Associate Architect Program 종합과제

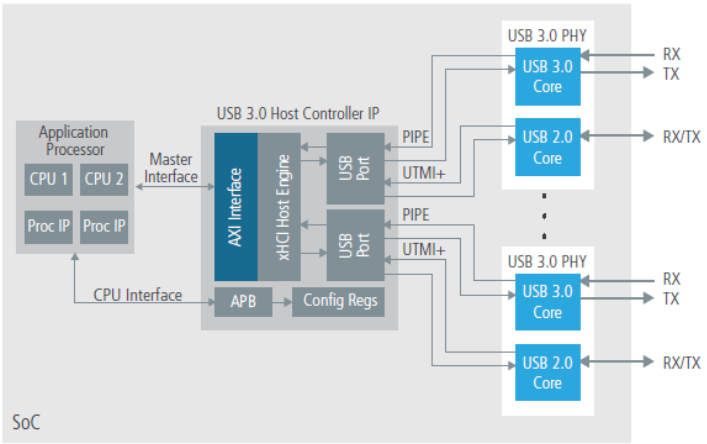
과제명: USB Host Driver 설계

2022년 C1차

주식회사 보이는 소프트웨어 연구소
http://www.bosornd.com

1. 시스템 개요

- **USB Host Driver 소프트웨어 구조를 설계하라.**



2. 기능적 요구사항

- **기본적인 기능**
  - USB 디바이스의 연결을 관리한다.
  - USB 디바이스의 제어 및 데이터 전송을 관리한다.
  - USB 디바이스의 전원을 관리한다.

- **부가적인 기능**
  - Mass Storage Class

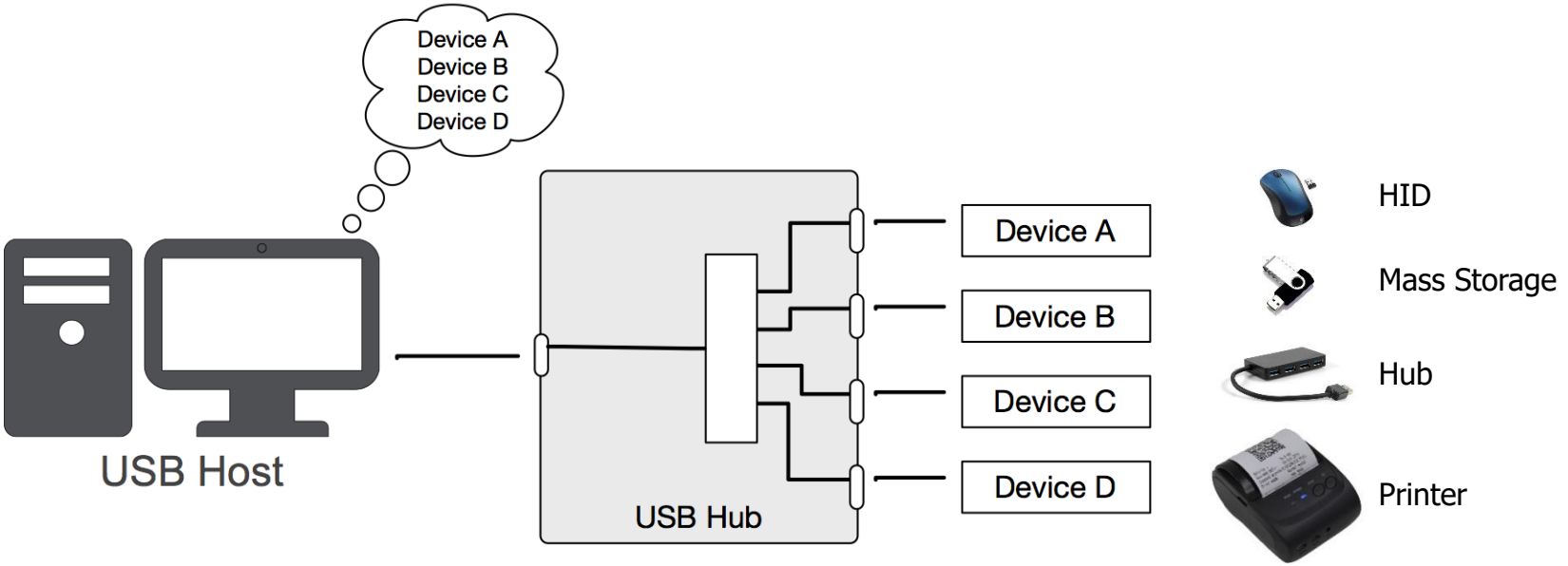
2. 기능적 요구사항

https://www.usb.org/defined-class-codes

USB defines class code information that is used to identify a device's functionality and to nominally load a device driver based on that functionality.

There is only one host in any USB system.

A root hub is integrated within the host to provide one or more attachment points.



2. 기능적 요구사항

https://www.usb.org/defined-class-codes

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Cont
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video

Base Class 08h (Mass Storage)

This base class is defined for devices that conform to the Mass Storage Device Class Specification found on the USB-IF website. That specification defines the usable set of SubClass and Protocol values. Values outside of that defined spec are reserved. These class codes can only be used in Interface Descriptors.

Base Class	SubClass	Protocol	Meaning
08h	xxh	xxh	Mass Storage device

Base Class 09h (Hub)

This base class is defined for devices that are USB hubs and conform to the definition in the USB specification. That specification defines the complete triples as shown below. All other values are reserved. These class codes can only be used in Device Descriptors.

Base Class	SubClass	Protocol	Meaning
09h	00h	00h	Full speed Hub
		01h	Hi-speed hub with single TT
		02h	Hi-speed hub with multiple TTs

3. 품질 요구사항

- **성능**
  - 디바이스의 연결, 제어 등의 성능이 빠를 수록 좋다.
- **변경 용이성 / 확장성**
  - 하드웨어 사양의 변경이 용이할 수록 좋다.
  - 다양한 클래스로의 확장이 용이할 수록 좋다.

4. 과제 환경에 대한 설정

- **시스템의 동작 및 사업 환경은 각자 설정한다.**
  - 어떤 기능 및 품질 측면에서 장점이 있는지 등.

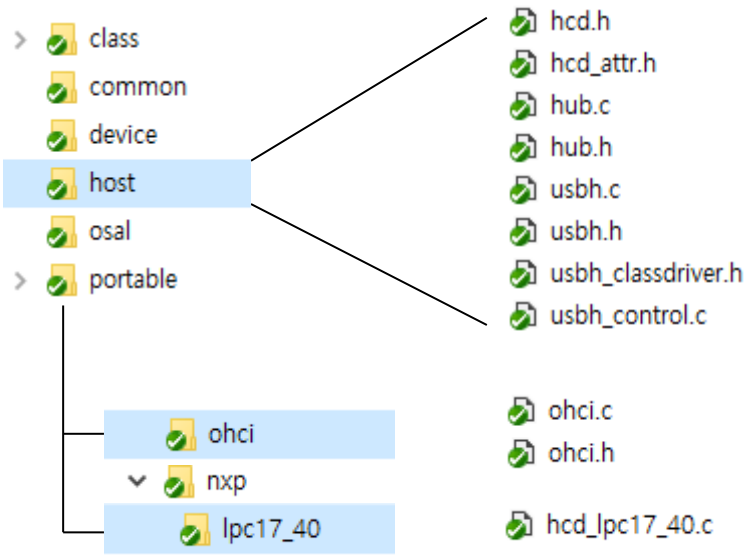
- **과제의 제약 사항에 대하여,**
  - 솔루션이 최적화된 것을 판단할 수 있는 품질 시나리오가 검토되어야 한다.
  - 제약 사항의 변경에 대해서도 검토되어야 한다.

APPENDIX

TinyUSB

https://www.tinyusb.org/

tiny is an open-source cross-platform USB Host/Device stack for embedded systems, designed to be memory-safe with no dynamic allocation and thread-safe with all interrupt events being deferred and then handled in the non-ISR task function.



Key Features

- ARM Cortex-M3 core
  - Up to 120 MHz operation
- Serial Peripherals
  - 10/100 Ethernet MAC
  - USB 2.0 full-speed device/Host/ OTG controller with on-chip PHY

Host Controller Interface

https://en.wikipedia.org/wiki/Host_controller_interface_(USB,_Firewire)

- **Standard Host Controller Interface for USB**
  - **Open Host Controller Interface** is a register-level description of a Host Controller for USB 1.1.
  - **Universal Host Controller Interface** is a proprietary interface created by Intel for USB 1.x (full and low speeds). It requires a license from Intel. A USB controller using UHCI does little in hardware and requires a software UHCI driver to do much of the work of managing the USB bus.
  - **Enhanced Host Controller Interface** for USB 2.0 is a high-speed controller standard applicable to USB 2.0. UHCI- and OHCI-based systems, as existed previously, entailed greater complexity and costs than necessary.
  - **Extensible Host Controller Interface** is the newest host controller standard that improves speed, power efficiency and virtualization over its predecessors. It supports all USB device speeds (USB 3.1 SuperSpeed+, USB 3.0 SuperSpeed, USB 2.0 Low-, Full-, and High-speed, USB 1.1 Low- and Full-speed).

USB 2.0

https://www.usb.org/document-library/usb-20-specification

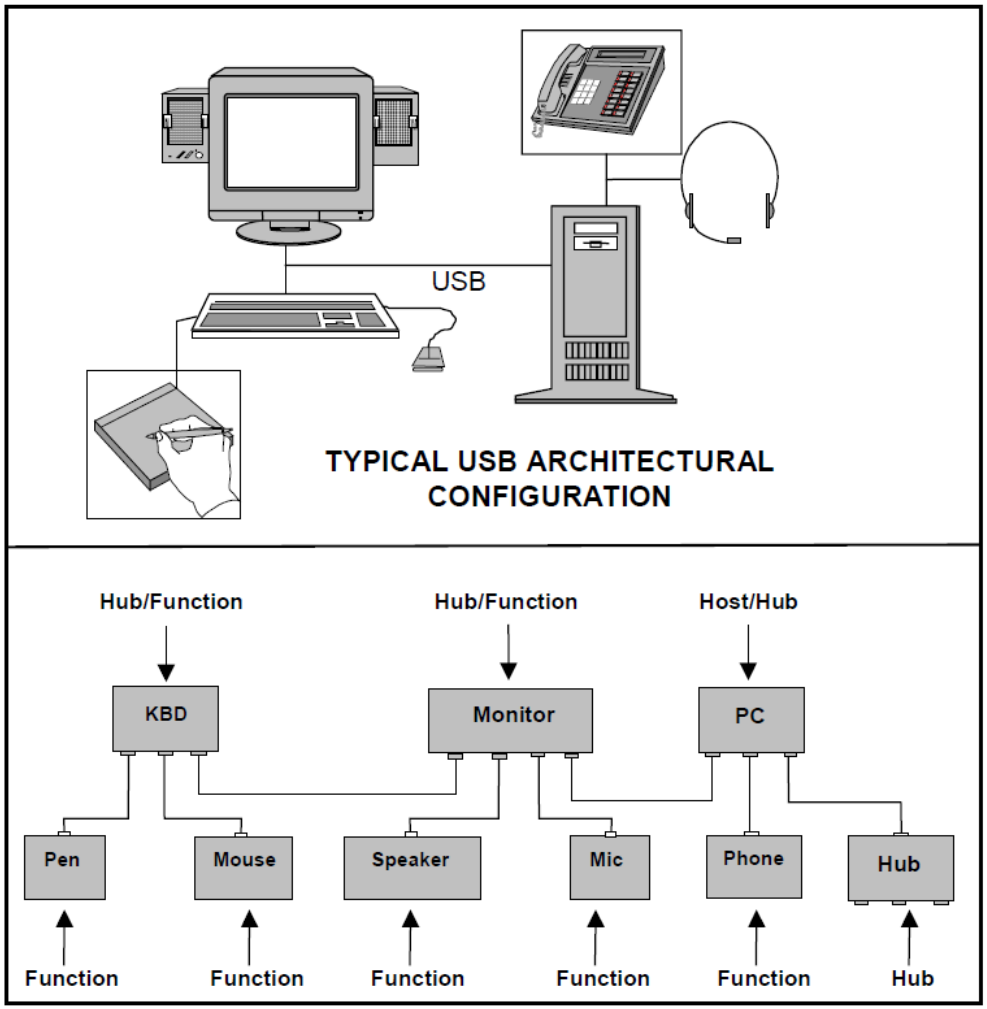


Figure 4-4. Hubs in a Desktop Computer Environment

There is only one host in any USB system.

The USB interface to the host computer system is referred to as the Host Controller.

The Host Controller may be implemented in a combination of hardware, firmware, or software.

A root hub is integrated within the host system to provide one or more attachment points.

4.9 USB Host: Hardware and Software

The USB host interacts with USB devices through the Host Controller. The host is responsible for the following:

- Detecting the attachment and removal of USB devices
- Managing control flow between the host and USB devices
- Managing data flow between the host and USB devices
- Collecting status and activity statistics
- Providing power to attached USB devices

The USB System Software on the host manages interactions between USB devices and host-based device software. There are five areas of interactions between the USB System Software and device software:

- Device enumeration and configuration
- Isochronous data transfers
- Asynchronous data transfers
- Power management
- Device and bus management information

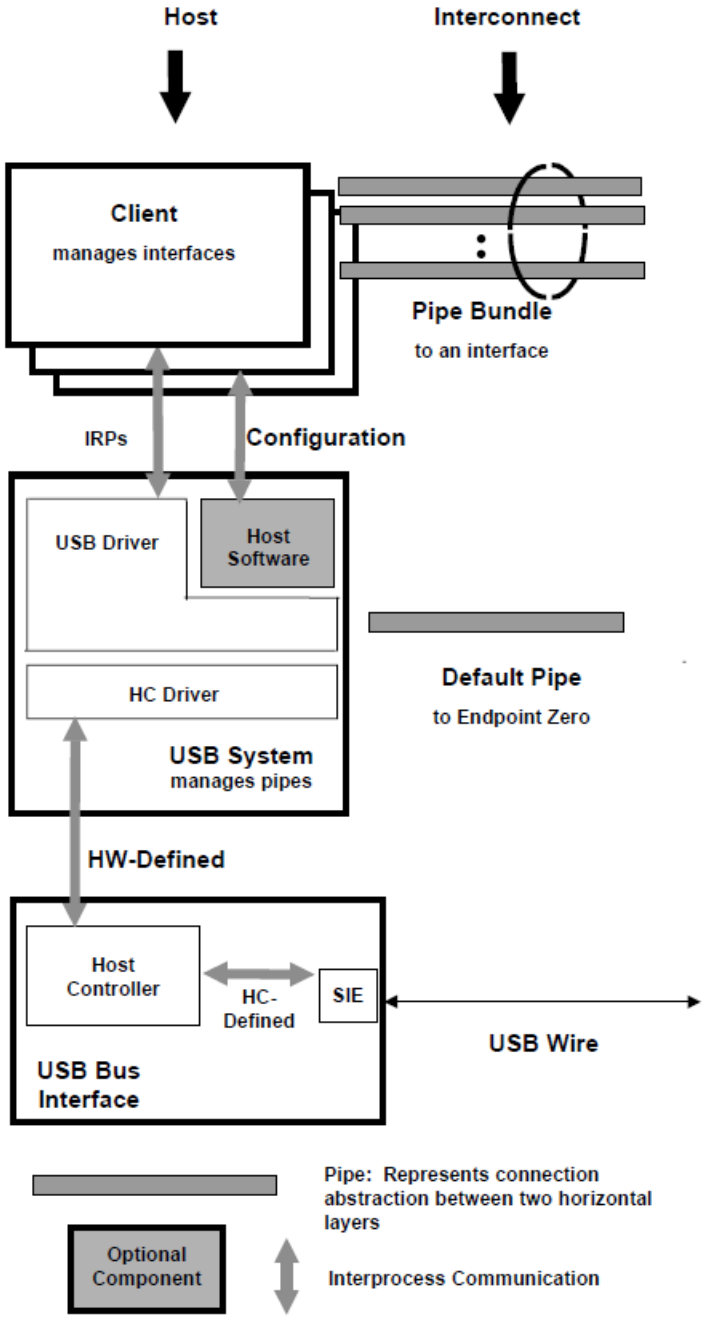


Figure 10-2. Host Communications

There is only one host for each USB.

The major layers of a host consist of the following:

- USB bus interface
- USB System
- Client

The USB bus interface handles interactions for the electrical and protocol layers. From the interconnect point of view, a similar USB bus interface is provided by both the USB device and the host, as exemplified by the Serial Interface Engine (SIE). On the host, however, the USB bus interface has additional responsibilities due to the unique role of the host on the USB and is implemented as the Host Controller. The Host Controller has an integrated root hub providing attachment points to the USB wire.

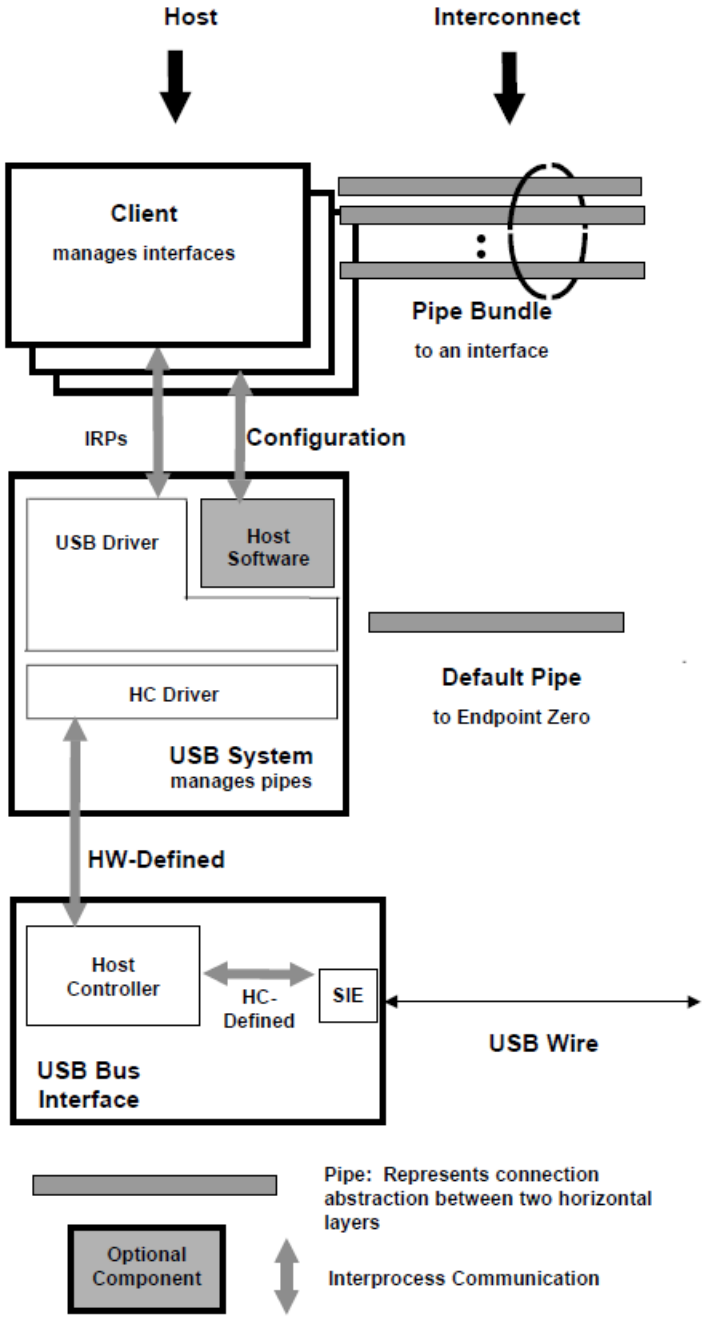


Figure 10-2. Host Communications

The USB System uses the Host Controller to manage data transfers between the host and USB devices.

The interface between the USB System and the Host Controller is dependent on the hardware definition of the Host Controller. The USB System, in concert with the Host Controller, performs the translation between the client's view of data transfers and the USB transactions appearing on the interconnect. This includes the addition of any USB feature support such as protocol wrappers. The USB System is also responsible for managing USB resources, such as bandwidth and bus power, so that client access to the USB is possible.

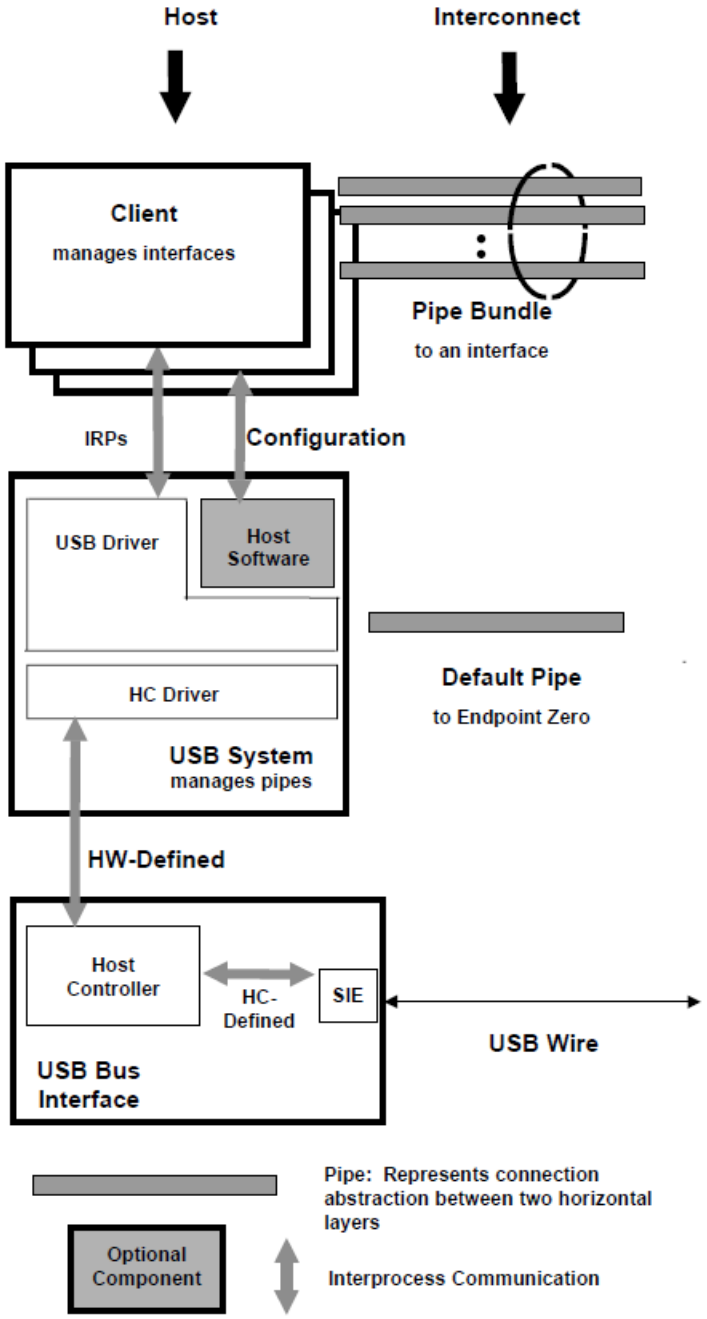


Figure 10-2. Host Communications

The USB System has three basic components:

- Host Controller Driver
- USB Driver
- Host Software

The Host Controller Driver (HCD) exists to more easily map the various Host Controller implementations into the USB System, such that a client can interact with its device without knowing to which Host Controller the device is connected. The USB Driver (USBD) provides the basic host interface (USBDI) for clients to USB devices. The interface between the HCD and the USBD is known as the Host Controller Driver Interface (HCDI). This interface is never available directly to clients and thus is not defined by the USB Specification. A particular HCDI is, however, defined by each operating system that supports various Host Controller implementations.

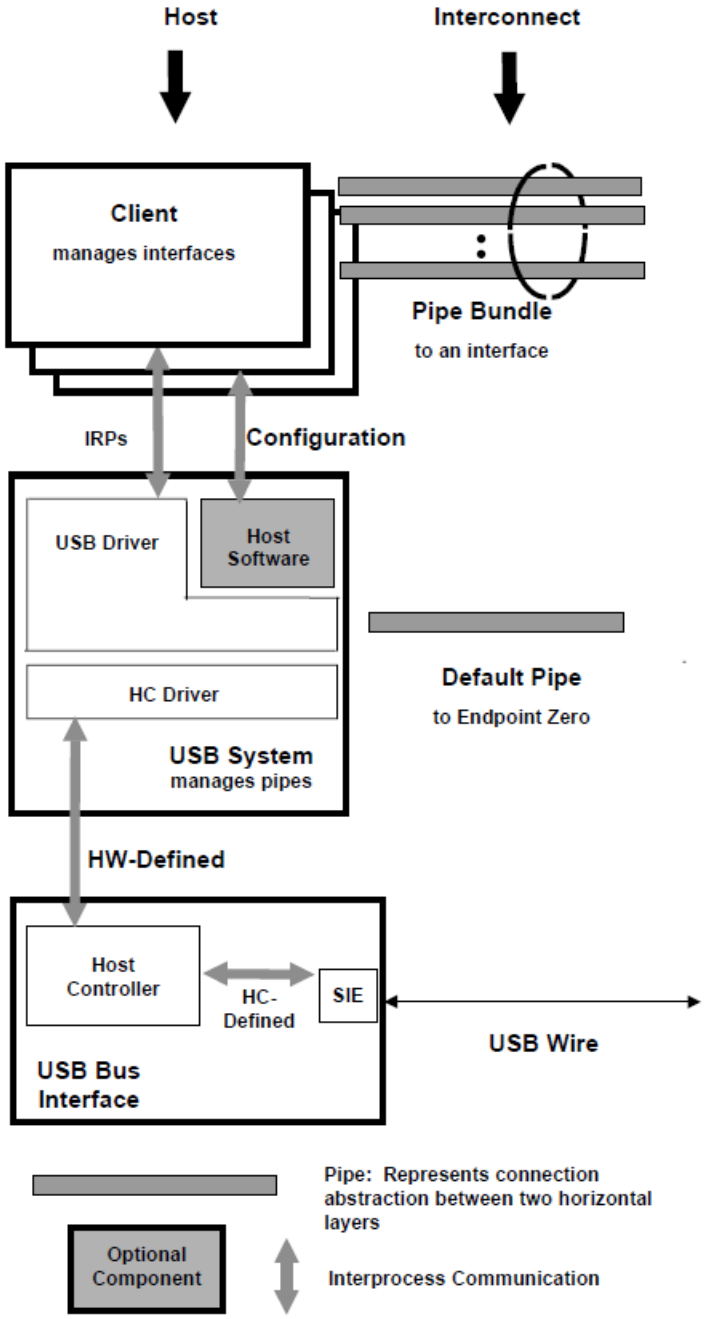


Figure 10-2. Host Communications

The USBD provides data transfer mechanisms in the form of I/O Request Packets (IRPs), which consist of a request to transport data across a specific pipe. In addition to providing data transfer mechanisms, the USBD is responsible for presenting to its clients an abstraction of a USB device that can be manipulated for configuration and state management. As part of this abstraction, the USBD owns the default pipe through which all USB devices are accessed for the purposes of standard USB control. This default pipe represents a logical communication between the USBD and the abstraction of a USB device.

In some operating systems, additional non-USB System Software is available that provides configuration and loading mechanisms to device drivers. In such operating systems, the device driver shall use the provided interfaces instead of directly accessing the USBDI mechanisms.

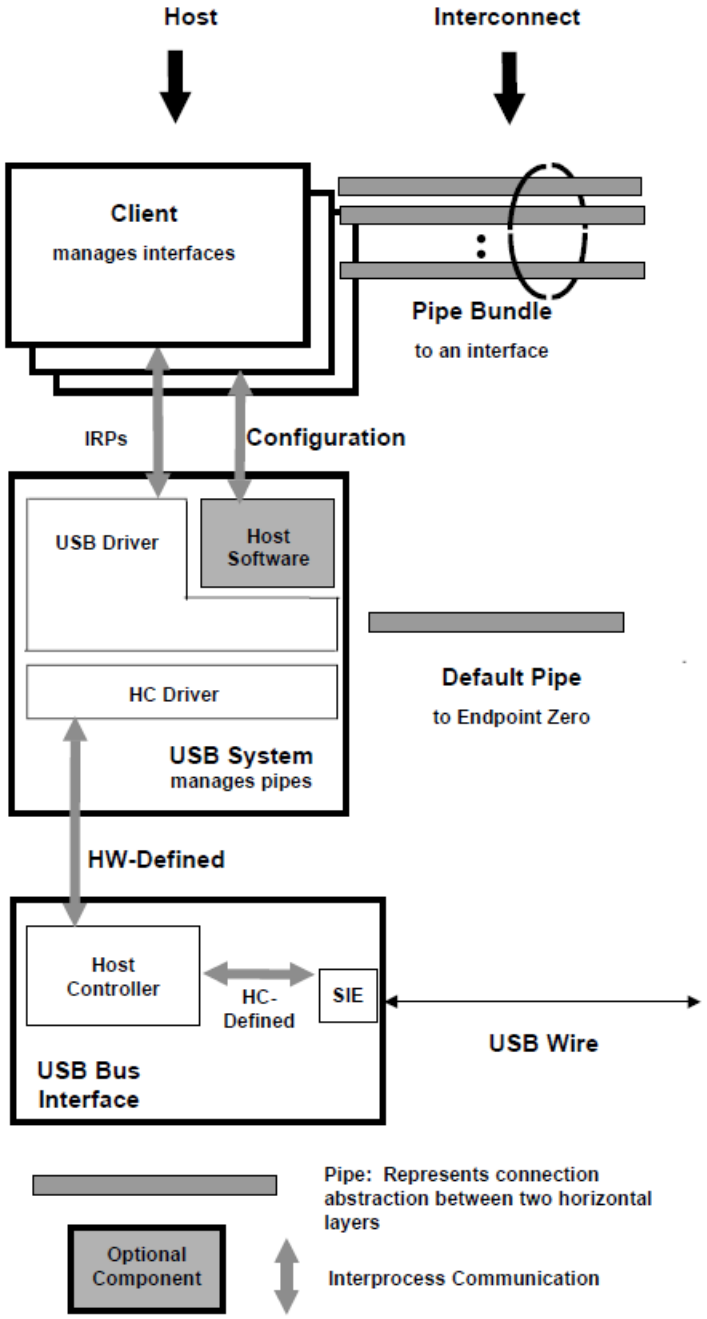
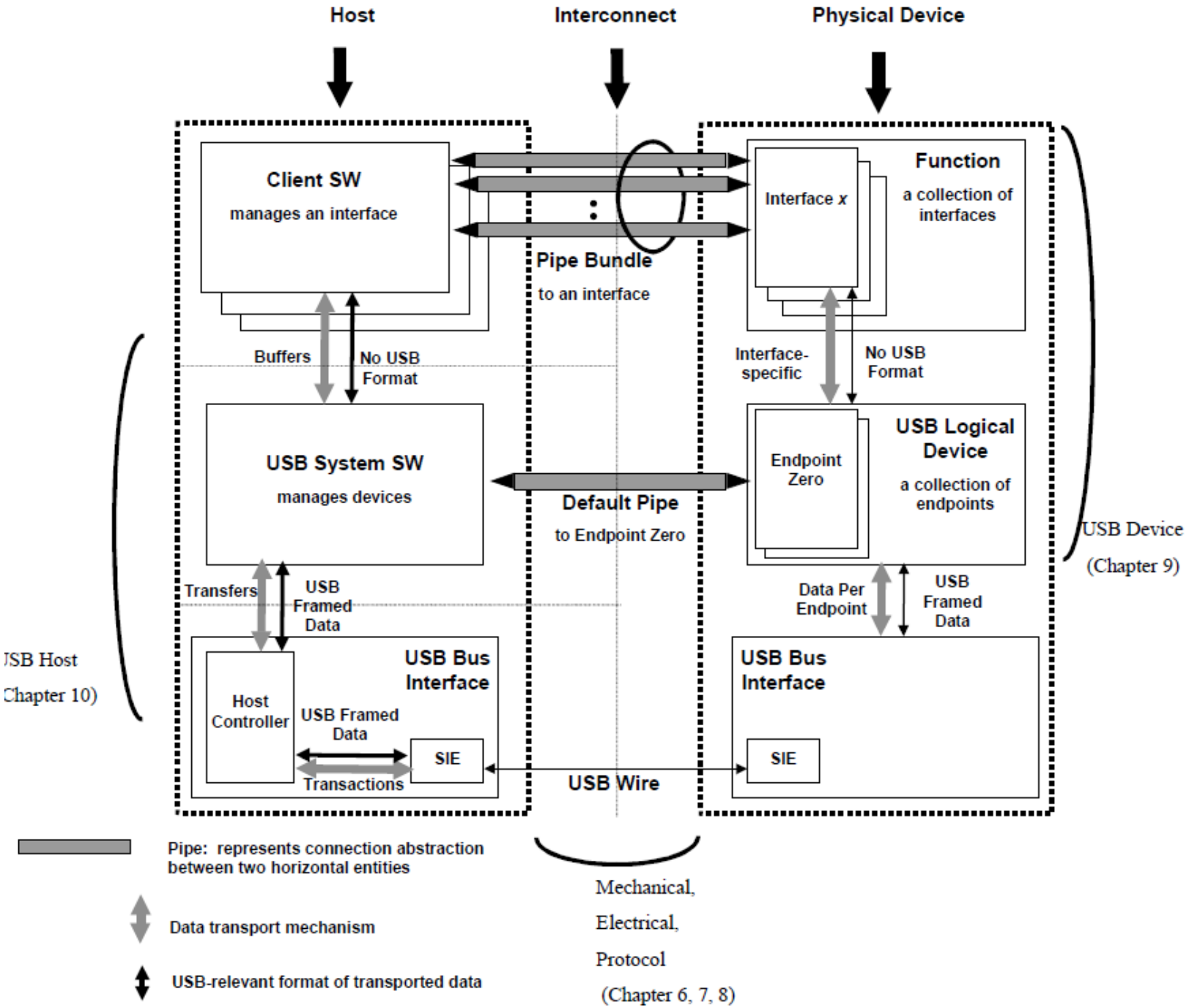


Figure 10-2. Host Communications

The client layer describes all the software entities that are responsible for directly interacting with USB devices. When each device is attached to the system, these clients might interact directly with the peripheral hardware. The shared characteristics of the USB place USB System Software between the client and its device; that is, a client cannot directly access the device's hardware.

Overall, the host layers provide the following capabilities:

- Detecting the attachment and removal of USB devices
- Managing USB standard control flow between the host and USB devices
- Managing data flow between the host and USB devices
- Collecting status and activity statistics
- Controlling the electrical interface between the Host Controller and USB devices, including the provision of a limited amount of power


Host
Interconnect
Physical Device
Client SW
manages an interface
Pipe Bundle
to an interface
Function
a collection of interfaces
Interface x
Buffers
No USB Format
Interface-specific
No USB Format
USB System SW
manages devices
Default Pipe
to Endpoint Zero
Endpoint Zero
USB Logical Device
a collection of endpoints
USB Device
(Chapter 9)
Transfers
USB Framed Data
Data Per Endpoint
USB Framed Data
USB Host
Chapter 10)
USB Bus Interface
Host Controller
USB Framed Data
SIE
Transactions
USB Wire
USB Bus Interface
SIE
Pipe: represents connection abstraction between two horizontal entities
Data transport mechanism
USB-relevant format of transported data
Mechanical,
Electrical,
Protocol
(Chapter 6, 7, 8)

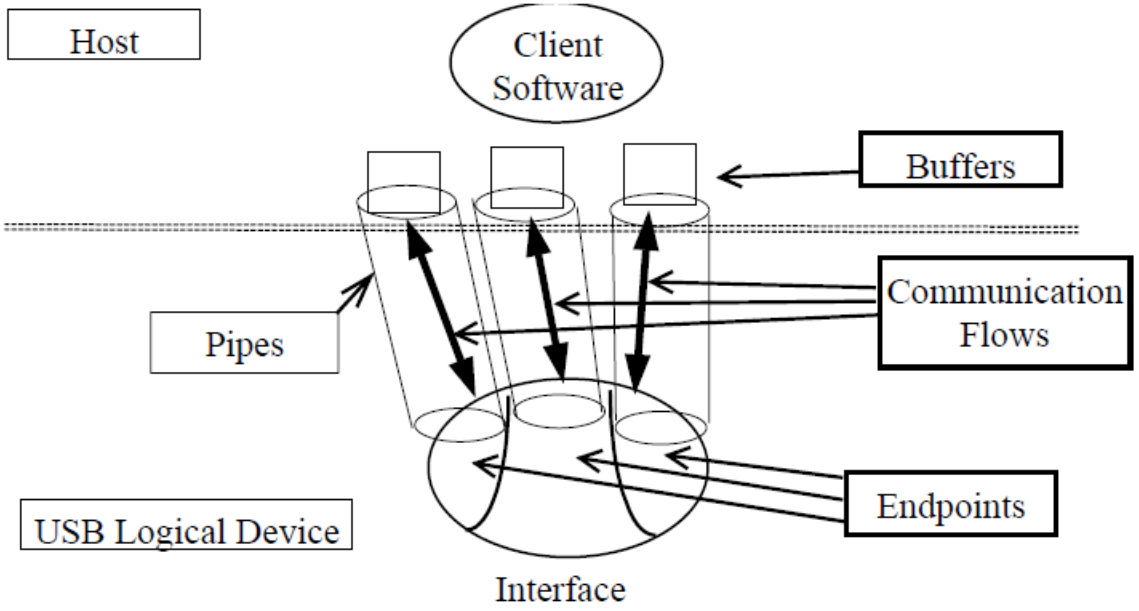


Figure 5-10. USB Communication Flow

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device. Each USB logical device is composed of a collection of independent endpoints.

An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software.

An endpoint describes itself by:

- Bus access frequency/latency requirement
- Bandwidth requirement
- Endpoint number
- Error handling behavior requirements
- Maximum packet size that the endpoint is capable of sending or receiving
- The transfer type for the endpoint (refer to Section 5.4 for details)
- The direction in which data is transferred between the endpoint and the host

Endpoints other than those with endpoint number zero are in an unknown state before being configured and may not be accessed by the host before being configured.

A USB pipe is an association between an endpoint on a device and software on the host. Pipes represent the ability to move data between software on the host via a memory buffer and an endpoint on a device. There are two mutually exclusive pipe communication modes:

• Stream: Data moving through a pipe has no USB-defined structure

• Message: Data moving through a pipe has some USB-defined structure

The USB does not interpret the content of data it delivers through a pipe. Even though a message pipe requires that data be structured according to USB definitions, the content of the data is not interpreted by the USB.

Additionally, pipes have the following associated with them:

• A claim on USB bus access and bandwidth usage.

• A transfer type.

• The associated endpoint's characteristics, such as directionality and maximum data payload sizes. The data payload is the data that is carried in the data field of a data packet within a bus transaction (as defined in Chapter 8).

The pipe that consists of the two endpoints with endpoint number zero is called the Default Control Pipe. The USB System Software retains "ownership" of the Default Control Pipe and mediates use of the pipe by other client software. The USB System Software ensures that multiple requests are not sent to a message pipe concurrently.

The USB defines four transfer types:

• Control Transfers: Bursty, non-periodic, host software-initiated request/response communication, typically used for command/status operations.

• Isochronous Transfers: Periodic, continuous communication between host and device, typically used for time-relevant information. This transfer type also preserves the concept of time encapsulated in the data. This does not imply, however, that the delivery needs of such data is always time-critical.

• Interrupt Transfers: Low-frequency, bounded-latency communication.

• Bulk Transfers: Non-periodic, large-packet bursty communication, typically used for data that can use any available bandwidth and can also be delayed until bandwidth is available.

USB 2.0

https://www.usb.org/document-library/usb-20-specification

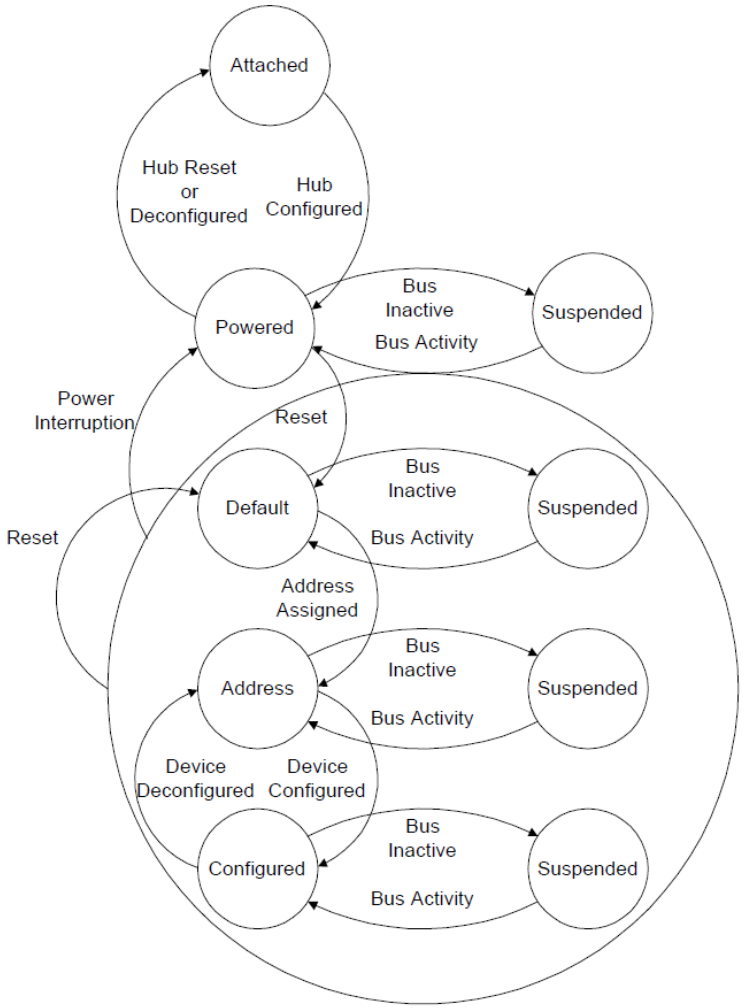


Figure 9-1. Device State Diagram

Before a USB device's function may be used, the device must be configured. The host is responsible for configuring a USB device.

In order to conserve power, USB devices automatically enter the Suspended state when the device has observed no bus traffic for a specified period. When suspended, the USB device maintains any internal status, including its address and configuration.

9.1.2 Bus Enumeration

When a USB device is attached to or removed from the USB, the host uses a process known as bus enumeration to identify and manage the device state changes necessary. When a USB device is attached to a powered port, the following actions are taken:

1. The hub to which the USB device is now attached informs the host of the event via a reply on its status change pipe.
   At this point, the USB device is in the Powered state and the port to which it is attached is disabled.
2. The host determines the exact nature of the change by querying the hub.
3. Now that the host knows the port to which the new device has been attached, the host then waits for at least 100ms to allow completion of an insertion process and for power at the device to become stable. The host then issues a port enable and reset command to that port.
4. The hub performs the required reset processing for that port. When the reset signal is released, the port has been enabled.
   The USB device is now in the Default state and can draw no more than 100mA from VBUS. All of its registers and state have been reset and it answers to the default address.
5. The host assigns a unique address to the USB device, moving the device to the Address state.
6. Before the USB device receives a unique address, its Default Control Pipe is still accessible via the default address.
   The host reads the device descriptor to determine what actual maximum data payload size this USB device's default pipe can use.
7. The host reads the configuration information from the device by reading each configuration zero to n-1, where n is the number of configurations. This process may take several milliseconds to complete.
8. Based on the configuration information and how the USB device will be used, the host assigns a configuration value to the device.
   The device is now in the Configured state and all the endpoints in this configuration have taken on their described characteristics.
   The USB device may now draw the amount of VBUS power described in its descriptor for the selected configuration.
   From the device's point of view, it is now ready for use.

When the USB device is removed, the hub again sends a notification to the host. Detaching a device disables the port to which it had been attached. Upon receiving the detach notification, the host will update its local topological information.

9.2.3 Configuration

A USB device must be configured before its function(s) may be used. The host is responsible for configuring a USB device. The host typically requests configuration information from the USB device to determine the device's capabilities.

Within a single configuration, a device may support multiple interfaces. An interface is a related set of endpoints that present a single feature or function of the device to the host. The protocol used to communicate with this related set of endpoints and the purpose of each endpoint within the interface may be specified as part of a device class or vendor-specific definition. In addition, an interface within a configuration may have alternate settings that redefine the number or characteristics of the associated endpoints.

In support of adaptive device drivers that are capable of managing a related group of USB devices, the device and interface descriptors contain Class, SubClass, and Protocol fields. These fields are used to identify the function(s) provided by a USB device and the protocols used to communicate with the function(s) on the device.

main

```
int main(void)
{
  application_init();
  tusb_init();

  while(1) // the mainloop
  {
    application_code();
    tuh_task(); // tinyusb host task
  }
}
```

usbh

```
void tuh_task(void)
{
  // Skip if stack is not initialized
  if ( !tusb_inited() ) return;

  // Loop until there is no more events in the queue
  while (1)
  {
    hcd_event_t event;
    if ( !osal_queue_receive(_usbh_q, &event) ) return;

    switch (event.event_id)
    {
      case HCD_EVENT_DEVICE_ATTACH:
          enum_new_device(&event);
      break;
      // …
    }
  }
}
```

usbh

```
static bool enum_new_device(hcd_event_t* event)
{
```

ohci

```
bool hcd_port_connect_status(uint8_t hostid)
{
  (void) hostid;
  return OHCI_REG->rhport_status_bit[0].current_connect_status;
}
```

ohci

```
tusb_speed_t hcd_port_speed_get(uint8_t hostid)
{
  (void) hostid;
  return OHCI_REG->rhport_status_bit[0].low_speed_device_attached
         ? TUSB_SPEED_LOW : TUSB_SPEED_FULL;
}
```

usbh

```
static bool enum_new_device(hcd_event_t* event)
{
  _dev0.rhport   = event->rhport;
  _dev0.hub_addr = event->connection.hub_addr;
  _dev0.hub_port = event->connection.hub_port;

  //--- connected/disconnected directly with roothub ---//
  if (_dev0.hub_addr == 0)
  {
    // wait until device is stable TODO non blocking
    osal_task_delay(RESET_DELAY);

    // device unplugged while delaying
    if ( !hcd_port_connect_status(_dev0.rhport) ) return true;

    _dev0.speed = hcd_port_speed_get(_dev0.rhport );

    enum_request_addr0_device_desc();
  }
  // ...
  return true;
}
```

usbh

```
static bool enum_request_addr0_device_desc(void)
```

usbh_control

```
bool tuh_control_xfer (uint8_t dev_addr,
         tusb_control_request_t const* request,
         void* buffer, tuh_control_complete_cb_t complete_cb)
{
  const uint8_t rhport = usbh_get_rhport(dev_addr);

  _ctrl_xfer.request     = (*request);
  _ctrl_xfer.buffer      = buffer;
  _ctrl_xfer.stage       = STAGE_SETUP;
  _ctrl_xfer.complete_cb = complete_cb;

  TU_ASSERT( hcd_setup_send(rhport, dev_addr,
              (uint8_t const*) &_ctrl_xfer.request) );

  return true;
}
```

usbh

```
static bool enum_request_addr0_device_desc(void)
{
  uint8_t const addr0 = 0;
  TU_ASSERT( usbh_edpt_control_open(addr0, 8) );

  // Get device descriptor to get Control Endpoint Size
  TU_LOG2("Get 8 byte of Device DescriptorWrWn");
  tusb_control_request_t const request =
  {
    .bmRequestType_bit =
    {
      .recipient = TUSB_REQ_RCPT_DEVICE,
      .type      = TUSB_REQ_TYPE_STANDARD,
      .direction = TUSB_DIR_IN
    },
    .bRequest = TUSB_REQ_GET_DESCRIPTOR,
    .wValue   = TUSB_DESC_DEVICE << 8,
    .wIndex   = 0,
    .wLength  = 8
  };
  TU_ASSERT( tuh_control_xfer(addr0, &request, _usbh_ctrl_buf,
                              enum_get_addr0_device_desc_complete) );

  return true;
}
```

usbh_control

```
bool tuh_control_xfer (uint8_t dev_addr,
         tusb_control_request_t const* request,
         void* buffer, tuh_control_complete_cb_t complete_cb)
{
  const uint8_t rhport = usbh_get_rhport(dev_addr);

  _ctrl_xfer.request     = (*request);
  _ctrl_xfer.buffer      = buffer;
  _ctrl_xfer.stage       = STAGE_SETUP;
  _ctrl_xfer.complete_cb = complete_cb;

  TU_ASSERT( hcd_setup_send(rhport, dev_addr,
              (uint8_t const*) &_ctrl_xfer.request) );

  return true;
}
```

ohci

```
bool hcd_setup_send(uint8_t rhport, uint8_t dev_addr,
uint8_t const setup_packet[8])
{
  (void) rhport;

  ohci_ed_t* ed   = &ohci_data.control[dev_addr].ed;
  ohci_gtd_t *qtd = &ohci_data.control[dev_addr].gtd;

  gtd_init(qtd, (uint8_t*) setup_packet, 8);
  qtd->index           = dev_addr;
  qtd->pid             = PID_SETUP;
  qtd->data_toggle     = GTD_DT_DATA0;
  qtd->delay_interrupt = 0;

  ed->td_head.address = (uint32_t) qtd;

  OHCI_REG->command_status_bit.control_list_filled = 1;

  return true;
}
```

<<hardware>>
Host Controller

bsp

```
void USB0_IRQHandler(void)
{
    tuh_int_handler(0);  // hcd_int_handler
}
```

usbh

```
void hcd_event_xfer_complete(uint8_t dev_addr, uint8_t
ep_addr, uint32_t xferred_bytes, xfer_result_t result, bool
in_isr)
{
  hcd_event_t event =
  {
    .rhport   = 0, // TODO correct rhport
    .event_id = HCD_EVENT_XFER_COMPLETE,
    .dev_addr = dev_addr,
    .xfer_complete =
    {
      .ep_addr = ep_addr,
      .result  = result,
      .len     = xferred_bytes
    }
  };

  hcd_event_handler(&event, in_isr);
}
```

ohci

```
void hcd_int_handler(uint8_t hostid)
{
  // …

  if (int_status & OHCI_INT_WRITEBACK_DONEHEAD_MASK)
  {
    done_queue_isr(hostid);
  }

  OHCI_REG->interrupt_status = int_status;
}

static void done_queue_isr(uint8_t hostid)
{
  // done head is written in reversed order of completion
  ohci_td_item_t* td_head = list_reverse (
   (ohci_td_item_t*) tu_align16(ohci_data.hcca.done_head) );

  while( td_head != NULL )
  {
      // …
      hcd_event_xfer_complete(ed->dev_addr, tu_edpt_addr(
         ed->ep_number, dir), xferred_bytes, event, true);
    }

    td_head = (ohci_td_item_t*) td_head->next;
  }
```

usbh

```
void hcd_event_xfer_complete(uint8_t dev_addr, uint8_t
ep_addr, uint32_t xferred_bytes, xfer_result_t result, bool
in_isr)
{
  hcd_event_t event =
  {
    .rhport   = 0, // TODO correct rhport
    .event_id = HCD_EVENT_XFER_COMPLETE,
    .dev_addr = dev_addr,
    .xfer_complete =
    {
      .ep_addr = ep_addr,
      .result  = result,
      .len     = xferred_bytes
    }
  };

  hcd_event_handler(&event, in_isr);
}
```

usbh

```
void hcd_event_handler(hcd_event_t const* event, bool in_isr)
{
  switch (event->event_id)
  {
    default:
      osal_queue_send(_usbh_q, event, in_isr);
    break;
  }
}
```

usbh

```
void tuh_task(void)
{
  // Skip if stack is not initialized
  if ( !tusb_inited() ) return;

  // Loop until there is no more events in the queue
  while (1)
  {
    hcd_event_t event;
    if ( !osal_queue_receive(_usbh_q, &event) ) return;

    switch (event.event_id)
    {
      case HCD_EVENT_XFER_COMPLETE:
      // …
```

usbh

```
static bool enum_request_addr0_device_desc(void)
{
  uint8_t const addr0 = 0;
  TU_ASSERT( usbh_edpt_control_open(addr0, 8) );

  // Get device descriptor to get Control Endpoint Size
  TU_LOG2("Get 8 byte of Device DescriptorWrWn");
  tusb_control_request_t const request =
  {
    .bmRequestType_bit =
    {
      .recipient = TUSB_REQ_RCPT_DEVICE,
      .type      = TUSB_REQ_TYPE_STANDARD,
      .direction = TUSB_DIR_IN
    },
    .bRequest = TUSB_REQ_GET_DESCRIPTOR,
    .wValue   = TUSB_DESC_DEVICE << 8,
    .wIndex   = 0,
    .wLength  = 8
  };
  TU_ASSERT( tuh_control_xfer(addr0, &request, _usbh_ctrl_buf,
                        enum_get_addr0_device_desc_complete) );

  return true;
}
```

usbh

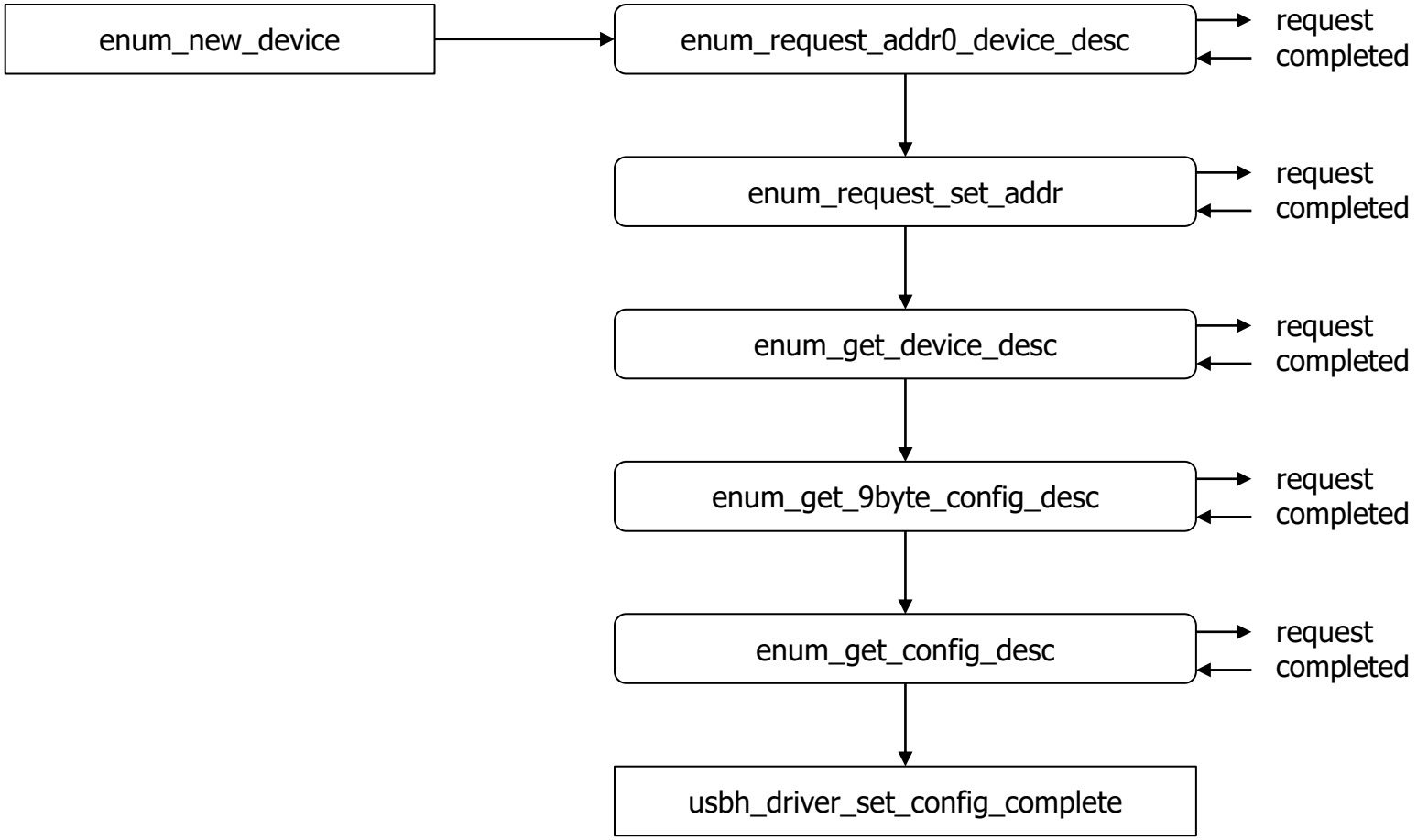
```
static bool enum_get_addr0_device_desc_complete(uint8_t dev_addr,
    tusb_control_request_t const * request, xfer_result_t result)
{
  // Reset device again before Set Address
  if (_dev0.hub_addr == 0)
  {
    // connected directly to roothub
    hcd_port_reset( _dev0.rhport ); // reset port
    osal_task_delay(RESET_DELAY);

    enum_request_set_addr();
  }

  return true;
}
```

usbh

```
static bool enum_request_set_addr(void)
{
  uint8_t const addr0 = 0;
  tusb_desc_device_t const * desc_device = (tusb_desc_device_t
const*) _usbh_ctrl_buf;

  // Get new address
  uint8_t new_addr = get_new_address(desc_device->bDeviceClass
                                     == TUSB_CLASS_HUB);
  TU_ASSERT(new_addr != ADDR_INVALID);

  usbh_device_t* new_dev = get_device(new_addr);

  new_dev->rhport    = _dev0.rhport;
  new_dev->hub_addr  = _dev0.hub_addr;
  new_dev->hub_port  = _dev0.hub_port;
  new_dev->speed     = _dev0.speed;
  new_dev->connected = 1;
  new_dev->ep0_size  = desc_device->bMaxPacketSize0;

  tusb_control_request_t const new_request =
  {
     // TUSB_REQ_SET_ADDRESS
  };

  TU_ASSERT( tuh_control_xfer(addr0, &new_request,
                        NULL, enum_set_address_complete) );
```

usbh

```
static bool enum_set_address_complete(uint8_t dev_addr,
    tusb_control_request_t const * request, xfer_result_t result)
{
  uint8_t const new_addr = (uint8_t const) request->wValue;

  usbh_device_t* new_dev = get_device(new_addr);
  new_dev->addressed = 1;

  // TODO close device 0, may not be needed
  hcd_device_close(_dev0.rhport, 0);

  // open control pipe for new address
  TU_ASSERT( usbh_edpt_control_open(new_addr,
                                    new_dev->ep0_size) );

  // Get full device descriptor
  TU_LOG2("Get Device DescriptorWrWn");
  tusb_control_request_t const new_request =
  {
    // TUSB_REQ_GET_DESCRIPTOR
  };

  TU_ASSERT(tuh_control_xfer(new_addr, &new_request,
               _usbh_ctrl_buf, enum_get_device_desc_complete));

  return true;
}
```


enum_new_device
enum_request_addr0_device_desc
request
completed
enum_request_set_addr
request
completed
enum_get_device_desc
request
completed
enum_get_9byte_config_desc
request
completed
enum_get_config_desc
request
completed
usbh_driver_set_config_complete

usbh

```
void usbh_driver_set_config_complete(uint8_t dev_addr, uint8_t itf_num)
{
  usbh_device_t* dev = get_device(dev_addr);

  for(itf_num++; itf_num < sizeof(dev->itf2drv); itf_num++)
  {
    // continue with next valid interface
    // TODO skip IAD binding interface such as CDCs
    uint8_t const drv_id = dev->itf2drv[itf_num];
    if (drv_id != DRVID_INVALID)
    {
      usbh_class_driver_t const * driver = &usbh_class_drivers[drv_id];
      TU_LOG2("%s set config: itf = %uWrWn", driver->name, itf_num);
      driver->set_config(dev_addr, itf_num);
      break;
    }
  }

  // all interface are configured
  if (itf_num == sizeof(dev->itf2drv))
  {
    // Invoke callback if available
    if (tuh_mount_cb) tuh_mount_cb(dev_addr);
  }
}
```

hub

```
bool hub_set_config(uint8_t dev_addr, uint8_t itf_num)
{
  hub_interface_t* p_hub = get_itf(dev_addr);
  TU_ASSERT(itf_num == p_hub->itf_num);

  // Get Hub Descriptor
  tusb_control_request_t const request =
  {
    .bmRequestType_bit =
    {
      .recipient = TUSB_REQ_RCPT_DEVICE,
      .type      = TUSB_REQ_TYPE_CLASS,
      .direction = TUSB_DIR_IN
    },
    .bRequest = HUB_REQUEST_GET_DESCRIPTOR,
    .wValue   = 0,
    .wIndex   = 0,
    .wLength  = sizeof(descriptor_hub_desc_t)
  };

  TU_ASSERT( tuh_control_xfer(dev_addr, &request,
                _hub_buffer, config_set_port_power) );

  return true;
}
```