USB Accessory

In this document

See also

USB accessory mode allows users to connect USB host hardware specifically designed for Android-powered devices. The accessories must adhere to the Android accessory protocol outlined in the Android Accessory Development Kit documentation. This allows Android-powered devices that cannot act as a USB host to still interact with USB hardware. When an Android-powered device is in USB accessory mode, the attached Android USB accessory acts as the host, provides power to the USB bus, and enumerates connected devices. Android 3.1 (API level 12) supports USB accessory mode and the feature is also backported to Android 2.3.4 (API level 10) to enable support for a broader range of devices.

Choosing the Right USB Accessory APIs

Although the USB accessory APIs were introduced to the platform in Android 3.1, they are also available in Android 2.3.4 using the Google APIs add-on library. Because these APIs were backported using an external library, there are two packages that you can import to support USB accessory mode. Depending on what Android-powered devices you want to support, you might have to use one over the other:

- `com.android.future.usb`: To support USB accessory mode in Android 2.3.4, the Google APIs add-on library includes the backported USB accessory APIs and they are contained in this namespace. Android 3.1 also supports importing and calling the classes within this namespace to support applications written with the add-on library. This add-on library is a thin wrapper around the `android.hardware.usb` accessory APIs and does not support USB host mode. If you want to support the widest range of devices that support USB accessory mode, use the add-on library and import this package. It is important to note that not all Android 2.3.4 devices are required to support the USB accessory feature. Each individual device manufacturer decides whether or not to support this capability, which is why you must declare it in your manifest file.
- `android.hardware.usb`: This namespace contains the classes that support USB accessory mode in Android 3.1. This package is included as part of the framework APIs, so Android 3.1 supports USB accessory mode without the use of an add-on library. Use this package if you only care about Android 3.1 or newer devices that have hardware support for USB accessory mode, which you can declare in your manifest file.

Installing the Google APIs add-on library

If you want to install the add-on, you can do so by installing the Google APIs Android API 10 package with the SDK Manager. See Installing the Google APIs Add-on for more information on installing the add-on library.

API Overview

Because the add-on library is a wrapper for the framework APIs, the classes that support the USB accessory feature are similar. You can use the reference documentation for the `android.hardware.usb` even if you are using the add-on library.

> **Note:** There is, however, a minor usage difference between the add-on library and framework APIs that you should be aware of.

The following table describes the classes that support the USB accessory APIs:

Class	Description
`UsbManager`	Allows you to enumerate and communicate with connected USB accessories.
`UsbAccessory`	Represents a USB accessory and contains methods to access its identifying information.

Usage differences between the add-on library and platform APIs

There are two usage differences between using the Google APIs add-on library and the platform APIs.

If you are using the add-on library, you must obtain the `UsbManager` object in the following manner:

```
UsbManager manager = UsbManager.getInstance(this);
```

If you are not using the add-on library, you must obtain the `UsbManager` object in the following manner:

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
```

When you filter for a connected accessory with an intent filter, the `UsbAccessory` object is contained inside the intent that is passed to your application. If you are using the add-on library, you must obtain the `UsbAccessory` object in the following manner:

```
UsbAccessory accessory = UsbManager.getAccessory(intent);
```

If you are not using the add-on library, you must obtain the `UsbAccessory` object in the following manner:

```
UsbAccessory accessory = (UsbAccessory) intent.getParcelableExtra(UsbManager.EXTRA_ACCESSORY);
```

Android Manifest requirements

The following list describes what you need to add to your application's manifest file before working with the USB accesory APIs. The manifest and resource file examples show how to declare these items:

- Because not all Android-powered devices are guaranteed to support the USB accessory APIs, include a `<uses-feature>` element that declares that your application uses the `android.hardware.usb.accessory` feature.
- If you are using the add-on library, add the `<uses-library>` element specifying `com.android.future.usb.accessory` for the library.
- Set the minimum SDK of the application to API Level 10 if you are using the add-on library or 12 if you are using the `android.hardware.usb` package.
- If you want your application to be notified of an attached USB accessory, specify an `<intent-filter>` and `<meta-data>` element pair for the `android.hardware.usb.action.USB_ACCESSORY_ATTACHED` intent in your main activity. The `<meta-data>` element points to an external XML resource file that declares identifying information about the accessory that you want to detect.

  In the XML resource file, declare `<usb-accessory>` elements for the accessories that you want to filter. Each `<usb-accessory>` can have the following attributes:

  - `manufacturer`
  - `model`
  - `version`

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 also shown 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.accessory" />

    <uses-sdk android:minSdkVersion="<version>" />
    ...
    <application>
      <uses-library android:name="com.android.future.usb.accessory" />
        <activity ...>
            ...
            <intent-filter>
                <action android:name="android.hardware.usb.action.USB_ACCESSORY_ATTACHED" />
            </intent-filter>

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

In this case, the following resource file should be saved in `res/xml/accessory_filter.xml` and specifies that any accessory that has the corresponding model, manufacturer, and version should be filtered. The accessory sends these attributes the Android-powered device:

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

<resources>
    <usb-accessory model="DemoKit" manufacturer="Google" version="1.0"/>
</resources>
```

Working with Accessories

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

1. Discover connected accessories by using an intent filter that filters for accessory attached events or by enumerating connected accessories and finding the appropriate one.
2. Ask the user for permission to communicate with the accessory, if not already obtained.
3. Communicate with the accessory by reading and writing data on the appropriate interface endpoints.

Discovering an accessory

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

Using an intent filter

To have your application discover a particular USB accessory, you can specify an intent filter to filter for the `android.hardware.usb.action.USB_ACCESSORY_ATTACHED` intent. Along with this intent filter, you need to specify a resource file that specifies properties of the USB accessory, such as manufacturer, model, and version. When users connect an accessory that matches your accessory filter,

The following example shows how to declare the intent filter:

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

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

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

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

<resources>
    <usb-accessory manufacturer="Google, Inc." model="DemoKit" version="1.0" />
</resources>
```

In your activity, you can obtain the `UsbAccessory` that represents the attached accessory from the intent like this (with the add-on library):

```
UsbAccessory accessory = UsbManager.getAccessory(intent);
```

or like this (with the platform APIs):

```
UsbAccessory accessory = (UsbAccessory)intent.getParcelableExtra(UsbManager.EXTRA_ACCESSORY);
```

Enumerating accessories

You can have your application enumerate accesories that have identified themselves while your application is running.

Use the `getAccessoryList()` method to get an array all the USB accessories that are connected:

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
UsbAccessory[] accessoryList = manager.getAcccessoryList();
```

> **Note:** Only one connected accessory is supported at a time.

Obtaining permission to communicate with an accessory

Before communicating with the USB accessory, your applicaton must have permission from your users.

> **Note:** If your application uses an intent filter to discover accessories 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 accessory.

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

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 accessory. 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) {
                UsbAccessory accessory = (UsbAccessory) intent.getParcelableExtra(UsbManager.EXTRA_ACCESSORY);

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

To register the broadcast receiver, put 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 accessory, call the `requestPermission()` method:

```
UsbAccessory accessory;
...
mUsbManager.requestPermission(accessory, 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 accessory.

Communicating with an accessory

You can communicate with the accessory by using the `UsbManager` to obtain a file descriptor that you can set up input and output streams to read and write data to descriptor. The streams represent the accessory's input and output bulk endpoints. You should set up the communication between the device and accessory in another thread, so you don't lock the main UI thread. The following example shows how to open an accessory to communicate with:

```
UsbAccessory mAccessory;
ParcelFileDescriptor mFileDescriptor;
FileInputStream mInputStream;
FileOutputStream mOutputStream;

...

private void openAccessory() {
    Log.d(TAG, "openAccessory: " + accessory);
    mFileDescriptor = mUsbManager.openAccessory(mAccessory);
    if (mFileDescriptor != null) {
        FileDescriptor fd = mFileDescriptor.getFileDescriptor();
        mInputStream = new FileInputStream(fd);
        mOutputStream = new FileOutputStream(fd);
        Thread thread = new Thread(null, this, "AccessoryThread");
        thread.start();
    }
}
```

In the thread's `run()` method, you can read and write to the accessory by using the `FileInputStream` or `FileOutputStream` objects. When reading data from an accessory with a `FileInputStream` object, ensure that the buffer that you use is big enough to store the USB packet data. The Android accessory protocol supports packet buffers up to 16384 bytes, so you can choose to always declare your buffer to be of this size for simplicity.

> **Note:** At a lower level, the packets are 64 bytes for USB full-speed accessories and 512 bytes for USB high-speed accessories. The Android accessory protocol bundles the packets together for both speeds into one logical packet for simplicity.

For more information about using threads in Android, see Processes and Threads.

Terminating communication with an accessory

When you are done communicating with an accessory or if the accessory was detached, close the file descriptor that you opened by calling `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_ACCESSORY_DETACHED.equals(action)) {
            UsbAccessory accessory = (UsbAccessory)intent.getParcelableExtra(UsbManager.EXTRA_ACCESSORY);
            if (accessory != null) {
                // call your method that cleans up and closes communication with the accessory
            }
        }
    }
};
```

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.