USB Device Drivers

Michael Mitchell

USB Device Basics

- Universal Serial Bus (USB) connects between a computer and peripheral devices
  - Created to replace various slow buses (parallel, serial, and keyboard connections)
    - USB 2.0: up to 480Mb/s (35 MB/s)
    - USB 3.0: up to 6Gb/s (625 MB/s)

USB Device Basics

- A USB device can never start sending data without first being asked by the host controller
- Single-master implementation
  - Host polls various devices
  - A device can request a fixed bandwidth (for audio and video I/O)
- Universal Serial *Bus* is a misnomer…
  - Actually a tree built out of point-to-point links
    - Links are four-wire cables (ground, power, and two signal wires)

USB Device Basics – The Protocol

- USB protocol defines a set of standards that any device can follow
  - No need to write a driver for a device that is in a predefined class and follows that standard,
  - Predefined classes: storage devices, keyboards, mice, joysticks, network devices, and modems
  - No defined standard for video devices and USB-to-serial devices
    - A driver is needed for every device

USB Device Basics – Driver Types

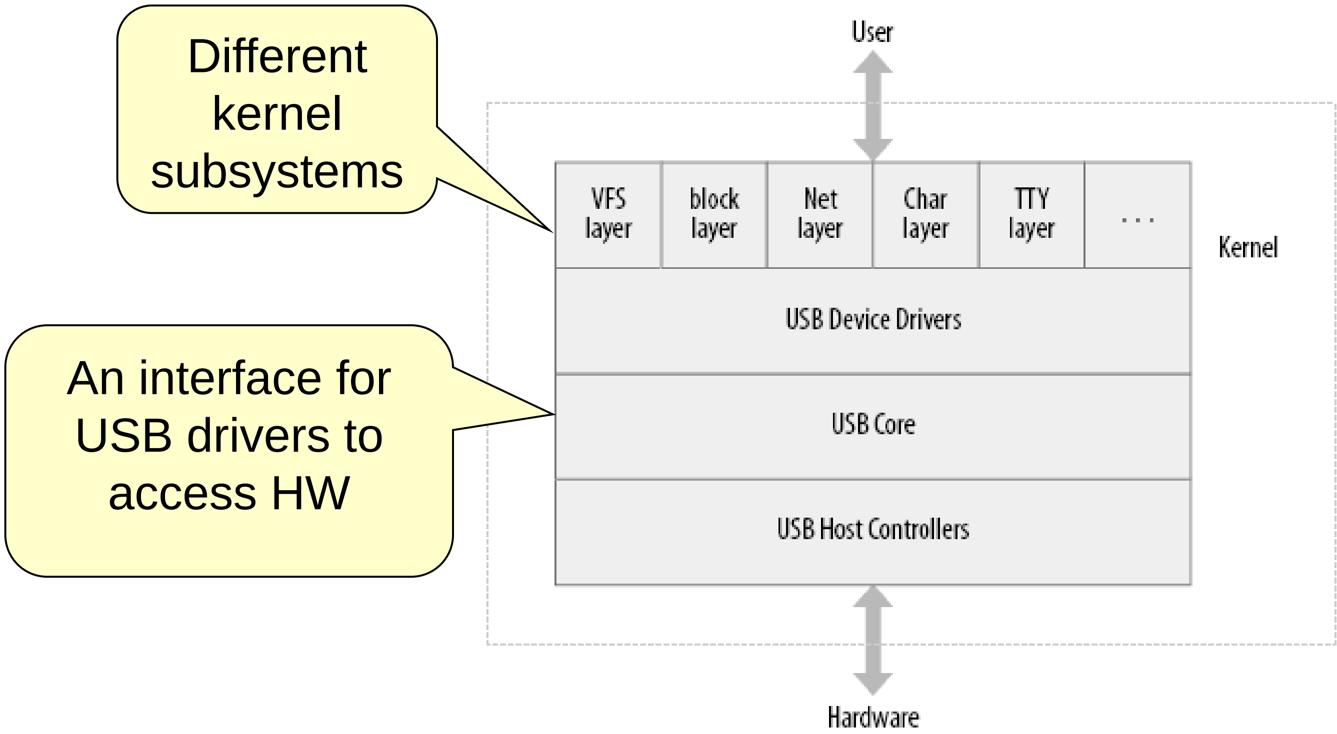
- Linux supports two types of USB drivers
  - Drivers on a host system
    - Control the USB devices that are plugged into it
  - Drivers on a device (USB gadget drivers)
    - Control how that single device looks to the host computer as a USB device
  - Some hardware devices can actually be both
    - Called USB OTG (On The Go),
    - E.g. Android 3.0+, some printers

USB Device Information

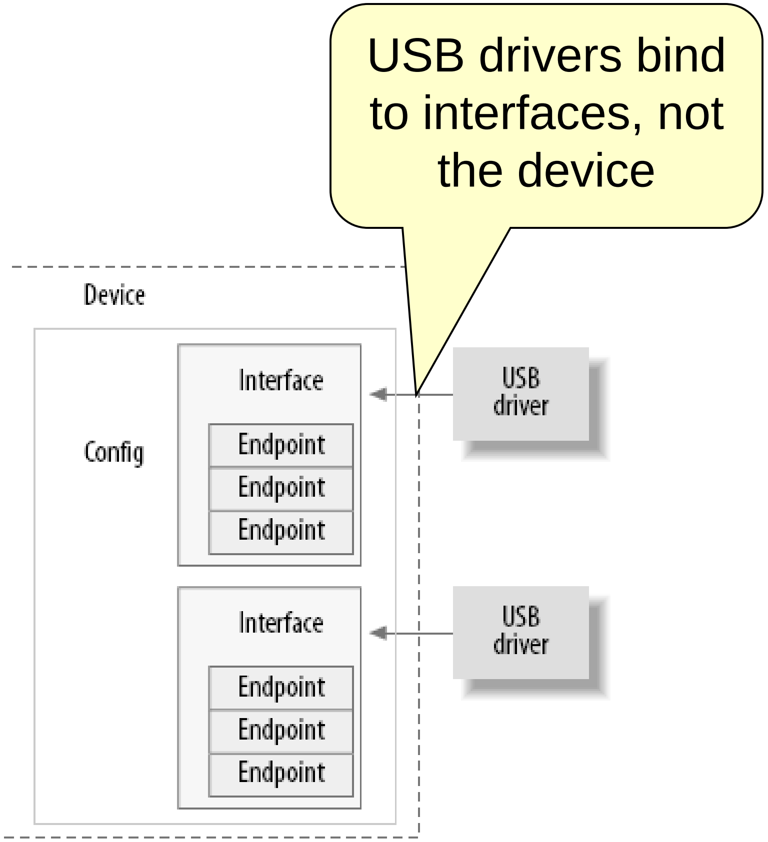
- View basic information about internally and externally connected USB hubs and devices using **`lsusb`** command

- More advanced usage covered later

USB Device Basics



USB Device Basics



USB Overview

- A USB device has one or more configurations
  - E.g., power and bandwidth requirements
- A configuration has one or more interfaces
  - E.g., audio data, knobs for speakers
- An interface has one or more settings
  - Different quality of services
    - E.g., different frame sizes for digital cameras
  - Also zero or more endpoints
    - E.g., bulk, interrupt endpoints.

Endpoints

- The most basic form of USB communication is through an *endpoint*
  - Unidirectional: Carries data in one direction
    - From the host to device (OUT endpoint)
    - From the device to the host (IN endpoint)

Endpoints

- Four endpoint types
  - CONTROL
  - INTERRUPT
  - BULK
  - ISOCHRONOUS

Endpoints

- CONTROL
  - Used for configuring the device, retrieving information and status about the device, or sending commands to the device
  - Every device has a control endpoint called endpoint 0
    - Used by USB core to configure the device at insertion time
    - Transfers are guaranteed with reserved bandwidth

Endpoints

- INTERRUPT
  - Transfer small amounts of data at a fixed rate
  - For USB keyboards and mice
  - Also used to control the device
  - Not for large transfers
  - Guaranteed reserved bandwidth

Endpoints

- BULK
  - Transfer large amounts of data
  - No data loss
  - Not time guaranteed
  - A BULK packet might be split up across multiple transfers
  - Used for printers, storage, and network devices

Endpoints

- ISOCHRONOUS
  - Transfer large amount of data
  - For real-time data collections, A/V devices
  - Unlike bulk endpoints, no guarantees (potential data loss)
- Control and bulk endpoints are used for asynchronous data transfers
- Interrupt and isochronous endpoints are periodic with reserved bandwidth

Endpoints

- Endpoint information is in **`struct usb_endpoint_descriptor`**
  - embedded in **`struct usb_host_endpoint`**
  - Note: defined by the USB standard, so not Linux looking
- Some important fields
  - **`bEndpointAddress`** (8-bit)
    - Use **`USB_DIR_OUT`** and **`USB_DIR_IN`** bit masks to determine the direction of data flow

Endpoints

- **bmAttributes**
  - Type of the endpoint
  - **& USB_ENDPOINT_XFERTYPE_MASK** to determine if the endpoint is of type **USB_ENDPOINT_XFER_ISOC**, **USB_ENDPOINT_XFER_BULK**, or **USB_ENDPOINT_XFER_INT**
- **wMaxPacketSize**
  - Maximum bytes that an endpoint can handle
  - Larger transfers will be split into multiple transfers

Endpoints

- **`bInterval`**
  - For interrupt endpoints, this value specifies the milliseconds between interrupt requests for the endpoint

Interfaces

- USB endpoints are bundled into *interfaces*
  - A interface handles only one type of logical connection (E.g., a mouse)
  - Some devices have multiple interfaces
    - E.g., a speaker
      - One interface for buttons and one for audio stream
- USB interface may have alternate settings
  - E.g., different settings to reserve different amounts of bandwidth for the device

Interfaces

- Described via **`struct usb_interface`**
  - Passed from USB core to USB drivers
- Some important fields
  - **`struct usb_host_interface *altsetting`**
    - An array of settings for this interface
  - **`unsigned num_altsetting`**
    - Number of alternative settings

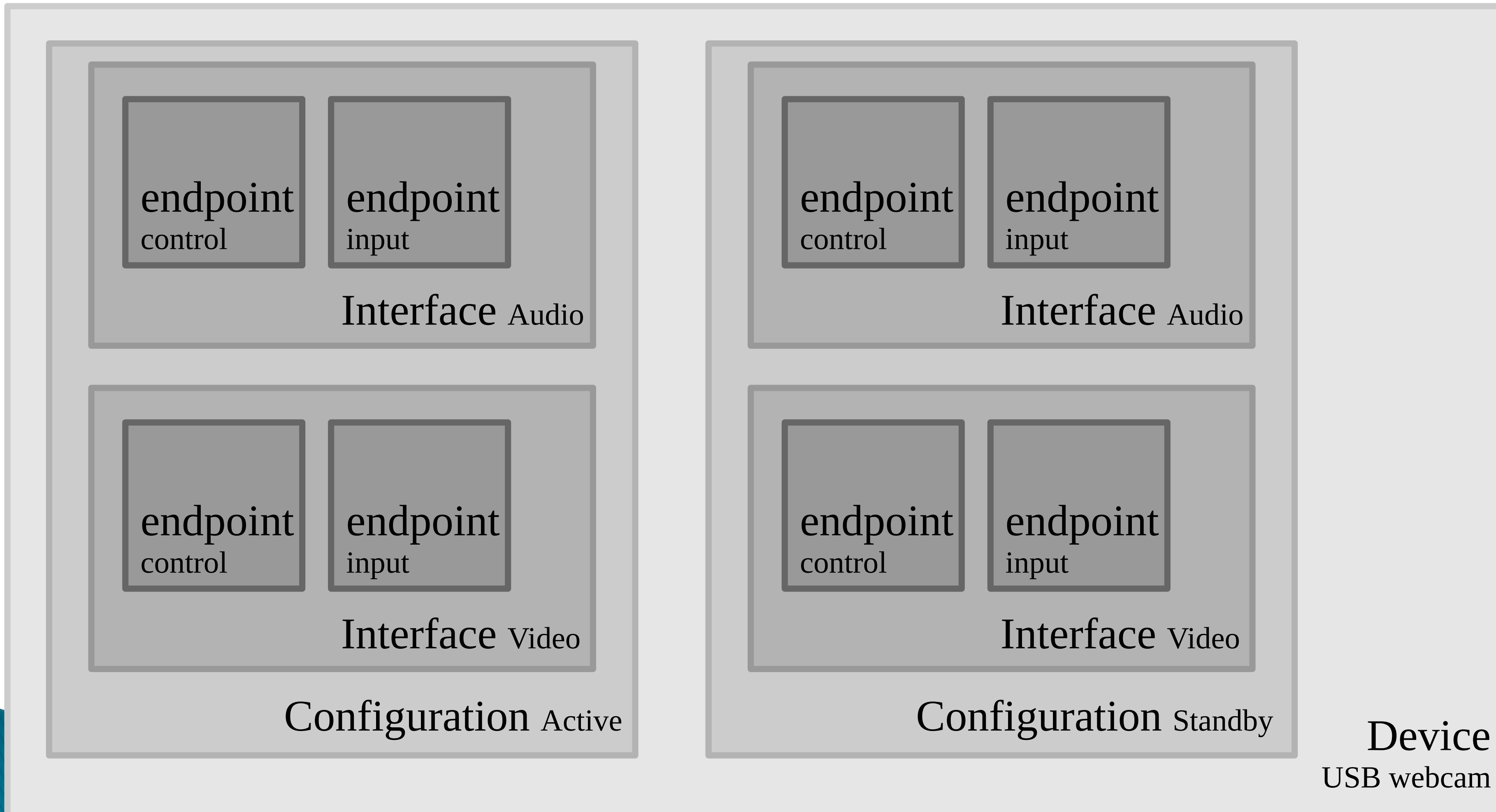
Interfaces

- **`struct usb_host_interface *cur_altsetting`**
  - A pointer into the **`altsetting`** array, denoting the current setting
- **`int minor`**
  - Minor number assigned by the USB core to the interface
  - Valid after a successful call to **`usb_register_dev`**

Configurations

- USB interfaces are bundled into *configurations*
- A USB device can have multiple configurations
  - Only one can be active at a time
  - Can switch between them
- Described in **`struct usb_host_config`**
  - embedded in **`struct usb_device`**

USB Webcam Device Example



USB and Sysfs

- Both USB devices and its interfaces are shown in **`sysfs`** as individual devices
- A USB mouse device can be represented as

  `/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1`
- The interface of the USB mouse device driver is represented as

  `/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0`

`root_hub-hub_port:configuration.interface`

USB and Sysfs

- For a two-level USB connection, the device name is in the following format
  `root_hub-hub_port-hub_port:configuration.interface`
- In the **`sysfs`** directory, all USB information is available
  - E.g., **`idVendor`**, **`idProduct`**, **`bMaxPower`**
  - **`bConfigurationValue`** can be written to change the active configuration

USB and Sysfs

- More information is available in ~~**/proc/bus/usb** directory~~

  /sys/kernel/debug/usb/devices
- User-space programs can directly communicate with USB devices via the directory
- Also verbose output from lsusb: **lsusb -v**

USB Urbs (USB Request Block)

- Communication between the host and device is done asynchronously using USB Request Blocks (URBs).
  - Similar to packets in network communications.
  - Every endpoint can handle a queue of URBs.
  - Every URB has a completion handler.
  - Flexible: 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.

USB Urbs (USB Request Block)

- **`struct urb`**
  - Used to send and receive data between endpoints
  - Asynchronous
  - Dynamically created
    - Contains reference count for garbage collection
  - Defined in **`<include/linux/usb.h>`**
  - Must be created with the usb_alloc_urb() function. Shouldn't be allocated statically or with kmalloc().
  - Must be deleted with usb_free_urb().

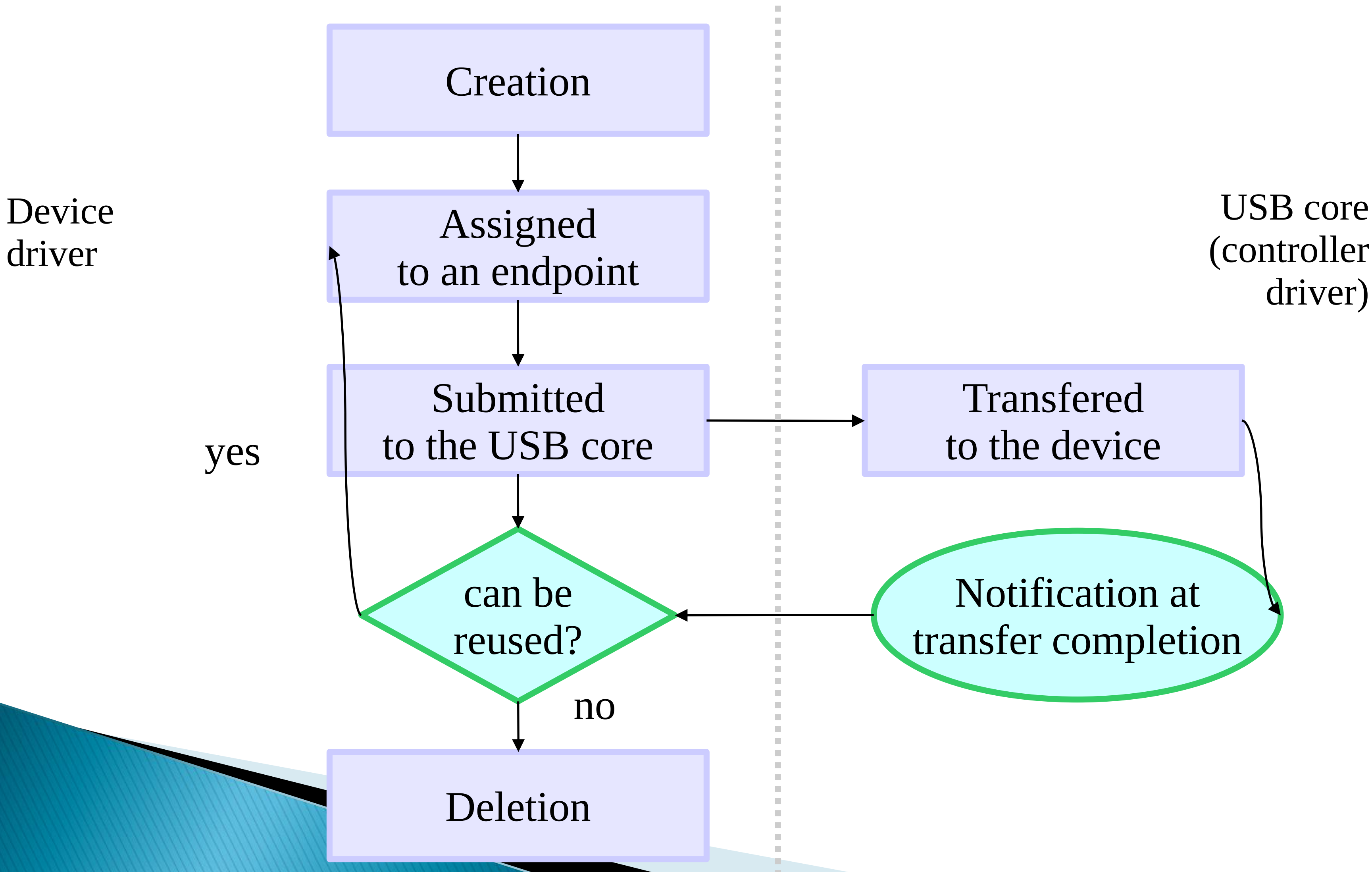
USB Urbs (USB Request Block)

- A typical lifecycle of an Urb
  - A USB device driver creates an Urb
    - Assigns it to a specific endpoint of a device
    - Submits it to the USB core
  - The USB core submits the Urb to specific USB host controller driver
  - The USB host controller driver processes the Urb and transfers it to the device
    - Notifies the USB device driver when the Urb is done

USB Urbs (USB Request Block)

- An Urb can be cancelled by the driver or the USB core if the device is removed from the system

Life Cycle of an Urb



struct urb

- Important fields

```
/* destination USB device */
/* must be initialized by the USB driver before the urb can be
   sent to the USB core */
struct usb_device *dev;

/* end point type information */
/* set to the return value from one of the usb send and
   receive pipe functions */
/* must be initialized by the USB driver before the urb can be
   sent to the USB core */
unsigned int pipe;
```

struct urb

```
/* assigned to one of the transfer flags */
unsigned int transfer_flags;

void *transfer_buffer; /* points to a kmalloced buffer */
dma_addr_t transfer_dma; /* buffer for DMA transfers */

/* buffer length for either the transfer_buffer or the
   transfer_dma variable, 0 if neither buffers are used */
int transfer_buffer_length;

/* pointer to a setup packet for a control urb */
/* transferred before the data in the transfer buffer */
unsigned char *setup_packet;

/* DMA buffer for the setup packet for a control urb */
dma_addr_t setup_dma;
```

struct urb

```
/* pointer to the completion handler called by USB core */
usb_complete_t complete;

/* pointer to a data blob that can be set by the USB driver */
void *context;

/* actual length of data sent/received by the urb */
int actual_length;

/* accessed in the completion handler */
/* see status values */
int status;

/* the initial frame number for isochronous transfers */
int start_frame;
```

struct urb

```
/* polling interval for the urb */
/* valid only for interrupt or isochronous urbs */
/* for slow devices, the unit is in frames or milliseconds */
/* for other devices, the unit is in 1/8 milliseconds */
int interval;

/* the number of isochronous transfer buffers handled by this
   urb */
/* must be set by the USB driver before the urb is sent to the
   USB core */
int number_of_packets;

/* number of isochronous transfers with errors */
int error_count;
```

struct urb

```
/* allows a single urb to define a number of isochronous
   transfers at once */
struct usb_iso_packet_descriptor iso_frame_desc[0];

struct usb_iso_packet_descriptor {
  unsigned int offset; /* byte into the transfer buffer */
  unsigned int length; /* length of the transfer buffer */

  /* length of data received into the transfer buffer */
  unsigned int actual_length;
  unsigned int status;  /* see status values */
};
```

USB send and receive pipe functions

```
/* specifies a control OUT endpoint for the specified USB
   device with the specified endpoint number */
unsigned int usb_sndctrlpipe(struct usb_device *dev,
                             unsigned int endpoint);

/* specifies a control IN endpoint for the specified USB
   device with the specified endpoint number */
unsigned int usb_rcvctrlpipe(struct usb_device *dev,
                             unsigned int endpoint);

/* specifies a bulk OUT endpoint for the specified USB device
   with the specified endpoint number */
unsigned int usb_sndbulkpipe(struct usb_device *dev,
                             unsigned int endpoint);
```

USB send and receive pipe functions

```
/* specifies a bulk IN endpoint for the specified USB device
   with the specified endpoint number */
unsigned int usb_rcvbulkpipe(struct usb_device *dev,
                             unsigned int endpoint);
/* specifies a interrupt OUT endpoint for the specified USB
   device with the specified endpoint number */
unsigned int usb_sndintpipe(struct usb_device *dev,
                            unsigned int endpoint);

/* specifies a interrupt IN endpoint for the specified USB
   device with the specified endpoint number */
unsigned int usb_rcvintpipe(struct usb_device *dev,
                            unsigned int endpoint);
```

USB send and receive pipe functions

```
/* specifies a isochronous OUT endpoint for the specified USB
   device with the specified endpoint number */
unsigned int usb_sndisocpipe(struct usb_device *dev,
                             unsigned int endpoint);

/* specifies a isochronous IN endpoint for the specified USB
   device with the specified endpoint number */
unsigned int usb_rcvisocpipe(struct usb_device *dev,
                             unsigned int endpoint);
```

URB Transfer flags

- **URB_SHORT_NOT_OK**
  - Partial read should be treated as an error by the USB core
- **URB_ISO_ASAP**
  - If the driver wants the isochronous urb to be scheduled as soon as bandwidth allows
  - Set the **start_frame** variable

URB Transfer flags

- **`URB_NO_TRANSFER_DMA_MAP`**
  - Set when the urb contains a DMA buffer to be transferred
  - Tells the USB core to use the buffer pointed by the **`transfer_dma`** pointer, not the **`transfer_buffer`** pointer

URB Transfer flags

- **`URB_NO_SETUP_DMA_MAP`**
  - Used for control urbs with DMA buffer already set up
  - Tells the USB core to use the buffer pointed by the **`setup_dma`** pointer instead of the **`setup_packet`** pointer
- **`URB_ASYNC_UNLINK`**
  - Tells **`usb_unlink_urb()`** to return immediate and unlink the urb in the background

URB Transfer flags

URB_ZERO_PACKET

- Tells a bulk out urb finishes by sending an empty packet when the data is aligned to an endpoint packet boundary

- **URB_NO_INTERRUPT**
  - Indicates that the HW may not generate an interrupt when the urb is finished
  - Used when queuing multiple urbs to the same endpoint
  - Used by USB core to perform DMA transfers

URB Status Values

- **0**
  - The urb transfer was successful
  - For isochronous urbs, only indicates whether the urb has been unlinked
    - Detailed status in **iso_frame_desc**
- **-ENOENT**
  - Urb stopped by **usb_kill_urb**
- **-ECONNRESET**
  - Urb was unlinked by **usb_unlink_urb**
  - **transfer_flags** set to **URB_ASYNC_UNLINK**

URB Status Values

- **-EINPROGRESS**
  - Urb still being processed by the USB host controller
  - A bug if seen at the driver level
- **-EPROTO** (a HW problem)
  - A bitstuff error happened during the transfer
  - No response packet was received
- **-EILSEQ** (a HW problem)
  - CRC mismatch

URB Status Values

- **-EPIPE**
  - The endpoint is now stalled
  - If not a control endpoint, can clear this error with **usb_clear_halt**
- **-ECOMM**
  - Data received faster than it could be written to system memory
- **-ENOSR**
  - Data cannot be retrieved from the system memory during the transfer fast enough to keep up with the requested USB data rate

URB Status Values

- **-EOVERFLOW** (a HW problem)
  - When the endpoint receives more data than the specified max
- **-EREMOTEIO**
  - Full amount of data was not received
  - Occurs when the **URB_SHORT_NOT_OK** is set
- **-ENODEV**
  - The USB device is gone from the system

URB Status Values

- **-ESHUTDOWN**
  - Host controller driver has been disabled or disconnected
  - Urb was submitted after the device was removed
  - Configuration change while the urb was submitted

URB Status Values

- **-EXDEV**
  - Only for a isochronous urb
  - Transfer was partially completed
- **-EINVAL**
  - Incorrect function parameter
  - ISO madness, if this happens: Log off and go home

USB URB debugging

- Real-time capture of USB URBs is possible using usbmon
- **`modprobe usbmon`**
- **`# cat /sys/kernel/debug/usb/usbmon/`**

  0s 0u 1s 1t 1u 2s 2t 2u 3s 3t 3u 4s 4t 4u
- **`# cat /sys/kernel/debug/usb/usbmon/3u > /tmp/3.mon.out`**

Creating and Destroying Urbs

- All URBs need to be created dynamically
  - Or the reference count would not work
  - To create a URB, call

```
struct urb *usb_alloc_urb(int iso_packets,
                          gfp_t mem_flags);
```

  - Returns pointer to the URB or NULL on failure
  - **iso_packets**: number of isochronous packets this urb should contain
  - **mem_flags**: same as **kmalloc** flags
- To destroy a urb, call

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

Interrupt urbs

- To initialize an interrupt urb, call

```
void
usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
                 unsigned int pipe, void *transfer_buffer,
                 int buffer_length, usb_complete_t complete,
                 void *context, int interval);
```

  - **`urb`**: a pointer to the urb to be initialized
  - **`dev`**: The destination USB device
  - **`pipe`**: the destination endpoint of this urb

Interrupt urbs

- **`transfer_buffer`**: a pointer to a **`kmalloc`**ed buffer
- **`buffer_length`**: the length of the transfer buffer
- **`complete`**: pointer to the completion handler
- **`context`**: pointer to the blob, retrieved by the completion handler function
- **`interval`**: scheduling interval for this urb

Bulk urbs

- To initialize an bulk urb, call

```
void
usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
                  unsigned int pipe, void *transfer_buffer,
                  int buffer_length, usb_complete_t complete,
                  void *context);
```

  - Similar to interrupt urb initialization
    - Exception: No final interval parameter

Control urbs

- To initialize a control urb, call

```
void
usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
                     unsigned int pipe,
                     unsigned char *setup_packet,
                     void *transfer_buffer, int buffer_length,
                     usb_complete_t complete, void *context);
```

  - Similar to bulk urb initialization
    - **`setup_packet`**: points to the setup packet data
    - Also, does not set the **`transfer_flags`**

Isochronous urbs

- Have no initialization functions
- Need to be initialized by hand

```
/* from /drivers/media/video/usbvideo/konicawc.c */
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp - 1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
  urb->iso_frame_desc[j].offset = j;
  urb->iso_frame_desc[j].length = 1;
}
```

Submitting Urbs

- To send a urb to the USB core, call

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

  - **`urb`**: a pointer to the urb
  - **`mem_flags`**: same as **`kmalloc`** flags
    - GFP_KERNEL, GFP_ATOMIC, etc.
  - Should not access a submitted urb until the **`complete`** function is called

Completing Urbs: The Completion Callback Handler

- Called exactly once when the urb is completed
  - When this function is called, the USB core is finished with the urb, and control is returned to the device driver

Completing Urbs: The Completion Callback Handler

- The completion handler is called under three conditions
  - The urb is successfully sent to the device and acknowledged
  - An error has occurred
    - Check the status variable
  - The urb was unlinked (the submission was cancelled) when a device is removed from the system

Canceling Urbs

- To stop a submitted urb, call

```
int usb_kill_urb(struct urb *urb);
```

  - Used when the device is disconnected from the system

```
int usb_unlink_urb(struct urb *urb);
```

  - Tells the USB core to stop an urb
  - Returns before the urb is fully stopped
    - Useful while in an interrupt handler
  - Requires setting the **`URB_ASYNC_UNLINK`**

Actually writing a USB Driver

- Similar to a **`pci_driver`**
  - Driver registers its driver object with the USB subsystem
  - Later uses vendor and device identifiers to tell if its hardware has been installed

What Devices Does the Driver Support?

- **`struct usb_device_id`** lists supported types of USB devices
- Important fields
  - **`__u16 match_flags`**
    - Determines which fields in the structure the device should be matched against
    - Check **`include/linux/mod_devicetable.h`**
  - **`__u16 idVendor`**
  - **`__u16 idProduct`**

What Devices Does the Driver Support?

- **`__u16 bcdDevice_lo`**
- **`__u16 bcdDevice_hi`**
  - Define low and high ends of the range of the vendor-assigned product version number
  - Expressed in binary-coded decimal (BCD)
- **`__u8 bDeviceClass`**
- **`__u8 bDeviceSubClass`**
- **`__u8 bDeviceProtocol`**
  - Define the class, subclass, and protocol of the device

What Devices Does the Driver Support?

- **`__u8 bInterfaceClass`**
- **`__u8 bInterfaceSubClass`**
- **`__u8 bInterfaceProtocol`**
  - Class, subclass, and protocol of the individual interface
- **`kernel_ulong_t driver_info`**
  - Used to differentiate different devices in the probe callback function

What Devices Does the Driver Support?

- To initialize **`usb_device_id`**, use the following macros

  `USB_DEVICE(vendor, product)`

  - Creates a **`usb_device_id`** that can be used to match only the specified vendor and product IDs

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

  - Creates a **`usb_device_id`** that can be used to match only the specified vendor and product IDs within a version range

  `USB_DEVICE_INFO(class, subclass, protocol)`

  - Creates a **`usb_device_id`** that can be used to match a specific class of USB devices

What Devices Does the Driver Support?

```
USB_INTERFACE_INFO(class, subclass, protocol)
```

- Creates a **`usb_device_id`** that can be used to match a specific class of USB interfaces

- Example

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table[] = {
  { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
  { } /* Terminating entry */
};

/* allow user-space tools to figure out what devices this
   driver can control */
MODULE_DEVICE_TABLE(usb, skel_table);
```

Registering a USB Driver

- The main structure for a USB driver is **`struct usb_driver`**
- Important fields
  - **`struct module *owner`**
    - Set to **`THIS_MODULE`** to track the reference count of the module owning this driver
  - **`const char *name`**
    - Points to a unique driver name

Registering a USB Driver

- **`const struct usb_device_id *id_table`**
  - Pointer to the list of supported USB devices
  - If you want your driver always be called for every USB device, create an entry that sets only the **`driver_info`** field

```
static struct usb_device_id usb_ids[] = {
  {.driver_info = 42},
  { }
};
```

Registering a USB Driver

- `int (*probe) (struct usb_interface *intf,`
  `const struct usb_device_id *id)`
  - Called when the USB core thinks it has a **`struct usb_interface`** that this driver can handle
  - The USB driver should initialize the usb interface and return 0, or return a negative error number on failure
- `void (*disconnect) (struct usb_interface *intf)`
  - Called when the **`usb_interface`** has been removed from the system, or when the driver is being unloaded

Registering a USB Driver

- To create a **struct usb_driver**, only five fields need to be initialized

```
static struct usb_driver skel_driver = {
   .owner = THIS_MODULE,
   .name = "skeleton",
   .id_table = skel_table,
   .probe = skel_probe,
   .disconnect = skel_disconnect,
};
```

Registering a USB Driver

- To register a USB driver call **`usb_register_driver`**
- Example

```
static int __init usb_skel_init(void) {
  int result;

  /* register this driver with the USB subsystem */
  result = usb_register(&skel_driver);
  if (result)
    err("usb_register failed. Error number %d", result);
  return result;
}
```

Registering a USB Driver

- To unload a USB driver call **`usb_deregister`**
- Example

```
static void __exit usb_skel_exit(void) {
  /* deregister this driver with the USB subsystem */
   /* invokes disconnect() within usb_deregister() */
  usb_deregister(&skel_driver);
}
```

Probe and Disconnect in Detail

- Called in the context of the USB hub kernel thread
  - Sleep is allowed
  - However, should do most of the work when the device is opened by a user
    - USB core handles addition and removal of USB devices in a single thread
    - A slow device driver can slow down USB device detection

Probe and Disconnect in Detail

- Probe function should
  - Initialize local structures that it might use to manage the USB device
  - Save any information that it needs to the local structure
  - Detect endpoint address and buffer sizes
  - Example **`usb/usb-skeleton.c`**

Advanced USB logging/debugging

- For actual USB driver creation, reverse engineering often required
- Comprehensive capture, logging and debugging of all USB communications can be done using Wireshark