

Android Connectivity

USB Host Part 2

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

Reference

► The following discussion is excerpted from

1. Android Developer:

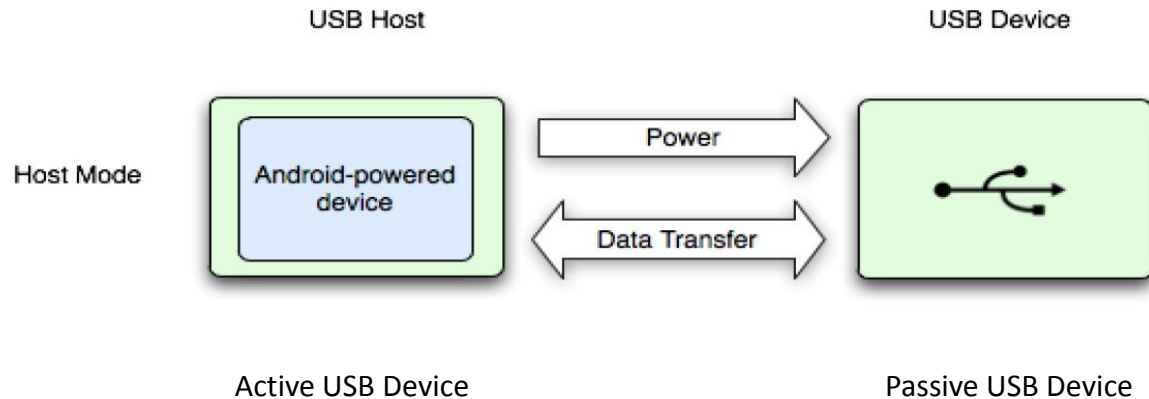
<http://developer.android.com/guide/topics/connectivity/usb/host.html>



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>



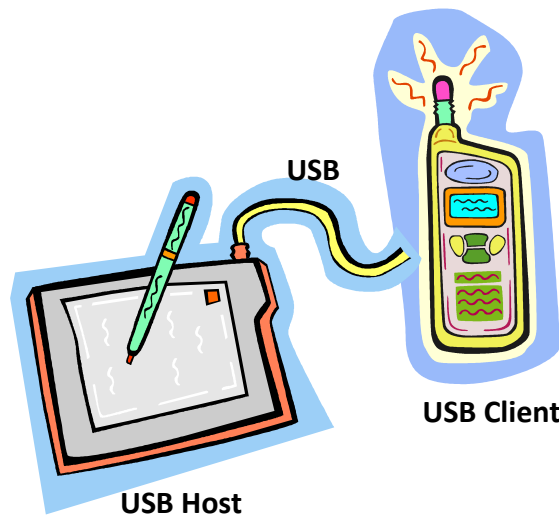
Another Example of USB Host App.

- ▶ The AdbTest



An example AdbTest

- ▶ This App is installed and run on the USB host. The program retrieves log text from the client and display on host's TextView.



An example AdbTest

- This is from the README.txt of the program.

AdbTest is a sample program that implements a subset of the *adb USB protocol*.

Currently it only implements the "*adb logcat*" command and displays the log output in a text view and only allows connecting to one device at a time.

However the support classes are structured in a way that would allow connecting to multiple devices and running multiple adb commands simultaneously.

This program serves as an example of the following USB host features:

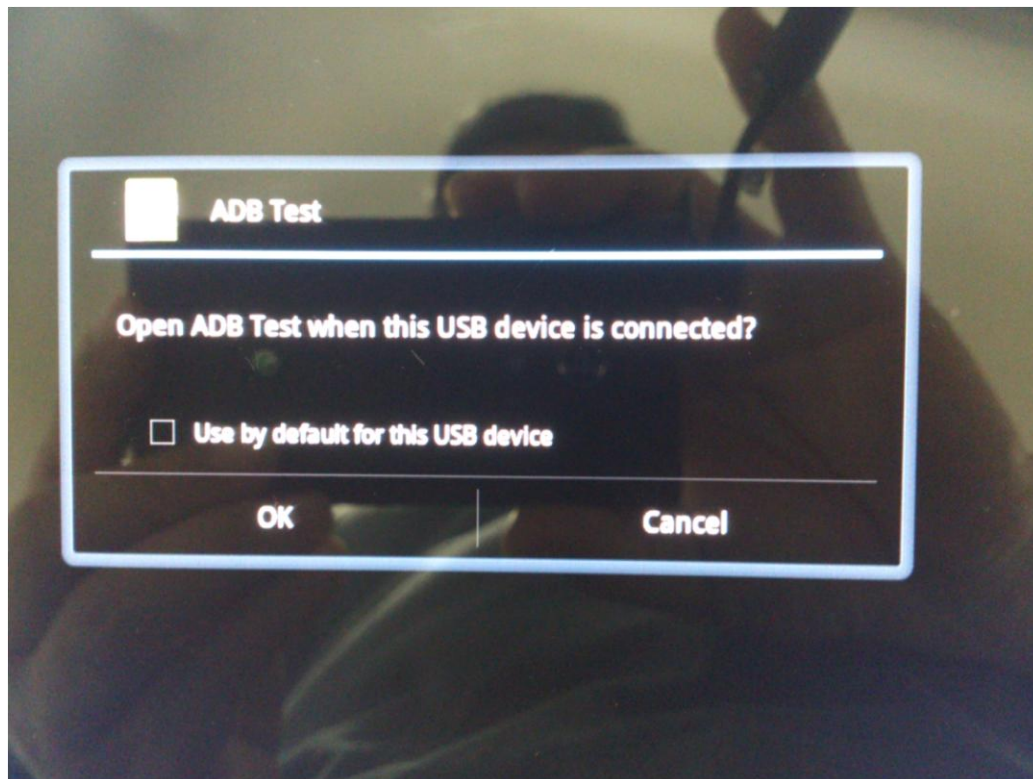
- Matching devices based on interface class, subclass and protocol (see device_filter.xml)
- Asynchronous IO on bulk endpoints

An android power device running in host mode is connected to a device with adb client (all Android device is build with adb capability by default.) The client runs in debug mode that dump all its log through its USB port to the host where AdbTest captures and displays the log. It is nothing new to those Eclipse users who do it all the time when debugging remote device by watching log at the logcat view. The program AdbTest simply runs independently from the Eclipse and shows the log inside the AdbTest instead.



An example AdbTest

- ▶ How does the program run:
 - ▶ When the program starts, it enumerates its connected USB device and asks user's permission to connect.



An example AdbTest

- ▶ How does the program run:
 - ▶ Once connected, the log of the USB client is displayed on the TextView of the Host.

```
I/caladbolg( 156): 915611540 cald_client.c (429) 156 I [INF] - Cald_Client_ICamera_GetBufInfo (0)
I/caladbolg( 156): 915612210 cald_client.c (697) 156 I [INF] + Cald_Client_ICamera_EnableThumbnail
I/caladbolg( 156): 915612290 cald_client.c (714) 156 I [INF] pBufNum[1]
I/caladbolg( 156): 915612352 cald_client.c (720) 156 I [INF] pBuf[0]:0x45839000
I/caladbolg( 156): 915613152 cald_client.c (734) 156 I [INF] - Cald_Client_ICamera_EnableThumbnail (0
I/caladbolg( 156): 915613272 cald_client.c (774) 156 I [INF] + Cald_Client_ICamera_TakeSnapshot
I/caladbolg( 156): 915613356 cald_client.c (791) 156 I [INF] pBufNum[1]
I/caladbolg( 156): 915613418 cald_client.c (797) 156 I [INF] pBuf[0]:0x45103000
I/caladbolg( 156): 915613579 cald_client.c (10755) 4477 I [INF] - Cald_Client_ICamera_TakeSnapshot (0)
I/AwesomePlayer( 156): setDataSource_I(/system/media/audio/camera/sound0/no_sound.m4a) CamCtrl_ICamera_Tak
I/SampleTable( 156): There are reordered frames present. (91) 4477 I [INF] - Cald_IqCtrl_Callback
D/AudioPolicyManager( 156): setStreamVolume() for output 1 stream 0, volume 1.000000, delay 0 CamCtrl_ICBiqCtrl_Pr
E/AudioHardwareMSM8660( 156): unknown stream (91) 4477 I [INF] - Cald_IqCtrl_Callback
D/AudioPolicyManager( 156): setStreamVolume() for output 1 stream 10, volume 1.000000, delay 0 CamCtrl_ICBiqCtrl_Pr
I/Sony Camera HAL( 156): HAL_getCameraInfo: E cameraId=0
I/Sony Camera HAL( 156): HAL_getCameraInfo: X
D/CameraService( 156): getParameter
D/CameraService( 156): getParameter
I/caladbolg( 156): 915610541 cald_client.c (406) 156 I [INF] + Cald_Client_ICamera_GetBufInfo index[0x
I/caladbolg( 156): 915610835 cald_client.c (429) 156 I [INF] - Cald_Client_ICamera_GetBufInfo (0) lock
I/caladbolg( 156): 915610935 cald_client.c (406) 156 I [INF] + Cald_Client_ICamera_GetBufInfo index[0x
I/caladbolg( 156): 915611540 cald_client.c (429) 156 I [INF] - Cald_Client_ICamera_GetBufInfo (0) Sta
I/caladbolg( 156): 915610200 cald_client.c (10697) 4477 I [INF] - Cald_IqCtrl_Callback EnableThumbnail
I/caladbolg( 156): 915610848 cald_camctrl.c (4640) 14477 I [PFM] 915510844 Cald_CamCtrl_ICBiqCtrl_Au
I/caladbolg( 156): 9155112379 cald_iqctrl.c (10755) 4477 I [INF] - Cald_IqCtrl_Callback EnableThumbnail (0
I/AwesomePlayer( 156): setDataSource_I(/system/media/audio/camera/sound0/no_sound.m4a) TakeSnapshot
I/SampleTable( 156): There are reordered frames present. (91) 156 I [INF] - Cald_Client_ICamera_TakeSnapshot
D/AudioPolicyManager( 156): setStreamVolume() for output 1 stream 0, volume 1.000000, delay 0
E/AudioHardwareMSM8660( 156): unknown stream
D/AudioPolicyManager( 156): setStreamVolume() for output 1 stream 10, volume 1.000000, delay 0
I/Sony Camera HAL( 156): HAL_getCameraInfo: E cameraId=0
I/Sony Camera HAL( 156): HAL_getCameraInfo: X
```


An example AdbTest

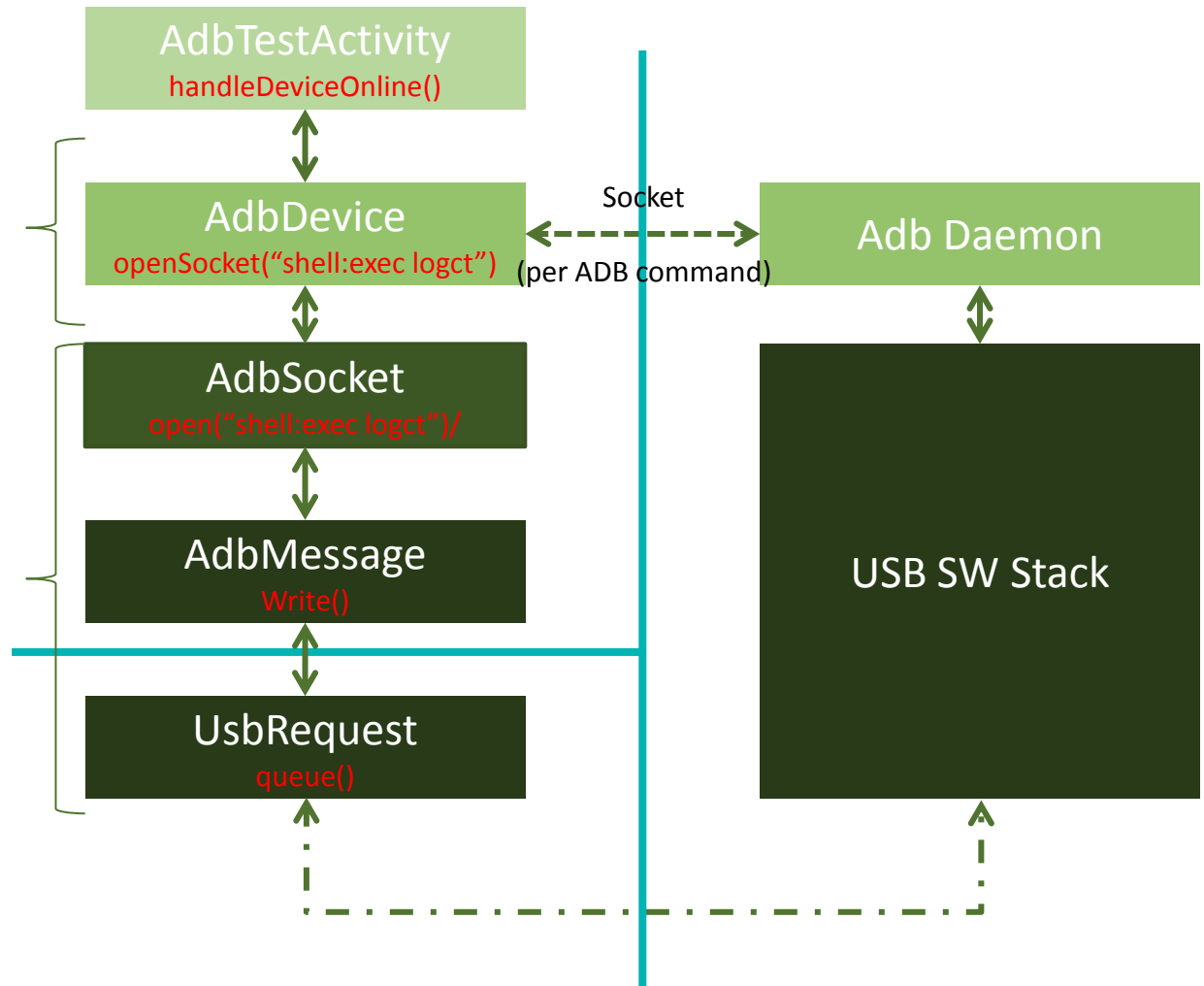
- ▶ The App/package consists of 4 files.
 - ▶ **AdbTestActivity.java**
 - ▶ Main activity for the adb test program.
 - ▶ **AdbDevice.java**
 - ▶ represents a USB device that supports the adb protocol.
 - ▶ **AdbMessage.java**
 - ▶ encapsulates and adb command packet.
 - ▶ **AdbSocket.java**
 - ▶ represents an adb socket. adb supports multiple independent socket connections to a single device. Typically a socket is created for each adb command that is executed.



An example AdbTest

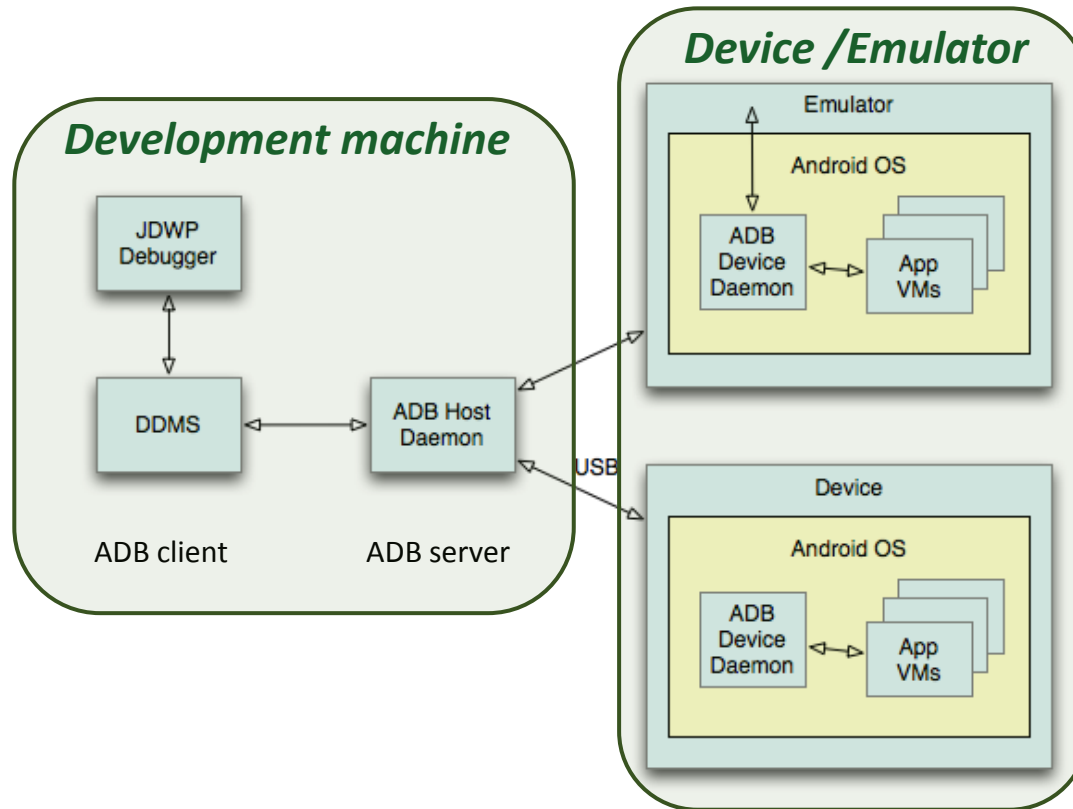
The AdbDevice services as a virtual device that runs all ADB protocol. This will simplify the work of the mina thread that all it has to do is communicating with the virtual device. In the current implementation, only the logcat is implemented.

Layering that provides abstractions that make communication with the peer USB device easy



Android Debug Bridge

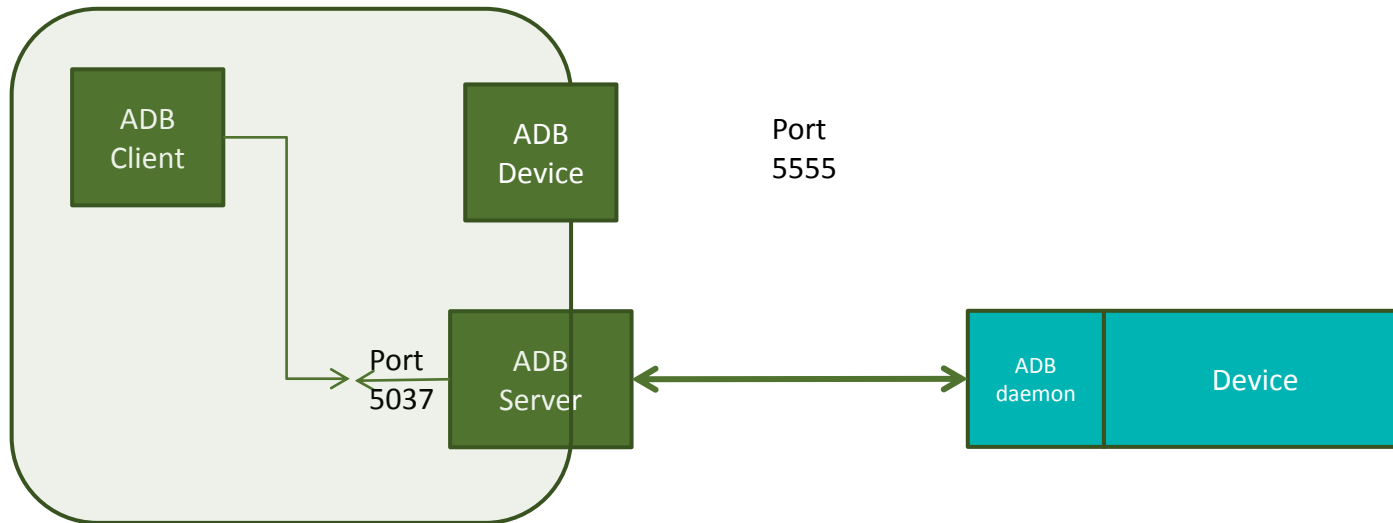
- is a client-server program that includes three components:



The Client, the Server and the Daemon.

Android Debug Bridge

- ▶ Server discovers and sets up connection to the emulator/device instances by scanning ports in the range 5555 to 5585 (the range used by emulators/devices.)



An example AdbTest

► The AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.adb">

    <uses-feature android:name="android.hardware.usb.host" />
    <uses-sdk android:minSdkVersion="12" />

    <application>
        <activity android:name="AdbTestActivity" android:label="ADB Test">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <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>
```

An example AdbTest

► The xml/device_filter.xml

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

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

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

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

-->
<resources>
    <usb-device class="255" subclass="66" protocol="1" />
</resources>
```

About (device) class and subclass http://www.usb.org/developers/defined_class



The Sony LT26W Android 4.0

- ▶ The USB Descriptor of an Android Device
 - ▶ A composite device (two interfaces sharing the same USB bus address)
 - ▶ Interface 0
 - ▶ Class 255 and subclass also 255 that vendor specific
 - ▶ Three endpoints: a pair of Bulk endpoints (IN/OUT) and an Interrupt IN endpoint.
 - ▶ Interface 1
 - ▶ Class 255 and subclass 66 with protocol version 1
 - ▶ has a pair of Bulk endpoints (IN/OUT).
 - ▶ This is the one connected to Android ADB client.



An example AdbTest

Bus 002 Device 006: ID 0fce:5176 Sony Ericsson Mobile Communications AB

Device Descriptor:

bLength	18
bDescriptorType	1
bcdUSB	2.00
bDeviceClass	0 (Defined at Interface level)
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	64
idVendor	0x0fce Sony Ericsson Mobile Communications AB
idProduct	0x5176
bcdDevice	2.31
iManufacturer	2 Sony
iProduct	3 LT26w
iSerial	4 CB5120XEZC
bNumConfigurations	1

Configuration Descriptor:

bLength	9
bDescriptorType	2
wTotalLength	62
bNumInterfaces	2
bConfigurationValue	1
iConfiguration	0
bmAttributes	0xc0
Self Powered	
MaxPower	500mA

Interface Descriptor:

bLength	9
bDescriptorType	4
bInterfaceNumber	0
bAlternateSetting	0
bNumEndpoints	3
bInterfaceClass	255 Vendor Specific Class
bInterfaceSubClass	255 Vendor Specific Subclass
bInterfaceProtocol	0
iInterface	5 MTP

Endpoint Descriptor:

bLength	7
bDescriptorType	5
bEndpointAddress	0x81 EP 1 IN
bmAttributes	2
Transfer Type	Bulk
Synch Type	None
Usage Type	Data
wMaxPacketSize	0x0200 1x 512 bytes
bInterval	0

Endpoint Descriptor:

bLength	7
bDescriptorType	5
bEndpointAddress	0x01 EP 1 OUT
bmAttributes	2
Transfer Type	Bulk
Synch Type	None
Usage Type	Data
wMaxPacketSize	0x0200 1x 512 bytes
bInterval	0

An example AdbTest

Endpoint Descriptor:

bLength	7
bDescriptorType	5
bEndpointAddress	0x82 EP 2 IN
bmAttributes	3
Transfer Type	Interrupt
Synch Type	None
Usage Type	Data
wMaxPacketSize	0x001c 1x 28 bytes
bInterval	6

Interface Descriptor:

bLength	9
bDescriptorType	4
bInterfaceNumber	1
bAlternateSetting	0
bNumEndpoints	2
bInterfaceClass	255 Vendor Specific Class
bInterfaceSubClass	66
bInterfaceProtocol	1
iInterface	0

Endpoint Descriptor:

bLength	7
bDescriptorType	5
bEndpointAddress	0x83 EP 3 IN
bmAttributes	2
Transfer Type	Bulk
Synch Type	None
Usage Type	Data
wMaxPacketSize	0x0200 1x 512 bytes
bInterval	0


Endpoint Descriptor:

bLength	7
bDescriptorType	5
bEndpointAddress	0x02 EP 2 OUT
bmAttributes	2
Transfer Type	Bulk
Synch Type	None
Usage Type	Data
wMaxPacketSize	0x0200 1x 512 bytes
bInterval	0

Device Qualifier (for other device speed):

bLength	10
bDescriptorType	6
bcdUSB	2.00
bDeviceClass	0 (Defined at Interface level)
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	64
bNumConfigurations	1
Device Status:	0x0000
(Bus Powered)	

AdbTestActivity.java



```
1      /*
2      * Copyright (C) 2011 The Android Open Source Project
3      *
4      * Licensed under the Apache License, Version 2.0 (the "License");
5      * you may not use this file except in compliance with the License.
6      * You may obtain a copy of the License at
7      *
8      * http://www.apache.org/licenses/LICENSE-2.0
9      *
10     * Unless required by applicable law or agreed to in writing, software
11     * distributed under the License is distributed on an "AS IS" BASIS,
12     * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13     * See the License for the specific language governing permissions and
14     * limitations under the License.
15     */

16     package com.android.adb;

17     import android.app.Activity;
18     import android.content.BroadcastReceiver;
19     import android.content.Context;
20     import android.content.Intent;
21     import android.content.IntentFilter;
22     import android.graphics.Rect;
23     import android.hardware.usb.UsbDevice;
24     import android.hardware.usb.UsbDeviceConnection;
```



AdbTestActivity.java

```
25     import android.hardware.usb.UsbInterface;
26     import android.hardware.usb.UsbManager;
27     import android.os.Bundle;
28     import android.os.Handler;
29     import android.os.Message;
30     import android.util.Log;
31     import android.widget.TextView;

32     /* Main activity for the adb test program */
33     public class AdbTestActivity extends Activity {

34         private static final String TAG = "AdbTestActivity";

35         private TextView mLog;
36         private UsbManager mManager;
37         private UsbDevice mDevice;
38         private UsbDeviceConnection mDeviceConnection;
39         private UsbInterface mInterface;
40         private AdbDevice mAdbDevice;

41         private static final int MESSAGE_LOG = 1;
42         private static final int MESSAGE_DEVICE_ONLINE = 2;
```

Noticed, in line 40 that the type `AdbDevice` is user defined class in `AdbDevice.java`. The class represents a USB device that supports the adb protocol.



AdbTestActivity.java

```
43      @Override
44      public void onCreate(Bundle savedInstanceState) {
45          super.onCreate(savedInstanceState);

46          setContentView(R.layout.adb);
47          mLog = (TextView)findViewById(R.id.log);

48          mManager = (UsbManager) getSystemService(Context.USB_SERVICE);

49          // check for existing devices
50          for (UsbDevice device : mManager.getDeviceList().values()) {
51              UsbInterface intf = findAdbInterface(device);
52              if (setAdbInterface(device, intf)) {
53                  break;
54              }
55          }

56          // listen for new devices
57          IntentFilter filter = new IntentFilter();
58          filter.addAction(UsbManager.ACTION_USB_DEVICE_ATTACHED);
59          filter.addAction(UsbManager.ACTION_USB_DEVICE_DETACHED);
60          registerReceiver(mUsbReceiver, filter);
61      }
```

Line 44-61 is the onCreate() callback.

Line 49-55 enumerates all connected USB devices and interfaces of each device until an interface that matched the defined interface class, subclass, and protocol is found. This operation is performed at runtime in contrast to the static one that using intent filter. Specifically, it is performed whenever the AdbTest is launched.

Line 50 iterates over all the connected USB devices.

Line 51 filters for the matching targeted interface that supporting ADB protocol. It is done by calling findAdbInterface() at line 139.

Once the targeted ADB interface is found, line 52 tries to connect and starts the device. If success, it break and having the fields:

mDevice,
mDeviceConnection,
mInterface and
mAdbDevice configured properly.



AdbTestActivity.java

```
43      @Override
44      public void onCreate(Bundle savedInstanceState) {
45          super.onCreate(savedInstanceState);

46          setContentView(R.layout.adb);
47          mLog = (TextView)findViewById(R.id.log);

48          mManager = (UsbManager) getSystemService(Context.USB_SERVICE);

49          // check for existing devices
50          for (UsbDevice device : mManager.getDeviceList().values()) {
51              UsbInterface intf = findAdbInterface(device);
52              if (setAdbInterface(device, intf)) {
53                  break;
54              }
55          }

56          // listen for new devices
57          IntentFilter filter = new IntentFilter();
58          filter.addAction(UsbManager.ACTION_USB_DEVICE_ATTACHED);
59          filter.addAction(UsbManager.ACTION_USB_DEVICE_DETACHED);
60          registerReceiver(mUsbReceiver, filter);
61      }
```

Line 56-60 instantiate the run time Intent filter, actions to be matched, and register the Broadcast Receiver mUsbReceiver and the defined filter.

The mUsbReceiver is given in line 151-170 which play an curial role in this program. This wraps up the onCreate() method of the program.

In all, when the method onCreate() is executed, it enumerates connected USB device for and usb accessory having the designated class, subclass protocol.

It also register and BroadcastReceiver that in case an USB device in attached/detached after the program is started.

So when the activity is created, it does three things

(1)instantiate a USB manager

(2) enumerate the attached devices for a target USB device

(3) setup a Broadcast Receiver with intent filter for USB device attachment and detachment.



AdbTestActivity.java

```
62      @Override
63      public void onDestroy() {
64          unregisterReceiver(mUsbReceiver);
65          setAdbInterface(null, null);
66          super.onDestroy();
67      }

68      public void log(String s) {
69          Message m = Message.obtain(mHandler, MESSAGE_LOG);
70          m.obj = s;
71          mHandler.sendMessage(m);
72      }
```

Line 63-67 is the onDestroy() method.

Line 64 unregisters the broadcastreceiver and, line 65 frees the device. Noticed, freeing the device is done by calling method setAdbInterface() and passing arguments null.

Line 66 calls the method onDestroy() of superclass. Noticed the order of removing all resources are done in reverse order of their' allocation.



AdbTestActivity.java

```
62      @Override
63      public void onDestroy() {
64          unregisterReceiver(mUsbReceiver);
65          setAdbInterface(null, null);
66          super.onDestroy();
67      }

68      public void log(String s) {
69          Message m = Message.obtain(mHandler, MESSAGE_LOG);
70          m.obj = s;
71          mHandler.sendMessage(m);
72      }
```

Using message queue will perform printing when the thread (main thread) is free and hence avoiding seizing the thread from doing other tasks.

Line 68-72 gives the method `log()` that a helper method which performs logging. The method basically wraps the log string `s` into a `MESSAGE_LOG` message and send it to the Message queue of the activity for logging.

Line 69 obtains a message `m` from the global message recycle pool. It also passes the `MESSAGE_LOG` to the what value of the Message which will be used in `handleMessage()`. The Message here is different to the message of `AdbMessage` of this package!! Please do not mix them together.

Line 70 sets the `obj` of the message to be `s` that the string passed as argument.

Line 71 sends the message through the `mHandler` to the message queue of the this activity for processing..



AdbTestActivity.java

```
73     private void appendLog(String text) {
74         Rect r = new Rect();
75         mLog.getDrawingRect(r);
76         int maxLines = r.height() / mLog.getLineHeight() - 1;
77         text = mLog.getText() + "\n" + text;

78         // see how many lines we have
79         int index = text.lastIndexOf('\n');
80         int count = 0;
81         while (index > 0 && count <= maxLines) {
82             count++;
83             index = text.lastIndexOf('\n', index - 1);
84         }

85         // truncate to maxLines
86         if (index > 0) {
87             text = text.substring(index + 1);
88         }
89         mLog.setText(text);
90     }
```

Line 73- 90 is the appendLog(). The method is called by handleMessage() of the mHandler that declared at line 196.

The method fits the printout log into the drawing window of the text view incase the log exceeds the size of text view window. This is done by trimming lines from top of the log.

Line 74-76 calculate the maximum number of lines that text view can accommodate.

Line 77 concatenates the passing text and the newline to the existing text.

Line 78-84 resolve line index from the end of the text to the top of maxLines that can be fit into the text view. It returns the index of the first line to be printed.

Line 86-87 trim off the lines that doesn't fit.

Line 89 prints the processed text to the textView of the host.



AdbTestActivity.java/deviceOnline()

```
91      public void deviceOnline(AdbDevice device) {  
92          Message m = Message.obtain(mHandler, MESSAGE_DEVICE_ONLINE);  
93          m.obj = device;  
94          mHandler.sendMessage(m);  
95      }  
  
96      private void handleDeviceOnline(AdbDevice device) {  
97          log("device online: " + device.getSerial());  
98          device.openSocket("shell:exec logcat");  
99      }
```

Line 91-95 define the method `deviceOnline()`. It is called by method `handleConnect()` from `AdbDevice`. This method will send a message with what value `"MESSAGE_DEVICE_ONLINE"` to the message queue of the main thread. TODO: I honestly believe that the method should be implemented inside of `AdbDevice` itself.

Line 92 pulls out a message from the cycled message pools and set the target to `mHanler` with what value `"MESSAGE_DEVICE_ONLINE."`

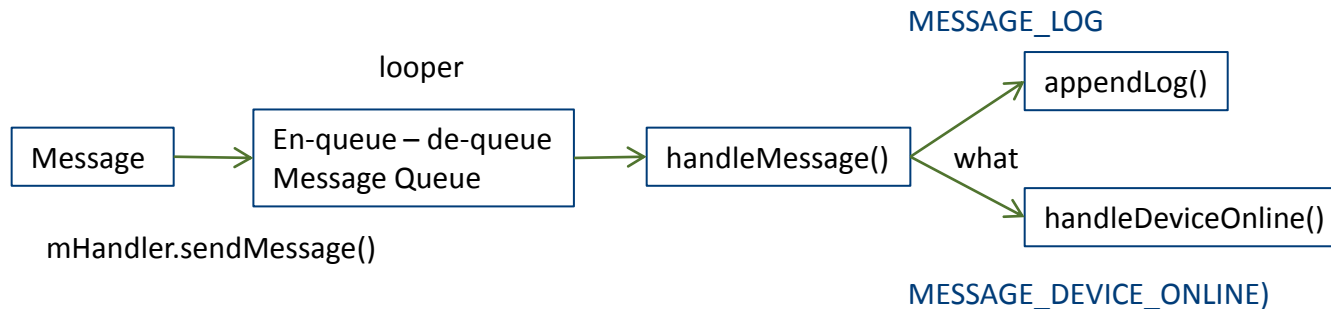
Line 93 sets the object of the message to be device. The device is eventually passed as argument to the method `handleDeviceOnline()` that given in line 96-99.



AdbTestActivity.java/ deviceOnline()

```
91      public void deviceOnline(AdbDevice device) {  
92          Message m = Message.obtain(mHandler, MESSAGE_DEVICE_ONLINE);  
93          m.obj = device;  
94          mHandler.sendMessage(m);  
95      }  
  
96      private void handleDeviceOnline(AdbDevice device) {  
97          log("device online: " + device.getSerial());  
98          device.openSocket("shell:exec logcat");  
99      }
```

Line 94 sends the message to the mHandler and hence the message queue of the main activity for processing. Later on, when the message is de-queued, it will be processed by the handleMessage() and eventually invoking handleDeviceOnline() to process the device. (Although the handler approach (en-queue and de-queueing) appears to be redundant, however it is required to make the thread working smoothly)



AdbTestActivity.java

```
91      public void deviceOnline(AdbDevice device) {
92          Message m = Message.obtain(mHandler, MESSAGE_DEVICE_ONLINE);
93          m.obj = device;
94          mHandler.sendMessage(m);
95      }

96      private void handleDeviceOnline(AdbDevice device) {
97          log("device online: " + device.getSerial());
98          device.openSocket("shell:exec logcat");
99      }
```

Line 96 is the `handleDeviceOnline()` which called by `handleMessage()`.

Line 97 calls the `log()` that again pack the message and en-queue the message again as a `MESSAGE_LOG` to the message queue of the thread 😊

Line 98 opens a socket to the peer USB device and runs “shell:exe logcat” on the device. The ADB shell command effectively runs `exec logcat` on the device. The `logcat` will transfer all succeeding log information of the device to the host through the USB connection. (See ADB manual and [figure](#).)

The `openSocket()` propagates 9 references to its end. can't figure out the logic behind the calls, too many methods referencing will make program difficult to follow and maintain. Unless it is really necessary, there should be a way to simplify it.



AdbTestActivity.java

The whole procedure when receiving a connection request from the peer device and sent out the logcat command:

1. `run()/WaiterThread/AdbDevice` (a message comes in) →
2. `dispatchMessage(message)/AdbDevice` (the message is `A_CNXX` that a connect command) →
3. `handleConnect(message)/AdbDevice` (if message data starts with “device:”) →
4. `deviceOnline(this)/AdbTestActivity` (“this” is the caller object that a `AdbDevice`, the device is packed into a `mHandler` message `m` and send to the message queue) →
5. `handleMessage(m)/AdbTestActivity` (determine the `m` is of type `MESSAGE_DEVICE_ONLINE` and) →
6. `handleDeviceOnline(device)/AdbTestActivity` (argument is the caller device that a `AdbDevice`) →
7. `openSocket(“shell:exec logcat”)/AdbDevice` →
8. `open((“shell:exec logcat”)/AdbSocket` (pack the argument to a `A_OPEN` `AdbMessage` and write it to the device which instantiate this `AdbSocket` object.) →
9. `message.write(mDevice)/AdbDevice` (creates a output request of `AdbDevice`, set the message as client data and queue it)



AdbTestActivity.java/setAdbInterface()

```
100      // Sets the current USB device and interface
101      private boolean setAdbInterface(UsbDevice device, UsbInterface intf) {
102          if (mDeviceConnection != null) {
103              if (mInterface != null) {
104                  mDeviceConnection.releaseInterface(mInterface);
105                  mInterface = null;
106              }
107              mDeviceConnection.close();
108              mDevice = null;
109              mDeviceConnection = null;
110          }
111      }
```

Line 100-137 define the method setAdbInterface(). The method is referenced at three places of the project: mUsbReceiver() that the broadcast receiver, the onCreate() and onDestroy() of the AdbTestActivity class.

The method is responsible of opening the device(a physical UsbDevice device and not the AdbDevice. Please try not to confuse them) and configure the device with proper values.

Line 102-110 reset the mInterface, mDeviceConnection, and mDevice if they are defined already. Please notice the order of nullifying of these members. First release the interface then connection and finally the device.



AdbTestActivity.java/setAdbInterface()

```
111         if (device != null && intf != null) {
112             UsbDeviceConnection connection = mManager.openDevice(device);
113             if (connection != null) {
114                 log("succeeded");
115                 if (connection.claimInterface(intf, false)) {
116                     log("claim interface succeeded");
117                     mDevice = device;
118                     mDeviceConnection = connection;
119                     mInterface = intf;
120                     mAdbDevice = new AdbDevice(this, mDeviceConnection, intf);
121                     log("call start");
122                     mAdbDevice.start();
123                     return true;
124                 } else {
125                     log("claim interface failed");
126                     connection.close();
127                 }
128             } else {
129                 log("open failed");
130             }
131         }
```

Line 111 -131 perform the initialization process which includes initialization of the device, the connection, the interface and the AdbDevice.

Line 111 checks if the argument device and intf are valid. If any of these arguments is null, the code branches to line 132.

Line 112 opens the device by calling openDevice(device) of UsbManager, and assigns the return object of UsbDeviceConnection to connection. All succeeding USB reading and writing transactions will implement this object.



AdbTestActivity.java/setAdbInterface()

```
111         if (device != null && intf != null) {
112             UsbDeviceConnection connection = mManager.openDevice(device);
113             if (connection != null) {
114                 log("open succeeded");
115                 if (connection.claimInterface(intf, false)) {
116                     log("claim interface succeeded");
117                     mDevice = device;
118                     mDeviceConnection = connection;
119                     mInterface = intf;
120                     mAdbDevice = new AdbDevice(this, mDeviceConnection, intf);
121                     log("call start");
122                     mAdbDevice.start();
123                     return true;
124                 } else {
125                     log("claim interface failed");
126                     connection.close();
127                 }
128             } else {
129                 log("open failed");
130             }
131         }
```

Line 113 checks if the connection is valid. If it does,

Line 114 calls log() to en-queue the message “open succeeded.”

Line 115 checks and tries to claim the interface (an UsbInterface that passed as argument to the method) by calling claimInterface(). If it success,

Line 115-119 are executed that initialization the fields (mDevice, mDeviceConnection, and mInterface) of the class AdbTestActivity.

Up to this point, lower lever USB APIs are used to established the necessary USB connection.

AdbTestActivity.java/setAdbInterface()

```
111         if (device != null && intf != null) {
112             UsbDeviceConnection connection = mManager.openDevice(device);
113             if (connection != null) {
114                 log("open succeeded");
115                 if (connection.claimInterface(intf, false)) {
116                     log("claim interface succeeded");
117                     mDevice = device;
118                     mDeviceConnection = connection;
119                     mInterface = intf;
120                     mAdbDevice = new AdbDevice(this, mDeviceConnection, intf);
121                     log("call start");
122                     mAdbDevice.start();
123                     return true;
124                 } else {
125                     log("claim interface failed");
126                     connection.close();
127                 }
128             } else {
129                 log("open failed");
130             }
131         }
```

•Line 120 instantiates a new AdbDevice with proper arguments that the device mDeviceConnection and the interface intf. These arguments are adapted by the AdbDevice as its own private members. The constructor AdbDevice() polls every endpoints of the interface intf, exams the type of the endpoint to see if it is a USB_ENDPOINT_XFER_BULK and determines its transfer direction.

At this point, the AdbDevice will take over all transactions to the peer USB device using sockets.



AdbTestActivity.java/setAdbInterface()

```
111         if (device != null && intf != null) {
112             UsbDeviceConnection connection = mManager.openDevice(device);
113             if (connection != null) {
114                 log("open succeeded");
115                 if (connection.claimInterface(intf, false)) {
116                     log("claim interface succeeded");
117                     mDevice = device;
118                     mDeviceConnection = connection;
119                     mInterface = intf;
120                     mAdbDevice = new AdbDevice(this, mDeviceConnection, intf);
121                     log("call start");
122                     mAdbDevice.start();
123                     return true;
124                 } else {
125                     log("claim interface failed");
126                     connection.close();
127                 }
128             } else {
129                 log("open failed");
130             }
131         }
```

Line 122 calls the start() of AdbDevice which in term starts the inner class WaiterThread of the AdbDevice (Noticed that, AdbDevice is not a thread but it's inner class WaiterThread is.)

Line 123 returns true. (Henceforth, the AdbDevice will take over all actual transition between the physical UsbDevices and the App.)

Line 124-127 close the connection if the test on line 115 that claiming the interface failed.

Line 128-130 perform logging when the connection is null.

AdbTestActivity.java/setAdbInterface()

```
132         if (mDeviceConnection == null && mAdbDevice != null) {  
133             mAdbDevice.stop();  
134             mAdbDevice = null;  
135         }  
136         return false;  
137     }
```

Line 132-136 release of a mAdbDevice when a connection to the physical USB connection is closed (user has unplugged the USB device)

Line 133 stops the mAdbDevice by calling stop().

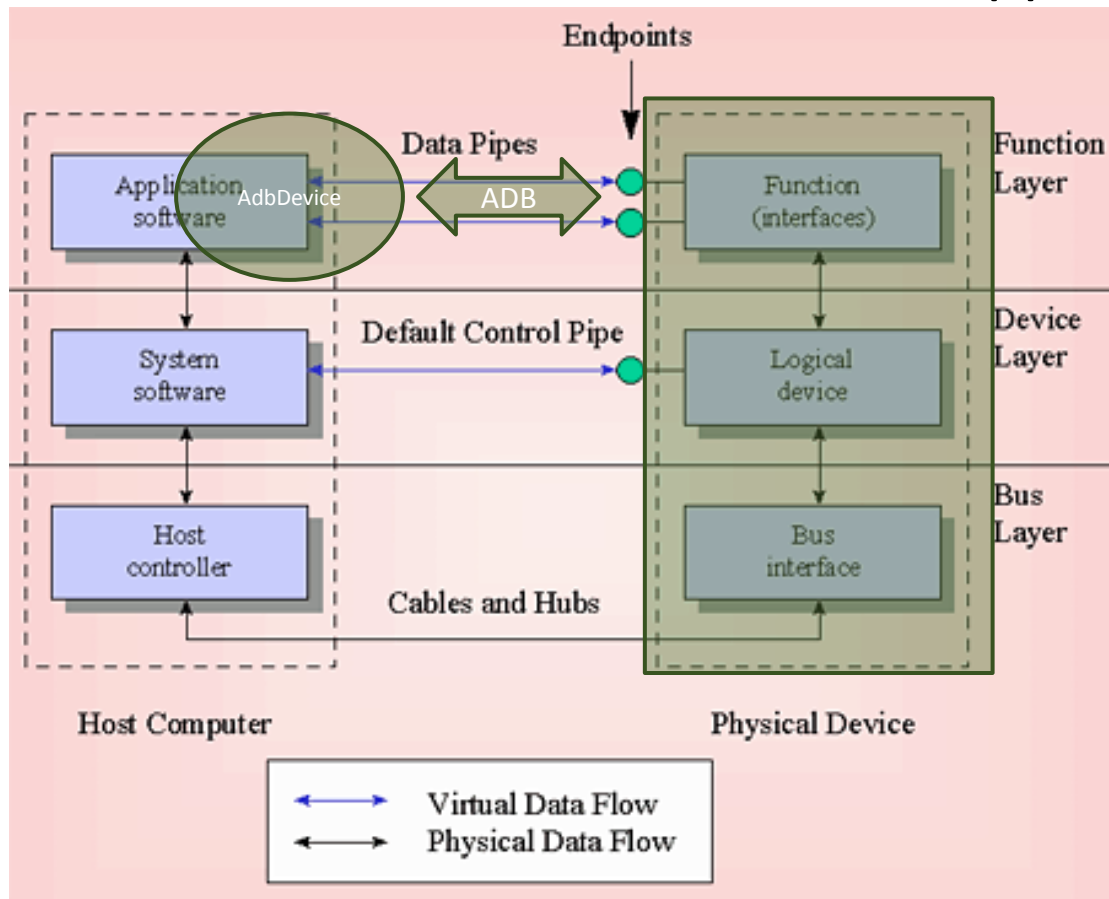
Line 134 sets the mAdbDevice reference to null.

Line 136 returns false.



AdbTestActivity.java/setAdbInterface()

- At this point, we have a AdbDevice client that handling all the transaction between the App and the device.



This is required since we are communicating with the device using ADB protocol at the application level of the host. We need a layer of abstraction to support ADB protocol between the application and USB Pipes

AdbTestActivity.java/findAdbInterface()

```
138      // searches for an adb interface on the given USB device
139      static private UsbInterface findAdbInterface(UsbDevice device) {
140          Log.d(TAG, "findAdbInterface " + device);
141          int count = device.getInterfaceCount();
142          for (int i = 0; i < count; i++) {
143              UsbInterface intf = device.getInterface(i);
144              if (intf.getInterfaceClass() == 255 && intf.getInterfaceSubclass() == 66 &&
145                  intf.getInterfaceProtocol() == 1) {
146                  return intf;
147              }
148          }
149          return null;
150      }
```

This method findAdbInterface() is referenced at several places of the project : onReceive() method of the inner class mUsbReceiver and onCreate() method of the AdbTestActivity class. In shot, it is called when a USB device attached or main activity started.

Having the device passed as argument, it iterates over all interfaces of the device and check if any interface matches designated USB device signature that stated in line 144†.

If all conditions match, the interface is return. The method ended when an matched interface is found. Else, the null is returned.

† Class 255 stands for vender specific, see http://www.usb.org/developers/defined_class/#BaseClassFFh
Interface Subclass 66 is Android ADB debugging. <http://projects.godelico.com/p/AJZaurusUSB/issues/117/>
It is however not documented in UsbConstants



AdbTestActivity.java/onReceive() of BroadcastReceiver

```
151  BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
152      public void onReceive(Context context, Intent intent) {
153          String action = intent.getAction();
154          if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
155              UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
156              UsbInterface intf = findAdbInterface(device);
157              if (intf != null) {
158                  log("Found adb interface " + intf);
159                  setAdbInterface(device, intf);
160              }
161          } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
162              UsbDevice device = intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
163              String deviceName = device.getDeviceName();
164              if (mDevice != null && mDevice.equals(deviceName)) {
165                  log("adb interface removed");
166                  setAdbInterface(null, null);
167              }
168          }
169      }
170  };
```

There are two possible scenarios: attachment and detachment of a USB device.

If attachment and if the attached device is a Google device that support adb protocol of version 1, then the device is configured by calling `setAdbInterface(device, intf)` as in line 159.

If a USB device is detached and the name of the device is the current `mDevice` (there could be more than one USB device connected at the same time), then the `mDevice` is completely removed. Line 166 `setAdbInterface(null, null)` is called.



AdbTestActivity.java/onReceive() of BroadcastReceiver

```
151 BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
152     public void onReceive(Context context, Intent intent) {
153         String action = intent.getAction();
154         if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
155             UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
156             UsbInterface intf = findAdbInterface(device);
157             if (intf != null) {
158                 log("Found adb interface " + intf);
159                 setAdbInterface(device, intf);
160             }
161         } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
162             UsbDevice device = intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
163             String deviceName = device.getDeviceName();
164             if (mDevice != null && mDevice.equals(deviceName)) {
165                 log("adb interface removed");
166                 setAdbInterface(null, null);
167             }
168         }
169     }
170 };
```

The code here declares an BroadcastReceiver that to be registered by the Activity.

Line 152 -169 declare the callback method onReceive().

Line 153-154 get the action from the received intent and see if the action equals constant ACTION_USB_DEVICE_ATTACHED.

If it does, then line 155-160 are executed, if not then line 161-168 are executed.



AdbTestActivity.java/onReceive() of BroadcastReceiver

```
151  BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
152      public void onReceive(Context context, Intent intent) {
153          String action = intent.getAction();
154          if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
155              UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
156              UsbInterface intf = findAdbInterface(device);
157              if (intf != null) {
158                  log("Found adb interface " + intf);
159                  setAdbInterface(device, intf);
160              }
161          } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
162              UsbDevice device = intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
163              String deviceName = device.getDeviceName();
164              if (mDevice != null && mDevice.equals(deviceName)) {
165                  log("adb interface removed");
166                  setAdbInterface(null, null);
167              }
168          }
169      }
170  };
```

Line 155 get the device form the extra information of the Intent.

Line 156 retrieves the interface of the device using findAdbInterface() method.

Line 157 check if the interface has been retrieve successfully. If it does, the interface is initialized by calling setAdbInterface() at line 159.



AdbTestActivity.java/onReceive() of BroadcastReceiver

```
151 BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
152     public void onReceive(Context context, Intent intent) {
153         String action = intent.getAction();
154         if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
155             UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
156             UsbInterface intf = findAdbInterface(device);
157             if (intf != null) {
158                 log("Found adb interface " + intf);
159                 setAdbInterface(device, intf);
160             }
161         } else if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
162             UsbDevice device = intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
163             String deviceName = device.getDeviceName();
164             if (mDevice != null && mDevice.equals(deviceName)) {
165                 log("adb interface removed");
166                 setAdbInterface(null, null);
167             }
168         }
169     }
170 };
```

If the action of the intent indicates a USB_DEVICE_DETACHED, then line 162-168 are executed.

Line 162 extracts the detached device using extra information from the intent.

Line 163 gets the detached device's name.

Line 164 checks if the device is the current mDevice, if it does:

Line 166 removes the mDevice by calling setAdbInterface() with null, null argument. Basically, the method setAdbInterface() will remove mDevice, mInterface, mDeviceConnection, and mAdbDevice.



AdbTestActivity.java/ handleMessage() of the thread handler.

```
171         Handler mHandler = new Handler() {
172             @Override
173             public void handleMessage(Message msg) {
174                 switch (msg.what) {
175                     case MESSAGE_LOG:
176                         appendLog((String)msg.obj);
177                         break;
178                     case MESSAGE_DEVICE_ONLINE:
179                         handleDeviceOnline((AdbDevice)msg.obj);
180                         break;
181                 }
182             }
183         };
184     }
```

This defines a handler for the message queue of the main thread. The class Handler has a nested interface Handler.Callback which defines a public method handleMessage().

The Handler has to implement the callback handleMessage() that responsible of processing de-queued/received message.

Line 173-181 implement the handleMessage() callback of the interface. The message is handled pending on the “what” value of the message. The value is either a MESSAGE_LOG or a MESSAGE_DEVICE_ONLINE.

If the message is a MESSAGE_LOG which is the most frequent cases, the method appendLog() is called with the msg.obj as argument which is a string. The receiving message (log from peer device) will be displayed on the text view of the Activity.

If the message is a MESSAGE_DEVICE_ONLINE, the method handleDeviceOnline() is called with the msg.obj as argument with is the device. Eventually, a shell:exec logcat is send to the peer device. [Also see the previous discussion.](#)



-
- ▶ This wrap-ups the AdbTestActivity .java
 - ▶ We exam the AdbDevice.java next.
 - ▶ The AdbDevice emulates a peer Adb Devices. It abstracts the USB operated and simplifies the coding effort.



AdbDevice.java

```
:
16      package com.android.adb;

17      import android.hardware.usb.UsbConstants;
18      import android.hardware.usb.UsbDeviceConnection;
19      import android.hardware.usb.UsbEndpoint;
20      import android.hardware.usb.UsbInterface;
21      import android.hardware.usb.UsbRequest;
22      import android.util.SparseArray;

23      import java.util.LinkedList;

24      /* This class represents a USB device that supports the adb protocol. */
25      public class AdbDevice {

26          private final AdbTestActivity mActivity;
27          private final UsbDeviceConnection mDeviceConnection;
28          private final UsbEndpoint mEndpointOut;
29          private final UsbEndpoint mEndpointIn;

30          private String mSerial;
```

The AdbDevice is instantiated by calling setAdbInterface() method of AdbTestActivity. The method is called after a physical USB device is detected (either attached or detached).

In the setAdbInterface(), the peer USB device is connected and configured. Only then, the constructor AdbDevice() is called where the connection and the interface to the peer USB device are passed as argument to the AdbDevice().

In general, the package names the peer USB device as mDevice while the AdbDevice as mAdbDevice. However, noticed that, in the class AdbSocket where the AdbDevice is name mDevice and this may cause confusion that a bad form.

This class represents a USB device that supports the adb protocol. It is instantiated and runs on the host side.

It 's primary goal is to emulating an Adb device at the host side to communicate with the peer Adb client and receiving the logcat packet from the client.

Line 16-29 declare some basic USB device features: connection and endpoints.



AdbDevice.java

```
31         // pool of requests for the OUT endpoint
32         private final LinkedList<UsbRequest> mOutRequestPool = new LinkedList<UsbRequest>();
33         // pool of requests for the IN endpoint
34         private final LinkedList<UsbRequest> mInRequestPool = new LinkedList<UsbRequest>();
35         // list of currently opened sockets
36         private final SparseArray<AdbSocket> mSockets = new SparseArray<AdbSocket>();
37         private int mNextSocketId = 1;

38         private final WaiterThread mWaiterThread = new WaiterThread();
```

Line 32-34 declare two `LinkedList<UsbRequest>`. They are `mOutRequestPool` and `mInRequestPool`. These pools are used to store in/out `UsbRequests` that object of class `UsbRequest`. According to the API, the `UsbRequest` is “A class representing USB request packet. It is used for both reading and writing data to or from a `UsbDeviceConnection`. “

Line 36 declares a `SparseArray<adbSocket>` that the `mSockets`. According the API that a sparse array “maps integers to Objects. Unlike a normal array of Objects, there can be gaps in the indices. “

Line 37 gives the index for accessing `mSockets`.

Requests are taken out/sending in by order while sockets are accessed by indices randomly. This is the main reason of using two different types of data structures. My personal opinion 😊

Line 38 instantiates a `WiterThread` that a `mWaiterThread`.



AdbDevice.java/ AdbDevice() that the constructor

```
39      public AdbDevice(AdbTestActivity activity, UsbDeviceConnection connection,
40                      UsbInterface intf) {
41          mActivity = activity;
42          mDeviceConnection = connection;
43          mSerial = connection.getSerial();

44          UsbEndpoint epOut = null;
45          UsbEndpoint epIn = null;
46          // look for our bulk endpoints
47          for (int i = 0; i < intf.getEndpointCount(); i++) {
48              UsbEndpoint ep = intf.getEndpoint(i);
49              if (ep.getType() == UsbConstants.USB_ENDPOINT_XFER_BULK) {
50                  if (ep.getDirection() == UsbConstants.USB_DIR_OUT) {
51                      epOut = ep;
52                  } else {
53                      epIn = ep;
54                  }
55              }
56          }
57          if (epOut == null || epIn == null) {
58              throw new IllegalArgumentException("not all endpoints found");
59          }
60          mEndpointOut = epOut;
61          mEndpointIn = epIn;
62      }
```

Here is the constructor for the AdbDevice.

Line 39 defines the method's signatures. It is called by setAdbInterface() from the AdbTestActivity. Noticed that, the argument connection and the interface are of the physical USB device and were created and claimed at setAdbInterface() of the AdbTestActivity.

Line 41-45 assign the passed arguments to the local fields.

Line 47-56 probe the bulk endpoints of the interface and determine whether they are in or out endpoint. There should be exactly one for each.



AdbDevice.java/ AdbDevice() that the constructor

```
39      public AdbDevice(AdbTestActivity activity, UsbDeviceConnection connection,
40                      UsbInterface intf) {
41          mActivity = activity;
42          mDeviceConnection = connection;
43          mSerial = connection.getSerial();

44          UsbEndpoint epOut = null;
45          UsbEndpoint epIn = null;
46          // look for our bulk endpoints
47          for (int i = 0; i < intf.getEndpointCount(); i++) {
48              UsbEndpoint ep = intf.getEndpoint(i);
49              if (ep.getType() == UsbConstants.USB_ENDPOINT_XFER_BULK) {
50                  if (ep.getDirection() == UsbConstants.USB_DIR_OUT) {
51                      epOut = ep;
52                  } else {
53                      epIn = ep;
54                  }
55              }
56          }
57          if (epOut == null || epIn == null) {
58              throw new IllegalArgumentException("not all endpoints found");
59          }
60          mEndpointOut = epOut;
61          mEndpointIn = epIn;
62      }
```

Line 57-58 throws exception if either in or out endpoint, or both are missing.

Line 60-62 assign the endpoints to corresponding member fields.

There are 4 types of endpoints defined in the USB standard and in the Android as
USB_ENDPOINT_XFER_CONTROL (endpoint zero)
USB_ENDPOINT_XFER_ISOC (isochronous endpoint)
USB_ENDPOINT_XFER_BULK (bulk endpoint)
USB_ENDPOINT_XFER_INT (interrupt endpoint)
<http://developer.android.com/reference/android/hardware/usb/UsbEndpoint.htm>

AdbDevice.java/getSerial()

```
63         // return device serial number
64         public String getSerial() {
65             return mSerial;
66         }
```

Line 64-66 declare the `getSerial()` method which eventually calls `getSerial()` of `UsbDeviceConnection`. Every USB device has a unique serial number the `iSerialNumber`. (However don't quit understand the logic of this method since there is already a public method `getSerial()` from `UsbDeviceConnection` ready. This seems kind of redundant.)



AdbDevice.java/getOutRequest()

```
67      // get an OUT request from our pool
68      public UsbRequest getOutRequest() {
69          synchronized(mOutRequestPool) {
70              if (mOutRequestPool.isEmpty()) {
71                  UsbRequest request = new UsbRequest();
72                  request.initialize(mDeviceConnection, mEndpointOut);
73                  return request;
74              } else {
75                  return mOutRequestPool.removeFirst();
76              }
77          }
78      }
```

This looks pretty straight forward.

The method `getOutRequest()` is called by `write()` from `AdbMessage`. This seems reasonable since when writing `Message` to the physical USB device, the `AdbMessage` first gets a `UsbRequest` form the `outRequestPool`, pack the `AdbMessage` as `clientData`, and then send the `Request` using `UsbRequest.queue()`. A request is an object representing USB request packet. It is for both reading and writing data to or from a specific endpoint (other than the endpoint-0) of a `UsbDeviceConnection`.

Line 69 locks the `mOutRequestPool` from race condition.

Line 70 checks if the pool is empty. If it is, line 71 instances a new request and line 72 initializes the newly created request with the designated connection and endpoint. This is important, as we can see later, the `write()` of `AdbMessage` queue relies this embedded information of the request (connection and endpoint) to send the `Message` to designate USB device. Recycling existing request from the pool can save processing time. 😊

Line 73 returns the request to the caller `AdbMessage`.



AdbDevice.java/releaseOutRequest()

```
79         // return an OUT request to the pool
80         public void releaseOutRequest(UsbRequest request) {
81             synchronized (mOutRequestPool) {
82                 mOutRequestPool.add(request);
83             }
84         }
```

Line 79-84 declare the method `releaseOutRequest()` that recycle an output request to the pool. This method is called by `run()` of `WaiterThread` and `write()` of `AdbMessage`.



AdbDevice.java/getInRequest()

```
85      // get an IN request from the pool
86      public UsbRequest getInRequest() {
87          synchronized(mInRequestPool) {
88              if (mInRequestPool.isEmpty()) {
89                  UsbRequest request = new UsbRequest();
90                  request.initialize(mDeviceConnection, mEndpointIn);
91                  return request;
92              } else {
93                  return mInRequestPool.removeFirst();
94              }
95          }
96      }
```

Line 85-95 declare the method `getInRequest()`. The method get a request from the `mInRequestPool`. If the Pool is empty, it instantiates a new request and return the new request. It is called at several places inside of the `run()` method from the `WaiterThread`. The thread using this method to get in requests.

Line 87 locks the `mInRequestPool`.

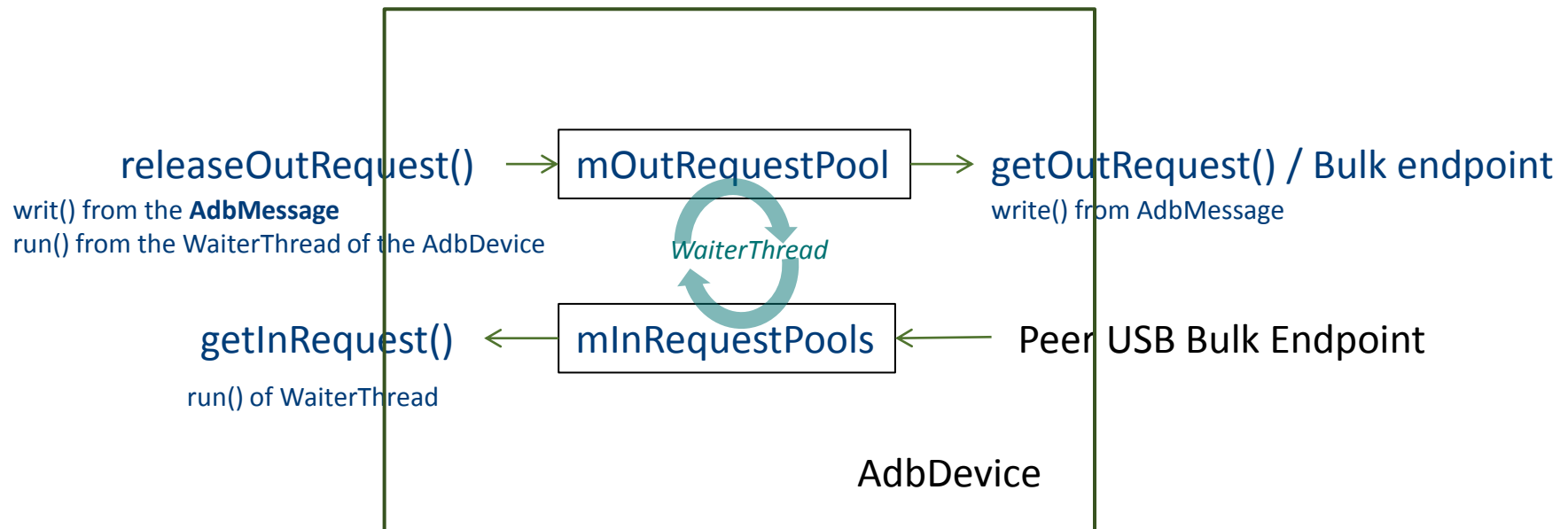
Line 88 checks if the pool is empty. If it is, it instances and initializes a new request of the communication endpoint and return the request(*This is kind of strange thought, for an In request that correspond to a in endpoint receive request form the client to the host. If no request from the pool, then why brothering generate a new request and return it to the caller?)*

Line 93 is executed if the pool is not empty. The program simply removes the first request from the pool and returns it.



AdbDevice.java

- ▶ About **mOutRequestPool** and **mInRequestPool**.
 - ▶ **mOutRequestPool** – **getOutRequest()** and **releaseOutRequest()**
 - ▶ **mInRequestPools** – **getInRequest()**



AdbDevice.java

- ▶ About `mOutRequestPool` and `mInRequestPool`.
 - ▶ `mOutRequestPool` – `getOutRequest()` and `releaseOutRequest()`
 - ▶ `mInRequestPools` – `getInRequest()`

There is one other thing that is who feed these `mOutRequestPool` and `mInRequestPool`.

The `mOutRequestPool` is reference by `getOutRequest()` and `releaseOutRequest()` of the `AdbDevice`.
`getOutRequest()` get the request out of the pool, while `releaseOutRequest()` send request to the pool.
`releaseOutRequest` is called by `writ()` from the `AdbMessage` and `run()` from the `WaiterThread` of the `AdbDevice`.

The `mInRequestPools` is reference by `getInRequest()` of `AdbDevice`. `getInRequest()` retrieve the request from the pools while the peer device fills the pools. `getInRequest()` is invoked at several places inside of `run()` of `WaiterThread`.

My understanding at this point, is that the `WaiterThread` retrieves request form `mInRequestPool` using `getInRequest()` constantly and in a stand along manner. The retrieved request is parse and processing by `dispatchMessage()` which eventually calls the `handleMessage()` of the `AdbSocket` to send the message to the `AdbDevice` for displaying.



AdbDevice.java/start()

```
97      public void start() {  
98          mWaiterThread.start();  
99          connect();  
100     }
```

The method `start()` is called by `setAdbInterface()` of `AdbTestActivity`. It is called after finishing configuring the USB device, connection, interface and `AdbDevice`.

The method does two things:

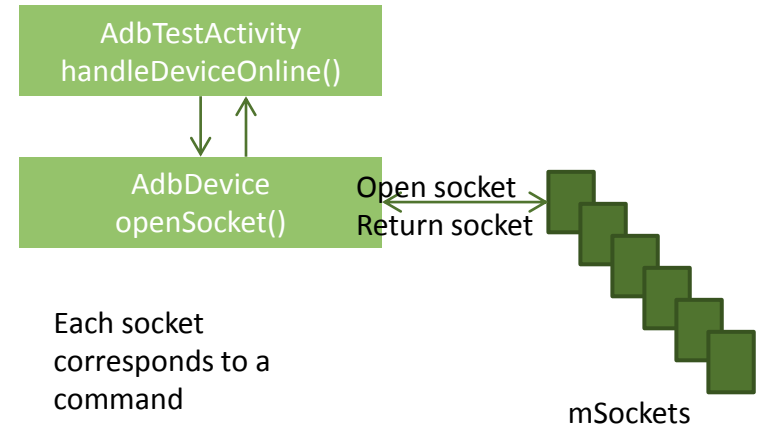
It starts the `WaiterThread` which handles in/out USB requests.

It then calls the method `connect()` of the `AdbDevice`. The method packs a `CONNECT` message and send it to the USB device which eventually connecting the USB device with the `AdbDevice`.



AdbDevice.java/openSocket()

```
101      public AdbSocket openSocket(String destination) {
102          AdbSocket socket;
103          synchronized (mSockets) {
104              int id = mNextSocketId++;
105              socket = new AdbSocket(this, id);
106              mSockets.put(id, socket);
107          }
108          if (socket.open(destination)) {
109              return socket;
110          } else {
111              return null;
112          }
113      }
```



Line 101-113 declare the method `openSocket()` which returns an object of `AdbSocket`. The method is called by `handleDeviceOnline()` of `AdbTestActivity` with destination "shell:exec logcat". (It should be possible to merge the method into the `handleDeviceOnline()` since it is the only method that invokes the `openSocket()` for now. But, I guess it is intentionally leaving this way for future extension where more Sockets are implemented.)

Line 103 locks the `mSockets`. Noticed that, `mSockets` is not a socket but a `SparseArray` of `AdbSocket`. It is instantiated at line 43 of the `AdbDevice`. A `SparseArray` is a class given in `android.util.SparseArray<E>`. A `SparseArray` maps sparse index (non contiguous integer) into object.

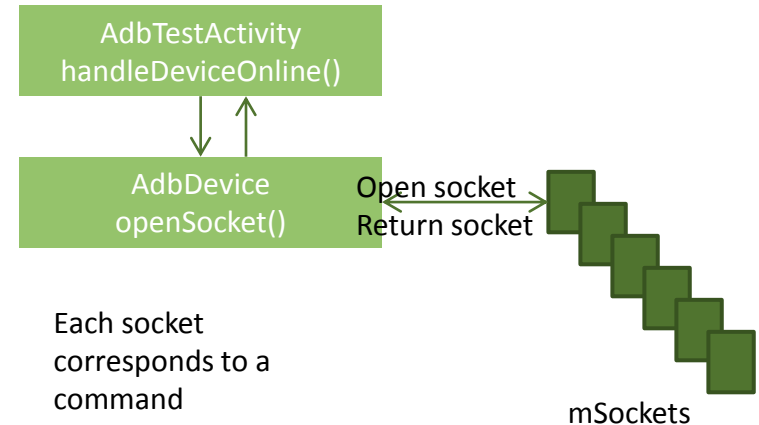
Line 104 increments `mNextSocketId` by one and assigns the value to `id`.

Line 105 instantiates a new `AdbSocket` which is indexed by the `id` from above.

Line 106 adds the socket of the `id` to the `mSockets` that is a `SparseArray` of `AdbSocket`.

AdbDevice.java/openSocket()

```
101     public AdbSocket openSocket(String destination) {
102         AdbSocket socket;
103         synchronized (mSockets) {
104             int id = mNextSocketId++;
105             socket = new AdbSocket(this, id);
106             mSockets.put(id, socket);
107         }
108         if (socket.open(destination)) {
109             return socket;
110         } else {
111             return null;
112         }
113     }
```



Line 108 tries to open the peer device using `open()` from `AdbSocket`.

What `open()` actually does is writing the "destination" to the peer device. The "destination" contains a Adb shell command and the exec logcat that instructs the peer device log all its operation and sent it back through USB.

If the `open()` operation success, then the socket is return (in the current implementation this returned socket is never used) Else, null is returned.

Incidentally, this is the only socket instantiated in the current implementation. As the [previous slide that the document mentioned](#) that a socket is created for each adb command that is executed, and the destination is the only command that implemented for now.

AdbDevice.java/getSocket()

```
114         private AdbSocket getSocket(int id) {  
115             synchronized (mSockets) {  
116                 return mSockets.get(id);  
117             }  
118         }
```

Line 114-118 declare the method `getSocket`.

The method is called by `dispatchMessage()` from the `AdbDevice` when an `AdbMessage` is obtained. The destination socket id is retrieved from the receiving `AdbMessage` and passed as argument to the `getSocket()`. In turn, the corresponding socket is retrieved from the `SparseArray` of the `AdbSocket` using that index. Finally, the `handleMessage()` method of the retrieved socket is called to handle the message. Each socket has its own `handleMessage()` method. Each `handleMessage()` of different socket is implemented differently that pending on the purpose of the socket. It might be difficult to see this from the current project for there is only one socket implemented up to now.

Line 115 locks the `mSocket`, retrieve and return the socket that under the designated id from the `mSocket`.

Adb Packet Format
local-id, remote-id: stream identified
CONNECT(version, maxdata, "system-identity-string")
OPEN(local-id, 0, "**destination**")
READY(local-id, remote-id, "")
WRITE(0, remote-id, "data")
CLOSE(local-id, remote-id, "")
SYNC(online, sequence, "")



AdbDevice.java/socketClosed()

```
119         public void socketClosed(AdbSocket socket) {  
120             synchronized (mSockets) {  
121                 mSockets.remove(socket.getId());  
122             }  
123         }
```

Line 119-123 declare the socketClosed() method.

Interestingly enough, the method is defined but never used in the project.

Line 120 locks the mSockets.

Line 121 calls remove() of the AdbSocket to remove the socket that under specific id.

From first appearance, these socket operation methods should be defined inside of the AdbSocket instead of the here.



AdbDevice.java/connect()

A_SYNC → synchronous

A_CNXXN → connect

A_OPEN → open

A_OKAY → ready

A_CLSE → close

A_WRITE → write

```
124      // send a connect command
125      private void connect() {
126          AdbMessage message = new AdbMessage();
127          message.set(AdbMessage.A_CNXXN, AdbMessage.A_VERSION, AdbMessage.MAX_PAYLOAD, "host::\0");
128          message.write(this);
129      }
```

Line 124 – 129 declare the connect() method.

The method is called by method start() of the AdbDevice. It is called after the WaiterThread is started.

The method connects the peer USB device by sending out a special CONNECT AdbMessage.

Line 126 instantiates an AdbMessage, and

Line 127 sets up such message. The message carries a CONNECT command, the version 0x01000000 of adb protocol, the maximum payload 4096, and the data that designated host and endpoint 0. Specifically, the system identify string “host::\0” indicates the message is sent form host, without any ID number, and banner \0. The phrase banner is a human-readable version or identifier string for informational only (In short, it has no effect on the program. See ADB protocol[†] for the meaning of these arguments)

Line 128 calls write() method of AdbMessage with “this” argument that representing the current AdbDevice. The write() eventually send the AdbMessage to the peer USB device.

[†]The Adb protocol is at ./system/core/adb/protocol.txt of Android Source Code.



AdbDevice.java/handleConnect()

```
130         // handle connect response
131         private void handleConnect(AdbMessage message) {
132             if (message.getDataString().startsWith("device:")) {
133                 log("connected");
134                 mActivity.deviceOnline(this);
135             }
136         }
```

Line 130-136 declare the handleConnect() method.

The method is called by the dispatchMessage() of the AdbDevice. It is invoked when the dispatched AdbMessage is an A_CNXX one. The A_CNXX stands for a CONNECT command (See adb protocol.) So what it does is, when a received AdbMessage carries a CONNECT command, the dispatchMessage() will call this method and pass the AdbMessage as argument. Each CONNECT AdbMessage contains a system identity string : <systemtype>:<serialno>:<banner>. The string of systemtype could be either a “bootloader”, “device”, or “host.” These systemtype indicates the type of the sender.

Line 132 checks if the AdbMessage contains a systemtype of “device:.”

line 133 calls the method log() with the string “connected” as argument. The method log() from AdbTestActivity instantiates a message of mHandler that pack the string “connected” as message object, and send the message to the message queue pending for processing. The message is eventually put into the Log and shown on the host’s text view.



AdbDevice.java

```
130         // handle connect response
131         private void handleConnect(AdbMessage message) {
132             if (message.getDataString().startsWith("device:")) {
133                 log("connected");
134                 mActivity.deviceOnline(this);
135             }
136         }
```

Line 134 calls `deviceOnline()` which instantiates a message with “what” equals “MESSAGE_DEVICE_ONLINE”, the “this” argument of the method stands for the current `adbDevice`. A message which packed with the device as its obj is then sent to the message queue pending for processing. The message will eventually invoke `handleDeviceOnline()` that log the device on the textview of the host and open a socket with command “shell:exec logcat”. The shell:exec locat will start a shell executing logcat on the peer device.

In all, the method handles an input `AdbMessage` that contains data of “device:” It then prompted that the local `AdbDevice` has been connected to an remote USB device. It then calls `deviceOnline()`.

Tow methods `connect()` and `handleConnect()` have been declared. It seems to me that they are one send by the host and the other send by the device. But, it is my understanding that the host should initiate all the connecting operating, so why is the `handleconnect()`?



AdbDevice.java

```
130         // handle connect response
131         private void handleConnect(AdbMessage message) {
132             if (message.getDataString().startsWith("device:")) {
133                 log("connected");
134                 mActivity.deviceOnline(this);
135             }
136         }
```

Interestingly enough, to get a USB device connect to a AdbDevice, it takes 9 steps to accomplish.....
This is simple **outrageous** !!

1. private class WaiterThread extends Thread / AdbDevice → receiving an connect message from device
 2. void dispatchMessage(AdbMessage message) / AdbDevice → filter out what kind of message and perform necessary calls
 3. private void handleConnect(AdbMessage message) / AdbDevice → check if it is form a device (well, what else could it be....)
 4. public void deviceOnline(AdbDevice device) / AdbDevice → set up a Connection Message m send it to Main thread's message queue.
 5. public void handleMessage(Message msg) / AdbSocket → direct the message to
 6. private void handleDeviceOnline(AdbDevice device) / AdbTestActivity →
 7. public AdbSocket openSocket(String destination) / AdbDevice → create a socket in the mSockets of the AdbDevice and try opening it
 8. public boolean open(String destination)/ AdbSocket → with destination : "shell:exec logcat" pack into an A_OPEN AdbMessage and
 9. public boolean write(AdbDevice device) / AdbDevice → constructing an OutRequest and sending the above A_OPEN Adbmessage and data to the peer USB device (Well my interpretation of the "shell:exec logcat" is the command a the local adb that create a shell connection to a device and execute logcat remotely <http://developer.android.com/tools/help/logcat.html>)
-



AdbDevice.java/stop()

```
137         public void stop() {  
138             synchronized (mWaiterThread) {  
139                 mWaiterThread.mStop = true;  
140             }  
141         }
```

Line 137-141 declare the method stop().

The method is called by setAdbInterface() of AdbTestActivity. It is called when both the connection is closed and the AdbDevice is still defined. Calling this method will terminate the WaiterThread.

Line 138 hold the lock of mWaiterThread. A WaiterThread is a subclass of Thread.

Line 139 set the field mStop of the WaiterThread to be true which will cause WaiterThread to stop. mStop is a public boolean variable defined inside of WaiterThread class.



AdbDevice.java/dispatchMessage()

```
142      // dispatch a message from the device
143      void dispatchMessage(AdbMessage message) {
144          int command = message.getCommand();
145          switch (command) {
146              case AdbMessage.A_SYNC:
147                  log("got A_SYNC");
148                  break;
149              case AdbMessage.A_CNXXN:
150                  handleConnect(message);
151                  break;
152              case AdbMessage.A_OPEN:
153              case AdbMessage.A_OKAY:
154              case AdbMessage.A_CLSE:
155              case AdbMessage.A_W RTE:
156                  AdbSocket socket = getSocket(message.getArg1());
157                  if (socket == null) {
158                      log("ERROR socket not found");
159                  } else {
160                      socket.handleMessage(message);
161                  }
162                  break;
163          }
164      }
```

A_SYNC → synchronous

A_CNXXN → connect

A_OPEN → open

A_OKAY → ready

A_CLSE → close

A_WRITE → write

Line 142-164 declare method dispatchMessage(). The method is called by run() method of class WaiterThread which handles both inbound or outbound requests.

Line 144 retrieves type of command from the AdbMessage that passing from the WaiterThread.

Line 145-163 perform a switch-case control flow that based on the command retrieved form line144.

Six cases are constructed that pending on the type of command that the AdbMessage carries, it could be either a: A_SYNC, A_CNXXN, A_OPEN, A_OKAY, A_CLSE, and A_W RTE.

Line 146 deals with synchronization command. Basically it does nothing except log the event and break as shown in line 147-148.

Line 149 deals with connect command. When a connect AdbMessage is received, it calls handleConnect() method and the rest of the jobs have been covered [in previous slides](#).



AdbDevice.java/dispatchMessage()

```
142      // dispatch a message from the device
143      void dispatchMessage(AdbMessage message) {
144          int command = message.getCommand();
145          switch (command) {
146              case AdbMessage.A_SYNC:
147                  log("got A_SYNC");
148                  break;
149              case AdbMessage.A_CNXXN:
150                  handleConnect(message);
151                  break;
152              case AdbMessage.A_OPEN:
153              case AdbMessage.A_OKAY:
154              case AdbMessage.A_CLSE:
155              case AdbMessage.A_W RTE:
156                  AdbSocket socket = getSocket(message.getArg1());
157                  if (socket == null) {
158                      log("ERROR socket not found");
159                  } else {
160                      socket.handleMessage(message);
161                  }
162                  break;
163          }
164      }
```

Line 152-155 deal with open, ready, close, write command all together in the code block from line 156-162.

Line 156, when these AdbMessage is received, the method retrieves a socket form mSocket using the id form arg1[†] of the AdbMessage.

Line 157 makes sure the socket is defined. If it is defined, then handleMessage() from line 160 is called with the AdbMessage as argument. The handleMessage() will process the AdbMessage according to its command type.

Line 158 deals with the case where the socket is undefined.

A_SYNC → synchronous

A_CNXXN → connect

A_OPEN → open

A_OKAY → ready

A_CLSE → close

A_WRITE → write

[†]As READY(local-id, remote-id, "")

The commend is packed into (Command, mId, mPeerId, data)

The local-id/mId is the sender's id while remote-id/mPeerId is the receiver's id.

On the receiving end, the remote-id stands for the id of the host socket ☺

AdbDevice.java/ log()

```
165         void log(String s) {  
166             mActivity.log(s);  
167         }
```



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
```

Line 168-220 declare inner class WaiterThread. The class is instantiated as mWaiterThread at line 38 of the class AdbDevice. The mWaiterThread is invoked by the method start() and stop() of AdbDevice.

Line 170-219 declare the run() method of the WaiterThread. After a new AdbDevice is instantiated at the setAdbInterface() of the AdbTestActivity, the WaiterThread of the AdbDevice is started. Accordingly, the run() method here is executed.

AdbTestActivity.BroadcastReceiver.onReceive(), AdbTestActivity.onCreate(), AdbTestActivity.onDestroy()

→AdbTestActivity.setAdbInterface.mAdbDevice.start()

→AdbDevice.start()

→AdbDevice.WaiterThread.run()



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
```

Line 169 declares a public field of boolean variable called mStop. The public field provides a way of stopping this thread for outside of the thread. The variable is set in the method of stop() of the AdbDevice.



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
```

†By saving the AdbMessage as a client data of the request and then updating the content of the AdbMessage may seem illogically. However, do noticed that it is the reference to the AdbMessage been saved and not the value of the AdbMessage. This means any further modification to the value of the AdbMessage is equally valid to the previously saved reference ☺

Line 172 instantiates currentCommand that a AdbMessage.

Line 173 declares currentData that a AdbMessage referencing to null.

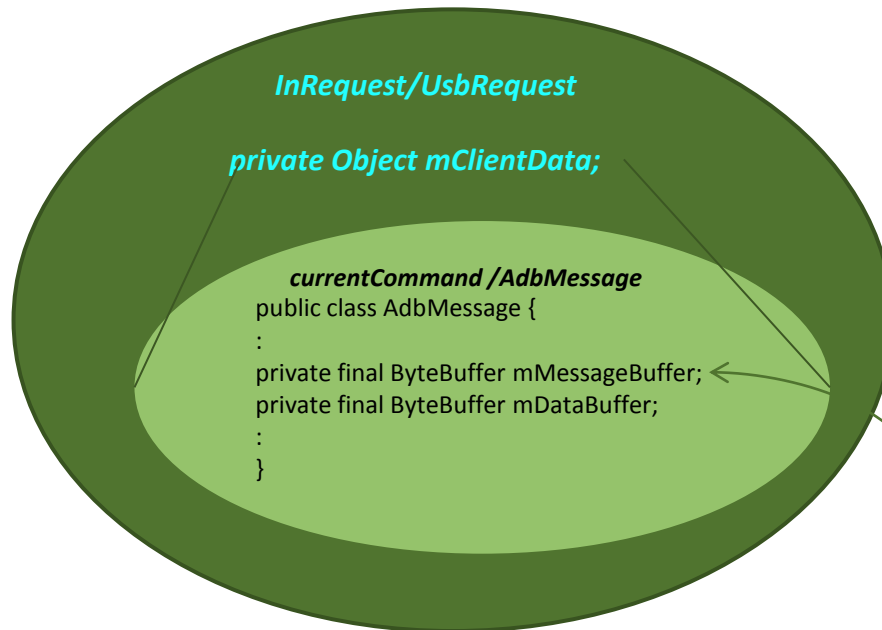
Line 175 gets a request from the input Request Queue. If the queue is empty, then a new Request is generated. (At this point, the peer Device is not connected yet, so the queue should be empty, see the method start() for detail.) ***Then, the readCommand() of the AdbMessage is called to set the client data of the request with the currentCommand (a new AdbMessage literally) and queue() the first 24 Bytes from the endpoint of the peer device into the mMessageBuffer of the currentCommand that an AdbMessage. †***

The queue() operates asynchronously, so the program continued. If the queue() operation runs successfully, then a true is returned (it is however never used in the code here)

The relation between the InRequist and currentCommand is depicted in following slide.



AdbDevice.java/WaiterThread()



The UsbRequest is an container object which contains the transfered data between the host and the peer USB devices. It wraps(encapsulates) the AdbMessage as "Client Data."

In a way, the UsbRequest is a basket that carries the message for easy of handling.



In endpoint



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
```

Line 176 -218 gives an infinite loop.

Line 177 -181 hold the lock (monitor) of the object AdbDevice.

Line 178 exams value of the mStop. The run() method is ended when mStop is true. This happens only when AdbDevice.stop() is called where mStop is set. This provides a mean to strop the WaiterThread externally.



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

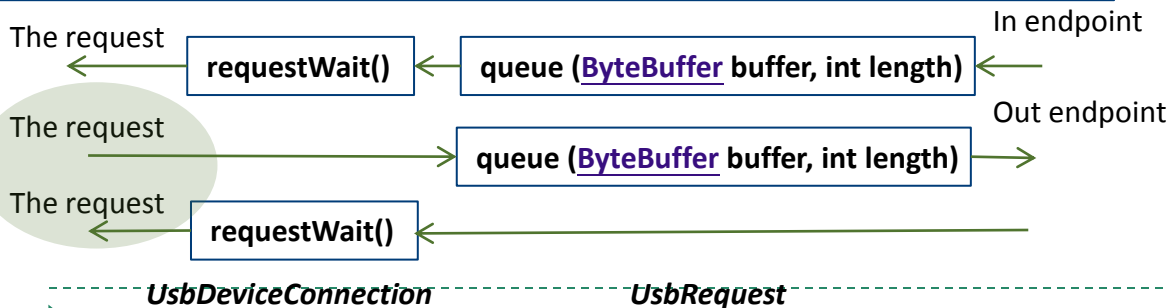
170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
        }
    }
}
```

Line 182 calls the requestWait() of UsbDeviceConnection. The method will wait (blocking the program) for the return of previously queued input/output requests.

Somehow, the method works like a gate keeper. It watches for result from previous queue() operation would it be an in/out request or any request.

The requestWait() returns whenever a queue() operation is returned, no matter what queue() operation it is. So, it is important to identify which queue() operation has return before reaction on the issue.



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
        }
    }
}
```

At the first glance, only IN request that from a IN endpoint is involved. However, be advised that the enqueue() operation is also called by the write() that performing an OUT queuing in several places (connect() of the AdbDevice, open() and sendReady() of the AdbSocket) of the program.

Take the sendReady() for example. The method send an A_OKY command to the peer device. This is done by calling write().

Since the returned request from the requestWait() could result from any of previous IN/OUT request to the same connection?

For an out request, this can be done by checking if the returned request is the same as the one sent out previously.

For an in request, however not much can be done but ...

How do you know the request is the one that you want since there could be several endpoints associate with the same connection? You can use getClientData() and setClientData() of the UsbRequest to verify this.



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176             while (true) {
177                 synchronized (this) {
178                     if (mStop) {
179                         return;
180                     }
181                 }
182                 UsbRequest request = mDeviceConnection.requestWait();
183                 if (request == null) {
184                     break;
185                 }
            }
```

If requestWait returns null, then something went wrong. Line 183 checks on this and breaks the infinite while loop as shown in line 184.



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176     while (true) {
177         synchronized (this) {
178             if (mStop) {
179                 return;
180             }
181         }
182         UsbRequest request = mDeviceConnection.requestWait();
183         if (request == null) {
184             break;
185         }
```

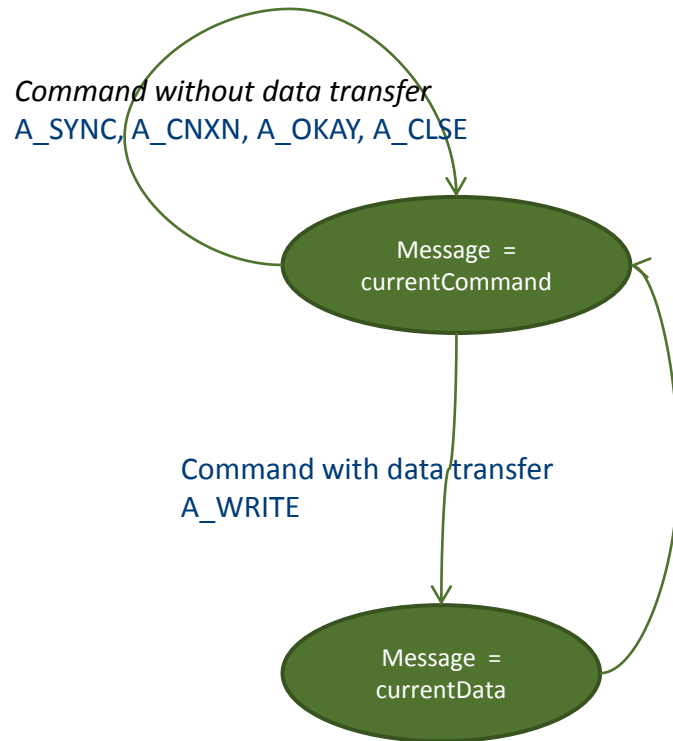
Now, the succeeding operation is kind of complicate. The control flows change dynamically that pending on the type of transaction.

It would be helpful if we have some sort of trace tool to provide the trace of the program...

Due to the design of USB bulk and interrupt transfer which divide the transfer into three states: : the request, the data and ack. The operation inside of the while() loop operates accordingly. It divide the AdbMessage into the currentCommand and currentData which correspond to the request/ack and data phase of the bulk transfer respectively.



AdbDevice.java/WaiterThread()



In a way, the WaitThread and the while() loop here implements the state machine of the USB bulk transfer...

To meet USB bulk transfer protocol that a transaction is divided into three stages: request, data, ack, the state machine is designed accordingly.

There are more than one runtime scenario to the code here and it will be exhaustive to explore all of these operation. We will only pick the typical one in our succeeding discussion. Specifically, we focus on the canonical request-data-ack operation.



AdbDevice.java/WaiterThread()

```
186         AdbMessage message = (AdbMessage)request.getClientData();
187         request.setClientData(null);
188         AdbMessage messageToDispatch = null;

189         if (message == currentCommand) {
190             int dataLength = message.getDataLength();
191             // read data if length > 0
192             if (dataLength > 0) {
193                 message.readData(getInRequest(), dataLength);
194                 currentData = message;
195             } else {
196                 messageToDispatch = message;
197             }
198             currentCommand = null;
199         } else if (message == currentData) {
200             messageToDispatch = message;
201             currentData = null;
202         }
```

First of all, the program checks if the current AdbMessage is a currentCommand (could be either a command or ack phases) or a currentData (could be an data phase).

If it is a currentData, then line 199-220 is executed. The code simply assign the currentData to the messageToDispatch and reset the currentData to null.

If it is a currentCommand, then line 189-197 are executed. In addition to that, the code will check whether there is data appended to the currentCommand.

If it does, such as a data transfer request command, then line 192-194 are executed. The code issues another queue() for data transfer and set the currentData to be message.

If however, the currentCommand does not carries any data such as a ack command, then we assign the messageToDispatch directly to be the message.



Request → Data → Ack

AdbDevice.java

```
186      AdbMessage message = (AdbMessage)request.getClientData();
187      request.setClientData(null);
188      AdbMessage messageToDispatch = null;

189      if (message == currentCommand) {
190          int dataLength = message.getDataLength();
191          // read data if length > 0
192          if (dataLength > 0) {
193              message.readData(getInRequest(), dataLength);
194              currentData = message;
195          } else {
196              messageToDispatch = message;
197          }
198          currentCommand = null;
199      } else if (message == currentData) {
200          messageToDispatch = message;
201          currentData = null;
202      }
```

```
setClientData(Object obj) {
    mClientData = obj;
}
```

Where the client data is define as
private object mClientData;
to a object UsbRequest

Setting the obj to null is effectively free mClientData,

Line 186 get the client data from the request which obtained just now using requestWait() and cast it to a AdbMessage.

Line 187 immediately sets the client data of the request to be null. The client data is declared internal to the UsbRequest as a private field mClientData of generic type Object.†

Line 188 declares yet another AdbMessage that messageToDispatch and giving it null referencing.

Line 189 checks if the obtained message equals to the currentCommand. (This currentCommand could be the one which we queue() just shortly before the while() loop started (the currentCommand is obtained at line 172-175 from **input** request pool) ☺ If they match then they are of the same request and line 190-195 are executed.)

Actually, I am having doubt about this part of design that the initial state of the state machine. What if an receiving request does not match either the currentCommand or currentData? Obviously, the request will be discarded. So it is important to enforce the USB bulk transfer protocol... Or the program will failed. On the other hand, the program should be more robust.



Request → Data → Ack

AdbDevice.java

```
186      AdbMessage message = (AdbMessage)request.getClientData();
187      request.setClientData(null);
188      AdbMessage messageToDispatch = null;

189      if (message == currentCommand) {
190          int dataLength = message.getDataLength();
191          // read data if length > 0
192          if (dataLength > 0) {
193              message.readData(getInRequest(), dataLength);
194              currentData = message;
195          } else {
196              messageToDispatch = message;
197          }
198          currentCommand = null;
199      } else if (message == currentData) {
200          messageToDispatch = message;
201          currentData = null;
202      }
```

Line 190 obtains the length of the message data.

Line 192 checks if the length of the data is larger than zero. If it does, which means it is a command with data and lines 193-198 is performed. If it is not, which means the command does not carries and data and line 196 is executed where the message(the command) is assigned to the messageToDispatch directly.

Line 193 reads and stores size of length data from the IN endpoint to the mDataBuffer of the message . First, the getInRequest() is executed that generates another request. Either a new request or the first request of the mInRequestPool is returned. The request is passed as the first argument and length of the data as the second argument to the readData() method. The readData() will first run setClientData(this) that set the client data of the request to be this message, it then will perform yet another queue() operation which read number of length data from the IN endpoint to the mDataBuffer of the message.

First of all, the message is retrieve form the request using getClientData(). It then checks if the message is a currentCommand or currentData. If the message is a currentData, it sets messageToDispatch to message and currentData = null; If the message equals to currentCommand, it further determind if the datalength is larger than 0. If datalength is larger than zero, it perform readData and set the currentData to message, it datalength equals o, it set messageToDispatch to message.



AdbDevice.java

```
186         AdbMessage message = (AdbMessage)request.getClientData();
187         request.setClientData(null);
188         AdbMessage messageToDispatch = null;

189         if (message == currentCommand) {
190             int dataLength = message.getDataLength();
191             // read data if length > 0
192             if (dataLength > 0) {
193                 message.readData(getInRequest(), dataLength);
194                 currentData = message;
195             } else {
196                 messageToDispatch = message;
197             }
198             currentCommand = null;
199         } else if (message == currentData) {
200             messageToDispatch = message;
201             currentData = null;
202         }
```

Both currentCommand and currentData are of type AdbMessage. It is so design for the USB make the bulk and interrupt transfer into three phase: the request, the data and ack.

Request → Data → Ack

Because the queue() in the readData() is an asynchronous operation, the program will immediately return and continued. The read data will however return via requestWait() at the beginning of next iteration of the loop.

Line 194 will immediate followed which set the currentData to be the message. This will be put to test where the receiving message on the next iteration of the loop equals the present currentData setting.

Line 195-197 are skilled.

Line 198 is executed that set the currentCommand to be null.

The following statement from 199-209 are skipped. Since the transaction has not ended yet and more data to come. Nothing to dispatch still.



AdbDevice.java

```
203         if (messageToDispatch != null) {
204             // queue another read first
205             currentCommand = new AdbMessage();
206             currentCommand.readCommand(getInRequest());

207             // then dispatch the current message
208             dispatchMessage(messageToDispatch);
209         }

210         // put request back into the appropriate pool
211         if (request.getEndpoint() == mEndpointOut) {
212             releaseOutRequest(request);
213         } else {
214             synchronized (mInRequestPool) {
215                 mInRequestPool.add(request);
216             }
217         }
218     }
219 }
220 }
221 }
```

Line 211-217 are then executed. The code here determine if the current request is an in or put request. It then return the request back to the proper request pool for recycling. We have retained all the information we need in the local variable `currentCommand` and `currentData` of type `AdbMessage`.

It makes you wonder why there isn't a `releaseInRequest()` since a `releaseOutRequest` exist.

† This makes you wonder, the request and the message are sent separately.



AdbDevice.java/WaiterThread()

```
168     private class WaiterThread extends Thread {
169         public boolean mStop;

170         public void run() {
171             // start out with a command read
172             AdbMessage currentCommand = new AdbMessage();
173             AdbMessage currentData = null;
174             // FIXME error checking
175             currentCommand.readCommand(getInRequest());

176         while (true) {
177             synchronized (this) {
178                 if (mStop) {
179                     return;
180                 }
181             }
182             UsbRequest request = mDeviceConnection.requestWait();
183             if (request == null) {
184                 break;
185             }
186         }
187     }
188 }
```

On the next iteration, the previous queue() from readData() operation is returned and the result of that operation is returned by requestWait() as client data inside of the returned request.



AdbDevice.java

```
186      AdbMessage message = (AdbMessage)request.getClientData();
187      request.setClientData(null);
188      AdbMessage messageToDispatch = null;

189      if (message == currentCommand) {
190          int dataLength = message.getDataLength();
191          // read data if length > 0
192          if (dataLength > 0) {
193              message.readData(getInRequest(), dataLength);
194              currentData = message;
195          } else {
196              messageToDispatch = message;
197          }
198          currentCommand = null;
199      } else if (message == currentData) {
200          messageToDispatch = message;
201          currentData = null;
202      }
```

Line 186 retrieves the client data that an AdbMessage and stores it in the message.

Line 187 nullify the client data of the request.

Line 188 rests the messageToDispatch.

Line 189 is executed first. This time however, the check fails for the receiving message is not a currentCommand which has been nullify during the last iteration.

Line 199 is then checked. Bingo, the receiving equals the currentData.

Line 200 set the messageToDispatch as message.

Both currentCommand and currentData are of type AdbMessage.

It is so design for the USB make the bulk and interrupt transfer into three phase: the request, the data and ack.

Line 201 rests the currentData.

Request → Data → Ack



AdbDevice.java

```
203         if (messageToDispatch != null) {
204             // queue another read first
205             currentCommand = new AdbMessage();
206             currentCommand.readCommand(getInRequest());

207             // then dispatch the current message
208             dispatchMessage(messageToDispatch);
209         }

210         // put request back into the appropriate pool
211         if (request.getEndpoint() == mEndpointOut) {
212             releaseOutRequest(request);
213         } else {
214             synchronized (mInRequestPool) {
215                 mInRequestPool.add(request);
216             }
217         }
218     }
219 }
220 }
221 }
```

It then comes to line 203.

Line 203 checks if `messageToDispatch` is defined which we know it is true.

Line 205 then creates another `currentCommand` and sets it to be the client data of the newly obtained request. In addition, a new `queue()` is issued inside of the `readCommand()` method. All these efforts are designed to set the state machine back to initial state (similar to the things done at line 172-175 that a prior to the `run()` starts).

Line 208 dispatches the `messageToDispatch`.

Line 211-215, as before, release the current request for recycling.



AdbDevice.java

```
186         AdbMessage message = (AdbMessage)request.getClientData();
187         request.setClientData(null);
188         AdbMessage messageToDispatch = null;

189         if (message == currentCommand) {
190             int dataLength = message.getDataLength();
191             // read data if length > 0
192             if (dataLength > 0) {
193                 message.readData(getInRequest(), dataLength);
194                 currentData = message;
195             } else {
196                 messageToDispatch = message;
197             }
198             currentCommand = null;
199         } else if (message == currentData) {
200             messageToDispatch = message;
201             currentData = null;
202         }
```

However, if the message equals the currentCommand and carries no data, then line 196 is executed that set the messageToDispatch to be message.

Line 198 set nullify the currentCommand.

It then repeat the operation between line 201 and 217. Specifically,

It re-initializes the state machine by generate a new AdbMessage that the currentCommand.

It then get a new in request and set the currentCommand to it's client data.

It performs an queue() on the new request.

It then dispatchs the messageToDispatch.

Finally, recycles the current command.



Aftermath

- ▶ So if I get this right,
 - ▶ Everything other than the server in the ADB design can be regarded as a Client. It can be a local software module running in the host or a remote physical USB device with ADB daemon.
 - ▶ The term Data in the method getClientData() and setClientData() are actual data/message that encapsulated inside of the corresponding UsbRequest.
 - ▶ For asynchronous Bulk endpoing transaction, actual data transfer is achieved using queue(). queue() will immediately return with a boolean value. But the actual result is returned by requestWait() as request.



Aftermath

- ▶ **UsbDeviceconnection** and **UsbRequest** can both be used for data transferring on USB device, what's the difference between them?
- ▶ The **UsbRequest** represents a USB request packet. You can set and get the packet data using `setClientData()` and `getClientData()` and queue the request for transferring using `queue()`.
- ▶ The **UsbDeviceconnection** is specifically used for sending and receiving data the control to a USB device.



Aftermath

- ▶ Everything falls back to the fundamental question, what is the class of `UsbRequest`? It is a,
 - ▶ A class representing USB request packet.
 - ▶ can be used for both reading and writing data to or from a `UsbDeviceConnection`.
 - ▶ `UsbRequests` can be used to transfer data on bulk and interrupt endpoints.
 - ▶ Requests on bulk endpoints can be sent synchronously via `bulkTransfer(UsbEndpoint, byte[], int, int)` or asynchronously via `queue(ByteBuffer, int)` and `requestWait()`. Requests on interrupt endpoints are only send and received asynchronously.
 - ▶ Requests on endpoint zero are not supported by this class; use `controlTransfer(int, int, int, int, byte[], int, int)` for endpoint zero requests instead.



Aftermath

- ▶ This is from

<http://stackoverflow.com/questions/12345953/android-usb-host-asynchronous-interrupt-transfer>

- ▶ In all, the `queue()` method of the `UsbRequest` can be used either for transferring request IN/OUT.
- ▶ For asynchronuous (non-blocking) transaction, the `requestWait()` from `UsbConnection` can be used.
- ▶ The `requestWait()` will return the request been transferred no matter it is an IN/OUT.



AdbSocket.java

In spite the name, the class has nothing to do with the `java.net.Socket`. It is to communicate with the `AdbDevice`. In this program, `AdbSocket` is used to implement ADB command. There should be one `AdbSocket` for each ADB command. Interestingly, in the current implementation, there is only one `AdbSocket` implemented that the “`shell:exec logcat`”. It is a shell command that open a shell at the peer device and execute `logcat` command.

```
1
:
16      package com.android.adb;

17      /* This class represents an adb socket.  adb supports multiple independent
18         * socket connections to a single device. Typically a socket is created
19         * for each adb command that is executed.
20         */
```



AdbSocket.java

```
21      public class AdbSocket {  
22          private final AdbDevice mDevice;  
23          private final int mId;  
24          private int mPeerId;  
  
25          public AdbSocket(AdbDevice device, int id) {  
26              mDevice = device;  
27              mId = id;  
28          }  
}
```

The class is instantiated once at the method `openSocket()` of `AdbDevice` of the whole program execution. `openSocket()` is further called by method `handleDeviceOnline()` of the `AdbTestActivity`. The argument passed to the `handleDeviceOnline()` is a string "shell:exec logcat" that the only ADB command implemented currently in this program.

Line 21 starts the definition of the class.

Line 23 and 24 define two stream id that the id of local socket `mId` and the id of the peer device that the `mPeerId`. Noticed that, the `mId` is modified by `final` which means it's value can not be changed once initialized.

Line 25-28 declares the constructor which takes two arguments, the device and id.

Line 26-27 assign arguments to local fields.

```
public AdbSocket openSocket(String destination) {  
    AdbSocket socket;  
    synchronized (mSockets) {  
        int id = mNextSocketId++;  
        socket = new AdbSocket(this, id);  
        mSockets.put(id, socket);  
    }  
    if (socket.open(destination)) {  
        return socket;  
    } else {  
        return null;  
    }  
}
```

`handleDeviceOnline()` / `AdbTestActivity` →
`openSocket()` / `AdbDevice` →
`AdbSocket`

AdbSocket.java

```
29      public int getId() {
30          return mId;
31      }

32      public boolean open(String destination) {
33          AdbMessage message = new AdbMessage();
34          message.set(AdbMessage.A_OPEN, mId, 0, destination);
35          if (! message.write(mDevice)) {
36              return false;
37          }
38          synchronized (this) {
39              try {
40                  wait();
41              } catch (InterruptedException e) {
42                  return false;
43              }
44          }
45          return true;
46      }
```

Line 29-31 is the getId() method which is trivial.

Line 32-46 is the open() method. In this program, the method is called by handleDeviceOnline() and passed with argument “destination” that a string of “shell:exec logcat.”

Line 33 instances a new AdbMessage message.

Line 34 calls set() methods to set the content of the message. The message is an OPEN command, the arg0 is the int mid, the arg1 is 0, and the data/destination contains the string “shell:exec logcat”

Line 35 tries to write() the above message to mDevice which is a AdbDevice passed from penSocket(). This message will be sent to the peer device and runs there are a shell command using queue() method.

Line 36, if write() failed, returns false.

Method/class

handleDeviceOnline()/AdbTestActivity →

openSocket(“shell:exec logct”)/AdbDevice →

open(“shell:exec logct”)/AdbSocket →

Write()/AdbMessage →

Queue()/UsbRequest or releaseOutRequest()/AdbDevice



AdbSocket.java

```
29      public int getId() {
30          return mId;
31      }

32      public boolean open(String destination) {
33          AdbMessage message = new AdbMessage();
34          message.set(AdbMessage.A_OPEN, mId, 0, destination);
35          if (! message.write(mDevice)) {
36              return false;
37          }
38          synchronized (this) {
39              try {
40                  wait();
41              } catch (InterruptedException e) {
42                  return false;
43              }
44          }
45          return true;
46      }
```

Line 38 locks the current AdbSocket object.

Line 39-43 gives a try-catch block waiting it to be unblocked by another thread issuing an notify() call. Actually, this is done when a A_OKY command that send from the peer device to the handleMessage() method of the AdbSocket. (This happens when the peer device sending an IN request that contains the command A_OKY and dispatched to the handleMessage() through dispatchMessage().)

Well, this makes perfectly sense since the thread will continue only when the peer device is ready.

Now, the notify() and wait() come from the same class. Will they interlock each other and cause deadlock !?

handleDeviceOnline()/AdbTestActivity →
openSocket("shell:exec logcat")/AdbDevice →
open("shell:exec logcat")/AdbSocket →
Write()/AdbMessage →
Queue()/UsbRequest or releaseOutRequest()/AdbDevice

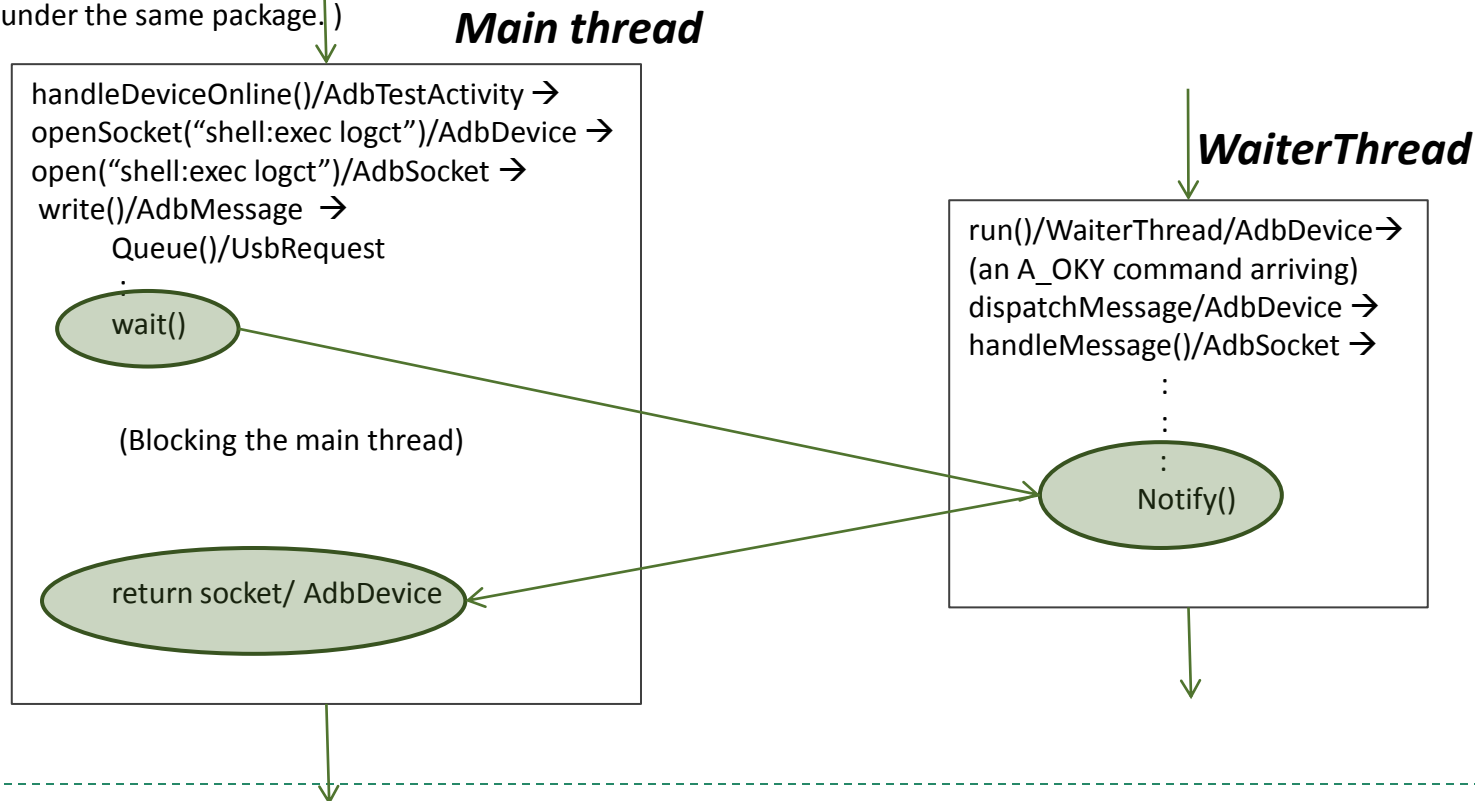
and wait()



AdbSocket.java

Since both `notify()` and `wait()` are implemented in the same class, is it possible for them to interlock each other and causing deadlock?

Well, this is how it goes. The `wait()` operation is executed from the main thread, while the `notify()` is executed from the `WaiterThread`. While the main thread is block by the `wait()`, the `WaiterThread` is still going. When the peer device send a `A_OKAY` command back to the host, the `run()` method of the `WaiterThread` will catch that message and `notify()` the unleash main thread to continue processing. (The whole project runs as a process, inside that process we can have multiple threads each with it's own data while running the same code. This says, the `WaiterThread` can call all the method and use public/default fields that under the same package.)



AdbSocket/handleMessage()

```
47     public void handleMessage(AdbMessage message) {
48         switch (message.getCommand()) {
49             case AdbMessage.A_OKAY:
50                 mPeerId = message.getArg0();
51                 synchronized (this) {
52                     notify();
53                 }
54                 break;
55             case AdbMessage.A_WRTE:
56                 mDevice.log(message.getDataString());
57                 sendReady();
58                 break;
59         }
60     }
```

READY(local-id, remote-id, "")

Line 47-60 declare the method handleMessage(). It is called by dispatchMessage() of AdbDevice. The dispatchMessage() is called when the command of the AdbMessage is either of : A_OPEN, A_OKAY, A_CLSE, and A_WRTE. Interestingly enough, inside of this method is yet another switch-case block just like the one in its caller dispatchMessage(). So, it kind of makes you wonder why not combine them together?

Line 49 deals with the case of A_OKAY that basically a command from the peer device.

Line 50 is executed. It gets the local-id (i.e. the sender of this message) that arg 0 of the receiving AdbMessage and assigns the id to the mPeerId (Do notice that the mPeerId is a private class member. Also, AdbMessage is sent from the peer device and the local-id here is the opposite to the receiving end that where this program is running 😊)

The value of mPeerId will be used in the method sendReady() of the AdbSocket.



AdbSocket/handleMessage()

```
47     public void handleMessage(AdbMessage message) {  
48         switch (message.getCommand()) {  
49             case AdbMessage.A_OKAY:  
50                 mPeerId = message.getArg0();  
51                 synchronized (this) {  
52                     notify();  
53                 }  
54                 break;  
55             case AdbMessage.A_WRITE:  
56                 mDevice.log(message.getDataString());  
57                 sendReady();  
58                 break;  
59         }  
60     }
```

Line 51-52 is the tricky part, line 51 locks the AdbSocket object. It then calls notify(). The notify() method will wake up threads that are waiting for this object and unblock it (the main thread.) [See discussion in slide for detail.](#)

READY(local-id, remote-id, "")

The local-id and remote-id are the so called stream id and are arbitrary chosen int.



AdbSocket/handleMessage()

```
47         public void handleMessage(AdbMessage message) {
48             switch (message.getCommand()) {
49                 case AdbMessage.A_OKAY:
50                     mPeerId = message.getArg0();
51                     synchronized (this) {
52                         notify();
53                     }
54                     break;
55                 case AdbMessage.A_WRTE:
56                     mDevice.log(message.getDataString());
57                     sendReady();
58                     break;
59             }
60         }
```

Line 55-59 deal with the case of A_WRTE. (Frankly, the command name A_WRITE is very confusing. Base on the USB terminologies, a write operation writes data from the host to the device. However, in the code here, it is done oppositely that writing data form the device to the host!!)

Line 56 calls the log() method with the data string (payload) of the message as argument. The logged data eventually will be printed on the textView of the UI thread.

Line 57 calls method sendReady() which send the command A_OKAY to the peer socket mPeerId.

So, the A_OKAY command goes both way. It is sent by the peer device when the device is opened. On the other hand, it is send by the host when then host is read to receive more data(log).



AdbSocket/sendReady()

```
61     private void sendReady() {  
62         AdbMessage message = new AdbMessage();  
63         message.set(AdbMessage.A_OKAY, mId, mPeerId);  
64         message.write(mDevice);  
65     }  
66 }
```

Line 61-65 declare the method sendReady().

Line 62 instantiates a AdbMessage.

Line 63 calls set() method to setup this message with argument: command-A_OKAY, host socket id and remote socket id. Well, this is basically all a READY command needed. See ADB protocol.

Line 64 calls method write() which write the above AdbMessage with argument mDevice that the AdbDevice. The message is “write” to the AdbDevice and hence to the peer USB device using queue() from UsbRequest.





AdbMessage.java

```
:
16      package com.android.adb;

17      import android.hardware.usb.UsbRequest;

18      import java.nio.ByteBuffer;
19      import java.nio.ByteOrder;

20      /* This class encapsulates and adb command packet */
21      public class AdbMessage {

22          // command names
23          public static final int A_SYNC = 0x434e5953;
24          public static final int A_CNXXN = 0x4e584e43;
25          public static final int A_OPEN = 0x4e45504f;
26          public static final int A_OKAY = 0x59414b4f;
27          public static final int A_CLSE = 0x45534c43;
28          public static final int A_W RTE = 0x45545257;
```

Lines 22-28 define adb protocol commands and theirs' corresponding constants.

A_SYNC → synchronous

A_CNXXN → connect

A_OPEN → open

A_OKAY → ready

A_CLSE → close

A_WRITE → write



AdbMessage.java

```
29      // ADB protocol version
30      public static final int A_VERSION = 0x01000000;

31      public static final int MAX_PAYLOAD = 4096;

32      private final ByteBuffer mMessageBuffer;
33      private final ByteBuffer mDataBuffer;

34      public AdbMessage() {
35          mMessageBuffer = ByteBuffer.allocate(24);
36          mDataBuffer = ByteBuffer.allocate(MAX_PAYLOAD);
37          mMessageBuffer.order(ByteOrder.LITTLE_ENDIAN);
38          mDataBuffer.order(ByteOrder.LITTLE_ENDIAN);
39      }
```

Line 29 defines a constant value of adb version A_VERSION that 0x01000000.

Line 30 defines a constant value of adb maximum payload that 4096.

Line 32 and 33 declare two final variables of ByteBuffer: mMessageBuffer and mDataBuffer. (Being a final variable of reference type means the reference itself is constant but the referenced value can be changed!!)

Line 34-39 declare the constructor of AdbMessage(). In the constructor,

Line 35 instantiates the mMessageBuffer by calling allocate() method and setting the size to be 24 bytes.

Line 36 instantiates the mDataBuffer by calling allocate() and having size of MAX_PAYLOAD.

Line 37-38 set the endianness of the corresponding buffers to be LITTLE_ENDIAN.

In all, an AdbMessage consists of two parts: the mMessageBuffer and mDataBuffer.

The size of mMessageBuffer is a constant of 24 bytes.

The size of mDataBuffer varies along with the size of data.



AdbMessage.java/set()

```
40      // sets the fields in the command header
41      public void set(int command, int arg0, int arg1, byte[] data) {
42          mMessageBuffer.putInt(0, command);
43          mMessageBuffer.putInt(4, arg0);
44          mMessageBuffer.putInt(8, arg1);
45          mMessageBuffer.putInt(12, (data == null ? 0 : data.length));
46          mMessageBuffer.putInt(16, (data == null ? 0 : checksum(data)));
47          mMessageBuffer.putInt(20, command ^ 0xFFFFFFFF);
48          if (data != null) {
49              mDataBuffer.put(data, 0, data.length);
50          }
51      }
```

Line 40-51 declare the set() method of the AdbMessage.

The method takes 4 arguments each of 4 bytes that constitute 24bytes of **mMessageBuffer**.

If there are data that given by line 48, then the method also initiates the **mDataBuffer** of the AdbMessage.

The code pack input parameters in order and 4 bytes a time.

The ^ operator from line 47 is a bitwise exclusion or, its use is unclear at the time (some kind of checksum? At the receiving end, the received command is bitwise and/or to this value and some how verify that the transition is valid one)



AdbMessage.java/set()

```
40      // sets the fields in the command header
41      public void set(int command, int arg0, int arg1, byte[] data) {
42          mMessageBuffer.putInt(0, command);
43          mMessageBuffer.putInt(4, arg0);
44          mMessageBuffer.putInt(8, arg1);
45          mMessageBuffer.putInt(12, (data == null ? 0 : data.length));
46          mMessageBuffer.putInt(16, (data == null ? 0 : checksum(data)));
47          mMessageBuffer.putInt(20, command ^ 0xFFFFFFFF);
48          if (data != null) {
49              mDataBuffer.put(data, 0, data.length);
50          }
51      }
```

Adb Packet Format

local-id, remote-id: stream identified

CONNECT(version, maxdata, "system-identity-string")

Currently, version=0x01000000 and maxdata=4096

system identity string should be "<systemtype>:<serialno>:<banner>"

systemtype is "bootloader", "device", or "host",

serialno is some kind of unique ID (or empty), and

banner is a human-readable version or identifier string (informational only).

OPEN(local-id, 0, "destination")

READY(local-id, remote-id, "")

WRITE(0, remote-id, "data")

CLOSE(local-id, remote-id, "")

SYNC(online, sequence, "")



AdbMessage.java/set()

Destination

"tcp:<host>:<port>" - host may be omitted to indicate localhost

"udp:<host>:<port>" - host may be omitted to indicate localhost

"local-dgram:<identifier>"

"local-stream:<identifier>"

"shell" - local shell service

"upload" - service for pushing files across (like apto's /sync)

"fs-bridge" - FUSE protocol filesystem bridge

Adb Packet Format

local-id, remote-id: stream identified

CONNECT(version, maxdata, "system-identity-string")

Currently, version=0x01000000 and maxdata=4096

system identity string should be "<systemtype>:<serialno>:<banner>"

systemtype is "bootloader", "device", or "host",

serialno is some kind of unique ID (or empty), and

banner is a human-readable version or identifier string (informational only).

OPEN(local-id, 0, "***destination***")

result either a READY or CLOSE message from the peer.

READY(local-id, remote-id, "")

WRITE(0, remote-id, "data")

CLOSE(local-id, remote-id, "")

SYNC(online, sequence, "")



AdbMessage.java

```
52      public void set(int command, int arg0, int arg1) {  
53          set(command, arg0, arg1, (byte[])null);  
54      }  
55      public void set(int command, int arg0, int arg1, String data) {  
56          // add trailing zero  
57          data += "\0";  
58          set(command, arg0, arg1, data.getBytes());  
59      }
```

In addition to the set() method defined in the previous slide, there are two other set() method that overload the previous one. These two wrap the 1st set() method, and are tailed to fit special usage and for convenience of programmer.

Line 52-54 is a set() method with 3 arguments: command, arg0, and arg1 that without the data argument. It is for setting up null data message.

Line 55-59 is a set() method with 4 arguments. This is basically the same as the first set() except the data is given in string instead of a byte array. It is used to set up message where data is passed as String.

Both of these method in terms calls the first set() method to complete the message setting.

Incidentally, the first set() is never used in this program. The second and third are used instead when required.



AdbMessage.java/set()

```
40      // sets the fields in the command header
41      public void set(int command, int arg0, int arg1, byte[] data) {
42          mMessageBuffer.putInt(0, command);
43          mMessageBuffer.putInt(4, arg0);
44          mMessageBuffer.putInt(8, arg1);
45          mMessageBuffer.putInt(12, (data == null ? 0 : data.length));
46          mMessageBuffer.putInt(16, (data == null ? 0 : checksum(data)));
47          mMessageBuffer.putInt(20, command ^ 0xFFFFFFFF);
48          if (data != null) {
49              mDataBuffer.put(data, 0, data.length);
50          }
51      }
```

Line 72 of the sendReady() of AdbSocket.java

message.set(AdbMessage.A_OKAY, mId, mPeerId);

It is an reference to the second set() method that

public void set(int command, int arg0, int arg1)

The command is AdbMessage.A_OKAY

The arg0 : mid

The arg1 : mPeerId

Line 146 of the connect() of the AdbDevice.java

It references to the method public void set(int command, int arg0, int arg1, String data)

message.set(AdbMessage.A_CNXXN, AdbMessage.A_VERSION, AdbMessage.MAX_PAYLOAD, "host::\0");

Line 40 of the open() of AdbSocket.java

It references to the method public void set(int command, int arg0, int arg1, String data)

message.set(AdbMessage.A_OPEN, mId, 0, destination);

AdbMessage.java/set()

Destination

"tcp:<host>:<port>" - host may be omitted to indicate localhost

"udp:<host>:<port>" - host may be omitted to indicate localhost

"local-dgram:<identifier>"

"local-stream:<identifier>"

"shell" - local shell service

"upload" - service for pushing files across (like apto's /sync)

"fs-bridge" - FUSE protocol filesystem bridge

Line 72 of the sendReady() of AdbSocket.java

```
message.set(AdbMessage.A_OKAY, mId, mPeerId);
```

It is an reference to the second set() method that

```
52 public void set(int command, int arg0, int arg1)
```

The command is AdbMessage.A_OKAY

The arg0 : mId

The arg1 : mPeerId

Line 146 of the connect() of the AdbDevice.java

It references to the method public void set(int command, int arg0, int arg1, String data)

```
message.set(AdbMessage.A_CNXXN, AdbMessage.A_VERSION, AdbMessage.MAX_PAYLOAD, "host::\0");
```

Line 40 of the open() of AdbSocket.java

It references to the method public void set(int command, int arg0, int arg1, String data)

```
message.set(AdbMessage.A_OPEN, mId, 0, destination);
```

Adb Packet Format

local-id, remote-id: stream identified

CONNECT(version, maxdata, "system-identity-string")

Currently, version=0x01000000 and maxdata=4096

system identity string should be "<systemtype>:<serialno>:<banner>"

systemtype is "bootloader", "device", or "host",

serialno is some kind of unique ID (or empty), and

banner is a human-readable version or identifier string (informational only).

OPEN(local-id, 0, "**destination**")

result either a READY or CLOSE message from the peer.

READY(local-id, remote-id, "")

WRITE(0, remote-id, "data")

~~CLOSE(local-id, remote-id, "")~~

SYNC(online, sequence, "")

AdbMessage.java

```
60      // returns the command's message ID
61      public int getCommand() {
62          return mMessageBuffer.getInt(0);
63      }

64      // returns command's first argument
65      public int getArg0() {
66          return mMessageBuffer.getInt(4);
67      }

68      // returns command's second argument
69      public int getArg1() {
70          return mMessageBuffer.getInt(8);
71      }

72      // returns command's data buffer
73      public ByteBuffer getData() {
74          return mDataBuffer;
75      }
```

Line 10-72 declare three methods, `getCommand()`, `getArg0()`, `getArg1()`, and `getData()`. These methods perform reverse operation of the `set()` method above, they retrieve individual entries from the message.

The Adb packet uses different format for different command. This complicates the implementation. The command `CONNECT` and `SYNC` are different from others, while `OPEN`, `READY`, `WRITE` and `CLOSE` are the same.

The argument 0, and 1 from these command are referring to stream identifiers which corresponds to a particular `AdbSocket`.

(I am still having difficult decipher the use of each entry of the message.. For instance, the `Arg1` seems given the `MAX_PAYLOAD` of the message, however it is used in the `dispatchMessage()` as index to the `mSocket` for retrieving specific socket.)

Adb Packet Format

local-id, remote-id: stream identified
`CONNECT(version, maxdata, "system-identity-string")`
`OPEN(local-id, 0, "destination")`
`READY(local-id, remote-id, "")`
`WRITE(0, remote-id, "data")`
`CLOSE(local-id, remote-id, "")`
`SYNC(online, sequence, "")`

AdbMessage.java

```
76      // returns command's data length
77      public int getDataLength() {
78          return mMessageBuffer.getInt(12);
79      }

80      // returns command's data as a string
81      public String getDataString() {
82          int length = getDataLength();
83          if (length == 0) return null;
84          // trim trailing zero
85          return new String(mDataBuffer.array(), 0, length - 1);
86      }
```

Relationship between the message and mDataBuffer.

```
public AdbMessage() {
    mMessageBuffer = ByteBuffer.allocate(24);
    mDataBuffer = ByteBuffer.allocate(MAX_PAYLOAD);
    mMessageBuffer.order(ByteOrder.LITTLE_ENDIAN);
    mDataBuffer.order(ByteOrder.LITTLE_ENDIAN);
}
```

Line 76-79 declare the `getDataLength()` method of the `AdbMessage`. It is called by the `write()` method.

Line 78 returns the int value of `mMessageBuffer` at index 12.

Line 80-86 declare the method `getDataString()`.

Line 82 calls `getDataLength()` to retrieve the length of the data from the message.

Line 83 checks if the length is 0.

Line 85 instantiate a new string using string constructor `String(byte[] data, int offset, int byteCount)`. The string is returned to the caller.



AdbMessage.java/write()

```
87         public boolean write(AdbDevice device) {
88             synchronized (device) {
89                 UsbRequest request = device.getOutRequest();
90                 request.setClientData(this);
91                 if (request.queue(mMessageBuffer, 24)) {
92                     int length = getDataLength();
93                     if (length > 0) {
94                         request = device.getOutRequest();
95                         request.setClientData(this);
96                         if (request.queue(mDataBuffer, length)) {
97                             return true;
98                         } else {
99                             device.releaseOutRequest(request);
100                            return false;
101                        }
102                    }
103                    return true;
104                } else {
105                    device.releaseOutRequest(request);
106                    return false;
107                }
108            }
109        }
```

Line 87-109 declare the write() method of the AdbMessage. The phrase write here means (the AdbDevice) transfers an out going message to the external USB device. It is called by the connect() of the AdbDevice and sendReady() and open() methods of the AdbSocket.

This might sounds strange for no a mountable data transfer is done here. But considering the host of this application is responsible only to initiate the application while the peer device is responsible of transferring all the log data.

Line 88 locks the device that an AdbDevice which passed as argument to the method.

Line 89 calls getOutRequest() of the AdbDevice to get an request for the specific connection and endpoint from the mOutRequestPool.

Line 90 need some explanation and we will leave it to the next slide below.

An AdbMessage is broken into the command part and data part and sent in separate out request packets. The Usb packet are transfer in the sequence of REQ-DATA-ACK



How does setClientData() work in the AdbMessage.java/write()

AdbDevice.java

```
124 // send a connect command
125 private void connect() {
126     AdbMessage message = new AdbMessage();
127     message.set(AdbMessage.A_CNXXN, AdbMessage.A_VERSION, AdbMessage.MAX_PAYLOAD, "host::\0");
128     message.write(this);
129 }
```

AdbMessage.java

```
87 public boolean write(AdbDevice device) {
88     synchronized (device) {
89         UsbRequest request = device.getOutRequest();
90         request.setClientData(this);
91         if (request.queue(mMessageBuffer, 24)) {
92             int length = getDataLength();
93             if (length > 0) {
94                 request = device.getOutRequest();
95                 request.setClientData(this);

```

Line 90 sets the USB request packet by calling `request.setClientData(this)`.

The packet is named as “request.” The content `mClientData` of the packet is set by “this” that referred to the “AdbMessage” of the `message.write()` from the caller.

As you can see, line 126-127 had prepared the message as a **CONNECT** command, the version `0x01000000` of adb protocol, the maximum payload 4096, and the data that designated host and endpoint 0. (see `AdbMessage.java` for the meaning of the arguments. I still having difficult to decipher the meaning of these arguments though. Also see adb protocol text)



AdbMessage.java/write()

```
87         public boolean write(AdbDevice device) {
88             synchronized (device) {
89                 UsbRequest request = device.getOutRequest();
90                 request.setClientData(this);
91                 if (request.queue(mMessageBuffer, 24)) {
92                     int length = getDataLength();
93                     if (length > 0) {
94                         request = device.getOutRequest();
95                         request.setClientData(this);
96                         if (request.queue(mDataBuffer, length)) {
97                             return true;
98                         } else {
99                             device.releaseOutRequest(request);
100                            return false;
101                        }
102                    }
103                    return true;
104                } else {
105                    device.releaseOutRequest(request);
106                    return false;
107                }
108            }
109        }
```

Line 91 perform a asynchronous write of the message by calling `queue()` method of the `UsbRequest`. It sends the first 24 bytes of the packet that the content of `mMessageBuffer` (see the `set()` method that span between line 41-50 of the `AdbMessage.java` for detail.) If the `queue()` successes, then immediately line 92-104 is executed. If not, line 104-107 are executed that release the request to the Output Request Pool and return fail.

Line 92 retrieve the length of data, if there is any, by calling `getDataLength()`. The `getDataLength()` retrieves an integer value form `mMessageBuffer` at offset of 12.

Line 93 checks if the retrieved value larger than zero. If it is, then this means there are more data to the message and line 94-97 are executed.

Line 94 get an request form the output request pool.

Line 95 sets the request packet with this message as client dat. Noticed that, the message part and data part are treated separately using two requests and sent separately.



AdbMessage.java/write()

```
87         public boolean write(AdbDevice device) {
88             synchronized (device) {
89                 UsbRequest request = device.getOutRequest();
90                 request.setClientData(this);
91                 if (request.queue(mMessageBuffer, 24)) {
92                     int length = getDataLength();
93                     if (length > 0) {
94                         request = device.getOutRequest();
95                         request.setClientData(this);
96                         if (request.queue(mDataBuffer, length)) {
97                             return true;
98                         } else {
99                             device.releaseOutRequest(request);
100                            return false;
101                        }
102                    }
103                    return true;
104                } else {
105                    device.releaseOutRequest(request);
106                    return false;
107                }
108            }
109        }
```

Line 96 send the data by calling queue() with length of data from mDataBuffer.

In this program, the write() is called in the connect() of the AdbDevice and sendReady() and open() methods of the AdbSocket. A set() method of the AdbMessage was made a prior to these calls. The set() method configured the AdbMessage that to be write() to the peer device.

The mDataBuffer is set by the set() method. However, interestingly enough, the method set () is declared but never referenced in this program. It is the overloaded set() been used, and these overloaded set() are called in coonect(), sendReady() and oepn().

AdbMessage.java/write()

```
87     public boolean write(AdbDevice device) {
88         synchronized (device) {
89             UsbRequest request = device.getOutRequest();
90             request.setClientData(this);
91             if (request.queue(mMessageBuffer, 24)) {
92                 int length = getDataLength();
93                 if (length > 0) {
94                     request = device.getOutRequest();
95                     request.setClientData(this);
96                     if (request.queue(mDataBuffer, length)) {
97                         return true;
98                     } else {
99                         device.releaseOutRequest(request);
100                        return false;
101                    }
102                }
103                return true;
104            } else {
105                device.releaseOutRequest(request);
106                return false;
107            }
108        }
109    }
```

Line 99-100 are executed if queue() of the data fails. The request is released to the output request pool, and a boolean false is return.

Line 103, If queue() the message and data success, then line 103 is executed.

Line 104 is executed, if queue() the message fails and line 105-106 are executed. It release the request and return false.



AdbMessage.java/readCommand() readData()

```
110      public boolean readCommand(UsbRequest request) {  
111          request.setClientData(this);  
112          return request.queue(mMessageBuffer, 24);  
113      }  
  
114      public boolean readData(UsbRequest request, int length) {  
115          request.setClientData(this);  
116          return request.queue(mDataBuffer, length);  
117      }
```

Line 110-113 declare method readCommand(). The method is called by run() of the WaiterThread of the class AdbDevice. The method handles a IN request and retrieves the command of the request to the mMessageBuffer.

Line 111 calls the setClientData() with “this” which refer to the current instance of AdbMessage of the request.

Line 112 calls the queue() with mMessageBuffer and 24 as arguments. Noticed that, the queue() can be either receiving or sending message that pending on the endpoint direction (IN/OUT). In this case however, it operates on an IN endpoint and hence read 24 bytes from the in endpoint of the peer device and stored the read data in the mMessageBuffer.

Line 114-117 declare the readData() method that read data from the peer device and stored the read data in the mDataBuffer.



AdbMessage.java/extractString()

```
118     private static String extractString(ByteBuffer buffer, int offset, int length) {
119         byte[] bytes = new byte[length];
120         for (int i = 0; i < length; i++) {
121             bytes[i] = buffer.get(offset++);
122         }
123         return new String(bytes);
124     }
```

Line 126-135 declare the method `extractString()`. The method is used in the `toString()` of the same class to retrieve command name of the message



AdbMessage.java/toString()

```
125         @Override
126         public String toString() {
127             String commandName = extractString(mMessageBuffer, 0, 4);
128             int dataLength = getDataLength();
129             String result = "Adb Message: " + commandName + " arg0: " + getArg0() +
130                 " arg1: " + getArg1() + " dataLength: " + dataLength;
131             if (dataLength > 0) {
132                 result += (" data: \"" + getDataString() + "\"");
133             }
134             return result;
135         }
```

Line 126-135 declare the method `toString()` which transform message into a formatted human readable sting representation. Interestingly enough, this method is declared but never being referenced in the project. Something like:

Adb Message: commandName arg0: Arg0 arg1: Arg1 dataLength: dataLength data: "DataString"



AdbMessage.java/checksum()

```
136     private static int checksum(byte[] data) {  
137         int result = 0;  
138         for (int i = 0; i < data.length; i++) {  
139             int x = data[i];  
140             // dang, no unsigned ints in java  
141             if (x < 0) x += 256;  
142             result += x;  
143         }  
144         return result;  
145     }  
146 }
```

Line 136-145 give a simple implementation of the data checksum method checksum().
It generates the checksum by summing the positive integer value of all data.

The method is called by set() method of the AdbMessage to compute checksum of a non-null data.



About the `setClientData()` and `getClientData()`

- ▶ This has been confusing for not much has been said in the API except:

public Object getClientData ()

Added in API level 12

Returns the client data for the request. This can be used in conjunction with `setClientData(Object)` to associate another object with this request, which can be useful for maintaining state between calls to `queue(ByteBuffer, int)` and `requestWait()`

Returns

the client data for the request



About the `setClientData()` and `getClientData()`

- ▶ This has been confusing for not much has been said in the API except:

`public void setClientData (Object data)`

Added in API level 12

Sets the client data for the request. This can be used in conjunction with `getClientData()` to associate another object with this request, which can be useful for maintaining state between calls to `queue(ByteBuffer, int)` and `requestWait()`

Parameters

`data` the client data for the request



About the setClientData() and getClientData()

- These methods are implemented in the UsbRequest and they are:

```
public class UsbRequest {  
  
    :  
    48  // for client use  
    49  private Object mClientData;  
    :  
  
    105 public Object getClientData() {  
    106     return mClientData;  
    107 }  
    :  
  
    118 public void setClientData(Object data) {  
    119     mClientData = data;  
    120 }
```

- From, what it appears, the methods simply storing/retrieving data to/from a generic private member mClientData of type Object.



About the `setClientData()` and `getClientData()`

- ▶ In summary,
 - ▶ The `UsbRequest` is like a basket carrying data to be transfer.
 - ▶ The basket itself is only for easy of handling, its content `ClientData` however is thing that matter.
 - ▶ The type of `ClientData` has been set to be generic `Object` which make the `UsbRequest` capable of carrying all kinds of data.
 - ▶ In this program, the client data is `AdbMessage`.



Demystify

- ▶ We try to resolve exactly what `requestWait()` does by tracing the source code of Android framework.
- ▶ However, because lack of information about what type of `_REAPURB` is, and the trace ended here. It has been unsuccessful...



Demystify

► /frameworks/base/core/java/android/hardware/usb/ UsbDeviceConnection.java

```
package android.hardware.usb;
import android.util.Log;
import java.io.FileDescriptor;
public class UsbDeviceConnection {

:

    public UsbRequest requestWait() {
        UsbRequest request = native_request_wait();
        if (request != null) {
            request.dequeue();
        }
        return request;
    }
:

:
    private native int native_bulk_request(int endpoint, byte[] buffer, int length, int timeout);
    private native UsbRequest native_request_wait();
    private native String native_get_serial();
}
```



Demystify

- ▶ /frameworks/base/core/jni/android_hardware_UsbDeviceConnection.cpp

```
/frameworks/base/core/jni/android_hardware_UsbDeviceConnection.cpp
#define LOG_TAG "UsbDeviceConnectionJNI"
#include "utils/Log.h"
#include "jni.h"
#include "JNIHelp.h"
#include "android_runtime/AndroidRuntime.h"
#include <usbhost/usbhost.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

static jobject
android_hardware_UsbDeviceConnection_request_wait(JNIEnv *env, jobject thiz)
{
    struct usb_device* device = get_device_from_object(env, thiz);
    if (!device) {
        ALOGE("device is closed in native_request_wait");
        return NULL;
    }

    struct usb_request* request = usb_request_wait(device);
    if (request)
        return (jobject)request->client_data;
    else
        return NULL;
}
```

Demystify

► /system/core/libusbhost/usbhost.c

- Unfortunately, we find no information about what type of _REAPURB is, and the trace ended here.

```
/system/core/libusbhost/usbhost.c

#include <linux/usbdevice_fs.h>
#include <asm/byteorder.h>
#include "usbhost/usbhost.h"

struct usb_request *usb_request_wait(struct usb_device *dev)
{
    struct usbdevfs_urb *urb = NULL;
    struct usb_request *req = NULL;

    while (1) {
        int res = ioctl(dev->fd, USBDEVFS_REAPURB, &urb);
        D("USBDEVFS_REAPURB returned %d\n", res);
        if (res < 0) {
            if (errno == EINTR) {
                continue;
            }
            D("[ reap urb - error ]\n");
            return NULL;
        } else {
            D("[ urb @%p status = %d, actual = %d ]\n",
              urb, urb->status, urb->actual_length);
            req = (struct usb_request *)urb->usercontext;
            req->actual_length = urb->actual_length;
        }
        break;
    }
    return req;
}
```