Android 多核心嵌入式多媒體系統設計與實作

Android Hardware abstraction layer

賴槿峰 (Chin-Feng Lai)
Assistant Professor, institute of CSIE, National Ilan University
Nov. 10th 2011
© 2011 MMN Lab. All Rights Reserved
Outline

• Hardware Abstraction Layer Introduction
• Hardware Native Development Kit
• Android Native Server
• LAB : Run Native Application on Android
• **Hardware Abstraction Layer Introduction**
• **Hardware Native Development Kit**
• **Android Native Server**
• **LAB : Run Native Application on Android**
Hardware Abstraction Layer Introduction

Java Native Interface
Hardware Abstraction Layer Introduction

- Mentioned in Google I/O, 2008
- HAL is used to separate:
  - Android framework
  - Linux kernel
- Define the hardware control interface
- Not having a standard format in Android development
Hardware Abstraction Layer Introduction

• what should we be concerned about user space and kernel space on android base on Linux kernel
  – general Linux operation system
    • Include standard lib.so
    • Dynamic library: *.so
    • Static Library: *.a
  – Android system
    • Legacy android Hardware Abstraction Layer
      – Define on <android_source>/hardware/libhardware_legacy
    • HAL Stub android Hardware Abstraction Layer
      – Define on <android_source>/hardware/libhardware
Hardware Abstraction Layer Introduction

- General Linux operation system
  - Use c code

- Kernel driver
  - Libc.so
  - System Call: open(/dev/fb0, ioctl, ...)
  - Some 3rd library like ffmpeg, vlc, etc
  - Standard C library
    - System Call: open(/dev/fb0, ioctl, ...)

- User Space
  - Process for c
  - 3rd Library

- Kernel Space
  - Kernel driver
  - Hardware
  - Linux kernel

© 2011 MMN Lab. All Rights Reserved
Hardware Abstraction Layer Introduction

- Android System for legacy

Diagram:
- Process for c
  - Libc.so
    - 3rd Library
      - Kernel driver
        - Hardware
  - General Linux
- Process for java
  - Android Framework
    - Android SDK API
    - Dalvik virtual machine
      - 3rd Library
        - Libc.so
      - Hardware Abstraction Layer
        - JNI
          - Handle thread and process
          - Some library
            - Shared library module
          - *so
          - Legacy android HAL
- Android Application

MMN Lab.
© 2011 MMN Lab. All Rights Reserved
Hardware Abstraction Layer Introduction

- Android System for HAL Stub

**Diagram:**
- **Android Framework**
  - Process for java
  - Dalvik virtual machine
    - 3rd Library
    - Libc.so
    - Hardware Abstraction Layer stub
    - Native service
    - Runtime service
  - Hardware
- **Android Application**
- **Android SDK API**
- **JNI**
- **Native service**
- **Shared library module**
- ***.so**
- **Handle thread and process**

**Note:**
- Some library
- Android Application
- Android SDK API
- Java application
- Native service
- Runtime service
- Dalvik virtual machine
- 3rd Library
- Libc.so
- Hardware Abstraction Layer stub
- Native service
- Runtime service
- Hardware
- Android Framework
- Android Application
- Android SDK API
- JNI
- Native service
- Shared library module
- *.so
- Handle thread and process
Hardware Abstraction Layer Introduction

- Android Layer Analysis

<table>
<thead>
<tr>
<th>Layer</th>
<th>Language</th>
<th>Form</th>
<th>Ship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>JAVA</td>
<td>*.apk</td>
<td>*.apk/system.img</td>
</tr>
<tr>
<td>Framework</td>
<td>JAVA</td>
<td>*.jar</td>
<td>system.img</td>
</tr>
<tr>
<td>Libraries</td>
<td>C/C++</td>
<td>*.so</td>
<td>system.img</td>
</tr>
<tr>
<td>HAL</td>
<td>C/C++</td>
<td>*.so</td>
<td>system.img</td>
</tr>
<tr>
<td>Kernel</td>
<td>C/asm</td>
<td>*.ko</td>
<td>ulimage</td>
</tr>
</tbody>
</table>
Hardware Abstraction Layer Introduction

- **Legacy android Hardware Abstraction Layer**
  - Define on `<android_source>/hardware/libhardware_legacy`
  - The controlling hardware library compiler to *.so file which will be used as *shared library*
  - java call directly

- **HAL Stub android Hardware Abstraction Layer**
  - Define on `<android_source>/hardware/libhardware`
  - Use **HAL module** to direct function call
    - `libhardware/include`
      - Design interface, harader file
    - `libhardware/module`
      - Design reuse, override
    - `hardware.c`
      - Load(), hw_get_module()
    - `libhardware/include/hardware/hardware.h`
      - Some structure
Hardware Abstraction Layer Introduction

- Legacy android Hardware Abstraction Layer
  - many methods
Hardware Abstraction Layer Introduction

- **Example: Power Control In android**
  - There is a set of API for Power Management

### Summary

#### Nested Classes

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerManager.WakeLock</td>
</tr>
</tbody>
</table>

#### Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACQUIRE_CAUSES_WAKEUP</td>
<td>Normally wake locks don't actually wake the device, they just cause it to remain on once it's already on.</td>
</tr>
<tr>
<td>FULL_WAKE_LOCK</td>
<td>Wake lock that ensures that the screen and keyboard are on at full brightness.</td>
</tr>
<tr>
<td>ON_AFTER_RELEASE</td>
<td>When this wake lock is released, pause the user activity timer so the screen stays on for a little longer.</td>
</tr>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
<td>Wake lock that ensures that the CPU is running.</td>
</tr>
<tr>
<td>SCREEN_BRIGHT_WAKE_LOCK</td>
<td>Wake lock that ensures that the screen is on at full brightness; the keyboard backlight will be allowed to go off.</td>
</tr>
<tr>
<td>SCREEN_DIM_WAKE_LOCK</td>
<td>Wake lock that ensures that the screen is on (but may be dimmed); the keyboard backlight will be allowed to go off.</td>
</tr>
</tbody>
</table>

#### Public Methods

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>goToSleep</td>
<td>Force the device to go to sleep.</td>
</tr>
<tr>
<td>boolean</td>
<td>isScreenOn</td>
<td>Returns whether the screen is currently on.</td>
</tr>
<tr>
<td>PowerManager.WakeLock</td>
<td>newWakeLock(int flags, String tag)</td>
<td>Get a wake lock at the level of the flags parameter.</td>
</tr>
</tbody>
</table>
Hardware Abstraction Layer Introduction

- Example: Power Control In android
- Using the Power Management API
  - In eclipse

```java
mButtonw01.setOnClickListener(new Button.OnClickListener()
{
    public void onClick(View arg0)
    {
        mPowerManager=(PowerManager)getSystemService(Context.POWER_SERVICE);
        mPowerManager.goToSleep(5000);

        text.setText("diousk sleep!");
    }
});
```
Hardware Abstraction Layer Introduction

- Example: Power Control In *android framework*
- Trace the power management code:
  - Android/frameworks/base/core/java/android/os/PowerManager.java

```java
public void goToSleep(long time) {
    try {
        mService.goToSleep(time);
    } catch (RemoteException e) {
    }
}
```
Hardware Abstraction Layer Introduction

- Example: Power Control In *android runtime service*
- Trace the power management code:
  - Android/frameworks/base/services/java/com/android/server/PowerManagerService.java

```
private void updateNativePowerStateLocked() {
    nativeSetPowerState(
        (mPowerState & SCREEN_ON_BIT) != 0,
        (mPowerState & SCREEN_BRIGHT) == SCREEN_BRIGHT);
}
```
Hardware Abstraction Layer Introduction

• Example: Power Control In **JNI Table**

• Trace the power management code:
  – Android/frameworks/base/services/services/jni/com_android_server_PowerManagerService.cpp

```cpp
static JNINativeMethod gPowerManagerServiceMethods[] = {
  /* name, signature, funcPtr */
  { "nativeInit", "()V", (void*) android_server_PowerManagerService_nativeInit },
  { "nativeSetPowerState", "(ZZ)V", (void*) android_server_PowerManagerService_nativeSetPowerState },
  { "nativeStartSurfaceFlingerAnimation", "(I)V", (void*)
    android_server_PowerManagerService_nativeStartSurfaceFlingerAnimation },
};
```

```cpp
static void android_server_PowerManagerService_nativeSetPowerState(JNIEnv* env, jobject serviceObj, jboolean screenOn, jboolean screenBright) {
  set_screen_stage(on);
}
```

Android/hardware/libhardware_legacy/power/power.c(HAL)
Hardware Abstraction Layer Introduction

- Example: Power Control In **JNI Table**

<table>
<thead>
<tr>
<th>Java 类型</th>
<th>符号</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Z</td>
</tr>
<tr>
<td>Byte</td>
<td>B</td>
</tr>
<tr>
<td>Char</td>
<td>C</td>
</tr>
<tr>
<td>Short</td>
<td>S</td>
</tr>
<tr>
<td>Int</td>
<td>I</td>
</tr>
<tr>
<td>Long</td>
<td>L</td>
</tr>
<tr>
<td>Float</td>
<td>F</td>
</tr>
<tr>
<td>Double</td>
<td>D</td>
</tr>
<tr>
<td>Void</td>
<td>V</td>
</tr>
</tbody>
</table>
Hardware Abstraction Layer Introduction

- Example: Power Control In android (Legacy_HAL)

```c
<in power.c>
const char * const NEW_PATHS[] = { "/sys/power/wake_lock", "/sys/power/wake_unlock", "/sys/power/state" };

int set_screen_state(int on) {
  ...
  len = write(g_fds[REQUEST_STATE], buf, len);
  if(len < 0) {
    LOGE("Failed setting last user activity: g_error=%d\n", g_error);
  }
  ...
}
```

Android/hardware/libhardware_legacy/power/power.c(HAL)
Hardware Abstraction Layer Introduction

- Example: Power Control In android

[Diagram showing the relationship between Android Activity, Power Manager, Power Manager Service, Power Java Native Interface, Kernel Power Management Interface, and Kernel Power Management.]
Hardware Native Development Kit

- Use Android Native Development Kit
  - Type conversion from Java to C

<table>
<thead>
<tr>
<th>JAVA type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>jboolean</td>
<td>boolean</td>
</tr>
<tr>
<td>jint</td>
<td>int</td>
</tr>
<tr>
<td>jlong</td>
<td>long</td>
</tr>
<tr>
<td>jdouble</td>
<td>double</td>
</tr>
<tr>
<td>jfloat</td>
<td>float</td>
</tr>
<tr>
<td>jchar</td>
<td>char</td>
</tr>
<tr>
<td>Jstring</td>
<td>string</td>
</tr>
</tbody>
</table>

```java
static void android_server_PowerManagerService_nativeSetPowerState(JNIEnv* env, jobject serviceObj, jboolean screenOn, jboolean screenBright) {
    set_screen_stage(on);
}
```
Hardware Abstraction Layer Introduction

- HAL Stub android Hardware Abstraction Layer
  - Stub provide operations and callbacks of hardware
  - Services use hardware module ID to get information and methods

```
HAL Stub android Hardware Abstraction Layer
  - Stub provide operations and callbacks of hardware
  - Services use hardware module ID to get information and methods
```

```
Hardware Abstraction Layer Introduction

- HAL Stub android Hardware Abstraction Layer
  - Stub provide operations and callbacks of hardware
  - Services use hardware module ID to get information and methods
```

```
Application
  └── Runtime service
    └── Native service Binding
      ├── Native service
      │   └── Native
      │       └── Native service
      │           └── Native service
      └── Native service
        └── Native service
          └── Native service
            └── Native
```

```
Kernel driver
  └── Hardware
```

```
Process for java
  └── Android Framework
    └── Dalvik virtual machine
      └── 3rd Library
        └── Libc.so
          └── Hardware Abstraction Layer
              └── stub
```

```
MMN Lab.
© 2011 MMN Lab. All Rights Reserved
```
Hardware Abstraction Layer Introduction

- Each hardware must implement the interface in stub format for using in android service
- Stub format
  - defined in android/hardware/libhardware/include/hardware/hardware.h
    - hw_module_t
    - hw_module_method_t
    - hw_device_t
  - defined in android/hardware/libhardware/hardware.c
    - Load()
      - Use dlopen() to load *.so
    - hw_get_module()
      - Get module information and call load() function
Hardware Abstraction Layer Introduction

- Each hardware must implement the interface in stub format for using in android service

```c
typedef struct hw_module_t {
    uint32_t tag;
    uint16_t version_major;
    uint16_t version_minor;
    const char *id;
    const char *name;
    const char *author;
    /** Modules methods */
    struct hw_module_methods_t* methods;
} hw_module_t;
```

```c
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);
} hw_device_t;
```

```c
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
                struct hw_device_t** device);
} hw_module_methods_t;
```
Hardware Abstraction Layer Introduction

- HAL Stub android Hardware Abstraction Layer

SDK →.Runtime Service → JNI Table → Native service → HAL → HAL Stub

1. Init function
2. Get system service
3. Control method
4. Return status

JNI NativeMethod

Hw_get_module()
Hw_module_t
Control method()

Return status

dlopen
Hw_module_t
Control system call

Return status
Hardware Abstraction Layer Introduction

• runtime service
  – Java code
  – Define on:
    `<android_source>/framework/base/services/java/com/android/server`
  – Define “private static native”

```java
private native void nativeInit();
private native void nativeSetPowerState(boolean screenOn, boolean screenBright);
private native void nativeStartSurfaceFlingerAnimation(int mode);
```

• JNI Table

```java
static JNINativeMethod gPowerManagerServiceMethods[] = {
    /* name, signature, funcPtr */
    { "nativeInit", "()V", (void*) android_server_PowerManagerService_nativeInit },
    { "nativeSetPowerState", "(ZZ)V", (void*) android_server_PowerManagerService_nativeSetPowerState },
    { "nativeStartSurfaceFlingerAnimation", "(I)V", (void*)
      android_server_PowerManagerService_nativeStartSurfaceFlingerAnimation },
};
```

<android_source>/framework/base/services/jni
<android_source>/framework/base/core/jni
Hardware Abstraction Layer Introduction

- Native service

```c
static android_server_PowerManagerService_nativeInit(JNIEnv *env, jclass clazz)
{
    module_t const * module;
    hw_get_module(HARDWARE_MODULE_ID, (const hw_module_t**)&module);
    return 0;
}
```

- Hal Stub

```c
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    status = load(id, path, module);
    return status;
}
```
Hardware Abstraction Layer Introduction

Android Activity1

Power Manger
<Android_src>/frameworks/base/core/java/android/os/PowerManager.java

Power Manger Service
<Android_src>/frameworks/base/services/java/com/android/server/PowerManagerService.java

Power Java Native Interface
<Android_src>/frameworks/base/core/jni/android_os_Power.java

HAL
<Android_src>/hardware/libhardware/hardward.c

HAL Stub
<Android_src>/hardware/libhardware/module/xxxx.c

Linux kernel power management
<kernel_src>/arch/arm/mach-omap2/pm.c
• Hardware Abstraction Layer Introduction
• Hardware Native Development Kit
• Android Native Server
• LAB : Run Native Application on Android
Android Native Development Kit

- Android NDK is a companion tool to the Android SDK that lets you build performance-critical portions of your apps in native code.
Hardware Native Development Kit

- **NDK**
  - Generate JNI dynamic Libraries (*.so)
  - Put the file to correct path on android filesystem
- A set of tools and build files used to generate native code libraries from C and C++ sources
- way to embed the corresponding native libraries into an application package file
- The latest release of the NDK supports these ARM instruction sets
  - ARMv5TE
  - ARMv7-A
  - x86 instructions
Hardware Native Development Kit

- Download Android NDK

**Download the Android NDK**

The Android NDK is a companion tool to the Android SDK that lets you build performance-critical portions of your apps in native code to build applications, handle user input, use hardware sensors, access application resources, and more, when programming is still packaged into an .apk file and they still run inside of a virtual machine on the device. The fundamental Android application framework does not result in an automatic performance increase, but always increases application complexity. If you do not need the NDK, read [What is the NDK?](http://developer.android.com/sdk/ndk/index.html) for more information about what the NDK offers.

The NDK is designed for use only in conjunction with the Android SDK. If you have not already installed and set up the Android environment, you will need to do so before you can use the NDK.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Package</th>
<th>Size</th>
<th>MD5 Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>android-ndk-r5b-windows.zip</td>
<td>61299831 bytes</td>
<td>a305ab639399161ab4f684c</td>
</tr>
<tr>
<td>Mac OS X (intel)</td>
<td>android-ndk-r5b-darwin-x86.tar.bz2</td>
<td>50210863 bytes</td>
<td>019a14622a377b3727ec789af6707037</td>
</tr>
<tr>
<td>Linux 32/64-bit (x86)</td>
<td>android-ndk-r5b-linux-x86.tar.bz2</td>
<td>44138539 bytes</td>
<td>4b0045ddc2bf657be9d5177d0e0b7e7</td>
</tr>
</tbody>
</table>
Hardware Native Development Kit

- **NDK**
  - The NDK includes a set of cross-toolchains and sample application
  - Use Android.mk

```bash
.
|-- GNUmakefile
|-- README.TXT
|-- RELEASE.TXT
|-- build
|-- docs
|-- documentation.html
|-- ndk-build
|-- ndk-gdb
|-- ndk-stack
|-- platforms
|-- samples
|-- sources
|-- tests
|-- tmp
`-- toolchains
```
Hardware Native Development Kit

• Develop in NDK
  1. Set Env “PATH”, “NDK_ROOT”, “NDK_Sample”
  2. Edit your JNI code on android-ndk-r6/samples/hello-jni
  3. Edit Android.mk file on android-ndk-r6/samples/hello-jni
     • LOCAL_PATH
     • LOCAL_MODULE
     • LOCAL_SRC_FILES
     • LOCAL_LDLIBS
     • BUILD_SHARED_LIBRARY
  4. Host$ ndk-build

Gdbserver : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup : libs/armeabi/gdb.setup
Compile thumb : hello-jni <= hello-jni.c
SharedLibrary : libhello-jni.so
Install : libhello-jni.so => libs/armeabi/libhello-jni.so

5. Generate so file on “android-ndk-r6/samples/hello-jni/libs/armeabi”
Hardware Native Development Kit

• All parameters to 'ndk-build'
  – ndk-build
    • rebuild required machine code.
  – ndk-build clean
    • clean all generated binaries.
  – ndk-build V=1
    • launch build, displaying build commands.
  – ndk-build NDK_DEBUG=1
    • generate debuggable native code.

• On java code

```java
static {
    System.loadLibrary("hello-jni");
}
```
• Hardware Abstraction Layer Introduction
• Hardware Native Development Kit
• Android Native Server
• LAB : Run Native Application on Android
Android Native Server

• The android native service are defined in `<android_rootfs>/init.rc`

• init.rc includes the startup services which run in android background
  – ex: mediaserver、network daemon、bluetooth daemon、adb daemon…etc.
The init.rc file will be loaded by `<android_fs>/init` as filesystem mounted.

This is why the bootargs is like:

```plaintext
bootargs=console=ttyS2,115200n8,noinitrd,video=omapfb:mode:4.3inch_LCD,root=/dev/nfs,ip=192.168.0.10:192.168.0.209::eth0:off,init=/init rw rootwait mem=99M omap_vout.vid1_static_vrfb_alloc=y omap_vout.video1_numbuffers=3 omap_vout.video1_bufsize=829440 omap_vout.video2_numbuffers=0 nfsroot=192.168.0.209:/home/diousk/omap_tools/android_fs
```
How to add service in init.rc?

Follow Init.rc format
- Actions
- Commands
- Services
- Options
Android Native Server

• Actions
  – on <trigger>
  – <command>
  – <command>

• EX:
  – on boot
  – setprop persist.sys.keylayout gpio-keys
  – mkdir /data/misc/dhcp 0770 dhcp dhcp
  – chmod 0770 /data/misc/dhcp
Android Native Server

• Triggers
  – Boot
  – \<name\>=\<value\>
  – service-exited-\<name\>
  – device-added-\<path\>
  – device-removed-\<path\>
Android Native Server

• Commands
  – setprop <name> <value>
  – trigger <event>
  – chmod <octal-mode> <path>
  – export <name> <value>
  – class_start <serviceclass>
  – class_stop <serviceclass>
  – mkdir <path> [mode] [owner] [group]
  – start <service>
  – stop <service>
  – insmod <modules>
Android Native Server

• Services
  – service <name> <pathname> [ <argument> ]*
  – <option>
  – <option>

• EX:
  – service sgx /system/etc/init.sgx.sh
  – oneshot
Android Native Server

- Options
  - critical
  - user <username>
  - oneshot
  - class <name>
  - onrestart <command>
Android Native Server

- Actual init.rc in <android_fs>/init.rc

```bash
on init
sysclktz 0
loglevel 3

# setup the global environment
export PATH /sbin:/system/sbin:/system/bin:/system/xbin
export LD_LIBRARY_PATH /system/lib
export ANDROID_BOOTLOGO 1
export ANDROID_ROOT /system
export ANDROID_ASSETS /system/app
export ANDROID_DATA /data
export EXTERNAL_STORAGE /sdcard
export BOOTCLASSPATH /system/framework/core.jar:/system/framework/ext.jar:/system/framework/framework.jar:/system/framework/framework.android.policy.jar:/system/framework/services.jar

# Backward compatibility
symlink /system/etc /etc
symlink /sys/kernel/debug /d

# create mountpoints and mount tmpfs on sqlite_stat journals
insmod dvb-core.ko
insmod dib0070.ko
insmod dib3000mb.ko
insmod dib3000mc.ko
insmod dibx000_common.ko
insmod dib3000mc.ko
insmod dib7000m.ko
insmod dib7000p.ko
```
• Android Framework Review
• Hardware Abstraction Layer Introduction
• Power Control Example
• In Progress HAL
• Android Native Development Kit
  • Android Application Issue
  • Android Native Services (init.rc)
• LAB : Run Native Application on Android
Lab files

- *socket.tar.gz
  - Client.c
    - Reference to this C code and build as library via NDK
  - Server.c
    - Build as an executable binary and define in init.rc

- android_ndk.tar.gz
  - android-ndk-1.5_r1
Lab : Run Native Application on Android

- Use Android Native Development Kit
  - Write a C code to be built as library

```c
#include <jni.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

jstring
Java_mmn_com_tw_HelloJni_stringFromJNI( JNILEnv env, jobject thiz, jint prog)
{
    return (env)->NewStringUTF(env,"This is returned string");
}
```
Lab: Run Native Application on Android

• Use Android Native Development Kit
  – Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := helloqq    //library name
LOCAL_SRC_FILES := helloqq.c    //jni native code

include $(BUILD_SHARED_LIBRARY)
```

– Application.mk

```
APP_PROJECT_PATH := $(call my-dir)/project
APP_MODULES := helloqq    //和Android.mk對應
```
Lab: Run Native Application on Android

- Use Android Native Development Kit
  - Build the library

```
make APP=helloqq
Android NDK: Building for application 'helloqq'
Compile thumb : helloqq <= sources/samples/helloqq/helloqq.c
sources/samples/helloqq/helloqq.c: In function 'socket_client':
sources/samples/helloqq/helloqq.c:88: warning: function returns address of local variable
SharedLibrary : libhelloqq.so
Install        : libhelloqq.so => apps/helloqq/project/libs/armeabi
```

Put the library into `<android_fs>/system/lib/`
Lab: Run Native Application on Android

- In eclipse, File >> New >> Android Project
Lab : Run Native Application on Android

- Use Android Native Development Kit
  - Use in Apk source

```java
package w nec . com . tw ;

import android . util . Log ;

public class HelloJni extends Activity {
    @Override
    public void run () {
        super . run () ;
        stringFromJNI ( prog ) ; // long-time task
        Message m = new Message () ;
        m . what = UPDATE_SETTING_SUCCESS ;
        handler . sendMessage ( m ) ;
    }

    public native String stringFromJNI ( int prog ) ;
    public native String unimplementedStringFromJNI ( int prog ) ;

    static {
        System . loadLibrary ( "helloqq" ) ;
    }
}
```
Lab : Run Native Application on Android

- Click Right on android project >> Run as >> Android Application
  - Test Result : the text is returned from C code
Lab : Run Native Application on Android

- From `/home/mmn/workspace/mmn/bin`, you can find out the `mmn.apk`.
- Copy the apk file to `<android_fs>/system/app/`
- Or Copy the apk file to `android_fs/data/app/`
- Run your apk on devkit8000 and play video!
- Note: if the apk is updated, you have to restart booting devkit8000
Step1: Use NDK to build the native library for android

- Reference script

```bash
0. sudo tar -xzvf android_jni_dev.tgz
1. cd android-ndk-1.5_r1/sources/samples/
2. mkdir helloqq
3. cd helloqq
4. vim Android.mk
5. vim helloqq.c
6. vim android-ndk-1.5_r1/app/helloqq/Application.mk
7. cd android-ndk-1.5_r1/
8. make APP=helloqq  //APP="module name"
```
Step2: Put library into android filesystem

- Successful message

- Copy the library to android filesystem
  - `sudo cp app/helloqq/project/libs/armeabi/libhelloqq.so <android_fs>/system/lib/`
public class HelloTest extends Activity {

    /** Called when the activity is first created. */
    TextView tv;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        tv = (TextView) findViewById(R.id.myTextView1);
        tv.setText("123");
        tv.setText(stringFromJNI(0));
    }

    public native String stringFromJNI(int prog);

    static {
        System.loadLibrary("helloqq");
    }
}
Step4: Put the modified apk and socket-server into android filesystem

- After compile the apk source the apk will appear under `<apk_project>/bin/xxx.apk`
Step5: Run Native Application on Android

- Compile and put the socket-server to `<android_fs>/system/bin/`
- Write a native service in `<android_fs>/init.rc`
- Reboot devkit8000
- Run your apk to see whether the socket-server do something or not.
  (refer to system/etc/omap_android.sh)
附錄
• Introduction to Android
• Android Architecture
• Android Multimedia Framework
• Android Porting
• Android start-up programming
• LAB : Mount Android Filesystem
Android start-up program

- What happened during Android booting stage? (con’d)

From Korea Android Community - www.kandroid.org
Android start-up program

• What happened during Android booting stage?(con’d)
  • Kernel will execute “init” for starting android initialization
  • “init” will read the init.rc file to set up the environment variable or properties and start android services
• “init” is the first process after kernel started. The corresponding source code lies in <android_src>/system/core/init
Android start-up program

- Init.rc (under android_src/system/core/rootdir/init.rc)

```
# early-init
start ueventd

on init
sysclktz 0
loglevel 3

# setup the global environment
export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
export LD_LIBRARY_PATH /vendor/lib:/system/lib
export ANDROID_BOOTLOGO
export ANDROID_ROOT /system
export ANDROID_ASSETS /system/app
export ANDROID_DATA /data
export EXTERNAL_STORAGE /mnt/sdcard

# mount mtd partitions
# Mount /system rw first to give the filesystem a chance to save a checkpoint
mount yaffs2 mtd@system /system
mount yaffs2 mtd@system /system ro remount
mount yaffs2 mtd@userdata /data nosuid nodev
mount yaffs2 mtd@cache /cache nosuid nodev

# basic network init
ifup lo
hostname localhost
domainname localdomain
```
Android start-up program

• “init” does the following tasks step by step:
  • 1. Initialize log system.
  • 2. Parse /init.rc
  • 3. Execute **early-init action** parsed in step 2.

```c
int main(int argc, char **argv)
{
    INFO("reading config file\n");
    init_parse_config_file("/init.rc");

    /* pull the kernel commandline and ramdisk properties file in */
    import_kernel_cmdline(0);

    get_hardware_name(hardware, &revision);
    snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
    init_parse_config_file(tmp);

    action_for_each_trigger("early-init", action_add_queue_tail);
}
```
Android start-up program

• “init” does the following tasks step by step (con’d):
  – 4. Device specific initialize. For example, make all device node in /dev
  – 5. Initialize property system. Actually the property system is working as a share memory. Logically it looks like a registry under Windows system.
  – 6. Execute *init action* in the two files parsed in step 2.

```c
action_for_each_trigger("early-init", action_add_queue_tail);
queue_builtin_action(wait_for_coldboot_done_action, "wait_for_coldboot_done");
queue_builtin_action(property_init_action, "property_init");
queue_builtin_action(keychord_init_action, "keychord_init");
queue_builtin_action(console_init_action, "console_init");
queue_builtin_action(set_init_properties_action, "set_init_properties");

/* execute all the boot actions to get us started */
action_for_each_trigger("init", action_add_queue_tail);
action_for_each_trigger("early-fs", action_add_queue_tail);
action_for_each_trigger("fs", action_add_queue_tail);
action_for_each_trigger("post-fs", action_add_queue_tail);
```
Android start-up program

• “init” does the following tasks step by step (con’d):
  • 7. Start property service.
  • 8. Execute *early-boot and boot actions* in the two files parsed in step 2.
  • 10. Enter into an indefinite loop to wait for device/property set/child process exit events.

```c
/* execute all the boot actions to get us started */
action_for_each_trigger("early-boot", action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);
```
After init process, there are two main functions (Zygote, System server) in booting:

- Zygote does the following tasks step by step:
  1. Create JAVA VM.
  2. Register android native function for JAVA VM.
  3. Call the main function in the JAVA class named `com.android.internal.os.ZygoteInit`
     - Call `Zygote::forkSystemServer` (implemented in `dalvik/vm/native/dalvik_system_Zygote.c`) to fork a new process.
  4. Call `IPCThreadState::self()`->`joinThreadPool()` to enter into service dispatcher.
Android start-up program

- SystemServer will start a new thread to start all JAVA services as follows:
- **Core Services:**
  - 1. Starting Power Manager
  - 2. Creating Activity Manager
  - 3. Starting Telephony Registry
  - 4. Starting Package Manager
  - 5. Set Activity Manager Service as System Process
  - 6. Starting Context Manager
  - 7. Starting System Context Providers
  - 8. Starting Battery Service
  - 9. Starting Alarm Manager
  - 10. Starting Sensor Service
  - 11. Starting Window Manager
  - 12. Starting Bluetooth Service
  - 13. Starting Mount Service
Android start-up program

- Booting diagram

```
Kernel
  ↓
Init
  ↓
Daemons  Service Manager  Media Server
  ↓  ↓  ↓
DalvikVM  System Server  System Services
          ↓  ↓
GUI               Home Activity
```

© 2011 MMN Lab. All Rights Reserved
## Android start-up program

- **Service in android**

### Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Service</th>
<th>Service</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entropy Service</td>
<td>Power Manager</td>
<td>Activity Manager</td>
<td>Telephony Registry</td>
</tr>
<tr>
<td>Package Manager</td>
<td>Account Manager</td>
<td>Content Manager</td>
<td>Sys Cont. Provider</td>
</tr>
<tr>
<td>Battery Service</td>
<td>Hardware Service</td>
<td>Alarm Manager</td>
<td>Sensor Service</td>
</tr>
<tr>
<td>Window Manager</td>
<td>Bluetooth Service</td>
<td>Status Bar</td>
<td>Clipboard Service</td>
</tr>
<tr>
<td>Inp. Method Service</td>
<td>NetStat Service</td>
<td>Connectivity Service</td>
<td>Accessibility Manager</td>
</tr>
<tr>
<td>Notification Manager</td>
<td>Mount Service</td>
<td>Dev. Storage Monitor</td>
<td>Location Manager</td>
</tr>
<tr>
<td>Search Service</td>
<td>Checkin Service</td>
<td>Wallpaper Service</td>
<td>Audio Service</td>
</tr>
<tr>
<td>Headset Observer</td>
<td>Dock Observer</td>
<td>Backup Service</td>
<td>AppWidget Service</td>
</tr>
</tbody>
</table>

*Android start-up program*