Aricent®

COP: USB Device Driver Advance

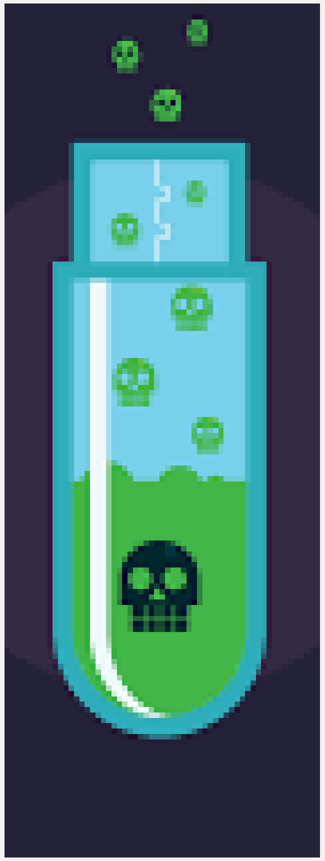
USB

By: Aricent USB Team

Ashish Agarwal

Agenda

- USB Overview
- USB Device & Driver Layouts
- USB Core & the USB protocol
- Registration & Hot-pluggability
- URB & Data Transfers
- Logging / Debugging Overview



USB Overview

USB Terminology

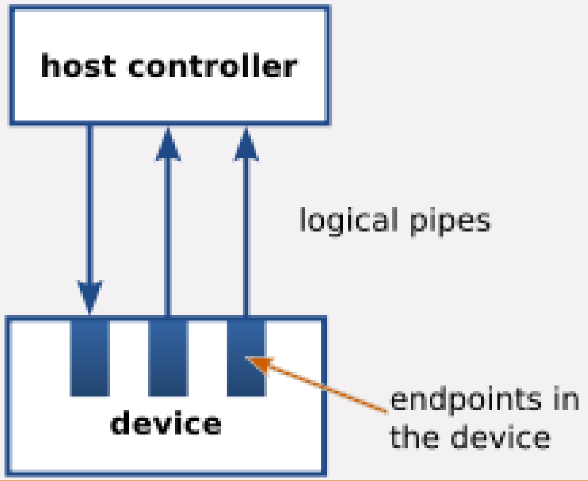
- **Host —** Computer that controls the interface
- **Host Controller** — device that provides USB capability to the host (USB Master Device)
- **Hub —** device with one or more connections to USB devices plus hardware to enable communications with each device
- **Device —** something that attaches to a USB port (sometimes synonymous with a function) (USB Slave Device)
- **Port —** a connector on the USB Host bus

USB Terminology

- **Suspend —** Device enters Suspend after 3mS of inactivity on the bus to minimize power consumption. Host uses timing packet to keep Peripherals active.
- **Enumeration** — Initialization sequence to inform the host what device was attached to the bus. Device parameters are conveyed at this point.
- **Descriptors —** List of tables that identify the capabilities of the device.
- **Endpoint —** All transmissions travel to/from an endpoint which is just a block of memory or a register. Endpoint 0 is the control endpoint which is the only bi-directional endpoint typically used for enumeration.

USB Host Controller Device And USB Devices

- One USB device, called the *USB host Controller*, acts as the master.
- Other USB devices, called *USB devices* or *USB peripherals*, act as slaves.
- The host / host controller initiates all bus transfers.
- Up to 127 USB devices can be connected to one USB host controller via up to 6 layers of cascaded hubs.



Bus Topology

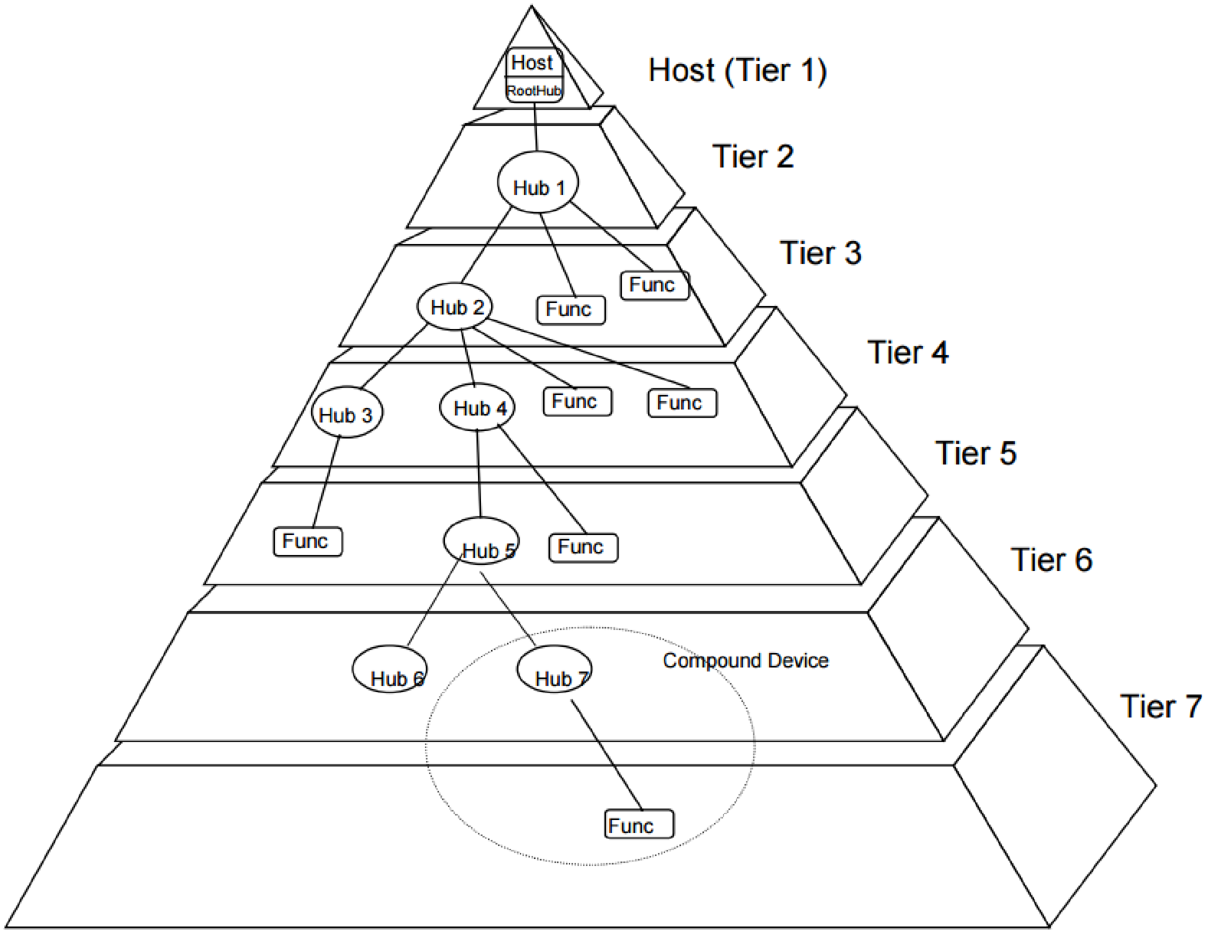
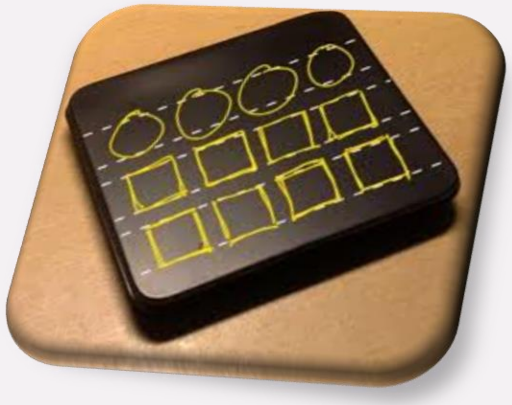
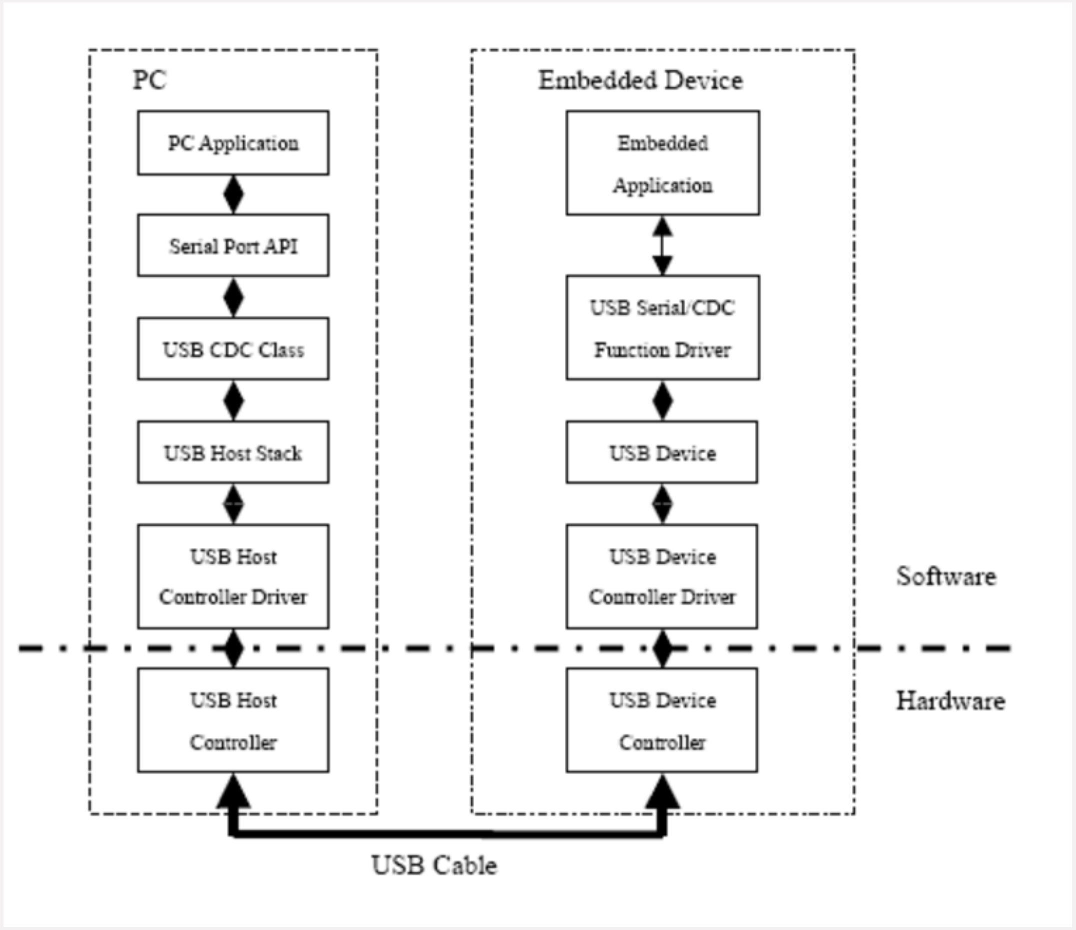


Figure 4-1. Bus Topology

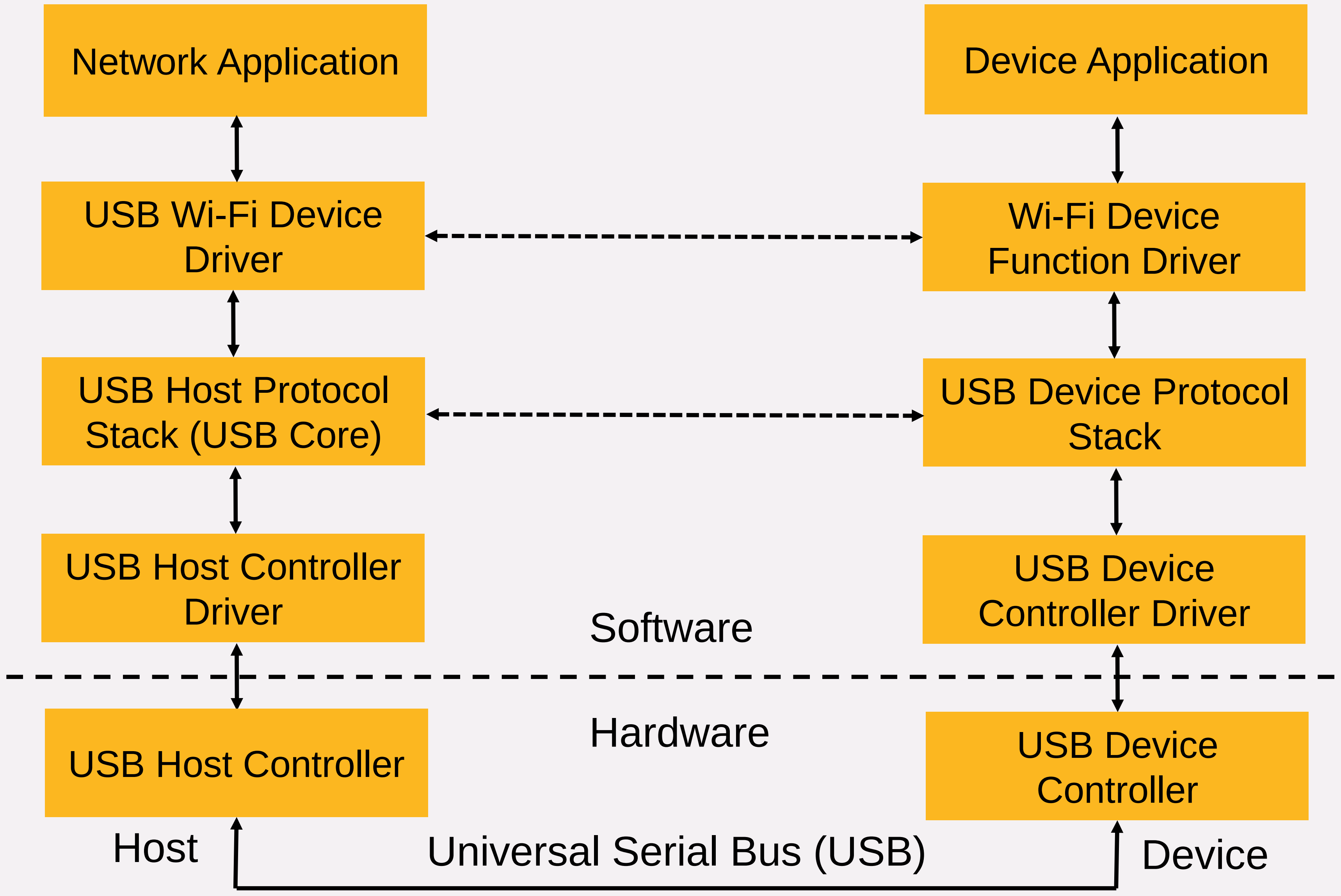
USB Device & Driver Layouts



PC TO DEVICE VIA SERIAL



Host USB S/W Stack & USB Wi-Fi DEVICE S/W Stack



Network Application
Device Application
USB Wi-Fi Device Driver
Wi-Fi Device Function Driver
USB Host Protocol Stack (USB Core)
USB Device Protocol Stack
USB Host Controller Driver
USB Device Controller Driver
Software
Hardware
USB Host Controller
USB Device Controller
Host
Universal Serial Bus (USB)
Device

USB Host Controller (USB Master Device)

- USB host requires a USB host controller and USB host software stack
- It is layered from the bottom up as follows:
  1. USB host controller device driver:
     - controls the USB host controller device – i.e. it reads and writes registers in the USB host controller and it transfers data
  2. USB host protocol stack:
     - implements the USB protocol and thus controls connected USB devices.
  3. USB class driver (USB slave/client device driver):
     - device-aware and communicates with and controls the actual USB device (e.g. USB disk drive, USB HID human interface device (keyboard, mouse etc), CDC communication device, etc.)
- **One USB host protocol stack can support multiple class drivers, simultaneously**

USB Device (USB Slave/Client Device)

- A USB device requires a USB device controller and USB device software.

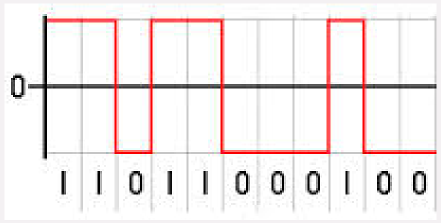
Layered from the bottom up as follows:

1. USB Device Controller driver:
   - controls the USB device controller – i.e. it reads and writes registers in the controller and it transfers data
2. USB device protocol stack:
   - Implements the USB protocol and thus communicates with the USB host protocol stack
3. USB function driver:
   - Communicates with the class driver in the host and provides the actual device control

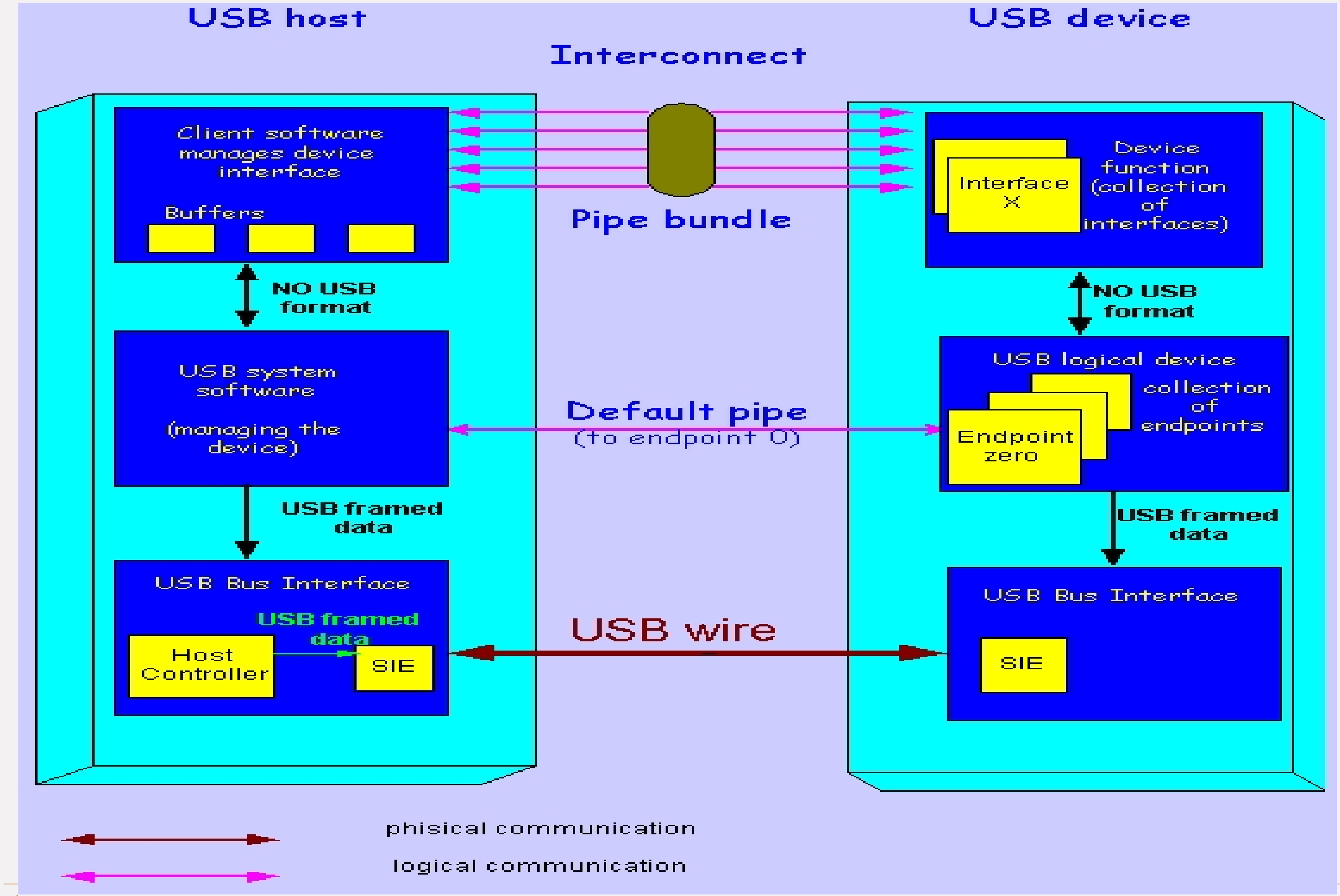
- One USB device protocol stack can support more than one function driver simultaneously, through the composite device framework.

Signaling On The Bus

- The USB cable is 4 wire cable
- Signal on the bus is done by signaling over two wires (D+ and D-)
- Data encoding and decoding is done using NRZI (Non Return to Zero Inverted)
  - a 0 bit is transmitted by toggling the data lines
  - a 1 bit is transmitted by leaving the data lines as-is.



Communication Flow: USB Host - USB Device



Aricent

USB Host Controllers Types

- EHCI
- OHCI
- UHCI

UHCI

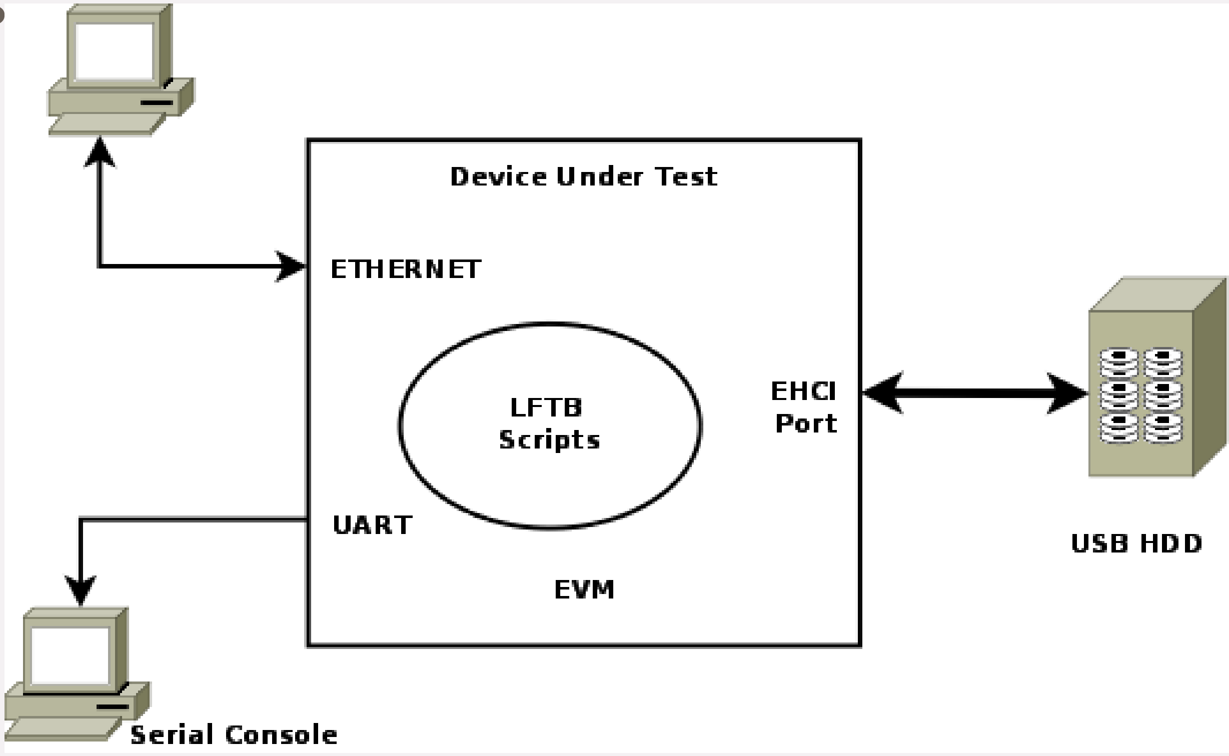
- Short for **Universal Host Controller** Interface, UHCI was developed by Intel and is a **USB 1.0 and 1.1** host controller that consists of two parts:
  - USB Host Controller Driver (HCD) and
  - USB Host Controller (HC).
- The HCD software is responsible for scheduling the traffic on USB by posting and maintaining transactions.
- HCD is part of the system software and is typically provided by the chip(SoC) vendor or operating system vendor.

OHCI

- Short for **Open Host** Controller Interface, OHCI was developed by Compaq and is a standard that allows a computer host to interface with USB 1.0 and 1.1 devices.

EHCI

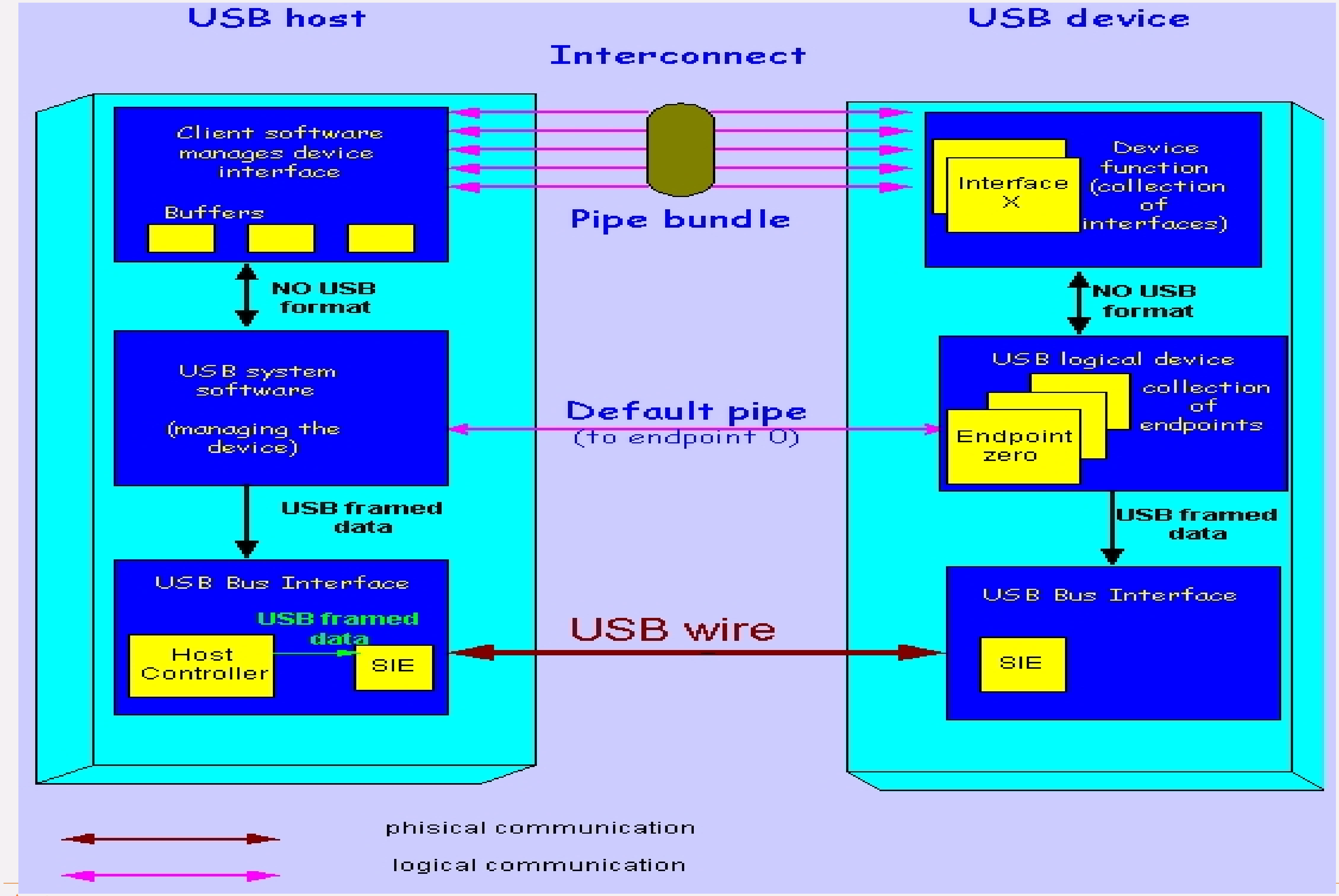
- Short for Enhanced Host Controller Interface.
- EHCI is a standard that allows a computer host to interface with **USB 2.0** devices



Queries

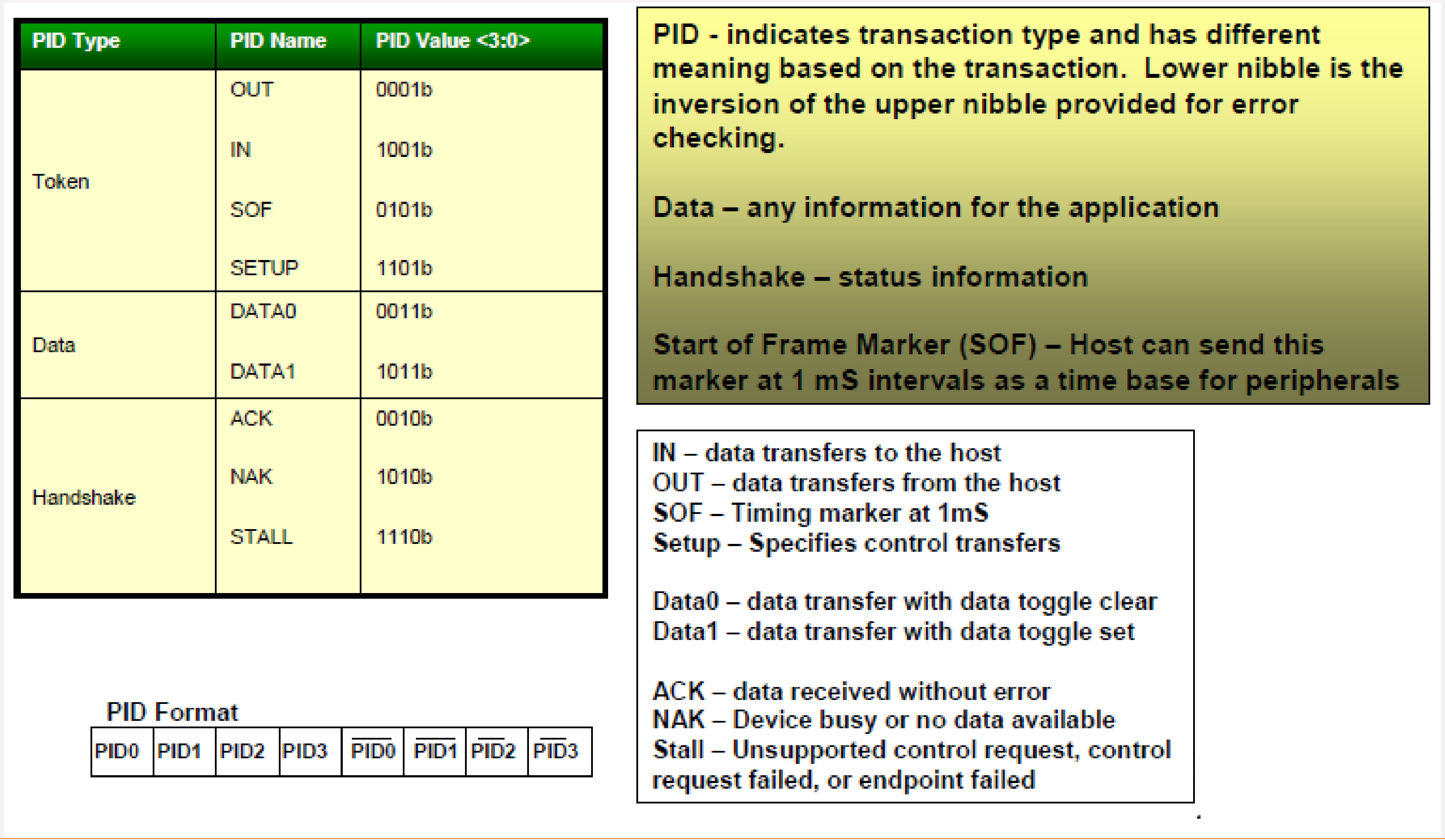
USB Core & the USB protocol

Communication Flow: USB Host – USB Device



USB Packet Identifiers (PID)

- The PID signals to the receiver that what the packet structure and content will be and how the receiver has to respond.



PID Type	PID Name	PID Value <3:0>
Token	OUT	0001b
	IN	1001b
	SOF	0101b
	SETUP	1101b
Data	DATA0	0011b
	DATA1	1011b
Handshake	ACK	0010b
	NAK	1010b
	STALL	1110b

PID Format

PID0	PID1	PID2	PID3	$\overline{\text{PID0}}$	$\overline{\text{PID1}}$	$\overline{\text{PID2}}$	$\overline{\text{PID3}}$

PID - indicates transaction type and has different meaning based on the transaction. Lower nibble is the inversion of the upper nibble provided for error checking.

Data – any information for the application

Handshake – status information

Start of Frame Marker (SOF) – Host can send this marker at 1 mS intervals as a time base for peripherals

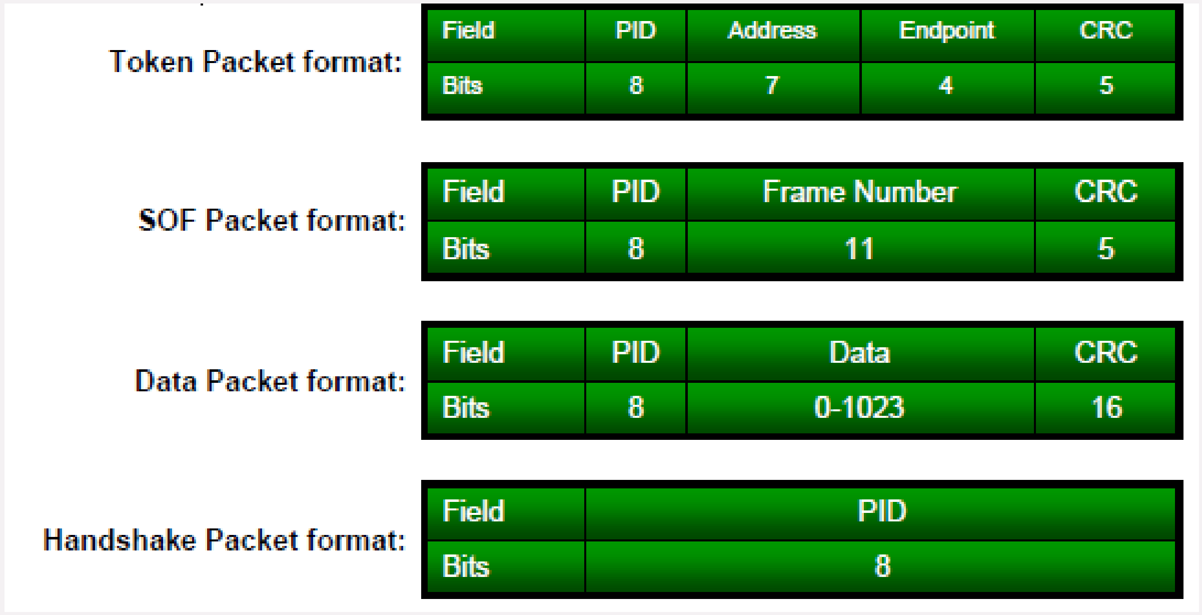
IN – data transfers to the host
OUT – data transfers from the host
SOF – Timing marker at 1mS
Setup – Specifies control transfers

Data0 – data transfer with data toggle clear
Data1 – data transfer with data toggle set

ACK – data received without error
NAK – Device busy or no data available
Stall – Unsupported control request, control request failed, or endpoint failed

USB Packets

- Packets—block of information with a defined data structure. The packet is the lowest level of the USB transfer hierarchy describing the physical layer of the interface.
- If you were to monitor D+ and D- you would see the packet fields:
  - ➢ □ Packet identifier
  - ➢ □ Address
  - ➢ □ Endpoint
  - ➢ □ Data
  - ➢ □ Frame number
  - ➢ □ CRC



Token Packet format:

Field	PID	Address	Endpoint	CRC
Bits	8	7	4	5

SOF Packet format:

Field	PID	Frame Number	CRC
Bits	8	11	5

Data Packet format:

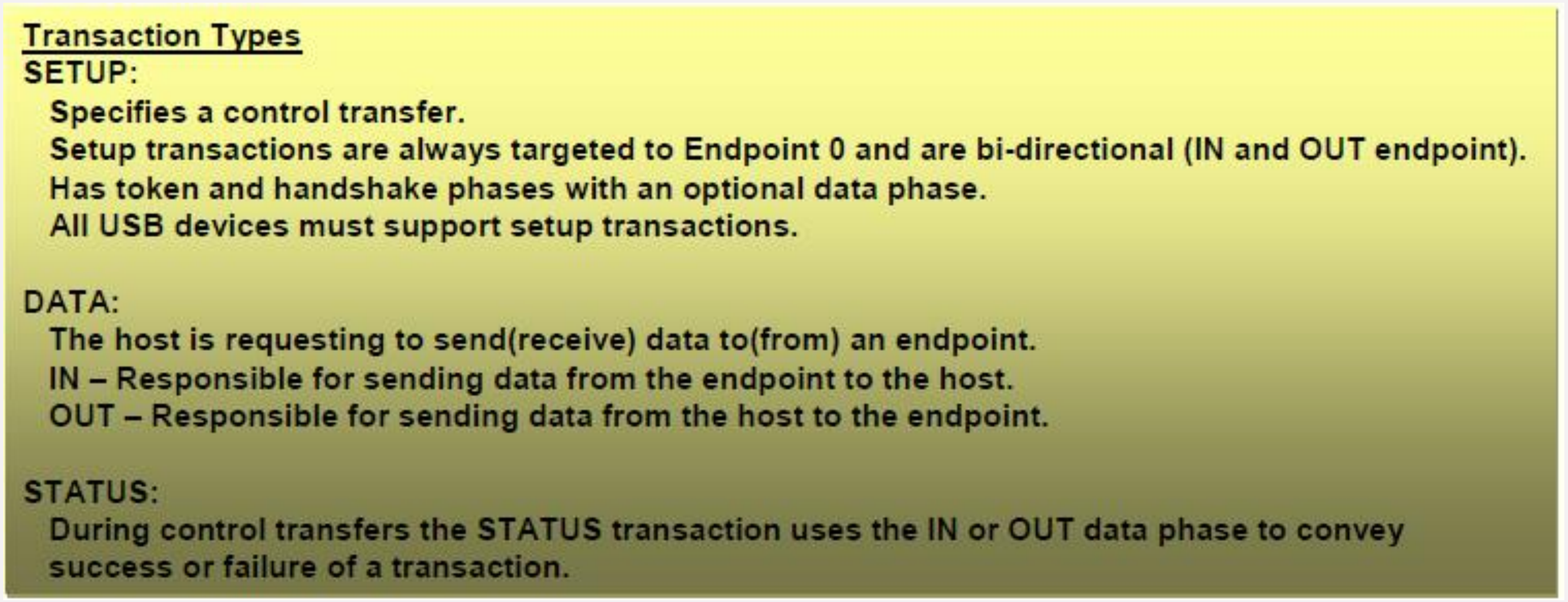
Field	PID	Data	CRC
Bits	8	0-1023	16

Handshake Packet format:

Field	PID
Bits	8

USB Transactions

A transaction allows a set of operations to be grouped in such a way that they either all succeed or all fail; further, the operations in the set appear from outside the transaction to occur simultaneously. If the transaction is unsuccessful then the host/device ignores any data that was received.



Transaction Types

SETUP:

Specifies a control transfer.
Setup transactions are always targeted to Endpoint 0 and are bi-directional (IN and OUT endpoint).
Has token and handshake phases with an optional data phase.
All USB devices must support setup transactions.

DATA:

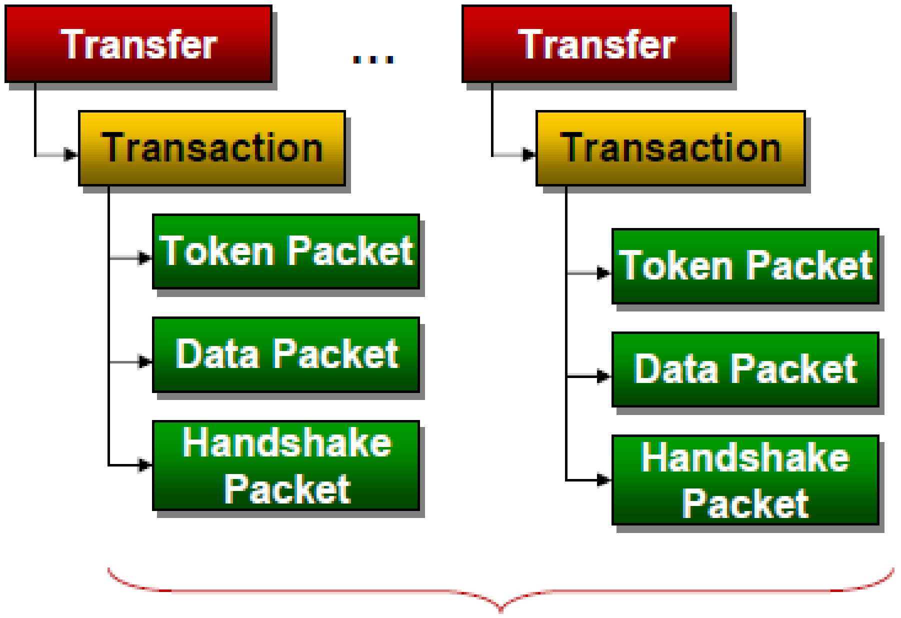
The host is requesting to send(receive) data to(from) an endpoint.
IN – Responsible for sending data from the endpoint to the host.
OUT – Responsible for sending data from the host to the endpoint.

STATUS:

During control transfers the STATUS transaction uses the IN or OUT data phase to convey success or failure of a transaction.

What Is A Transfer?

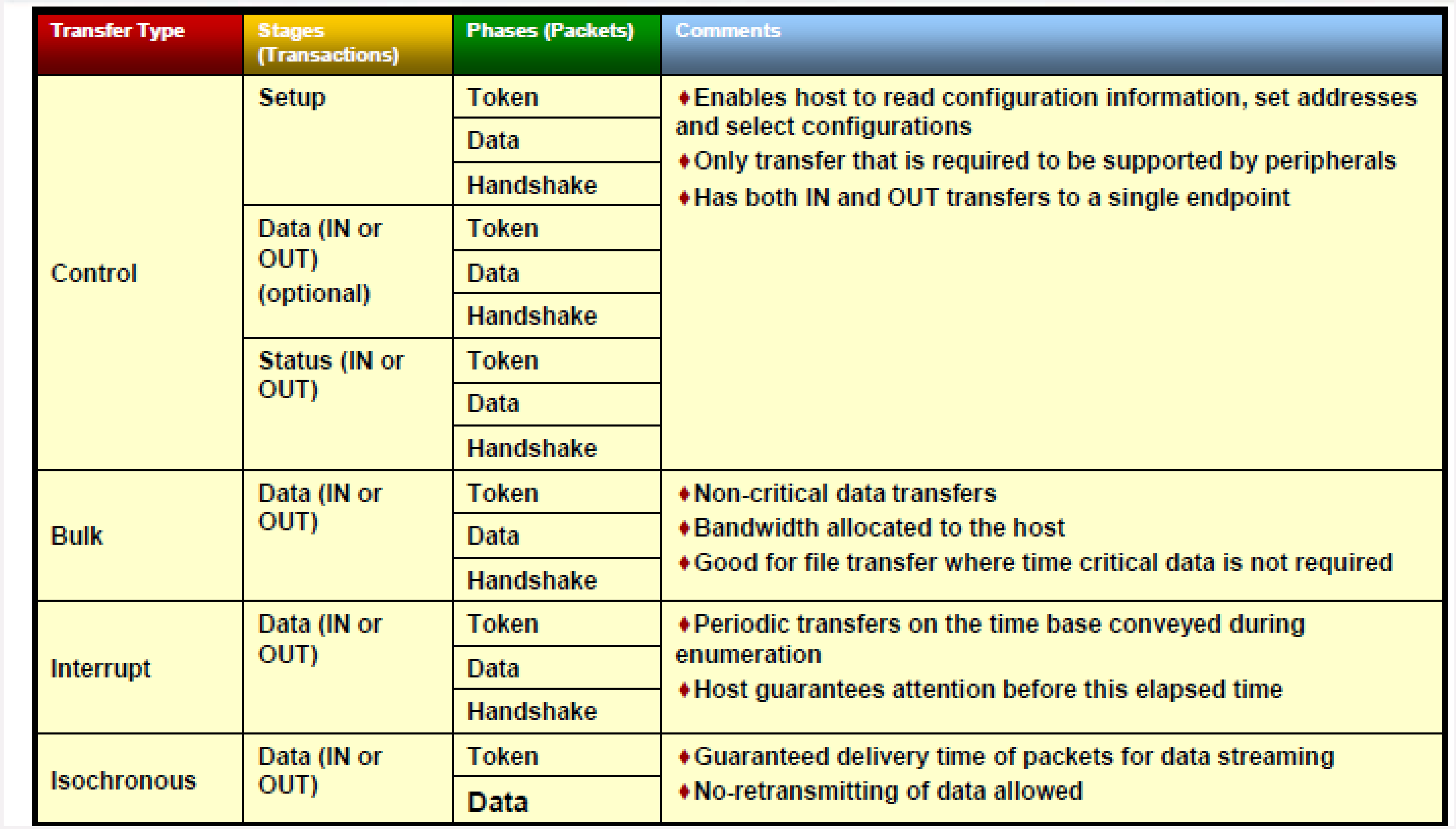
- The **transfer** is the process of making a communications request with an endpoint. Transfers determine aspects of the communications flow such as:
  - Data format imposed by the USB
  - Direction of communication flow
  - Packet size constraints
  - Bus access constraints
  - Latency constraints
  - Required data sequences
  - Error Handling
- A **transfer** has one or more transactions which then has one, two or three packets



- Transfers are divided into transactions.
- Transactions are made up of packets.
- The host controls transfers by allocating transactions to a frame.
- Transfers may span multiple frames.

Transfer Format

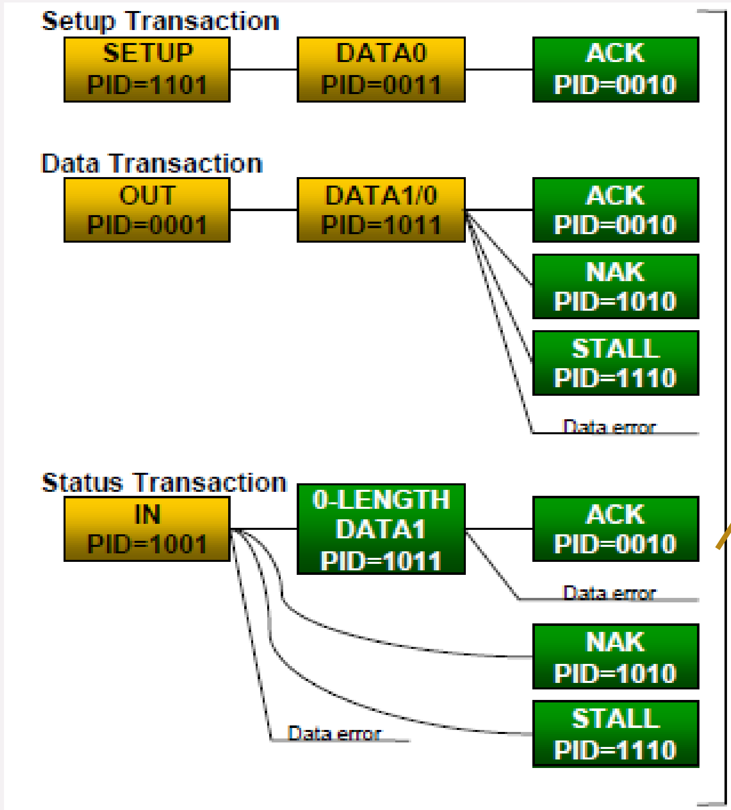
Match the color with previous slide which denote transfer, transaction and packets.



Transfer Type	Stages (Transactions)	Phases (Packets)	Comments
Control	Setup	Token	♦Enables host to read configuration information, set addresses and select configurations ♦Only transfer that is required to be supported by peripherals ♦Has both IN and OUT transfers to a single endpoint
		Data	
		Handshake	
	Data (IN or OUT) (optional)	Token	
		Data	
		Handshake	
	Status (IN or OUT)	Token	
		Data	
		Handshake	
Bulk	Data (IN or OUT)	Token	♦Non-critical data transfers ♦Bandwidth allocated to the host ♦Good for file transfer where time critical data is not required
		Data	
		Handshake	
Interrupt	Data (IN or OUT)	Token	♦Periodic transfers on the time base conveyed during enumeration ♦Host guarantees attention before this elapsed time
		Data	
		Handshake	
Isochronous	Data (IN or OUT)	Token	♦Guaranteed delivery time of packets for data streaming ♦No-retransmitting of data allowed
		Data	

Control Write Transfers (OUT)

- Control Write Transfers (OUT)—contains Setup, Data (optional), and handshake transactions.



Host sends Setup Packet followed by 8 bytes of data for the request. The device must return an ACK.

Host sends OUT Packet along with the data for writes. Device responds with handshake. No response indicates a data error. DATA1 is first and alternates DATA0 and DATA1.

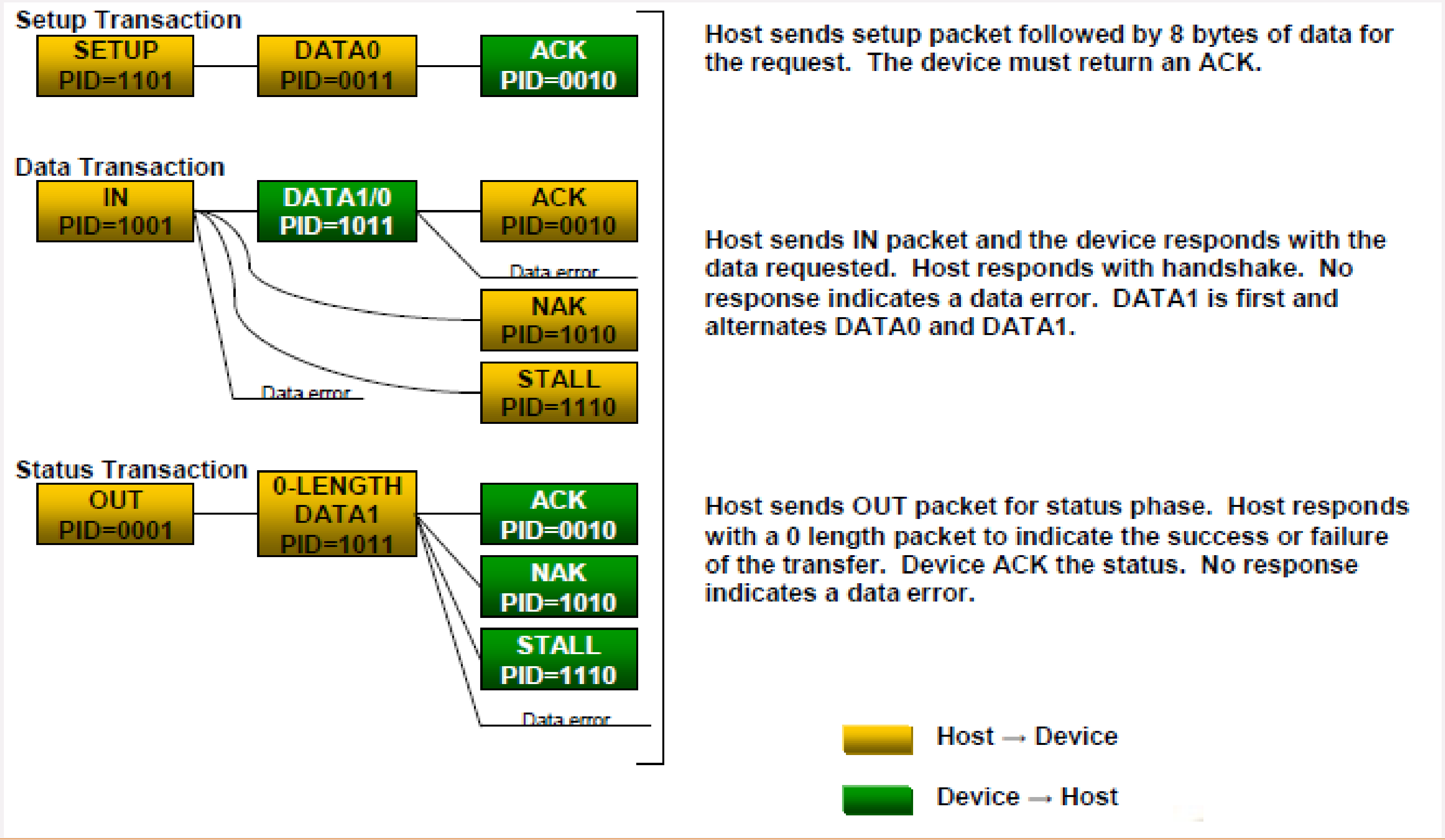
This should be yellow.

Host sends IN Packet for Status phase. Device responds with a 0 length packet to indicate the success or failure of the transfer. Host ACK the status. No response indicates a data error.

🟨	Host → Device
🟩	Device → Host

Control Read Transfers (IN) 1 OF 2

- Control Read Transfers (IN)—contains Setup, Data (optional), and handshake transactions.

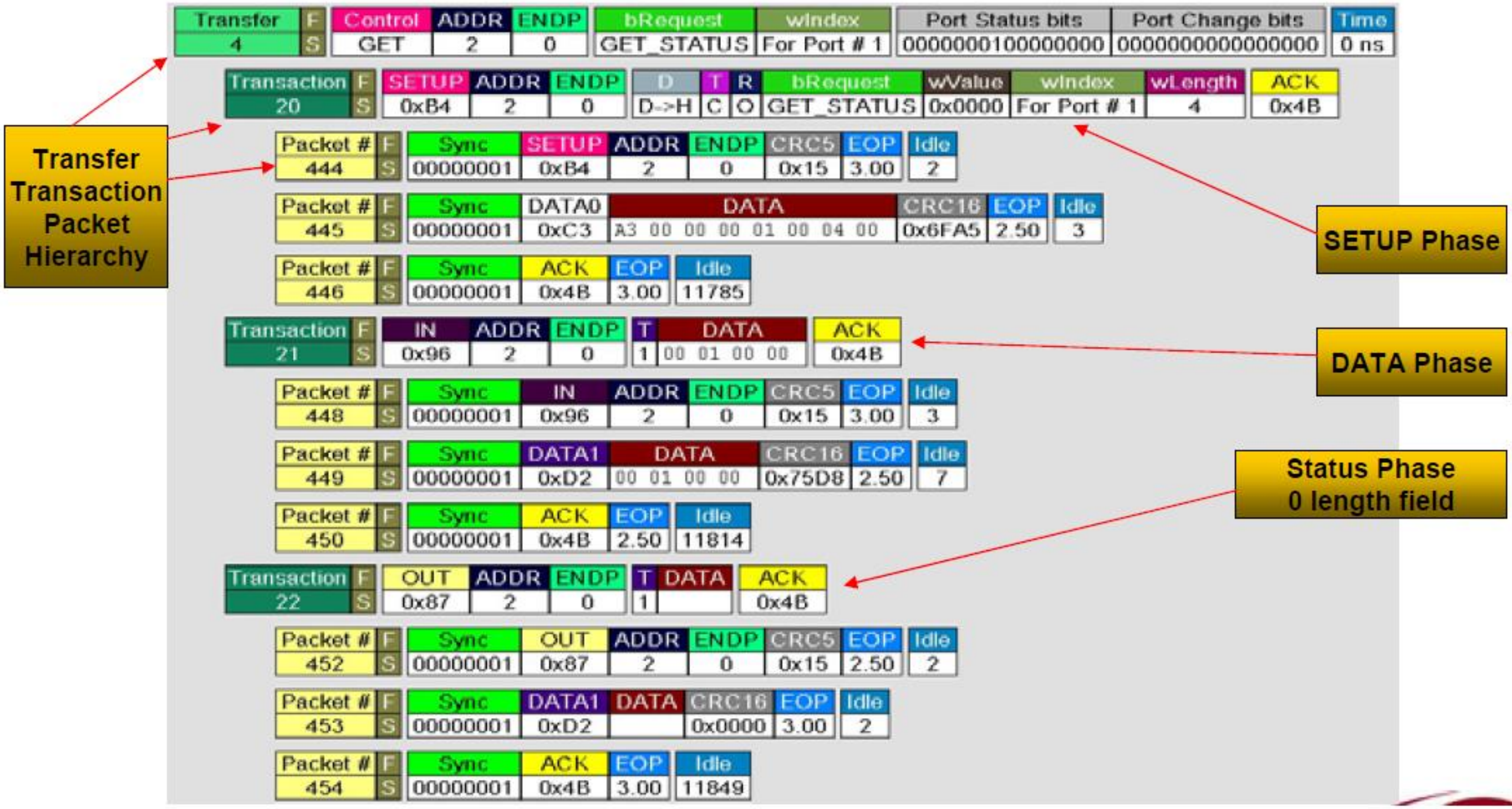


Host sends setup packet followed by 8 bytes of data for the request. The device must return an ACK.

Host sends IN packet and the device responds with the data requested. Host responds with handshake. No response indicates a data error. DATA1 is first and alternates DATA0 and DATA1.

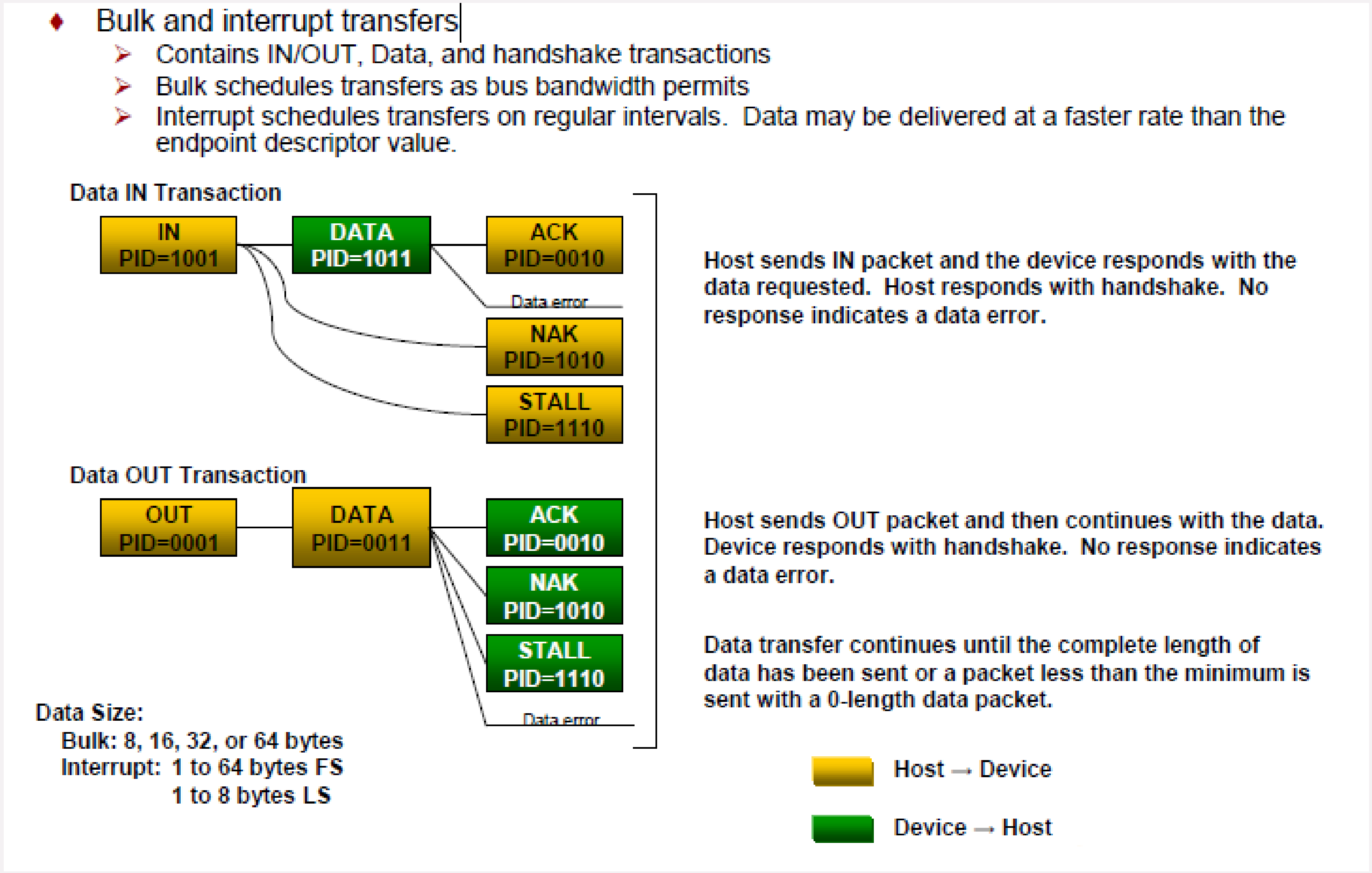
Host sends OUT packet for status phase. Host responds with a 0 length packet to indicate the success or failure of the transfer. Device ACK the status. No response indicates a data error.

Control Read Transfers (IN) 2 OF 2


Transfer Transaction Packet Hierarchy
SETUP Phase
DATA Phase
Status Phase 0 length field

```
Transfer F | Control ADDR ENDP | bRequest   | wIndex         | Port Status bits  | Port Change bits  | Time
4        S | GET     2    0    | GET_STATUS | For Port # 1   | 0000000100000000  | 0000000000000000  | 0 ns

  Transaction F | SETUP ADDR ENDP | D    T R | bRequest   | wValue | wIndex       | wLength | ACK
  20          S | 0xB4  2    0    | D->H C O | GET_STATUS | 0x0000 | For Port # 1 | 4       | 0x4B

    Packet # F | Sync     | SETUP | ADDR | ENDP | CRC5 | EOP  | Idle
    444      S | 00000001 | 0xB4  | 2    | 0    | 0x15 | 3.00 | 2

    Packet # F | Sync     | DATA0 | DATA                    | CRC16  | EOP  | Idle
    445      S | 00000001 | 0xC3  | A3 00 00 00 01 00 04 00 | 0x6FA5 | 2.50 | 3

    Packet # F | Sync     | ACK   | EOP  | Idle
    446      S | 00000001 | 0x4B  | 3.00 | 11785

  Transaction F | IN    ADDR ENDP | T | DATA        | ACK
  21          S | 0x96  2    0    | 1 | 00 01 00 00 | 0x4B

    Packet # F | Sync     | IN    | ADDR | ENDP | CRC5 | EOP  | Idle
    448      S | 00000001 | 0x96  | 2    | 0    | 0x15 | 3.00 | 3

    Packet # F | Sync     | DATA1 | DATA        | CRC16  | EOP  | Idle
    449      S | 00000001 | 0xD2  | 00 01 00 00 | 0x75D8 | 2.50 | 7

    Packet # F | Sync     | ACK   | EOP  | Idle
    450      S | 00000001 | 0x4B  | 2.50 | 11814

  Transaction F | OUT   ADDR ENDP | T | DATA | ACK
  22          S | 0x87  2    0    | 1 |      | 0x4B

    Packet # F | Sync     | OUT   | ADDR | ENDP | CRC5 | EOP  | Idle
    452      S | 00000001 | 0x87  | 2    | 0    | 0x15 | 2.50 | 2

    Packet # F | Sync     | DATA1 | DATA | CRC16  | EOP  | Idle
    453      S | 00000001 | 0xD2  |      | 0x0000 | 3.00 | 2

    Packet # F | Sync     | ACK   | EOP  | Idle
    454      S | 00000001 | 0x4B  | 3.00 | 11849
```

Bulk And Interrupt Transfers 1 of 2



- Bulk and interrupt transfers
  - Contains IN/OUT, Data, and handshake transactions
  - Bulk schedules transfers as bus bandwidth permits
  - Interrupt schedules transfers on regular intervals. Data may be delivered at a faster rate than the endpoint descriptor value.

Host sends IN packet and the device responds with the data requested. Host responds with handshake. No response indicates a data error.

Host sends OUT packet and then continues with the data. Device responds with handshake. No response indicates a data error.

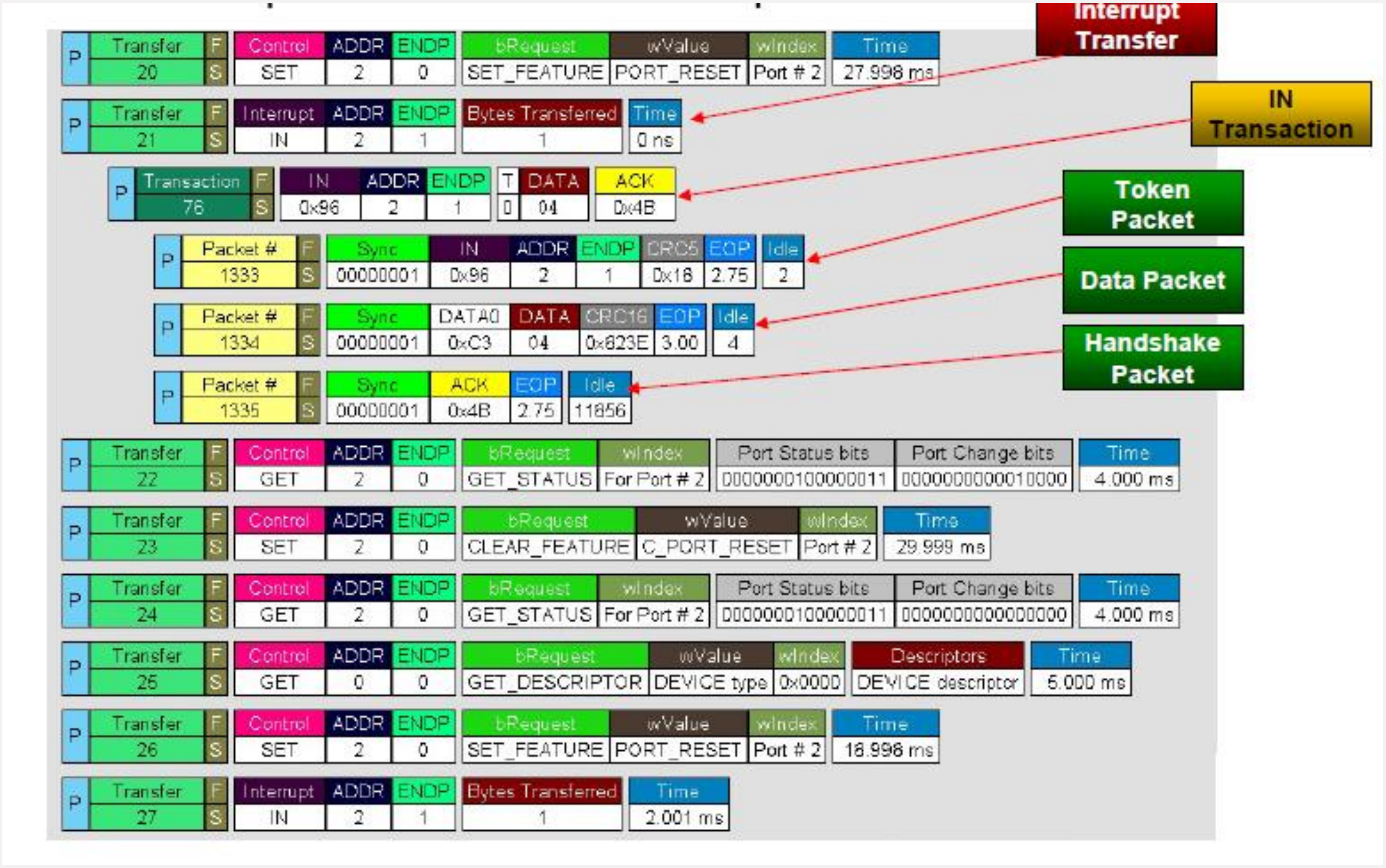
Data transfer continues until the complete length of data has been sent or a packet less than the minimum is sent with a 0-length data packet.

Data Size:
Bulk: 8, 16, 32, or 64 bytes
Interrupt: 1 to 64 bytes FS
1 to 8 bytes LS

🟨 Host → Device

🟩 Device → Host

Bulk And Interrupt Transfers 2 of 2



P	Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Time
	20	S	SET	2	0	SET_FEATURE	PORT_RESET	Port # 2	27.998 ms

P	Transfer	F	Interrupt	ADDR	ENDP	Bytes Transferred	Time
	21	S	IN	2	1	1	0 ns

P	Transaction	F	IN	ADDR	ENDP	T	DATA	ACK
	76	S	0x96	2	1	0	04	0x4B

P	Packet #	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
	1333	S	00000001	0x96	2	1	0x18	2.75	2

P	Packet #	F	Sync	DATA0	DATA	CRC16	EOP	Idle
	1334	S	00000001	0xC3	04	0x823E	3.00	4

P	Packet #	F	Sync	ACK	EOP	Idle
	1335	S	00000001	0x4B	2.75	11856

P	Transfer	F	Control	ADDR	ENDP	bRequest	wIndex	Port Status bits	Port Change bits	Time
	22	S	GET	2	0	GET_STATUS	For Port # 2	0000000100000011	0000000000010000	4.000 ms

P	Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Time
	23	S	SET	2	0	CLEAR_FEATURE	C_PORT_RESET	Port # 2	29.999 ms

P	Transfer	F	Control	ADDR	ENDP	bRequest	wIndex	Port Status bits	Port Change bits	Time
	24	S	GET	2	0	GET_STATUS	For Port # 2	0000000100000011	0000000000000000	4.000 ms

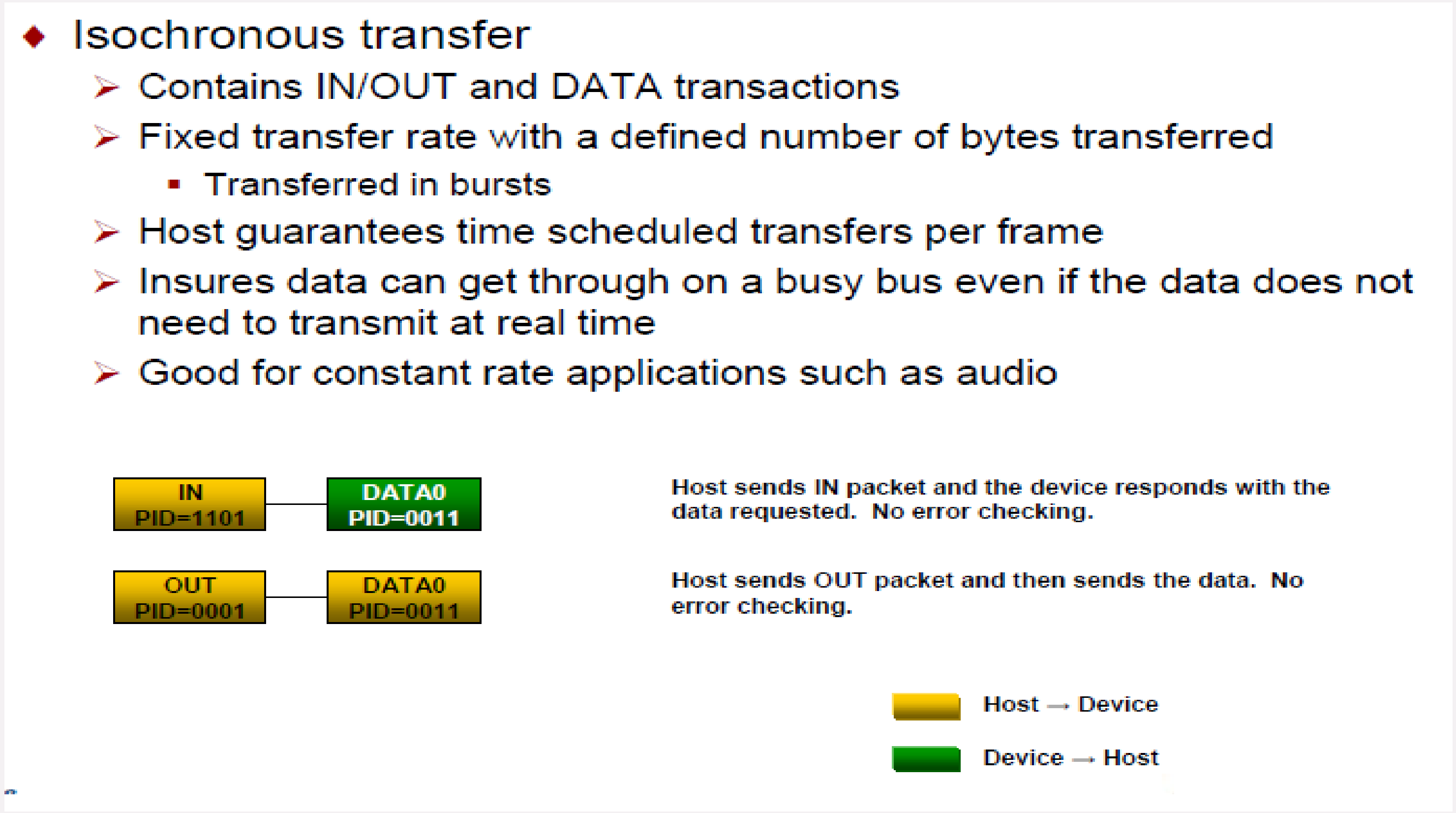
P	Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time
	25	S	GET	0	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE descriptor	5.000 ms

P	Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Time
	26	S	SET	2	0	SET_FEATURE	PORT_RESET	Port # 2	16.998 ms

P	Transfer	F	Interrupt	ADDR	ENDP	Bytes Transferred	Time
	27	S	IN	2	1	1	2.001 ms

Isochronous Transfer 1 of 2

- This is the Isochronous transfer and there is no handshake packets associated with this transfer type.



- Isochronous transfer
  - Contains IN/OUT and DATA transactions
  - Fixed transfer rate with a defined number of bytes transferred
    - Transferred in bursts
  - Host guarantees time scheduled transfers per frame
  - Insures data can get through on a busy bus even if the data does not need to transmit at real time
  - Good for constant rate applications such as audio

IN PID=1101 — DATA0 PID=0011

Host sends IN packet and the device responds with the data requested. No error checking.

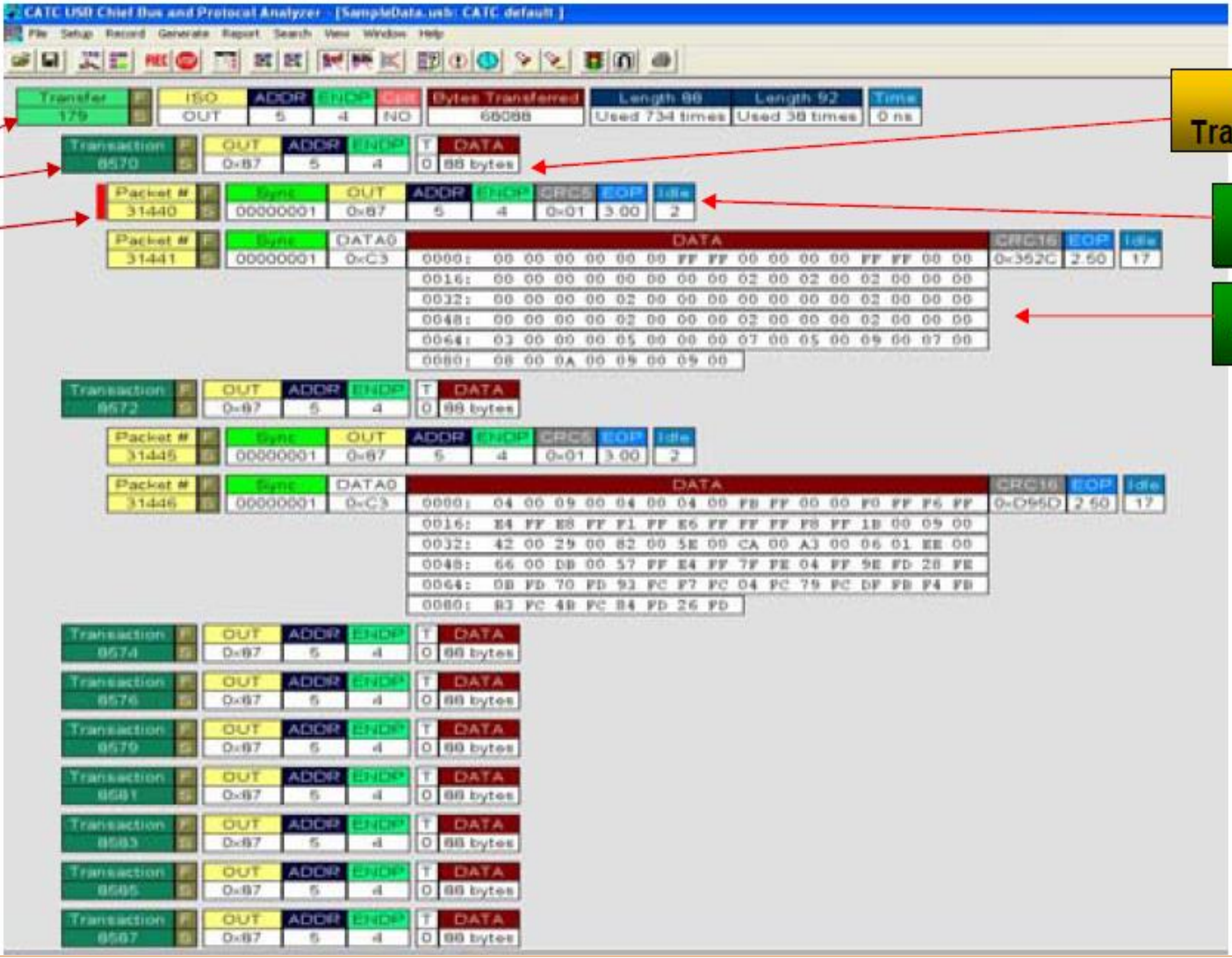
OUT PID=0001 — DATA0 PID=0011

Host sends OUT packet and then sends the data. No error checking.

🟨 Host → Device

🟩 Device → Host

Isochronous Transfer 2 of 2


CATC USB Chief Bus and Protocol Analyzer - [SampleData.usb: CATC default]
File Setup Record Generate Report Search View Window Help
OUT Transaction
Transfer Transaction Packet Hierarchy
Token Packet
Data Packet

Transfer	ISO	ADDR	ENDP	Cpl	Bytes Transferred	Length 88	Length 92	Time
179	OUT	5	4	NO	68088	Used 734 times	Used 38 times	0 ns

Transaction	OUT	ADDR	ENDP	T	DATA
8570	0x87	5	4	0	88 bytes

Packet #	Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle
31440	00000001	0x87	5	4	0x01	3.00	2

Packet #	Sync	DATA0	DATA	CRC16	EOP	Idle
31441	00000001	0xC3	0000: 00 00 00 00 00 00 FF FF 00 00 00 00 FF FF 00 00	0x362C	2.50	17
			0016: 00 00 00 00 00 00 00 00 02 00 02 00 02 00 00 00			
			0032: 00 00 00 00 02 00 00 00 00 00 00 00 02 00 00 00			
			0048: 00 00 00 00 02 00 00 00 02 00 00 00 02 00 00 00			
			0064: 03 00 00 00 05 00 00 00 07 00 05 00 09 00 07 00			
			0080: 08 00 0A 00 09 00 09 00			

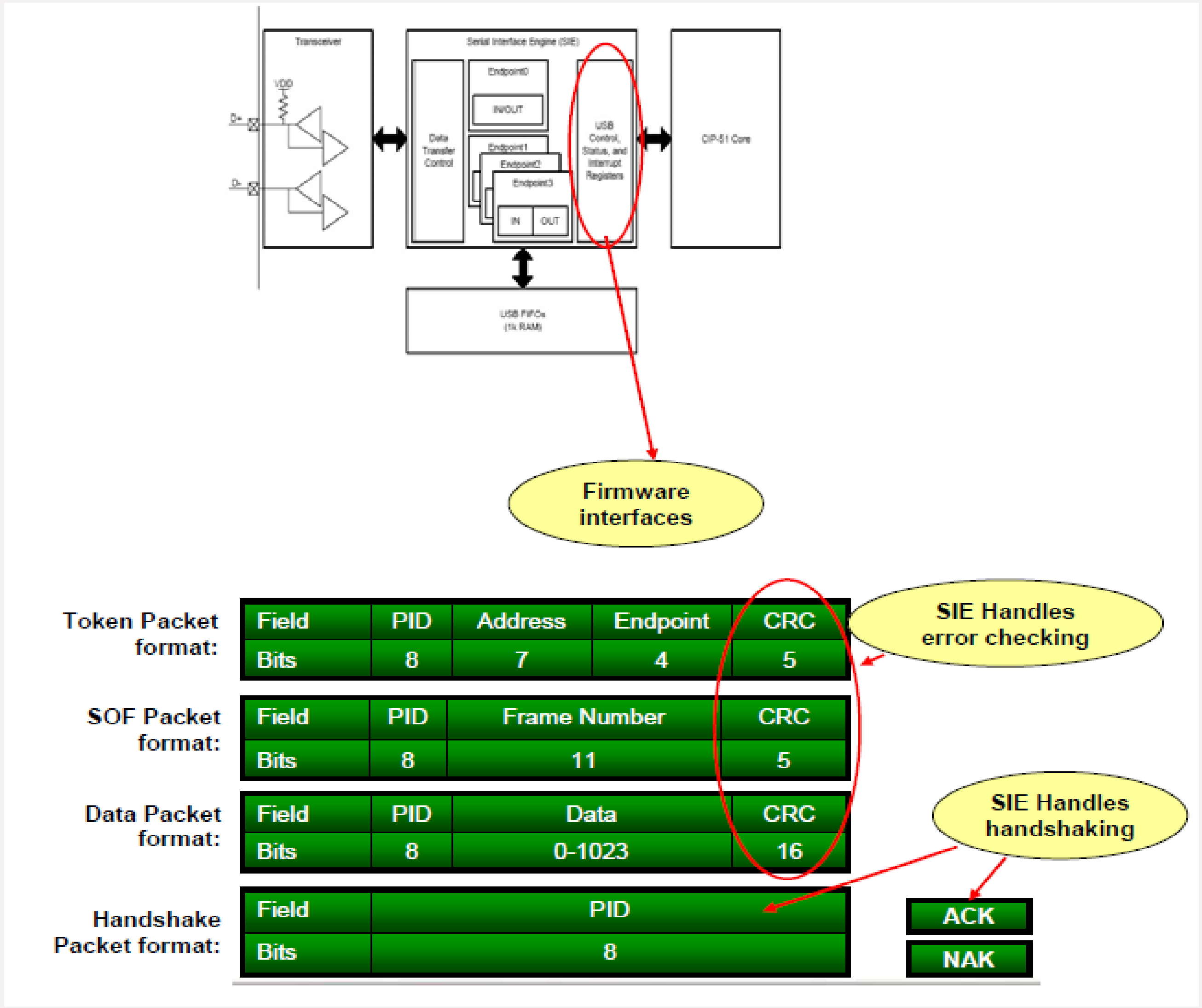
Transaction	OUT	ADDR	ENDP	T	DATA
8572	0x87	5	4	0	88 bytes

Packet #	Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle
31445	00000001	0x87	5	4	0x01	3.00	2

Packet #	Sync	DATA0	DATA	CRC16	EOP	Idle
31446	00000001	0xC3	0000: 04 00 09 00 04 00 04 00 FB FF 00 00 F0 FF F6 FF	0xD95D	2.50	17
			0016: E4 FF E8 FF F1 FF E6 FF FF FF F8 FF 1B 00 09 00			
			0032: 42 00 29 00 82 00 5E 00 CA 00 A3 00 06 01 EE 00			
			0048: 66 00 DB 00 57 FF E4 FF 7F FE 04 FF 9E FD 28 FE			
			0064: 0B FD 70 FD 93 FC F7 FC 04 FC 79 FC DF FB F4 FB			
			0080: B3 FC 4B FC B4 FD 26 FD			

Transaction	OUT	ADDR	ENDP	T	DATA
8574	0x87	5	4	0	88 bytes
8576	0x87	5	4	0	88 bytes
8579	0x87	5	4	0	88 bytes
8581	0x87	5	4	0	88 bytes
8583	0x87	5	4	0	88 bytes
8585	0x87	5	4	0	88 bytes
8587	0x87	5	4	0	88 bytes

Serial Interface Engine (SIE)



Token Packet format:

Field	PID	Address	Endpoint	CRC
Bits	8	7	4	5

SOF Packet format:

Field	PID	Frame Number	CRC
Bits	8	11	5

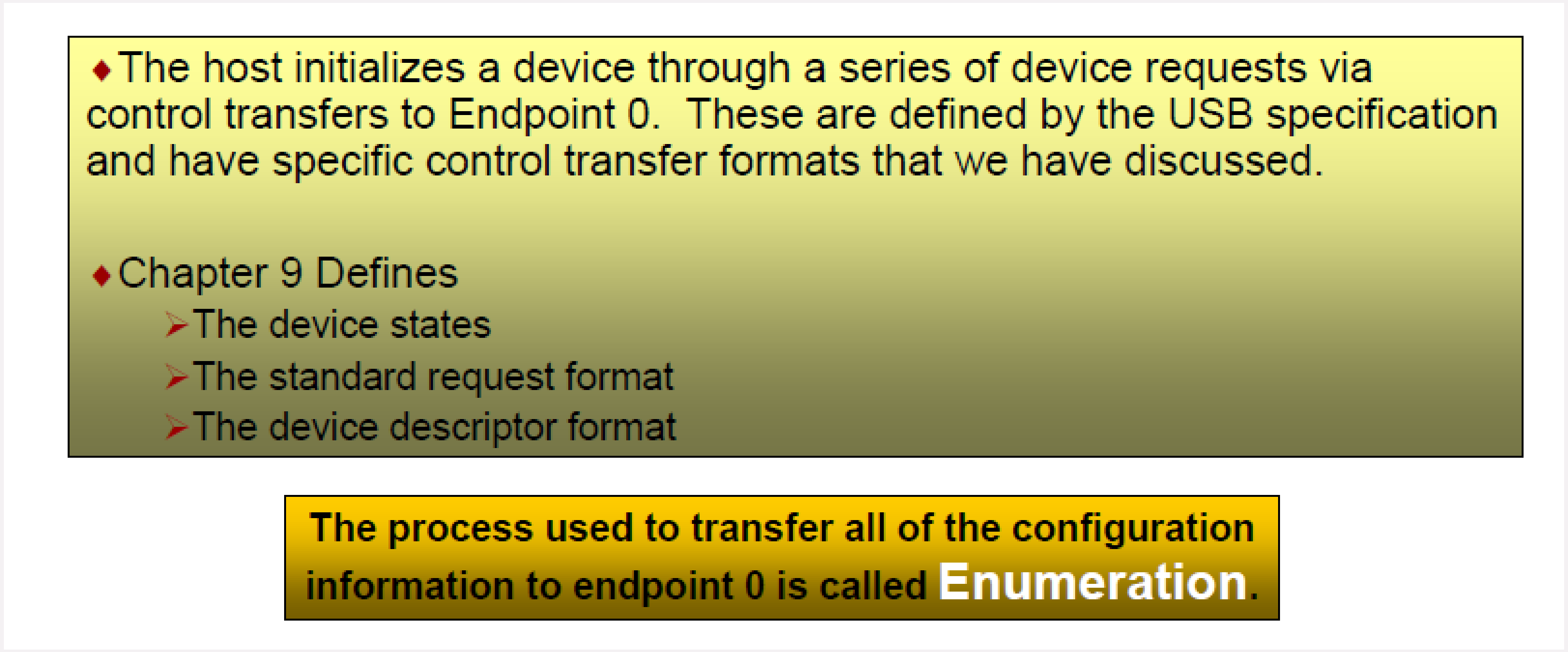
Data Packet format:

Field	PID	Data	CRC
Bits	8	0-1023	16

Handshake Packet format:

Field	PID
Bits	8

USB Protocol Implementation


- The host initializes a device through a series of device requests via control transfers to Endpoint 0. These are defined by the USB specification and have specific control transfer formats that we have discussed.
- Chapter 9 Defines
  - The device states
  - The standard request format
  - The device descriptor format

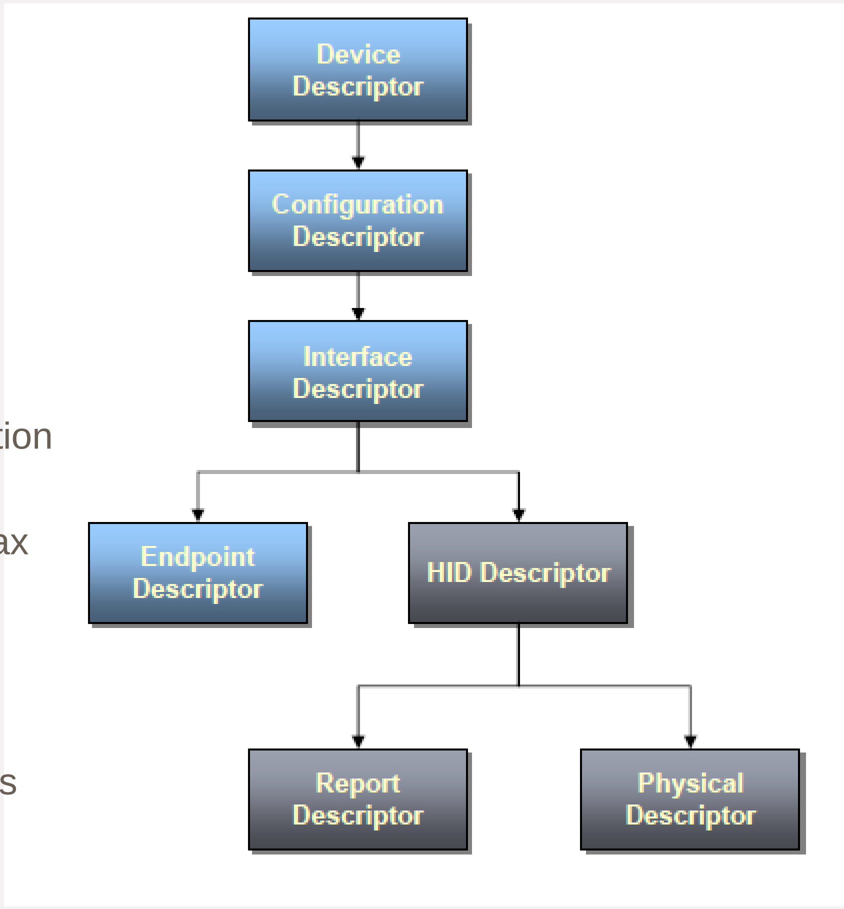
The process used to transfer all of the configuration information to endpoint 0 is called Enumeration.

The enumeration process begins:

- The host initiates a set of communication requests to the device to determine the who, what, and how about the device.
- The device has pre-defined structures located in flash that describe what it does and how it needs to do it.
- These are called descriptors.

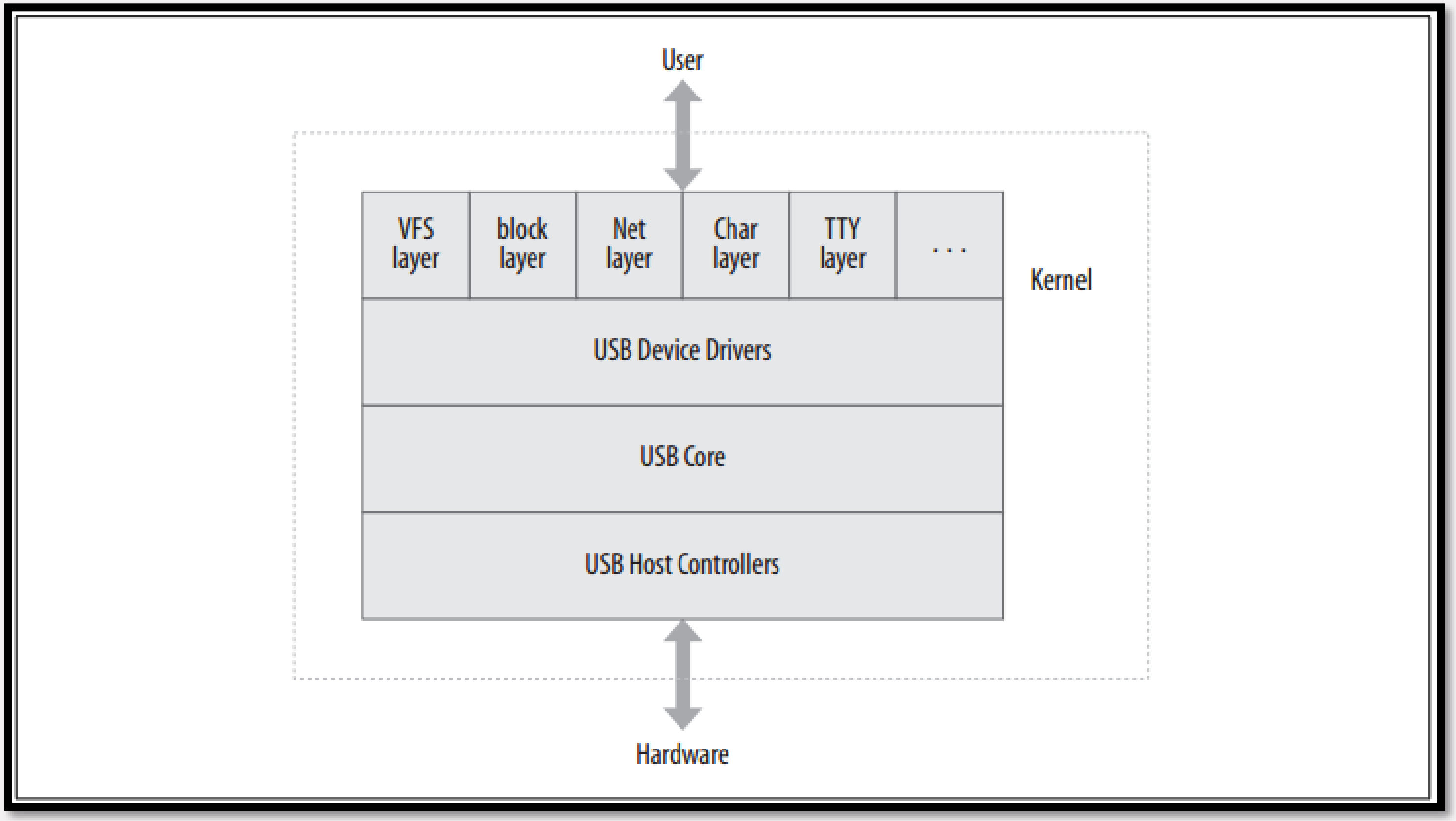
USB Descriptors Types

- **Device descriptor**
  - General info about a USB device (vendor ID etc)
  - Contains info that applies globally to the device
  - Only one device descriptor
- **Configuration descriptor**
  - USB devices can have multiple configurations
  - Each configuration contains one or more interfaces
  - All associated interface and endpoint descriptors get loaded with a request from the host for the configuration descriptor
  - Contains fields like remote wake-up capability and max power requirements
- **Interface descriptor**
  - Lists the endpoint descriptors for the interface
  - Identifies if the interface belongs to a predefined Class (such as the Human Interface Device or HID)
- **Endpoint descriptor**
  - Info required by host to determine bandwidth requirements
  - Describes endpoint number and address, IN or OUT endpoint and the transfer types requested



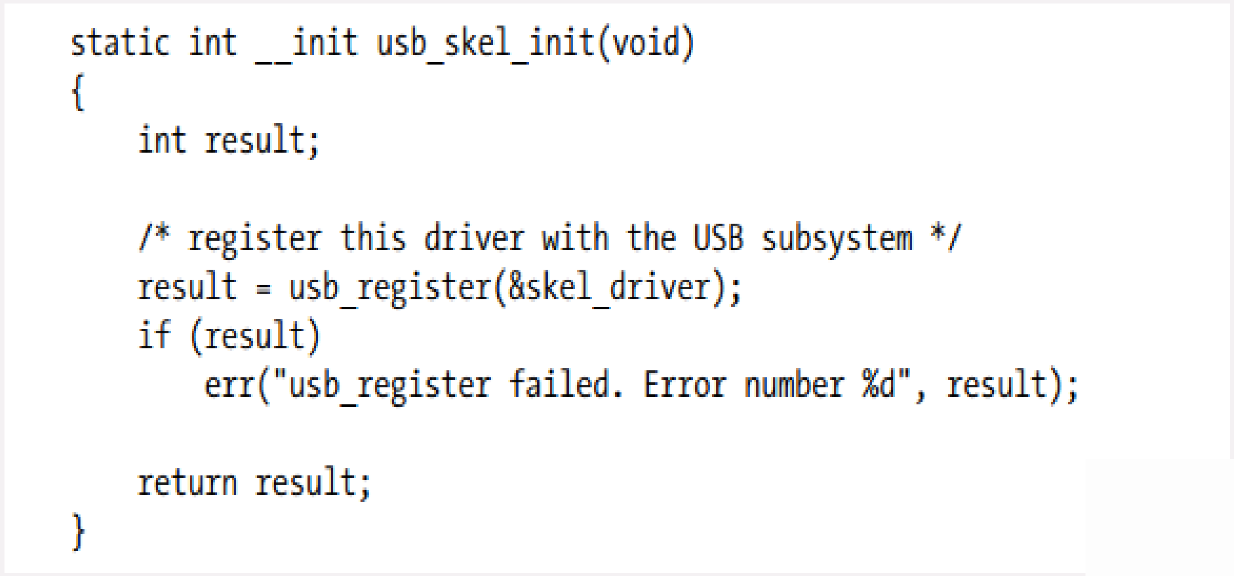
USB Driver, Device Registration & Hot-pluggability

USB Sub-system overview



USB device driver registration

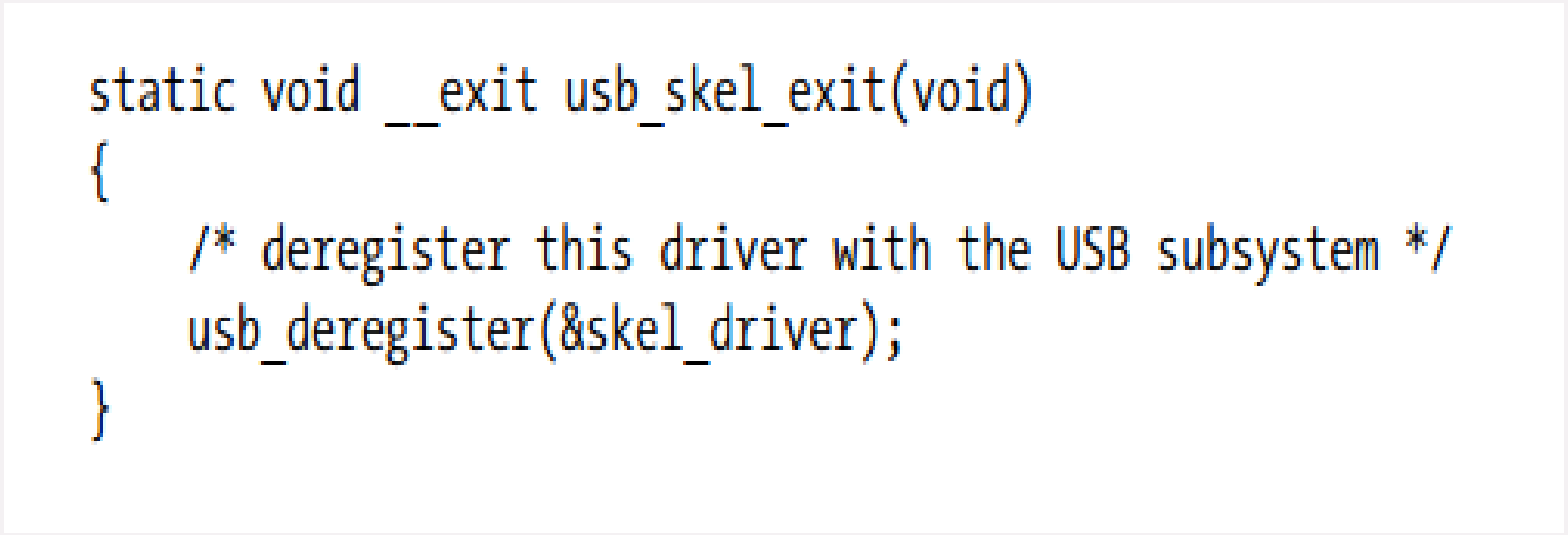
- To register the struct usb_driver with the USB core, a call to usb_register is made with a pointer to the struct usb_driver



```
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;
}
```

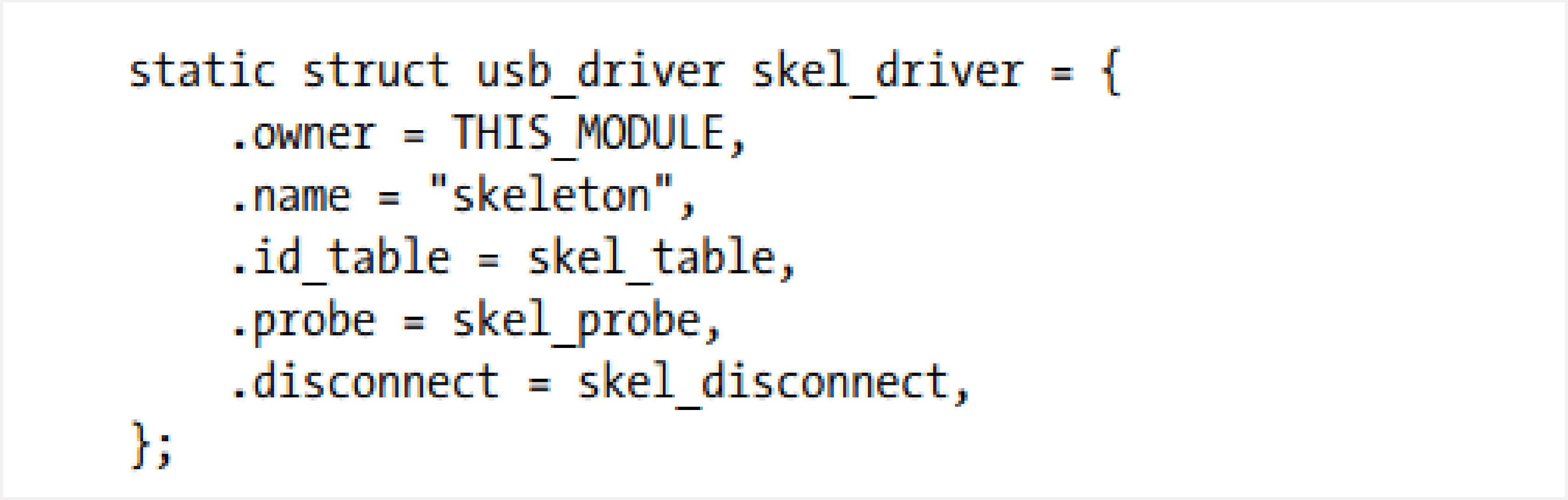
USB device driver deregistration

- When the USB driver is to be unloaded, the struct usb_driver needs to be unregistered from the kernel. This is done with a call to usb_deregister.
- When this call happens, any USB interfaces that were currently bound to this driver are disconnected and the disconnect function is called for them



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

usb_driver structure



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

struct module *owner

- Pointer to the module owner of this driver. The USB core uses it to properly reference count this USB device driver so that it is not unloaded at inopportune moments.

const char *name

- Pointer to the name of the driver

int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)

- Pointer to the probe function in the USB device driver. This function is called by the USB core when it thinks it has a struct usb_interface that this driver can handle.

usb_driver structure

void (*disconnect) (struct usb_interface *intf)

- Pointer to the disconnect function in the USB device driver. This function is called by the USB core when the struct usb_interface has been removed from the system or when the device driver is being unloaded from the USB core.

const struct usb_device_id *id_table

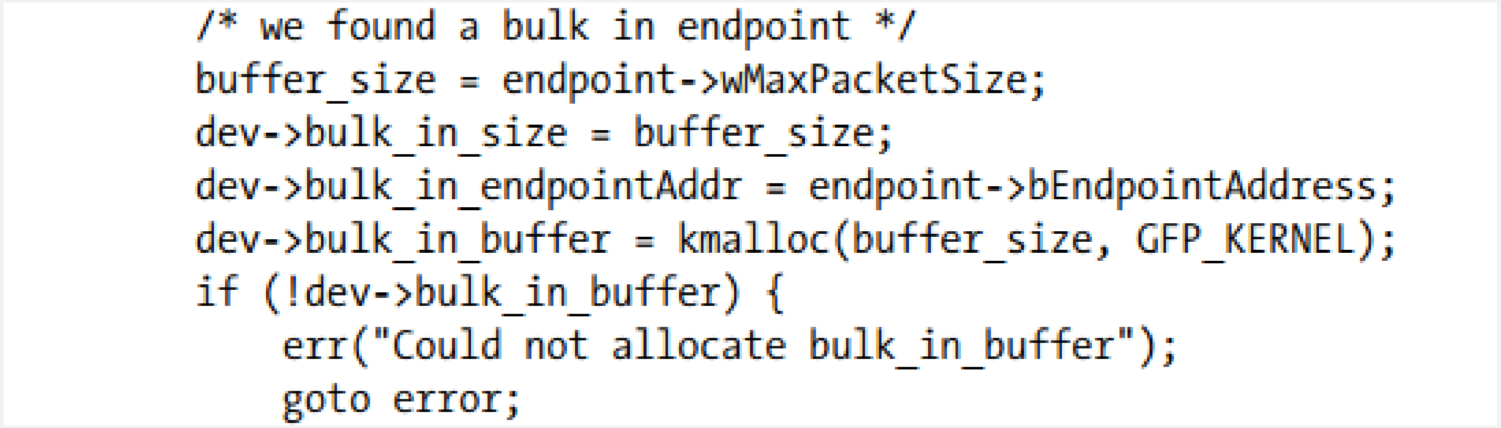
- Pointer to the struct usb_device_id table that contains a list of all of the different kinds of USB devices this device driver can accept. If this variable is not set, the probe function callback in the USB device driver is never called.

Probe() function

- The probe function is called when a USB device is installed that the USB core thinks this USB device driver should handle.
- The probe function should perform checks on the information passed to it about the USB device and decide whether the USB device driver is really appropriate for that USB device.
- USB device drivers usually want to detect what the endpoint address and buffer sizes are for the USB device, as they are needed in order to communicate with the device

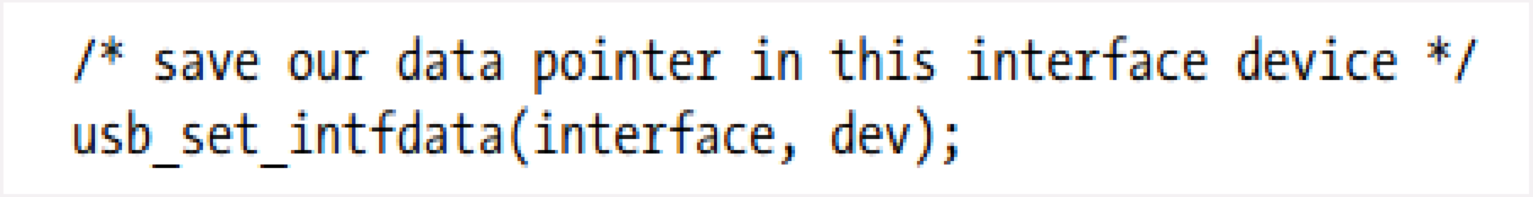
Probe() function

- If driver finds the proper type of endpoint and can save the information about the endpoint that it will later need to communicate over it in a local structure



```
/* we found a bulk in endpoint */
buffer_size = endpoint->wMaxPacketSize;
dev->bulk_in_size = buffer_size;
dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
if (!dev->bulk_in_buffer) {
    err("Could not allocate bulk_in_buffer");
    goto error;
```

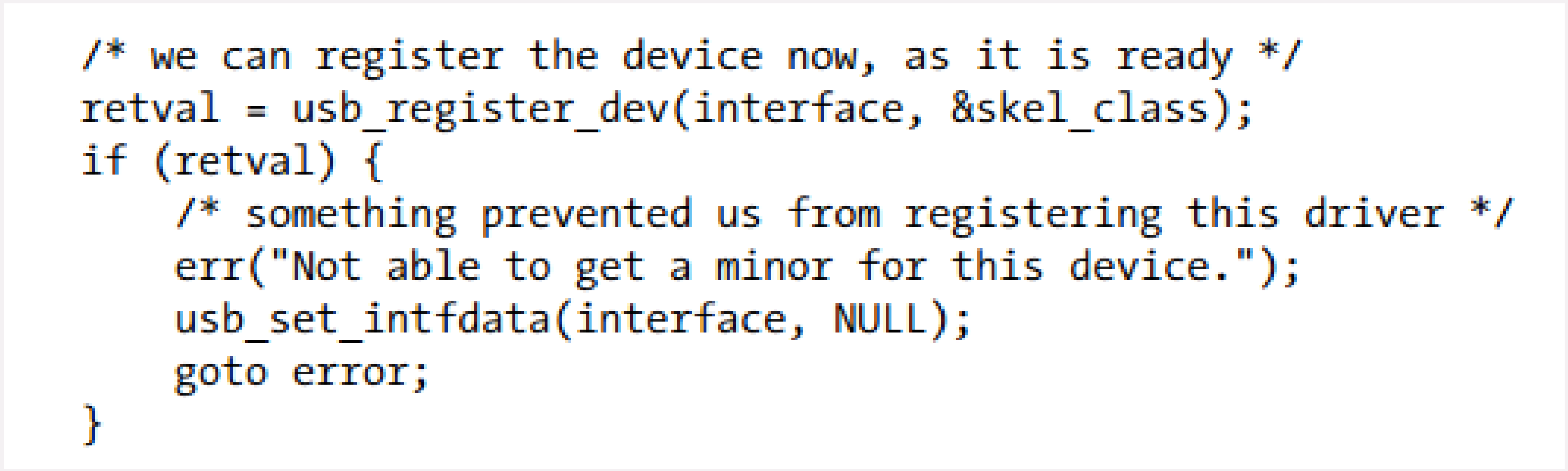
- This function accepts a pointer to any data type and saves it in the struct usb_interface structure for later access



```
/* save our data pointer in this interface device */
usb_set_intfdata(interface, dev);
```

Probe() function

- The USB device driver must call the usb_register_dev function in the probe function when it wants to register a device with the USB core



```
/* we can register the device now, as it is ready */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /* something prevented us from registering this driver */
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
```

Hot-pluggability

- Hot plugging is the ability to install a device without shutting down the host computer
- Hot plugging gets its name because devices are plugged while a computer is running - or hot. Such a device is referred to as being "hot-pluggable“
- Starting with kernel 2.4 (in January 2001), hotplugging became a standard feature of GNU/Linux. It's been backported to kernel 2.2 (for USB)

URB & Data Transfers

USB Request Block (URB)

- URB is used to send/receive data to/from a specific USB endpoints on a specific USB device in asynchronous manner.
- An URB consists of all relevant information to execute any USB transaction.
- Every endpoint can handle queues of URBs.
- A USB device driver may allocate many URBs for a single endpoint or may reuse a single URB for many different endpoints, depending on the need of the driver.

Structure of URB

```
struct urb
{
// (IN) device and pipe specify the endpoint queue
        struct usb_device *dev;         // pointer to associated USB device
        unsigned int pipe;              // endpoint information

        unsigned int transfer_flags;    // ISO_ASAP, SHORT_NOT_OK, etc.

// (IN) all urbs need completion routines
        void *context;                  // context for completion routine
        void (*complete)(struct urb *); // pointer to completion routine

// (OUT) status after each completion
        int status;                     // returned status

// (IN) buffer used for data transfers
        void *transfer_buffer;          // associated data buffer
        int transfer_buffer_length;     // data buffer length
        int number_of_packets;          // size of iso_frame_desc

// (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
        int actual_length;              // actual data buffer length

// (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
        unsigned char* setup_packet;    // setup packet (control only)

// Only for PERIODIC transfers (ISO, INTERRUPT)
    // (IN/OUT) start_frame is set unless ISO_ASAP isn't set
        int start_frame;                // start frame
        int interval;                   // polling interval

    // ISO only: packets are only "best effort"; each can have errors
        int error_count;                // number of errors
        struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

Types of URB – Interrupt & Bulk URB Initializers

- **Interrupt URB**

➢ 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);

- **Bulk URB**

➢ 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);

Types of URB – Control URB Initializer

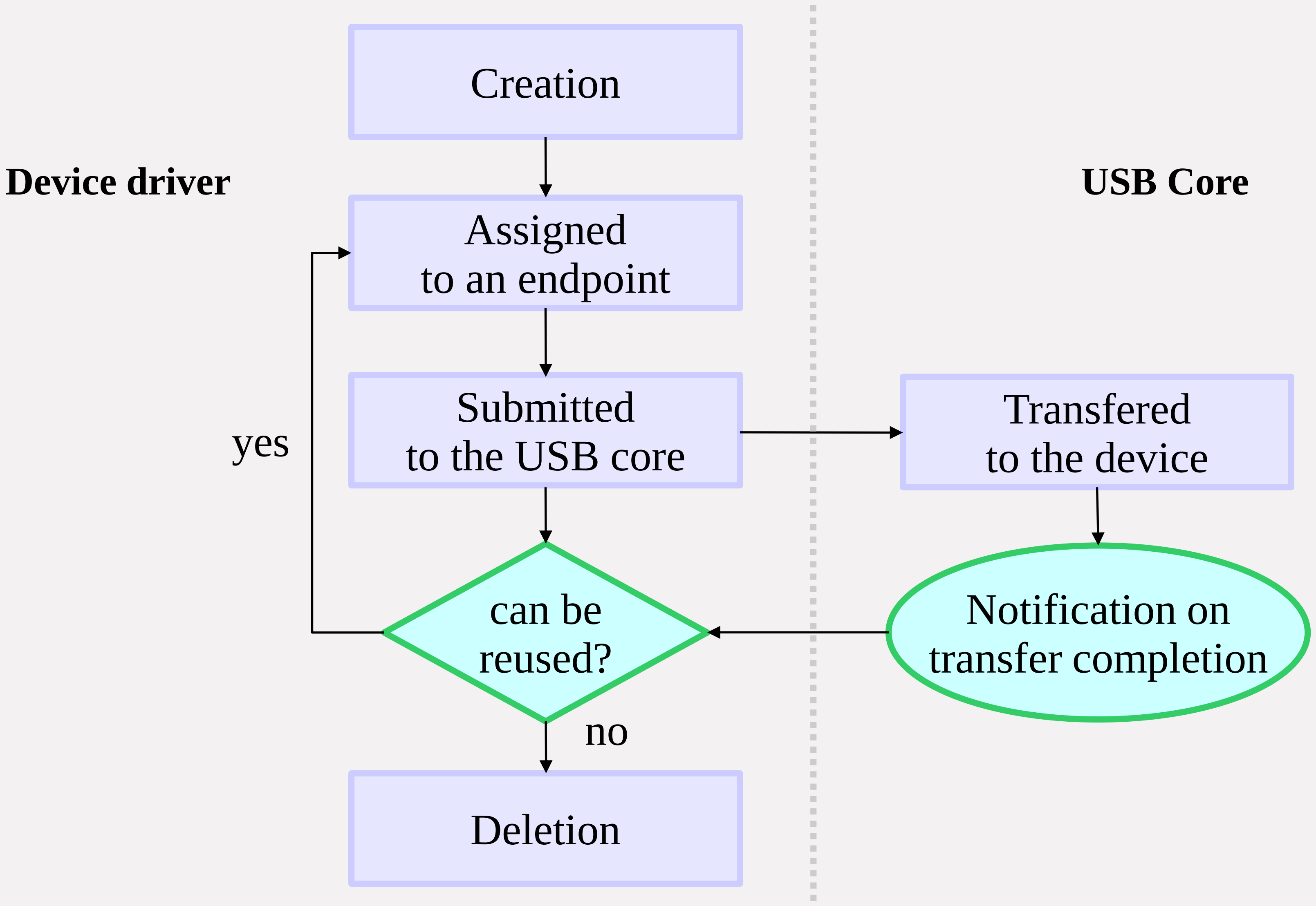
- <u>**Isochronous URB**</u>

➢ Isochronous URBs unfortunately do not have an initializer function like the interrupt, control, and bulk URBs do. So they must be initialized “by hand” in the device driver before they can be submitted to the USB core.

- <u>**Control URB**</u>

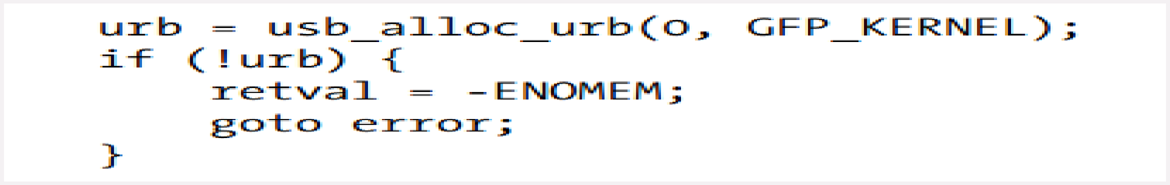
➢ void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, unsigned char *setup_packet, void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context);

Life Cycle of an URB



URB Allocation

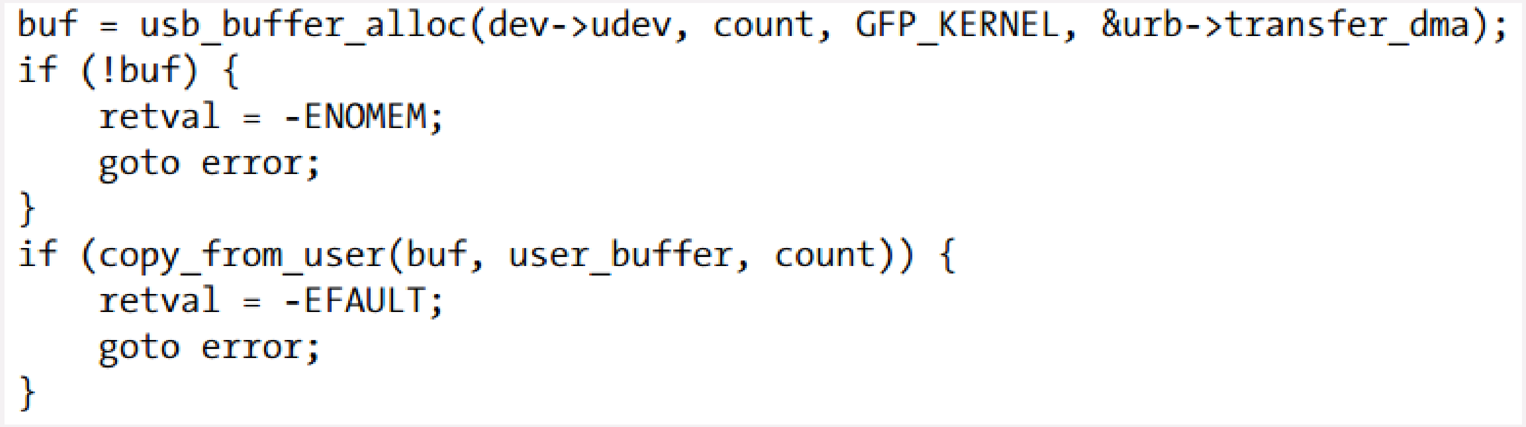
- An URB must be allocated for transmitting the data to the device



```
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
        retval = -ENOMEM;
        goto error;
}
```

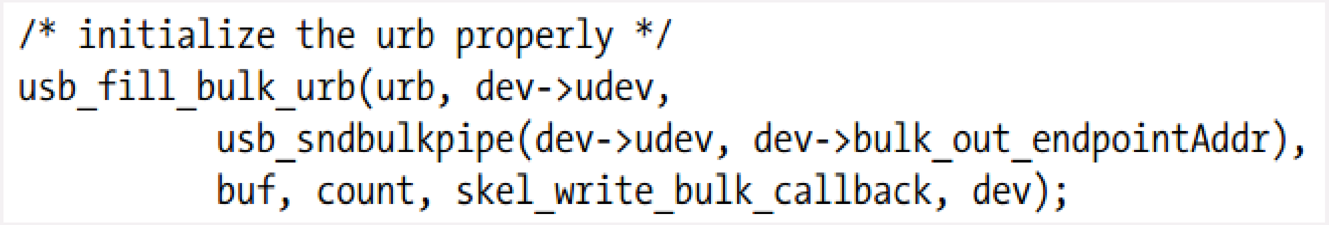
- DMA buffer should also be created to send the data to the device in the most efficient manner, and the data that is passed to the driver should be copied into that buffer

URB Buffer Allocation & URB Initialization



```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);
if (!buf) {
    retval = -ENOMEM;
    goto error;
}
if (copy_from_user(buf, user_buffer, count)) {
    retval = -EFAULT;
    goto error;
}
```

- Once the data is properly copied from the user space into the local buffer, the urb must be initialized correctly before it can be submitted to the USB core



```
/* initialize the urb properly */
usb_fill_bulk_urb(urb, dev->udev,
          usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
          buf, count, skel_write_bulk_callback, dev);
```

Submitting and Controlling an URB

- Now that the URB is properly allocated, the data is properly copied, and the URB is properly initialized, it can be submitted to the USB core to be transmitted to the device

```
/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
    err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
    goto error;
}
```

- After the URB is successfully transmitted to the USB device (or something happens in transmission), the URB callback is called by the USB core.

```
static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
{
    /* sync/async unlink faults aren't errors */
    if (urb->status &&
        !(urb->status == -ENOENT ||
          urb->status == -ECONNRESET ||
          urb->status == -ESHUTDOWN)) {
        dbg("%s - nonzero write bulk status received: %d",
            __FUNCTION__, urb->status);
    }

    /* free up our allocated buffer */
    usb_buffer_free(urb->dev, urb->transfer_buffer_length,
            urb->transfer_buffer, urb->transfer_dma);
}
```

Cancelling an URB

- int usb_kill_urb(struct urb *urb)
  - Urb lifecycle is stopped
  - Called on device disconnection (disconnect callback)
- int usb_unlink_urb(struct urb *urb)
  - To tell the USB core to stop an urb
  - Does not wait for urb to be fully stopped before returning to the caller

Logging / Debugging Overview

ETL (Event Trace Logs) Traces

- USB ETL Traces
  - Capture with scripts
  - ETL run in admin mode (Lets see the script)
- Analysis (Use Microsoft Message Analyzer)
  - Install from MSDN
- Analyzer
  - Sort with time stamp/ Message number
  - Logs grouping/ Filtering
  - Adding new columns (like EP etc.)

ETL Script

PCAP (Packet CAPture) Logs

- Install wireshark with USB PCAP support
  - Wireshark-win64-2.0.3.exe
  - Installation Requirement :
    WIN 7 (Service Pack 1 + Windows security patch “KB3033929”) or above
  - Run “C:\Program Files\Wireshark\extcap\USBPcapCMD.exe” as **admin**
- Analysis (Use Wireshark tool)
  - Install from Internet
- Analyzer
  - **Set time format**
  - Logs grouping/ Filtering

Analyzing USB X-fer Traces

- USB protocol Analyzer
  - Elisys USB explorer 280
  - LeCroy
  - MCCI Catena 1910 (Specially for HSIC) & MCCI Catena 2210 (for NCM)
  - Total Phase Beagle
- Scope Traces
  - Agilent MSO9254A

TEAMWORK MAKES THE DREAM WORK

Aricent®

Engineering Excellence. Sourced.

Queries

THANK YOU

Aricent®

Engineering Excellence. Sourced.