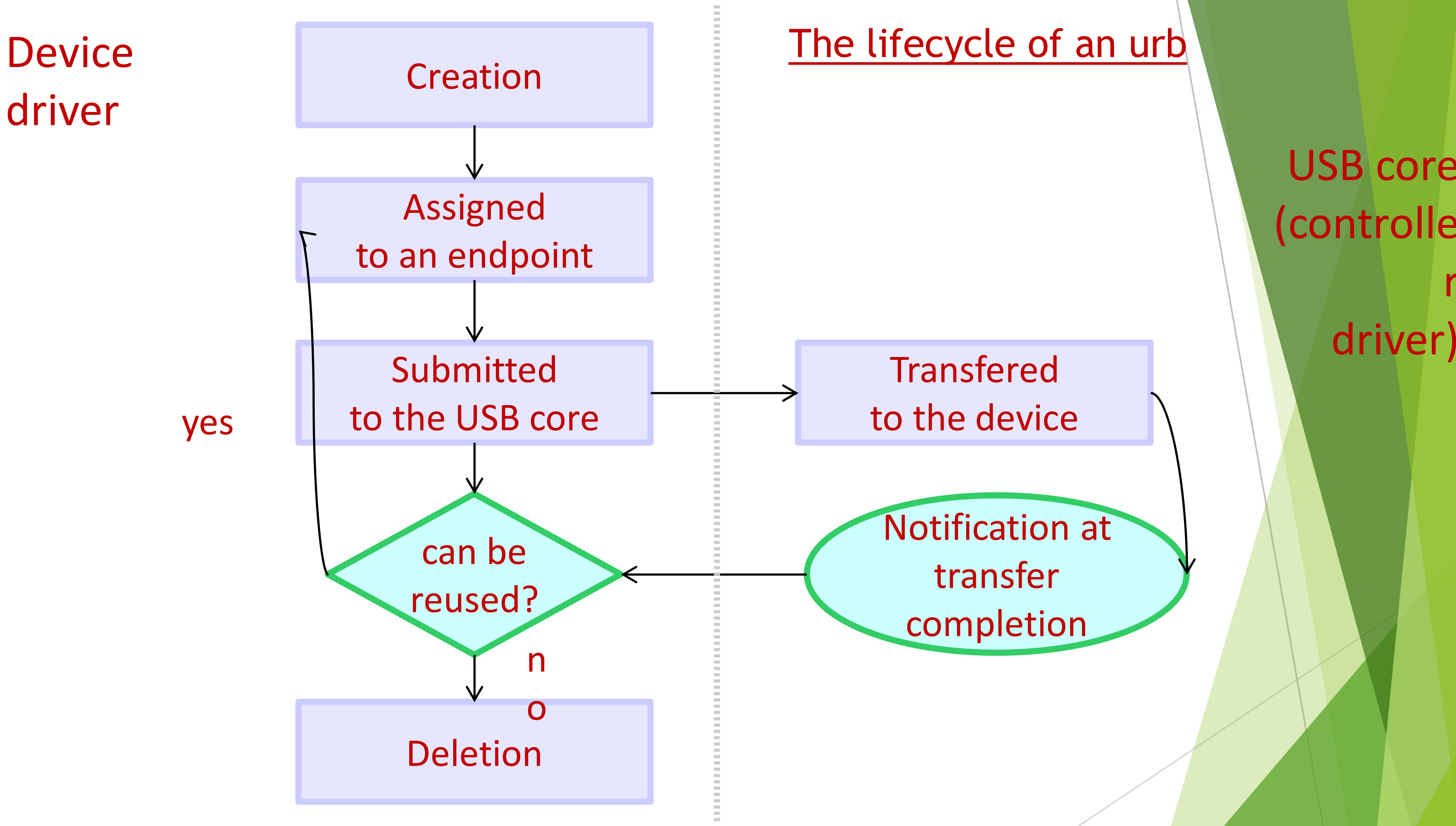
USB Request Blocks

USB Request Blocks

- Any communication between the host and device is done asynchronously using USB Request Blocks (urbs).
- They are similar to packets in network communications.
- Every endpoint can handle a queue of urbs.
- Every urb has a completion handler.
- A driver may allocate many urbs for a single endpoint, or reuse the same urb for different endpoints.

See `Documentation/usb/URB.txt` in kernel sources.

Urban life


The lifecycle of an urb
Device driver
USB core (controller driver)
Creation
Assigned to an endpoint
Submitted to the USB core
Transfered to the device
Notification at transfer completion
can be reused?
yes
no
Deletion

The `urb` structure (1)

Fields of the `urb` structure useful to USB device drivers:

- `struct usb_device *dev;`
  Device the urb is sent to.
- `unsigned int pipe;`
  Information about the endpoint in the target device.
- `int status;`
  Transfer status.
- `unsigned int transfer_flags;`
  Instructions for handling the urb.

The `urb` structure (2)

- `void * transfer_buffer;`
  Buffer storing transferred data.
  Must be created with `kmalloc()`!
- `dma_addr_t transfer_dma;`
  Data transfer buffer when DMA is used.
- `int transfer_buffer_length;`
  Transfer buffer length.
- `int actual_length;`
  Actual length of data received or sent by the urb.
- `usb_complete_t complete;`
  Completion handler called when the transfer is complete.

The `urb` structure (3)

- `void *context;`
  Data blob which can be used in the completion handler.
- `unsigned char *setup_packet;` (control urbs)
  Setup packet transferred before the data in the transfer buffer.
- `dma_addr_t setup_dma;` (control urbs)
  Same, but when the setup packet is transferred with DMA.
- `int interval;` (isochronous and interrupt urbs)
  Urb polling interval.
- `int error_count;` (isochronous urbs)
  Number of isochronous transfers which reported an error.

The `urb` structure (4)

- `int start_frame;` (isochronous urbs)
  Sets or returns the initial frame number to use.
- `int number_of_packets;` (isochronous urbs)
  Number of isochronous transfer buffers to use.
- `struct usb_iso_packet_descriptor` (isochronous urbs)
  `iso_frame_desc[0];`
  Allows a single urb to define multiple isochronous transfers at once.

Creating pipes

Functions used to initialize the `pipe` field of the `urb` structure:

- Control pipes
  `usb_sndctrlpipe()`, `usb_rcvctrlpipe()`
- Bulk pipes
  `usb_sndbulkpipe()`, `usb_rcvbulkpipe()`
- Interrupt pipes
  `usb_sndintpipe()`, `usb_rcvintpipe()`
- Isochronous pipes
  `usb_sndisocpipe()`, `usb_rcvisocpipe()`

send (out)

receive (in)

Prototype

```
unsigned int usb_[snd/rcv][ctrl/bulk/int/isoc]pipe(
    struct usb_device *dev, unsigned int endpoint);
```

Creating urbs

- `urb` structures must always be allocated with the `usb_alloc_urb()` function.
  That's needed for reference counting used by the USB core.

```
#include <linux/usb.h>

struct urb *usb_alloc_urb(
   int iso_packets,      // Number of isochronous
                      // packets the urb should contain.
                           // 0 for other transfer types
   gfp_t mem_flags);     // Standard kmalloc() flags
```

- Check that it didn't return `NULL` (allocation failed)!
- Typical example:
  `urb = usb_alloc_urb(0, GFP_KERNEL);`

Freeing urbs

- Similarly, you have to use a dedicated function to release urbs:

```
void usb_free_urb(struct urb *urb);
```

USB Request Blocks - Summary

- Basic data structure used in any USB communication.
- Implemented by the `struct urb` type.
- Must be created with the `usb_alloc_urb()` function. Shouldn't be allocated statically or with `kmalloc()`.
- Must be deleted with `usb_free_urb()`.

Linux USB drivers

Linux USB communication

Initializing and submitting urbs

Initializing interrupt urbs

```
void usb_fill_int_urb (
   struct urb *urb,                // urb to be initialized
   struct usb_device *dev, // device to send the urb to
   unsigned int pipe,              // pipe (endpoint and device
   specific)
   void *transfer_buffer,  // transfer buffer
   int buffer_length,              // transfer buffer size
   usb_complete_t complete,        // completion handler
   void *context,                  // context (for handler)
   int interval                    // Scheduling interval (see next
   page)
   );
```

- This doesn't prevent you from making more changes to the urb fields before urb submission.
- The `transfer_flags` field needs to be set by the driver.

urb scheduling interval

For interrupt and isochronous transfers

- Low-Speed and Full-Speed devices:
  the `interval` unit is frames (ms)
- Hi-Speed devices:
  the `interval` unit is microframes (1/8 ms)

Initializing bulk urbs

Same parameters as in `usb_fill_int_urb()`, except that there is no `interval` parameter.

```
void usb_fill_bulk_urb (
  struct urb *urb,                  // urb to be initialized
  struct usb_device *dev,  // device to send the urb to
  unsigned int pipe,           // pipe (endpoint and device specific)
  void *transfer_buffer,    // transfer buffer
  int buffer_length,           // transfer buffer size
  usb_complete_t complete,      // completion handler
  void *context,                  // context (for handler)
  );
```

Initializing control urbs

Same parameters as in usb_fill_bulk_urb(), except that there is a setup_packet parameter.

```
void usb_fill_control_urb (
   struct urb *urb,                // urb to be initialized
   struct usb_device *dev,         // device to send the urb to
   unsigned int pipe,              // pipe (endpoint and device specific)
   unsigned char *setup_packet,    // setup packet data
   void *transfer_buffer,          // transfer buffer
   int buffer_length,              // transfer buffer size
   usb_complete_t complete,        // completion handler
   void *context,                  // context (for handler)
   );
```

Note that many drivers use the usb_control_msg() function instead (explained later).

Initializing isochronous urbs

No helper function. Has to be done manually by the driver.

```
for (i=0; i < USBVIDEO_NUMSBUF; i++) {
    int j, k;
    struct urb *urb = uvd->sbuf[i].urb;
    urb->dev = dev;
    urb->context = uvd;
    urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp);
    urb->interval = 1;
    urb->transfer_flags = URB_ISO_ASAP;
    urb->transfer_buffer = uvd->sbuf[i].data;
    urb->complete = usbvideo_IsocIrq;
    urb->number_of_packets = FRAMES_PER_DESC;
    urb->transfer_buffer_length = uvd->iso_packet_len * FRAMES_PER_DESC;
    for (j=k=0; j < FRAMES_PER_DESC; j++, k += uvd->iso_packet_len) {
        urb->iso_frame_desc[j].offset = k;
        urb->iso_frame_desc[j].length = uvd->iso_packet_len;
    }
}
```

drivers/media/video/usbvideo/usbvideo.c

Allocating DMA buffers (1)

You can use the usb_buffer_alloc() function to allocate a DMA consistent buffer:

```
void *usb_buffer_alloc (
    struct usb_device *dev, // device
    size_t size,            // buffer size
    gfp_t mem_flags,        // kmalloc() flags
    dma_addr_t *dma         // (output) DMA address
);                          // of the buffer.
```

Example:

```
buf = usb_buffer_alloc(dev->udev,
    count, GFP_KERNEL, &urb->transfer_dma);
```

Allocating DMA buffers (2)

- To use these buffers, use the `URB_NO_TRANSFER_DMA_MAP` or `URB_NO_SETUP_DMA_MAP` settings for `urb->transfer_flags` to indicate that `urb->transfer_dma` or `urb->setup_dma` are valid on submit.
- Examples:

```
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
u->transfer_flags |= URB_NO_SETUP_DMA_MAP;
```

- Freeing these buffers:

```
void usb_buffer_free (
 struct usb_device *dev,    // device
 size_t size,               // buffer size
 void *addr,                // CPU address of buffer
 dma_addr_t dma             // DMA address of buffer
);
```

Submitting urbs

After creating and initializing the urb

```
int usb_submit_urb(
   struct urb *urb,        // urb to submit
   int mem_flags);     // kmalloc() flags
```

`mem_flags` is used for internal allocations performed by `usb_submit_urb()`. Settings that should be used:

- `GFP_ATOMIC`: called from code which cannot sleep: a urb completion handler, hard or soft interrupts. Or called when the caller holds a spinlock.
- `GPF_NOIO`: in some cases when block storage is used.
- `GFP_KERNEL`: in other cases.

usb_submit_urb return values

usb_submit_urb() immediately returns:

- 0: Request queued
- -ENOMEM: Out of memory
- -ENODEV: Unplugged device
- -EPIPE: Stalled endpoint
- -EAGAIN: Too many queued ISO transfers
- -EFBIG: Too many requested ISO frames
- -EINVAL: Invalid INT interval
  More than one packet for INT

Canceling urbs asynchronously

To cancel a submitted urb without waiting

- `int usb_unlink_urb(struct urb *urb);`
- Success: returns `-EINPROGRESS`
- Failure: any other return value. It can happen:
  - When the urb was never submitted
  - When the has already been unlinked
  - When the hardware is done with the urb,
    even if the completion handler hasn't been called yet.
- The corresponding completion handlers will still be run
  and will see `urb->status == -ECONNRESET`.

Canceling urbs synchronously

To cancel an urb and wait for all completion handlers to complete

- This guarantees that the urb is totally idle and can be reused.
- `void usb_kill_urb(struct urb *urb);`
- Typically used in a `disconnect()` callback or `close()` function.
- Caution: this routine mustn't be called in situations which can not sleep: in interrupt context, in a completion handler, or when holding a spinlock.

See comments in `drivers/usb/core/urb.c` in kernel sources for useful details.

Initializing and submitting urbs - Summary

- urb structure fields can be initialized with helper functions usb_fill_int_urb(), usb_fill_bulk_urb() , usb_fill_control_urb()
- Isochronous urbs have to be initialized by hand.
- The transfer_flags field must be initialized manually by each driver.
- Use the usb_submit_urb() function to queue urbs.
- Submitted urbs can be canceled using usb_unlink_urb() (asynchronous) or usb_kill_urb() (synchronous).

Linux USB drivers

Linux USB communication

Completion handlers

When is the completion handler called?

The completion handler is called in interrupt context, in only 3 situations.
Check the error value in `urb->status`.

- After the data transfer successfully completed.
  `urb->status == 0`
- Error(s) happened during the transfer.
- The urb was unlinked by the USB core.

`urb->status` should only be checked from the completion handler!

Transfer status (1)

Described in Documentation/usb/error-codes.txt

The urb is no longer “linked” in the system

- -ECONNRESET
  The urb was unlinked by usb_unlink_urb().
- -ENOENT
  The urb was stopped by usb_kill_urb().
- -ESHUTDOWN
  Error in from the host controller driver. The device was disconnected from the system, the controller was disabled, or the configuration was changed while the urb was sent.
- -ENODEV
  Device removed. Often preceded by a burst of other errors, since the hub driver doesn't detect device removal events immediately.

Transfer status (2)

Typical hardware problems with the cable or the device (including its firmware)

- -EPROTO
  Bitstuff error, no response packet received in time by the hardware,
  or unknown USB error.
- -EILSEQ
  CRC error, no response packet received in time, or unknown USB error.
- -EOVERFLOW
  The amount of data returned by the endpoint was greater than either the max packet size of the endpoint or the remaining buffer size. "Babble".

Transfer status (3)

Other error status values

- -EINPROGRESS
  Urb not completed yet. Your driver should never get this value.
- -ETIMEDOUT
  Usually reported by synchronous USB message functions when the specified timeout was exceed.
- -EPIPE
  Endpoint stalled. For non-control endpoints, reset this status with usb_clear_halt().
- -ECOMM
  During an IN transfer, the host controller received data from an endpoint faster than it could be written to system memory.

Transfer status (4)

- `-ENOSR`
  During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with the USB data rate.
- `-EREMOTEIO`
  The data read from the endpoint did not fill the specified buffer, and `URB_SHORT_NOT_OK` was set in `urb->transfer_flags`.
- `-EXDEV`
  Isochronous transfer only partially completed.
  Look at individual frame status for details.
- `-EINVAL`
  Typically happens with an incorrect urb structure field or `usb_submit_urb()` function parameter.

Completion handler implementation

- Prototype:

```
void (*usb_complete_t)(
 struct urb *,      // The completed urb
 struct pt_regs *      // Register values at the time
                       // of the corresponding interrupt (if any)
);
```

- Remember you are in interrupt context:
  - Do not execute call which may sleep (use GFP_ATOMIC, etc.).
  - Complete as quickly as possible.
    Schedule remaining work in a tasklet if needed.

Completion handler - Summary

- The completion handler is called in interrupt context. Don't run any code which could sleep!
- Check the `urb->status` value in this handler, and not before.
- Success: `urb->status == 0`
- Otherwise, error status described in `Documentation/usb/error-codes.txt`.

Linux USB drivers

Writing USB drivers

Supported devices

What devices does the driver support?

Or what driver supports a given device?

- Information needed by user-space, to find the right driver to load or remove after a USB hotplug event.
- Information needed by the driver, to call the right `probe()` and `disconnect()` driver functions (see later).

Such information is declared in a `usb_device_id` structure by the driver `init()` function.

The `usb_device_id` structure (1)

Defined according to USB specifications and described in `include/linux/mod_devicetable.h`.

- `__u16 match_flags`
  Bitmask defining which fields in the structure are to be matched against. Usually set with helper functions described later.
- `__u16 idVendor`, `idProduct`
  USB vendor and product id, assigned by the USB-IF.
- `__u16 bcdDevice_lo`, `bcdDevice_hi`
  Product version range supported by the driver, expressed in binary-coded decimal (BCD) form.

The `usb_device_id` structure (2)

- `__u8 bDeviceClass, bDeviceSubClass, bDeviceProtocol`
  Class, subclass and protocol of the device.
  Numbers assigned by the USB-IF.
  Products may choose to implement classes, or be vendor-specific.
  Device classes specify the behavior of all the interfaces on a device.
- `__u8 bInterfaceClass, bInterfaceSubclass, bInterfaceProtocol`
  Class, subclass and protocol of the individual interface.
  Numbers assigned by the USB-IF.
  Interface classes only specify the behavior of a given interface.
  Other interfaces may support other classes.
- `kernel_ulong_t driver_info`

The `usb_device_id` structure (3)

- `kernel_ulong_t driver_info`
  Holds information used by the driver. Usually it holds a pointer to a descriptor understood by the driver, or perhaps device flags.
  This field is useful to differentiate different devices from each other in the `probe()` function.

Declaring supported devices (1)

`USB_DEVICE(vendor, product)`

- Creates a `usb_device_id` structure which can be used to match only the specified vendor and product ids.
- Used by most drivers for non-standard devices.

`USB_DEVICE_VER(vendor, product, lo, hi)`

- Similar, but only for a given version range.
- Only used 11 times throughout Linux 2.6.18!

Declaring supported devices (2)

USB_DEVICE_INFO (class, subclass, protocol)

- Matches a specific class of USB devices.

USB_INTERFACE_INFO (class, subclass, protocol)

- Matches a specific class of USB interfaces.

The above 2 macros are only used in the implementations of standard device and interface classes.

Declaring supported devices (3)

Created usb_device_id structures are declared with the MODULE_DEVICE_TABLE() macro as in the below example:

```
/* Example from drivers/usb/net/catc.c */
static struct usb_device_id catc_id_table [] = {
    { USB_DEVICE(0x0423, 0xa) }, /* CATC Netmate, Belkin F5U011 */
    { USB_DEVICE(0x0423, 0xc) }, /* CATC Netmate II, Belkin F5U111
*/
    { USB_DEVICE(0x08d1, 0x1) }, /* smartBridges smartNIC */
    { }                           /* Terminating entry */
};

MODULE_DEVICE_TABLE(usb, catc_id_table);
```

Note that MODULE_DEVICE_TABLE() is also used with other subsystems: `pci`, `pcmcia`, `serio`, `isapnp`, `input`...

Supported devices - Summary

- Drivers need to announce the devices they support in `usb_device_id` structures.
- Needed for user space to know which module to (un)load, and for the kernel which driver code to execute, when a device is inserted or removed.
- Most drivers use `USB_DEVICE()` to create the structures.
- These structures are then registered with `MODULE_DEVICE_TABLE(usb, xxx)`.

Linux USB drivers

Writing USB drivers

Registering a USB driver

The `usb_driver` structure

USB drivers must define a usb_driver structure:

- `const char *name`
  Unique driver name. Usually be set to the module name.
- `const struct usb_device_id *id_table;`
  The table already declared with MODULE_DEVICE_TABLE().
- ```
  int (*probe) (struct usb_interface *intf,
                const struct usb_device_id *id);
  ```
  Probe callback (detailed later).
- ```
  void (*disconnect) (struct usb_interface
  *intf);
  ```
  Disconnect callback (detailed later).

Optional `usb_driver` structure fields

- ```
  int (*suspend) (struct usb_interface *intf,
                  pm_message_t message);
  int (*resume) (struct usb_interface *intf);
  ```
  Power management: callbacks called before and after the USB core suspends and resumes the device.
- ```
  void (*pre_reset) (struct usb_interface *intf);
  void (*post_reset) (struct usb_interface
  *intf);
  ```
  Called by `usb_reset_composite_device()` before and after it performs a USB port reset.

Driver registration

Use usb_register() to register your driver. Example:

```
/* Example from drivers/usb/input/mtouchusb.c */
static struct usb_driver mtouchusb_driver = {
        .name           = "mtouchusb",
        .probe          = mtouchusb_probe,
        .disconnect     = mtouchusb_disconnect,
        .id_table       = mtouchusb_devices,
};

static int __init mtouchusb_init(void)
{
        dbg("%s - called", __FUNCTION__);
        return usb_register(&mtouchusb_driver);
}
```

Driver unregistration

Use usb_deregister() to register your driver. Example:

```
/* Example from drivers/usb/input/mtouchusb.c */
static void __exit mtouchusb_cleanup(void)
{
        dbg("%s - called", __FUNCTION__);
        usb_deregister(&mtouchusb_driver);
}
```

probe() and disconnect() functions

- The `probe()` function is called by the USB core to see if the driver is willing to manage a particular interface on a device.
- The driver should then make checks on the information passed to it about the device.
- If it decides to manage the interface, the `probe()` function will return `0`. Otherwise, it will return a negative value.
- The `disconnect()` function is called by the USB core when a driver should no longer control the device (even if the driver is still loaded), and should do some clean-up.

Context: USB hub kernel thread

- The `probe()` and `disconnect()` callbacks are called in the context of the USB hub kernel thread.
- So, it is legal to call functions which may sleep in these functions.
- However, all addition and removal of devices is managed by this single thread.
- Most of the probe function work should indeed be done when the device is actually opened by a user. This way, this doesn't impact the performance of the kernel thread in managing other devices.

probe() function work

- In this function the driver should initialize local structures which it may need to manage the device.
- In particular, it can take advantage of information it is given about the device.
- For example, drivers usually need to detect endpoint addresses and buffer sizes.

Time to show and explain examples in detail!

usb_set_intfdata() / usb_get_intfdata()

```
static inline void usb_set_intfdata (
    struct usb_interface *intf,
    void *data);
```

- Function used in `probe()` functions to attach collected device data to an interface. Any pointer will do!
- Useful to store information for each device supported by a driver, without having to keep a static data array.
- The `usb_get_intfdata()` function is typically used in the device open functions to retrieve the data.
- Stored data need to be freed in `disconnect()` functions:
  `usb_set_intfdata(interface, NULL);`

Plenty of examples are available in the kernel sources.

Linux USB drivers

Writing USB drivers

USB transfers without URBs

Transfers without URBs

The kernel provides two `usb_bulk_msg()` and `usb_control_msg()` helper functions that make it possible to transfer simple bulk and control messages, without having to:

- Create or reuse an urb structure,
- Initialize it,
- Submit it,
- And wait for its completion handler.

Transfers without URBs - constraints

- These functions are synchronous and will make your code sleep. You must not call them from interrupt context or with a spinlock held.
- You cannot cancel your requests, as you have no handle on the URB used internally. Make sure your `disconnect()` function can wait for these functions to complete.

See the kernel sources for examples using these functions!

USB device drivers - Summary

Module loading

- Declare supported devices (interfaces).
- Bind them to `probe()` and `disconnect()` functions.

Supported devices are found

- probe() functions for matching interface drivers are called.
- They record interface information and register resources or services.

Devices are opened

- This calls data access functions registered by the driver.
- URBs are initialized.
- Once the transfers are over, completion functions are called. Data are copied from/to user-space.

Devices are removed

- The `disconnect()` functions are called.
- The drivers may be unloaded.

Advice for embedded system developers

If you need to develop a USB device driver for an embedded Linux system.

- Develop your driver on your GNU/Linux development host!
- The driver will run with no change on the target Linux system (provided you wrote portable code!): all USB device drivers are platform independent.
- Your driver will be much easier to develop on the host, because of its flexibility and the availability of debugging and development tools.

PCI Driver

Some background on PCI

ISA: Industry Standard Architecture (1981)

PCI: Peripheral Component Interconnect

An Intel-backed industry initiative (1992-9)

Main goals:

- Improve data-xfers to/from peripheral devices
- Eliminate (or reduce) platform dependencies
- Simplify adding/removing peripheral devices
- Lower total consumption of electrical power

Some background on PCI

The PCI architecture was designed as a replacement for the ISA standard, with three main goals:

- to get better performance when transferring data between the computer and its peripherals
- to be as platform independent as possible
- and to simplify adding and removing peripherals to the system.

Some background on PCI

The PCI bus achieves better performance by using a higher clock rate than ISA; its clock runs at 25 or 33 MHz (its actual rate being a factor of the system clock), and 66-MHz and even 133-MHz implementations have recently been deployed as well.

It is equipped with a 32-bit data bus, and a 64-bit extension has been included in the specification.

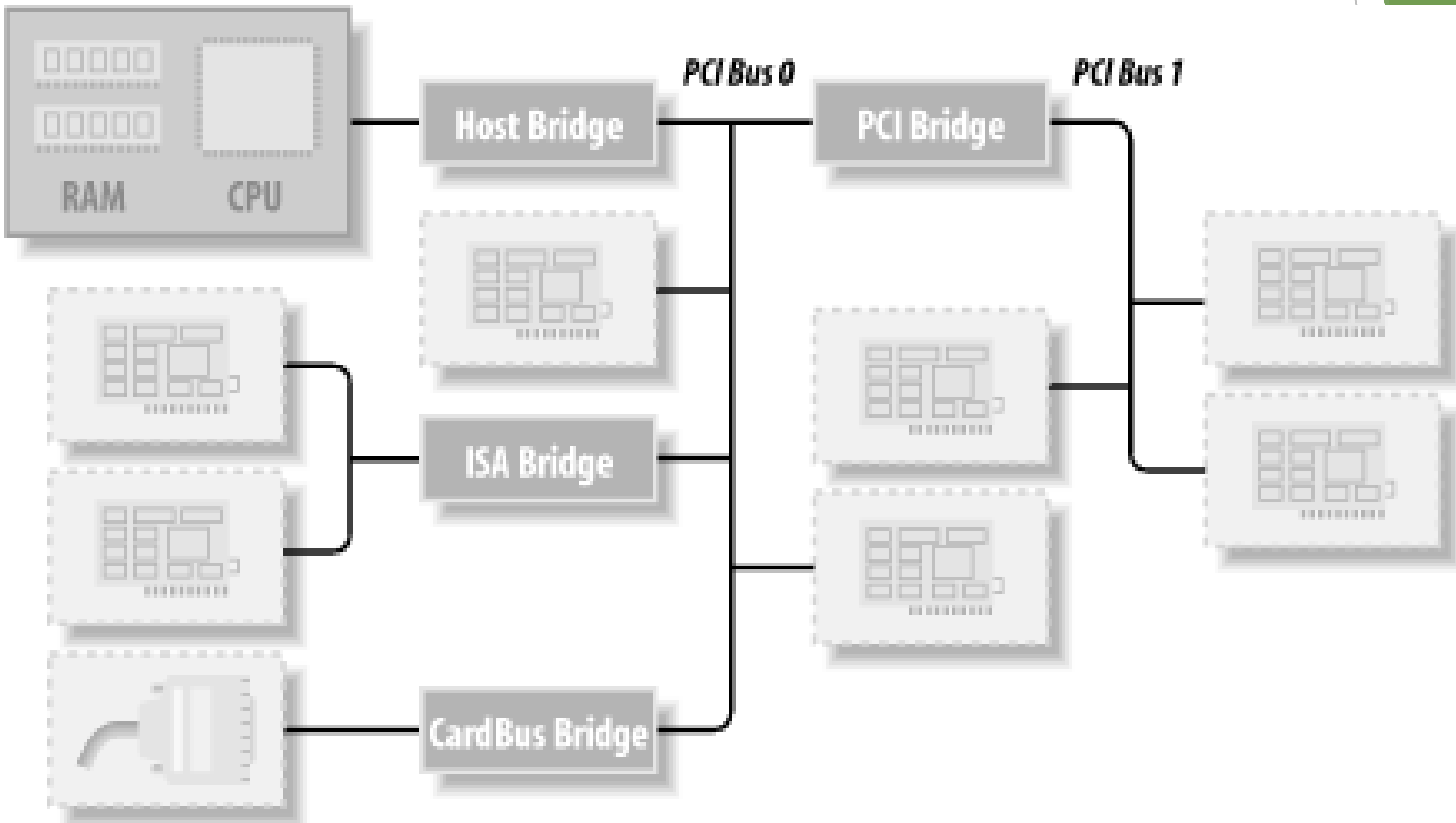
The device driver must be able to access configuration information in the device in order to complete initialization. This happens without the need to perform any probing.

PCI Addressing

Each PCI peripheral is identified by a bus number, a device number, and a function number. The PCI specification permits a single system to host up to 256 buses, but because 256 buses are not sufficient for many large systems, Linux now supports PCI domains. Each PCI domain can host up to 256 buses. Each bus hosts up to 32 devices, and each device can be a multifunction board (such as an audio device with an accompanying CD-ROM drive) with a maximum of eight functions.

When the hardware address is displayed, it can be shown as two values (an 8-bit bus number and an 8-bit device and function number), as three values (bus, device, and function), or as four values (domain, bus, device, and function); all the values are usually displayed in hexadecimal.

PCI Addressing



The ‘lspci’ command

Linux scans PCI Configuration Space

It builds a list of ‘pci_dev_struct’ objects

It exports partial info using a ‘/proc’ file

You can view this info using a command: $ /sbin/lspci

Or you can directly view the /proc/pci file:

$ cat /proc/pci

1. Default Usage

By default it will display all the device information as shown below. The first field is the slot information in this format: [domain:]bus:device.function

In this example, since all the domain are 0, lspci will not display the domain.

```
# lspci

00:00.0 Host bridge: Intel Corporation 5500 I/O Hub to ESI Port (rev 13)

00:01.0 PCI bridge: Intel Corporation 5520/5500/X58 I/O Hub PCI Express Root Port 1 (rev 13)

00:09.0 PCI bridge: Intel Corporation 7500/5520/5500/X58 I/O Hub PCI Express Root Port 9 (rev 13)

00:14.0 PIC: Intel Corporation 7500/5520/5500/X58 I/O Hub System Management Registers (rev 13)

00:14.1 PIC: Intel Corporation 7500/5520/5500/X58 I/O Hub GPIO and Scratch Pad Registers (rev 13)

00:14.2 PIC: Intel Corporation 7500/5520/5500/X58 I/O Hub Control Status and RAS Registers (rev 13)

00:1a.0 USB controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #4 (rev 02)

00:1c.0 PCI bridge: Intel Corporation 82801I (ICH9 Family) PCI Express Port 1 (rev 02)

00:1d.0 USB controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #1 (rev 02)

00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 92)

00:1f.0 ISA bridge: Intel Corporation 82801IB (ICH9) LPC Interface Controller (rev 02)

00:1f.2 IDE interface: Intel Corporation 82801IB (ICH9) 2 port SATA Controller [IDE mode] (rev 02)

01:00.0 Ethernet controller: Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet (rev 20)

01:00.1 Ethernet controller: Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet (rev 20)

03:00.0 RAID bus controller: LSI Logic / Symbios Logic MegaRAID SAS 2108 [Liberator] (rev 05)

06:03.0 VGA compatible controller: Matrox Electronics Systems Ltd. MGA G200eW WPCM450 (rev 0a)
```

Dump PCI Info in Different Format

If you want to pass the output of the lspci command to a shell script, you may want to use -m option (or -mm option) as shown below.

This option is also helpful when you want to view the subsystem information. For example, for the RAID controller, the default output just says that is is using LSI Logic RAID controller. But, the following output displays the subsystem, which is DELL PERC H700 Integrated RAID controller system.

```
# lspci -m

00:00.0 "Host bridge" "Intel Corporation" "5500 I/O Hub to ESI Port" -r13 "Dell"
"PowerEdge R610 I/O Hub to ESI Port"

00:09.0 "PCI bridge" "Intel Corporation" "7500/5520/5500/X58 I/O Hub PCI Express Root
Port 9" -r13 "" ""

00:14.0 "PIC" "Intel Corporation" "7500/5520/5500/X58 I/O Hub System Management
Registers" -r13 "" ""

00:1a.0 "USB controller" "Intel Corporation" "82801I (ICH9 Family) USB UHCI
Controller #4" -r02 "Dell" "PowerEdge R610 USB UHCI Controller"

00:1f.0 "ISA bridge" "Intel Corporation" "82801IB (ICH9) LPC Interface Controller" -
r02 "Dell" "PowerEdge R610 82801IB (ICH9) LPC Interface Controller"

00:1f.2 "IDE interface" "Intel Corporation" "82801IB (ICH9) 2 port SATA Controller
[IDE mode]" -r02 -p8f "Dell" "PowerEdge R610 SATA IDE Controller"

01:00.0 "Ethernet controller" "Broadcom Corporation" "NetXtreme II BCM5709 Gigabit
Ethernet" -r20 "Dell" "PowerEdge R610 BCM5709 Gigabit Ethernet"

03:00.0 "RAID bus controller" "LSI Logic / Symbios Logic" "MegaRAID SAS 2108
[Liberator]" -r05 "Dell" "PERC H700 Integrated"

06:03.0 "VGA compatible controller" "Matrox Electronics Systems Ltd." "MGA G200eW
WPCM450" -r0a "Dell" "PowerEdge R610 MGA G200eW WPCM450"
```

Output in Tree Format

The -t option will display the output in tree format with information about bus, and how devices are connected to those buses as shown below. The output will be only using the numerical ids.

```
# lspci -t

-[0000:00]-+-00.0

           +-01.0-[01]--+-00.0

           |            \-00.1

           +-03.0-[02]--+-00.0

           |            \-00.1

           +-07.0-[04]--

           +-09.0-[05]--

           +-14.0

           +-14.1

           +-1c.0-[03]----00.0

           +-1d.0

           +-1e.0-[06]----03.0

           +-1f.0
```

Detailed Device Information

If you want to look into details of a particular device, use -v to get more information. This will display information about all the devices. The output of this command will be very long, and you need to scroll down and view the appropriate section.

For additional level for verbosity, you can use -vv or -vvv.

In the following example, I've given output of only the RAID controller device.

```
# lspci -v

 03:00.0 RAID bus controller: LSI Logic / Symbios Logic MegaRAID SAS 2108 [Liberator] (rev 05)

        Subsystem: Dell PERC H700 Integrated

        Flags: bus master, fast devsel, latency 0, IRQ 16

        I/O ports at fc00 [size=256]

        Memory at df1bc000 (64-bit, non-prefetchable) [size=16K]

        Memory at df1c0000 (64-bit, non-prefetchable) [size=256K]

        Expansion ROM at df100000 [disabled] [size=256K]

        Capabilities: [50] Power Management version 3
```

Display Device Codes in the Output

If you want to display the PCI vendor code, and the device code only as the numbers, use -n option. This will not lookup the PCI file to get the corresponding values for the numbers.

```
# lspci -n

01:00.1 0200: 14e4:1639 (rev 20)

02:00.0 0200: 14e4:1639 (rev 20)

02:00.1 0200: 14e4:1639 (rev 20)

03:00.0 0104: 1000:0079 (rev 05)

06:03.0 0300: 102b:0532 (rev 0a)
```

If you want to display both the description and the number, use the option -nn as shown below.

```
# lspci -nn
```

Lookup a Specific Device

When you know the slot number in the domain:bus:slot.func format, you can query for a particular device as shown below. In the following example, we didn't specify the domain number, as it is 0, which can be left out.

```
# lspci -s 03:00.0

03:00.0 RAID bus controller: LSI Logic / Symbios Logic MegaRAID SAS 2108 [Liberator]
(rev 05)
```

When you know the device number in the vendor:device format, you can query for a particular device as shown below.

```
# lspci -d 1000:0079

03:00.0 RAID bus controller: LSI Logic / Symbios Logic MegaRAID SAS 2108 [Liberator]
(rev 05)
```

If you know only either the vendor id, or the device id, you can omit the other id. For example, both the following command will return the same output as the above.

Display Kernel Drivers

This is very helpful when you like to know the name of the kernel module that will be handling the operations of a particular device. Please note that this option will work only on Kernel 2.6 version and above.

```
# lspci -k

00:1f.2 IDE interface: Intel Corporation 82801IB (ICH9) 2 port SATA Controller [IDE mode] (rev 02)

        Subsystem: Dell PowerEdge R610 SATA IDE Controller

        Kernel driver in use: ata_piix

        Kernel modules: ata_generic, pata_acpi, ata_piix

02:00.0 Ethernet controller: Broadcom Corporation NetXtreme II BCM5709 Gigabit Ethernet (rev 20)

        Subsystem: Dell PowerEdge R610 BCM5709 Gigabit Ethernet

        Kernel driver in use: bnx2
```

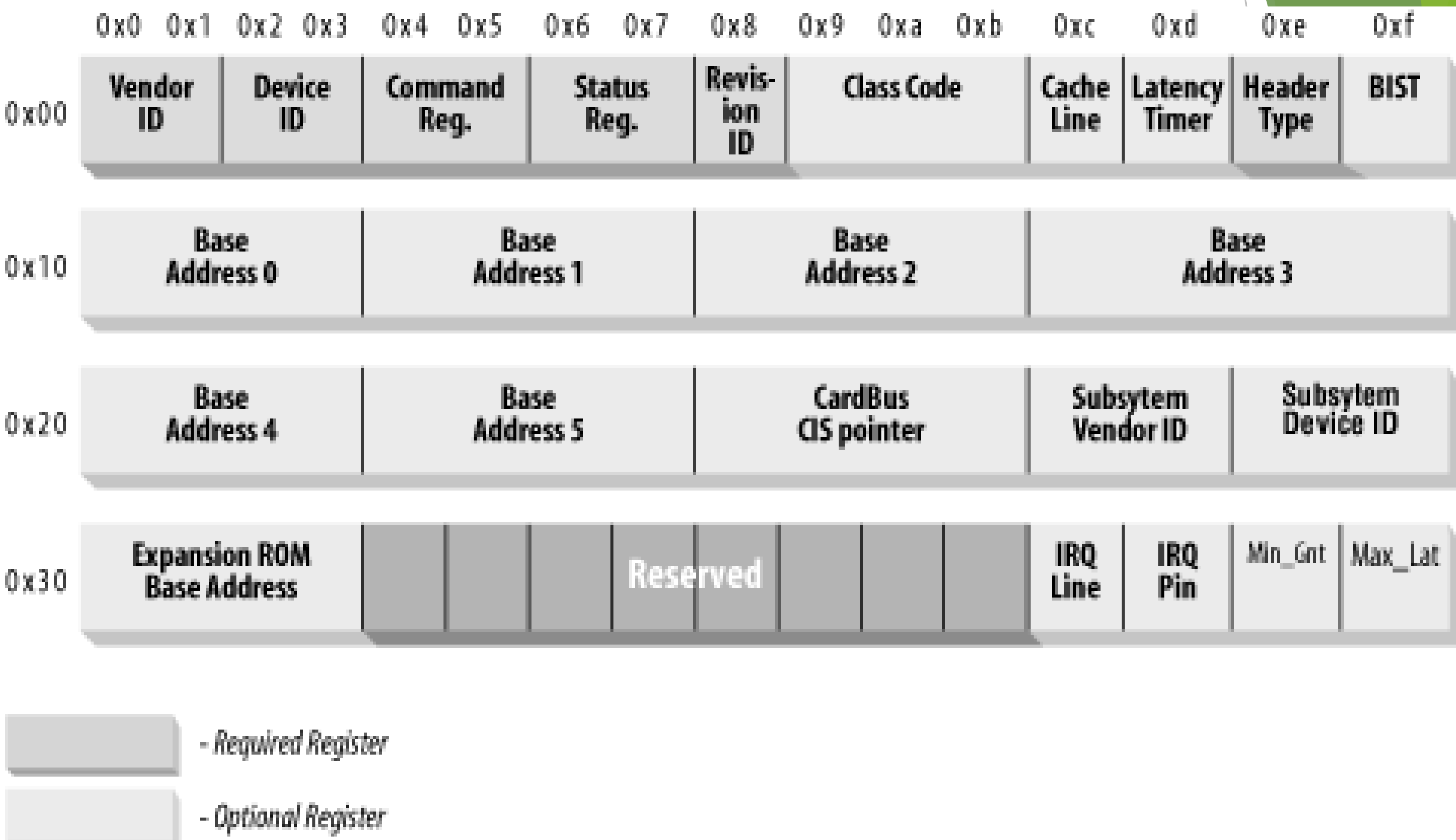
PCI Configuration Space

PCI devices have a set of registers referred to as *configuration space* and PCI Express introduces *extended configuration space* for devices. Configuration space registers are mapped to memory locations. Device drivers and diagnostic software must have access to the configuration space, and operating systems typically use APIs to allow access to device configuration space. When the operating system does not have access methods defined or APIs for memory mapped configuration space requests, the driver or diagnostic software has the burden to access the configuration space in a manner that is compatible with the operating system's underlying access rules. In all systems, device drivers are encouraged to use APIs provided by the operating system to access the configuration space of the device.

PCI Configuration Space

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Cap. Pointer	34h
Reserved				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

PCI Configuration Space



Features for driver-writers

Support for “auto-detection” of devices

Device configuration is “programmable”

Introduces “PCI Configuration Space”

A nonvolatile data-structure of device info

A standard “header” layout: 64 longwords

Linux provides some interface functions:

```
#include <linux/pci.h>
```

struct pci_device_id

The struct pci_device_id structure is used to define a list of the different types of PCI devices that a driver supports. This structure contains the following fields:

_ _u32 vendor;

_ _u32 device;

These specify the PCI vendor and device IDs of a device. If a driver can handle any vendor or device ID, the value PCI_ANY_ID should be used for these fields.

_ _u32 subvendor;

_ _u32 subdevice;

These specify the PCI subsystem vendor and subsystem device IDs of a device. If a driver can handle any type of subsystem ID, the value PCI_ANY_ID should be used for these fields.

struct pci_device_id

_ _u32 class;

_ _u32 class_mask;

These two values allow the driver to specify that it supports a type of PCI class device. The different classes of PCI devices (a VGA controller is one example) are described in the PCI specification. If a driver can handle any type of subsystem ID, the value PCI_ANY_ID should be used for these fields.

kernel_ulong_t driver_data;

This value is not used to match a device but is used to hold information that the PCI driver can use to differentiate between different devices if it wants to.

struct pci_device_id

There are two helper macros that should be used to initialize a struct pci_device_id structure:

PCI_DEVICE(vendor, device)

This creates a struct pci_device_id that matches only the specific vendor and device ID. The macro sets the subvendor and subdevice fields of the structure to PCI_ANY_ID.

PCI_DEVICE_CLASS(device_class, device_class_mask)

This creates a struct pci_device_id that matches a specific PCI class.

struct pci_device_id

An example of using these macros to define the type of devices a driver supports can be found in the following kernel files:

```
drivers/usb/host/ehci-hcd.c:

static const struct pci_device_id pci_ids[  ] = { {
        /* handle any USB 2.0 EHCI controller */
        PCI_DEVICE_CLASS(((PCI_CLASS_SERIAL_USB << 8) | 0x20), ~0),
        .driver_data =  (unsigned long) &ehci_driver,
        },
        { /* end: all zeroes */ }
};

drivers/i2c/busses/i2c-i810.c:

static struct pci_device_id i810_ids[  ] = {
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG1) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG3) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810E_IG) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82815_CGC) },
    { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82845G_IG) },
    { 0, },
};
```

MODULE_DEVICE_TABLE

This pci_device_id structure needs to be exported to user space to allow the hotplug and module loading systems know what module works with what hardware devices. The macro MODULE_DEVICE_TABLE accomplishes this. An example is:

MODULE_DEVICE_TABLE(pci, i810_ids);

Registering a PCI Driver

The main structure that all PCI drivers must create in order to be registered with the kernel properly is the struct pci_driver structure. This structure consists of a number of function callbacks and variables that describe the PCI driver to the PCI core. Here are the fields in this structure that a PCI driver needs to be aware of:

const char *name;

The name of the driver. It must be unique among all PCI drivers in the kernel and is normally set to the same name as the module name of the driver. It shows up in sysfs under /sys/bus/pci/drivers/ when the driver is in the kernel.

const struct pci_device_id *id_table;

Pointer to the struct pci_device_id table

Registering a PCI Driver

int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);

Pointer to the probe function in the PCI driver. This function is called by the PCI core when it has a struct pci_dev that it thinks this driver wants to control. A pointer to the struct pci_device_id that the PCI core used to make this decision is also passed to this function. If the PCI driver claims the struct pci_dev that is passed to it, it should initialize the device properly and return 0. If the driver does not want to claim the device, or an error occurs, it should return a negative error value. More details about this function follow later in this chapter.

void (*remove) (struct pci_dev *dev);

Pointer to the function that the PCI core calls when the struct pci_dev is being removed from the system, or when the PCI driver is being unloaded from the kernel. More details about this function follow later in this chapter.

Registering a PCI Driver

int (*suspend) (struct pci_dev *dev, u32 state);

Pointer to the function that the PCI core calls when the struct pci_dev is being suspended. The suspend state is passed in the state variable. This function is optional; a driver does not have to provide it.

int (*resume) (struct pci_dev *dev);

Pointer to the function that the PCI core calls when the struct pci_dev is being resumed. It is always called after suspend has been called. This function is optional; a driver does not have to provide it.

Registering a PCI Driver

In summary, to create a proper `struct pci_driver` structure, only four fields need to be initialized:

```
static struct pci_driver pci_driver = {
    .name = "pci_skel",
    .id_table = ids,
    .probe = probe,
    .remove = remove,
};
```

To register the `struct  pci_driver` with the PCI core, a call to *pci_register_driver* is made with a pointer to the `struct pci_driver`. This is traditionally done in the module initialization code for the PCI driver:

```
static int _ _init pci_skel_init(void)
{
    return pci_register_driver(&pci_driver);
}
```

UnRegistering a PCI Driver

When the PCI driver is to be unloaded, the struct pci_driver needs to be unregistered from the kernel. This is done with a call to pci_unregister_driver. When this call happens, any PCI devices that were currently bound to this driver are removed, and the remove function for this PCI driver is called before the pci_unregister_driver function returns.

```
static void __exit pci_skel_exit(void)
{
    pci_unregister_driver(&pci_driver);
}
```

Enabling the PCI Device

In the probe function for the PCI driver, before the driver can access any device resource (I/O region or interrupt) of the PCI device, the driver must call the pci_enable_device function:

int pci_enable_device(struct pci_dev *dev);

This function actually enables the device. It wakes up the device and in some cases also assigns its interrupt line and I/O regions. This happens, for example, with CardBus devices (which have been made completely equivalent to PCI at the driver level).

Accessing the Configuration Space

After the driver has detected the device, it usually needs to read from or write to the three address spaces: memory, port, and configuration. In particular, accessing the configuration space is vital to the driver, because it is the only way it can find out where the device is mapped in memory and in the I/O space.

As far as the driver is concerned, the configuration space can be accessed through 8-bit, 16-bit, or 32-bit data transfers. The relevant functions are prototyped in <linux/pci.h>:

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
```

Accessing the Configuration Space

int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);

int pci_write_config_word(struct pci_dev *dev, int where, u16 val);

int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);

Accessing the Configuration Space

```
int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

Accessing the I/O and Memory Spaces

The preferred interface for getting region information consists of the following functions:

unsigned long pci_resource_start(struct pci_dev *dev, int bar);

The function returns the first address (memory address or I/O port number) associated with one of the six PCI I/O regions. The region is selected by the integer bar (the base address register), ranging from 0-5 (inclusive).

unsigned long pci_resource_end(struct pci_dev *dev, int bar);

The function returns the last address that is part of the I/O region number bar. Note that this is the last usable address, not the first address after the region.

Accessing the I/O and Memory Spaces

unsigned long pci_resource_flags(struct pci_dev *dev, int bar);

This function returns the flags associated with this resource.

All resource flags are defined in <linux/ioport.h>; the most important are:

IORESOURCE_IO

IORESOURCE_MEM

If the associated I/O region exists, one and only one of these flags is set.

IORESOURCE_PREFETCH

IORESOURCE_READONLY

These flags tell whether a memory region is prefetchable and/or write protected. The latter flag is never set for PCI resources.

PCI Interrupt

```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result) {
    /* deal with error */
}
```