USB and the Real World

Signal11
Software

Alan Ott
Embedded Linux Conference
April 28, 2014
About the Presenter

- Chief Bit-Banger at **Signal 11 Software**
  - Products and **consulting services**
- **Linux Kernel**
- **Firmware**
- **Userspace**
- **Training**
- **USB**
  - **M-Stack** USB Device Stack for PIC
- **802.15.4** wireless
USB Overview
USB Bus Speeds

- **Low** Speed
  - 1.5 Mb/sec
- **Full** Speed
  - 12 Mb/sec
- **High** Speed
  - 480 Mb/sec
- **Super** Speed
  - 5.0 Gb/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
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

USB Device

Configuration 1
- Interface 0
  - Endpoint 1 OUT
  - Endpoint 1 IN
  - Endpoint 2 IN
- Interface 1
  - Endpoint 3 OUT
  - Endpoint 3 IN

Configuration 2
- Interface 0
  - Endpoint 1 OUT
  - Endpoint 1 IN
- Interface 1
  - Endpoint 2 OUT
  - Endpoint 2 IN
Endpoints

- Four types of Endpoints
  - **Control**
    - **Bi-directional** endpoint
      - Status stage can return success/failure
    - **Multi-stage** transfers
    - Used for **enumeration**
    - Can be used for application
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** data transfers
  - 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

• 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**
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!*
Transfers

- Transfers contain one or more transactions.
- Transfers are ended by:
  - A short transaction
  - OR
  - When the desired amount of data has been transferred
    - As requested by 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
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.
Transactions

- **IN** Transaction (Device to Host)
  - Host sends an **IN token**
  - If the device has data:
    - Device sends data
    - Host sends **ACK**
  - else
    - Device sends **NAK**
  - If the device sends a **NAK**, the host will retry repeatedly until timeout.
Transactions

- **OUT** Transaction (Host to Device)
  - Host sends an **OUT token**
  - Host sends the data (up to endpoint length)
  - Device sends an **ACK** (or **NAK**).

  - The data is sent before the device has a chance to respond at all.
  - In the case of a **NAK**, the host will **retry** until timeout or success.
Transactions

- 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_submit_transfer()`
Transactions

• In kernel space, this is done from:
  • Synchronous:
    - `usb_control_msg()`
    - `usb_bulk_msg()`
    - `usb_interrupt_msg()`
  • Asynchronous:
    - `usb_submit_urb()`
Transactions

For All types of Endpoint:

- The Host **will not send** any IN or OUT tokens on the bus unless a **transfer is active**.
- The bus is **idle** otherwise
- Create and submit a transfer using the functions on the preceding slides.
Linux USB Gadget Interface and Hardware
USB Gadget Interface

- Linux supports **USB Device Controllers (UDC)** through the **Gadget** framework.
  - Kernel sources in drivers/usb/gadget/
- The gadget framework is transitioning to use **configfs** for its configuration
  - See Matt Porter's presentation:
    - *Kernel USB Gadget Configfs Interface*
    - Thursday, May 1 at 4:00 PM
USB Device Hardware

- UDC hardware is not standardized
  - This is different from most host controllers
  - We will focus on musb, EG20T, and PIC32
- musb
  - IP core by Mentor Graphics
    - Recently becoming usable
  - Common on ARM SoC’s such as the AM335x on the BeagleBone Black (BBB)
  - Host and Device
USB Device Hardware

- **Intel EG20T Platform Controller Hub (PCH)**
  - Common on Intel-based x86 embedded platforms
  - Part of many industrial System-on-Module (SoM) parts
  - Device Only (EHCI typically used for Host)

- **Microchip PIC32MX**
  - Microcontroller
  - Does not run Linux (firmware solution)
  - Full-speed only
  - **M-Stack** OSS USB Stack
Test Hardware
Test Hardware

- **BeagleBone Black**
  - Texas Instruments / CircuitCo
  - AM3359, ARM Cortex-A8 SOC
  - 3.3v I/O, 0.1” spaced connectors
  - Boots mainline kernel and u-boot!
  - Ethernet, USB host and device (musb), Micro SD
  - Great for breadboard prototypes
  - [http://www.beagleboard.org](http://www.beagleboard.org)

Image from beagleboard.org
Test Hardware

- OEM **Intel Atom**-based board
  - Intel Atom E680
  - 1.6 GHz x86 hyperthreaded 32-bit CPU
  - 1 GB RAM
  - Intel **EG20T** platform controller
    - Supports USB Device (pch_udc driver)
    - Serial, CAN, Ethernet, more...
Test Hardware

- **ChipKit Max32**
  - **PIC32MX795F512L**
    - 32-bit Microcontroller
    - Up to 80 MHz (PLL)
      - Running at 60 MHz here
    - Full Speed USB
      - **M-Stack** OSS USB Stack
    - 512 kB flash
    - 128 kB RAM
    - Serial, CAN, Ethernet, SPI, I2C, A/D, RTCC
    - [http://chipkit.net](http://chipkit.net)
Performance

• Three classes of USB device:
  1. Designer wants an easy, well-supported connection to a PC
  2. Designer wants to make use of an existing device class and not write drivers
  3. Designer wants #1 but also wants to move a lot of data quickly.
Performance

- For Cases #1 and #2, naïve methods can get the job done:
  - HID (*Not recommended for generic devices*)
  - Simplistic software on both the host and device side
    - For #2, **no software** on the host side!
  - Synchronous interfaces copied from examples

- What about where we need performance?
Performance

- A simple example:
  - High-speed Device
  - 512-byte bulk endpoints
  - **Receive** data from device using `libusb` in logical application-defined blocks
    - In this case let's use **64-bytes**
unsigned char buf[64];
int actual_length;

do {
    /* Receive data from the device */
    res = libusb_bulk_transfer(handle, 0x81, buf,
                                sizeof(buf), &actual_length, 100000);
    if (res < 0) {
        fprintf(stderr, "bulk transfer (in): %s\n",
                libusb_error_name(res));
        return 1;
    }
} while (res >= 0);
#!/bin/sh -ex

# Setup the device (configfs)
modprobe libcomposite
mkdir -p config
mount none config -t configfs
cd config/usb_gadget/
mkdir g1
cd g1

echo 0x1a0a >idVendor
echo 0xbadd >idProduct
mkdir strings/0x409

echo 12345 >strings/0x409/serialnumber
echo "Signal 11" >strings/0x409/manufacturer
echo "Test" >strings/0x409/product
mkdir configs/c.1
mkdir configs/c.1/strings/0x409

echo "Config1" >configs/c.1/strings/0x409/configuration
# Setup functionfs
mkdir functions/ffs.usb0
ln -s functions/ffs.usb0 configs/c.1

cd ../../../
mkdir -p ffs
mount usb0 ffs -t functionfs
cd ffs
../ffs-test 64 & # from the Linux kernel, with mods!
sleep 3
cd ..

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

➢ Again, see Matt Porter’s presentation for exact steps regarding configfs and gadgets.
Simple Example - Results

- On the BeagleBone Black:
  - Previous example will transfer at **4 Mbit/sec**!
  - Remember this is a high-speed device!
  - Clearly far too slow!
  - What can be done?
Performance Enhancements

• 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 a transfer completes, the device **will not send any more data** until another transfer is created and submitted!
  - In our simple example, this is done with `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 callback when transfers complete
- When a transfer completes, there is another (submitted) transfer already queued.
  - No downtime between transfers!
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;
}
```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 userspace**.
- Results on BeagleBone Black:
  - **15 Mbit/sec**
    - A little better, but still not good!
Transfer Size

- The previous examples used a **64-byte** transfer size.
  - One short transaction per transfer
  - *The max bulk endpoint size is 512-bytes.*
- Larger transactions mean less overhead.
  - Each transaction requires three packets
    - **Token** phase
    - **Data** phase
    - **Handshake** phase (ACK/NAK)
  - Longer data packets means fewer transactions.
Transfer Size

Results:

- On BeagleBone Black, 512-byte transfers using the asynchronous API yields: 82 Mbit/sec
- Better, but still sub-optimal
- Why still so slow?
  - Transaction size is maximal...
  - Host side latency is minimal...
  - Use Analyzer to find out.
USB Analyzer

- TotalPhase Beagle Analyzers
  - Beagle USB 480 Power Protocol Analyzer
  - Well supported on Linux
  - Class-level debugging
  - Power (current/voltage) analysis
- http://www.totalphase.com
USB Analyzer

512-byte transfers

<table>
<thead>
<tr>
<th>Time</th>
<th>Sequence</th>
<th>Length</th>
<th>Type</th>
<th>Status</th>
<th>Polls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:51.485.059</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[L SOF]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.020</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[25 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.075</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.124</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.184</td>
<td>83 ns</td>
<td>05 01</td>
<td>IN</td>
<td>[L SOF]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.186</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[19 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.210</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.309</td>
<td>83 ns</td>
<td>05 01</td>
<td>IN</td>
<td>[L SOF]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.260</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[27 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.324</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.374</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.434</td>
<td>83 ns</td>
<td>05 01</td>
<td>IN</td>
<td>[L SOF]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.436</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[21 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.472</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[33 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.559</td>
<td>66 ns</td>
<td>05 01</td>
<td>IN</td>
<td>[L SOF]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.520</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[25 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.574</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.623</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[34 POLL]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.684</td>
<td>66 ns</td>
<td>05 01</td>
<td>IN</td>
<td>[L SOF]</td>
<td></td>
</tr>
<tr>
<td>1:51.485.686</td>
<td>512 B</td>
<td>05 01</td>
<td>IN</td>
<td>[21 POLL]</td>
<td></td>
</tr>
</tbody>
</table>

~55 uSec per transaction
USB Analyzer

- Opening the transactions gives more insight

| HS | 259957 | 1:51.484.936 | 512 B | 05 01 | IN txn [21 POLL] |
| HS | 259962 | 1:51.484.971 | 512 B | 05 01 | IN txn [33 POLL] |
| HS | 259967 | 1:51.485.059 | 83 ns | 05 01 | [1 SOF] |
| HS | 259968 | 1:51.485.020 | 512 B | 05 01 | IN txn [25 POLL] |
| HS | 259969 | 1:51.485.020 | 25.2 us | 05 01 | [25 IN-NAK] |
| HS | 259970 | 1:51.485.061 | 3 B | 05 01 | IN packet |
| HS | 259971 | 1:51.485.061 | 515 B | 05 01 | [DATA1 packet] |
| HS | 259972 | 1:51.485.070 | 1 B | 05 01 | ACK packet |
| HS | 259973 | 1:51.485.075 | 512 B | 05 01 | IN txn [34 POLL] |
| HS | 259974 | 1:51.485.075 | 34.9 us | 05 01 | [34 IN-NAK] |
| HS | 259975 | 1:51.485.110 | 3 B | 05 01 | IN packet |
| HS | 259975 | 1:51.485.110 | 515 B | 05 01 | [DATA0 packet] |
| HS | 259977 | 1:51.485.119 | 1 B | 05 01 | ACK packet |
| HS | 259978 | 1:51.485.124 | 512 B | 05 01 | IN txn [34 POLL] |
| HS | 259983 | 1:51.485.184 | 83 ns | 05 01 | [1 SOF] |
| HS | 259984 | 1:51.485.186 | 512 B | 05 01 | IN txn [19 POLL] |
| HS | 259989 | 1:51.485.219 | 512 B | 05 01 | IN txn [34 POLL] |
| HS | 259994 | 1:51.485.309 | 83 ns | 05 01 | [1 SOF] |
| HS | 259995 | 1:51.485.268 | 512 B | 05 01 | IN txn [27 POLL] |
| HS | 270000 | 1:51.485.324 | 512 B | 05 01 | IN txn [132 POLL] |

Device sends NAKs for 41 us. (device latency)

5 us between ACK and next request (host latency)
USB Analyzer

• Observations
  • Certainly the 41us of NAK time is less than ideal.
  • Don't be fooled by the displayed 5us between transactions.
    – There's more to the story!
  • The bus scheduler can adapt to the actual time between packets.
    – Number of IN-NAKs will go down
    – Time will stay the same.
  – Don't count NAKs; look at times!
Transfer Sizes

• What changes with multi-transaction transfers?
  – Depends on the UDC hardware.
  – Many UDC controllers use DMA at the Transfer-level.
    • One DMA transfer per USB transfer.
    • Minimizing the number of DMA transfers will decrease DMA overhead.
    • Decrease the number of transfers by increasing the transfer size.
  – Fewer trips to user-space!
Transfer Sizes

• Increased transfer size
  • Limited by hardware/DMA/Driver
  • 64kB seems to work well
    – Performance increases with transfer size up to 64k and plateaus in testing.
• Performance with 64kB transfers:
  – BeagleBone Black: 211 Mbit/sec
  – Intel E680 Board: 305 Mbit/sec
USB Analyzer – Large Transfers

Example: Transfer size = 2047 (512 * 3 + 511)

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Duration</th>
<th>Size</th>
<th>Type</th>
<th>Polls</th>
</tr>
</thead>
<tbody>
<tr>
<td>353613</td>
<td>0:06:625.332</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353617</td>
<td>0:06:625.343</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353622</td>
<td>0:06:625.363</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353627</td>
<td>0:06:625.414</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353632</td>
<td>0:06:625.432</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353637</td>
<td>0:06:625.456</td>
<td>66 ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>353638</td>
<td>0:06:625.457</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353642</td>
<td>0:06:625.471</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353647</td>
<td>0:06:625.521</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353652</td>
<td>0:06:625.537</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
</tr>
<tr>
<td>353657</td>
<td>0:06:625.554</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
</tr>
</tbody>
</table>

**Single Transfer**

Transfers end with the 511-byte transaction
USB Analyzer – Large Transfers

Same Transfer, but with first two transactions open

<table>
<thead>
<tr>
<th>Time</th>
<th>Size</th>
<th>Lost Time</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:625.343</td>
<td>511B</td>
<td>03 01</td>
<td>IN txn [7 POLL]</td>
</tr>
<tr>
<td>00:625.363</td>
<td>512B</td>
<td>03 01</td>
<td>IN txn [39 POLL]</td>
</tr>
<tr>
<td>00:625.404</td>
<td>3B</td>
<td>03 01</td>
<td>[39 IN-NAK]</td>
</tr>
<tr>
<td>00:625.404</td>
<td>515B</td>
<td>03 01</td>
<td>DATA0 packet</td>
</tr>
<tr>
<td>00:625.413</td>
<td>1B</td>
<td>03 01</td>
<td>ACK packet</td>
</tr>
<tr>
<td>00:625.414</td>
<td>512B</td>
<td>03 01</td>
<td>IN txn [7 POLL]</td>
</tr>
<tr>
<td>00:625.414</td>
<td>6.61us</td>
<td>03 01</td>
<td>[7 IN-NAK]</td>
</tr>
<tr>
<td>00:625.421</td>
<td>3B</td>
<td>03 01</td>
<td>IN packet</td>
</tr>
<tr>
<td>00:625.422</td>
<td>515B</td>
<td>03 01</td>
<td>DATA1 packet</td>
</tr>
<tr>
<td>00:625.431</td>
<td>1B</td>
<td>03 01</td>
<td>ACK packet</td>
</tr>
<tr>
<td>00:625.432</td>
<td>512B</td>
<td>03 01</td>
<td>IN txn [7 POLL]</td>
</tr>
<tr>
<td>00:625.456</td>
<td>66ns</td>
<td>03 01</td>
<td>[1 SOF]</td>
</tr>
<tr>
<td>00:625.457</td>
<td>511B</td>
<td>03 01</td>
<td>IN txn</td>
</tr>
<tr>
<td>00:625.471</td>
<td>512B</td>
<td>03 01</td>
<td>IN txn [39 POLL]</td>
</tr>
<tr>
<td>00:625.705</td>
<td>83ns</td>
<td>03 01</td>
<td>[1 SOF]</td>
</tr>
</tbody>
</table>

First Transaction:
- 39.4 us lost between transfers

Single Transfer:
- 6.6 us lost between transactions

A significant improvement over losing ~40 us between each transaction!
Large Transfers

• What about Full Speed?
  • PIC32MX tops out around 8.6 Mbit/sec.
    – 64 kB transfer
  • Using the **asynchronous** API, performance improvement with transfer size is not as dramatic:
    – **8.2 Mbit/sec** with 64-byte transfers
Large Transfers

• Limitations
  • USB is a **message-based** protocol.
    – It's **convenient** to put one logical piece of data into its own transfer.
    – Packing multiple logical pieces of data into one large buffer **loses some of the benefit** of the USB protocol.
    – A **necessary trade-off** if performance is desired.
  • Queuing of messages can cause **increased latency** (marginal).
Other Considerations

• User space vs Kernel space

  • The above examples use the kernel's Functionfs interface on the device side.
    - Functionfs, using the userspace code from mainline, takes transfers from a user space process synchronously.
      • Synchronous → delay between transfers
      • Mitigated by larger transfers
    - Functionfs can also use Linux's Asynchronous I/O capability
      • Better performance
      • User space AIO code is pending merge
Other Considerations

- User space vs Kernel Space (cont'd)
  - Custom gadget function driver
    - Can *queue* packets on the *device side* inside the kernel.
      - Queuing can happen even when the hardware is busy.
Custom Driver

- Driver details
  - Custom Driver has a queue of 32 transfers
  - Device node at /dev/user-gadget

- Performance
  - BeagleBone Black:
    - 227 Mbit/sec, \(\sim 7.6\%\) better than functionfs
  - EG20T:
    - 328 Mbit/sec, \(\sim 7.5\%\) better
Out Transfers

• One might expect OUT transfers to behave similarly to IN transfers.
• On musb, they do not
  – musb: Max throughput of 65.5 Mbit/sec
    • Same for sync and async
    • 64 kB transfers
  – For data received, a DMA transfer is done for every USB Transaction.
    • Overhead is high
    • Large transfers don't help :(
Out Transfers

• On EG20T
  – Max throughput of **255 Mbit/sec**
    • 64 kB transfers
  – Still slower than IN transfers
  – Throughput scales with transfer size.
Results
Test Methodology

- Test with the **synchronous** and **asynchronous** libusb API's
- Test **idle** and under **load**
  - **Device** load (musb):
    - stress -c 1 -m 1
  - **Device** load (EG20T):
    - stress -c 2 -m 2
      - Host machine has one hyperthreaded core
  - **Host** load:
    - stress -c 4 -m 4
      - Host machine has 4 cores
Results

• Warning:
  • Comparisons between controllers should be considered **cautiously**.
    – Plenty of **differences** between boards/platforms.
    – Different **CPU speeds** affect performance tremendously.
      • One Dual core, one single core
    – We know what they say about benchmarks.
    – Use the data to compare effects **within** a controller type
Results

- musb/EG20T (Input) Analysis
  - Larger transfer size is much better
  - Sync/Async affects smaller transfers more than larger transfers.
    - Less time proportionally lost between transfers
  - Transfer size affects EG20T even more than musb
  - Host Load doesn't make much difference
  - Device Load makes more difference
    - Data is sourced from user space
PIC32MX Results (IN Transfers)

![Graph showing Mbit/sec vs Transfer Size]

- Transfer Size: 32, 64, 512, 1024, 65536
- Mbit/sec: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Legend:
- Purple: Idle Sync
- Blue: Idle Async
- Green: Load (Host) Sync
- Light Green: Load (Host) Async
### PIC32MX Results (IN TRF with hub)

<table>
<thead>
<tr>
<th>Transfer Size</th>
<th>Mbit/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>65536</td>
<td>8.0</td>
</tr>
<tr>
<td>1024</td>
<td>8.0</td>
</tr>
<tr>
<td>512</td>
<td>6.5</td>
</tr>
<tr>
<td>64</td>
<td>4.0</td>
</tr>
<tr>
<td>32</td>
<td>2.0</td>
</tr>
</tbody>
</table>

- **Idle Sync**
- **Idle Async**
- **Load (Host) Sync**
- **Load (Host) Async**
Results

- PIC32MX (Input) Analysis
  - Larger transfer sizes don't help as much for sync as they do for async.
  - Addition of a hub has a surprising affect
    - Analyzer shows more frequent IN tokens when connected through a hub.
    - Synchronous transfers are faster
    - Asynchronous transfers slightly slower
musb Results (OUT Transfers)

![Graph showing transfer sizes and Mbit/sec](image-url)
EG20T Results (OUT Transfers)

Mbit/sec vs. Transfer Size

- 65536 bytes
- 1024 bytes
- 512 bytes
- 64 bytes

Transfer Types:
- Idle Sync
- Idle Async
- Load (Device) Sync
- Load (Device) Async
- Load (Host) Sync
- Load (Host) Async
- Idle Fast Sync
- Idle Fast Async

Embedded Linux Conference
Results

- musb/EG20T (OUT) Analysis
  - musb does one DMA transfer per USB transaction.
  - musb OUT Performance **tops out** with 512-byte transfers
    - Endpoint size is 512.
  - EG20T OUT performance **scales similarly to IN** performance.
  - Hub numbers are similar but slightly slower (see spreadsheet)
PIC32MX Results (OUT Transfers)

Transfer Size

65536
1024
512
64
32

Mbit/sec

Idle Sync
Idle Async
Load (Host) Sync
Load (Host) Async
PIC32MX Results (OUT TRF with hub)

![Bar chart showing transfer sizes and Mbit/sec values for Idle Sync, Idle Async, Load (Host) Sync, and Load (Host) Async.]
Results

- PIC32MX (Output) Analysis
  - OUT transfers are affected by the hub the same way IN transactions are
  - Speed is comparable to IN transfers
Isochronous Endpoints

• Features
  • **Un-acknowledged**, non-guaranteed
  • Bandwidth reserved
  • Up to 3x1024 bytes per 125us microframe
    - 3072 bytes/frame: **196 Mbit/sec** per endpoint

• Issues
  • Requires AlternateSetting
    - Not supported by functionfs
  • Bandwidth must be available
Multiple Endpoints

- Using multiple bulk endpoints can increase performance.
  - All endpoints and devices share bus time
  - If bottleneck is DMA, extra concurrency could increase performance.
  - More complex to manage.
  - Depends also on host scheduling.
High-Bandwidth Interrupt

- High-speed Interrupt endpoints at > 1024 bytes
  - Can go as high as 3072
  - Reserved Bandwidth
  - Acknowledged
  - `AlternateSetting` required
  - Bus bandwidth **must be available**
    - Device will **fail to enumerate** or change `AlternateSetting` if bandwidth is not available.
Common Pitfalls
Common Pitfalls

• HID
  • Based on **Interrupt Transfers**.
  • Host will poll interrupt endpoints at up to once per **1ms frame** at **full speed**.
  • Interrupt transfers at full speed can be up to **64 bytes** in length.
  • Simple math is 64,000 bytes/sec
    - Good enough for many applications
  • Except....
Common Pitfalls

• HID
  
  • ... Except you don't always get it! Many hosts don't actually poll you that often!
  
  2-4 frames is much more realistic (sometimes worse!)
  
  • Some write synchronous protocols with HID
    
    • Those are even slower!
      
      2-4 frames for data, 2-4 frames for acknowledgement!

  • Use Bulk/Isoc endpoints!

    • Use libusb on the host side
Common Pitfalls

• Serial Gadget
  • The f_serial gadget function creates /dev/ttyGSn nodes.
    − Data is written/read to/from these nodes from the gadget/device side.
    − Since the data goes through the tty framework, it is broken into small transfers.
    − Performance is suboptimal, but ease of use is high.
Tracepoint Analysis
Tracepoints

- The kernel provides a **tracing** mechanism
  - Tracepoints are placed in source code
  - **Enabled/disabled** at runtime
  - Tracepoints can log **data**
  - `trace-cmd` utility to log data
  - `kernelshark` GUI to view/analyze it
  - Useful for finding latencies
Tracepoints

• Available Tracers
  • Additional tracers need to be enabled in menuconfig
    - Log every kernel function
    - Log max call stack size
    - Trace system calls
    - Scheduling latency
    - Others...
KernelShark

- **GUI** for trace analysis
  - Graphically show tracepoints
    - Per-CPU
    - Per-process
  - Show tracepoint data
  - Complex **filtering**
    - By process, CPU, event type or name
  - Excellent documentation
    - [http://people.redhat.com/srostedt/kernelshark/HTML/](http://people.redhat.com/srostedt/kernelshark/HTML/)
KernelShark

Filtered for musb

<table>
<thead>
<tr>
<th>#</th>
<th>CPU</th>
<th>Time Stamp</th>
<th>Task</th>
<th>PID</th>
<th>Latency</th>
<th>Event</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>11462</td>
<td>0</td>
<td>86.556956</td>
<td>ffs-test</td>
<td>124</td>
<td>d...</td>
<td>musb_gadget_queue</td>
<td>endpoint: 0x81</td>
</tr>
<tr>
<td>11463</td>
<td>0</td>
<td>86.556950</td>
<td>ffs-test</td>
<td>124</td>
<td>d...</td>
<td>musb_map_dma_buffer</td>
<td>endpoint: 0x01</td>
</tr>
<tr>
<td>11464</td>
<td>0</td>
<td>86.557262</td>
<td>&lt;idle&gt;</td>
<td>0</td>
<td>d.h.</td>
<td>musb_g_buf</td>
<td></td>
</tr>
<tr>
<td>11465</td>
<td>0</td>
<td>86.557264</td>
<td>&lt;idle&gt;</td>
<td>0</td>
<td>d.h.</td>
<td>musb_unmap_dma_buffer</td>
<td>endpoint: 0x01</td>
</tr>
<tr>
<td>11466</td>
<td>0</td>
<td>86.557335</td>
<td>ffs-test</td>
<td>124</td>
<td>d...</td>
<td>musb_gadget_queue</td>
<td>endpoint: 0x81</td>
</tr>
<tr>
<td>11467</td>
<td>0</td>
<td>86.557337</td>
<td>ffs-test</td>
<td>124</td>
<td>d...</td>
<td>musb_map_dma_buffer</td>
<td>endpoint: 0x81</td>
</tr>
<tr>
<td>11468</td>
<td>0</td>
<td>86.557634</td>
<td>&lt;idle&gt;</td>
<td>0</td>
<td>d.h.</td>
<td>musb_g_buf</td>
<td></td>
</tr>
<tr>
<td>11469</td>
<td>0</td>
<td>86.557637</td>
<td>&lt;idle&gt;</td>
<td>0</td>
<td>d.h.</td>
<td>musb_unmap_dma_buffer</td>
<td>endpoint: 0x01</td>
</tr>
<tr>
<td>11470</td>
<td>0</td>
<td>86.557705</td>
<td>ffs-test</td>
<td>124</td>
<td>d...</td>
<td>musb_gadget_queue</td>
<td>endpoint: 0x81</td>
</tr>
</tbody>
</table>
Tracepoints

- musb driver was **modified** to add tracepoints
  - Declare tracepoints:
    - musb-trace.h
  - Call tracepoint functions (with data):
    - musb_gadget.c
    - musbhsdma.c
Tracepoints

• Results
  • Results show the **latency** involved in the **context switch**.
    - Along with DMA overhead, another reason to use large transfers.
Lessons Learned

- Gadget interface is Fragile
- Functionfs doesn't support AltSettings
  - No Isochronous endpoints
  - No high-bandwidth Interrupt endpoints
- Performance is host-dependent
- Hubs
  - Can have strange effects
  - Some good, some bad.