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 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 Hostpolls 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 host has a chance to respond at all.
  - *In the case of a NAK, the host will retry until timeout or success.*
Transactions

- All Transactions are 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)
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
Performance
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
  - Simplistic software on both the host and device side
    - For #2, no software on the host side!
  - Synchronous interfaces copied from examples
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
Simple Example – Device (cont'd)

# 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 \textbf{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 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!
Better Example - Host

```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;
}
```
Better Example – Host (cont'd)

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

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Time stamp</th>
<th>Bytes</th>
<th>Dev</th>
<th>Dir</th>
<th>Flags</th>
<th>Polls</th>
</tr>
</thead>
<tbody>
<tr>
<td>269962</td>
<td>1:51.484.971</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>33 Poll</td>
</tr>
<tr>
<td>269967</td>
<td>1:51.485.059</td>
<td>83 ns</td>
<td>05</td>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>269963</td>
<td>1:51.485.020</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>25 Poll</td>
</tr>
<tr>
<td>269973</td>
<td>1:51.485.075</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>269978</td>
<td>1:51.485.124</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>269983</td>
<td>1:51.485.184</td>
<td>83 ns</td>
<td>05</td>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>269984</td>
<td>1:51.485.186</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>19 Poll</td>
</tr>
<tr>
<td>269989</td>
<td>1:51.485.210</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>269994</td>
<td>1:51.485.309</td>
<td>83 ns</td>
<td>05</td>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>269995</td>
<td>1:51.485.260</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>27 Poll</td>
</tr>
<tr>
<td>270000</td>
<td>1:51.485.324</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>270005</td>
<td>1:51.485.374</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>270010</td>
<td>1:51.485.434</td>
<td>83 ns</td>
<td>05</td>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>270011</td>
<td>1:51.485.436</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>21 Poll</td>
</tr>
<tr>
<td>270016</td>
<td>1:51.485.472</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>33 Poll</td>
</tr>
<tr>
<td>270021</td>
<td>1:51.485.559</td>
<td>66 ns</td>
<td>05</td>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>270022</td>
<td>1:51.485.520</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>25 Poll</td>
</tr>
<tr>
<td>270027</td>
<td>1:51.485.574</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>270032</td>
<td>1:51.485.623</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>34 Poll</td>
</tr>
<tr>
<td>270037</td>
<td>1:51.485.684</td>
<td>66 ns</td>
<td>05</td>
<td>01</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>270038</td>
<td>1:51.485.686</td>
<td>512 B</td>
<td>05</td>
<td>01</td>
<td>IN bxn</td>
<td>21 Poll</td>
</tr>
</tbody>
</table>

≈55 uSec per transaction

512-byte transfers
USB Analyzer

Opening the transactions gives more insight

<table>
<thead>
<tr>
<th>Time</th>
<th>Address</th>
<th>Value</th>
<th>Length</th>
<th>Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[21 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[33 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>83 ns</td>
<td></td>
<td></td>
<td>[1 SOF]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[25 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>25.2 us</td>
<td></td>
<td></td>
<td>[25 IN-NAK]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>3 B</td>
<td></td>
<td></td>
<td>IN packet</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>515 B</td>
<td></td>
<td>DATA1</td>
<td>packet</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>1 B</td>
<td></td>
<td></td>
<td>ACK packet</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[34 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>34.9 us</td>
<td></td>
<td></td>
<td>[34 IN-NAK]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>3 B</td>
<td></td>
<td></td>
<td>IN packet</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>515 B</td>
<td></td>
<td>DATA0</td>
<td>packet</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>1 B</td>
<td></td>
<td></td>
<td>ACK packet</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[34 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>83 ns</td>
<td></td>
<td>[1 SOF]</td>
<td></td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[19 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[34 POLL]</td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>83 ns</td>
<td></td>
<td>[1 SOF]</td>
<td></td>
</tr>
<tr>
<td>05:01</td>
<td>05:01</td>
<td>512 B</td>
<td></td>
<td>IN bxn</td>
<td>[27 POLL]</td>
</tr>
</tbody>
</table>

Host Requests data

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.
    – In this case the host is spinning on IN-NAK
  • 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>Time</th>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Poll</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:06.625</td>
<td>353613</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353617</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [7 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353622</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [39 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353627</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [7 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353632</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [7 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353637</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
<td>[1 SOF]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353638</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
<td>IN txn</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353642</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [39 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353647</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [6 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353652</td>
<td>512 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [6 POLL]</td>
</tr>
<tr>
<td>0:06.625</td>
<td>353657</td>
<td>511 B</td>
<td>03</td>
<td>01</td>
<td>IN txn [6 POLL]</td>
</tr>
</tbody>
</table>

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

Same Transfer, but with first two transactions open

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Time</th>
<th>Data</th>
<th>Type</th>
<th>Packet Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>353617</td>
<td>0:06.625.343</td>
<td>511 B</td>
<td>IN txn [7 POLL]</td>
<td></td>
</tr>
<tr>
<td>353622</td>
<td>0:06.625.363</td>
<td>512 B</td>
<td>IN txn [39 POLL]</td>
<td></td>
</tr>
<tr>
<td>353623</td>
<td>0:06.625.363</td>
<td>39.4 us</td>
<td>[39 IN-NAK]</td>
<td></td>
</tr>
<tr>
<td>353624</td>
<td>0:06.625.404</td>
<td>3 B</td>
<td>IN packet</td>
<td></td>
</tr>
<tr>
<td>353625</td>
<td>0:06.625.404</td>
<td>515 B</td>
<td>DATA0 packet</td>
<td></td>
</tr>
<tr>
<td>353626</td>
<td>0:06.625.413</td>
<td>1 B</td>
<td>ACK packet</td>
<td></td>
</tr>
<tr>
<td>353627</td>
<td>0:06.625.414</td>
<td>512 B</td>
<td>IN txn [7 POLL]</td>
<td></td>
</tr>
<tr>
<td>353628</td>
<td>0:06.625.414</td>
<td>6.61 us</td>
<td>[7 IN-NAK]</td>
<td></td>
</tr>
<tr>
<td>353629</td>
<td>0:06.625.421</td>
<td>3 B</td>
<td>IN packet</td>
<td></td>
</tr>
<tr>
<td>353630</td>
<td>0:06.625.422</td>
<td>515 B</td>
<td>DATA1 packet</td>
<td></td>
</tr>
<tr>
<td>353631</td>
<td>0:06.625.431</td>
<td>1 B</td>
<td>ACK packet</td>
<td></td>
</tr>
<tr>
<td>353632</td>
<td>0:06.625.432</td>
<td>512 B</td>
<td>IN txn [7 POLL]</td>
<td></td>
</tr>
<tr>
<td>353637</td>
<td>0:06.625.456</td>
<td>66 ns</td>
<td>[1 SOF]</td>
<td></td>
</tr>
<tr>
<td>353638</td>
<td>0:06.625.457</td>
<td>511 B</td>
<td>IN txn</td>
<td></td>
</tr>
<tr>
<td>353642</td>
<td>0:06.625.471</td>
<td>512 B</td>
<td>IN txn [39 POLL]</td>
<td></td>
</tr>
<tr>
<td>353683</td>
<td>0:06.625.705</td>
<td>83 ns</td>
<td>[1 SOF]</td>
<td></td>
</tr>
</tbody>
</table>

First Transaction

- 39.4 us lost between transfers

Only 6.6 us lost between transactions

Single Transfer

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 takes transfers from a user space process synchronously.
      - Synchronous $\rightarrow$ delay between transfers
      - Larger transfers $\rightarrow$ fewer trips to user space
    - It would be better to queue packets on the device side inside the kernel.
      - Queuing can happen even when the hardware is busy.
      - Currently requires a custom driver.
Custom Driver

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

• Performance
  • BeagleBone Black:
    – 227 Mbit/sec, ~7.6% better than functionfs
  • EG20T:
    – 328 Mbit/sec, ~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*
musb Results (IN Transactions)

![Graph showing musb results with different transfer sizes and transactions types.]
EG20T Results (IN Transactions)

![Graph showing Mbit/sec vs Transfer Size for different scenarios: Driver (65535), 65536, 1024, 512, 64. The legend includes: Idle Sync, Idle Async, Load (Device) Sync, Load (Device) Async, Load (Host) Sync, Load (Host) Async, Idle Fast Sync, Idle Fast Async.]
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
  - Host Load doesn't make much difference
  - Device Load makes more difference
    - Data is sourced from user space
PIC32MX Results (IN Transactions)

Transfer Size

- 65536
- 1024
- 512
- 64
- 32

Mbit/sec

- Idle Sync
- Idle Async
- Load (Host) Sync
- Load (Host) Async
PIC32MX Results (IN TXN 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>7.0</td>
</tr>
<tr>
<td>64</td>
<td>6.0</td>
</tr>
<tr>
<td>32</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Legend:
- 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 Transactions)

Transfer Size

- 65536
- 1024
- 512
- 64

Mbit/sec

- Idle Sync
- Idle Async
- Load (Device) Sync
- Load (Device) Async
- Load (Host) Sync
- Load (Host) Async
EG20T Results (OUT Transactions)

Transfer Size

Mbit/sec

EG20T Results (OUT Transactions)
Results

• musb/EG20T (OUT) Analysis
  • musb does one DMA transfer per USB transaction.
  • 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 Transactions)

Transfer Size

<table>
<thead>
<tr>
<th>Transfer Size</th>
<th>32</th>
<th>64</th>
<th>512</th>
<th>1024</th>
<th>65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>64</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1024</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>65536</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Mbit/sec

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

<table>
<thead>
<tr>
<th>Transfer Size</th>
<th>65536</th>
<th>1024</th>
<th>512</th>
<th>64</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mbit/sec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle Sync</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Idle Async</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Load (Host) Sync</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Load (Host) Async</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Embedded Linux Conference
Results

- PIC32MX (Output) Analysis
  - OUT transfers are affected by the hub the same way IN transactions are
  - Speed is *comparable* to IN transfers
Further Optimizations
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!
          - 8 kB/sec in this case
  - Use Bulk/Isoc endpoints!
    - Use libusb on the host side
Common Pitfalls

• Serial Gadget
  • The `f_serial` gadget interface 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.
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 call stack
    - 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
KernelShark

Filtered for musb
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
- Function doesn't support AltSettings
  - No Isochronous endpoints
  - No high-bandwidth Interrupt endpoints
- Hubs
  - Can have strange effects
  - Some good, some bad.
Signal 11
SOFTWARE

Alan Ott
alan@signal11.us
www.signal11.us
+1 407-222-6975 (GMT -5)

[Image of a penguin]