USB Host

In this document

Related Samples

- AdbTest
- MissileLauncher

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.

API Overview

Before you begin, it is important to understand the classes that you need to work with. The following table describes the USB host APIs in the `android.hardware.usb` package.

Table 1. USB Host APIs

Class	Description
`UsbManager`	Allows you to enumerate and communicate with connected USB devices.
`UsbDevice`	Represents a connected USB device and contains methods to access its identifying information, interfaces, and endpoints.
`UsbInterface`	Represents an interface of a USB device, which defines a set of functionality for the device. A device can have one or more interfaces on which to communicate on.
`UsbEndpoint`	Represents an interface endpoint, which is a communication channel for this interface. An interface can have one or more endpoints, and usually has input and output endpoints for two-way communication with the device.
`UsbDeviceConnection`	Represents a connection to the device, which transfers data on endpoints. This class allows you to send data back and forth sychronously or asynchronously.
`UsbRequest`	Represents an asynchronous request to communicate with a device through a `UsbDeviceConnection`.
`UsbConstants`	Defines USB constants that correspond to definitions in linux/usb/ch9.h of the Linux kernel.

In most situations, you need to use all of these classes (`UsbRequest` is only required if you are doing asynchronous communication) when communicating with a USB device. In general, you obtain a `UsbManager` to retrieve the desired `UsbDevice`. When you have the device, you need to find the appropriate `UsbInterface` and the `UsbEndpoint` of that interface to communicate on. Once you obtain the correct endpoint, open a `UsbDeviceConnection` to communicate with the USB device.

Android Manifest Requirements

The following list describes what you need to add to your application's manifest file before working with the USB host APIs:

- Because not all Android-powered devices are guaranteed to support the USB host APIs, include a `<uses-feature>` element that declares that your application uses the `android.hardware.usb.host` feature.
- Set the minimum SDK of the application to API Level 12 or higher. The USB host APIs are not present on earlier API levels.
- If you want your application to be notified of an attached USB device, specify an `<intent-filter>` and `<meta-data>` element pair for the `android.hardware.usb.action.USB_DEVICE_ATTACHED` intent in your main activity. The `<meta-data>` element points to an external XML resource file that declares identifying information about the device that you want to detect.

  In the XML resource file, declare `<usb-device>` elements for the USB devices that you want to filter. The following list describes the attributes of `<usb-device>`. In general, use vendor and product ID if you want to filter for a specific device and use class, subclass, and protocol if you want to filter for a group of USB devices, such as mass storage devices or digital cameras. You can specify none or all of these attributes. Specifying no attributes matches every USB device, so only do this if your application requires it:

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

  Save the resource file in the `res/xml/` directory. The resource file name (without the .xml extension) must be the same as the one you specified in the `<meta-data>` element. The format for the XML resource file is in the example below.

Manifest and resource file examples

The following example shows a sample manifest and its corresponding resource file:

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

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

In this case, the following resource file should be saved in `res/xml/device_filter.xml` and specifies that any USB device with the specified attributes should be filtered:

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="1234" product-id="5678" class="255" subclass="66" protocol="1" />
</resources>
```

Working with Devices

When users connect USB devices to an Android-powered device, the Android system can determine whether your application is interested in the connected device. If so, you can set up communication with the device if desired. To do this, your application has to:

1. Discover connected USB devices by using an intent filter to be notified when the user connects a USB device or by enumerating USB devices that are already connected.
2. Ask the user for permission to connect to the USB device, if not already obtained.
3. Communicate with the USB device by reading and writing data on the appropriate interface endpoints.

Discovering a device

Your application can discover USB devices by either using an intent filter to be notified when the user connects a device or by enumerating USB devices that are already connected. Using an intent filter is useful if you want to be able to have your application automatically detect a desired device. Enumerating connected USB devices is useful if you want to get a list of all connected devices or if your application did not filter for an intent.

Using an intent filter

To have your application discover a particular USB device, you can specify an intent filter to filter for the `android.hardware.usb.action.USB_DEVICE_ATTACHED` intent. Along with this intent filter, you need to specify a resource file that specifies properties of the USB device, such as product and vendor ID. When users connect a device that matches your device filter, the system presents them with a dialog that asks if they want to start your application. If users accept, your application automatically has permission to access the device until the device is disconnected.

The following example shows how to declare the intent filter:

```
<activity ...>
...
    <intent-filter>
        <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
    </intent-filter>

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

The following example shows how to declare the corresponding resource file that specifies the USB devices that you're interested in:

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="1234" product-id="5678" />
</resources>
```

In your activity, you can obtain the `UsbDevice` that represents the attached device from the intent like this:

```
UsbDevice device = (UsbDevice) intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
```

Enumerating devices

If your application is interested in inspecting all of the USB devices currently connected while your application is running, it can enumerate devices on the bus. Use the `getDeviceList()` method to get a hash map of all the USB devices that are connected. The hash map is keyed by the USB device's name if you want to obtain a device from the map.

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

If desired, you can also just obtain an iterator from the hash map and process each device one by one:

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
...
HashMap<String, UsbDevice> deviceList = manager.getDeviceList();
Iterator<UsbDevice> deviceIterator = deviceList.values().iterator();
while(deviceIterator.hasNext()){
    UsbDevice device = deviceIterator.next();
    //your code
}
```

Obtaining permission to communicate with a device

Before communicating with the USB device, your application must have permission from your users.

> **Note:** If your application uses an intent filter to discover USB devices as they're connected, it automatically receives permission if the user allows your application to handle the intent. If not, you must request permission explicitly in your application before connecting to the device.

Explicitly asking for permission might be neccessary in some situations such as when your application enumerates USB devices that are already connected and then wants to communicate with one. You must check for permission to access a device before trying to communicate with it. If not, you will receive a runtime error if the user denied permission to access the device.

To explicitly obtain permission, first create a broadcast receiver. This receiver listens for the intent that gets broadcast when you call `requestPermission()`. The call to `requestPermission()` displays a dialog to the user asking for permission to connect to the device. The following sample code shows how to create the broadcast receiver:

```
private static final String ACTION_USB_PERMISSION =
    "com.android.example.USB_PERMISSION";
private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {

    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (ACTION_USB_PERMISSION.equals(action)) {
            synchronized (this) {
                UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);

                if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
                    if(device != null){
                      //call method to set up device communication
                   }
                }
                else {
                    Log.d(TAG, "permission denied for device " + device);
                }
            }
        }
    }
};
```

To register the broadcast receiver, add this in your `onCreate()` method in your activity:

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

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

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

When users reply to the dialog, your broadcast receiver receives the intent that contains the `EXTRA_PERMISSION_GRANTED` extra, which is a boolean representing the answer. Check this extra for a value of true before connecting to the device.

Communicating with a device

Communication with a USB device can be either synchronous or asynchronous. In either case, you should create a new thread on which to carry out all data transmissions, so you don't block the UI thread. To properly set up communication with a device, you need to obtain the appropriate `UsbInterface` and `UsbEndpoint` of the device that you want to communicate on and send requests on this endpoint with a `UsbDeviceConnection`. In general, your code should:

- Check a `UsbDevice` object's attributes, such as product ID, vendor ID, or device class to figure out whether or not you want to communicate with the device.
- When you are certain that you want to communicate with the device, find the appropriate `UsbInterface` that you want to use to communicate along with the appropriate `UsbEndpoint` of that interface. Interfaces can have one or more endpoints, and commonly will have an input and output endpoint for two-way communication.
- When you find the correct endpoint, open a `UsbDeviceConnection` on that endpoint.
- Supply the data that you want to transmit on the endpoint with the `bulkTransfer()` or `controlTransfer()` method. You should carry out this step in another thread to prevent blocking the main UI thread. For more information about using threads in Android, see Processes and Threads.

The following code snippet is a trivial way to do a synchronous data transfer. Your code should have more logic to correctly find the correct interface and endpoints to communicate on and also should do any transferring of data in a different thread than the main UI thread:

```
private Byte[] bytes;
private static int TIMEOUT = 0;
private boolean forceClaim = true;

...

UsbInterface intf = device.getInterface(0);
UsbEndpoint endpoint = intf.getEndpoint(0);
UsbDeviceConnection connection = mUsbManager.openDevice(device);
connection.claimInterface(intf, forceClaim);
connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT); //do in another thread
```

To send data asynchronously, use the `UsbRequest` class to `initialize` and `queue` an asynchronous request, then wait for the result with `requestWait()`.

For more information, see the AdbTest sample, which shows how to do asynchronous bulk transfers, and the MissileLauncher sample, which shows how to listen on an interrupt endpoint asynchronously.

Terminating communication with a device

When you are done communicating with a device or if the device was detached, close the `UsbInterface` and `UsbDeviceConnection` by calling `releaseInterface()` and `close()`. To listen for detached events, create a broadcast receiver like below:

```
BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

      if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
            UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
            if (device != null) {
                // call your method that cleans up and closes communication with the device
            }
        }
    }
};
```

Creating the broadcast receiver within the application, and not the manifest, allows your application to only handle detached events while it is

running. This way, detached events are only sent to the application that is currently running and not broadcast to all applications.