USB and Linux

Alan Ott
SCaLE 16x
March 8-11, 2018
About the Presenter

• **Platform Software at SoftIron**
  - Data center appliances (storage, transcoding)
  - Ceph-based storage appliances
  - OverDrive 3000/1000 ARM servers

• **OSS Development**
  - Linux Kernel
  - Firmware
  - Training
  - USB
    • **M-Stack** USB Device Stack for PIC
  - 802.15.4 wireless
USB Overview
Universal Serial Bus

- Universal Serial Bus (USB)
- Standard for a high-speed, bi-directional, low-cost, dynamic bus.
- Created by the USB Implementers Forum (USB-IF)
  - USB-IF is a non-profit corporation formed by its member companies.
  - USB-IF develops and owns copyrights on the standards documents and logos.
  - http://www.usb.org
USB Bus Speeds

- **Low** Speed
  - 1.5 Mb/sec
- **Full** Speed
  - 12 Mb/sec
- **High** Speed
  - 480 Mb/sec
- **SuperSpeed**
  - 5.0 Gb/sec
- **SuperSpeed+**
  - 10Gb/sec
USB Bus Speeds

- Bus speeds are the **rate of bit transmission** on the bus.
- Bus speeds are **NOT** data transfer speeds.
- USB protocol can have **significant overhead**.
- USB overhead **can be mitigated** if your protocol is designed correctly.
USB Standards

• **USB 1.1** – 1998
  - Low Speed / Full Speed

• **USB 2.0** – 2000
  - High Speed added

• **USB 3.0** – 2008
  - SuperSpeed added

• USB Standards **do NOT imply a bus speed!**
  
  ➢ A **USB 2.0** device can be High Speed, Full Speed, or Low Speed
Host and Device

• **Host**
  - Often a PC, server, or embedded Linux system
  - Responsible for **control** of the bus
  - Responsible for **initiating communication** with devices
  - Responsible for **enumeration** of attached devices.
  - One host per bus
Host and Device

- **Device**
  - Provide functionality to the host
  - **Many devices** per bus
  - Can connect through **hubs**
    - Hubs are transparent to the device!
    - Hubs are transparent to host APIs
      - Hub drivers are built into the OS
The Bus

• **USB is a Host-controlled bus**
  • Nothing on the bus happens without the **host** first initiating it.
  • Devices cannot initiate a transaction.
• The USB is a **Polled Bus**
  • The Host polls each device, requesting data or sending data.
  • Devices cannot interrupt the host!
Terminology

• In/Out
  • In USB parlance, the terms **In** and **Out** indicate direction from the **Host** perspective.
    - **Out**: Host to Device
    - **In**: Device to Host
USB Terminology

- **Device** – Logical or physical entity which performs a function.
  - Thumb drive, joystick, etc.
- **Configuration** – A mode in which to operate.
  - Many devices have one configuration.
  - Only one configuration is active at a time.
USB Terminology

- **Interface** – A related set of Endpoints which present a single feature or function to the host.
  - A configuration may have **multiple** interfaces
  - All interfaces in a configuration are **active at the same time**.

- **Endpoint** – A source or sink of data
  - Interfaces often contain **multiple endpoints**, each active all the time.
Logical USB Device

• Important to note:
  • A **device** can have multiple **configurations**.
    – Only one active at a time
  • A **configuration** can have multiple **interfaces**.
    – All active at the same time
  • An **interface** can have multiple **endpoints**.
    – All active at the same time
Logical USB Device

- Most USB devices only have one Configuration.
- Only **one configuration** can be active at a time.
- **All interfaces** within a configuration are active at the same time.
  - This is how **composite** devices are implemented.
Endpoint Terminology

- An **Endpoint Number** is a 4-bit integer associated with an endpoint (0-15).
- An endpoint transfers data in a **single direction**.
- An **Endpoint Direction** is either **IN** or **OUT**.
- An **Endpoint Address** is the combination of an endpoint number and an endpoint direction. Examples:
  - EP 1 IN
  - EP 1 OUT
  - EP 3 IN
Endpoint Terminology

- Endpoint addresses are encoded with the direction and number in a **single byte**.
  - **Direction** is the MSb (1=IN, 0=OUT)
  - **Number** is the lower four bits.
  - Examples:
    - EP 1 IN = 0x81
    - EP 1 OUT = 0x01
    - EP 3 IN = 0x83
    - EP 3 OUT = 0x03
  - Tools like `lsusb` will show both
Endpoint Terminology

- Endpoint terminology is tricky (but important!)
  - A device can have up to **32 endpoints**.
    - IN and OUT endpoints for numbers 0-15.
- The same **Endpoint Number** is used to describe TWO endpoints.
  - EP 1 IN and EP 1 OUT are separate endpoints!
  - There is no such thing as a physical and logical endpoint.
Real-Life Example

- **Composite Device:**
  - Communication Device Class (CDC)
    - Often virtual **serial port**
    - **Two interfaces** are required for this class (control and data).
  - Vendor-Defined class
    - Can be used for generic data transfer
Descriptors

• **USB is a **self-describing** bus**

• Each USB device contains all the information required for the host to be able to communicate with it (drivers aside)
  - No manual setting of baud rates, IRQ lines, base addresses, etc.
  - Plug devices in and they work

• Devices communicate this data to the host using **descriptors**.
Descriptors

• The host will ask for a set of standard descriptors during **enumeration**, immediately upon a device being attached.

• The descriptors describe:
  • The device identifier (vendor/product IDs)
  • The logical structure of the device
    - Configurations, interfaces, endpoints
  • Which device classes are supported (if any)
Descriptors

• Typically, devices contain at least:
  • **Device** descriptor
  • **Configuration** descriptor
  • **Interface** descriptor
  • Class-specific descriptors
  • **Endpoint** descriptor

➢ *Chapter 9 of the USB spec describes these standard descriptors*
Descriptors

- One tricky thing is that the host will request all descriptors which are part of a configuration as a single block.
  - This includes Configuration, Interface, class-specific, and endpoint descriptors
    - The *Get Descriptor (Configuration)* request means all descriptors of a configuration
Device Descriptor

```c
const struct device_descriptor this_device_descriptor = {
    sizeof(struct device_descriptor), // bLength
    DESC_DEVICE, // bDescriptorType
    0x0200, // USB Version: 0x0200 = USB 2.0, 0x0110 = USB 1.1
    0x00, // Device class (0 = defined at interface level)
    0x00, // Device Subclass
    0x00, // Protocol
    EP_0_LEN, // bMaxPacketSize0 (endpoint 0 in/out length)
    0xA0A0, // Vendor ID (Fake VID!! Don't use this one!)
    0x0001, // Product ID
    0x0001, // device release (BCD 1.0)
    1, // Manufacturer String Index
    2, // Product String Index
    0, // Serial Number String Index
    NUMBER_OF_CONFIGURATIONS // NumConfigurations
};
```
The Configuration Packet, in this example, consists of four descriptor structs. Note that there is a single configuration, a single interface, and two endpoints.

```c
struct configuration_1_packet {
    struct configuration_descriptor config;
    struct interface_descriptor interface;
    struct endpoint_descriptor ep;
    struct endpoint_descriptor ep1_out;
};
```
static const struct configuration_1_packet configuration_1 = {
{

 // Members from struct configuration_descriptor
 sizeof(struct configuration_descriptor),
 DESC_CONFIGURATION,
 sizeof(configuration_1), // wTotalLength (length of the whole packet)
 1, // bNumInterfaces
 1, // bConfigurationValue
 2, // iConfiguration (index of string descriptor)
 0X80, // bmAttributes
 100/2, // 100/2 indicates 100mA 

},
}
Configuration Descriptor (cont'd)

{
  // Members from struct interface_descriptor
  sizeof(struct interface_descriptor), // bLength;
  DESC_INTERFACE,
  0x0, // InterfaceNumber
  0x0, // AlternateSetting
  0x2, // bNumEndpoints (num besides endpoint 0)
  0xff, // bInterfaceClass: 0xFF=VendorDefined
  0x00, // bInterfaceSubclass
  0x00, // bInterfaceProtocol
  0x02, // iInterface (index of string describing interface)
},
Configuration Descriptor (cont'd)

```c
{
    // Members of the Endpoint Descriptor (EP1 IN)
    sizeof(struct endpoint_descriptor),
    DESC_ENDPOINT,
    0x01 | 0x80, // endpoint #1 0x80=IN
    EP_BULK, // bmAttributes
    64, // wMaxPacketSize
    1, // bInterval in ms.
},
{
    // Members of the Endpoint Descriptor (EP1 OUT)
    sizeof(struct endpoint_descriptor),
    DESC_ENDPOINT,
    0x01, // endpoint #1 OUT (msb clear => OUT)
    EP_BULK, // bmAttributes
    64, // wMaxPacketSize
    1, // bInterval in ms.
};
```
Configuration Descriptor

- Preceding configuration descriptor described:
  - One Configuration
  - One interface (vendor defined)
  - Two Bulk Endpoints

- See examples in `usb_descriptors.c` in any of the M-Stack examples.
Endpoints

- Four types of Endpoints
  - **Control**
    - Bi-directional pair of endpoints
    - **Multi-stage** transfers
      - Transfers acknowledged on the software level
        - Not just hardware!
      - Status stage can return success/failure
    - Used during **enumeration**
    - Can also be used for application
    - Mostly used for configuration items
    - Most robust type of endpoint
Endpoints

- **Interrupt**
  - Transfers a *small amount* of *low-latency* data
  - Reserves bandwidth on the bus
  - Used for *time-sensitive* data (HID).

- **Bulk**
  - Used for *large, time-insensitive* data (Network packets, Mass Storage, etc).
  - Does not reserve bandwidth on bus
    - Uses whatever time is left over
Endpoints

• Isochronous
  – Transfers a large amount of time-sensitive data
  – Delivery is not guaranteed
    • No ACKs are sent
  – Used for Audio and Video streams
    • Late data is as good as no data
    • Better to drop a frame than to delay and force a re-transmission
Endpoints

• **Reserved Bandwidth**
  • Different endpoint types will cause the bus to **reserve bandwidth** when devices are connected.
    - This is how **guaranteed, bounded latency** is implemented.

• **Interrupt, Isochronous, and Control** endpoints reserve bandwidth.

• **Bulk** gets whatever bandwidth is left unused each frame.
Endpoints

- **Endpoint Length**
  - The **maximum amount of data** an endpoint can support sending or receiving **per transaction**.
  - Max endpoint sizes:
    - Full-speed:
      - Bulk/Interrupt: **64**
      - Isoc: **1024**
    - High-Speed:
      - Bulk: **512**
      - Interrupt: **3072**
      - Isoc: **1024 x3**
Transactions

• Basic process of moving data to and from a device.

• USB is **host-controlled**. All transactions are initiated by the host.
  • Much like everything else in USB

• A single transaction on an endpoint can move bytes up to the **Endpoint Length**
Transactions

- Transactions have three **phases**
  - **Token** Phase
    - Host sends a token packet to the device
      - Indicates start of transaction
      - Indicates **type** of transaction (IN/OUT/SETUP)
  - **Data** Phase
    - Host or Device sends data
  - **Handshake** Phase
    - Device or host sends acknowledgement (ACK/NAK/Stall)
Transactions

- Transactions are handled on the **Hardware** level.
  - Strict timing is necessary
  - Software will **configure the hardware** to handle the transaction conditions before they occur.
    - This means the software/firmware must be prepared for what is coming!
    - not reacting to what has happened
  - Hardware will NAK if not configured
Transactions

- Endpoints are typically implemented in a hardware peripheral
  - Typically the USB hardware device is called the **Serial Interface Engine** (SIE)
  - SIE contains registers for each endpoint.
    - Pointer to data buffer (and length)
    - Firmware will configure these registers for transactions which are expected
- SIE generates Interrupts when transactions complete
Transactions

- **Token** Phase
  - The host will initiate every transaction by sending a token. Tokens contain a **token type** and an **endpoint number**.
  - The device SIE will handle receipt of the **token** and will handle the **data** and **handshake** phases automatically.
    - This means the SIE endpoint will need to be configured **before** the token comes from the host.
Transactions

• For most cases, the token types are:
  • **IN**
    - The transaction will be an IN transaction, where the device sends data to the host using an IN endpoint.
    - Data phase will be **device-to-host** (ie: in)
    - Handshake phase (ack) will be **host-to-device**
Transactions

• Token types (cont'd):
  • **OUT**
    - The transaction will be an OUT transaction, where the host sends data to the device using an OUT endpoint.
    - Data phase will be **host-to-device** (ie: out)
    - Handshake phase (ack) will be **device-to-host**.
Transactions

- Token types (cont'd):
  - **SETUP**
    - The transaction will be a SETUP transaction
      - SETUP transactions are used to start a **Control Transfer** on a Control endpoint pair.
        - Usually endpoint 0
      - Setup transactions indicate there will be more transactions following, and what types they will be.
    - A Setup transaction is like an OUT transaction, but the data phase always contains a SETUP packet.
Transactions

- **Data** Phase
  - The data phase contains the data which is to be transferred.
  - The data phase packet can be from zero bytes up to the **endpoint length**.
  - For **IN** transactions, the data packet is sent from the **device** to the **host**.
  - For **OUT** or **SETUP** transactions, the data packet is sent from the **host** to the **device**.
Transactions

- **Data** Phase (cont'd)
  - If there is no data to be sent (IN transaction), or if the device is unable to receive (OUT transaction), the device can skip the data stage and send a **NAK**.
    - This ends the transaction prematurely.
  - A NAK tells the host to **try again later**.
    - It is **not a failure** of any kind.
    - NAKs are a normal part of the flow regulation of USB.
      - *The Host is often faster than the device!*
Transactions

- **Handshake** Phase
  - ACK, NAK, or STALL
  - Opposite in direction from the data phase
  - Indicates **hardware** reception of data
  - ACK – Reception OK
  - NAK
    - Unable to receive, try again
  - STALL
    - Protocol error
**Transaction**

- **IN Transaction**

  - The device can NAK as long as it's not ready to send data.
  - The Host will retry (up to a timeout) as long as the device NAKs.
Transaction

- **OUT** Transaction

- The device can NAK as long as it's not ready to receive data.
- The Host will retry (up to a timeout) as long as the device NAKs.
Transactions

- The timing between the phases is very tight
  - Too tight for software/firmware
- The hardware SIE handles this timing
  - The hardware endpoint needs to be setup *before* the IN token arrives.
- This means you must be *ahead* of the host, in a manner of speaking.
Transactions

- For **IN** transactions (**device-to-host**)
  - Device firmware will put data to send in the hardware SIE buffer
  - Host will (sometime later) send the IN token
  - Device SIE will send the data (data stage)
    - Device SIE will resend until ACK is received
  - Host will send an ACK to the device
    - *Note that the data will not get sent until the host initiates the transaction by sending the IN token to the device*
Transactions

- For **OUT** transactions (**host-to-device**)
  - Device firmware configures a hardware SIE buffer to **receive** data
  - Host will (sometime later) send the OUT token
  - Host will send the data.
  - Device SIE will send an ACK
  - Device SIE will interrupt the MCU/CPU.
Transactions

- Note that for OUT transactions, the data is sent **before** the device can respond with a NAK.
  - This is inefficient, as the host will send the entire data phase for each retry.
- USB 2.0 introduced the **PING** token for high-speed devices.
  - After an OUT-NAK, the host can send PING tokens to the device.
  - The device will ACK when it is ready to receive data.
Transactions

- PING packets are sent as long as the device NAKs them.
  - Once the device responds with an ACK, a normal OUT transaction is sent.
- Typically PING is handled by the SIE hardware.
  - It is part of the timing-critical part of a USB transaction.
  - This is automatic and transparent to device drivers running on the host.
Transactions and Transfers

• **Transaction**
  • Delivery of service to an endpoint
  • Max data size: **Endpoint length**

• **Transfer**
  • **One or more** transactions moving information between host and device.
    ▶ **Transfers can be large, even on small endpoints!**
Small Transfers

- The simplest transfer contains a **single transaction**.
- A transaction's size can be any length from zero bytes up to the **endpoint length**.
Large Transfers

- Transfers can contain more than one transaction.

- Transfers are ended by:
  - A short transaction
  - OR
  - When the **desired amount of data** has been transferred
    - As requested by the host
Large Transfers

- Transfers are ended when:
  - A *short transaction* happens
  - The requested amount of data has been transferred
- A *short transaction* is one which is smaller than the endpoint length.
  - This means in a multi-transaction transfer, all transactions except the last must be the endpoint length
Large Transfers

- Sometimes a host **does not know** the number of bytes it is asking for.
  - For example a string descriptor.
- The host will ask for the **maximum** number of bytes it can accept and will rely on the **device** to end the transfer early.
- This gives an interesting **edge case**
Large Transfers

• There are four cases of large transfers. Let's consider **IN** transfers:

  • Case 1:
    - Host asks for a number of bytes which is **not a multiple** of the endpoint length.
    - device returns this many bytes.

  • Case 2:
    - Host asks for a **multiple** of the endpoint length.
    - device returns this many bytes.
Large Transfers

• Four cases (cont'd):
  • Case 3:
    – Host asks for a number of bytes
    – device returns fewer than requested, which is not a multiple of the endpoint length.
  • Case 4:
    – Host asks for a number of bytes
    – device returns fewer than requested, but it is a multiple of the endpoint length.
Large Transfers

• In cases #1, #2, and #3, the device can simply return the number of bytes it intends to return.
Large Transfers – Case 1

- Case 1:
  - Host asks for a number of bytes which is not a multiple of the endpoint length.
  - Device Returns this many bytes.
  - Transfer is ended by:
    - A short transaction
    - The desired amount of data has been transferred

- 16-byte endpoint length
- Requested 76 bytes
- 4x 16-byte transactions
- 1x 12-byte transaction
Large Transfers – Case 2

- Case 2:
  - Host asks for a number of bytes which is a multiple of the endpoint length.
  - Device Returns this many bytes.
  - Transfer is ended by:
    - The requested amount of data has been transferred

- 16-byte endpoint length
- Requested 64 bytes
- 4x 16-byte transactions
Case 3:
- Host asks for a number of bytes.
- Device returns fewer than requested, which is not a multiple of the endpoint length.
- Transfer is ended by:
  - A short transaction

- 16-byte endpoint length
- Requested 255 bytes
- Device returns 44 bytes
- 2x 16-byte transactions
- 1x 12-byte transaction
Large Transfers

• Case #4 is an edge case
  - Host requested a number of bytes
  - Device returns fewer than requested, which
    is a multiple of the endpoint length.
• Since the number of bytes being
  returned is a multiple of the endpoint
  length, the transfer will not naturally
  end with a short transaction.
• Device must add a zero-length
  packet!
Large Transfers – Case 4

Case 4:
- Host asks for a number of bytes.
- Device returns fewer than requested, which is a multiple of the endpoint length.
- Transfer is ended by:
  - A short transaction, in this case a zero-length packet

- 16-byte endpoint length
- Requested 255 bytes
- Device returns 32 bytes
- 2x 16-byte transactions
- 1x 0-byte transaction
Control Transfers

- The transfers discussed so far have been **Bulk** or **Interrupt** transfers.
- **Control transfers** are different and more complicated.
  - Control transfers have additional structure and are bi-directional.
  - Consist of multiple **stages**
    - Each stage is one or more transactions
  - Information is sent both ways (IN and OUT)
Control Transfers

• Control transfers begin with the **SETUP** stage.
  • A SETUP transaction is like an OUT transaction except that the data stage is an 8-byte SETUP packet.
    - The SETUP packet has information on:
      • The logical *recipient* of the transfer
      • The *direction* of the transfer
      • The *number of bytes* which will be sent or requested
      • The *identifier* or *type* of the request
Control Transfers

- Chapter 9 of the USB specification defines **standard requests** which are used during enumeration of a device.
  - Set Address
  - Get Descriptor
  - Get Configuration
  - Set Configuration
  - others...
Control Transfers

- Device classes also define their own requests:
  - **CDC** (*Communication Device Class*)
    - Set Line Coding
    - Set Control Line State
    - Send Break
  - **HID** (*Human Interface Device*)
    - Get Report Descriptor
    - Get Report
    - Set Report
Control Transfers

• Next is the **data** stage
  • The data stage is one or more IN or OUT transactions which contain the data.
  • The same rules apply about multi-transaction transfers, lengths and the zero-length packet.
  • The direction and desired data length are in the SETUP packet (part of the setup stage).
Control Transfers

• Next is the **status** stage
  • The status stage is a single, zero-length IN or OUT transaction
    - Opposite in direction from the data stage
  • It serves as an acknowledgement
    - Not just of receipt, but of validity
    - It’s acknowledgment from the software layer that the data in the data stage has been processed correctly.
• On error, the endpoint should **STALL**
Control Transfers

• Additional Token:
  • **STALL**
    – Used to indicate failure or rejection of data
    – Used in control transfers
    – Sometimes used on other endpoints
      • Mass Storage uses it on bulk endpoints
    – Sending a STALL token on any endpoint marks that endpoint as stalled.
    – Stalls must be cleared by the host
      • Automatically cleared with a SETUP for control endpoints.
Control Transfers

- The terminology gets confusing:
  - **Transactions**
    - The lowest level of communication
    - Handled by hardware
    - Three **phases** (token, data, handshake)
  - **Control Transfers**
    - Three **stages** (setup, data, status)
Linux USB Gadget
USB Gadget Subsystem

• In addition to providing the USB host subsystem you are familiar with, Linux also provides a device subsystem, called gadget.
  • Gadget is a Linux-specific name.
• The gadget subsystem provides a framework for creating USB devices using a Linux system.
  • …if the hardware supports it. Most embedded USB controllers do.
USB Gadget Subsystem

- USB gadget subsystem provides:
  - Framework
  - **USB Device Controller** (UDC) drivers
    - Hardware drivers
  - **USB device class** implementations
    - Software drivers (so to speak)
  - Configuration through **configfs**
    - Pseudo-filesystem for configuring certain kernel services
Configfs

• Configfs is a pseudo-filesystem used to manage kernel objects.
  • Pseudo-filesystems contain files which are not present on any disk.
  • The files are backed by objects in the running kernel.
    - Creating, deleting, or changing files and directories will immediately have an effect in the kernel
      • Kernel callback functions are called
Configfs

• Configfs (cont'd)
  • Data integrity is enforced by mechanism.
    - Only valid file / directory names will be allowed to be created
    - Invalid values will not be allowed to be written to files
    - System calls (read/write/mkdir, etc) will simply fail if invalid names or values are used.

➤ This is far better than silent failure
USB Gadget

To configure your USB Gadget:

- Mount configfs (if not already done)
- Create a directory for the gadget
- Set the vid/pid/strings
- Create a directory for the **configuration**
- Create a directory for the **function**
  - Mass storage, HID, CDC/ACM, etc.
- **Link** the function to the configuration
- Enable the gadget
#!/bin/sh -ex

modprobe libcomposite

# Mount configfs locally
mkdir -p config
mount none config -t configfs

# Create the USB gadget configuration
mkdir -p config/usb_gadget/
cd config/usb_gadget/

# Create a gadget called g1
mkdir g1
cd g1

# Set the VID/PID/Strings
echo 0x1a0a >idVendor
echo 0xbadd >idProduct
mkdir strings/0x409
# Set the VID/PID/Strings (cont'd)
```bash
echo 12345 >strings/0x409/serialnumber
echo "Signal 11" >strings/0x409/manufacturer
echo "Test" >strings/0x409/product
```

# Create a configuration called c.1
```bash
mkdir configs/c.1
mkdir configs/c.1/strings/0x409
echo "Config1" >configs/c.1/strings/0x409/configuration
```

# Create a function (tty CDC/ACM) named usb0
```bash
mkdir functions/acm.usb0
```

# Link that function to configuration c.1
```bash
ln -s functions/acm.usb0 configs/c.1
```

# Enable the USB device. Find the device
```bash
# name in /sys/class/udc/.
echo musb-hdrc.0 >UDC
```
Simple Example - Device

- The above example will create a CDC/ACM device.
  - A Linux host will identify this device as `/dev/ttyACMn`.
  - The device/gadget side will create a device node at `/dev/ttyGS0`.
    - Read and write to/from this node from the gadget to communicate with the host.
Simple Example – ACM Device

- Lab #1
  - On the device:
    - Run the script from the previous slides
  - On the host, run:
    - dmesg
      - Check for the new device name
    - sudo screen /dev/ttyACMn
  - On the device, run:
    - echo “some text” >/dev/ttyGS0
    - cat /dev/ttyGS0
Simple Example – ACM Device

- In theory, the device can be disabled, changed, and re-enabled.
- In practice, on many parts, this is fraught with oopses, hangups, and other peril.
- Generally, you will want to setup your gadget and leave it.
Simple Example – ACM Device

• The previous example is designed to be as simple as possible
  • No source code, even!

• CDC/ACM is the best solution for emulating a serial port, and that's all.

• Don't use CDC/ACM as an arbitrary solution for connectivity
  • It's inefficient
  • It's burdensome on the end user
USB Gadget

- We showed ACM, but what other protocols are implemented?
  - Start in Kernel source at: Documentation/filesystems/gadget_configfs.txt
  - Which references: Documentation/ABI/testing
  - Where you can: ls *usb-gadget*
USB Gadget

As of 4.15, these gadgets are documented:

- configfs-usb-gadget
- configfs-usb-gadget-acm
- configfs-usb-gadget-ecm
- configfs-usb-gadget-eem
- configfs-usb-gadget-ffs
- configfs-usb-gadget-hid
- configfs-usb-gadget-loopback
- configfs-usb-gadget-mass-storage
- configfs-usb-gadget-midi
- configfs-usb-gadget-ncm
- configfs-usb-gadget-obex
- configfs-usb-gadget-phonet
- configfs-usb-gadget-printer
- configfs-usb-gadget-rndis
- configfs-usb-gadget-serial
- configfs-usb-gadget-sourcesink
- configfs-usb-gadget-subset
- configfs-usb-gadget-tcm
- configfs-usb-gadget-uac1
- configfs-usb-gadget-uac2
- configfs-usb-gadget-uvc

See these files in: Documentation/ABI/testing/
USB Gadget

- Briefly, some supported functions are:
  - acm – CDC/ACM virtual serial port
  - ecm/eem/ncm/phonet/rndis/subset – Virtual network device
  - ffs – function filesystem
    - Define a custom class from userspace
  - hid – Human interface device
  - loopback – for testing
  - mass-storage – present drives to the host
USB Gadget

- Supported functions (cont'd):
  - midi – musical instrument
  - printer – printers
  - serial – serial interface on the gadget side, but bulk interface on the host side
  - sourcesink – source and sink for testing
  - tcm – USB-attached SCSI
  - uac1/2 – USB audio class, v1 and v2
  - uvc - video
USB Gadget

- Find more information about each gadget in it's respective source file:
  
  
  `drivers/usb/gadget/function/f_*.c`

- As usual in the kernel, documentation is hit-or-miss
FunctionFS

- The gadget subsystem provides a function called FunctionFS, which allows complete configuration of the device through a user space application.

- The user space application provides:
  - All the descriptors/strings
  - Functionfs will then use the descriptors to create **device nodes** for each endpoint.
FunctionFS

- The user space application can then read and write to/from these device nodes to move data across the bus.
- This is better than using ACM because there is no TTY layer in the way.
  - The TTY layer will chop up your write()s
  - Using FFS, one write() is one USB transfer.
    - This can get you close to wire speed
FunctionFS

- Modifying the previous script which (which creates an ACM device), use the ffs function instead of the acm function.
- After this, mount the FunctionFS pseudo-filesystem for your device.
  - This filesystem will give you an ep0 pseudo-file.
  - Writing to this ep0 file with descriptors will configure your device.
FunctionFS

- Start your user space program which will write to the ep0 file and configure the gadget.
  - Based on your configuration (the descriptors) FunctionFS will then create a pseudo-file for each endpoint.
- Enable the USB device
- Read/write the endpoint pseudo-files to transfer data.
# Setup the function, FunctionFS (named usb0)
mkdir functions/ffs.usb0
ln -s functions/ffs.usb0 configs/c.1

# Mount the function filesystem for usb0
cd ../../
mkdir -p ffs
mount usb0 ffs -t functionfs

# From inside the mounted ffs directory, run your user space program and wait until it's started.
  cd ffs
  ../ffs-test/ffs-test & # from the Linux kernel
  sleep 3
  cd..

# Enable the USB device
echo musb-hdrc.0 >config/usb_gadget/g1/UDC
FunctionFS Example

- The kernel provides a sample user space program for FunctionFS.
- Unfortunately, it's more of a test program than an example to learn from.
  - Few comments
  - Complex design and indirection
  - Ambiguous naming
- Find it in:
  tools/usb/ffs-test.c
FunctionFS Example

- The kernel sample creates:
  - ep0 (unused for this example)
  - ep1 – Bulk IN, 512 bytes (high speed)
  - ep2 – Bulk OUT, 512 bytes (high speed)
- ep1 creates an endless source of data
- ep2 sinks an endless stream of data
- ep1/2 are address 0x81 and 0x80
- Also note that it's explicitly GPL.
FunctionFS Lab

- Lab #2
  - Modify the gadget script for functionFS
  - Build the `ffs-test` program from the kernel
  - Copy it to the target's home directory
  - Run the gadget script.
  - Use `lsusb` and `dmesg` to observe the device is detected on the host.
libusb
libusb

- libusb is a multi-platform host-side USB library
  - Linux, BSD, OS X, Windows, others
- Runs in user space. No kernel programming required.
- Easy to use synchronous API
- High-performance asynchronous API
- Supports all versions of USB
libusb

- A libusb host runs on a general purpose multi-process OS.
  - Sufficient permissions are required to open a device
  - Opening a device or interface may be exclusive (only one process at a time).
libusb

• From a host perspective, the basic unit of a USB connection is the USB interface, not the device.
  • This is because devices can have multiple interfaces, each of which may require a different driver.
  • Some composite devices may have some standard interfaces (eg: CDC) and also some vendor-defined interfaces (eg: earlier example)
int main(int argc, char **argv)
{
    libusb_device_handle *handle;
    unsigned char buf[64];
    int length = 64, actual_length, i, res;

    /* Init libusb */
    if (libusb_init(NULL))
        return -1;

    /* Open the device. This is a shortcut function. */
    handle = libusb_open_device_with_vid_pid(NULL, 0xa0a0, 0x0001);
    if (!handle) {
        perror("libusb_open failed: ");
        return 1;
    }

    /* Claim the interface for this process */
    res = libusb_claim_interface(handle, 0);
    if (res < 0) {
        perror("claim interface");
        return 1;
    }
}
libusb Example (cont'd)

/* Initialize the data */
my_init_data_function(buf, length);

/* Send some data to the device */
res = libusb_bulk_transfer(
    handle, 0x01, buf, length, &actual_length, 5000);
if (res < 0) {
    fprintf(stderr, "bulk transfer (out): %s\n",
            libusb_error_name(res));
    return 1;
}

/* Receive data from the device */
res = libusb_bulk_transfer(handle, 0x81, buf, length,
                            &actual_length, 5000);
if (res < 0) {
    fprintf(stderr, "bulk transfer (in): %s\n",
            libusb_error_name(res));
    return 1;
}

/* Process the data */
my_process_received_data_function(buf, &actual_length);

return 0;
libusb

• Observations:
  • **libusb, and libusb_bulk_transfer()** deal with **transfers**, not transactions.
    - The length can be **arbitrarily long** and longer than the endpoint length.
    - If so, libusb will behave as expected, initiating transactions until the required amount of data has been transferred.
    - If the device returns a short packet, the transfer will end, and **actual_length** will indicate the actual amount of data received.
libusb

- Observations (cont'd):
  - The `libusb_bulk_transfer()` function is used for both IN and OUT transfers
    - The endpoint address (which contains the direction) is used to determine whether it's an IN or OUT transfer.
libusb

• Observations (cont'd):
  • The interface must be **claimed** before it can be used.
    - If another process, or a kernel driver, is using this interface, it will kick the other driver off.
    - This can be good or bad depending on your point of view.
libusb

• Observations (cont'd):
  • The libusb functions take a **timeout** parameter.
    - This timeout is how long the device has to complete the transfer.
    - It can be any value the host desires
      - The host is in charge of the bus!
    - 5 seconds is good for general purposes, but the author recently made one over 90 seconds!
      - It all depends on the use case!
libusb

- The simple example used libusb's synchronous API.
  - Good for infrequent, single transfers.
    - Easy to use, blocking, return code
  - Bad for any kind of performance-critical applications.
    - Why? Remember the nature of the USB bus....
**Synchronous API Issues**

- **The USB Bus**
  - Entirely host controlled
  - Device only sends data when the host controller specifically *asks* for it.
  - The host controller will only ask for data when a *transfer is active*.  
    - libusb **creates a transfer** when (in our example) libusb_bulk_transfer() is called.
Synchronous API Issues

**Host**

- libusb_bulk_transfer()
- ioctl(IOCTL_USBFS_SUBMITURB)

- *HCI
  - Send IN token
  - Send ACK

**Device**

- USB Transaction
  - Send data packet
Synchronous API Issues

• USB Bus
  • After one transfer completes, nothing happens on the bus until the next libusb transfer function is called.
  • One might think it's good enough to call `libusb_bulk_transfer()` in a **tight loop**.
    - Tight loops are **not tight enough**!
      • For short transfers, time spent in software will be more than time spent in hardware!
      • All time spent in software is time a **transfer is not active**!
Asynchronous API

- Fortunately libusb and the kernel provide an **asynchronous API**.
  - Create **multiple** transfer objects
  - **Submit** transfer objects to the kernel
  - Receive a **callback** when transfers complete
- When a transfer completes, there is another (submitted) transfer already queued.
  - **No downtime** between transfers!
Asynchronous API Example

```c
static struct libusb_transfer*
create_transfer(libusb_device_handle *handle, size_t length) {
    struct libusb_transfer *transfer;
    unsigned char *buf;

    /* Set up the transfer object. */
    buf = malloc(length);
    transfer = libusb_alloc_transfer(0);
    libusb_fill_bulk_transfer(transfer, handle,
                             0x81 /* ep */, 
                             buf, 
                             length,
                             read_callback, 
                             NULL /* cb data */, 
                             5000 /* timeout */);

    return transfer;
}
```
Asynchronous API Example (cont'd)

```c
static void read_callback(struct libusb_transfer *transfer) {
    int res;

    if (transfer->status == LIBUSB_TRANSFER_COMPLETED) {
        /* Success! Handle data received */
    }
    else {
        printf("Error: %d\n", transfer->status);
    }

    /* Re-submit the transfer object. */
    res = libusb_submit_transfer(transfer);
    if (res != 0) {
        printf("submitting. error code: %d\n", res);
    }
}
```
/** Create Transfers **/ 
for (i = 0; i < 32; i++) {
    struct libusb_transfer *transfer =
        create_transfer(handle, buflen);
    libusb_submit_transfer(transfer);
}

/** Handle Events **/
while (1) {
    res = libusb_handle_events(usb_context);
    if (res < 0) {
        printf("handle_events()error # %d\n", res);

        /* Break out of this loop only on fatal error.*/
        if (res != LIBUSB_ERROR_BUSY &&
            res != LIBUSB_ERROR_TIMEOUT &&
            res != LIBUSB_ERROR_OVERFLOW &&
            res != LIBUSB_ERROR_INTERRUPTED) {
            break;
        }
    }
}
Asynchronous API

- This example creates and queues 32 transfers.
- When a transfer completes, the completed transfer object is re-queued.
- All the transfers in the queue can conceivably complete without a trip to user space.
Asynchronous API

• For All types of Endpoint:
  • The Host **will not send** any IN or OUT tokens on the bus unless a **transfer object is active**.
  • The bus is **idle** otherwise
  • Create and submit transfer objects using the functions on the preceding slides.
Performance

- For more information on USB performance, see my ELC 2014 presentation titled *USB and the Real World*
    - Several devices and methods compared
API Summary

- All traffic is initiated by the **Host**
- In user space, this is done from **libusb**:
  - Synchronous:
    - `libusb_control_transfer()`
    - `libusb_bulk_transfer()`
    - `libusb_interrupt_transfer()`
  - Asynchronous:
    - `libusb_create_transfer()`
    - `libusb_submit_transfer()`
Libusb Lab

- Lab #3
  - Create a user space application to talk to the FunctionFS gadget device you created earlier
  - Remember:
    - Find the VID/PID from the script
    - ep81 is bulk IN, ep01 is bulk OUT