Table of Contents

Background

Android is expected to proliferate across a wide variety of devices in a relatively short time. Not just smart phones and tablets but devices like laptops, gaming consoles, embedded devices…It has undergone 8 releases in 2 years, more than 310 android devices of all shapes and sizes and 100 million activated android devices. Even with so much growth, when compared, general computing devices support a lot more hardware than android. Android was built with Mobile Internet Devices (MID) in mind, but newer use cases demand support for other devices. Hence there is a growing need for android to support other hardware devices of different shapes and sizes.

Existing Support

Android Open Accessory Development Kit:

The Android 3.1 platform (also backported to Android 2.3.4) introduces Android Open Accessory support, which allows external USB hardware (an Android USB accessory) to interact with an Android-powered device in a special "accessory" mode. When an Android-powered powered device is in accessory mode, the connected accessory acts as the USB host (powers the bus and enumerates devices) and the Android-powered device acts as the USB device. Android USB accessories are specifically designed to attach to Android-powered devices and adhere to a simple protocol (Android accessory protocol) that allows them to detect Android-powered devices that support accessory mode. Accessories must also provide 500mA at 5V for charging power. Many previously released Android-powered devices are only capable of acting as a USB device and cannot initiate connections with external USB devices. Android Open Accessory support overcomes this limitation and allows you to build accessories that can interact with an assortment of Android-powered devices by allowing the accessory to initiate the connection.

USB Host and Accessory Support:

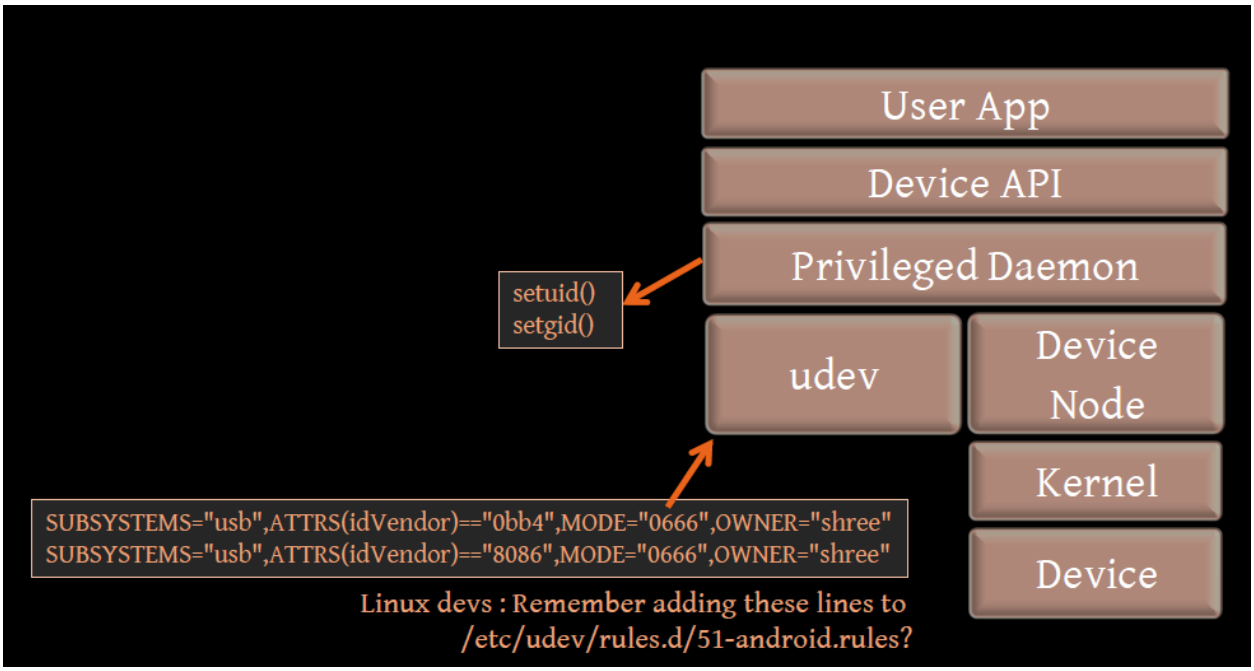
USB accessory mode allows users to connect USB host hardware specifically designed for Android-powered devices. The accessories must adhere to the Android accessory protocol outlined in the Android Accessory Development Kit documentation. This allows Android-powered devices that cannot act as a USB host 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.

Overview

We first describe a generic device support for a standard linux stack followed by a layered approach for android. Then we provide two examples, integrating a joystick and an industrial barcode scanner.

In the end we provide a device makers perspective and provide market compatibility.

Device support in Linux

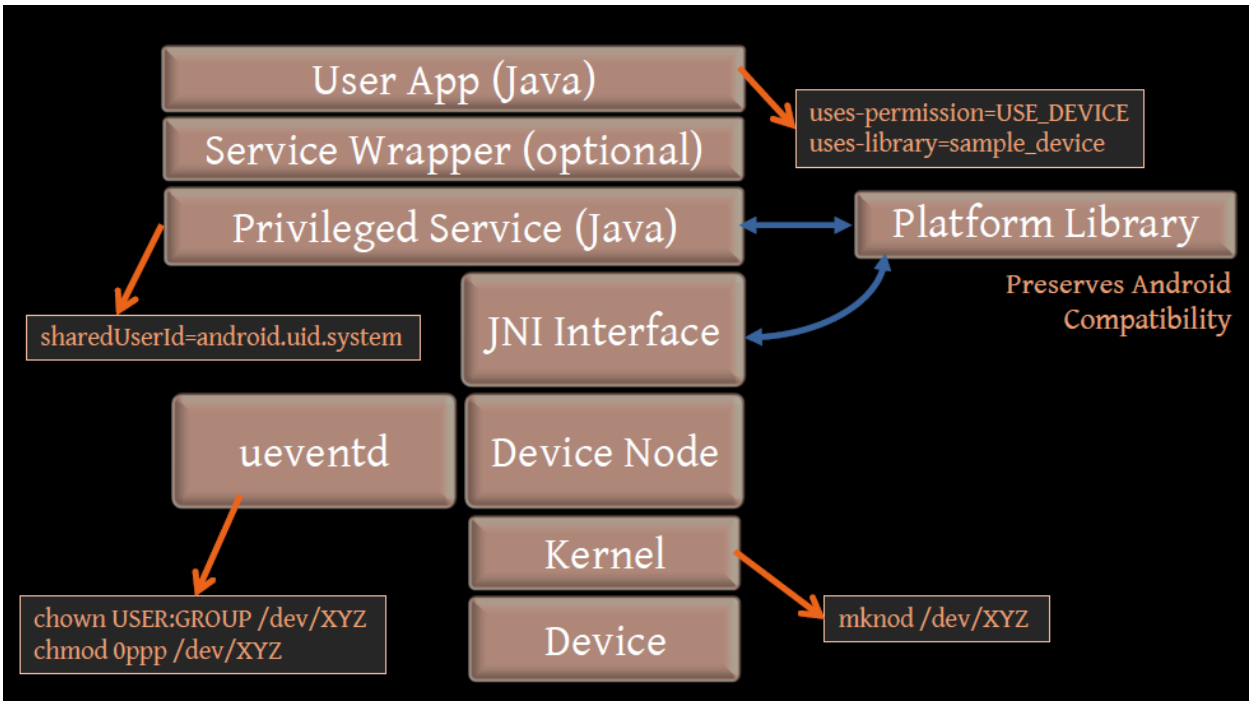


The above diagram represents a standard linux stack. At the bottom, we have the hardware with the kernel directly interacting with it. Kernel provides access to this hardware through file-like Device Nodes. Each node points to a part of the system (a device), which may or may not exist. Userspace applications can use these device nodes to interface with the system's hardware, for example, the X server listens to /dev/input/mice to relate the user's mouse movements to moving the visual mouse pointer. In order to create and name /dev device nodes corresponding to devices that are present in the system, udev relies on matching information provided by "sysfs" with rules provided by user. The rules are read from the files located at /etc/udev/rules.d/ or at the location specified by the udev_rules in the /etc/udev/udev.conf file. The access rights and privileges to these device nodes are maintained by these rules.

The "linux input subsystem" is the part of the linux kernel that manages the various input devices(such as keyboards, mice, joysticks, tablets…) that a user uses to interact with the kernel, command line and the graphical user interface. This subsystem is included in the kernel because these devices usually are accessed through special hardware interfaces (such as serial ports, PS/2 ports, usb…), which are protected and managed by the linux kernel. The kernel then exposes the user input in a consistent, device-independent way to the user space through a range of defined APIs. Applications in the userspace utilize these device APIs and performs corresponding actions according to its program logic.

Thus, to support a device in linux, one should first provide a device driver for the kernel to communicate with the hardware. Then provide appropriate rules to the device along with APIs corresponding to the required features.

Android Approach



The above diagram represents a standard android stack. Although android uses, the modified linux kernel, the architecture of the system itself is different from linux as android's architecture was specifically designed for MID devices.

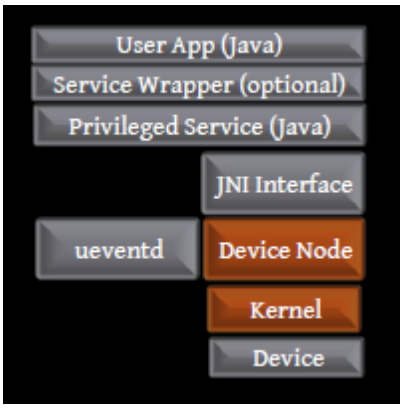
Like linux, the kernel communicates directly with the devices, and represents them as device nodes. Instead of udev, android uses ueventd which is much more simplified version of udev. Above this, the stack is different from the linux stack explained above. Android uses Java for applications with its own JVM called Dalvik Virtual Machine. The standard system level components and libraries are written in C/C++. These are made accessible for java code through the JNI interface. Each device extends it support to top level applications using the android Services. To maintain uniformity and standardize the set of APIs provided by android, changing of an API is not permitted in android unlike linux. Hence new devices mean new APIs. To preserve compatibility, the support should be provided through Platform Libraries which is an android mechanism to ensure unnecessary modifications to the android APIs.

Therefore, to provide a support for new device in the android framework, one must, like linux, provide necessary drivers for the kernel to communicate with the device along with necessary permissions. Then provide an android service utilizing the JNI interface for the apps to use the device's features and a platform library.

Example 1: Supporting a USB Joystick

Here we provide steps to support Logitech's Precision Gamepad, a 10 key usb joystick, for android 2.2 (froyo). We chose this device because of its simplicity, and also for the author's interest in gaming. We use android-x86 for development as we wanted to present our demo on a laptop to showcase the support of android on other devices. We provide individual steps along the diagram of the android stack indicating which layer of the stack is modified at each step.

Step 1: Kernel Drivers & Device node



The Android input subsystem nominally consists of an event pipeline that traverses multiple layers of the system.

At the lowest layer, the physical input device produces signals that describe state changes such as key presses and touch contact points. The device firmware encodes and transmits these signals in some way such as by sending USB HID reports to the system or by producing interrupts on an I2C bus.

The signals are then decoded by a device driver in the Linux kernel. The Linux kernel provides drivers for many standard peripherals, particularly those that adhere to the HID protocol. However, an OEM must often provide custom drivers for embedded devices that are tightly integrated into the system at a low-level, such as touch screens.

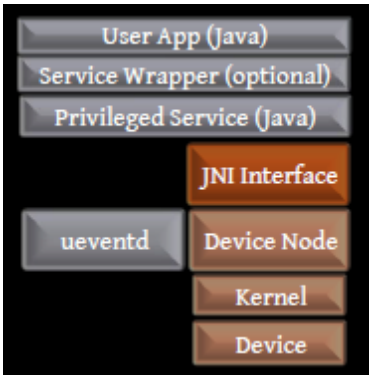
The input device drivers are responsible for translating device-specific signals into a standard input event format, by way of the Linux input protocol. The Linux input protocol defines a standard set of event types and codes in the linux/input.h kernel header file. In this way, components outside the kernel do not need to care about the details such as physical scan codes, HID usages, I2C messages, GPIO pins, and the like.

Since our joystick is based on usb, the driver defaults to HID raw driver already present in the kernel. This will be included in most of the kernels, if not, can be added to the kernel config file and compiled again. So all we have to do is boot up android, connect the joystick and check if the kernel is able to provide a device node for our joystick, which means that the kernel can successfully communicate with the device.

```
 $ cat /proc/bus/input/devices | grep ^[NH] | grep -A 1 Gamepad \
> | grep event
  N: Logitech Precision Gamepad
  H: Handlers=event15
```

The above diagram shows that we were able to see our joystick as an input device node in the system as /dev/input/event15

Step 2: Interface to the device node



Now that our joystick is shown up as a device node, we have to provide an interface to it. We use native code (C++) for this and then provide a JNI wrapper around it.

We provide three methods (functions/interfaces) for the joystick

1. Open – Check for device and open the device for reading if available.
2. GetKey – Get the scan code of the latest event (key press)
3. Close – Stop reading from the device and close it.

Open

```
fd = open("/dev/input/event15", O_RDONLY);
```

/proc/bus/input/devices maintains a list of all the input devices. To obtain the path for our joystick input event node, we have to do a basic search for our device's name and its corresponding event number. We search for the name "Gamepad" in the list and extract the corresponding event number. Then we can open the device in read only mode and read the "input_event" structure to get single key code per event.

GetKey

The input_event structure has 3 main attributes – "type", "value", "code" which are of interest to us.

- Type – used to differentiate between the absolute key(action keys) and the relative key (axis keys)
- Value – represents a key down (value = 1) or a key up (value = 0) for an absolute key. For relative keys, it represents low (value = 1), mid (value = 128) and high (value = 255).

- Code – for the absolute key, it gives the unique scan code. For the relative keys it represents x-axis (code = 0) and y-axis (code = 1)

```
struct input_event ev;
rd = read (fd, &ev, sizeof(struct input_event);
```

Therefore, if value is “1” we directly return the scan code of the key pressed else we have to check the combination of “code” and “value” to identify the x-axis and y-axis clicks.

Close

To stop and listening and close the device, we simply close the file descriptor of the device that was returned when we opened the file for reading.

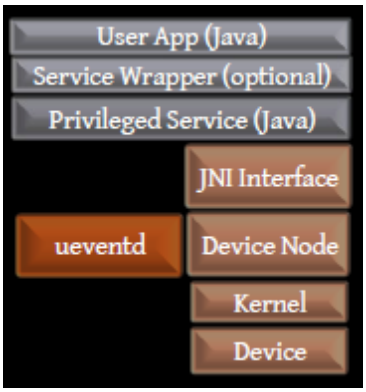
Since the android service that will utilize this interface will be written in Java, we have to provide a JNI wrapper around our native code.

```
static const JNINativeMethod gMethods[] = {
        {“open”, “()Z”, (void *)Java_Joystick_open},
        {“getKey”, “()I”, (void *)Java_Joystick_getKey},
        {“close”, “()V”, (void *)Java_Joystick_close},
};

Jint JNI_OnLoad(JavaVM* vm, void* reserved) {

        //register your methods
                ...
}
```

Next we have to generate a shared library of this. Our service will utilize this library to access our joystick methods.

```
LOCAL_MODULE := libsample_joystick
```

Step 3: Permissions



In android, /dev/input/ has root ownership. To access the files under /dev/input/ our service should be running with root privilege which is not acceptable. Hence we have to change the access privilege to a lower privilege than root, but not totally open to user applications. Therefore set the ownership to "system".

These changes can be made in the android source code in the file

<Android-src>/system/core/init/devices.c – android 2.2 (froyo)

<Android-src/system/core/rootdir/ueventd.rc – android 2.3 (gingerbread)

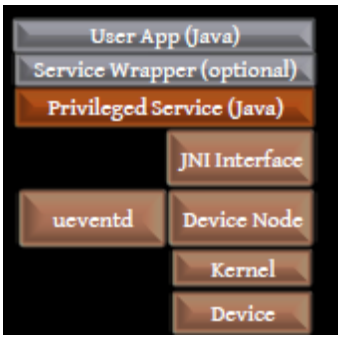
where <Android-src> corresponds to the root of the android source directory. The figure below depicts the changes we have made to the android source code.

```
Static struct perms_devperms[] = {
                ...
{"/dev/input", 0666, AID_SYSTEM, AID_INPUT, 1},
                ...
}
```

```
$ls -l /dev/input
crw-rw---- system input 13, 79 2011-11-18 14:05 event15
crw-rw---- system input 13, 79 2011-11-18 14:05 event11
crw-rw---- system input 13, 79 2011-11-18 14:05 event4
       ...
```

Once these permission changes are made, we have to build the android image from this source and use this image for our future steps.

Step 4: Services



We have to provide an android service that will allow the user applications to access the features of our joystick. Our service is designed as follows

- Allows multiple clients from different applications to access our service for IPC
- Initiated when a client binds to our service and is closed when no more clients are bound to our service
- Allows a client to register and remove a callback, which is called when a joystick event is occurred and passes the key code to the callback function.
- Is multithreaded – Spawns a worker thread for each callback

The obvious choice in our case is to use the "Android Interface Definition Language (AIDL)".

– .../ JoystickAPI.aidl

```
interface JoystickAPI {
        boolean setCallback(in JoystickCallback cb);
        boolean clearCallback();
}
```

– ... / JoystickService.java

```
Private JoystickAPI.Stub api = new JoystickAPI.Stub(){
        public boolean setCallback(JoystickCallback cb){
        // enable setting callback
        }
        public boolean clearCallback(){
        // enable clearing callback
        }
}
```

We create a "JoystickAPI.aidl" interface that enables user applications to set callback and clear callback. We also declare a "JoystickCallback" interface that defines the "onKeyPress(key)" method which will be called when a joystick key is pressed. The above figure shows our service implementing our aidl interface.

In order for our service to use the “open()”, “getKeyCode()” and “close()” function, we have to use the shared library that we generated in step 2. We add this line in our manifest file.

<uses-library android:name=”com.sample.hardware.joystick” />

Since our service requires restricted permissions it has to signed by a private key. We use the “platform” certificate for our needs.

```
# we ask for restricted permissions for our service,
# so the apk has to be signed
LOCAL_CERTIFICATE := platform
```

In the previous step, we changed the ownership of /dev/input/ files to “system”. This requires that our service must also be running under “system” privileges. To achieve this, we use the android’s concept of “Shared User Id “. In the manifest file of our service, we assign our shared user id to “system”

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.sample.android.service" android:sharedUserId="android.uid.system">

We also define our permission in the AndroidManifestfile.xml so that the apps must use this permission to access our service.

```
<permission android:name = “com.sample.USE_JOYSTICK” />
```

Our service is organized as shown below

```
|
+---com
|   \---sample
|       +---hardware
|       |   \---joystick
|       |           JoystickAPI.aidl
|       |           JoystickCallback.aidl
|       |           Joystick.java
|       |           Joystick.class
|       |
|       \---service
|               JoystickService.java
|
+---jni
|       Android.mk
|       Joystick.h
|       Joystick.cpp
|       Jsfunctions.h
|       Jsfunctions.cpp
```

Step 5: Platform Library

To declare our library to the framework, you must place a file with a .xml extension in the <android-src>/system/etc/permissions directory with the contents as shown below.

- Declare your library to the framework
  - /system/etc/permissions
  - com.sample.hardware.joystick.xml

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
    <library name="com.sample.hardware.joystick"
        file="/system/framework/com.sample.hardware.joystick.jar"/>
</permissions>
```

The top-level Android.mk file in our source defines the rules to build the shard library itself, whose target is “com.sample.hardware.joystick”. The code for this library is under <our-joystick-source>/com/sample/. The library is a raw .jar file, not and .apk which means that there is no manifest file or resources associated with our library.

[note: since our lib is not a .apk, if we need any resources for the library, such as drawables or layout files, we have to add them to the core framework resources under frameworks/base/res]

Application programmers can now include this .jar file in their app’s user libraries to access our service.

Step 6: Installing our service & pushing our library

Now the framework for our joystick is ready. All that is remaining is to install our service and our libraries to our custom android that we built earlier.

adb push <android-src>/out/target/product/generic_x86/system/framework/ \
com.sample.hardware.joystick.jar /system/framework

adb install -r <android-src>/out/target/product/generic_x86/system/app/JoystickService.apk

adb push <android-src>/out/target/product/generic_x86/system/lib/libsample_joystick.so /system/lib

At this point, we have successfully provided support for our joystick in our custom android we built from our modified source … *mazel tov!!*

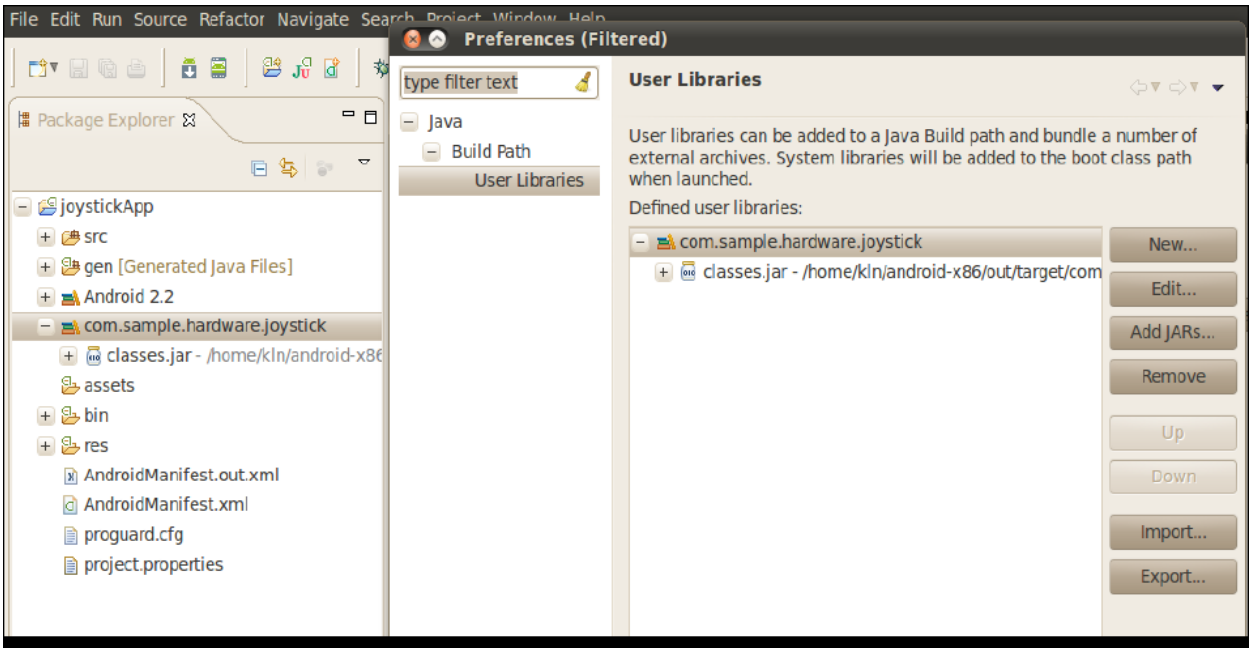
Step 7: End User Application

Finally we provide a sample application that uses our services to access the device and perform some action.

Application Settings

When we begin our application development, the standard android SDK that we use does not have our joystick service. Our joystick service is available as a .jar library that we created in step 5. Hence, in our android application, we have to include a user library with our com.sample.hardware.joystick.jar

The figure below shows our sample application using our com.sample.hardware.joystick.jar



In step 5, we also declared our own “joystick” permission. Therefore, for our app to use the service, we have to use this permission. We need to declare it in the app’s manifest file

<uses-permission android:name=”com.sample.android.permission.USE_JOYSTICK”/>

Application Development

Our sample app is a simple android application with the following features

1. Shows a visual representation of the our joystick buttons on the screen

2. Binds to our service and registers a callback
3. When user clicks a button on the joystick, the corresponding button on the screen glows

The first step of the application is to create our service intent and bind to it.

```
Intent intent = new Intent("com.sample.android.service.JoystickService");
bindService(intent, serviceConnection, Service.BIND_AUTO_CREATE);
```

When the connection to the service is established successfully, the app instantiates our "api" and registers a callback.

```
ServiceConnection sc = new ServiceConnection(){
        private void onServiceConnected(ComponentName n, Ibinder service){
                api = Ijoystick.stub.asInterface(service);
                try{
                        api.setCallback(jsCallback);

                }catch (RemoteException e){
                }
        }
}
```

Every time the joystick event occurs, our service calls the JoystickCallback.onKeyPress() function. In this app, this function updates the UI based on the key code sent as an input parameter. I.e. the button corresponding to the key code glows on the screen, indicating that the button was pressed.

```
    JoystickCallback jsCallback = new JoystickCallback.Stub() {
        private void onKeyPress(int arg0) throws RemoteException {
                final int key = arg0;
                runOnUiThread(new Runnable(){
                        public void run(){
                                updateUI(key);
                        }
                });
        }
    };
```

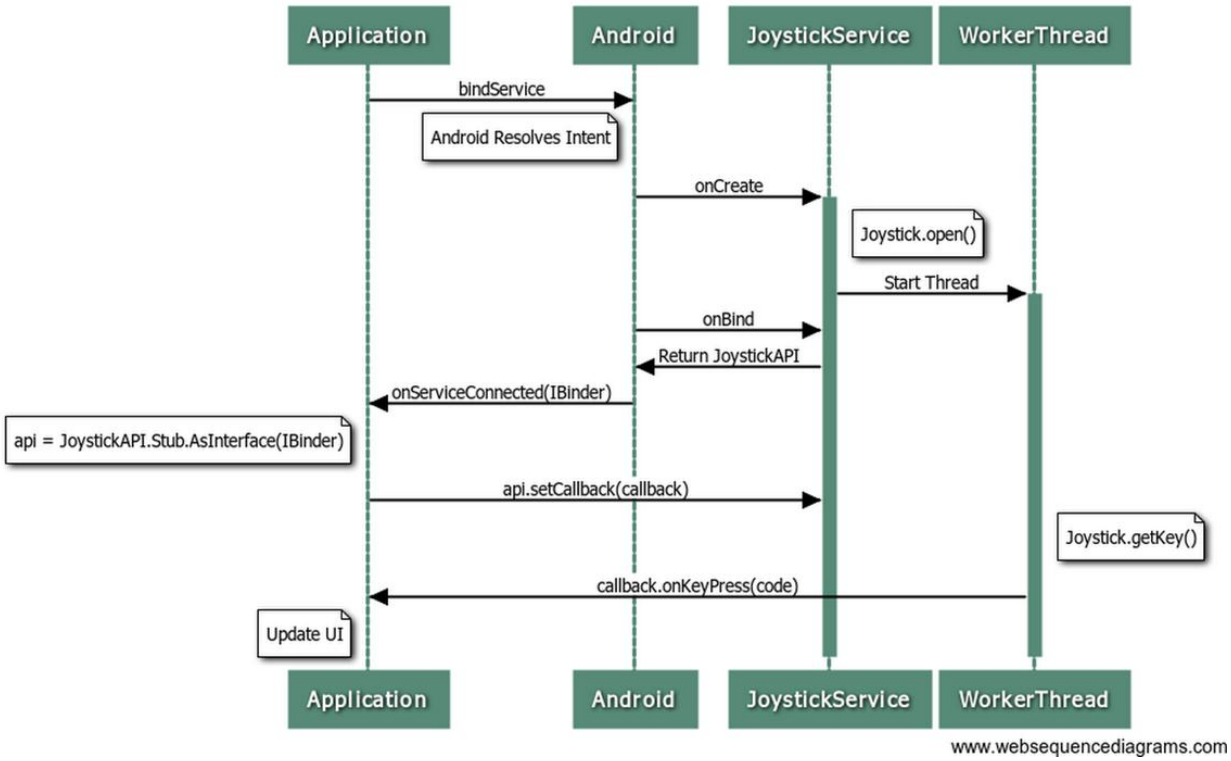
Our joystick service executes the callback function by the app in a separate worker thread. Since the android UI is not thread safe, any modifications to the UI in the callback must happen inside "runOnUiThread". The above figure showcases this scenario.

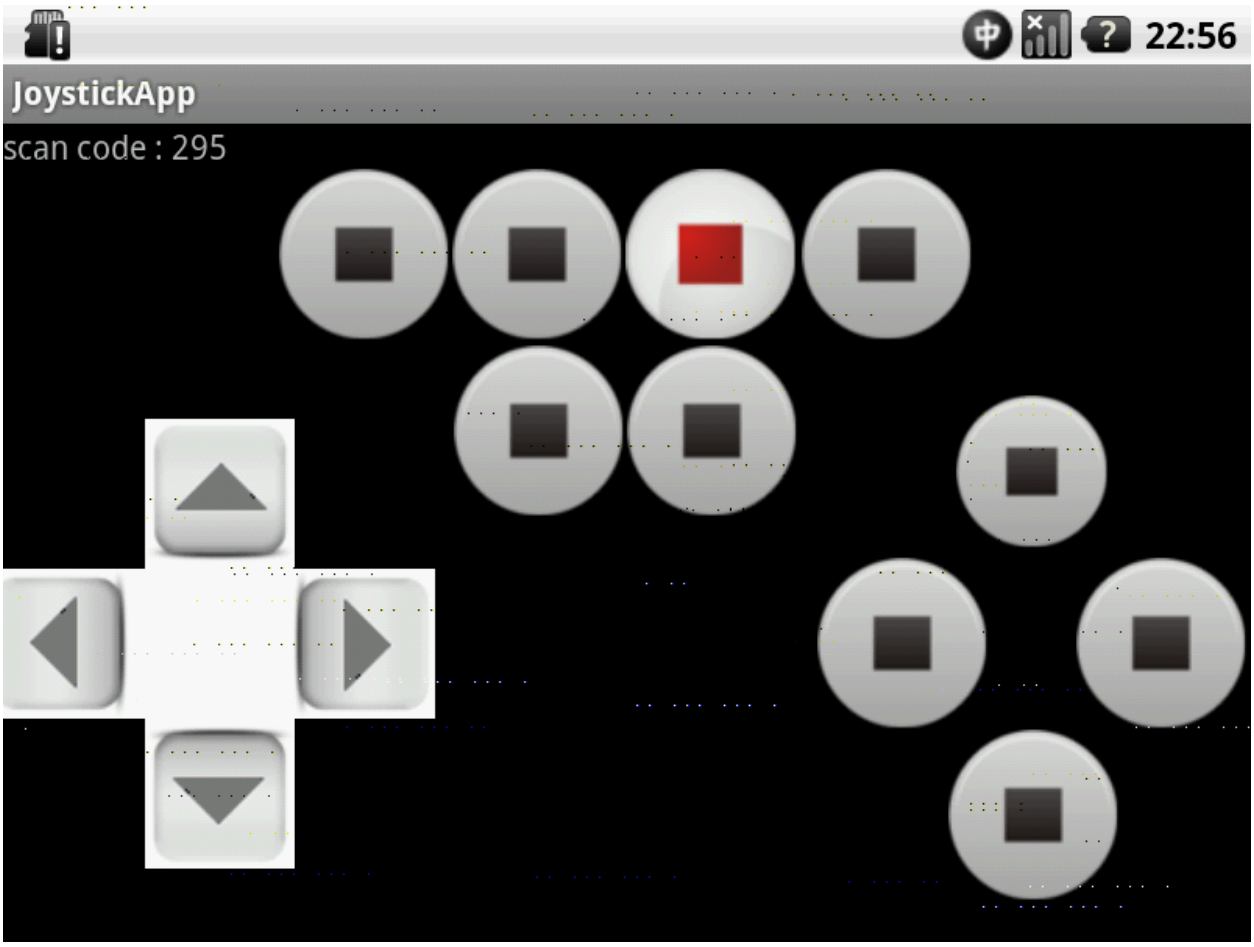
This way, an end user can develop any joystick based applications using our framework.

Overall:

To give the reader a complete picture of our framework, we provide a sequence diagram which shows the interaction of the application with the android system and our service.

We also have added a screenshot of our end use application.


Application
Android
JoystickService
WorkerThread
bindService
Android Resolves Intent
onCreate
Joystick.open()
Start Thread
onBind
Return JoystickAPI
onServiceConnected(IBinder)
api = JoystickAPI.Stub.AsInterface(IBinder)
api.setCallback(callback)
Joystick.getKey()
callback.onKeyPress(code)
Update UI
Application
Android
JoystickService
WorkerThread
www.websequencediagrams.com


22:56
JoystickApp
scan code : 295

Example 2: Support a Barcode Scanner

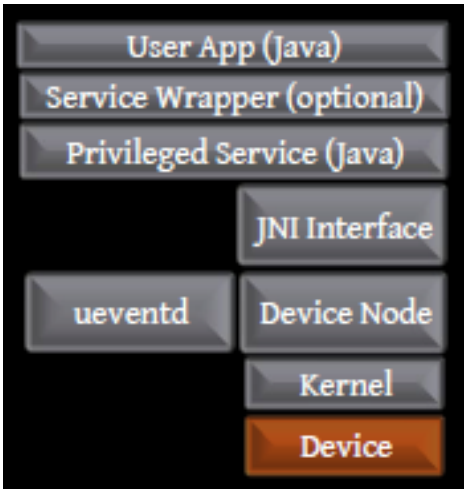
In the previous example we provided detailed descriptions of each of the steps involved in supporting a joystick on android. Since most of the steps to support barcode scanner remain the same as for joystick, we provide a brief description of each step. Readers can easily map the concepts explained in the previous example to steps we describe to support our barcode scanner.

The barcode we use is an "Opticon MDI 2300", 2D scanner. Few people may argue that we can use the camera itself to scan the bar code instead of a standalone barcode scanner. Although this is true, we provide the following reasons to use a standalone device

1. Supports tons of symbologies
2. Purpose made – fast, long cables, laser illumination
3. Available as a module for development
4. Would you drop your smart phone? The industrial barcode scanner is made to with stand such accidents ☺

The development environment is same as that of our joystick.

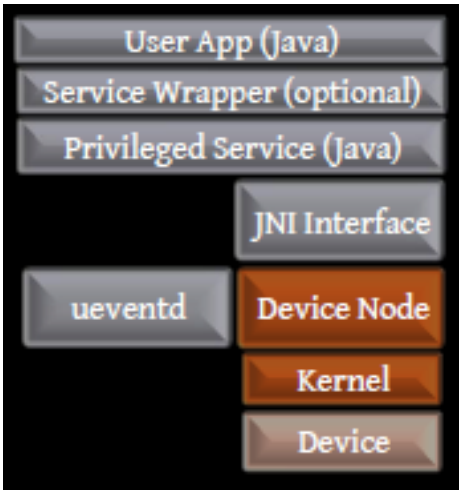
Step 1: Device setup



The first step is to configure our barcode scanner based on our needs. Our scanner can be configured by scanning appropriate barcodes itself or by sending commands manually through a serial port. We will use USB VCP (virtual COM port) mode

Our joystick did not have any configurable modes and hence we did not have any such steps for our joystick and moved directly to the kernel.

Step 2: Kernel Drivers



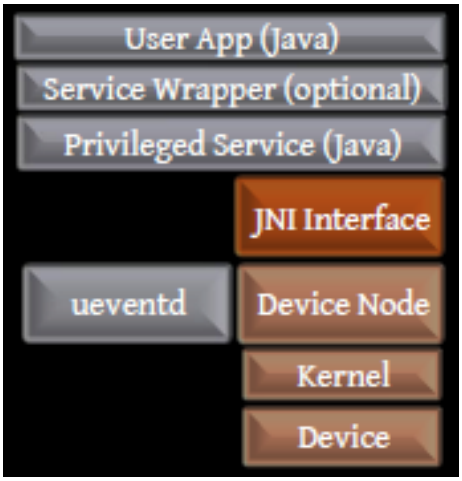
Since our scanner uses usb VCP mode, we have to use the CDC ACM driver. This device driver exposes the usb device as a virtual modem or a virtual COM port to the operation system. The specific steps depends on android/kernel version. Whatever be the steps for different versions, the kernel that is being used for development must include support for CDC ACM devices.

Ensure the support for protocol=”None” in cdc-acm.c

```
/* control interfaces without any protocol set */
        { USB_INTERFACE_INFO(USB_CLASS_COMM, USB_CDC_SUBCLASS_ACM,
                USB_CDC_PROTO_NONE) },
```

Boot with the new kernel and check if the barcode scanner shows up a device node. In our case the device showed up as /dev/ttyACM0

Step 3: Interface to the driver



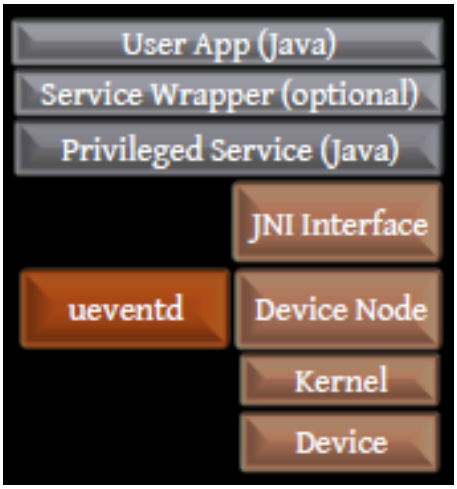
The codes scanned by the barcode scanner are decoded and sent directly to /dev/ttyACM0, single barcode per line.

Hence just doing

"cat /dev/ttyACM0"

would print out all the codes that is being decoded by the device. Since this is not linux, we have to do the same in native code and provide a JNI wrapper to it.

Step 4: Ueventd permissions



We need to change the owner ship of /dev/ttyACM0 from root to system, just like in the case of our joystick.

```
/dev/ttyACM0       0660       system     root
```

Add this line to : system/core/rootdir/ueventd.rc

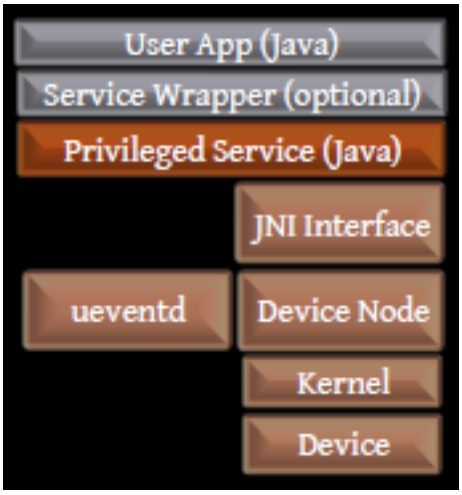
Instead, we could also use "com.sample.uid.acm" as owner.

- Passes Android Compatibility !

```
mSettings.addSharedUserLP("android.uid.system",
        Process.SYSTEM_UID, ApplicationInfo.FLAG_SYSTEM);
...
mSettings.addSharedUserLP("com.sample.uid.acm",
        1018, ApplicationInfo.FLAG_SYSTEM);
```

frameworks/base/services/java/com/android/server/PackageManagerService.java

Step 5: Services



The next step is to provide an android service that uses the jni interface to read data from the scanner. Like the case of joystick, we use the AIDL to define interfaces which will enable the user apps to register a callback with our service.

```
interface IBarcodeScanner {
        boolean setCallback(in BarcodeScannerCallback callback);
        boolean scanBarcode(int timeout);
        boolean clearCallback();
};

Interface BarcodeScannerCallback {
        void handlerBarcode(boolean timedout, String barcode);
}
```

Sign the service apk to gain privileged access

```
android:sharedUsedId="android.uid.system"
```

Step 6: Platform Libs & service installation

We then generate a platform library for our barcode scanner and install the service and the generated libraries to our android system. The procedures is same like that of our joystick, except for the namespace.

Step 7: End User App

Now that our system supports the barcode scanner, we write a simple application to test it. The app just binds to the service and registers a callback, which when a scanner event occurs, displays the decoded message on the screen.

The steps are the same here also

1. Bind to the service
2. Register a callback when successfully connected to the service.
3. Print the decoded message received in the callback method using the “runOnUiThread()” method
4. Unbind from service when exiting

The app uses the “barcode” permission and also includes “com.sample.hardware.barcode.jar” in the user libraries of the app to access our barcode service.

Summary

We have showcased a simple mechanism that will enable to the board/device manufacturers using android to provide support for new devices without having to extensively modify the android framework. We have kept the examples simple on purpose and skipped the “power management” and “deeper system integration” so that the readers can easily understand the concepts behind supporting a new device. Readers who want to implement this may want to rethink the interfaces and build on these steps to create a full-fledged support for similar devices on the android platform.

References

http://developer.android.com/index.html

http://www.android-x86.org/

http://www.android-x86.org/getsourcecode

http://stackoverflow.com/questions/2604727/how-can-i-connect-to-android-with-adb-over-tcp

http://developer.android.com/guide/developing/tools/aidl.html

http://source.android.com/index.html

http://source.android.com/tech/input/overview.html

Android applications on Intel architecture

Installing the android sdk for Intel Architecture

Android emulator for Intel architecture

Notes, system setup, issues & patches

We used the source code from the android-x86 repository. Our target device was a Dell Inspiron 1464 with Intel's core i3 processor and 3 GB ram multi-booted with Ubuntu 10.04, android-x86(froyo), fedora-15. Android on our target machine was allocated a space of 10 GB.

The development environment on the host machine – a pc running on 64 bit Ubuntu – eclipse IDE with android plugins, SDK, NDK and platform tools (revision 15).

The target machine and the host machine were connected through an Ethernet cable. The adb daemon on the target machine was made to listen on the tcp socket rather than usb. The steps are as follows

1. Connect the target and the host machine with Ethernet cable
2. On the target machine, do the following
    a. ifconfig eth0 192.168.1.1
    b. setprop service.adb.tcp.port 5555
    c. stop adbd
    d. start adbd
3. On the host machine, do the following
    a. ifconfig eth1 192.168.1.2
    b. adb connect 192.168.1.1:5555

Gingerbread cannot be built with 32 bit systems. Requires 64bit systems
We faced a few issues when we used GCC 4.6 with Ubuntu 11.10 and reverted back to GCC 4.5

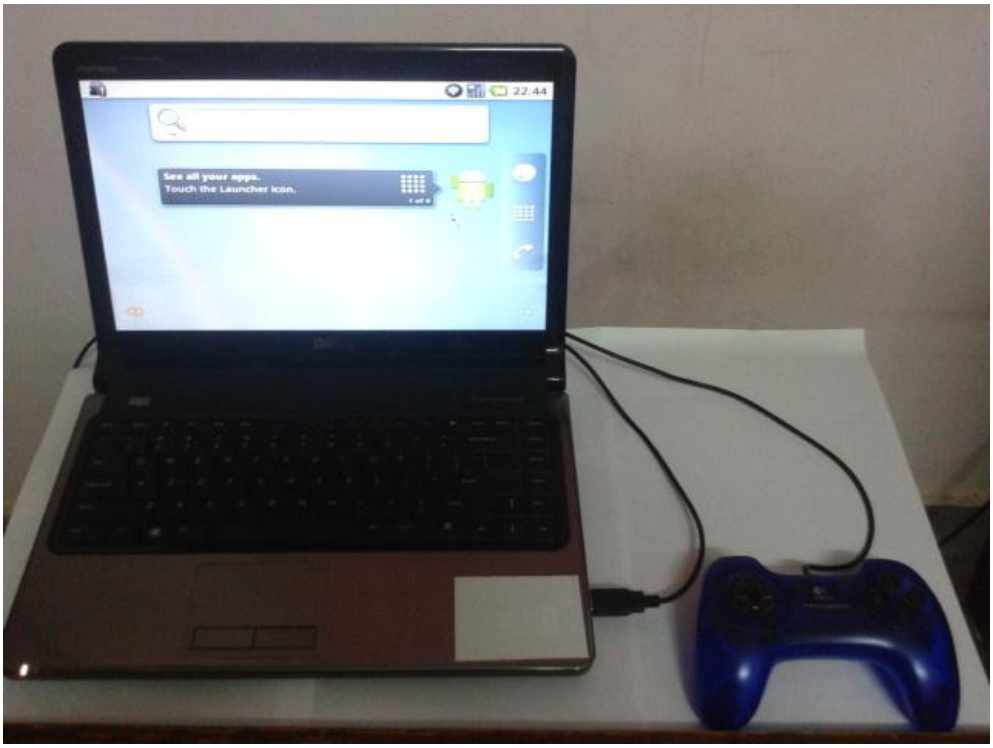
When we used the custom built images of android-x86 from source, we had display issues on the laptop i.e. the display would not come up when we booted android. The solution was to provide extra boot parameters to the kernel which enforces to use VESA driver for X and tell the kernel to not set the graphics resolution, let X do that instead. We modified the grub entries and provided these two parameters –"nomodeset" & "xforcevesa" to the kernel.

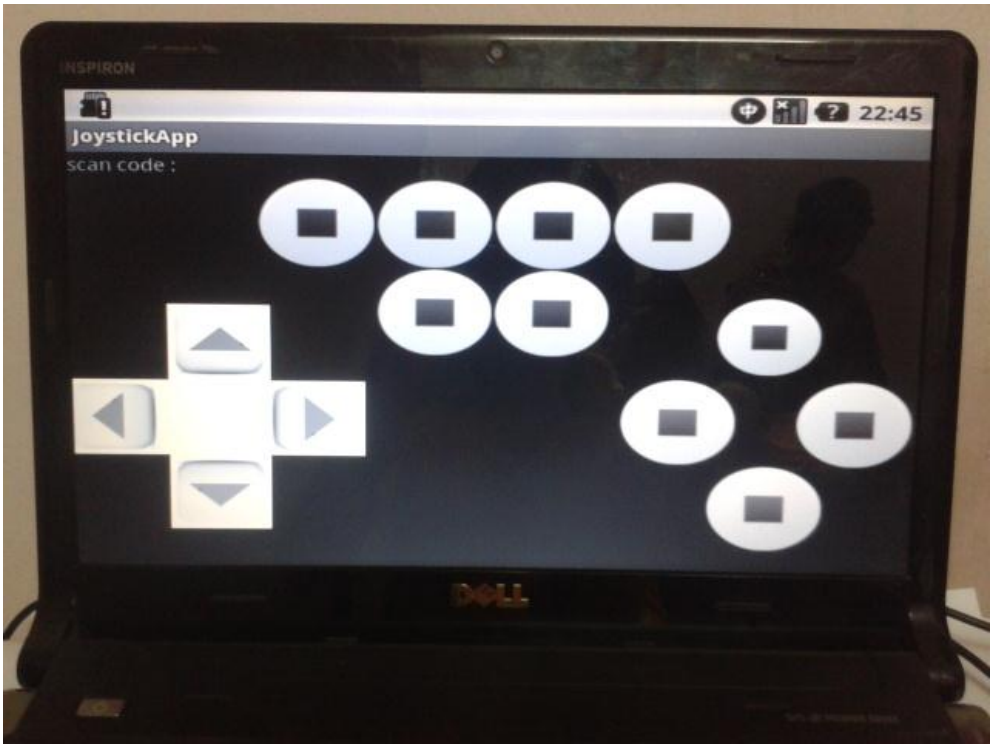
During the build process of the android-x86 2.2 source, we encountered a few errors corresponding to the "lpthread" shared libraries. The error was
"undefined reference to pthread_create, pthread_getspecific" for the file "<android-src>/system/core/libcutils/threads.c"

The patch for this is

```
+++ framework/base/tools/aapt/Android.mk
…
Ifeq($(HOST_OS), linux)
- LOCAL_LDLIBS += -lrt
+LOCAL_LDLIBS += -lrt –lpthread


+++ framework/base/tools/localize/Android.mk
…
Ifeq($(HOST_OS), linux)
- LOAL_LDLIBS += lrt
- LOCAL_LDLIBS += -lrt -lpthread
```

Screenshots





```
I: Bus=0011 Vendor=0002 Product=0008 Version=0000
N: Name="PS/2 Mouse"
P: Phys=isa0060/serio1/input1
S: Sysfs=/devices/platform/i8042/serio1/input/input6
U: Uniq=
H: Handlers=mouse0 event6
B: PROP=0
B: EV=7
B: KEY=70000 0 0 0 0 0 0 0
B: REL=3

I: Bus=0011 Vendor=0002 Product=0008 Version=7321
N: Name="AlpsPS/2 ALPS GlidePoint"
P: Phys=isa0060/serio1/input0
S: Sysfs=/devices/platform/i8042/serio1/input/input7
U: Uniq=
H: Handlers=mouse1 event7
B: PROP=0
B: EV=b
B: KEY=420 0 70000 0 0 0 0 0 0 0
B: ABS=1000003

I: Bus=0003 Vendor=046d Product=c21a Version=0110
N: Name="Logitech Logitech(R) Precision(TM) Gamepad"
P: Phys=usb-0000:00:1d.0-1.1/input0
S: Sysfs=/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.1/2-1.1:1.0/input/input8
U: Uniq=
H: Handlers=event8
B: PROP=0
B: EV=1b
B: KEY=3ff 0 0 0 0 0 0 0 0 0
B: ABS=3
B: MSC=10

I: Bus=0003 Vendor=413c Product=8161 Version=0111
:_
```

```
root@android:/ # dmesg | tail
<3>[ 1526.994072] cdc_acm 2-1.3:1.0: This device cannot do calls on its own. It is not a modem.
<6>[ 1526.994162] cdc_acm 2-1.3:1.0: ttyACM0: USB ACM device
<6>[ 1526.996724] scsi6 : usb-storage 2-1.3:1.3
<5>[ 1527.989162] scsi 6:0:0:0: Direct-Access     SAMSUNG  GT-S5830 Card    ffff PQ: 0 ANSI: 2
<5>[ 1527.989432] sd 6:0:0:0: Attached scsi generic sg2 type 0
<5>[ 1527.990791] sd 6:0:0:0: [sdb] Attached SCSI removable disk
<6>[ 1655.155103] usb 2-1.1: USB disconnect, address 3
<6>[ 1658.199821] usb 2-1.1: new low speed USB device using ehci_hcd and address 7
<6>[ 1658.319169] input: Logitech Logitech(R) Precision(TM) Gamepad as /devices/pci0000:00/0000:00:1
d.0/usb2/2-1/2-1.1/2-1.1:1.0/input/input16
<6>[ 1658.319219] generic-usb 0003:046D:C21A.0004: input: USB HID v1.10 Joystick [Logitech Logitech(
R) Precision(TM) Gamepad] on usb-0000:00:1d.0-1.1/input0
root@android:/ # _
```

```
root@android:/ # joystick_test
Device = '/dev/input/event8'

opening device 1
Waiting for key 0

key 250
Waiting for key 1

key 288
Waiting for key 2

key 250
Waiting for key 3

key 289
Waiting for key 4

key 251
Waiting for key 5

key 252
Waiting for key 6

key 290
Waiting for key 7

key 288
Waiting for key 8

key 291
Waiting for key 9

key 290
root@android:/ #
```

```
root@android:/ # ls -l /dev/input/
crw-rw---- system   input     13,  72 2011-12-23 23:03 event8
crw-rw---- system   input     13,  76 2011-12-23 22:35 event12
crw-rw---- system   input     13,  75 2011-12-23 22:35 event11
crw-rw---- system   input     13,  78 2011-12-23 22:35 event14
crw-rw---- system   input     13,  74 2011-12-23 22:35 event10
crw-rw---- system   input     13,  34 2011-12-23 22:35 mouse2
crw-rw---- system   input     13,  73 2011-12-23 22:35 event9
crw-rw---- system   input     13,  67 2011-12-23 22:35 event3
crw-rw---- system   input     13,  66 2011-12-23 22:35 event2
crw-rw---- system   input     13,  65 2011-12-23 22:35 event1
crw-rw---- system   input     13,  64 2011-12-23 22:35 event0
crw-rw---- system   input     13,  68 2011-12-23 22:35 event4
crw-rw---- system   input     13,  79 2011-12-23 22:35 event15
crw-rw---- system   input     13,  63 2011-12-23 22:35 mice
crw-rw---- system   input     13,  71 2011-12-23 22:35 event7
crw-rw---- system   input     13,  33 2011-12-23 22:35 mouse1
crw-rw---- system   input     13,  70 2011-12-23 22:35 event6
crw-rw---- system   input     13,  32 2011-12-23 22:35 mouse0
crw-rw---- system   input     13,  69 2011-12-23 22:35 event5
crw-rw---- system   input     13,  77 2011-12-23 22:35 event13
root@android:/ # _
```

```
system    1106  1     976    264   c12696e3 801110ee S /system/bin/servicemanager
root      1107  1     4344   556   ffffffff 802119c9 S /system/bin/vold
root      1108  1     4328   640   ffffffff 802119c9 S /system/bin/netd
radio     1109  1     4732   764   ffffffff 801119c9 S /system/bin/rild
root      1110  1     128104 28704 c1094825 80111284 S zygote
media     1112  1     29540  5768  ffffffff 802110ee S /system/bin/mediaserver
bluetooth 1113  1     1640   848   c1094825 8011248e S /system/bin/dbus-daemon
root      1114  1     1004   288   c12c28ec 80110d3e S /system/bin/installd
keystore  1115  1     2200   544   c127d250 80111e3b S /system/bin/keystore
system    1131  1110  235384 44008 ffffffff 801110ee S system_server
dhcp      1165  1     1060   292   c1094825 8001248e S /system/bin/dhcpcd
radio     1209  1110  182852 24984 ffffffff 801125d1 S com.android.phone
app_29    1215  1110  183036 29184 ffffffff 801125d1 S com.android.launcher
app_0     1245  1110  184872 29408 ffffffff 801125d1 S android.process.acore
app_6     1256  1110  166300 21816 ffffffff 801125d1 S com.andappstore.androidclient
app_2     1267  1110  169968 22912 ffffffff 801125d1 S com.android.email
app_15    1272  1110  166984 21772 ffffffff 801125d1 S android.process.media
app_8     1282  1110  166104 21744 ffffffff 801125d1 S com.android.deskclock
app_16    1294  1110  167136 21624 ffffffff 801125d1 S com.android.providers.calendar
app_23    1304  1110  166172 21488 ffffffff 801125d1 S com.android.bluetooth
app_35    1311  1110  179488 22236 ffffffff 801125d1 S com.android.mms
app_12    1329  1110  167348 21424 ffffffff 801125d1 S com.android.quicksearchbox
app_26    1338  1110  167064 21088 ffffffff 801125d1 S com.android.music
app_30    1344  1110  165364 20260 ffffffff 801125d1 S com.android.protips
app_18    1350  1110  167948 21936 ffffffff 801125d1 S com.cooliris.media
app_40    1473  1110  168072 22716 ffffffff 801125d1 S com.sample.SampleApp
root      1544  1     3500   236   ffffffff 0805239c S /sbin/adbd
app_41    1769  1110  165488 21608 ffffffff 801125d1 S com.sample.barcodeapp
system    1775  1110  165128 19944 ffffffff 801125d1 S com.sample.service.barcode:remote
root      1797  1105  984    544   c100200e 80011c92 S sh
root      1879  2     0      0     c10431c9 00000000 S kworker/0:2
app_0     1901  1110  169296 22804 ffffffff 801125d1 S com.android.inputmethod.pinyin
root      1929  2     0      0     c11d7bd4 00000000 S scsi_eh_6
root      1930  2     0      0     c122949e 00000000 S usb-storage
system    2148  1110  166164 19904 ffffffff 801125d1 S com.sample.android.service:remote
root      2157  1797  1112   304   00000000 80110d3e R ps
root@android:/ # _
```