Android 系统开发编译环境配置

主机系统：Ubuntu9.04
(1)安装如下软件包
sudo apt-get install git-core
sudo apt-get install gnupg
sudo apt-get install sun-java5-jdk
sudo apt-get install flex
sudo apt-get install bison
sudo apt-get install gperf
sudo apt-get install libsdl-dev
sudo apt-get install libesd0-dev
sudo apt-get install build-essential
sudo apt-get install zip
sudo apt-get install curl
sudo apt-get install libncurses5-dev
sudo apt-get install zlib1g-dev
android 编译对 java 的需求只支持 jdk5.0 低版本，jdk5.0 update 12 版本和 java 6 不支持。
(2)下载 repo 工具
curl http://android.git.kernel.org/repo >/bin/repo
chmod a+x /bin/repo
(3)创建源代码下载目录：
mkdir /work/android-froyo-r2
(4)用 repo 工具初始化一个版本(以 android2.2r2 为例)
cd /work/android-froyo-r2
repo init -u git://android.git.kernel.org/platform/manifest.git -b froyo
初始化过程中会显示相关的版本的 TAG 信息，同时会提示你输入用户名和邮箱地址，以上面的方式初始化的是 android2.2 froyo 的最新版本，
android2.2 本身也会有很多个版本，这可以从 TAG 信息中看出来，当前 froyo 的所有版本如下：
 * [new tag]   android-2.2.1_r1 -> android-2.2.1_r1
 * [new tag]   android-2.2_r1 -> android-2.2_r1
 * [new tag]   android-2.2_r1.1 -> android-2.2_r1.1
 * [new tag]   android-2.2_r1.2 -> android-2.2_r1.2
 * [new tag]   android-2.2_r1.3 -> android-2.2_r1.3

 * [new tag]   android-cts-2.2_r1 -> android-cts-2.2_r1
 * [new tag]   android-cts-2.2_r2 -> android-cts-2.2_r2
 * [new tag]   android-cts-2.2_r3 -> android-cts-2.2_r3
这样每次下载的都是最新的版本，当然我们也可以根据 TAG 信息下载某一特定的版本如下：
repo init -u git://android.git.kernel.org/platform/manifest.git -b android-cts-2.2_r3
(5)下载代码
repo sync
froyo 版本的代码大小超过 2G，漫长的下载过程。

(6)编译代码
cd /work/android-froyo-r2
make

Ubuntu 下使用 Simba 服务实现局域网内文件共享

Ubuntu 下安装 Simba 服务器将 linux 电脑上的内容共享，同一局域网内的另外一台 Windows PC 即可访问其共享内容，
从而实现 Windows 电脑向访问本地文件一样访问 Linux 文件系统的内容。

(1)安装 Simaba 服务器
sudo apt-get install samba

(2)安装 samba 图形化配置软件
sudo apt-get install system-config-samba

(3)创建一个 Simba 专用用户
从“系统”—“系统管理”—“用户和组”，来创建。如图，先点击“解锁”，然后“添加新用户”
然后输入新用户名字(如 Simba)和密码(如 111111)，然后在“高级”里面，选择“主组”为 sambashare 后点击"确定"即可
一句话来概括，就是创建一个主组为 sambashare 的用户

(4)配置 samba 共享
从“系统”—“系统管理—”samba“，运行配置界面
然后”首选项“—”服务器设置“。点击：安全性，在最后的”来宾帐号“里面，
选择我们新建立的那个用户 simba 后点击确定

(5)修改 samba 配置文件
打开/etc/samba/smb.conf，修改 valid users = XXXX 为 valid users = simba

(6)重启 samba 服务
sudo /etc/init.d/samba restart

(7)添加共享文件
从“系统”—“系统管理—”samba“，运行配置界面
点击"添加"来添加共享文件夹,点击"浏览"来选择需要共享的文件夹，选择"可擦写"和"显示"，点击"访问"可以设置访问权限，最好设置成"允许所有用户访问"

Ubuntu 下 tftp 服务器的创建

实验平台：Ubuntu9.04

(1)安装 tftp 服务
sudo apt-get install tftp tftpd openbsd-inetd

(2)在根目录下创建文件夹 tftpboot 文件夹并修改权限
cd /
sudo mkdir tftpboot
sudo chmod 777 tftpboot

(3)修改/etc/inetd.conf 文件如下：
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /tftpboot

(4)开启 tftp 服务
sudo /etc/init.d/openbsd-inetd reload
sudo in.tftpd -l /tftpboot

(5)重启电脑，然后将测试文件放入/tftpboot 目录下即可开始测试，出现问题可能一般都是权限问题 /tftpboot 目录下的文件访问权限改成 0777

本文来自 CSDN 博客，转载请标明出处：http://blog.csdn.net/jiajie961/archive/2010/11/05/5989328.aspx

创建一个新的 Android 产品项目

从 google 网站上下载的 android 源代码编译时默认是编译 google 设定的产品，如果要开发自己的产品，则需要重新定义一个产品项目，过程如下：
首先我们定义产品的规格，举例如下：
公司名称 ardent
产品名称 MTP
主板名称 merlin
然后安装下面的步骤新建产品项目：
(1)在源代码目录下创建一个用户目录
mkdir vendor

(2)在用户目录下创建一个公司目录
mkdir vendor/merlin

(3)在公司目录下创建一个 products 目录
mkdir vendor/merlin/products

(4)在上面创建的 products 下创建一个产品 makefile 文件 MTP.mk，内容如下：

```
PRODUCT_PACKAGES := \
    AlarmClock \
    Email \
    Fallback \
    Launcher2 \
    Music \
    Camera \
    Settings \
    LatinIME \
    NotePad \
    SoundRecorder \
    Bluetooth \
    CertInstaller \
    DeskClock

$(call inherit-product, $(SRC_TARGET_DIR)/product/core.mk)
#
# Overrides
PRODUCT_MANUFACTURER := ardent
PRODUCT_BRAND := ardent
PRODUCT_NAME := MTP
PRODUCT_MODEL := MultiMedia Touch Phone
PRODUCT_DEVICE := merlin
PRODUCT_LOCALES := zh_CN
```

产品 makefile 文件的编写有一套规则，详细情况见此文后面的补充内容。

(5)在 vendor/merlin/products 目录下创建一个 AndroidProducts.mk 文件，定义 Android 产品配置文件的路径，具体如下：

```
PRODUCT_MAKEFILES := \
  $(LOCAL_DIR)/MTP.mk
```

(6)在公司目录下创建一个主板目录

```
mkdir vendor/ardent/merlin
```

(7)在主板目录下新建一个主板配置文件 BoardConfig.mk，内容如下：

```
TARGET_NO_BOOTLOADER := true
TARGET_NO_KERNEL := true
TARGET_CPU_ABI := armeabi
BOARD_USES_GENERIC_AUDIO := true
USE_CAMERA_STUB := true
```

(8)如果你希望修改系统属性，则可以在主板目录下新建一个 system.prop 文件，该文件中可以修改系统属性，举例如下：

```
# system.prop for
# This overrides settings in the products/generic/system.prop file
```

```
#
# rild.libpath=/system/lib/libreference-ril.so
# rild.libargs=-d /dev/ttyS0
```

(9)在主板目录下建议一个 Android 的主板配置文件 AndroidBoard.mk，此文件是编译系统接口文件，内容如下：

```
# make file for new hardware  from
#
LOCAL_PATH := $(call my-dir)

#
# this is here to use the pre-built kernel
ifeq ($(TARGET_PREBUILT_KERNEL),)
TARGET_PREBUILT_KERNEL := $(LOCAL_PATH)/kernel
endif

file := $(INSTALLED_KERNEL_TARGET)
ALL_PREBUILT += $(file)
$(file): $(TARGET_PREBUILT_KERNEL) | $(ACP)
        $(transform-prebuilt-to-target)

#
# no boot loader, so we don't need any of that stuff..
#
LOCAL_PATH := vendor/ardent/merlin
include $(CLEAR_VARS)
#
# include more board specific stuff here? Such as Audio parameters.
#
```

(10)编译新的项目

```
. build/envsetup.sh
make PRODUCT-MTP-user
```

补充内容：

(1) 上面的新建的几个文件的编写可以参考 build/target/board/generic 目录下的 AndroidBoard.mk，BoardConfig.mk 和 system.prop

(2)产品 makefile 的编写规则，变量定义解释如下：

PRODUCT_NAME　终端用户可见的产品名称，对应到“Settings“中的“About the phone”信息

PRODUCT_MODEL　End-user-visible name for the end product

PRODUCT_LOCALES 1 个以空格分隔开的两个字母的语言码加上 2 字节的国家码的列表，影响到"Settings"中的语言，时间，日期和货币格式设置，

举例：en_GB de_DE es_ES fr_CA

PRODUCT_PACKAGES 需要安装的 APK 应用程序列表

PRODUCT_DEVICE　工作设计名称，即主板名称

PRODUCT_MANUFACTURER　生产厂家
PRODUCT_BRAND　软件设计针对的客户品牌
PRODUCT_PROPERTY_OVERRIDES　以"key=value"为格式的属性列表
PRODUCT_COPY_FILES 文件复制列表，格式为“原文件路径：目的文件路径”，编译过程中会按照此规则复制文件
PRODUCT_OTA_PUBLIC_KEYS 产品的 OTA 公共密匙列表
PRODUCT_POLICY　声明此产品使用的政策
PRODUCT_PACKAGE_OVERLAYS 指示是否使用默认资源或添加任何产品特定的资源，例如：vendor/acme/overlay
PRODUCT_CONTRIBUTORS_FILE HTML 文件中包含项目的贡献者
PRODUCT_TAGS 以空格分隔开的指定产品关键词列表

本文来自 CSDN 博客，转载请标明出处：ttp://blog.csdn.net/jiajie961/archive/2010/11/07/5993126.aspx

制作 ubifs 文件系统

1,安装相关的软件包
apt-get install liblzo2-dev

2,获取相关的工具 mkfs.ubifs 和 ubinize
这两个工具是制作 ubifs 文件系统的时候用到，它们是 mtd-utils 工具包中的内容，mtd-utils 工具包你可以从下面的网站下载和编译出来：
官方网站：http://www.linux-mtd.infradead.org/index.html
资源下载网站：http://git.infradead.org/

3,创建一个 create-ubifs.sh 脚本，主要是调用 mkfs.ubifs 和 ubinize 工具和相关参数来制作 ubifs 文件系统，内容如下：

```
#!/bin/bash

#############################################################
#    Script to generate ubifs filesystem image.    #
#############################################################

##### ubinize configuration file
config_file=rootfs_ubinize.cfg

##### Function to check result of the command
check_result() {
if [ $? -ne 0 ]
then
    echo "FAILED"
else
    echo "SUCCESSFUL"
```

```
fi
}

######  Function to check whether an application exists
check_program() {
for cmd in "$@"
do
        which ${cmd} > /dev/null 2>&1
        if [ $? -ne 0 ]
        then
                echo
                echo "Cannot find command \"${cmd}\""
                echo
                exit 1
        fi
done
}

if [ $# -ne 5 ]
then
    echo
            echo 'Usage: create-ubifs.sh [page_size_in_bytes] [pages_per_block] [partition_size_in_bytes]
[blocks_per_device] [path_to_rootfs]'
    echo
    exit
fi

page_size_in_bytes=$1
echo "Page size                                  [$page_size_in_bytes]bytes."
pages_per_block=$2
echo "Pages per block                              [$pages_per_block]"
partition_size_in_bytes=$3
echo "File-system partition size                     [$partition_size_in_bytes]bytes."
blocks_per_device=$4
echo "Blocks per device                            [$blocks_per_device]"
path_to_rootfs=$5

# wear_level_reserved_blocks is 1% of total blcoks per device
wear_level_reserved_blocks=`expr $blocks_per_device / 100`
echo "Reserved blocks for wear level                 [$wear_level_reserved_blocks]"

#logical_erase_block_size is physical erase block size minus 2 pages for UBI
logical_pages_per_block=`expr $pages_per_block - 2`
logical_erase_block_size=`expr $page_size_in_bytes \* $logical_pages_per_block`
echo "Logical erase block size                       [$logical_erase_block_size]bytes."
```

```
#Block size = page_size * pages_per_block
block_size=`expr $page_size_in_bytes \* $pages_per_block`
echo "Block size                                    [$block_size]bytes."

#physical blocks on a partition = partition size / block size
partition_physical_blocks=`expr $partition_size_in_bytes / $block_size`
echo "Physical blocks in a partition                [$partition_physical_blocks]"

#Logical blocks on a partition = physical blocks on a partitiion - reserved for wear level
patition_logical_blocks=`expr $partition_physical_blocks - $wear_level_reserved_blocks`
echo "Logical blocks in a partition                 [$patition_logical_blocks]"

#File-system volume = Logical blocks in a partition * Logical erase block size
fs_vol_size=`expr $patition_logical_blocks \* $logical_erase_block_size`
echo "File-system volume                            [$fs_vol_size]bytes."

echo
echo "Generating configuration file..."
echo "[rootfs-volume]"  > $config_file
echo "mode=ubi" >> $config_file
echo "image=rootfs_ubifs.img" >> $config_file
echo "vol_id=0" >> $config_file
echo "vol_size=$fs_vol_size" >> $config_file
echo "vol_type=dynamic" >> $config_file
echo "vol_name=system" >> $config_file
echo

# Note: Check necessary program for installation
#echo -n "Checking necessary program for installation....."
#check_program mkfs.ubifs ubinize
#echo "Done"

#Generate ubifs image
echo -n "Generating ubifs..."
./mkfs.ubifs -x lzo -m $page_size_in_bytes -e $logical_erase_block_size -c $patition_logical_blocks -o rootfs_ubifs.img -d $path_to_rootfs
check_result
echo -n "Generating ubi image out of the ubifs..."
./ubinize -o ubi.img -m $page_size_in_bytes -p $block_size -s $page_size_in_bytes $config_file -v
check_result

rm -f rootfs_ubifs.img
rm -f $config_file
```

(4)将 mkfs.ubifs 和 ubinize 以及 create-ubifs.sh 放置在同一目录下，然后调用 create-ubifs.sh 即可创建 ubifs 文件系统，create-ubifs.sh 用法如下：

create-ubifs.sh page_size_in_bytes(页大小) pages_per_block(每个扇区的页数量) partition_size_in_bytes(分区大小) blocks_per_device(扇区数量) path_to_rootfs(文件系统路径)
举例如下：
./create-ubifs.sh 2048 64 83886080 4096 ./rootfs
上面命令的意思是调用 create-ubifs.sh 将当前目录下的 rootfs 文件夹的内容制作成 ubifs 文件系统，nand flash 的页大小为 2k,每个扇区有 64 页，
总共有 4096 个扇区，要制作的文件系统的大小为 83886080 字节。

android 编译系统 makefile(Android.mk)写法

android 编译系统的 makefile 文件 Android.mk 写法如下

(1)Android.mk 文件首先需要指定 LOCAL_PATH 变量，用于查找源文件。由于一般情况下
Android.mk 和需要编译的源文件在同一目录下，所以定义成如下形式：
LOCAL_PATH:=$(call my-dir)
上面的语句的意思是将 LOCAL_PATH 变量定义成本文件所在目录路径。

(2)Android.mk 中可以定义多个编译模块，每个编译模块都是以 include $(CLEAR_VARS)开始
以 include $(BUILD_XXX)结束。
include $(CLEAR_VARS)
CLEAR_VARS 由编译系统提供，指定让 GNU MAKEFILE 为你清除除 LOCAL_PATH 以外的所有 LOCAL_XXX 变量，
如
LOCAL_MODULE，LOCAL_SRC_FILES，LOCAL_SHARED_LIBRARIES，LOCAL_STATIC_LIBRARIES 等。
include $(BUILD_STATIC_LIBRARY)表示编译成静态库
include $(BUILD_SHARED_LIBRARY)表示编译成动态库。
include $(BUILD_EXECUTABLE)表示编译成可执行程序

(3)举例如下(frameworks/base/libs/audioflinger/Android.mk)：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)  模块一
ifeq ($(AUDIO_POLICY_TEST),true)
  ENABLE_AUDIO_DUMP := true
endif
LOCAL_SRC_FILES:= \
    AudioHardwareGeneric.cpp \
    AudioHardwareStub.cpp \
    AudioHardwareInterface.cpp
ifeq ($(ENABLE_AUDIO_DUMP),true)
  LOCAL_SRC_FILES += AudioDumpInterface.cpp
```

```
  LOCAL_CFLAGS += -DENABLE_AUDIO_DUMP
endif
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libbinder \
    libmedia \
    libhardware_legacy
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
  LOCAL_CFLAGS += -DGENERIC_AUDIO
endif
LOCAL_MODULE:= libaudiointerface
ifeq ($(BOARD_HAVE_BLUETOOTH),true)
  LOCAL_SRC_FILES += A2dpAudioInterface.cpp
  LOCAL_SHARED_LIBRARIES += liba2dp
  LOCAL_CFLAGS += -DWITH_BLUETOOTH -DWITH_A2DP
  LOCAL_C_INCLUDES += $(call include-path-for, bluez)
endif
include $(BUILD_STATIC_LIBRARY)  模块一编译成静态库
include $(CLEAR_VARS)  模块二
LOCAL_SRC_FILES:=              \
    AudioPolicyManagerBase.cpp
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libmedia
ifeq ($(TARGET_SIMULATOR),true)
 LOCAL_LDLIBS += -ldl
else
 LOCAL_SHARED_LIBRARIES += libdl
endif
LOCAL_MODULE:= libaudiopolicybase
ifeq ($(BOARD_HAVE_BLUETOOTH),true)
  LOCAL_CFLAGS += -DWITH_A2DP
endif
ifeq ($(AUDIO_POLICY_TEST),true)
  LOCAL_CFLAGS += -DAUDIO_POLICY_TEST
endif
include $(BUILD_STATIC_LIBRARY) 模块二编译成静态库
include $(CLEAR_VARS) 模块三
LOCAL_SRC_FILES:=              \
    AudioFlinger.cpp           \
    AudioMixer.cpp.arm         \
    AudioResampler.cpp.arm     \
    AudioResamplerSinc.cpp.arm  \
    AudioResamplerCubic.cpp.arm \
```

```
    AudioPolicyService.cpp
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libbinder \
    libmedia \
    libhardware_legacy
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
  LOCAL_STATIC_LIBRARIES += libaudiointerface libaudiopolicybase
  LOCAL_CFLAGS += -DGENERIC_AUDIO
else
  LOCAL_SHARED_LIBRARIES += libaudio libaudiopolicy
endif
ifeq ($(TARGET_SIMULATOR),true)
 LOCAL_LDLIBS += -ldl
else
 LOCAL_SHARED_LIBRARIES += libdl
endif
LOCAL_MODULE:= libaudioflinger
ifeq ($(BOARD_HAVE_BLUETOOTH),true)
  LOCAL_CFLAGS += -DWITH_BLUETOOTH -DWITH_A2DP
  LOCAL_SHARED_LIBRARIES += liba2dp
endif
ifeq ($(AUDIO_POLICY_TEST),true)
  LOCAL_CFLAGS += -DAUDIO_POLICY_TEST
endif
ifeq ($(TARGET_SIMULATOR),true)
    ifeq ($(HOST_OS),linux)
        LOCAL_LDLIBS += -lrt -lpthread
    endif
endif
ifeq ($(BOARD_USE_LVMX),true)
    LOCAL_CFLAGS += -DLVMX
    LOCAL_C_INCLUDES += vendor/nxp
    LOCAL_STATIC_LIBRARIES += liblifevibes
    LOCAL_SHARED_LIBRARIES += liblvmxservice
#   LOCAL_SHARED_LIBRARIES += liblvmxipc
endif
include $(BUILD_SHARED_LIBRARY) 模块三编译成动态库
```

(4)编译一个应用程序(APK)

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

  # Build all java files in the java subdirectory
```

```
LOCAL_SRC_FILES := $(call all-subdir-java-files)

# Name of the APK to build
LOCAL_PACKAGE_NAME := LocalPackage

# Tell it to build an APK
include $(BUILD_PACKAGE)
```

(5)编译一个依赖于静态 Java 库(static.jar)的应用程序

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

# List of static libraries to include in the package
LOCAL_STATIC_JAVA_LIBRARIES := static-library

# Build all java files in the java subdirectory
LOCAL_SRC_FILES := $(call all-subdir-java-files)

# Name of the APK to build
LOCAL_PACKAGE_NAME := LocalPackage

# Tell it to build an APK
include $(BUILD_PACKAGE)
```

(6)编译一个需要用平台的 key 签名的应用程序

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

# Build all java files in the java subdirectory
LOCAL_SRC_FILES := $(call all-subdir-java-files)

# Name of the APK to build
LOCAL_PACKAGE_NAME := LocalPackage

LOCAL_CERTIFICATE := platform

# Tell it to build an APK
include $(BUILD_PACKAGE)
```

(7)编译一个需要用特定 key 前面的应用程序

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

# Build all java files in the java subdirectory
LOCAL_SRC_FILES := $(call all-subdir-java-files)
```

```
  # Name of the APK to build
  LOCAL_PACKAGE_NAME := LocalPackage

  LOCAL_CERTIFICATE := vendor/example/certs/app

  # Tell it to build an APK
  include $(BUILD_PACKAGE)
```

(8)添加一个预编译应用程序

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

  # Module name should match apk name to be installed.
  LOCAL_MODULE := LocalModuleName
  LOCAL_SRC_FILES := $(LOCAL_MODULE).apk
  LOCAL_MODULE_CLASS := APPS
  LOCAL_MODULE_SUFFIX := $(COMMON_ANDROID_PACKAGE_SUFFIX)

  include $(BUILD_PREBUILT)
```

(9)添加一个静态 JAVA 库

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

  # Build all java files in the java subdirectory
  LOCAL_SRC_FILES := $(call all-subdir-java-files)

  # Any libraries that this library depends on
  LOCAL_JAVA_LIBRARIES := android.test.runner

  # The name of the jar file to create
  LOCAL_MODULE := sample

  # Build a static jar file.
  include $(BUILD_STATIC_JAVA_LIBRARY)
```

(10)Android.mk 的编译模块中间可以定义相关的编译内容，也就是指定相关的变量如下：

```
LOCAL_AAPT_FLAGS

LOCAL_ACP_UNAVAILABLE

LOCAL_ADDITIONAL_JAVA_DIR

LOCAL_AIDL_INCLUDES
```

LOCAL_ALLOW_UNDEFINED_SYMBOLS

LOCAL_ARM_MODE

LOCAL_ASFLAGS

LOCAL_ASSET_DIR

LOCAL_ASSET_FILES 在 Android.mk 文件中编译应用程序(BUILD_PACKAGE)时设置此变量，表示资源文件，

通常会定义成 LOCAL_ASSET_FILES += $(call find-subdir-assets)

LOCAL_BUILT_MODULE_STEM

LOCAL_C_INCLUDES 额外的 C/C++编译头文件路径，用 LOCAL_PATH 表示本文件所在目录

举例如下：

LOCAL_C_INCLUDES += extlibs/zlib-1.2.3

LOCAL_C_INCLUDES += $(LOCAL_PATH)/src

LOCAL_CC 指定 C 编译器

LOCAL_CERTIFICATE 签名认证

LOCAL_CFLAGS 为 C/C++编译器定义额外的标志(如宏定义)，举例：LOCAL_CFLAGS += -DLIBUTILS_NATIVE=1

LOCAL_CLASSPATH

LOCAL_COMPRESS_MODULE_SYMBOLS

LOCAL_COPY_HEADERS install 应用程序时需要复制的头文件，必须同时定义 LOCAL_COPY_HEADERS_TO

LOCAL_COPY_HEADERS_TO install 应用程序时复制头文件的目的路径

LOCAL_CPP_EXTENSION 如果你的 C++文件不是以 cpp 为文件后缀，你可以通过 LOCAL_CPP_EXTENSION 指定 C++文件后缀名

如：LOCAL_CPP_EXTENSION := .cc

注意统一模块中 C++文件后缀必须保持一致。

LOCAL_CPPFLAGS 传递额外的标志给 C++编译器，如：LOCAL_CPPFLAGS += -ffriend-injection

LOCAL_CXX 指定 C++编译器

LOCAL_DX_FLAGS

LOCAL_EXPORT_PACKAGE_RESOURCES

LOCAL_FORCE_STATIC_EXECUTABLE 如果编译的可执行程序要进行静态链接(执行时不依赖于任何动态库)，则设置 LOCAL_FORCE_STATIC_EXECUTABLE:=true

目前只有 libc 有静态库形式，这个只有文件系统中/sbin 目录下的应用程序会用到，这个目录下的应用程序在运行时通常

文件系统的其它部分还没有加载，所以必须进行静态链接。

LOCAL_GENERATED_SOURCES

LOCAL_INSTRUMENTATION_FOR

LOCAL_INSTRUMENTATION_FOR_PACKAGE_NAME

LOCAL_INTERMEDIATE_SOURCES

LOCAL_INTERMEDIATE_TARGETS

LOCAL_IS_HOST_MODULE

LOCAL_JAR_MANIFEST

LOCAL_JARJAR_RULES

LOCAL_JAVA_LIBRARIES 编译 java 应用程序和库的时候指定包含的 java 类库，目前有 core 和 framework 两种

多数情况下定义成：LOCAL_JAVA_LIBRARIES := core framework

注意 LOCAL_JAVA_LIBRARIES 不是必须的，而且编译 APK 时不允许定义(系统会自动添加)

LOCAL_JAVA_RESOURCE_DIRS

LOCAL_JAVA_RESOURCE_FILES

LOCAL_JNI_SHARED_LIBRARIES

LOCAL_LDFLAGS 传递额外的参数给连接器(务必注意参数的顺序)

LOCAL_LDLIBS 为可执行程序或者库的编译指定额外的库，指定库以"-lxxx"格式，举例：

```
LOCAL_LDLIBS += -lcurses -lpthread
LOCAL_LDLIBS += -Wl,-z,origin
```

LOCAL_MODULE 生成的模块的名称(注意应用程序名称用 LOCAL_PACKAGE_NAME 而不是 LOCAL_MODULE)

LOCAL_MODULE_PATH 生成模块的路径

LOCAL_MODULE_STEM

LOCAL_MODULE_TAGS 生成模块的标记

LOCAL_NO_DEFAULT_COMPILER_FLAGS

LOCAL_NO_EMMA_COMPILE

LOCAL_NO_EMMA_INSTRUMENT

LOCAL_NO_STANDARD_LIBRARIES

LOCAL_OVERRIDES_PACKAGES

LOCAL_PACKAGE_NAME APK 应用程序的名称

LOCAL_POST_PROCESS_COMMAND

LOCAL_PREBUILT_EXECUTABLES 预编译 including $(BUILD_PREBUILT) 或者 $(BUILD_HOST_PREBUILT)时所用,指定需要复制的可执行文件

LOCAL_PREBUILT_JAVA_LIBRARIES

LOCAL_PREBUILT_LIBS 预编译 including $(BUILD_PREBUILT)或者$(BUILD_HOST_PREBUILT)时所用,指定需要复制的库.

LOCAL_PREBUILT_OBJ_FILES

LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES

LOCAL_PRELINK_MODULE 是否需要预连接处理(默认需要，用来做动态库优化)

LOCAL_REQUIRED_MODULES 指定模块运行所依赖的模块(模块安装时将会同步安装它所依赖的模块)

LOCAL_RESOURCE_DIR

LOCAL_SDK_VERSION

LOCAL_SHARED_LIBRARIES 可链接动态库

LOCAL_SRC_FILES 编译源文件

LOCAL_STATIC_JAVA_LIBRARIES

LOCAL_STATIC_LIBRARIES 可链接静态库

LOCAL_UNINSTALLABLE_MODULE

LOCAL_UNSTRIPPED_PATH

LOCAL_WHOLE_STATIC_LIBRARIES 指定模块所需要载入的完整静态库(这些精通库在链接是不允许链接器删除其中无用的代码)

LOCAL_YACCFLAGS

OVERRIDE_BUILT_MODULE_PATH

Android 系统移植(一)-让 android 系统在目标平台上运行起来

Android 系统由于用的是 linux 内核，因此内核移植和嵌入式 linux 内核移植差异不大，过程如下：

(1)移植 boot-loader 和 linux2.6 内核到目标平台上，让 linux 内核可以启动起来，基本的驱动允许正常。
此过程完全是嵌入式 linux 的开发，这里直接跳过。需要注意的是，由于 android 已经被 linux 官方开除，因此从
网站上(如 http://www.kernel.org/)下载的最新 linux 内核源代码已经不包含 android 的专有驱动，因此建议
从 google 网上下下载 Linux 内核，android 源代码浏览网站如下：
http://android.git.kernel.org/
从该网站上发现内核相关的包如下：
kernel/common.git 通用 android 内核项目
kernel/experimental.git 实验性内核项目
kernel/linux-2.6.git 这个是标准的 Linux 内核，没有 android 的驱动
kernel/lk.git 微内核项目
kernel/msm.git 这个是高通 msm7xxx 系列芯片所用内核
kernel/omap.git
kernel/tegra.git NVIDIA Tegra 系列芯片所用内核
下载内核代码的方法如下：
git clone git://android.git.kernel.org/kernel/common.git
下载完后用 git branch -a 查看所有 git 分支,结果如下：
  android-2.6.27
  origin/HEAD
  origin/android-2.6.25
  origin/android-2.6.27

origin/android-2.6.29
origin/android-2.6.32
origin/android-2.6.35
origin/android-2.6.36
origin/android-goldfish-2.6.27
origin/android-goldfish-2.6.29
然后切换到最新分支 git checkout origin/android-2.6.36

(2)修改内核配置文件，打开 Android 必须的驱动(日志和 BINDER)如下：
CONFIG_ANDROID=y
CONFIG_ANDROID_BINDER_IPC=y
CONFIG_ANDROID_LOGGER=y
此部分的代码在内核 drivers/staging/android 目录下。

(3)为了提高启动速度，采用 ramdisk，将 android 文件系统的部分内容压缩到内核中。
首先打开内核驱动：
CONFIG_BLK_DEV_INITRD=y
CONFIG_INITRAMFS_SOURCE="root"
CONFIG_INITRAMFS_ROOT_UID=0
CONFIG_INITRAMFS_ROOT_GID=0
然后在 android 源代码编译出来的 out/target/product/merlin/root 目录复制到内核目录下。

(4)根据 android 文件系统的要求对 nand flash 进行重新分区，举例如下：
将 nand flash 分区以下 8 个分区
NTIM
OBM
U-boot
Kernel
System
UserData
Mass Storage
BBT

(5)根据分区表修改内核启动参数如下：
CONFIG_CMDLINE="ubi.mtd=4 ubi.mtd=5 ubi.mtd=6 root=ubi0_0 rootfstype=ubifs console=ttyS1,115200 uart_dma init=/init"
参数的意思是：载入的文件系统部分有 3 个分区，分别为 nand flash 的第 4,5,6 分区(从 0 编号)，文件系统采用 ubifs 格式，控制台设备为 ttyS1,波特率为 115200
启动的第一个应用程序是/init

(6)确保控制台的设置和硬件保持一致，如：硬件上串口用的是 UART1，则内核启动参数中设置有 console=ttyS1,而且 android 的启动过程中设要设置正确，修改
部分位于 android 源代码 system/core/init/init.c 文件中，将
static char *console_name = "/dev/console";
修改成
static char *console_name = "/dev/ttyS1";

(7)修改 android 源代码 system/core/rootdir 目录下的 init.rc 文件，作如下修改(android 默认 yaffs2 文件系统)：
首先将 android 文件系统修改成可读写，将
```
mount rootfs rootfs / ro remount
```
修改成
```
mount rootfs rootfs / rw remount
```
然后修改挂载 system 和 userdata 部分的代码，将
```
# 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

# We chown/chmod /data again so because mount is run as root + defaults
mount yaffs2 mtd@userdata /data nosuid nodev
chown system system /data
chmod 0771 /data
```
改成
```
# Mount /system rw first to give the filesystem a chance to save a checkpoint
mount ubifs ubi0_0 /system ro

# We chown/chmod /data again so because mount is run as root + defaults
mount ubifs ubi1_0 /data nosuid nodev
chown system system /data
chmod 0771 /data
```

(8)完成后编译内核，可以启动文件系统，控制台可用，但是没有显示启动 log,而且不停的重启。

(9)系统不停的重启，因此控制台已经可用了，自然而然的想到看到 logcat 日志，一看，发现 logcat 设备居然没起来，配置文件里面都定义了
居然没起来，查看了下内核 drivers/staging/android 目录，没有.o 文件，证明是没编译到，在看内核目录下的.config 文件，发现居然没有了
logcat 和 binder 的宏定义，配置文件里面有定义而.config 文件中无定义，肯定是相关 Kconfig 文件的问题，通过分析 drivers/staging 目录下的
Kconfig 文件发现是因为 STAGING_EXCLUDE_BUILD 宏默认是 y,在配置文件中否定此宏即可，在配置文件中 CONFIG_STAGING 定义后加上即可，如下：
```
CONFIG_STAGING=y
# CONFIG_STAGING_EXCLUDE_BUILD is not set
```
修改后重新编译发现系统完成正常启动，启动过程中启动 log 也显示正常。

至此，android 初步移植工作已经完成，当然，系统还有很多问题，需要下一步继续修改。

总结：android 的移植按如下流程：
(1)android linux 内核的普通驱动移植，让内核可以在目标平台上运行起来。
(2)正确挂载文件系统，确保内核启动参数和 android 源代码 system/core/rootdir 目录下的 init.rc 中的文件系统挂载正确。
(3)调试控制台，让内核启动参数中的 console 参数以及 android 源代码 system/core/init/init.c 中的

console_name 设置和硬件保持一致
(4)打开 android 相关的驱动(logger,binder 等)，串口输入 logcat 看 logger 驱动起来，没有的话调试 logger 驱动。

说明：ARM 的内核配置文件定义在内核 arch/arm/configs 目录下

Android 系统移植(二)-按键移植

这一部分主要是移植 android 的键盘和按键
(1)Android 使用标准的 linux 输入事件设备(/dev/input 目录下)和驱动，按键定义在内核 include/linux/input.h 文件中，
按键定义形式如下：

```
#define KEY_ESC         1
#define KEY_1           2
#define KEY_2           3
```

(2)内核中(我的平台是 arch/arm/mach-mmp/merlin.c 文件)中按键的定义如下形式：

```
static struct gpio_keys_button btn_button_table[] = {
   [0] = {
      .code          =    KEY_F1,
      .gpio          =    MFP_PIN_GPIO2,
      .active_low    =    1,       /* 0 for down 0, up 1; 1 for down 1, up 0 */
      .desc          =    "H_BTN button",
      .type          =    EV_KEY,
      /* .wakeup          = */
      .debounce_interval    =    10,       /* 10 msec jitter elimination */
   },
   [1] = {
      .code          =    KEY_F2,
      .gpio          =    MFP_PIN_GPIO3,
      .active_low    =    1,       /* 0 for down 0, up 1; 1 for down 1, up 0 */
      .desc          =    "O_BTN button",
      .type          =    EV_KEY,
      /* .wakeup          = */
      .debounce_interval    =    10,       /* 10 msec jitter elimination */
   },
   [2] = {
      .code          =    KEY_F4,
      .gpio          =    MFP_PIN_GPIO1,
      .active_low    =    1,       /* 0 for down 0, up 1; 1 for down 1, up 0 */
      .desc          =    "S_BTN button",
      .type          =    EV_KEY,
```

```
        /* .wakeup          = */
        .debounce_interval    =    10,       /* 10 msec jitter elimination */
    },
};
static struct gpio_keys_platform_data gpio_keys_data = {
    .buttons  = btn_button_table,
    .nbuttons = ARRAY_SIZE(btn_button_table),
};

static struct platform_device gpio_keys = {
    .name = "gpio-keys",
    .dev  = {
        .platform_data = &gpio_keys_data,
    },
    .id   = -1,
};
```

上面定义是将 MFP_PIN_GPIO2 这个 GPIO 口的按键映射到 Linux 的 KEY_F1 按键，MPF_PIN_GPIO3 映射到 KEY_F2，MFP_PIN_GPIO1 映射到 KEY_F4

(3)上面(2)步实现了从硬件 GPIO 口到内核标准按键的映射,但是 android 并没有直接使用映射后的键值，而且对其再进行了一次映射，从内核标准键值
到 android 所用键值的映射表定义在 android 文件系统的/system/usr/keylayout 目录下。标准的映射文件为 qwerty.kl，定义如下：

```
key 399   GRAVE
key 2     1
key 3     2
key 4     3
key 5     4
key 6     5
key 7     6
key 8     7
key 9     8
key 10    9
key 11    0
key 158   BACK            WAKE_DROPPED
key 230   SOFT_RIGHT       WAKE
key 60    SOFT_RIGHT       WAKE
key 107   ENDCALL          WAKE_DROPPED
key 62    ENDCALL          WAKE_DROPPED
key 229   MENU             WAKE_DROPPED
key 139   MENU             WAKE_DROPPED
key 59    MENU             WAKE_DROPPED
key 127   SEARCH           WAKE_DROPPED
key 217   SEARCH           WAKE_DROPPED
key 228   POUND
key 227   STAR
```

key 231 CALL WAKE_DROPPED
key 61 CALL WAKE_DROPPED
key 232 DPAD_CENTER WAKE_DROPPED
key 108 DPAD_DOWN WAKE_DROPPED
key 103 DPAD_UP WAKE_DROPPED
key 102 HOME WAKE
key 105 DPAD_LEFT WAKE_DROPPED
key 106 DPAD_RIGHT WAKE_DROPPED
key 115 VOLUME_UP
key 114 VOLUME_DOWN
key 116 POWER WAKE
key 212 CAMERA

key 16 Q
key 17 W
key 18 E
key 19 R
key 20 T
key 21 Y
key 22 U
key 23 I
key 24 O
key 25 P
key 26 LEFT_BRACKET
key 27 RIGHT_BRACKET
key 43 BACKSLASH

key 30 A
key 31 S
key 32 D
key 33 F
key 34 G
key 35 H
key 36 J
key 37 K
key 38 L
key 39 SEMICOLON
key 40 APOSTROPHE
key 14 DEL

key 44 Z
key 45 X
key 46 C
key 47 V
key 48 B
key 49 N

key 50 M
key 51 COMMA
key 52 PERIOD
key 53 SLASH
key 28 ENTER

key 56 ALT_LEFT
key 100 ALT_RIGHT
key 42 SHIFT_LEFT
key 54 SHIFT_RIGHT
key 15 TAB
key 57 SPACE
key 150 EXPLORER
key 155 ENVELOPE

key 12 MINUS
key 13 EQUALS
key 215 AT

(4)android 对底层按键的处理方法
android 按键的处理是 Window Manager 负责，主要的映射转换实现在 android 源代码 frameworks/base/libs/ui/EventHub.cpp
此文件处理来自底层的所有输入事件，并根据来源对事件进行分类处理，对于按键事件，处理过程如下

(a)记录驱动名称为
(b) 获取环境变量 ANDROID_ROOT 为系统路径(默认是/system，定义在 android 源代码/system/core/rootdir/init.rc 文件中)
(c)查找路径为"系统路径/usr/keylayout/驱动名称.kl"的按键映射文件，如果不存在则默认用路径为"系统路径/usr/keylayout/qwerty.kl"
这个默认的按键映射文件，映射完成后再把经映射得到的 android 按键码值发给上层应用程序。
所以我们可以在内核中定义多个按键设备，然后为每个设备设定不同的按键映射文件，不定义则会默认用 qwerty.kl

(5)举例
上面(2)步我们在内核中声明了一个名为"gpio-keys"的按键设备，此设备定义在内核 drivers/input/keyboard/gpio_keys.c 文件中
然后我们在内核启动过程中注册此设备： platform_device_register(&gpio_keys);
然后我们可以自己定义一个名为 gpio-keys.kl 的 android 按键映射文件，此文件的定义可以参考 querty.kl 的内容，比如说我们想将 MPF_PIN_GPIO3
对应的按键作 android 中的 MENU 键用，首先我们在内核中将 MPF_PIN_GPIO3 映射到 KEY_F2，在内核 include/linux/input.h 中查找 KEY_F2 发现
#define KEY_F2 60
参照 KEY_F2 的值我们在 gpio-keys.kl 中加入如下映射即可
key 60 MENU WAKE
其它按键也照此添加，完成后将按键表放置到/system/usr/keylayout 目录下即可。

补充：
(1)android 按键设备的映射关系可以在 logcat 开机日志中找的到(查找 EventHub 即可)
(2)android 按键设备由 Window Manager 负责，Window Manager 从按键驱动读取内核按键码，然后将内核按键码转换成 android 按键码，转换完成
后 Window Manager 会将内核按键码和 android 按键码一起发给应用程序来使用，这一点一定要注意。

Android 系统开发小知识-在 android 产品开发中添加新的编译模块

Android 开发中用户内容定义在 vendor 目录下，而用户产品的内容都定义在 vendor/<company_name>/<board_name>目录下
如果需要添加新的内容，可以在该目录下新建子目录，同时修改 AndroidBoard.mk 文件即可。比如说要添加一个按键映射文件：
(1)在 vendor/<company_name>/<board_name>目录下建立一个 keymaps 子目录
(2)将我们需要的按键映射文件 gpio-keys.kl 和 power-button.kl 复制到 keymaps 目录下
(3)在 keymaps 目录下新建一个 Mdroid.mk 文件，内容如下：

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

file := $(TARGET_OUT_KEYLAYOUT)/gpio-keys.kl
ALL_PREBUILT += $(file)
$(file): $(LOCAL_PATH)/gpio-keys.kl | $(ACP)
    $(transform-prebuilt-to-target)

file := $(TARGET_OUT_KEYLAYOUT)/power-button.kl
ALL_PREBUILT += $(file)
$(file): $(LOCAL_PATH)/power-button.kl | $(ACP)
    $(transform-prebuilt-to-target)
```

(4)在 vendor/<company_name>/<board_name>目录下的 AndroidBoard.mk 添加如下内容：

```
include $(LOCAL_PATH)/keymaps/Mdroid.mk
```

Android 系统移植(三)-按键字符表

上节讲 android 的 Window Manager 将内核按键码通过按键映射表转换成 android 按键码，
这节讲的是 android 按键码向 android 字符的转换，转换也是通过 Window Manager 来完成的
(1)原始按键字符表，我们知道一个按键是可以显示多个字符的，决定显示字符的是 CAPS(大小写),FN,NUNMBER 等按键
举例如下：

```
[type=QWERTY]

# keycode      display number  base    caps    fn      caps_fn

A               'A'     '2'     'a'     'A'     '#'     0x00
B               'B'     '2'     'b'     'B'     '<'     0x00
```

C	'C'	'2'	'c'	'C'	'9'	0x00E7
D	'D'	'3'	'd'	'D'	'5'	0x00
E	'E'	'3'	'e'	'E'	'2'	0x0301
F	'F'	'3'	'f'	'F'	'6'	0x00A5
G	'G'	'4'	'g'	'G'	'-'	'_'
H	'H'	'4'	'h'	'H'	'['	'{'
I	'I'	'4'	'i'	'I'	'$'	0x0302
J	'J'	'5'	'j'	'J'	']'	'}'
K	'K'	'5'	'k'	'K'	'"'	'~'
L	'L'	'5'	'l'	'L'	''	'`'
M	'M'	'6'	'm'	'M'	'!'	0x00
N	'N'	'6'	'n'	'N'	'>'	0x0303
O	'O'	'6'	'o'	'O'	'('	0x00
P	'P'	'7'	'p'	'P'	')'	0x00
Q	'Q'	'7'	'q'	'Q'	'*'	0x0300
R	'R'	'7'	'r'	'R'	'3'	0x20AC
S	'S'	'7'	's'	'S'	'4'	0x00DF
T	'T'	'8'	't'	'T'	'+'	0x00A3
U	'U'	'8'	'u'	'U'	'&'	0x0308
V	'V'	'8'	'v'	'V'	'='	'^'
W	'W'	'9'	'w'	'W'	'1'	0x00
X	'X'	'9'	'x'	'X'	'8'	0xEF00
Y	'Y'	'9'	'y'	'Y'	'%'	0x00A1
Z	'Z'	'9'	'z'	'Z'	'7'	0x00

on pc keyboards

COMMA	','	','	','	';'	';'	'|'
PERIOD	'.'	'.'	'.'	':'	':'	0x2026
AT	'@'	'0'	'@'	'0'	'0'	0x2022
SLASH	'/'	'/'	'/'	'?'	'?'	'\'

SPACE	0x20	0x20	0x20	0x20	0xEF01	0xEF01
ENTER	0xa	0xa	0xa	0xa	0xa	0xa

TAB	0x9	0x9	0x9	0x9	0x9	0x9
0	'0'	'0'	'0'	')'	')'	')'
1	'1'	'1'	'1'	'!'	'!'	'!'
2	'2'	'2'	'2'	'@'	'@'	'@'
3	'3'	'3'	'3'	'#'	'#'	'#'
4	'4'	'4'	'4'	'$'	'$'	'$'
5	'5'	'5'	'5'	'%'	'%'	'%'
6	'6'	'6'	'6'	'^'	'^'	'^'
7	'7'	'7'	'7'	'&'	'&'	'&'
8	'8'	'8'	'8'	'*'	'*'	'*'
9	'9'	'9'	'9'	'('	'('	'('

```
GRAVE          '`'    '`'    '`'    '~'    '`'    '~'
MINUS          '-'    '-'    '-'    '_'    '-'    '_'
EQUALS         '='    '='    '='    '+'    '='    '+'
LEFT_BRACKET   '['    '['    '['    '{'    '['    '{'
RIGHT_BRACKET  ']'    ']'    ']'    '}'    ']'    '}'
BACKSLASH      '\'    '\'    '\'    '|'    '\'    '|'
SEMICOLON      ';'    ';'    ';'    ':'    ';'    ':'
APOSTROPHE     '"'    '"'    '"'    '"'    '"'    '"'
STAR           '*'    '*'    '*'    '*'    '*'    '*'
POUND          '#'    '#'    '#'    '#'    '#'    '#'
PLUS           '+'    '+'    '+'    '+'    '+'    '+'
```

(2)android 为了减少载入时间，并没有使用原始按键表文件，而是将其转换成二进制文件
转换的工具源代码在 android 源代码 build/tools/kcm 目录下，android 在编译过程中会
首先编译转换工具，然后利用转换工具将 android 源代码 sdk/emulator/keymaps 目录下
的 qwerty.kcm 和 qwerty2.kcm 文件分别转换成 qwerty.kcm.bin 和 qwerty2.kcm.bin
转换后的二进制文件复制到 out/target/product/<board_name>/system/usr/keychars
目录下，也就是目标平台的/system/usr/keychars 目录中。

(3)Window Manager 对按键的处理在 android 源代码 frameworks/base/libs/ui/EventHub.cpp 文件中
Window Manager 从内核接收到一个按键输入事件后会首先调用按键映射表将内核按键码映射成 android 按键码(这部分上节已讲)，然后会
将 android 按键码转换成字符，具体过程如下：
(a)设置系统系统属性 hw.keyboards.设备号.devname 的值为设备名
以上节的 gpio-keys 设备为例，会设置系统属性 hw.keyboards.65539.devname 的值为 gpio-keys
(b)载入按键字符表，首先载入/system/usr/keychars 目录下的设备名.kcm.bin 文件(此例即 gpio-keys.kcm.bin 文件)，如果载入失败
则载入该目录下的 querty.kcm.bin.
(c)利用载入的按键字符表将 android 按键转换成按键字符发给上层应用程序。

(4)一般情况下一个控制按键是不需要作按键字符表的，系统会调用默认的去处理，但是如果要开发一个全功能键盘(包含了字母和数字)，那可能就需要
自己作一个专用的按键字符表了。

android 系统开发小问题－启动过程中 android 字符没有显示出来

android 目标平台可以正常启动，但是启动过程中的 android 字符没有显示出来，这个是 linux 内核配置的问题
打开内核 framebuffer 控制台即可。
(1)make menuconifg 后选择 Device Drivers->Graphics support->Console display driver support->Framebuffer Console support
然后打开相关的几个配置选项即可。
(2)直接修改内核配置文件，如下：
CONFIG_FRAMEBUFFER_CONSOLE=y

```
CONFIG_FRAMEBUFFER_CONSOLE_DETECT_PRIMARY=y
# CONFIG_FRAMEBUFFER_CONSOLE_ROTATION is not set
CONFIG_FONTS=y
CONFIG_FONT_8x8=y
CONFIG_FONT_8x16=y
CONFIG_FONT_6x11=y
# CONFIG_FONT_7x14 is not set
# CONFIG_FONT_PEARL_8x8 is not set
# CONFIG_FONT_ACORN_8x8 is not set
# CONFIG_FONT_MINI_4x6 is not set
# CONFIG_FONT_SUN8x16 is not set
# CONFIG_FONT_SUN12x22 is not set
# CONFIG_FONT_10x18 is not set
```

(3)android 启动过程中的 android 字符显示在源代码的 system/core/init.c 中，如下：

```
    if( load_565rle_image(INIT_IMAGE_FILE) ) {
    fd = open("/dev/tty0", O_WRONLY);
    if (fd >= 0) {
        const char *msg;
            msg = "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"  // console is 40 cols x 30 lines
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "\n"
        "           A N D R O I D ";
        write(fd, msg, strlen(msg));
        close(fd);
    }
    }
```

android 启动过程配置文件的解析与语法

(1)android 启动文件系统后调用的第一个应用程序是/init，此文件的很重要的内容是解析了 init.rc 和 init.xxx.rc
两个配置文件，然后执行解析出来的任务。相关代码在 android 源代码/system/core/init/init.c 文件中，

如下：

```
    parse_config_file("/init.rc");

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

    get_hardware_name();
    snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
    parse_config_file(tmp);
```

(2)从上面代码可以看到，第一个配置文件名称固定为 init.rc,而第二个配置文件格式为 init.xxx.rc，其中 xxx 部分的内容
是从内核读取的，具体是读取文件/proc/cpuinfo 中的 Hardware 部分，然后截取其部分内容 。Hardware 部分是定义在内核的
主板定义文件中，我的平台是定义在内核 arch/arm/mach-mmp/merlin.c 中，我的平台定义如下：

```
MACHINE_START(ARDENT_MERLIN, "PXA168-based Merlin Platform")
    .phys_io = APB_PHYS_BASE,
    .boot_params = 0x00000100,
    .io_pg_offst = (APB_VIRT_BASE >> 18) & 0xfffc,
    .map_io = pxa_map_io,
    .init_irq = pxa168_init_irq,
    .timer = &pxa168_timer,
    .init_machine = merlin_init,
MACHINE_END
```

这样截取到的 hardware 部分的内容就为 pxa168-based，也就是说我的平台的第二个配置文件应该命名为 init.pxa168-based.rc

(3)从上面看 init.xxx.rc 中的 xxx 内容是取决是内核中主板的定义的，如果觉得麻烦，可以将其在代码中写死，例如：

```
    parse_config_file("init.merlin.rc");
```

(4)配置文件的语法如下：
(a)配置文件的内容包含有 4 种：

- 动作(Action)
- 命令(Commands)
- 服务(Services)
- 选项(Options)

(b)动作和命令一起使用，形式如下：

```
on <trigger>
  <command>
  <command>
  <command>
```

其中 trigger 是触发条件，也就是说在满足触发条件的情况下执行 1 个或多个相应的命令，举例如下：

```
on property:persist.service.adb.enable=1
    start adbd
```

(c)服务和选项一起使用，形式如下：
 service <name> <pathname> [<argument>]*
 <option>
 <option>
 ...
上面内容解释为：
 service 服务名称 服务对应的命令的路径 命令的参数
  选项
  选项
 ...
举例如下：
service ril-daemon /system/bin/rild
  socket rild stream 660 root radio
  socket rild-debug stream 660 radio system
  user root
  group radio cache inet misc audio
上面的服务对应到/system/bin/rild 命令，没有参数，服务名称为 ril-daemon,后面的内容都是服务的选项。

(d)选项是影响服务启动和运行的参数，主要的选项如下：

disabled 禁用服务，此服务开机时不会自动启动，但是可以在应用程序中手动启动它。

socket <type> <name> <perm> [<user> [<group>]]
套接字 类型 名称 权限 用户 组
创建一个名为/dev/socket/<name>，然后把它的 fd 传给启动程序
类型 type 的值为 dgram 或者 stream
perm 表示该套接字的访问权限,user 和 group 表示改套接字所属的用户和组，这两个参数默认都是 0，因此可以不设置。

user <username>
执行服务前切换到用户<username>，此选项默认是 root，因此可以不设置。

group <groupname> [<groupname>]*
执行服务前切换到组<groupname>,此选项默认是 root,因此可以不设置

capability [<capability>]+
执行服务前设置 linux capability，没什么用。

oneshot
服务只启动一次，一旦关闭就不能再启动。

class <name>
为服务指定一个类别，默认为"default"，同一类别的服务必须一起启动和停止

(e)动作触发条件<trigger>
boot 首个触发条件，初始化开始(载入配置文件)的时候触发

<name>=<value>
当名为<name>的属性(property)的值为<value>的时候触发

device-added-<path>
路径为<path>的设置添加的时候触发

device-removed-<path>
路径为<path>的设置移除的时候触发

service-exited-<name>
名为<name>的服务关闭的时候触发

(f)命令(Command)的形式
exec <path> [<argument>]*
复制(fork)和执行路径为<path>的应用程序，<argument>为该应用程序的参数，在该应用程序执行完前，此命令会屏蔽，

export <name> <value>
声明名为<name>的环境变量的值为<value>，声明的环境变量是系统环境变量，启动后一直有效。

ifup <interface>
启动名为<interface>的网络接口

import <filename>
加入新的位置文件，扩展当前的配置。

hostname <name>
设置主机名

class_start <serviceclass>
启动指定类别的所有服务

class_stop <serviceclass>
停止指定类别的所有服务

domainname <name>
设置域名

insmod <path>
加载路径为<path>的内核模块

mkdir <path>

创建路径为<path>目录

mount <type> <device> <dir> [<mountoption>]*
挂载类型为<type>的设备<device>到目录<dir>,<mountoption>为挂载参数，距离如下：
    mount ubifs ubi1_0 /data nosuid nodev

setkey
暂时未定义

setprop <name> <value>
设置名为<name>的系统属性的值为<value>

setrlimit <resource> <cur> <max>
设置资源限制，举例：
set RLIMIT_NICE to allow priorities from 19 to -20
    setrlimit 13 40 40
没看懂是什么意思。

start <service>
启动服务(如果服务未运行)

stop <service>
停止服务(如果服务正在运行)

symlink <target> <path>
创建一个从<path>指向<target>的符号链接，举例：
    symlink /system/etc /etc

write <path> <string> [<string>]*
打开路径为<path>的文件并将一个多这多个字符串写入到该文件中。

(g)系统属性(Property)
android 初始化过程中会修改一些属性，通过 getprop 命令我们可以看到属性值，这些属性指示了某些动作或者服务的状态，主要如下：
init.action　　如果当前某个动作正在执行则 init.action 属性的值等于该动作的名称，否则为""
init.command　　如果当前某个命令正在执行则 init.command 属性的值等于该命令的名称，否则为""
init.svc.<name>　此属性指示个名为 <name> 的服务的状态 ("stopped", "running", 或者 "restarting").

android 系统开发(四)-触摸屏 tslib 移植(内核)和原理分析

首先了解一下 tslib 的运行原理，tslib 的运行分成两部分
(1)校验
在 LCD 固定坐标位置依次显示出 5 个坐标让用户触摸，把 LCD 坐标和用户触摸时驱动屏驱动底层的坐标总共 5 组值保存起来
运行 tslib 库的算法对其进行运算，得出校准用 7 个值

(2)校准
每次触摸屏驱动读取到硬件坐标时应用校准用的 7 个值对该坐标进行一次运算，然后将运算后的坐标作为正常坐标即可。

按照上面的原理，
(1) 我们先修改内核部分，我的平台用的触摸屏幕驱动是 tsc2007，驱动文件为内核/drivers/input/touchscreen
目录下的 tsc2007.c 和 ts_linear.c
其中，ts_linear.c 中定义的是校准模块，该模块在 proc 文件系统中建立了 7 个文件，用来存放校准用的 7 个点，7 的点的默认值
为 1,0,0,0,1,0,1，对应的目标平台文件系统的位置为 /proc/sys/dev/ts_device 目录下 a0,a1,a2,a3,a4,a5,a6 等 7 个文件
此模块中还定义了一个校准函数 ts_linear_scale，此函数的主要内容是读取 a0,a1,a2,a3,a4,a5,a6 等 7 个文件中的值作为 7 个
校准值与传入的触摸平坐标值进行运算，返回运算结果。
ts_linear_scale 函数定义如下：

```
int ts_linear_scale(int *x, int *y, int swap_xy)
{
    int xtemp, ytemp;

    xtemp = *x;
    ytemp = *y;

    if (cal.a[6] == 0)
        return -EINVAL;

    *x = (cal.a[2] + cal.a[0] * xtemp + cal.a[1] * ytemp) / cal.a[6];
    *y = (cal.a[5] + cal.a[3] * xtemp + cal.a[4] * ytemp) / cal.a[6];

    if (swap_xy) {
        int tmp = *x;
        *x = *y;
        *y = tmp;
    }
```

```
    return 0;
}
```

ts2007.c 为触摸屏驱，与其他驱动不同的地方是在取得硬件坐标值发送之前先调用了 ts_linear_scale 函数对坐标值进行了校准

```
        if (x > 0 && y > 0)
        {
            ts_linear_scale(&x, &y, invert);
            input_report_abs(input, ABS_X, x);
            input_report_abs(input, ABS_Y, y);
            input_report_abs(input, ABS_PRESSURE, 255);
            input_report_abs(input, ABS_TOOL_WIDTH, 1);
            input_report_key(input, BTN_TOUCH, 1);
            input_sync(input);
        }
```

(2)在 android 源代码/system/core/rootdir/init.rc 文件中添加 tslib 相关的宏定义如下：

```
# touchscreen parameters
    export TSLIB_FBDEVICE /dev/graphics/fb0
    export TSLIB_CALIBFILE /data/etc/pointercal
    export TSLIB_CONFFILE  /system/etc/ts.conf
    export TSLIB_TRIGGERDEV /dev/input/event0
    export TSLIB_TSDEVICE /dev/input/event1
```

(2)移植 tslib 库到 android 系统，比较麻烦，看下一节的内容。

(3)校验程序完成后会将生成的 7 个校准值写入到环境变量 TSLIB_CALIBFILE 对应的路径/data/etc/pointercal 文件中

(4)校验完后将 pointercal 文件中的 7 个值分别写入到/proc/sys/dev/ts_device 目录下 a0,a1,a2,a3,a4,a5,a6 文件即可。

(5)开机启动的时候我们编写一个应用程序，首先判断环境变量 TSLIB_CALIBFILE 对应的路径/data/etc/pointercal 文件是否存在，如果
文件存在而且非空，则将该文件中的 7 个值取出来分别写入到/proc/sys/dev/ts_device 目录下 a0,a1,a2,a3,a4,a5,a6 文件

(6)为了确保未校验前触摸屏可用，我们将一次校验后得出的 7 个坐标值作为初始值，修改到内核 ts_linear.c 文件中。

下面是源代码：
ts_linear.c 文件

```
/*
 *  Touchscreen Linear Scale Adaptor
 *
 *  Copyright (C) 2009 Marvell Corporation
```

```
 *
 *  Author: Mark F. Brown <markb@marvell.com>
 *  Based on tslib 1.0 plugin linear.c by Russel King
 *
 * This library is licensed under GPL.
 *
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/input.h>
#include <linux/interrupt.h>
#include <linux/wait.h>
#include <linux/delay.h>
#include <linux/platform_device.h>
#include <linux/proc_fs.h>
#include <linux/sysctl.h>
#include <asm/system.h>

/*
 * sysctl-tuning infrastructure.
 */
static struct ts_calibration {
/* Linear scaling and offset parameters for x,y (can include rotation) */
    int a[7];
} cal;

static ctl_table ts_proc_calibration_table[] = {
    {
     .ctl_name = CTL_UNNUMBERED,
     .procname = "a0",
     .data = &cal.a[0],
     .maxlen = sizeof(int),
     .mode = 0666,
     .proc_handler = &proc_dointvec,
     },
    {
     .ctl_name = CTL_UNNUMBERED,
     .procname = "a1",
     .data = &cal.a[1],
     .maxlen = sizeof(int),
     .mode = 0666,
     .proc_handler = &proc_dointvec,
     },
    {
```

```
    .ctl_name = CTL_UNNUMBERED,
    .procname = "a2",
    .data = &cal.a[2],
    .maxlen = sizeof(int),
    .mode = 0666,
    .proc_handler = &proc_dointvec,
    },
   {
    .ctl_name = CTL_UNNUMBERED,
    .procname = "a3",
    .data = &cal.a[3],
    .maxlen = sizeof(int),
    .mode = 0666,
    .proc_handler = &proc_dointvec,
    },
   {
    .ctl_name = CTL_UNNUMBERED,
    .procname = "a4",
    .data = &cal.a[4],
    .maxlen = sizeof(int),
    .mode = 0666,
    .proc_handler = &proc_dointvec,
    },
   {
    .ctl_name = CTL_UNNUMBERED,
    .procname = "a5",
    .data = &cal.a[5],
    .maxlen = sizeof(int),
    .mode = 0666,
    .proc_handler = &proc_dointvec,
    },
   {
    .ctl_name = CTL_UNNUMBERED,
    .procname = "a6",
    .data = &cal.a[6],
    .maxlen = sizeof(int),
    .mode = 0666,
    .proc_handler = &proc_dointvec,
    },

   {.ctl_name = 0}
};

static ctl_table ts_proc_root[] = {
   {
    .ctl_name = CTL_UNNUMBERED,
```

```
    .procname = "ts_device",
    .mode = 0555,
    .child = ts_proc_calibration_table,
    },
   {.ctl_name = 0}
};

static ctl_table ts_dev_root[] = {
   {
    .ctl_name = CTL_DEV,
    .procname = "dev",
    .mode = 0555,
    .child = ts_proc_root,
    },
   {.ctl_name = 0}
};

static struct ctl_table_header *ts_sysctl_header;

int ts_linear_scale(int *x, int *y, int swap_xy)
{
   int xtemp, ytemp;

   xtemp = *x;
   ytemp = *y;

   if (cal.a[6] == 0)
      return -EINVAL;

   *x = (cal.a[2] + cal.a[0] * xtemp + cal.a[1] * ytemp) / cal.a[6];
   *y = (cal.a[5] + cal.a[3] * xtemp + cal.a[4] * ytemp) / cal.a[6];

   if (swap_xy) {
      int tmp = *x;
      *x = *y;
      *y = tmp;
   }
   return 0;
}

EXPORT_SYMBOL(ts_linear_scale);

static int __init ts_linear_init(void)
{
   ts_sysctl_header = register_sysctl_table(ts_dev_root);
   /* Use default values that leave ts numbers unchanged after transform */
```

```
    cal.a[0] = 1;
    cal.a[1] = 0;
    cal.a[2] = 0;
    cal.a[3] = 0;
    cal.a[4] = 1;
    cal.a[5] = 0;
    cal.a[6] = 1;
    return 0;
}

static void __exit ts_linear_cleanup(void)
{
    unregister_sysctl_table(ts_sysctl_header);
}

module_init(ts_linear_init);
module_exit(ts_linear_cleanup);

MODULE_DESCRIPTION("touch screen linear scaling driver");
MODULE_LICENSE("GPL");
```

ts2007.c文件

```
/*
 *  linux/drivers/input/touchscreen/tsc2007.c
 *
 *  touch screen driver for tsc2007
 *
 *  Copyright (C) 2006, Marvell Corporation
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License version 2 as
 *  published by the Free Software Foundation.
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/input.h>
#include <linux/interrupt.h>
#include <linux/wait.h>
#include <linux/delay.h>
#include <linux/platform_device.h>
#include <linux/freezer.h>
#include <linux/proc_fs.h>
```

```
#include <linux/clk.h>
#include <linux/i2c.h>
#include <mach/gpio.h>

#include <linux/sysctl.h>
#include <asm/system.h>

extern int ts_linear_scale(int *x, int *y, int swap_xy);

/* Use MAV filter */
#define TSC_CMD_SETUP 0xb0

/* Use 12-bit */
#define TSC_CMD_X 0xc0
#define TSC_CMD_PLATEX 0x80
#define TSC_CMD_Y 0xd0
#define TSC_CMD_PLATEY 0x90

#define TSC_X_MAX 4096
#define TSC_Y_MAX 4096
#define TSC_X_MIN 0
#define TSC_Y_MIN 0

/* delay time for compute x, y, computed as us */

#define DEBUG
#ifdef DEBUG
#define TS_DEBUG(fmt,args...) printk(KERN_DEBUG fmt, ##args )
#else
#define TS_DEBUG(fmt,args...)
#endif
static int x_min=TSC_X_MIN;
static int y_min=TSC_Y_MIN;
static int x_max=TSC_X_MAX;
static int y_max=TSC_Y_MAX;
static int invert = 0;
static int debounce_time  = 150;
static int init_debounce = true;
static int delay_time = 1;

enum tsc2007_status {
    PEN_UP,
    PEN_DOWN,
};

struct _tsc2007 {
```

```
    struct input_dev *dev;
    int x;        /* X sample values */
    int y;        /* Y sample values */

    int status;
    struct work_struct irq_work;
    struct i2c_client *client;
    unsigned long last_touch;
};
struct _tsc2007 *g_tsc2007;

/* update abs params when min and max coordinate values are set */
int tsc2007_proc_minmax(struct ctl_table *table, int write, struct file *filp,
                    void __user *buffer, size_t *lenp, loff_t *ppos)
{
    struct _tsc2007 *tsc2007= g_tsc2007;
    struct input_dev *input = tsc2007->dev;

    /* update value */
    int ret = proc_dointvec(table, write, filp, buffer, lenp, ppos);

    /* updated abs params */
    if (input) {
        TS_DEBUG(KERN_DEBUG "update x_min %d x_max %d"
            " y_min %d y_max %d\n", x_min, x_max,
            y_min, y_max);
        input_set_abs_params(input, ABS_X, x_min, x_max, 0, 0);
        input_set_abs_params(input, ABS_Y, y_min, y_max, 0, 0);
    }
    return ret;
}

static ctl_table tsc2007_proc_table[] = {
    {
        .ctl_name    = CTL_UNNUMBERED,
        .procname    = "x-max",
        .data        = &x_max,
        .maxlen        = sizeof(int),
        .mode        = 0666,
        .proc_handler    = &tsc2007_proc_minmax,
    },
    {
        .ctl_name    = CTL_UNNUMBERED,
        .procname    = "y-max",
        .data        = &y_max,
        .maxlen        = sizeof(int),
```

```
        .mode       = 0666,
        .proc_handler   = &tsc2007_proc_minmax,
    },
    {
        .ctl_name   = CTL_UNNUMBERED,
        .procname   = "x-min",
        .data       = &x_min,
        .maxlen       = sizeof(int),
        .mode       = 0666,
        .proc_handler   = &tsc2007_proc_minmax,
    },
    {
        .ctl_name   = CTL_UNNUMBERED,
        .procname   = "y-min",
        .data       = &y_min,
        .maxlen       = sizeof(int),
        .mode       = 0666,
        .proc_handler   = &tsc2007_proc_minmax,
    },
    {
        .ctl_name   = CTL_UNNUMBERED,
        .procname   = "invert_xy",
        .data       = &invert,
        .maxlen       = sizeof(int),
        .mode       = 0666,
        .proc_handler   = &proc_dointvec,
    },
    {
        .ctl_name   = CTL_UNNUMBERED,
        .procname   = "debounce_time",
        .data       = &debounce_time,
        .maxlen       = sizeof(int),
        .mode       = 0666,
        .proc_handler   = &proc_dointvec,
    },
    {
        .ctl_name   = CTL_UNNUMBERED,
        .procname   = "delay_time",
        .data       = &delay_time,
        .maxlen       = sizeof(int),
        .mode       = 0666,
        .proc_handler   = &proc_dointvec,
    },
    { .ctl_name = 0 }
};
```

```
static ctl_table tsc2007_proc_root[] = {
   {
      .ctl_name    = CTL_UNNUMBERED,
      .procname    = "ts_device",
      .mode        = 0555,
      .child       = tsc2007_proc_table,
   },
   { .ctl_name = 0 }
};

static ctl_table tsc2007_proc_dev_root[] = {
   {
      .ctl_name    = CTL_DEV,
      .procname    = "dev",
      .mode        = 0555,
      .child       = tsc2007_proc_root,
   },
   { .ctl_name = 0 }
};

static struct ctl_table_header *sysctl_header;

static int __init init_sysctl(void)
{
   sysctl_header = register_sysctl_table(tsc2007_proc_dev_root);
   return 0;
}

static void __exit cleanup_sysctl(void)
{
   unregister_sysctl_table(sysctl_header);
}

static int tsc2007_measure(struct i2c_client *client, int *x, int * y)
{
   u8 x_buf[2] = {0, 0};
   u8 y_buf[2] = {0, 0};

   i2c_smbus_write_byte(client, TSC_CMD_PLATEX);
   msleep_interruptible(delay_time);

   i2c_smbus_write_byte(client, TSC_CMD_X);
   i2c_master_recv(client, x_buf, 2);
   *x = (x_buf[0]<<4) | (x_buf[1] >>4);
```

```
    i2c_smbus_write_byte(client, TSC_CMD_PLATEY);
    msleep_interruptible(delay_time);

    i2c_smbus_write_byte(client, TSC_CMD_Y);
    i2c_master_recv(client, y_buf, 2);
    *y = (y_buf[0]<<4) | (y_buf[1] >>4);
    *y = 4096 - *y; //added by allen
    printk("\ntouchscreen x = 0x%x, y = 0x%x\n",*x,*y);
    return 0;
}

static void tsc2007_irq_work(struct work_struct *work)
{
    struct _tsc2007 *tsc2007= g_tsc2007;
    struct i2c_client *client = tsc2007-> client;
    struct input_dev *input = tsc2007->dev;

    int x = -1, y = -1, is_valid = 0;
    int tmp_x = 0, tmp_y = 0;

    int gpio = irq_to_gpio(client->irq);


    /* Ignore if PEN_DOWN */
    if(PEN_UP == tsc2007->status){

        if (gpio_request(gpio, "tsc2007 touch detect")) {
            printk(KERN_ERR "Request GPIO failed, gpio: %X\n", gpio);
            return;
        }
        gpio_direction_input(gpio);

        while(0 == gpio_get_value(gpio)){

                    if ((jiffies_to_msecs(
                            ((long)jiffies - (long)tsc2007->last_touch)) <
                 debounce_time &&
                tsc2007->status == PEN_DOWN) ||
                init_debounce)
                    {
                init_debounce = false;
                            tsc2007_measure(client, &tmp_x, &tmp_y);
                            TS_DEBUG(KERN_DEBUG
                "dropping pen touch %lu %lu (%u)\n",
                            jiffies, tsc2007->last_touch,
```

```
                    jiffies_to_msecs(
            (long)jiffies - (long)tsc2007->last_touch));
                    schedule();
            continue;
                }


            /* continue report x, y */
            if (x > 0 && y > 0)
            {
               ts_linear_scale(&x, &y, invert);
               input_report_abs(input, ABS_X, x);
               input_report_abs(input, ABS_Y, y);
               input_report_abs(input, ABS_PRESSURE, 255);
               input_report_abs(input, ABS_TOOL_WIDTH, 1);
               input_report_key(input, BTN_TOUCH, 1);
               input_sync(input);
            }

            tsc2007->status = PEN_DOWN;
            tsc2007_measure(client, &x, &y);
            TS_DEBUG(KERN_DEBUG "pen down x=%d y=%d!\n", x, y);
            is_valid = 1;
            schedule();
         }

         if (is_valid)
         {
            /*consider PEN_UP */
            tsc2007->status = PEN_UP;
            input_report_abs(input, ABS_PRESSURE, 0);
            input_report_abs(input, ABS_TOOL_WIDTH, 1);
            input_report_key(input, BTN_TOUCH, 0);
            input_sync(input);
            tsc2007->last_touch = jiffies;
            TS_DEBUG(KERN_DEBUG "pen up!\n");
         }

         gpio_free(gpio);
      }
   }

   static irqreturn_t tsc2007_interrupt(int irq, void *dev_id)
   {
      schedule_work(&g_tsc2007->irq_work);
```

```
    return IRQ_HANDLED;
}

static int __devinit tsc2007_probe(struct i2c_client *client,
                const struct i2c_device_id *id)
{
    struct _tsc2007 *tsc2007;
    struct input_dev *input_dev;
    int ret;

    tsc2007 = kzalloc(sizeof(struct _tsc2007), GFP_KERNEL);
    input_dev = input_allocate_device();

    g_tsc2007 = tsc2007;

    if (!tsc2007 || !input_dev) {
        ret = -ENOMEM;
        goto fail1;
    }

    i2c_set_clientdata(client, tsc2007);

    tsc2007->dev = input_dev;

    input_dev->name = "tsc2007";
    input_dev->phys = "tsc2007/input0";

    //input_dev->id.bustype = BUS_HOST;
    input_dev->dev.parent = &client->dev;

    __set_bit(EV_KEY, input_dev->evbit);
    __set_bit(BTN_TOUCH, input_dev->keybit);

    __set_bit(EV_ABS, input_dev->evbit);
    __set_bit(ABS_PRESSURE, input_dev->evbit);
    __set_bit(ABS_X, input_dev->evbit);
    __set_bit(ABS_Y, input_dev->evbit);

    input_set_abs_params(input_dev, ABS_X, x_min, x_max, 0, 0);
    input_set_abs_params(input_dev, ABS_Y, y_min, y_max, 0, 0);
    input_set_abs_params(input_dev, ABS_PRESSURE, 0, 255, 0, 0);

    ret = request_irq(client->irq, tsc2007_interrupt,
        IRQF_DISABLED | IRQF_TRIGGER_FALLING,
         "tsc2007 irq", NULL);
    if (ret){
```

```
        printk(KERN_ERR "tsc2007 request irq failed\n");
        goto fail2;
    }

    ret = input_register_device(tsc2007->dev);
    if (ret){
        printk(KERN_ERR "tsc2007 register device fail\n");
        goto fail2;
    }

    /*init */
    tsc2007->status = PEN_UP;
    tsc2007->client = client;
    tsc2007->last_touch = jiffies;

    INIT_WORK(&tsc2007->irq_work, tsc2007_irq_work);

    /* init tsc2007 */
    i2c_smbus_write_byte(client, TSC_CMD_SETUP);

    return 0;

 fail2:
    free_irq(client->irq, client);
 fail1:
    i2c_set_clientdata(client, NULL);
    input_free_device(input_dev);
    kfree(tsc2007);
    return ret;
}

static int __devexit tsc2007_remove(struct i2c_client *client)
{
    struct _tsc2007 *tsc2007 = i2c_get_clientdata(client);

    if(client->irq)
        free_irq(client->irq, client);

    i2c_set_clientdata(client, NULL);
    input_unregister_device(tsc2007->dev);
    kfree(tsc2007);

    return 0;
}

static struct i2c_device_id tsc2007_idtable[] = {
```

```
    { "tsc2007", 0 },
    { }
};

MODULE_DEVICE_TABLE(i2c, tsc2007_idtable);

static struct i2c_driver tsc2007_driver = {
    .driver = {
        .name     = "tsc2007",
    },
    .id_table       = tsc2007_idtable,
    .probe        = tsc2007_probe,
    .remove         = __devexit_p(tsc2007_remove),
};

static int __init tsc2007_ts_init(void)
{
    init_sysctl();
    return i2c_add_driver(&tsc2007_driver);
}

static void __exit tsc2007_ts_exit(void)
{
    cleanup_sysctl();
    i2c_del_driver(&tsc2007_driver);
}

module_init(tsc2007_ts_init);
module_exit(tsc2007_ts_exit);

MODULE_DESCRIPTION("tsc2007 touch screen driver");
MODULE_LICENSE("GPL");
```

android 系统开发(五)-tslib 移植

(1)切换至 tslib 目录然后执行如下命令(以 marvell 平台为例)

```
./autogen.sh
echo "ac_cv_func_malloc_0_nonnull=yes" > arm-marvell-linux.cache
./configure --host=arm-marvell-linux-gnueabi --prefix=/work/svn/ts_build --cache-file=arm-marvell-linux.cache
```

上面三步仅仅是为了取得 tslib 目录下的 config.h 文件

(2)将 tslib 复制到 android 源代码 vendor/<company_name>/<board_name>目录下

(3)修改 vendor/<company_name>/<board_name>目录下的 AndroidBoard.mk 文件，加入如下内容
include $(LOCAL_PATH)/tslib/Mdroid.mk
一定要主义 LOCAL_PATH 这个宏的时效性

(4)在 tslib 目录下创建 Mdroid.mk，内容如下：
```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

TS_PATH := $(LOCAL_PATH)

include $(TS_PATH)/src/Mdroid.mk
include $(TS_PATH)/plugins/Mdroid.mk
include $(TS_PATH)/tests/Mdroid.mk

include $(CLEAR_VARS)
file := $(TARGET_OUT_ETC)/ts.conf
$(file) : $(TS_PATH)/etc/ts.conf | $(ACP)
    $(transform-prebuilt-to-target)
ALL_PREBUILT += $(file)
```

(5)在 tslib/src 目录下创建 Mdroid.mk，内容如下：
```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= ts_attach.c ts_close.c ts_config.c \
    ts_error.c ts_fd.c ts_load_module.c ts_open.c ts_parse_vars.c \
    ts_read.c ts_read_raw.c ts_option.c

LOCAL_C_INCLUDES += \
        $(LOCAL_PATH)/../

LOCAL_SHARED_LIBRARIES += libutils libcutils

LOCAL_SHARED_LIBRARIES += libdl
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE := libts

include $(BUILD_SHARED_LIBRARY)
```

(6)在 tslib/plugins 目录下创建 Mdroid.mk，内容如下：
```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
```

```
LOCAL_SRC_FILES:= input-raw.c

LOCAL_C_INCLUDES += \
	$(LOCAL_PATH)/../ \
	$(LOCAL_PATH)/../src

LOCAL_SHARED_LIBRARIES := libts
LOCAL_MODULE := input
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)


include $(CLEAR_VARS)
LOCAL_SRC_FILES:= pthres.c

LOCAL_C_INCLUDES += \
	$(LOCAL_PATH)/../ \
	$(LOCAL_PATH)/../src

LOCAL_SHARED_LIBRARIES := libts
LOCAL_MODULE := pthres
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)


include $(CLEAR_VARS)
LOCAL_SRC_FILES:= variance.c

LOCAL_C_INCLUDES += \
	$(LOCAL_PATH)/../ \
	$(LOCAL_PATH)/../src

LOCAL_SHARED_LIBRARIES := libts
LOCAL_MODULE := variance
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)


include $(CLEAR_VARS)
LOCAL_SRC_FILES:= dejitter.c

LOCAL_C_INCLUDES += \
	$(LOCAL_PATH)/../ \
	$(LOCAL_PATH)/../src
```

```
LOCAL_SHARED_LIBRARIES := libts
LOCAL_MODULE := dejitter
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_SRC_FILES:= linear.c

LOCAL_C_INCLUDES += \
	$(LOCAL_PATH)/../ \
	$(LOCAL_PATH)/../src

LOCAL_SHARED_LIBRARIES := libts
LOCAL_MODULE := linear
LOCAL_PRELINK_MODULE := false
include $(BUILD_SHARED_LIBRARY)
```

(7)在 tslib/tests 目录下创建 Mdroid.mk，内容如下：

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

LOCAL_SRC_FILES:= ts_calibrate.c fbutils.c testutils.c font_8x8.c font_8x16.c

LOCAL_C_INCLUDES += \
	$(LOCAL_PATH)/../ \
	$(LOCAL_PATH)/../src

LOCAL_SHARED_LIBRARIES := libts

LOCAL_SHARED_LIBRARIES += libutils libcutils

LOCAL_MODULE := tscalibrate

include $(BUILD_EXECUTABLE)
```

(8)在 tslib/config.h 文件中加入如下定义：

```
#define TS_CONF  "/system/etc/ts.conf"
#define PLUGIN_DIR "/system/lib"
#define TS_POINTERCAL "/data/etc/pointercal"
```

(9)将下面路径文件

tslib/src/ts_open.c

tslib/tests/ts_calibrate.c
tslib/tests/fbutils.c
中的
#include <sys/fcntl.h>
修改成
#include <fcntl.h>

(10)将 tslib/tests/ts_calibrate.c 文件中
static int clearbuf(struct tsdev *ts)
修改为
static void clearbuf(struct tsdev *ts)

(11)修改 tslib/etc/ts.conf 内容如下：
module_raw input
module pthres pmin=1
module variance delta=30
module dejitter delta=100
module linear

(12)在 android 源代码 init.rc 中声明 tslib 相关的宏如下：
touchscreen parameters
    export TSLIB_FBDEVICE /dev/graphics/fb0
    export TSLIB_CALIBFILE /data/etc/pointercal
    export TSLIB_CONFFILE /system/etc/ts.conf
    export TSLIB_TRIGGERDEV /dev/input/event0
    export TSLIB_TSDEVICE /dev/input/event1

(13)重新编译后即可调用 tscalibrate 命令来校验触摸屏，校验后产生一个/data/etc/pointercal 文件

Ubuntu 下 svn 安装和使用

1 第一步：安装软件。
安装客户端
sudo apt-get install subversion
安装服务器端
sudo apt-get install libapache2-svn

2 svn 的基本操作
(1)从服务器上下载代码：svn checkout
举例：
svn checkout svn://192.168.6.10/project/Source_code/trunk/src
svn checkout 可以所写成 svn co

(2)添加新的文件或者文件夹到本地版本控制中
svn add 文件(夹)
如果指定的是一个文件夹，则会将文件夹的所有内容都添加进来，如果你只想要添加文件夹而不是文件夹里面的内容，则用如下参数
svn add --non-recursive 目录名

(3)提交本地更改到服务器
svn commit -m "说明信息" [-N] [--no-unlock] [文件(夹)]
文件(夹)不填写则代码提交当前目录下(包含子目录)的所有更改
举例：
svn commit -m "modify some code"

(4)显示本地修改状态
svn status [path]缩写成 svn st [path]
path 为空则代码显示当前目录下的所有修改文件(递归到子目录)的状态，状态显示如下：
? 不在 svn 控制中
M 内容被修改
C 发生冲突
A 预定加入到版本库中
K 被锁定
举例：svn st

(5)显示指定目录下所有文件(递归到子目录)的状态
svn status -v [path] 缩写成 svn st -v [path]

(6)同步服务器代码到本地仓库
svn up

(7)显示指定目录下的文件和目录
svn list [path]缩写成 svn ls [path]

(8)恢复本地修改(撤销指定目录下的未提交的所有修改)
svn revert path [--depth infinity]

(9)删除文件(夹)
svn delete 文件(夹)

3 svn 的配置文件
修改/root/.subversion 目录下的 config 文件。
比如说修改 svn 所控制的文件类型，则可以修改 config 文件中的 global-ignores 参数，这个参数是指定了 svn 版本控制忽略的
文件类型，举例如下：
global-ignores = *.o *.lo *.la *.al .[0-9]* *.a *.pyc *.pyo

android 系统开发(六)-HAL 层开发基础

Android HAL 层，即硬件抽象层，是 Google 响应厂家“希望不公开源码”的要求推出的新概念
1，源代码和目标位置
源代码： /hardware/libhardware 目录,该目录的目录结构如下：
/hardware/libhardware/hardware.c 编译成 libhardware.so，目标位置为/system/lib 目录
/hardware/libhardware/include/hardware 目录下包含如下头文件：
hardware.h 通用硬件模块头文件
copybit.h copybit 模块头文件
gralloc.h gralloc 模块头文件
lights.h 背光模块头文件
overlay.h overlay 模块头文件
qemud.h qemud 模块头文件
sensors.h 传感器模块头文件
/hardware/libhardware/modules 目录下定义了很多硬件模块
这些硬件模块都编译成 xxx.xxx.so，目标位置为/system/lib/hw 目录

2，HAL 层的实现方式
JNI->通用硬件模块->硬件模块->内核驱动接口
具体一点：JNI->libhardware.so->xxx.xxx.so->kernel
具体来说：android frameworks 中 JNI 调用/hardware/libhardware/hardware.c 中定义的 hw_get_module 函数来获取硬件模块，
然后调用硬件模块中的方法，硬件模块中的方法直接调用内核接口完成相关功能

3,通用硬件模块(libhardware.so)
(1)头文件为：/hardware/libhardware/include/hardware/hardware.h
头文件中主要定义了通用硬件模块结构体 hw_module_t，声明了 JNI 调用的接口函数 hw_get_module
hw_module_t 定义如下：

```
typedef struct hw_module_t {
    /** tag must be initialized to HARDWARE_MODULE_TAG */
    uint32_t tag;

    /** major version number for the module */
    uint16_t version_major;

    /** minor version number of the module */
    uint16_t version_minor;

    /** Identifier of module */
    const char *id;

    /** Name of this module */
    const char *name;

    /** Author/owner/implementor of the module */
    const char *author;
```

```
    /** Modules methods */
    struct hw_module_methods_t* methods; //硬件模块的方法

    /** module's dso */
    void* dso;

    /** padding to 128 bytes, reserved for future use */
    uint32_t reserved[32-7];

} hw_module_t;
```

硬件模块方法结构体 hw_module_methods_t 定义如下：

```
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;
```

只定义了一个 open 方法，其中调用的设备结构体参数 hw_device_t 定义如下：

```
typedef struct hw_device_t {
    /** tag must be initialized to HARDWARE_DEVICE_TAG */
    uint32_t tag;

    /** version number for hw_device_t */
    uint32_t version;

    /** reference to the module this device belongs to */
    struct hw_module_t* module;

    /** padding reserved for future use */
    uint32_t reserved[12];

    /** Close this device */
    int (*close)(struct hw_device_t* device);

} hw_device_t;
```

hw_get_module 函数声明如下：

```
int hw_get_module(const char *id, const struct hw_module_t **module);
```

参数 id 为模块标识，定义在/hardware/libhardware/include/hardware 目录下的硬件模块头文件中，
参数 module 是硬件模块地址，定义了/hardware/libhardware/include/hardware/hardware.h 中

(2)hardware.c 中主要是定义了 hw_get_module 函数如下：

```
#define HAL_LIBRARY_PATH "/system/lib/hw"
static const char *variant_keys[] = {
    "ro.hardware",
    "ro.product.board",
```

```
    "ro.board.platform",
    "ro.arch"
};
static const int HAL_VARIANT_KEYS_COUNT =
    (sizeof(variant_keys)/sizeof(variant_keys[0]));

int hw_get_module(const char *id, const struct hw_module_t **module)
{
    int status;
    int i;
    const struct hw_module_t *hmi = NULL;
    char prop[PATH_MAX];
    char path[PATH_MAX];
    for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++)
    {
        if (i < HAL_VARIANT_KEYS_COUNT)
        {
            if (property_get(variant_keys[i], prop, NULL) == 0)
            {
                continue;
            }
            snprintf(path, sizeof(path), "%s/%s.%s.so",
                    HAL_LIBRARY_PATH, id, prop);
        }
        else
        {
            snprintf(path, sizeof(path), "%s/%s.default.so",
                    HAL_LIBRARY_PATH, id);
        }
        if (access(path, R_OK))
        {
            continue;
        }
        /* we found a library matching this id/variant */
        break;
    }

    status = -ENOENT;
    if (i < HAL_VARIANT_KEYS_COUNT+1) {
        /* load the module, if this fails, we're doomed, and we should not try
         * to load a different variant. */
        status = load(id, path, module);
    }

    return status;
}
```

从源代码我们可以看出，hw_get_module 完成的主要工作是根据模块 id 寻找硬件模块动态连接库地址，然后调用 load 函数去打开动态连接库
并从动态链接库中获取硬件模块结构体地址。硬件模块路径格式如下：
HAL_LIBRARY_PATH/id.prop.so
HAL_LIBRARY_PATH 定义为/system/lib/hw
id 是 hw_get_module 函数的第一个参数所传入,prop 部分首先按照 variant_keys 数组中的名称逐一调用 property_get 获取对应的系统属性，
然后访问 HAL_LIBRARY_PATH/id.prop.so，如果找到能访问的就结束，否则就访问 HAL_LIBRARY_PATH/id.default.so
举例如下：
假定访问的是背光模块，id 定义为"lights"则系统会按照如下的顺序去访问文件：
/system/lib/hw/lights.[ro.hardware 属性值].so
/system/lib/hw/lights.[ro.product.board 属性值].so
/system/lib/hw/lights.[ro.board.platform 属性值].so
/system/lib/hw/lights.[ro.arch 属性值].so
/system/lib/hw/lights.default.so
所以开发硬件模块的时候 Makefile 文件(Android.mk)中模块的命名 LOCAL_MODULE 要参考上面的内容，否则就会访问不到没作用了。

load 函数的关键部分代码如下：

```
    handle = dlopen(path, RTLD_NOW);  //打开动态链接库
    if (handle == NULL) {
        char const *err_str = dlerror();
        LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
        status = -EINVAL;
        goto done;
    }

    const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
    hmi = (struct hw_module_t *)dlsym(handle, sym); //从动态链接库中获取硬件模块结构体的指针
    if (hmi == NULL) {
        LOGE("load: couldn't find symbol %s", sym);
        status = -EINVAL;
        goto done;
    }
```

HAL_MODULE_INFO_SYM_AS_STR 是硬件模块在动态链接库中的标志，定义在 hardware.h 中如下：

```
#define HAL_MODULE_INFO_SYM         HMI
#define HAL_MODULE_INFO_SYM_AS_STR  "HMI"
```

4,硬件模块
硬件模块的开发主要是完成/hardware/libhardware/include/hardware 目录下对应的头文件中的内容，主要是硬件模块头文件和 hardware.h 中
的结构体中定义了一些函数指针，调用内核提供的接口将具体的函数实现，然后编译成指定名称的动态链接库放到/system/lib/hw 目录下即可。
用一句话来概括：硬件模块的开发就是定义一个 hardware.h 中定义的 hw_module_t 结构体，结构体名称为宏 HAL_MODULE_INFO_SYM，然后实现结构体

的相关内容即可。

5,内核驱动
主要是要向用户层开放接口，让硬件模块和内核可以交互。

android 系统开发(七)-背光模块

1,总论
背光模块属于 HAL 层开发，HAL 层开发，用一句话来概括就是定义一个 hardware.h 中定义的名称为宏 HAL_MODULE_INFO_SYM 的 hw_module_t 结构体，
然后实现结构体的相关内容

2,驱动方面的准备
简单的嵌入式 linux 驱动，编写 LCD 背光驱动，并提供接口给上层修改，我所用的是直接修改接口文件，接口如下：
/sys/class/backlight/pwm-backlight/brightness 这个是亮度调节
/sys/class/backlight/pwm-backlight/max_brightness 这个是最大亮度，按照 android 系统的要求应该设置成 255
控制亮度直接写 brightness 文件即可
背光驱动主要是通过 PWM 来完成，这里不详细说明。

3,需要包含的头文件
/hardware/libhardware/include/hardware 目录下的 hardware.h 和 lights.h
其中 hardware.h 中定义了通用硬件模块，lights.h 中定义了背光设备相关的内容

4,android 已有的硬件模块在/hardware/libhardware/modules 目录下，为了区分，我们开发的背光模块放置在如下的目录：
vendor/ardent/merlin/lights 目录下，编译成 lights.default.so 放置到/system/lib/hw 目录下，模块命名规则可以
参考上一节的内容。

5,修改 vendor/ardent/merlin 目录下 AndroidBoard.mk 文件，添加如下内容：
include $(LOCAL_PATH)/lights/Mdroid.mk

6,lights 目录新建 Mdroid.mk 文件，内容如下：
```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SRC_FILES:= lights.c
```

```
LOCAL_SHARED_LIBRARIES := \
    libutils \
    libcutils \
    libhardware

LOCAL_PRELINK_MODULE := false

LOCAL_MODULE := lights.default

include $(BUILD_SHARED_LIBRARY)
```

7,lights 目录下新建一个 lights.c 文件，如下：

```
const struct hw_module_t HAL_MODULE_INFO_SYM = {
    .tag = HARDWARE_MODULE_TAG,
    .version_major = 1,
    .version_minor = 0,
    .id = LIGHTS_HARDWARE_MODULE_ID,
    .name = "lights module",
    .author = "allen",
    .methods = NULL,
};
```

8，上面的内容可以直接编译通过，但是因为我将其 methods 部分指向了空指针，因此没有任何功能，下面来实现此部分

hw_module_t 机构体的 methods 成员是一个指向 hw_module_methods_t 结构体的一个指针，hw_module_methods_t 结构体定义如下：

```
typedef struct hw_module_methods_t {
    int (*open)(const struct hw_module_t* module, const char* id,struct hw_device_t** device);
} hw_module_methods_t;
```

据此我们定义一个 hw_module_methods_t 类型的参数 lights_module_methods 如下：

```
struct hw_module_methods_t lights_module_methods = {
    .open = lights_device_open
};
```

然后将上面的 methods 由 NULL 改成 lights_module_methods

9,接下来就是定义 lights_device_open 函数了，此函数的参数和返回值由 hw_module_methods_t 结构体的 open 成员决定，此函数定义如下：

```
static int lights_device_open(const struct hw_module_t *module,const char *id, struct hw_device_t **device)
```

从 lights_device_open 函数的参数来看，第一个参数和第二个参数是常量，第三个参数是 一个指向 hw_device_t 结构体的指针，因此可以断定

实现此函数也就是要完成第三个参数的内容，详细的内容我们可以参考直接调用该函数的内容，在 frameworks/base/services/jni 目录下的

com_android_server_LightsService.cpp 文件中，内容如下：

```
static light_device_t* get_device(hw_module_t* module, char const* name)
{
    int err;
```

```
    hw_device_t* device;
    err = module->methods->open(module, name, &device);
    if (err == 0) {
        return (light_device_t*)device;//device 由 hw_device_t 指针强制转换成 light_device_t 指针
    } else {
        return NULL;
    }
}

static jint init_native(JNIEnv *env, jobject clazz)
{
    int err;
    hw_module_t* module;
    Devices* devices;

    devices = (Devices*)malloc(sizeof(Devices));

    err = hw_get_module(LIGHTS_HARDWARE_MODULE_ID, (hw_module_t const**)&module);
    if (err == 0) {
        devices->lights[LIGHT_INDEX_BACKLIGHT]
                = get_device(module, LIGHT_ID_BACKLIGHT);
        devices->lights[LIGHT_INDEX_KEYBOARD]
                = get_device(module, LIGHT_ID_KEYBOARD);
        devices->lights[LIGHT_INDEX_BUTTONS]
                = get_device(module, LIGHT_ID_BUTTONS);
        devices->lights[LIGHT_INDEX_BATTERY]
                = get_device(module, LIGHT_ID_BATTERY);
        devices->lights[LIGHT_INDEX_NOTIFICATIONS]
                = get_device(module, LIGHT_ID_NOTIFICATIONS);
        devices->lights[LIGHT_INDEX_ATTENTION]
                = get_device(module, LIGHT_ID_ATTENTION);
        devices->lights[LIGHT_INDEX_BLUETOOTH]
                = get_device(module, LIGHT_ID_BLUETOOTH);
        devices->lights[LIGHT_INDEX_WIFI]
                = get_device(module, LIGHT_ID_WIFI);
    } else {
        memset(devices, 0, sizeof(Devices));
    }

    return (jint)devices;
}
```

从上面的内容我们可以看出 lights_device_open 的第一个参数是 JNI 层用 hw_get_module 所获得，第二个参数根据设备的不同有很多种情况

该参数的内容定义在 lights.h 中，全部情况如下：

```
#define LIGHT_ID_BACKLIGHT          "backlight"
#define LIGHT_ID_KEYBOARD           "keyboard"
```

```
#define LIGHT_ID_BUTTONS           "buttons"
#define LIGHT_ID_BATTERY           "battery"
#define LIGHT_ID_NOTIFICATIONS     "notifications"
#define LIGHT_ID_ATTENTION         "attention"
#define LIGHT_ID_BLUETOOTH         "bluetooth"
#define LIGHT_ID_WIFI              "wifi"
```

lights 调节有背光，键盘，按键，电池，通知，提醒，蓝牙和 WIF

第三个参数是一个指向一个 hw_device_t 的指针，但是 com_android_server_LightsService.cpp 文件中的背光调节函数定义如下：

```
static void setLight_native(JNIEnv *env, jobject clazz, int ptr,
        int light, int colorARGB, int flashMode, int onMS, int offMS, int brightnessMode)
{
    Devices* devices = (Devices*)ptr;
    light_state_t state;

    if (light < 0 || light >= LIGHT_COUNT || devices->lights[light] == NULL) {
        return ;
    }

    memset(&state, 0, sizeof(light_state_t));
    state.color = colorARGB;
    state.flashMode = flashMode;
    state.flashOnMS = onMS;
    state.flashOffMS = offMS;
    state.brightnessMode = brightnessMode;

    devices->lights[light]->set_light(devices->lights[light], &state);
}
```

get_device 函数中将 hw_device_t 指针强制转换成 light_device_t 指针给调节背光用，而 light_device_t 定义如下：

```
struct light_device_t {
    struct hw_device_t common;
    int (*set_light)(struct light_device_t* dev,
            struct light_state_t const* state);
};
```

因此在实现 lights_device_open 的第三个参数的时候，我们应该定义一个 light_device_t 类型结构体，然后将起 common 域的指针地址传递过去。这样虽然传递的是一个 hw_device_t 指针地址，但是 JNI 层可以将其强制转换

成 light_device_t 指针地址用，否则 devices->lights[light]->set_light 就会起不到作用了。实现如下：

```
static int lights_device_open(const struct hw_module_t *module,const char *id, struct hw_device_t **device)
{
    struct light_device_t *dev = NULL;
    int resvalue = -1;
    dev = calloc(sizeof(struct light_device_t),1);
    dev->common.tag = HARDWARE_DEVICE_TAG;
    dev->common.version = 0;
```

```
    dev->common.module = (struct hw_module_t *)module;
    dev->common.close = lights_device_close;
    if(!strcmp(id, LIGHT_ID_BACKLIGHT))
    {
        dev->set_light = lcd_set_light;
        resvalue = 0;
    }
    else
    {
        dev->set_light = other_set_light;
        resvalue = 0;
    }
    *device = &dev->common;
    return resvalue;
}
```

10，实现 lights_device_close，lcd_set_light 和 other_set_light，这个主要是调用驱动提供的接口直接控制硬件，举例如下：

```
static int lights_device_close(struct hw_device_t* device)
{
    struct light_device_t *m_device = (struct light_device_t *)device;
    if(m_device)
        free(m_device);
    return 0;
}
static int lcd_set_light(struct light_device_t* dev,struct light_state_t const* state)
{
    int fd = -1;
    int bytes = 0;
    int rlt = -1;
    unsigned char brightness = ((77*((state->color>>16)&0x00ff))
                                + (150*((state->color>>8)&0x00ff))
                                + (29*(state->color&0x00ff))) >> 8;
    fd = open("/sys/class/backlight/pwm-backlight/brightness", O_RDWR);
    if(fd>0)
    {
        char buffer[20];
     memset(buffer, 0, 20);
    bytes = sprintf(buffer, "%d", brightness);
    rlt = write(fd, buffer, bytes);
        if(rlt>0)
        {
           close(fd);
           return 0;
        }
    }
```

```
    close(fd);
    return -1;
}

static int other_set_light(struct light_device_t* dev,struct light_state_t const* state)
{
    return 0;
}
```

11，因为上面调节背光是通过写/sys/class/backlight/pwm-backlight/brightness 文件来完成，因此一定要设置该文件的权限，
在 init.xxx.rc 文件中添加如下的内容：

```
    # for control LCD backlight
    chown system system /sys/class/backlight/pwm-backlight/brightness
    chmod 0666 /sys/class/backlight/pwm-backlight/brightness
```

12，修改完成后经验证亮度调节可用，上面的例子只是实现了 lights 部分功能，如果需要完成所有的功能，请参考 hardware.h, lights.h 和 com_android_server_LightsService.cpp 文件中的内容。

android 系统开发(八)-SDCARD

关于 android 系统开发 sdcard 移植，主要有如下工作：
1，内核驱动开发，完成后每次插入和拔出 sdcard 系统都会有相关的信息显示，而且 sdcard 可以手动挂载。

2，android 的 sdcard 挂载主要是 vold 来完成，vold 的源代码在/system/vold 目录下，编译成/system/bin/vold
init.rc 文件中有 vold 系统服务，确保 android 系统开机后 vold 有正常运行。

3，添加 vold 的配置文件，先查看/system/bin/vold/main.cpp 文件中的 process_config 函数，发现配置文件路径如下：
/etc/vold.fstab
android2.2 下/etc 目录指向了/system/etc 目录，因此我们要新建一个 vold.fstab 文件，目标路径为/system/etc/vold.fstab

4,vold.fstab 文件的写法，参考/system/core/rootdir/etc 目录下的 vold.fstab,里面有详细的说明和例子，写法如下：
dev_mount <label> <mount_point> <part> <sysfs_path1...>
dev_mount 命令 标签 挂载点 子分区 设备在 sysfs 文件系统下的路径(可多个)
按照上面的要求和我的平台的实际情况，在 vold.fstab 中添加如下内容：
dev_mount sdcard /mnt/sdcard auto /block/mmcblk0

上面的/block/mmcblk0 表示 sysfs 下的路径，由于 linux 的 sysfs 文件系统是在 sys 目录下，所以对应到/sys/block/mmcblk0 目录

5，完成后发现 android 系统中 sdcard 可用了，总结下载，sdcard 部分很简单，主要是找到 sdcard 设备对应的 sysfs 文件系统路径

android 系统开发(九)-Audio 部分准备工作 alsa 移植

1,audio 内核部分的开发,包含 codec 驱动，cpu 的声音驱动和内核 alsa 驱动等，
这个是完全的嵌入式 linux 的开发内容，这里跳过，请确保这几部分正确。

2,从 alsa 的官方网站下载最新的 alsa-lib-1.0.23 和 alsa-utils-1.0.23
官方网站：http://www.alsa-project.org
alsa-lib 基于内核 alsa 驱动，它将对内核 alsa 接口的操作封装成 libasound 库
alsa-utils 是一个工具包，基于 alsa-lib 来控制底层 alsa 驱动，包含 aplay/amixer/control 等工具
alsa 的系统架构如下：

```
alsa 应用
  |
alsa-utils
  |
alsa-lib
  |
alsa-driver
```

alsa-driver 已经集成在 linux 内核中，alsa 应用直接调用 alsa-utils 工具包的工具来控制底层驱动以操作声卡

3,在 vendor/ardent/merlin 目录下新建一个 alsa 目录，然后将下载的 alsa-lib-1.0.23 和 alsa-utils-1.0.23
解压缩到 alsa 目录下，将解压缩后的文件夹去掉版本号改成 alsa-lib 和 alsa-utils

4,在 vendor/ardent/merlin/AndroidBoard.mk 文件中加入如下内容：

```
L_PATH := $(LOCAL_PATH)
include $(L_PATH)/alsa/Mdroid.mk
```

5,在 vendor/ardent/merlin/alsa 目录下新建 Mdroid.mk 文件，内容如下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
ALSA_PATH := $(LOCAL_PATH)
include $(ALSA_PATH)/alsa-lib/Mdroid.mk
include $(ALSA_PATH)/alsa-utils/Mdroid.mk
```

6,在 vendor/ardent/merlin/alsa/alsa-lib 目录下新建 Mdroid.mk 文件，内容如下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
include $(LOCAL_PATH)/src/Mdroid.mk
```

7,在 vendor/ardent/merlin/alsa/alsa-lib/src 目录下新建 Mdroid.mk 文件，内容在文章后面。

注：alsa-lib 中编译的内容很多，我们可以先将 alsa-lib 当成普通的 linux 库来编译，编译完成后通过查找 lo 文件的方法
看那些文件被编译到了，同而找到需要编译的 c 文件，通过 make install 到指定目录找到需要复制的库和其它文件。代码中
的很多部分是不需要用到了，目前暂时未作详细处理，alsa-lib/modules/mixer/simple 目录下的内容编译成了另外的几个
动态库(smixer-ac97,smixer-hda.so,smixer-sbase.so),alsa-lib/aserver 目录下的内容编译成 aserver,
这两部分因为不会用到，所以未加入到 android 编译系统中。

8,找个目录将 alsa-lib 当成普通的 linux 库编译一次，在 include 目录下会生成 config.h 文件，将该文件复制到
vendor/ardent/merlin/alsa/alsa-lib/include 目录下并修改 config.h 的部分内容如下：

```
#define ALOAD_DEVICE_DIRECTORY "/dev/snd"
#define ALSA_CONFIG_DIR "/etc"
#define ALSA_DEVICE_DIRECTORY "/dev/snd/"
//#define HAVE_WORDEXP_H 1
//#define VERSIONED_SYMBOLS
```

9,修改 alsa-lib/include/global.h 文件，删除如下内容：

```
#if !defined(_POSIX_C_SOURCE) && !defined(_POSIX_SOURCE)
struct timeval {
    time_t      tv_sec;     /* seconds */
    long        tv_usec;    /* microseconds */
};

struct timespec {
    time_t      tv_sec;     /* seconds */
    long        tv_nsec;    /* nanoseconds */
};
#endif
```

10,将源代码中所有的

```
#include <sys/shm.h>
```

改成

```
#include <linux/shm.h>
```

类似

```
#include <sys/sem.h>
```

改成
#include <linux/sem.h>

11,修改 alsa-lib/src/alisp/alisp.c，在 obj_type_str 函数最后面位置加上如下内容：
return NULL;

12,将 alsa-lib 当普通 linux 库编译时 alsa-lib/src/control 目录下生成的 ctl_symbols_list.c 文件和 alsa-lib/src/pcm 目录下生成的 pcm_symbols_list.c 文件复制到 android 中 alsa-lib 对应位置。

13,修改 alsa-lib/src/pcm/pcm_direct.c 文件，删除如下内容：
```
union semun {
    int             val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO (Linux specific) */
};
```

14,查找 alsa-lib 源代码所有文件，确保
#include <linux/shm.h>的前面有
#include <stdlib.h>
没有的自己加上，否则会报告错误说 size_t 未定义

15,修改 alsa-lib/src/pcm/pcm_ladspa.c 文件，将
*strrchr (labellocale, '.') = *lc->decimal_point;
改成
*strrchr (labellocale, '.') = ".";
屏蔽掉如下内容：
//lc = localeconv ();
这个是因为 android 用的 C 库是 bionic,和标准 C 库不同，对应的 locale.h 文件中的 lconv 结构体定义不同所导致。

16, 修改 alsa-lib/src/pcm/pcm_mmap.c 文件中的 snd_pcm_mmap 函数，将 switch (i->type) 语句下 SND_PCM_AREA_SHM 分支的内容
屏蔽掉，同时修改该文件中 snd_pcm_munmap 函数，将 switch (i->type)语句下的 SND_PCM_AREA_SHM 分支内容屏蔽掉。

17,搜索 alsa-lib/src 目录下的所有文件，搜索 shmctl，shmget，shmat，shmdt 等 4 个函数的调用处，将调用到的地方删除。
这个主要是因为 android 的 bionic libc 库不支持 System V IPC 所导致，具体的可以从头文件中看出来 。System V IPC 通过共享
内存的方式来实现，GNU C 库对应共享内存头文件为 linux pc 的/usr/include/sys/shm.h 文件，在此文件中，你可以看到
shmctl，shmget，shmat，shmdt 等 4 个函数的声明，bionic libc 库也有一个同样的头文件，在 android 源代码目录的
bionic/libc/kernel/common/linux 目录下，但是文件中的内容却没有上面 4 个函数的声明。上面 16 所作的修改也是基于这个原因。

18,按照 16 和 17 的结论，由于 bionic libc 所引发的 System V IPC 功能的缺失，导致 alsa 库中的相关功能不能正常实现，所以最好的
方法是将相关的部分不编译进来，以免找成不必要的错误。据此将一些文件从编译中删除，修改 alsa-lib/src/Mdroid.mk 文件即可
alsa-lib/src/control/control_shm.c
alsa-lib/src/pcm/pcm_direct.c
alsa-lib/src/pcm/pcm_dmix.c
alsa-lib/src/pcm/pcm_dshare.c
alsa-lib/src/pcm/pcm_dsnoop.c
alsa-lib/src/pcm/pcm_ladspa.c
alsa-lib/src/pcm/pcm_shm.c
alsa-lib/src/shmarea.c
删除了这几个模块后要将 alsa-lib/src/control 目录下的 ctl_symbols_list.c 文件和
alsa-lib/src/pcm 目录下的 pcm_symbols_list.c 文件中的相关内容删除，否则会编译不过。

19,最后要实现的功能当然是复制 alsa-lib 的配置文件了，在 alsa-lib/src/conf 目录下，复制操作在 alsa-lib/src/Mdroid.mk 中实现，
最终的 Mdroid.mk 文件内容如下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
    async.c conf.c confmisc.c dlmisc.c error.c input.c \
    names.c output.c socket.c userfile.c \
    alisp/alisp.c \
    control/cards.c control/control.c control/control_ext.c \
    control/control_hw.c control/control_symbols.c \
    control/hcontrol.c control/namehint.c control/setup.c control/tlv.c \
    hwdep/hwdep.c hwdep/hwdep_hw.c hwdep/hwdep_symbols.c \
    mixer/bag.c mixer/mixer.c mixer/simple.c mixer/simple_abst.c mixer/simple_none.c \
    pcm/atomic.c pcm/interval.c pcm/mask.c pcm/pcm.c pcm/pcm_adpcm.c \
    pcm/pcm_alaw.c pcm/pcm_asym.c pcm/pcm_copy.c pcm/pcm_empty.c \
    pcm/pcm_extplug.c pcm/pcm_file.c pcm/pcm_generic.c pcm/pcm_hooks.c \
    pcm/pcm_hw.c pcm/pcm_iec958.c pcm/pcm_ioplug.c \
    pcm/pcm_lfloat.c pcm/pcm_linear.c pcm/pcm_meter.c pcm/pcm_misc.c \
    pcm/pcm_mmap.c pcm/pcm_mmap_emul.c pcm/pcm_mulaw.c pcm/pcm_multi.c \
    pcm/pcm_null.c pcm/pcm_params.c pcm/pcm_plug.c pcm/pcm_plugin.c \
    pcm/pcm_rate.c pcm/pcm_rate_linear.c pcm/pcm_route.c pcm/pcm_share.c \
    pcm/pcm_simple.c pcm/pcm_softvol.c pcm/pcm_symbols.c \
    rawmidi/rawmidi.c rawmidi/rawmidi_hw.c rawmidi/rawmidi_symbols.c \
    rawmidi/rawmidi_virt.c \
    seq/seq.c seq/seq_event.c seq/seq_hw.c seq/seqmid.c \
    seq/seq_midi_event.c seq/seq_old.c seq/seq_symbols.c \
    timer/timer.c timer/timer_hw.c timer/timer_query.c \
    timer/timer_query_hw.c timer/timer_symbols.c
```

```
LOCAL_C_INCLUDES += \
        $(LOCAL_PATH)/../include

LOCAL_SHARED_LIBRARIES := libdl

LOCAL_ARM_MODE := arm

LOCAL_PRELINK_MODULE := false
LOCAL_MODULE := libasound
include $(BUILD_SHARED_LIBRARY)

TARGET_ALSA_CONF_DIR := $(TARGET_OUT)/usr/share/alsa
LOCAL_ALSA_CONF_DIR  := $(LOCAL_PATH)/conf

copy_from := \
    alsa.conf \
    pcm/dsnoop.conf \
    pcm/modem.conf \
    pcm/dpl.conf \
    pcm/default.conf \
    pcm/surround51.conf \
    pcm/surround41.conf \
    pcm/surround50.conf \
    pcm/dmix.conf \
    pcm/center_lfe.conf \
    pcm/surround40.conf \
    pcm/side.conf \
    pcm/iec958.conf \
    pcm/rear.conf \
    pcm/surround71.conf \
    pcm/front.conf \
    cards/aliases.conf

copy_to   := $(addprefix $(TARGET_ALSA_CONF_DIR)/,$(copy_from))
copy_from := $(addprefix $(LOCAL_ALSA_CONF_DIR)/,$(copy_from))

$(copy_to) : $(TARGET_ALSA_CONF_DIR)/% : $(LOCAL_ALSA_CONF_DIR)/% | $(ACP)
    $(transform-prebuilt-to-target)

ALL_PREBUILT += $(copy_to)
```

20,alsa-utils 的移植方法也类似，这里就不再介绍，上面的过程只是体验了一下 android 下开源库的移植方法，

实际上 google 服务器上已经有 alsa 的代码，直接下载下载便可用，下载方法如下：

git clone git://android.git.kernel.org/platform/external/alsa-lib.git

git clone git://android.git.kernel.org/platform/external/alsa-utils.git

将下载的 alsa-lib 和 alsa-utils 部分复制到 vendor/ardent/merlin/alsa 目录下参考上面的方法只需对以上的 4 和 5 部分稍作修改即可。

android 系统开发编译过程中的汇编错误

android 系统开发移植 alsa-lib 库的过程中编译的时候出现了如下的错误：
错误 1
/tmp/cckyaR40.s: Assembler messages:
/tmp/cckyaR40.s:2763: Error: selected processor does not support `mrs ip,cpsr'
/tmp/cckyaR40.s:2764: Error: unshifted register required -- `orr r2,ip,#128'
/tmp/cckyaR40.s:2765: Error: selected processor does not support `msr cpsr_c,r2'
/tmp/cckyaR40.s:2777: Error: selected processor does not support `msr cpsr_c,ip'
/tmp/cckyaR40.s:2945: Error: selected processor does not support `mrs r3,cpsr'
/tmp/cckyaR40.s:2946: Error: unshifted register required -- `orr r2,r3,#128'
/tmp/cckyaR40.s:2947: Error: selected processor does not support `msr cpsr_c,r2'
/tmp/cckyaR40.s:2959: Error: selected processor does not support `msr cpsr_c,r3'
/tmp/cckyaR40.s:3551: Error: selected processor does not support `mrs ip,cpsr'
/tmp/cckyaR40.s:3552: Error: unshifted register required -- `orr r1,ip,#128'
/tmp/cckyaR40.s:3553: Error: selected processor does not support `msr cpsr_c,r1'
/tmp/cckyaR40.s:3564: Error: selected processor does not support `msr cpsr_c,ip'
字面的意思报的是汇编错误，选择的处理器不支持 mrs 和 msr 指令。
原来的 ARM 指令有 32 位和 16 位两种指令模式，16 位为 thumb 指令集，thumb 指令集编译出的代码占用空间小，
而且效率也高，所以 android 的 arm 编译器默认用的是 thumb 模式编译，问题在于 alsa 的代码中有部分的内容
用到了 32 位的指令，所以才会报如下的错误，修改的方法也很简单，在 Android.mk 中加入如下内容即可：
LOCAL_ARM_MODE := arm
android 的编译系统中 LOCAL_ARM_MODE 变量的取值为 arm 或者 thumb，代表 32 位和 16 位两种 arm 指令集，
默认为 thumb

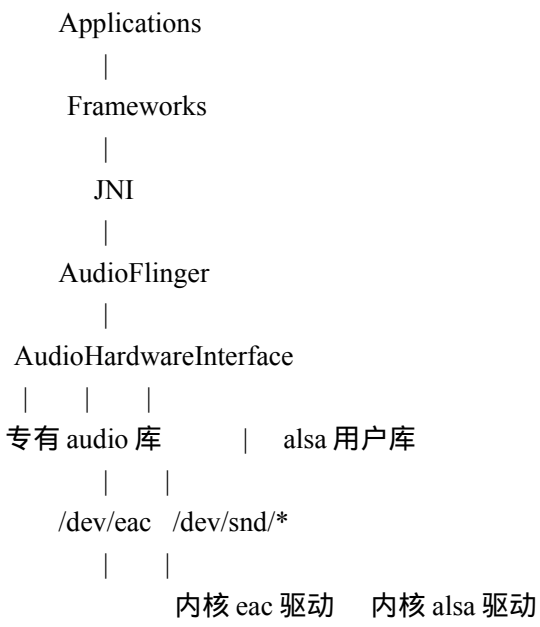
错误 2
target SharedLib: libasound (out/target/product/merlin/obj/SHARED_LIBRARIES/libasound_intermediates/LINKED/libasound.so)
/work/android-froyo-r3/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/../lib/gcc/arm-eabi/4.4.0/../../../../arm-eabi/bin/ld: out/target/product/merlin/obj/SHARED_LIBRARIES/libasound_intermediates/LINKED/libasound.so: version node not found for symbol snd_pcm_sw_params_get_start_threshold@ALSA_0.9
/work/android-froyo-r3/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/../lib/gcc/arm-eabi/4.4.0/../../../../arm-eabi/bin/ld: failed to set dynamic section sizes: Bad value

collect2: ld returned 1 exit status
make: *** [out/target/product/merlin/obj/SHARED_LIBRARIES/libasound_intermediates/LINKED/libasound.so] 错误 1
解决此问题将 alsa-lib/include/config.h 文件中的如下宏定义去掉即可：
#define VERSIONED_SYMBOLS

android 系统开发(十)-audio 移植

1,移植基础：
(1)内核声音驱动和 alsa 驱动
(2)alsa-lib 和 alsa-utils 库移植
这两部分上一节已经介绍过了。

2,android 的 audio 最核心的部分是 audioflinger，audioflinger 向上处理来自于应用程序的声音相关的所有请求
向下通过 AudioHardwareInterface 访问硬件，android 的 audio 架构如下所示：



AudioHardwareInterface 是 audioflinger 和硬件驱动之间的桥梁，android 默认编译的是 generic audio，此时 AudioHardwareInterface 直接指向了/dev/eac 驱动，它通过 eac 驱动来操作声卡，android audio 移植就是要让
AudioHardwareInterface 直接或者间接指向我们自己定义的声音驱动，一般都采用 alsa 声音体系，所以我们的目的就是
要让 AudioHardwareInterface 指向 alsa 用户库。下面的内容开始移植 alsa-audio

3,修改 vendor/ardent/merlin/BoardConfig.mk 文件内容如下：
BOARD_USES_GENERIC_AUDIO := false

BOARD_USES_ALSA_AUDIO := true
BUILD_WITH_ALSA_UTILS := true
上面配置的目的就是为了让要让 AudioHardwareInterface 指向 alsa 用户库

4,下面来添加 audio 库的编译
在 vendor/ardent/merlin 目录下新建一个 libaudio 目录,修改 AndroidBoard.mk 文件，添加编译路径如下：
LOCAL_PATH := $(call my-dir)
L_PATH := $(LOCAL_PATH)
include $(L_PATH)/libaudio/Mdroid.mk

5,vendor/ardent/merlin/libaudio 目录下新建一个 Mdroid.mk 文件，内容如下：
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := libaudio

LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libmedia \
    libhardware

LOCAL_SRC_FILES += AudioHardwareMerlin.cpp

LOCAL_CFLAGS +=

LOCAL_C_INCLUDES +=

LOCAL_STATIC_LIBRARIES += libaudiointerface

include $(BUILD_SHARED_LIBRARY)

6,android audio 的实现方法，我们现看看接口部分，上面有说道 audioflinger 是通过 AudioHardwareInterface 指向驱动的
AudioHardwareInterface 类的代码在 frameworks/base/libs/audioflinger/AudioHardwareInterface.cpp 文件中
该文件中的 create 函数中定义了 AudioHardwareInterface 指向驱动的代码如下：
```
    AudioHardwareInterface* hw = 0;
    char value[PROPERTY_VALUE_MAX];

#ifdef GENERIC_AUDIO
    hw = new AudioHardwareGeneric();
#else
    // if running in emulation - use the emulator driver
    if (property_get("ro.kernel.qemu", value, 0)) {
        LOGD("Running in emulation - using generic audio driver");
```

```
        hw = new AudioHardwareGeneric();
    }
    else {
        LOGV("Creating Vendor Specific AudioHardware");
        hw = createAudioHardware();
    }
#endif
    return hw;
```

当系统为 generic audio 的时候此函数返回的是一个指向 AudioHardwareGeneric 对象的指针，其实是返回一个指向 AudioHardwareInterface 对象
的指针，因为 AudioHardwareGeneric 是 AudioHardwareInterface 的子类，继承关系如下：
AudioHardwareInterface->AudioHardwareBase->AudioHardwareGeneric
如果系统不是 generic audio，则通过调用 createAudioHardware 函数来返回一个指向一个指向 AudioHardwareInterface 对象的指针，
所以，简单的将，我们要做的事情就是实现这个函数以及它相关的内容即可。createAudioHardware 函数我们可以在
hardware/libhardware_legacy/include/hardware_legacy/AudioHardwareInterface.h 中也就是 AudioHardwareInterface
的声明中找到原型如下：

```
extern "C" AudioHardwareInterface* createAudioHardware(void);
```

7，通过 6 我们不难知道，我们实现自己的 audio 接口完全可以模仿 generic audio 的做法，只是要多实现一个 createAudioHardware 函数而已，
因此我们将 frameworks/base/libs/audioflinger/AudioHardwareInterface.h 文件复制到
vendor/ardent/merlin/libaudio 目录下，改名为 AudioHardwareMerlin.h，然后将此文件中所有的 Generic 字段通通替换成 Merlin
然后将 frameworks/base/libs/audioflinger/AudioHardwareInterface.cpp 复制到
vendor/ardent/merlin/libaudio 目录下，改名为 AudioHardwareMerlin.cpp，然后将此文件中所有的 Generic 字段通通替换成 Merlin
最后在 AudioHardwareMerlin.cpp 中定义 createAudioHardware 函数如下：

```
extern "C" AudioHardwareInterface* createAudioHardware(void)
{
    return new AudioHardwareMerlin();
}
```

8,进行到 7 后直接编译，发现编译不过，错误如下

```
target thumb C++: libaudioflinger <= frameworks/base/libs/audioflinger/AudioFlinger.cpp
```

make: *** 没有规则可以创建“ out/target/product/merlin/obj/SHARED_LIBRARIES/libaudioflinger_intermediates/LINKED/libaudioflinger.so ”需要的目标“out/target/product/merlin/obj/lib/libaudiopolicy.so”。 停止。
原来是编译 audioflinger 的时候需要 libaudiopolicy.so 的支持
查看 frameworks/base/libs/audioflinger/Android.mk 文件，发现有如下内容：

```
ifeq ($(strip $(BOARD_USES_GENERIC_AUDIO)),true)
  LOCAL_STATIC_LIBRARIES += libaudiointerface libaudiopolicybase
  LOCAL_CFLAGS += -DGENERIC_AUDIO
else
```

```
  LOCAL_SHARED_LIBRARIES += libaudio libaudiopolicy
endif
```

看来 generic audio 的时候需要的是 libaudiointerface 和 libaudiopolicybase 静态库，否则需要 libaudio 和 libaudiopolicy 动态库

libaudio 库上面我们已经实现，看来下面的内容就是要实现 libaudiopolicy 库了

9,audio policy 接口的调用在 frameworks/base/libs/audioflinger/AudioPolicyService.cpp 文件中的 AudioPolicyService 类

的构造函数中，如下：

```
#if (defined GENERIC_AUDIO) || (defined AUDIO_POLICY_TEST)
    mpPolicyManager = new AudioPolicyManagerBase(this);
    LOGV("build for GENERIC_AUDIO - using generic audio policy");
#else
    // if running in emulation - use the emulator driver
    if (property_get("ro.kernel.qemu", value, 0)) {
        LOGV("Running in emulation - using generic audio policy");
        mpPolicyManager = new AudioPolicyManagerBase(this);
    }
    else {
        LOGV("Using hardware specific audio policy");
        mpPolicyManager = createAudioPolicyManager(this);
    }
#endif
```

该目录下的 AudioPolicyService.h 文件中定义了 mpPolicyManager 如下：

```
    AudioPolicyInterface* mpPolicyManager;          // the platform specific policy manager
```

可见，当系统为 generic audio 或者运行在模拟器上时，mpPolicyManager 是一个指向 AudioPolicyManagerBase 对象的指针

否则就要通过 createAudioPolicyManager 函数来返回。

AudioPolicyInterface 类和 AudioPolicyManagerBase 类声明在 hardware/libhardware_legacy/include/hardware_legacy

目录下的 AudioPolicyInterface.h 和 AudioPolicyManagerBase.h 文件中，而且 AudioPolicyManagerBase 类是 AudioPolicyInterface

的子类。

10，实现 libaudiopolicy 库

libaudiopolicy 库的实现我们也可以模仿 generic audio 的实现方式，从 8 我们可以看出，generic audio 的时候 audiopolicy 用的是

静态的 libaudiopolicybase 库，从 frameworks/base/libs/audioflinger/Android.mk 文件可以找到该静态库的编译内容如下：

```
include $(CLEAR_VARS)

LOCAL_SRC_FILES:=               \
    AudioPolicyManagerBase.cpp

LOCAL_SHARED_LIBRARIES := \
    libcutils \
```

```
    libutils \
    libmedia

ifeq ($(TARGET_SIMULATOR),true)
 LOCAL_LDLIBS += -ldl
else
 LOCAL_SHARED_LIBRARIES += libdl
endif

LOCAL_MODULE:= libaudiopolicybase

ifeq ($(BOARD_HAVE_BLUETOOTH),true)
  LOCAL_CFLAGS += -DWITH_A2DP
endif

ifeq ($(AUDIO_POLICY_TEST),true)
  LOCAL_CFLAGS += -DAUDIO_POLICY_TEST
endif

include $(BUILD_STATIC_LIBRARY)
```

由此可见，libaudiopolicybase 静态库编译的就是 frameworks/base/libs/audioflinger/AudioPolicyManagerBase.cpp 文件

11, 通过 9 和 10 的分析，结合 libaudio 库的写法，要完成 libaudiopolicy 库，我们可以将 AudioPolicyManagerBase.cpp
和 AudioPolicyManagerBase.h 复制到 vendor/ardent/merlin/libaudio 目录下，然后将这两个文件名改成和其中的内容作
一定修改，让它变成两外一个类如 AudioPolicyManagerMerlin 类的定义，然后在 cpp 文件中定义接口函数 createAudioPolicyManager 如下：

```
extern "C" AudioPolicyInterface* createAudioPolicyManager(AudioPolicyClientInterface *clientInterface)
{
    return new AudioPolicyManagerMerlin(clientInterface);
}
```

然后再修改相关 Mdroid.mk 文件编译成 libaudiopolicy.so 即可
采用这种方法可以实现，但是却不必要，因为 generic audio 所用的 AudioPolicyManagerBase 已经非常完善，所以我们只需要直接继承这个类即可
下面来实现它。

12,在 vendor/ardent/merlin/libaudio 目录下创建一个 AudioPolicyManagerMerlin.h 文件，内容如下：

```
#include <stdint.h>
#include <sys/types.h>
#include <utils/Timers.h>
#include <utils/Errors.h>
#include <utils/KeyedVector.h>
#include <hardware_legacy/AudioPolicyManagerBase.h>
namespace android {
```

```
class AudioPolicyManagerMerlin: public AudioPolicyManagerBase
{

public:
        AudioPolicyManagerMerlin(AudioPolicyClientInterface *clientInterface)
        : AudioPolicyManagerBase(clientInterface) {}

    virtual ~AudioPolicyManagerMerlin() {}
};
};
```

主要是声明我们所用的 AudioPolicyManagerMerlin，通过直接继承 generic audio 所用 AudioPolicyManagerBase 类来实现

13,在 vendor/ardent/merlin/libaudio 目录下创建一个 AudioPolicyManagerMerlin.cpp 文件，内容如下：

```
#include "AudioPolicyManagerMerlin.h"
#include <media/mediarecorder.h>
namespace android {
extern "C" AudioPolicyInterface* createAudioPolicyManager(AudioPolicyClientInterface *clientInterface)
{
    return new AudioPolicyManagerMerlin(clientInterface);
}
}; // namespace android
```

主要就是定义了接口函数 createAudioPolicyManager

14,修改 vendor/ardent/merlin/libaudio/Mdroid.mk 函数，添加 libaudiopolicy 的编译如下：

```
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= \
        AudioPolicyManagerMerlin.cpp
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libmedia
LOCAL_STATIC_LIBRARIES += libaudiopolicybase
LOCAL_MODULE:= libaudiopolicy
include $(BUILD_SHARED_LIBRARY)
```

经过以上过程再修改一些小错误，基本上就能编译通过，声音的框架也已经起来了，但是系统还是没哟声音，因为还需要进一步的工作，

下一节继续。

android 系统开发小知识-启动脚本文件内部的执行顺序

我们知道 android 在启动的时候通过 init 进程来解析 init.rc 和 init.xxx.rc 文件，
然后执行这两个文件解析出来的内容，init.rc 和 init.xxx.rc 文件中的内容却并不是
按照顺序来执行的，而是有固定的执行顺序，首先，init.rc 和 init.xxx.rc 文件中的内容
全部会放在 4 个关键字下：
early-init, init, early-boot, boot
所以一个典型的 rc 文件的写法如下：
on early-init
---

on init
---

on early-boot
---

on boot
---
rc 文件中这 4 个部分是可以打乱顺序随便写的，甚至可以有多个部分出现，但是解析完了以后的执行
顺序确实固定的，执行顺序如下：
early-init -> init -> early-boot -> boot