Android Connectivity
USB Host Part 1

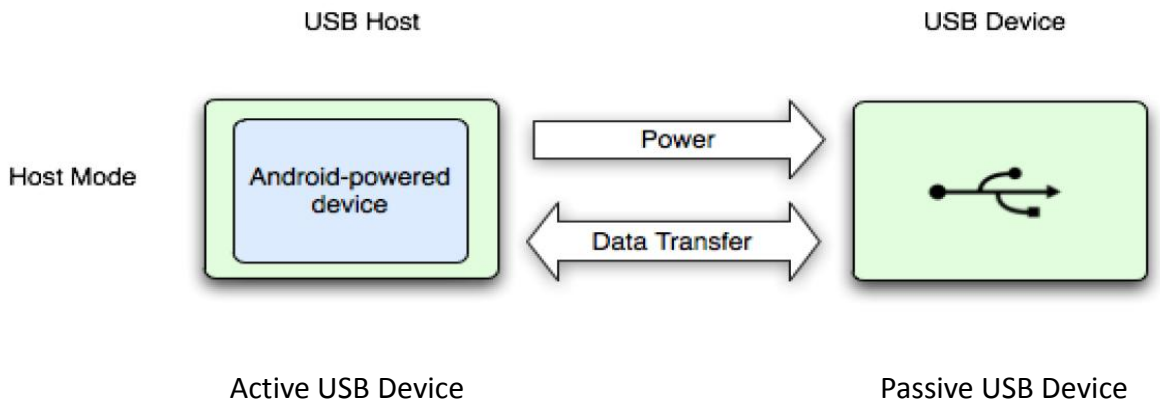
By Shinping R. Wang
CSIE Dept. of Da-yeh University

Reference

- The following discussion is excerpted from
  1. Android Developer: http://developer.android.com/guide/topics/connectivity/usb/host.html

USB Host

- “When your Android-powered device is in USB host mode, it acts as the USB host, powers the bus, and enumerates connected USB devices.”
- USB host mode is supported in Android 3.1 and higher.
  http://developer.android.com/guide/topics/connectivity/usb/host.html



USB Host

- Examples?
  - Passive device, such as Missile Launcher, USB memory stick,

USB Host APIs

- USB Host mode APIs are directly supported in
  - ✓ **Platform API – android.hardware.usb** , since Android 3.1 (API Level 12). The USB host APIs are **not present** on earlier API levels.

Note: Support for USB host and accessory modes are ultimately dependant on the device's hardware, regardless of platform level. You can filter for devices that support USB host and accessory through a <uses-feature> element. See the USB accessory and host documentation for more details.

USB host APIs

- The following table describes the USB host APIs in the android.hardware.usb package.

Class	Description
UsbManager	Allows you to enumerate and communicate with connected USB devices.
UsbDevice	Represents a connected USB device and contains methods to access its identifying information, interfaces, and endpoints.
UsbInterface	Represents an interface of a USB device, which defines a set of functionality for the device. A device can have one or more interfaces on which to communicate on.
UsbEndpoint	Represents an interface endpoint, which is a communication channel for this interface. An interface can have one or more endpoints, and usually has input and output endpoints for two-way communication with the device.
UsbDeviceConnection	Represents a connection to the device, which transfers data on endpoints. This class allows you to send data back and forth synchronously or asynchronously.
UsbRequest	Represents an ***asynchronous*** request to communicate with a device through a UsbDeviceConnection.

- In general, you need all of these APIs except the UsbRequest which is for asynchronous request only.

USB host APIs

Class	Description
UsbManager	Allows you to enumerate and communicate with connected USB devices.
UsbDevice	Represents a connected USB device and contains methods to access its identifying information, interfaces, and endpoints.
UsbInterface	Represents an interface of a USB device, which defines a set of functionality for the device. A device can have one or more interfaces on which to communicate on.
UsbEndpoint	Represents an interface endpoint, which is a communication channel for this interface. An interface can have one or more endpoints, and usually has input and output endpoints for two-way communication with the device.
UsbDeviceConnection	Represents a connection to the device, which transfers data on endpoints. This class allows you to send data back and forth synchronously or asynchronously.
UsbRequest	Represents an ***asynchronous*** request to communicate with a device through a UsbDeviceConnection.

UsbRequest is only required if you are doing asynchronous communication

Basics of USB

- Universal Serial Buses is probable the most complex bus design ever….
- http://today.java.net/pub/a/today/2006/07/06/java-and-usb.html

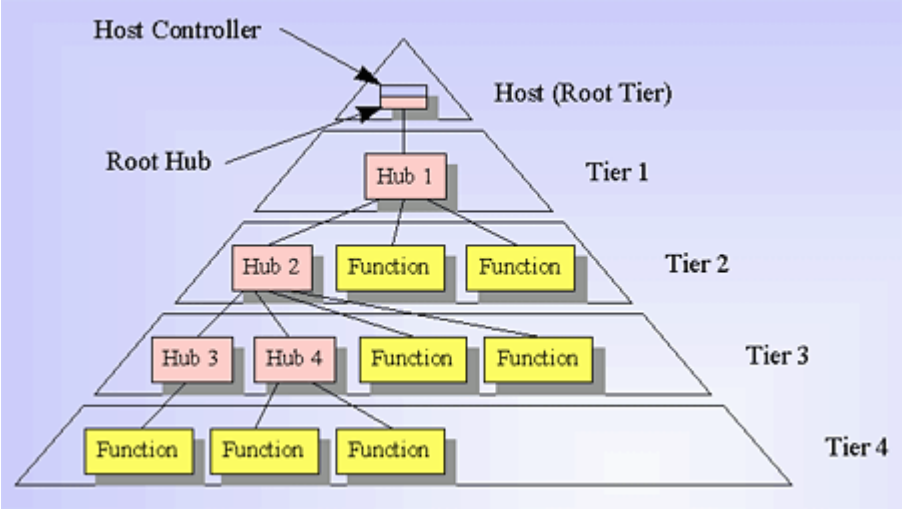


Figure 1. Three external hubs (Hub 1/Hub 2/Hub 3 or Hub 1/Hub 2/Hub 4) are chained together from the root hub

Basics of USB

- http://today.java.net/pub/a/today/2006/07/06/java-and-usb.html

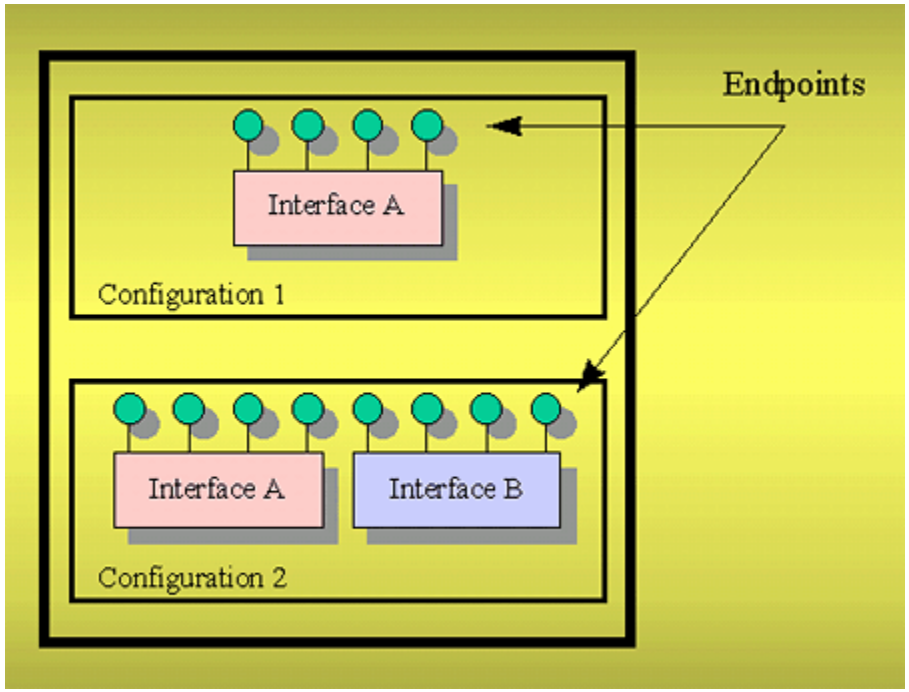


Figure 2. The relationship between a function's configurations, interfaces, and endpoints

Basics of USB

- **Control Transfer**
  - Control Transfer is mainly intended to support configuration, command and status operations between the software on the host and the device.
  - This transfer type is used for low-, full- and high-speed devices.
  - Each USB device has at least one control pipe (default pipe)
  - Control transfer is bursty, non-periodic communication.
  - The control pipe is bi-directional – i.e., data can flow in both directions.
  - Control transfer has a robust error detection, recovery and retransmission mechanism and retries are made without the involvement of the driver.
  - The maximum packet size for control endpoints can be only 8 bytes for low-speed devices; 8, 16, 32, or 64 bytes for full-speed devices; and only 64 bytes for high-speed devices.

pipe is a logical endpoint at the host side

Basics of USB

- **Isochronous Transfer**
  - Isochronous Transfer is most commonly used for time-dependent information, such as multimedia streams and telephony.
  - This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.
  - Isochronous transfer is periodic and continuous.
  - The isochronous pipe is unidirectional, i.e., a certain endpoint can either transmit or receive information. Bi-directional isochronous communication requires two isochronous pipes, one in each direction.
  - USB guarantees the isochronous transfer access to the USB bandwidth (i.e., it reserves the required amount of bytes of the USB frame) with bounded latency, and guarantees the data transfer rate through the pipe, unless there is less data transmitted.
  - Since timeliness is more important than correctness in this type of transfer, no retries are made in case of error in the data transfer. However, the data receiver can determine that an error occurred on the bus.

Basics of USB

- **Interrupt Transfer**
  - Interrupt Transfer is intended for devices that send and receive small amounts of data ***infrequently or in an asynchronous time frame***.
  - This transfer type can be used for low-, full- and high-speed devices.
  - Interrupt transfer type guarantees a maximum service period and that delivery will be re-attempted in the next period if there is an error on the bus.
  - The interrupt pipe, like the isochronous pipe, is unidirectional and periodical.
  - The maximum packet size for interrupt endpoints can be 8 bytes or less for low-speed devices; 64 bytes or less for full-speed devices; and 1,024 bytes or less for high-speed devices.

Basics of USB

- **Bulk Transfer**
  - Bulk Transfer is typically used for devices that transfer large amounts of non-time sensitive data, and that can use any available bandwidth, such as printers and scanners.
  - This transfer type can be used by full-speed and high-speed devices, but not by low-speed devices.
  - Bulk transfer is non-periodic, large packet, bursty communication.
  - Bulk transfer allows access to the bus on an "as-available" basis, guarantees the data transfer but not the latency, and provides an error check mechanism with retries attempts. If part of the USB bandwidth is not being used for other transfers, the system will use it for bulk transfer.
  - Like the other stream pipes (isochronous and interrupt), the bulk pipe is also unidirectional, so bi-directional transfers require two endpoints.
  - The maximum packet size for bulk endpoints can be 8, 16, 32, or 64 bytes for full-speed devices, and 512 bytes for high-speed devices.

Basics of USB

- http://today.java.net/pub/a/today/2006/07/06/java-and-usb.html

Data virtually flows horizontally across the corresponding sides of the upper two layers by way of *pipes*, logical channels that associate host software memory buffers with endpoints.

Pipes can be categorized as *message pipes* and *stream pipes*. Message pipes transfer data with some USB structure. In contrast, stream pipes transfer data with no USB structure.

Every HID must have an interrupt IN endpoint for sending data to the host. An interrupt OUT endpoint is optional.

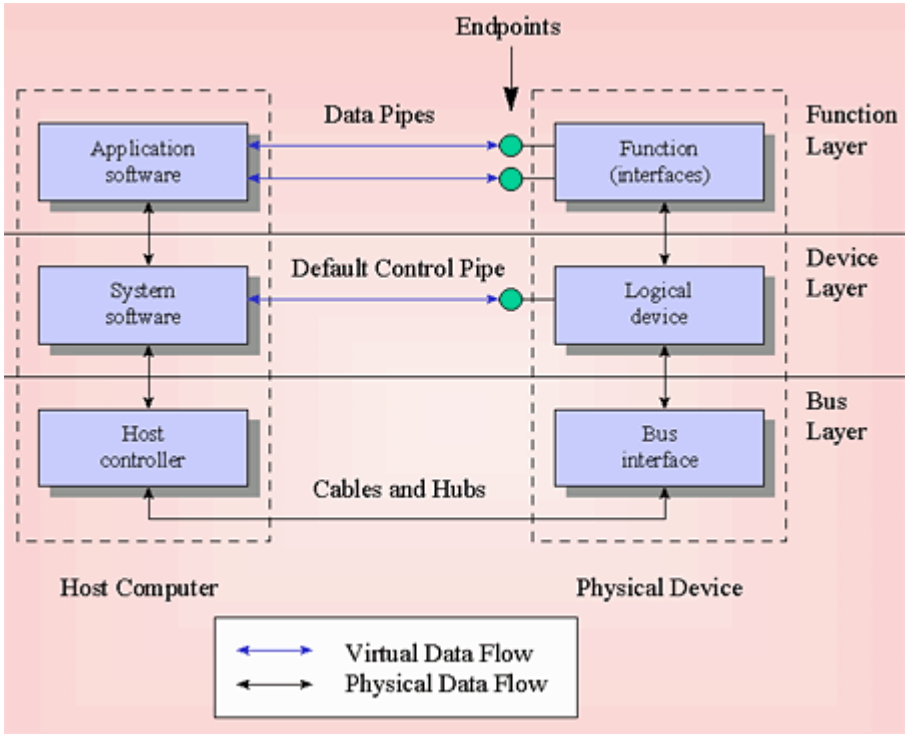


Figure 3. The USB data flow model divides into three layers

Basics of USB

- **Descriptor**
  - Describes device's function (or functions) identify itself (or themselves) to software running on the host:
    - *Device* descriptors
    - *Device qualifier* descriptors
    - *Configuration* descriptors
    - *Other speed configuration* descriptors
    - *Interface* descriptors
    - *Endpoint* descriptors
    - *String* descriptors

http://today.java.net/pub/a/today/2006/07/06/java-and-usb.html

Basics of USB

- **Device Classes**

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video
0Fh	Interface	Personal Healthcare
10h	Interface	Audio/Video Devices
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous

- http://www.usb.org/developers/defined_class

Android Manifest Requirement

- include a

  <uses-feature android:name="android.hardware.usb.host" />

  in the application to assure installed device support USB Host operation.

- Set the minimum SDK of the application to API Level 12 or higher such as

  <uses-sdk android:minSdkVersion="12" />.

  The USB host APIs are not present on earlier API levels.

Android Manifest Requirement

- Specify an ***<intent-filter>*** and ***<meta-data>*** element pair for the ***android.hardware.usb.action.USB_DEVICE_ATTACHED*** intent in the main activity, if you want your application to be notified of an attached USB device.
- The ***<meta-data>*** element points to an external XML resource file that declares identifying information about the device that you want to detect.

Android Manifest Requirement

- In the XML resource file, declare ***<usb-device>*** elements for the USB devices that you want to filter.
- The following list describes the attributes of ***<usb-device>.***

Attribute	
vendor-id	
product-id	
class	
subclass	
protocol (device or interface)	

- Save the resource file in the ***res/xml/*** directory. The resource file name (without the .xml extension) must be the same as the one you specified in the **<meta-data>** element.

Specifying no attributes matches every USB device, so only do this if your application requires it.

Android Manifest Requirement

- A Manifest example

Make sure the platform supports API level 12 or above.

Make sure the platform supports USB Host operation

If you want the system notify you when the target USB device is attached.

Specify which USB device is qualified.

```
<manifest ...>
    <uses-feature android:name="android.hardware.usb.host" />
    <uses-sdk android:minSdkVersion="12" />
     ...
        <application>
            <activity ...>
              ...
                <intent-filter>
                    <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
                </intent-filter>
                <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                        android:resource="@xml/device_filter" />
            </activity>
        </application>
</manifest>
```

Android Manifest Requirement

- A resources file example
- file should be saved in ***res/xml/device_filter.xml***

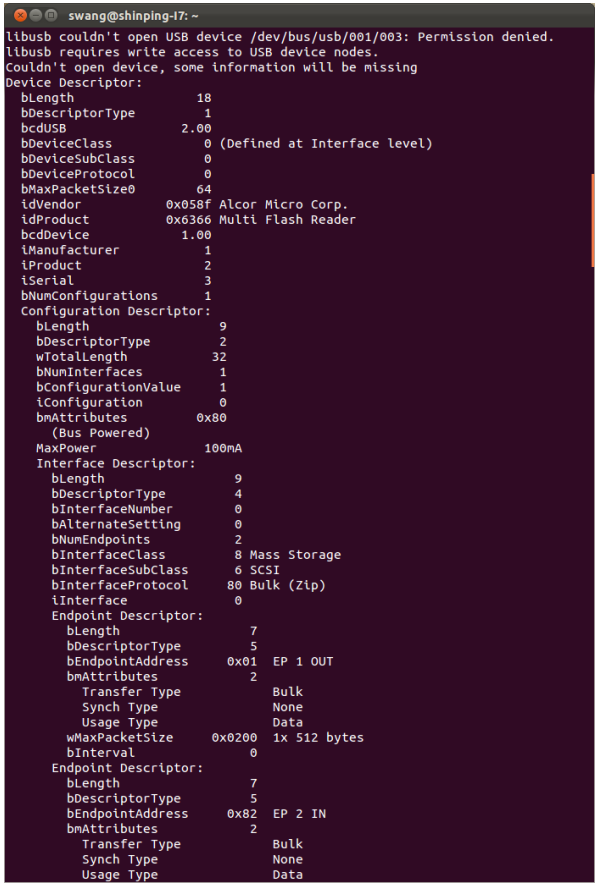
```
<?xml version="1.0" encoding="utf-8"?>
 <resources>
        <usb-device vendor-id="1234" product-id="5678" class="255" subclass="66" protocol="1" />
</resources>
```

Android Manifest Requirement

- Detail information of the connected USB device can be retrieved using

$lsusb –v

Under Linux



Application frameworks

- Application has to:
  - **Discovering a device:** Discover connected USB devices either by system notification or by enumeration.
  - **Request User's Permission:** Ask for user permission to connect to the USB device, if not already obtained.
  - **Perform Communication:** Communicate with the USB device through appropriate interface endpoints.

Discovering/Requesting device permission

- Two ways of Discovering/Requesting device permission
  - Statically
    - Asynchronously
    - Use Androidmanifest file
  - Dynamically
    - At run time
    - Synchronous
    - Use device enumeration and requirePermission()

Discovering/Requesting device permission

- “If your application uses an intent filter to discover USB devices as they're connected, your application will automatically receives permission if the user(owner of the device) allows your application to handle the intent. If not, you must request permission explicitly in your application before connecting to the device.”

  A scenario: the App is informed about the attached device by the system using an Intent, however the user decided not to use the App at the time. Later on, the service of the App is required and the App is started by the user. When this happen, the App has to discover and get permission at run time.

Discovering/Requesting device permission statically

- **Using an intent filter inside of a Androidmanifest file**

An intent with action "android.hardware.usb.action.USB_DEVICE_ATTACHED" will pass the filter and invoke the callback BroadcastReceiver.onReceive() method registered inside of onCreate() of the main Activity.

```
<activity ...>
        ...
        <intent-filter>
                <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
        </intent-filter>
        <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                android:resource="@xml/device_filter" />
</activity>
```

The callback method onReceive() will exam the USB device to make sure it matches the defined resource file.

When users connect a device that matches your device filter, the system presents them with a dialog that asks if they want to start your application. This seems a little bit awkward asking the same person who attach the device if he want to use the device ☹ Well, come to think about it, the permission is for to the App and not the USB peripheral.

Discovering/Requesting device permission at run time

- **Enumerating devices**
  - Use the **getDeviceList()** method of the **UsbManager** to obtain a hash map of all the USB devices that are connected.

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
 ...
HashMap<String, UsbDevice> deviceList = manager.getDeviceList();
UsbDevice device = deviceList.get("deviceName");
```

Get a specific device that under the “deviceName”

Discovering/Requesting device permission at run time

- **Enumerating devices**
  - Use the **getDeviceList()** method of the **UsbManager** to obtain a hash map of all the USB devices that are connected.

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
...
HashMap<String, UsbDevice> deviceList = manager.getDeviceList();
Iterator<UsbDevice> deviceIterator = deviceList.values().iterator();

while(deviceIterator.hasNext()){
        UsbDevice device = deviceIterator.next() //your code
}
```

The values() method returns a object of Collection<UsbDescriptor> that implements interface Iterator<T>

Calling the iterator() returns a Iterator <UsbDevice>

Iterate over the iterator. The method hasNext() resturns a boolean true as long as there are still entries left in the iterator, the next() method returns the next entry of the iterator.
(Hmmm what an intriguing implementation)

Discovering/Requesting device permission at run time

- Getting the permission of using the device at run time
  - Using **getDeviceList()** to find all connected USB device, then iterate over these devices until a match is found†,
  - Using **requestPermission()** to send an request dialog to the user for permission of using the device,
  - At the same time, construct a **BroadcastReceiver()** for the intent send by the user when the above dialog is either granted or denied.

†Actually, for most of the current design, there should be only one USB port available.

Discovering/Requesting device permission at run time

- An example of **BroadcastReceiver()**

```
1 private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
2 private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
3       public void onReceive(Context context, Intent intent) {
4             String action = intent.getAction();
5             if (ACTION_USB_PERMISSION.equals(action)) {
6                   synchronized (this) {
7                         UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
8                         if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
9                               if(device != null){
10                                    //call method to set up device communication
11                              }
12                        } else {
13                              Log.d(TAG, "permission denied for device " + device);
14                        }
15                  }
16          }
17      }
18 };
```

Line 1 defines a static constant string of intent action that will be used the whole program. Noticed that, the action USB.PERMISSION is prefixed with the package name com.android.example. This is for security consideration. Specifically, it avoids creating conflicting action name with other application.

Discovering/Requesting device permission at run time

- An example of **BroadcastReceiver()**

```
1 private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
2 private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
3       public void onReceive(Context context, Intent intent) {
4             String action = intent.getAction();
5             if (ACTION_USB_PERMISSION.equals(action)) {
6                   synchronized (this) {
7                         UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
8                         if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
9                               if(device != null){
10                                    //call method to set up device communication
11                              }
12                        } else {
13                              Log.d(TAG, "permission denied for device " + device);
14                        }
15                  }
16          }
17      }
18 };
```

Line 2-18 instantiate an object of type BroadcastReceiver. Being a abstract class, the onReceive() method is implemented here.

Discovering/Requesting device permission at run time

- An example of **BroadcastReceiver()**

```
1 private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
2 private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
3         public void onReceive(Context context, Intent intent) {
4                 String action = intent.getAction();
5                 if (ACTION_USB_PERMISSION.equals(action)) {
6                         synchronized (this) {
7                                 UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
8                                 if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
9                                         if(device != null){
10                                                //call method to set up device communication
11                                        }
12                                } else {
13                                        Log.d(TAG, "permission denied for device " + device);
14                                }
15                        }
16                }
17        }
18 };
```

Line 3-17, being an abstract class, you will have to implement method onReceive() yourself.
Line 5 checks if the received intent has the same action
Line 6 raises the monitor of the object to avoid racing condition (more than one thread work on the same critical section of the code concurrently.)
Line 7, retrieves the device form the received intent.

Discovering/Requesting device permission at run time

- An example of **BroadcastReceiver()**

```
1 private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
2 private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
3         public void onReceive(Context context, Intent intent) {
4                 String action = intent.getAction();
5                 if (ACTION_USB_PERMISSION.equals(action)) {
6                         synchronized (this) {
7                                 UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
8                                 if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
9                                         if(device != null){
10                                                //call method to set up device communication
11                                        }
12                                } else {
13                                        Log.d(TAG, "permission denied for device " + device);
14                                }
15                        }
16                }
17        }
18 };
```

Line 8 checks if the intent has been granted the user.
Line 9 makes sure a valid device has been retrieved.
Line 10, set up the device for communication.
Line 12-14 provides debugging log when permission is denied.

Discovering/Requesting device permission at run time

- You will have to register the **BroadcastReceiver()** along with the intent filter inside the onCreate().

```
1  UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB_SERVICE);
2  private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
...
3  mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION_USB_PERMISSION), 0);
4  IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);
5  registerReceiver(mUsbReceiver, filter);
```

Line 1 instantiates a UsbManager that the mUsbManager.
Line 2 defines the static string ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION"

Discovering/Requesting device permission at run time

- You will have to register the **BroadcastReceiver()** along with the intent filter inside the onCreate().

```
1  UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB_SERVICE);
2  private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
...
3  mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION_USB_PERMISSION), 0);
4  IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);
5  registerReceiver(mUsbReceiver, filter);
```

Line 3, Noticed, the same action “ACTION_USB_PERMISSION” is used in the PendingIntent as the one given in BroadcastReceiver from previous slides. Doing this way will make sure the intent matches the one given in the registered BroadcastReceive. The instance of PendingIntent will be used in the requestPermission() which will raise a dialog asks for user permission. If the permission is granted, the intent inside of the PendingInternt will be issued.

Requesting device permission at run time

- You will have to register the **BroadcastReceiver()** along with the intent filter as following inside the onCreate().

```
1 UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB_SERVICE);
2 private static final String ACTION_USB_PERMISSION = "com.android.example.USB_PERMISSION";
...
3 mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION_USB_PERMISSION), 0);
4 IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);
5 registerReceiver(mUsbReceiver, filter);
```

Line 5, since the mUsbReceiver is created at runtime, registration is necessary However, if the BroadcastReceiver has been defined statically and registered inside of the Androidmanifest file using <receiver>, they you don't have to register it at run time.

Discovering/Requesting device permission at run time

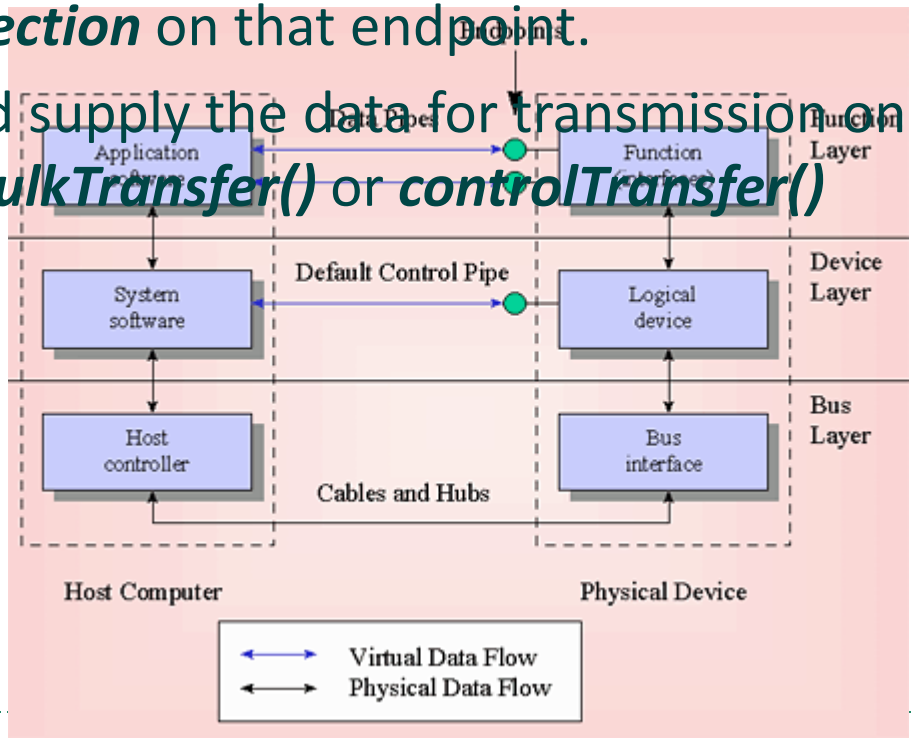
- To display the dialog that asks users for permission to connect to the device, call the **requestPermission()** method:

```
UsbDevice device;
...
mUsbManager.requestPermission(device, mPermissionIntent);
```

  - When users reply to the dialog, a PendingIntent that the mPermissionIntent defined earlier is issued and the broadcast receiver receives the intent. The intent contains the EXTRA_PERMISSION_GRANTED extra, which is a boolean representing the answer. Check this extra for a value of true before connecting to the device.

Communicating with a device

- In general, your code should:
  - Verify the ***UsbDevice*** object's attributes for communication
  - Find the appropriate ***UsbInterface*** and the appropriate ***UsbEndpoint*** of that interface.
  - Open a ***UsbDeviceConnection*** on that endpoint.
  - Create a new thread and supply the data for transmission on the endpoint with the ***bulkTransfer()*** or ***controlTransfer()*** method.



Communicating with a device

- A simple code example

```
1    private Byte[] bytes;
2    private static int TIMEOUT = 0;
3    private boolean forceClaim = true;
...
4    UsbInterface intf = device.getInterface(0);
5    UsbEndpoint endpoint = intf.getEndpoint(0);
6    UsbDeviceConnection connection = mUsbManager.openDevice(device);
7    connection.claimInterface(intf, forceClaim);
8    connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT);     //do in another thread
```

http://developer.android.com/reference/android/hardware/usb/UsbConstants.html

Obtaining permission to communicate with a device

- A simple code example

```
1 private Byte[] bytes;
2 private static int TIMEOUT = 0;
3 private boolean forceClaim = true;
...
4 UsbInterface intf = device.getInterface(0);
5 UsbEndpoint endpoint = intf.getEndpoint(0);
6 UsbDeviceConnection connection = mUsbManager.openDevice(device);
7 connection.claimInterface(intf, forceClaim);
8 connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT); //do in another thread
```

http://developer.android.com/reference/android/hardware/usb/UsbConstants.html

Terminating communication with a device

- A simple code example

An example MissileLauncher

- MissileLauncher is a simple program that controls Dream Cheeky USB missile launchers. You control the left/right/up/down orientation of the launcher using the accelerometer. Tilt the tablet to change the direction of the launcher. Pressing the "Fire" button will fire one missile.
- Reference:
  1. http://matthias.vallentin.net/blog/2007/04/writing-a-linux-kernel-driver-for-an-unknown-usb-device/
  2. http://www.dreamcheeky.com/storm-oic-missile-launcher

An example MissileLauncher

- MissileLauncher serves as an example of the following USB host features:
  - Filtering for multiple devices based on vendor and product IDs (see device_filter.xml)
  - Sending control requests on endpoint zero that contain data
  - Receiving packets on an interrupt endpoint using a thread that calls UsbRequest.queue() and UsbDeviceConnection.requestWait()

MissileLauncher/AndroidManifestfile.xml

```
1   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2     package="com.android.missilelauncher">
3
4     <uses-feature android:name="android.hardware.usb.host" />
5     <uses-sdk android:minSdkVersion="12" />
6
7     <application>
8       <activity android:name="MissileLauncherActivity"
9         android:label="Missile Launcher"
10        android:screenOrientation="nosensor">
11        <intent-filter>
12          <action android:name="android.intent.action.MAIN" />
13          <category android:name="android.intent.category.DEFAULT" />
14          <category android:name="android.intent.category.LAUNCHER" />
15        </intent-filter>
16
17        <intent-filter>
18          <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
19        </intent-filter>
20
21        <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
22          android:resource="@xml/device_filter" />
23      </activity>
24    </application>
25  </manifest>
```

Line 4,5 make sure the device support host mode operation and with proper API version.

Line 18 define the intent will be sent when a specific USB device is attached.

Line 21,22 given the specific device that the corresponding intent is sent.

MissileLauncher/ res/xml/device_filter.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <!-- Copyright (C) 2011 The Android Open Source Project

3       Licensed under the Apache License, Version 2.0 (the "License");
4       you may not use this file except in compliance with the License.
5       You may obtain a copy of the License at

6           http://www.apache.org/licenses/LICENSE-2.0

7       Unless required by applicable law or agreed to in writing, software
8       distributed under the License is distributed on an "AS IS" BASIS,
9       WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
10      See the License for the specific language governing permissions and
11      limitations under the License.
12  -->
13  <resources>
14      <!-- vendor and product ID for Dream Cheeky USB Missle Launcher -->
15      <usb-device vendor-id="2689" product-id="1793" />
16      <!-- vendor and product ID for Dream Cheeky Wireless USB Missle Launcher -->
17      <usb-device vendor-id="2689" product-id="65281" />
18  </resources>
```

Line 15, 17 says the code is for devices with vendor id 2689 and product id of 1793 or 65281.
http://www.dreamcheeky.com/storm-oic-missile-launcher

An example MissileLauncher

```
1       Program license…..
16
17      package com.android.missilelauncher;
18
19      import java.nio.ByteBuffer;
20
21      import android.app.Activity;
22      import android.content.Context;
23      import android.content.Intent;
24      import android.hardware.Sensor;
25      import android.hardware.SensorEvent;
26      import android.hardware.SensorEventListener;
27      import android.hardware.SensorManager;
28      import android.hardware.usb.UsbConstants;
29      import android.hardware.usb.UsbDevice;
30      import android.hardware.usb.UsbDeviceConnection;
31      import android.hardware.usb.UsbEndpoint;
32      import android.hardware.usb.UsbInterface;
33      import android.hardware.usb.UsbManager;
34      import android.hardware.usb.UsbRequest;
35      import android.os.Bundle;
36      import android.util.Log;
37      import android.view.View;
38      import android.widget.Button;
39
```

MissileLauncher

```
40  public class MissileLauncherActivity extends Activity
41      implements View.OnClickListener, Runnable {
42
43      private static final String TAG = "MissileLauncherActivity";
44
45      private Button mFire;
46      private UsbManager mUsbManager;
47      private UsbDevice mDevice;
48      private UsbDeviceConnection mConnection;
49      private UsbEndpoint mEndpointIntr;
50      private SensorManager mSensorManager;
51      private Sensor mGravitySensor;
52
53      // USB control commands
54      private static final int COMMAND_UP = 1;
55      private static final int COMMAND_DOWN = 2;
56      private static final int COMMAND_RIGHT = 4;
57      private static final int COMMAND_LEFT = 8;
58      private static final int COMMAND_FIRE = 16;
59      private static final int COMMAND_STOP = 32;
60      private static final int COMMAND_STATUS = 64;
61
62      // constants for accelerometer orientation
63      private static final int TILT_LEFT = 1;
64      private static final int TILT_RIGHT = 2;
65      private static final int TILT_UP = 4;
66      private static final int TILT_DOWN = 8;
67      private static final double THRESHOLD = 5.0;
```

Line 40-41 declare the class MissileLauncherActivity which extends Activity and implement two interfaces: the View.OnClickListener and Runnable.

Being Runnable means the class can spawn a thread that runs the run() method of the class.

Line 45-51 declare some private USB and Sensor objects.

Line 53-60 define some USB control command constants. These commands are derived using reverse engineering (usning USB Snoppy, see reference 1.)

Line 62-67 define some accelerometer orientation constants.

MissileLauncher/onCreate()

```
68
69          @Override
70          public void onCreate(Bundle savedInstanceState) {
71              super.onCreate(savedInstanceState);
72
73              setContentView(R.layout.launcher);
74              mFire = (Button)findViewById(R.id.fire);
75              mFire.setOnClickListener(this);
76
77              mUsbManager = (UsbManager)getSystemService(Context.USB_SERVICE);
78
79              mSensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
80              mGravitySensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY);
81          }
82
```

Line 70-81 defines the onCreate() callback method.
The method is invoked when the program launched.
The method does several things

Line 73 sets the main activity view
Line 74 sets the button Fire.
Line 75 set the OnClickListener(this) where “this” means this object. Specifically the onClick() method of the object.
Line 77 instances a UsbManager that the mUsbManager
Line 79 instances a SensorManager that mSensorManager
Line 80 instances a GravitySensor

An example MissileLauncher

```
83          @Override
84          public void onPause() {
85              super.onPause();
86              mSensorManager.unregisterListener(mGravityListener);
87          }
88
```

When pause(lost focus), unregister the mGravityListener. Unless you specifically unregister the sensor, the sensor will continuously operate and consume power even when the activity is paused.

An example MissileLauncher

```
89          @Override
90          public void onResume() {
91              super.onResume();
92              mSensorManager.registerListener(mGravityListener, mGravitySensor,
93                      SensorManager.SENSOR_DELAY_NORMAL);
94
95              Intent intent = getIntent();
96              Log.d(TAG, "intent: " + intent);
97              String action = intent.getAction();
98
99              UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
100             if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
101                 setDevice(device);
102             } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
103                 if (mDevice != null && mDevice.equals(device)) {
104                     setDevice(null);
105                 }
106             }
107         }
108
```

Line 90-107 is the onResume() callback, this method is invoked where the activity returns to forground after being paused.
Line 92 register the listener which set the listener mGravityListener† to the sensor mGravitySensor with rate SeneorManager.SENSOR_DELAY_NORMAL.

† mGravityListener is declared as an anonymous class in line 171 of the program.

An example MissileLauncher

```
89          @Override
90          public void onResume() {
91              super.onResume();
92              mSensorManager.registerListener(mGravityListener, mGravitySensor,
93                      SensorManager.SENSOR_DELAY_NORMAL);
94
95              Intent intent = getIntent();
96              Log.d(TAG, "intent: " + intent);
97              String action = intent.getAction();
98
99              UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
100             if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
101                 setDevice(device);
102             } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
103                 if (mDevice != null && mDevice.equals(device)) {
104                     setDevice(null);
105                 }
106             }
107         }
108
```

Line 95-97 retrieve the Intent that starts this activity. It also get the Action form the Intent.
Line 99 get the EXTRA data part of the Intent that under the name of EXTRA_DEVICE, the retrieved value is a Parcelable object of UsbDevice (it is not difficult to understand the operation at all, but it makes you wonder who sent and pack this Intent in the first place? In addition, on line 95 where getIntent() will return the Intent that start this activity… It is noting particular about this except what if the activity had already been started and just returned from onPause()? Will the Intent “sticky” to the program long enough to be get again and again?)

An example MissileLauncher

```
 89          @Override
 90          public void onResume() {
 91             super.onResume();
 92             mSensorManager.registerListener(mGravityListener, mGravitySensor,
 93                   SensorManager.SENSOR_DELAY_NORMAL);
 94
 95             Intent intent = getIntent();
 96             Log.d(TAG, "intent: " + intent);
 97             String action = intent.getAction();
 98
 99             UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
100             if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
101                setDevice(device);
102             } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
103                if (mDevice != null && mDevice.equals(device)) {
104                   setDevice(null);
105                }
106             }
107          }
108
```

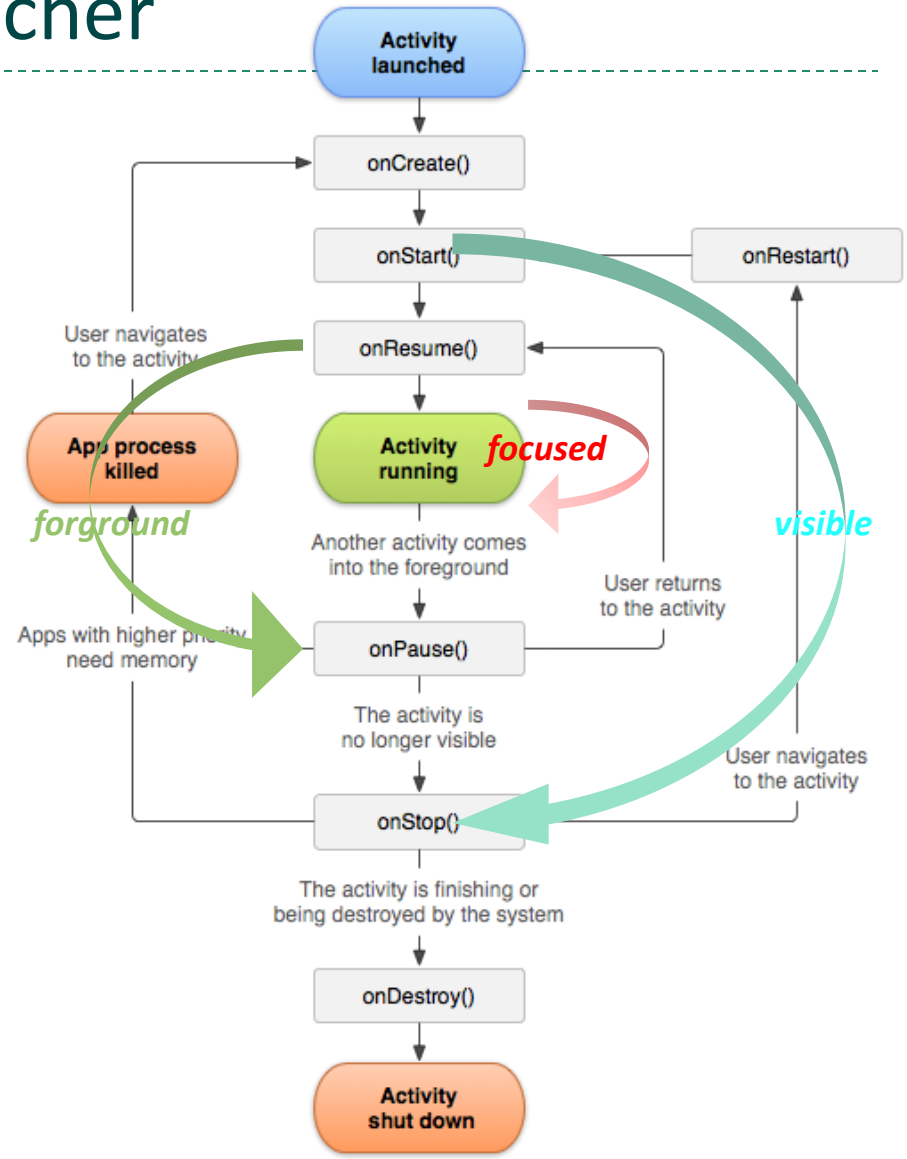
Line 100 checks if the Action of the Intent is a ACTION_USB_DEVICE_ATTACHED. If it is, then line 101 that method setDevice() is called to setup the device.
Line 102 checks if the Action is ACTION_USB_DEVICE_DETACHED, and if the mDevice is define and equals to device(you don't want to remove the wrong device accidentally.) If true, then Line 104 removes the device by calling setDevice() with null argument.

An example MissileLauncher

Some interesting observations:

1. The program dose not define the visible lifetime which consist of onStart() to onStop(). Only 1/3 of program attends it visible lifetime, while most of them given only the forground lifetime.

2. Also the registration of sensor's listener and setup of the USB device are done in the onResume() instead of onCreate(). This is necessary when dealing with exclusive-access resources (camera, USB, accelerator, and etc).



An example MissileLauncher

```
109         @Override
110         public void onDestroy() {
111             super.onDestroy();
112         }
113
```

Line 110-112 define the onDestroy() methods.

MissileLauncher/setDevice()

```
114         private void setDevice(UsbDevice device) {
115             Log.d(TAG, "setDevice " + device);
116             if (device.getInterfaceCount() != 1) {
117                 Log.e(TAG, "could not find interface");
118                 return;
119             }
120             UsbInterface intf = device.getInterface(0);
121             // device should have one endpoint
122             if (intf.getEndpointCount() != 1) {
123                 Log.e(TAG, "could not find endpoint");
124                 return;
125             }
126             // endpoint should be of type interrupt
127             UsbEndpoint ep = intf.getEndpoint(0);
128             if (ep.getType() != UsbConstants.USB_ENDPOINT_XFER_INT) {
129                 Log.e(TAG, "endpoint is not interrupt type");
130                 return;
131             }
132             mDevice = device;
133             mEndpointIntr = ep;
```

Line 114-147 is the setDevice() method of the program. The method eventually sets up a UsbDeviceConnection where all succeeding USB transaction is operated with. The method is called by onResume() to establish a connection when device is attached or perform diss-connection when the device is removed.

The operations carried in this method is canonical and can be adapted to similar program.

Line 116-119 make sure the device has exactly one interface.

Line 120 get the first interface of the device and set it to intf that of type UsbInterface.

Line 122 finds how many endpoints in the given interface and makes sure exactly one endpoint exists.

Line 127 gets the endpoint 0 from the interface and makes it ep.

Line 128 checks if the type of the endpoint is an interrupt one.

MissileLauncher/setDevice()

```
114         private void setDevice(UsbDevice device) {
115             Log.d(TAG, "setDevice " + device);
116             if (device.getInterfaceCount() != 1) {
117                 Log.e(TAG, "could not find interface");
118                 return;
119             }
120             UsbInterface intf = device.getInterface(0);
121             // device should have one endpoint
122             if (intf.getEndpointCount() != 1) {
123                 Log.e(TAG, "could not find endpoint");
124                 return;
125             }
126             // endpoint should be of type interrupt
127             UsbEndpoint ep = intf.getEndpoint(0);
128             if (ep.getType() != UsbConstants.USB_ENDPOINT_XFER_INT) {
129                 Log.e(TAG, "endpoint is not interrupt type");
130                 return;
131             }
132             mDevice = device;
133             mEndpointIntr = ep;
```

If all the above checking are successful, we assign the device to mDevice and ep to mEndpointIntr.

MissileLauncher/setDevice()

```
134             if (device != null) {
135                 UsbDeviceConnection connection = mUsbManager.openDevice(device);
136                 if (connection != null && connection.claimInterface(intf, true)) {
137                     Log.d(TAG, "open SUCCESS");
138                     mConnection = connection;
139                     Thread thread = new Thread(this);
140                     thread.start();
141 
142                 } else {
143                     Log.d(TAG, "open FAIL");
144                     mConnection = null;
145                 }
146             }
147         }
148 
```

Line 134 checks if the device is defined. If it does, then
Line 135 opens the device and assigns it to connection.
Line 136 checks if the connection is null and claims the interface. If success, then
Line 138 assigns the connection to the mConnection,
Line 139 instances a new thread and line 140 starts the threads.
(the thread runs the run() method define in line 216)
If the condition on line 135 if false, then the setDevice() ended by logging open FAIL and set the mConnection to null as shown in line 143 and 146.

MissileLauncher/sendCommand()

```
149         private void sendCommand(int control) {
150             synchronized (this) {
151                 if (control != COMMAND_STATUS) {
152                     Log.d(TAG, "sendMove " + control);
153                 }
154                 if (mConnection != null) {
155                     byte[] message = new byte[1];
156                     message[0] = (byte)control;
157                     // Send command via a control request on endpoint zero
158                     mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);
159                 }
160             }
161         }
162
```

Line 149 -161 is the method sendCommand(), it is invoked in several occasions of the program. Specifically, the newSensorEventLintener() calls the method whenever a new device gesture changed.
It is also called when the Fire button is clicked.
The run() method also invokes two of the sendCommand().

Noticed that, these commands are send from endpoint zero of the device using method controlTransfer().

Line 150 locks this object for synchronization.
Line 151-153 check if the passed command is a COMMAND_STATUS. If it isn't, a move command is issued and the command is logged.

MissileLauncher/sendCommand()

```
149         private void sendCommand(int control) {
150             synchronized (this) {
151                 if (control != COMMAND_STATUS) {
152                     Log.d(TAG, "sendMove " + control);
153                 }
154                 if (mConnection != null) {
155                     byte[] message = new byte[1];
156                     message[0] = (byte)control;
157                     // Send command via a control request on endpoint zero
158                     mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);
159                 }
160             }
161         }
162
```

Line 153 checks if the mConnection is defined. If it is, then line 155-159 are executed.

Line 155 instantiates a byte array of size one and assigns it to message.
Line 156 assigns the first byte of the message to be passed control.
Line 158 stage the command and send it using method controlTrnasfer() which as method of class UsbDeviceConnection.

MissileLauncher/sendCommand()

public int controlTransfer (
int requestType,
int request,
int value,
int index,
byte[] buffer,
int length,
int timeout)

Added in API level 12
Performs a control transaction on endpoint zero for this device. The direction of the transfer is determined by the request type. If requestType & USB_ENDPOINT_DIR_MASK is USB_DIR_OUT, then the transfer is a write, and if it is USB_DIR_IN, then the transfer is a read.

public static final int USB_ENDPOINT_DIR_MASK
Added in API level 12
Bitmask used for extracting the UsbEndpoint direction from its address field.

USB_ENDPOINT_DIR_MASK :	Constant Value: 128 (0x00000080)
USB_DIR_OUT :	Constant Value: 0 (0x00000000)
USB_DIR_IN:	Constant Value: 128 (0x00000080)

MissileLauncher/sendCommand()

```
public int controlTransfer (
            int requestType,
            int request,
            int value,
            int index,
            byte[] buffer,
            int length,
            int timeout             )
```

This is form the Android Developer Document and it is really not very helpful ☹

```
158         mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);
```

Parameters		
requestType	request type for this transaction	0x21,
Request	request ID for this transaction	0x9,
value	value field for this transaction	0x200,
index	index field for this transaction	0,
buffer	buffer for data portion of transaction, or null if no data needs to be sent or received	message,
length	the length of the data to send or receive	message.length,
timeout	in milliseconds	0,

Returns

length of data transferred (or zero) for success, or negative value for failure

MissileLauncher/sendCommand()

Table 9-2. Format of Setup Data

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

See, http://stackoverflow.com/questions/10467846/explanation-about-controltransfer-in-android-to-set-up-the-usb-communication and Section 9.3 USB Device Requests of USB Specification 2.0.

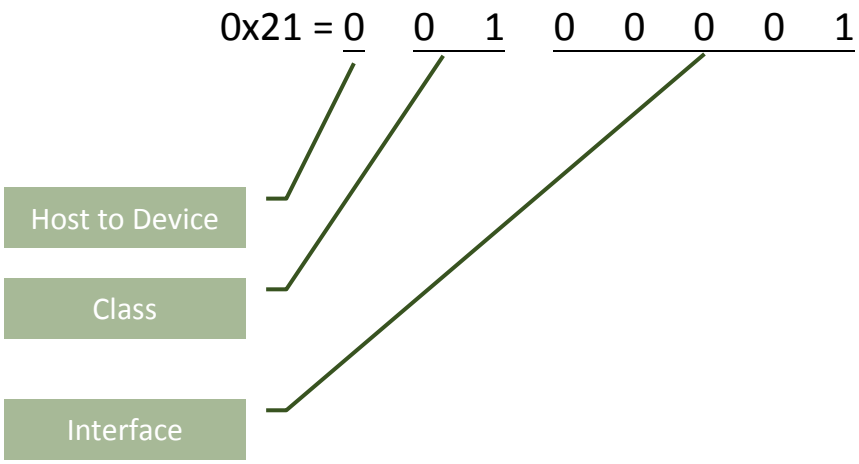
MissileLauncher/sendCommand()

bmRequestType 158 mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);

0x21 = 00100001B
Characteristics of request:
D7: Data transfer direction
- 0 = Host-to-device
- 1 = Device-to-host

D6...5: Type
- 0 = Standard
- 1 = Class
- 2 = Vendor
- 3 = Reserved

D4...0: Recipient
- 0 = Device
- 1 = Interface
- 2 = Endpoint
- 3 = Other
- 4...31 = Reserved



This request 0x21 is from host to device, a class request, and directed to an Interface.

MissileLauncher/sendCommand()

bRequest 158 mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);

0x9 = 00001001B
Presumably the MissileLauncher is a HID type class,
The page 52 of Device Class Definition for Human Interface Device(HID) Version 1.1† says the request 0x9 is a Set_Report Request:

† http://www.usb.org/developers/devclass_docs/HID1_11.pdf

MissileLauncher/sendCommand()

The page 52 of Device Class Definition for Human Interface Device(HID) Version 1.1† says the request 0x9 is a Set_Report Request:

7.2.2 Set_Report Request

Description: The Set_Report request allows the host to send a report to the device, possibly setting the state of input, output, or feature controls.

Part	**Description**
bmRequestType	*00100001*
bRequest	*SET_REPORT*
wValue	*Report Type and Report ID*
wIndex	*Interface*
wLength	*Report Length*
Data	*Report*

† http://www.usb.org/developers/devclass_docs/HID1_11.pdf

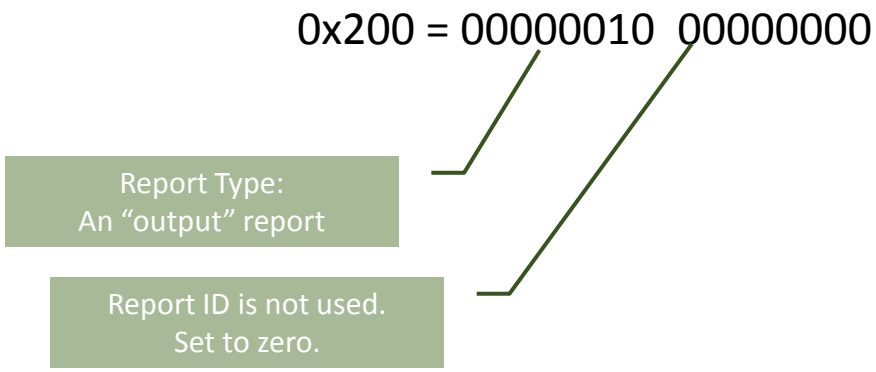
MissileLauncher/sendCommand()

bRequest 158 mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);

0x200 = 00000010 00000000B
The *wValue (bRequest) field specifies the Report Type in the high byte and the Report* ID in the low byte. Set Report ID to 0 (zero) if Report IDs are not used.

Possible Report Types are specified as follows:

Value	Report Type
01	Input
02	***Output (seems redundant since it has be declared as a host to device)***
03	Feature
04-FF	Reserved



MissileLauncher/sendCommand()

wIndex 158 mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);

The value 0 means the request is for interface 0.

Part	**Description**
bmRequestType	*00100001*
bRequest	*SET_REPORT*
wValue	*Report Type and Report ID*
wIndex	*Interface*
wLength	*Report Length*
Data	*Report*

MissileLauncher/sendCommand()

Data 158 mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);

The message argument corresponds to the data in the request and
The message.length corresponds to the *wLength(Report Length)* of the request.
Do noticed the order differ to the definition given in the Spec.

Part	**Description**
bmRequestType	*00100001*
bRequest	*SET_REPORT*
wValue	*Report Type and Report ID*
wIndex	*Interface*
wLength	*Report Length*
Data	*Report*

MissileLauncher/sendCommand()

Timeout 158 mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);

Timeout value is not in the Request syntax given by the Spec.
It is Android specific that define how long the request will be sustained before expired.

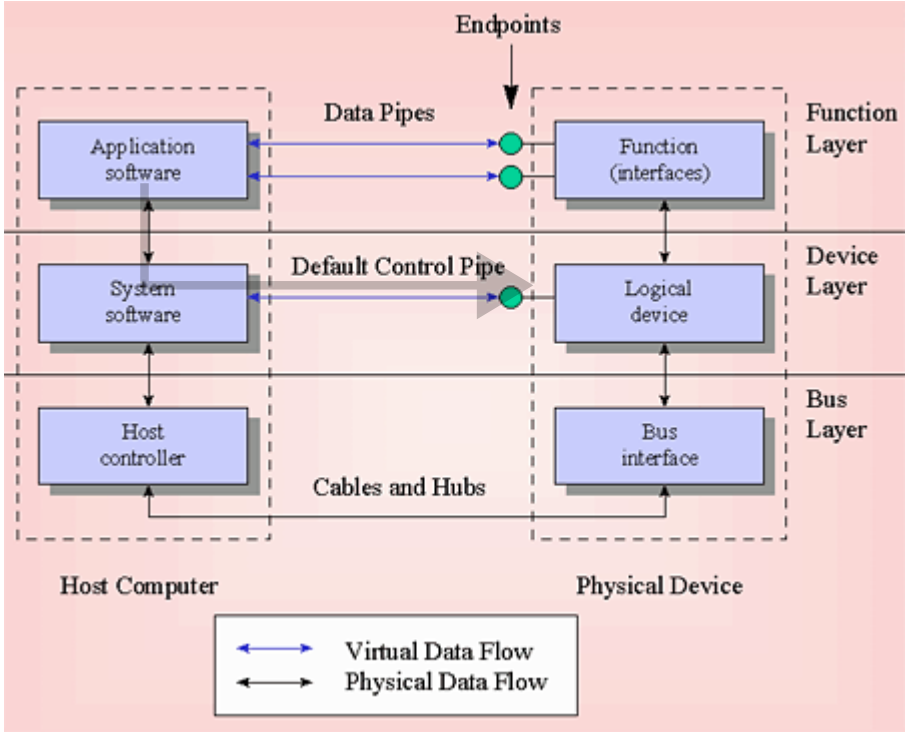
Part	**Description**
bmRequestType	*00100001*
bRequest	*SET_REPORT*
wValue	*Report Type and Report ID*
wIndex	*Interface*
wLength	*Report Length*
Data	*Report*

MissileLauncher/sendCommand()

```
158        mConnection.controlTransfer(0x21, 0x9, 0x200, 0, message, message.length, 0);
```

In all, the line define a USB request packet that a host to device SET_REPORT packet with report data in message which will be sent to the device.

In short, the controlTransfer() send a wrapped USB SET_REPORT packet to the USB device through control endpoint 0.



MissileLauncher/onClick()

```
163         public void onClick(View v) {
164             if (v == mFire) {
165                 sendCommand(COMMAND_FIRE);
166             }
167         }
168
```

Line 163-167 is the callback onClick() method which is invoked when the “Fire” button is clicked.
Do notice that, the Activity implements android.view.View.OnClickListener. This means when a view is clicked the action defined in the callback onClick() will be invoked automatically. So, what this block does is just for that.

Line 164 makes sure the mFire is clicked. If true, then
Line 165 sends out a command COMMAND_FIRE through sendCommand() method.

MissileLauncher/mGravityListener()

```
169         private int mLastValue = 0;
170
171         SensorEventListener mGravityListener = new SensorEventListener() {
172             public void onSensorChanged(SensorEvent event) {
173
174                 // compute current tilt
175                 int value = 0;
176                 if (event.values[0] < -THRESHOLD) {
177                     value += TILT_LEFT;
178                 } else if (event.values[0] > THRESHOLD) {
179                     value += TILT_RIGHT;
180                 }
181                 if (event.values[1] < -THRESHOLD) {
182                     value += TILT_UP;
183                 } else if (event.values[1] > THRESHOLD) {
184                     value += TILT_DOWN;
185                 }
186
```

Line 171-213 give an inner class mGravityListener which implements the interface SensorEventListener.

Two callback methods are defined and they are,

Line 172-208 is the onSensorChange() callback method.

Line 210-212 is the onAccuracyChanaged() callback method.

The class receiving broadcast from the SensorManager when sensor values have changed. This is accomplished though proper callbacks.

MissileLauncher/mGravityListener()

```
169         private int mLastValue = 0;
170
171         SensorEventListener mGravityListener = new SensorEventListener() {
172             public void onSensorChanged(SensorEvent event) {
173
174                 // compute current tilt
175                 int value = 0;
176                 if (event.values[0] < -THRESHOLD) {
177                     value += TILT_LEFT;
178                 } else if (event.values[0] > THRESHOLD) {
179                     value += TILT_RIGHT;
180                 }
181                 if (event.values[1] < -THRESHOLD) {
182                     value += TILT_UP;
183                 } else if (event.values[1] > THRESHOLD) {
184                     value += TILT_DOWN;
185                 }
186
```

Line 172-208 is the onSensorChange() callback method.

This callback is invoked when sensor value have changed. Do notice that, in the callback onResume(), the gravity sensor had been registered. The registration is done at lien 80. So, this says the listener will focus on the gravity sensor.

Line 169 defines and imitates an int value mLastValue to be zero.

Line 176-158 determine which event take placed: TILT LEFT/RIGHT or UP/DOWN one at a time. (what is the sampling rate of these sensor?)

MissileLauncher/mGravityListener()

```
187                 if (value != mLastValue) {
188                     mLastValue = value;
189                     // send motion command if the tilt changed
190                     switch (value) {
191                         case TILT_LEFT:
192                             sendCommand(COMMAND_LEFT);
193                             break;
194                         case TILT_RIGHT:
195                           sendCommand(COMMAND_RIGHT);
196                             break;
197                         case TILT_UP:
198                             sendCommand(COMMAND_UP);
199                             break;
200                         case TILT_DOWN:
201                             sendCommand(COMMAND_DOWN);
202                             break;
203                         default:
204                             sendCommand(COMMAND_STOP);
205                             break;
206                     }
207                 }
208             }
209
```

Line 187 checks if value changes. This is made be comparing the current value to the last value that mLastValue. (This implies the sensor will send event even if the state of the sensor is unchanged. It sounds like the sensor is time-driven, or the system polls each sensor periodically)

If the values do not match, then the mLastValue is updated and the corresponding command is sent accordingly as in line 190 to 206.

MissileLauncher/mGravityListener()

```
210             public void onAccuracyChanged(Sensor sensor, int accuracy) {
211                 // ignore
212             }
213         };
214
```

Line 210 is the callback method onAccuracyChanged() which is not implemented.

MissileLauncher/run()

```
215         @Override
216         public void run() {
217             ByteBuffer buffer = ByteBuffer.allocate(1);
218             UsbRequest request = new UsbRequest();
219             request.initialize(mConnection, mEndpointIntr);
220             byte status = -1;
221             while (true) {
222                 // queue a request on the interrupt endpoint
223                 request.queue(buffer, 1);
224                 // send poll status command
225                 sendCommand(COMMAND_STATUS);
226
```

The USB host controller(HC) will periodically polls the device to see if any interrupt is pending. This is from section 5.7 of the USB spec 2.0 “An interrupt pipe is a stream pipe and is therefore always uni-directional. An endpoint description identifies whether a given interrupt pipe’s communication flow is into or out of the host.”

Examining the interrupt endpoint descriptor of the Missile Launcher shows the endpoint is an IN endpoint.

Line 216 to 246 give the method run(). This is the part of the program constitute the operation of thread and is executed when thread.start() is invoked inside of the setDevice(). It happens when a connection to the device and claiming of the interface are accomplished. Once this is done inside of the setDevice(), a new thread is instantiated and started.

Currently, I believe the run() method performs an IN Request that when COMMAND_STATUS polls the current status of the missile launcher. If the status returned is COMMAND_FIRE, then the run() method send out COMMAN_STOP.

Well, it is difficult to get a full picture of what the program does for lacking of information about what the peer device does (more specifically, what the device responds to each COMMANDs send from the control transfer pipe.)

MissileLauncher/run()

```
215         @Override
216         public void run() {
217             ByteBuffer buffer = ByteBuffer.allocate(1);
218             UsbRequest request = new UsbRequest();
219             request.initialize(mConnection, mEndpointIntr);
220             byte status = -1;
221             while (true) {
222                 // queue a request on the interrupt endpoint
223                 request.queue(buffer, 1);
224                 // send poll status command
225                 sendCommand(COMMAND_STATUS);
226
```

The size of the buffer is one, and this value is implementation dependent. In particular this depends on the buffer size in the Missile Launcher. The value can be obtained using method mEndpointIntr.getMaxPacketSize() as well as the direction using mEcnpointIntr.getdirection().

Line 217 instantiates a byte buffer of size one. (Why the size is one? Again, this is pending on the implementation of the Missile Launcher.)

Line 218 instantiates a request of type UsbRequest.

Line 219 initialize the request with the designated connection and the endpoint. Noticed that, the designated endpoint here is an interruptible endpoint which operates differently from the control endpoint. It uses queue() from UsbRequest() while the control endpoint uses controlTransfer() from UsbConnection().

Line 220 defines and initialize the byte variable status to be -1 Presumably 10000000B.

Line 221- 245 is an infinitive while loop.

Line 223 queue a length 1 byte buffer for receiving/reading IN request on the buffer.

Line 225 send a COMMAND_STATUS through control transfer pipe. Presumably, the command polls the status of the Missile Launcher.

MissileLauncher/run()

```
226                 // wait for status event
227                 if (mConnection.requestWait() == request) {
228                     byte newStatus = buffer.get(0);
229                     if (newStatus != status) {
230                         Log.d(TAG, "got status " + newStatus);
231                         status = newStatus;
232                         if ((status & COMMAND_FIRE) != 0) {
233                             // stop firing
234                             sendCommand(COMMAND_STOP);
235                         }
236                     }
237                     try {
238                         Thread.sleep(100);
239                     } catch (InterruptedException e) {
240                     }
241                 } else {
242                     Log.e(TAG, "requestWait failed, exiting");
243                     break;
244                 }
245             }
246         }
247     }
248
249
```

http://stackoverflow.com/questions/12345953/andoid-usb-host-asynchronous-interrupt-transfer

Line 227, the mConnection.requestWait() waits for the receiving of return request from the interrupt pipe. It then compares the receiving request to the previously queued request. If they are the same, precedes the operation, If not, branch to line 241 and terminate the while() loop. Noticed that, polling the device and getting the result from interrupt pipe could generate measurable latency and this is the part of the code that potentially causing delay and has to be process in a separate working thread from the UI thread.

Line 228 assigned the first byte of the buffer (the request) to the newStatus.

Line 229 compares the newly received newStatus to the status, if they are different which means a new status is updated and line 230- 236 are executed.

Line 230 put a log on the newStatus and line 231 update the status with the newStatus.

Line 232 checks if the returned status is a COMMAND_FIRE.

Line 230 -236 basically check if the status is a COMMAN_FIRE command. If it is, then the COMMAND_STOP is send through the control transfer pipe (although the action COMMAND_STOP is unclear, an educated guess would be that the fire command is an on-off command and has to be toggle each time.)

MissileLauncher/run()

```
226                 // wait for status event
227                 if (mConnection.requestWait() == request) {
228                     byte newStatus = buffer.get(0);
229                     if (newStatus != status) {
230                         Log.d(TAG, "got status " + newStatus);
231                         status = newStatus;
232                         if ((status & COMMAND_FIRE) != 0) {
233                             // stop firing
234                             sendCommand(COMMAND_STOP);
235                         }
236                     }
237                     try {
238                         Thread.sleep(100);
239                     } catch (InterruptedException e) {
240                     }
241                 } else {
242                     Log.e(TAG, "requestWait failed, exiting");
243                     break;
244                 }
245             }
246         }
247     }
248
249
```

Line 234 send the co COMMAND_STOP through the sendComamnd ().

On the other hand, if the newly received status newStatus equals status, then line 230-236 are bypassed and control goes into the try loop line 237-240 directly.

Line 237-240 is a try loop which performs a 0.1 sec sleep and resume the while loop from line222.

If the comparison at line 227 fails which means the receiving request does not match the queue request at line 223, then go to line 241 which would terminate the infinitive while loop and end the run() method and the thread.

Unless a “stop” command (0x00000000) is sent to the device, it keeps the state of the last command that is firing.

Missile Launcher Descriptors

$lsusb –v

```
Bus 004 Device 004: ID 2123:1010
Device Descriptor:
  bLength                 18
  bDescriptorType          1
  bcdUSB                1.10
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x2123
  idProduct          0x1010
  bcdDevice            0.01
  iManufacturer           1 Syntek
  iProduct                2 USB Missile Launcher
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           34
    bNumInterfaces          1
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0x80
      (Bus Powered)
    MaxPower              500mA
```

```
Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           1
      bInterfaceClass         3 Human Interface Device
      bInterfaceSubClass      0 No Subclass
      bInterfaceProtocol      0 None
      iInterface              0
        HID Device Descriptor:
          bLength                 9
          bDescriptorType        33
          bcdHID                1.10
          bCountryCode           0 Not supported
          bNumDescriptors         1
          bDescriptorType        34 Report
          wDescriptorLength      37
         Report Descriptors:
           ** UNAVAILABLE **
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            3
          Transfer Type         Interrupt
          Synch Type            None
          Usage Type            Data
        wMaxPacketSize     0x0008  1x 8 bytes
        bInterval              10
Device Status:     0x0000
  (Bus Powered)
```

http://www.beyondlogic.org/usbnutshell/usb5.shtml#EndpointDescriptors

MissileLauncher