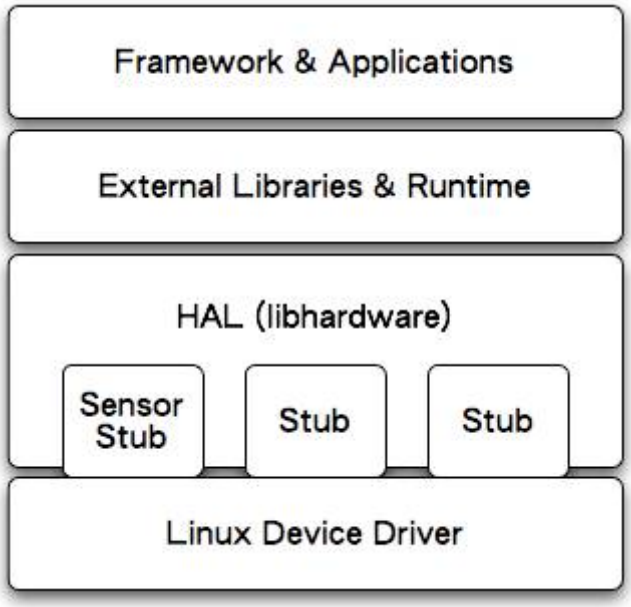
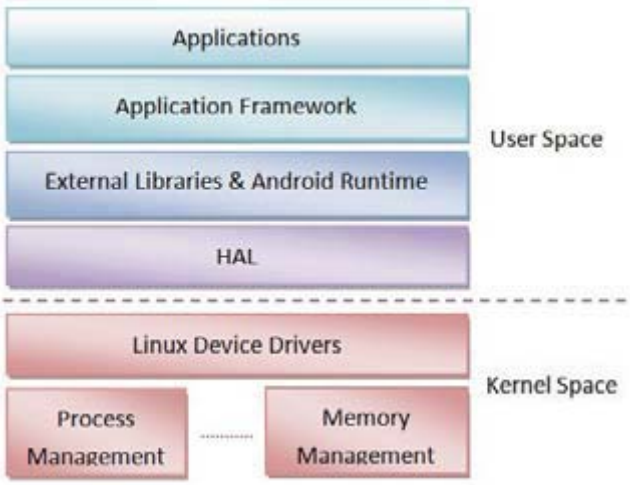
Android驱动编写及导出为服务

Android硬件抽象层（HAL）概要介绍

Android 的硬件抽象层，简单来说，就是对 Linux 内核驱动程序的封装，向上提供接口，屏蔽低层的实现细节。也就是说，把对硬件的支持分成了两层，一层放在用户空间（UserSpace），一层放在内核空间（KernelSpace），其中，硬件抽象层运行在用户空间，而 Linux 内核驱动程序运行在内核空间。为什么要这样安排呢？把硬件抽象层和内核驱动整合在一起放在内核空间不可行吗？从技术实现的角度来看，是可以的，然而从商业的角度来看，把对硬件的支持逻辑都放在内核空间，可能会损害厂家的利益。知道，Linux 内核源代码版权遵循 GNU License，而 Android 源代码版权遵循 Apache License，前者在发布产品时，必须公布源代码，而后者无须发布源代码。如果把对硬件支持的所有代码都放在 Linux 驱动层，那就意味着发布时要公开驱动程序的源代码，而公开源代码就意味着把硬件的相关参数和实现都公开了，在手机市场竞争激烈的今天，这对厂家来说，损害是非常大的。因此，Android 才会想到把对硬件的支持分成硬件抽象层和内核驱动层，内核驱动层只提供简单的访问硬件逻辑，例如读写硬件寄存器的通道，至于从硬件中读到了什么值或者写了什么值到硬件中的逻辑，都放在硬件抽象层中去了，这样就可以把商业秘密隐藏起来了。也正是由于这个分层的原因，Android 被踢出了 Linux 内核主线代码树中。Android 放在内核空间的驱动程序对硬件的支持是不完整的，把 Linux 内核移植到别的机器上去时，由于缺乏硬件抽象层的支持，硬件就完全不能用了，这也是为什么说 Android 是开放系统而不是开源系统的原因。

撇开这些争论，学习 Android 硬件抽象层，对理解整个 Android 整个系统，都是极其有用的，因为它从下到上涉及到了 Android 系统的硬件驱动层、硬件抽象层、运行时库和应用程序框架层等等，下面这个图阐述了硬件抽象层在 Android 系统中的位置，以及它和其它层的关系：





一.在Android内核源代码工程中编写硬件驱动程序。

二.在Android系统中增加C可执行程序来访问硬件驱动程序。

三.在Android硬件抽象层增加接口模块访问硬件驱动程序。

四. 在Android系统的应用程序框架层增加硬件服务接口。

五. 在Android系统中编写APP通过应用程序框架层访问硬件服务。

六. 总结思路和实现过程

一 编写驱动程序

程序位置:　　Linux 内核

一.准备好 Android 内核驱动程序开发环境。

二. 进入到 kernel/common/drivers 目录，新建 hello 目录：

USER-NAME@MACHINE-NAME:~$ cd Android_kernel/drivers/

USER-NAME@MACHINE-NAME:~ Android_kernel/drivers/$ mkdir hello

三. 在 hello 目录中增加 hello.h 文件：

```
#ifndef _HELLO_ANDROID_H_
#define _HELLO_ANDROID_H_

#include <linux/cdev.h>
#include <linux/semaphore.h>

#define HELLO_DEVICE_NODE_NAME  "hello"
#define HELLO_DEVICE_FILE_NAME  "hello"
#define HELLO_DEVICE_PROC_NAME  "hello"
#define HELLO_DEVICE_CLASS_NAME "hello"

struct hello_android_dev {
    int val;
```

```
    struct semaphore sem;
    struct cdev dev;
};

#endif
```

这个头文件定义了一些字符串常量宏，在后面要用到。此外，还定义了一个字符设备结构体 hello_android_dev，这个就是虚拟的硬件设备了，val 成员变量就代表设备里面的寄存器，它的类型为 int，sem 成员变量是一个信号量，是用同步访问寄存器 val 的，dev 成员变量是一个内嵌的字符设备，这个 Linux 驱动程序自定义字符设备结构体的标准方法。

四.在 hello 目录中增加 hello.c 文件，这是驱动程序的实现部分。驱动程序的功能主要是向上层提供访问设备的寄存器的值，包括读和写。这里，提供了三种访问设备寄存器的方法，一是通过 proc 文件系统来访问，二是通过传统的设备文件的方法来访问，三是通过 devfs 文件系统来访问。下面分段描述该驱动程序的实现。

首先是包含必要的头文件和定义三种访问设备的方法：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <linux/device.h>
#include <asm/uaccess.h>

#include "hello.h"

/*主设备和从设备号变量*/
```

```
static int hello_major = 0;

static int hello_minor = 0;


/*设备类别和设备变量*/

static struct class* hello_class = NULL;

static struct hello_android_dev* hello_dev = NULL;


/*传统的设备文件操作方法*/

static int hello_open(struct inode* inode, struct file* filp);

static int hello_release(struct inode* inode, struct file* filp);

static ssize_t hello_read(struct file* filp, char __user *buf, size_t count, loff_t*
f_pos);

static ssize_t hello_write(struct file* filp, const char __user *buf, size_t count,
loff_t* f_pos);


/*设备文件操作方法表*/

static struct file_operations hello_fops = {

    .owner = THIS_MODULE,

    .open = hello_open,

    .release = hello_release,

    .read = hello_read,

    .write = hello_write,

};


/*访问设置属性方法*/

static ssize_t hello_val_show(struct device* dev, struct device_attribute* attr,
char* buf);

static ssize_t hello_val_store(struct device* dev, struct device_attribute* attr,
const char* buf, size_t count);


/*定义设备属性*/
```

```
static DEVICE_ATTR(val, S_IRUGO | S_IWUSR, hello_val_show, hello_val_store);
```

定义传统的设备文件访问方法，主要是定义 hello_open、hello_release、hello_read 和 hello_write 这四个打开、释放、读和写设备文件的方法：

```
/*打开设备方法*/
static int hello_open(struct inode* inode, struct file* filp) {
    struct hello_android_dev* dev;

    /*将自定义设备结构体保存在文件指针的私有数据域中，以便访问设备时拿来用*/
    dev = container_of(inode->i_cdev, struct hello_android_dev, dev);
    filp->private_data = dev;

    return 0;
}

/*设备文件释放时调用，空实现*/
static int hello_release(struct inode* inode, struct file* filp) {
    return 0;
}

/*读取设备的寄存器 val 的值*/
static ssize_t hello_read(struct file* filp, char __user *buf, size_t count, loff_t* f_pos) {
    ssize_t err = 0;
    struct hello_android_dev* dev = filp->private_data;

    /*同步访问*/
    if(down_interruptible(&(dev->sem))) {
        return -ERESTARTSYS;
    }
```

```
    if(count < sizeof(dev->val)) {
        goto out;
    }

    /*将寄存器 val 的值拷贝到用户提供的缓冲区*/
    if(copy_to_user(buf, &(dev->val), sizeof(dev->val))) {
        err = -EFAULT;
        goto out;
    }

    err = sizeof(dev->val);

out:
    up(&(dev->sem));
    return err;
}

/*写设备的寄存器值 val*/
static ssize_t hello_write(struct file* filp, const char __user *buf, size_t count,
loff_t* f_pos) {
    struct hello_android_dev* dev = filp->private_data;
    ssize_t err = 0;

    /*同步访问*/
    if(down_interruptible(&(dev->sem))) {
        return -ERESTARTSYS;
    }

    if(count != sizeof(dev->val)) {
```

```
            goto out;
        }

        /*将用户提供的缓冲区的值写到设备寄存器去*/
        if(copy_from_user(&(dev->val), buf, count)) {
            err = -EFAULT;
            goto out;
        }

        err = sizeof(dev->val);

    out:
        up(&(dev->sem));
        return err;
    }
```

定义通过 devfs 文件系统访问方法，这里把设备的寄存器 val 看成是设备的一个属性，通过读写这个属性来对设备进行访问，主要是实现 hello_val_show 和 hello_val_store 两个方法，同时定义了两个内部使用的访问 val 值的方法__hello_get_val 和__hello_set_val：

```
    /*读取寄存器 val 的值到缓冲区 buf 中，内部使用*/
    static ssize_t __hello_get_val(struct hello_android_dev* dev, char* buf) {
        int val = 0;

        /*同步访问*/
        if(down_interruptible(&(dev->sem))) {
            return -ERESTARTSYS;
        }

        val = dev->val;
```

```
    up(&(dev->sem));

    return snprintf(buf, PAGE_SIZE, "%d\n", val);
}

/*把缓冲区 buf 的值写到设备寄存器 val 中去，内部使用*/
static ssize_t __hello_set_val(struct hello_android_dev* dev, const char* buf,
size_t count) {
    int val = 0;

    /*将字符串转换成数字*/
    val = simple_strtol(buf, NULL, 10);

    /*同步访问*/
    if(down_interruptible(&(dev->sem))) {
        return -ERESTARTSYS;
    }

    dev->val = val;
    up(&(dev->sem));

    return count;
}

/*读取设备属性 val*/
static ssize_t hello_val_show(struct device* dev, struct device_attribute* attr,
char* buf) {
    struct hello_android_dev* hdev = (struct
hello_android_dev*)dev_get_drvdata(dev);

    return __hello_get_val(hdev, buf);
```

```
}


/*写设备属性 val*/

static ssize_t hello_val_store(struct device* dev, struct device_attribute* attr,
const char* buf, size_t count) {

    struct hello_android_dev* hdev = (struct
hello_android_dev*)dev_get_drvdata(dev);


    return __hello_set_val(hdev, buf, count);

}
```

定义通过 proc 文件系统访问方法，主要实现了 hello_proc_read 和 hello_proc_write 两个方法，同时定义了在 proc 文件系统创建和删除文件的方法 hello_create_proc 和 hello_remove_proc：

```
/*读取设备寄存器 val 的值，保存在 page 缓冲区中*/

static ssize_t hello_proc_read(char* page, char** start, off_t off, int count, int*
eof, void* data) {

    if(off > 0) {

        *eof = 1;

        return 0;

    }


    return __hello_get_val(hello_dev, page);

}


/*把缓冲区的值 buff 保存到设备寄存器 val 中去*/

static ssize_t hello_proc_write(struct file* filp, const char __user *buff, unsigned
long len, void* data) {

    int err = 0;

    char* page = NULL;
```

```
    if(len > PAGE_SIZE) {
        printk(KERN_ALERT"The buff is too large: %lu.\n", len);
        return -EFAULT;
    }

    page = (char*)__get_free_page(GFP_KERNEL);
    if(!page) {
        printk(KERN_ALERT"Failed to alloc page.\n");
        return -ENOMEM;
    }

    /*先把用户提供的缓冲区值拷贝到内核缓冲区中去*/
    if(copy_from_user(page, buff, len)) {
        printk(KERN_ALERT"Failed to copy buff from user.\n");
        err = -EFAULT;
        goto out;
    }

    err = __hello_set_val(hello_dev, page, len);

out:
    free_page((unsigned long)page);
    return err;
}

/*创建/proc/hello 文件*/
static void hello_create_proc(void) {
    struct proc_dir_entry* entry;
```

```
    entry = create_proc_entry(HELLO_DEVICE_PROC_NAME, 0, NULL);
    if(entry) {
        entry->owner = THIS_MODULE;
        entry->read_proc = hello_proc_read;
        entry->write_proc = hello_proc_write;
    }
}

/*删除/proc/hello 文件*/
static void hello_remove_proc(void) {
    remove_proc_entry(HELLO_DEVICE_PROC_NAME, NULL);
}
```

最后，定义模块加载和卸载方法，这里只要是执行设备注册和初始化操作：

```
/*初始化设备*/
static int  __hello_setup_dev(struct hello_android_dev* dev) {
    int err;
    dev_t devno = MKDEV(hello_major, hello_minor);

    memset(dev, 0, sizeof(struct hello_android_dev));

    cdev_init(&(dev->dev), &hello_fops);
    dev->dev.owner = THIS_MODULE;
    dev->dev.ops = &hello_fops;

    /*注册字符设备*/
    err = cdev_add(&(dev->dev),devno, 1);
    if(err) {
        return err;
```

```
    }

    /*初始化信号量和寄存器 val 的值*/
    init_MUTEX(&(dev->sem));
    dev->val = 0;

    return 0;
}

/*模块加载方法*/
static int __init hello_init(void){
    int err = -1;
    dev_t dev = 0;
    struct device* temp = NULL;

    printk(KERN_ALERT"Initializing hello device.\n");

    /*动态分配主设备和从设备号*/
    err = alloc_chrdev_region(&dev, 0, 1, HELLO_DEVICE_NODE_NAME);
    if(err < 0) {
        printk(KERN_ALERT"Failed to alloc char dev region.\n");
        goto fail;
    }

    hello_major = MAJOR(dev);
    hello_minor = MINOR(dev);

    /*分配 helo 设备结构体变量*/
    hello_dev = kmalloc(sizeof(struct hello_android_dev), GFP_KERNEL);
```

```
    if(!hello_dev) {
        err = -ENOMEM;
        printk(KERN_ALERT"Failed to alloc hello_dev.\n");
        goto unregister;
    }

    /*初始化设备*/
    err = __hello_setup_dev(hello_dev);
    if(err) {
        printk(KERN_ALERT"Failed to setup dev: %d.\n", err);
        goto cleanup;
    }

    /*在/sys/class/目录下创建设备类别目录 hello*/
    hello_class = class_create(THIS_MODULE, HELLO_DEVICE_CLASS_NAME);
    if(IS_ERR(hello_class)) {
        err = PTR_ERR(hello_class);
        printk(KERN_ALERT"Failed to create hello class.\n");
        goto destroy_cdev;
    }

    /*在/dev/目录和/sys/class/hello 目录下分别创建设备文件 hello*/
    temp = device_create(hello_class, NULL, dev, "%s",
HELLO_DEVICE_FILE_NAME);
    if(IS_ERR(temp)) {
        err = PTR_ERR(temp);
        printk(KERN_ALERT"Failed to create hello device.");
        goto destroy_class;
    }
```

```
    /*在/sys/class/hello/hello 目录下创建属性文件 val*/
    err = device_create_file(temp, &dev_attr_val);
    if(err < 0) {
        printk(KERN_ALERT"Failed to create attribute val.");
        goto destroy_device;
    }

    dev_set_drvdata(temp, hello_dev);

    /*创建/proc/hello 文件*/
    hello_create_proc();

    printk(KERN_ALERT"Succedded to initialize hello device.\n");
    return 0;

destroy_device:
    device_destroy(hello_class, dev);

destroy_class:
    class_destroy(hello_class);

destroy_cdev:
    cdev_del(&(hello_dev->dev));

cleanup:
    kfree(hello_dev);

unregister:
    unregister_chrdev_region(MKDEV(hello_major, hello_minor), 1);
```

```
fail:
    return err;
}

/*模块卸载方法*/
static void __exit hello_exit(void) {
    dev_t devno = MKDEV(hello_major, hello_minor);

    printk(KERN_ALERT"Destroy hello device.\n");

    /*删除/proc/hello 文件*/
    hello_remove_proc();

    /*销毁设备类别和设备*/
    if(hello_class) {
        device_destroy(hello_class, MKDEV(hello_major, hello_minor));
        class_destroy(hello_class);
    }

    /*删除字符设备和释放设备内存*/
    if(hello_dev) {
        cdev_del(&(hello_dev->dev));
        kfree(hello_dev);
    }

    /*释放设备号*/
    unregister_chrdev_region(devno, 1);
}
```

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("First Android Driver");

module_init(hello_init);

module_exit(hello_exit);
```

五. 在 hello 目录中新增 Kconfig 和 Makefile 两个文件，其中 Kconfig 是在编译前执行配置命令 make menuconfig 时用到的，而 Makefile 是执行编译命令 make 是用到的：

Kconfig 文件的内容

```
config HELLO
    tristate "First Android Driver"
    default n
    help
    This is the first android driver.
```

Makefile 文件的内容

```
obj-$(CONFIG_HELLO) += hello.o
```

在 Kconfig 文件中，tristate 表示编译选项 HELLO 支持在编译内核时，hello 模块支持以模块、内建和不编译三种编译方法，默认是不编译，因此，在编译内核前，还需要执行 make menuconfig 命令来配置编译选项，使得 hello 可以以模块或者内建的方法进行编译。

在 Makefile 文件中，根据选项 HELLO 的值，执行不同的编译方法。

六. 修改 arch/arm/Kconfig 和 drivers/kconfig 两个文件，在 menu "Device Drivers"和 endmenu 之间添加一行：

source "drivers/hello/Kconfig"

这样，执行 make menuconfig 时，就可以配置 hello 模块的编译选项了。.

七. 修改 drivers/Makefile 文件，添加一行：

obj-$(CONFIG_HELLO) += hello/

八. 配置编译选项：

USER-NAME@MACHINE-NAME:~/Android/kernel/common$ make menuconfig

找到"Device Drivers" => "First Android Drivers"选项，设置为 y。

注意，如果内核不支持动态加载模块，这里不能选择 m，虽然在 Kconfig 文件中配置了 HELLO 选项为 tristate。要支持动态加载模块选项 必须要在配置菜单中选择 Enable loadable module support 选项；在支持动态卸载模块选项，必须要在 Enable loadable module support 菜单项中，选择 Module unloading 选项。

九. 编译：

USER-NAME@MACHINE-NAME:~/Android/kernel/common$ make

编译成功后，就可以在 hello 目录下看到 hello.o 文件了，这时候编译出来的 zImage 已经包含了 hello 驱动。

十.运行新编译的内核文件，验证 hello 驱动程序是否已经正常安装：

USER-NAME@MACHINE-NAME:~/Android$ emulator -kernel ./kernel/common/arch/arm/boot/zImage &

USER-NAME@MACHINE-NAME:~/Android$ adb shell

进入到 dev 目录，可以看到 hello 设备文件：

root@android:/ # cd dev

root@android:/dev # ls

进入到 proc 目录，可以看到 hello 文件：

root@android:/ # cd proc

root@android:/proc # ls

访问 hello 文件的值：

root@android:/proc # cat hello

0

root@android:/proc # echo '5' > hello

root@android:/proc # cat hello

5

进入到 sys/class 目录，可以看到 hello 目录：

root@android:/ # cd sys/class

root@android:/sys/class # ls

进入到 hello 目录，可以看到 hello 目录：

root@android:/sys/class # cd hello

root@android:/sys/class/hello # ls

进入到下一层 hello 目录，可以看到 val 文件：

root@android:/sys/class/hello # cd hello

root@android:/sys/class/hello/hello # ls

访问属性文件 val 的值：

root@android:/sys/class/hello/hello # cat val

5

root@android:/sys/class/hello/hello # echo '0' > val

root@android:/sys/class/hello/hello # cat val

0

至此，hello 内核驱动程序就完成了，并且验证一切正常。这里采用的是系统提供的方法和驱动程序进行交互，也就是通过 proc 文件系统和 devfs 文件系统的方法，下一节中，将通过自己编译的 C 语言程序来访问/dev/hello 文件来和 hello 驱动程序交互，敬请期待。

二 测试驱动程序

程序位置: Android 系统

一. 参照第一节，准备好 Linux 驱动程序。使用 Android 模拟器加载包含这个 Linux 驱动程序的内核文件，并且使用 adb shell 命令连接上模拟，验证在/dev 目录中存在设备文件 hello。

二. 进入到 Android 源代码工程的 external 目录，创建 hello 目录:

USER-NAME@MACHINE-NAME:~ $ cd Android_OS/external/

USER-NAME@MACHINE-NAME:~ Android_OS/external/$ mkdir hello

三. 在 hello 目录中新建 hello.c 文件：

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#define DEVICE_NAME "/dev/hello"
int main(int argc, char** argv)
{
    int fd = -1;
    int val = 0;
    fd = open(DEVICE_NAME, O_RDWR);
    if(fd == -1) {
        printf("Failed to open device %s.\n", DEVICE_NAME);
        return -1;
    }

    printf("Read original value:\n");
    read(fd, &val, sizeof(val));
    printf("%d.\n\n", val);
    val = 5;
    printf("Write value %d to %s.\n\n", val, DEVICE_NAME);
        write(fd, &val, sizeof(val));

    printf("Read the value again:\n");
        read(fd, &val, sizeof(val));
        printf("%d.\n\n", val);
    close(fd);
    return 0;
}
```

这个程序的作用中，打开/dev/hello 文件，然后先读出/dev/hello 文件中的值，接着写入值 5 到/dev/hello 中去，最后再次读出/dev/hello 文件中的值，看看是否是刚才写入的值 5。从/dev/hello 文件读写的值实际上就是虚拟的硬件的寄存器 val 的值。

四. 在 hello 目录中新建 Android.mk 文件：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := hello
LOCAL_SRC_FILES := $(call all-subdir-c-files)
include $(BUILD_EXECUTABLE)
```

注意，BUILD_EXECUTABLE 表示要编译的是可执行程序。

五.使用 mmm 命令进行编译：

```
USER-NAME@MACHINE-NAME:~/Android$ mmm ./external/hello
```

编译成功后，就可以在 out/target/product/gerneric/system/bin 目录下，看到可执行文件 hello 了。

六. 重新打包 Android 系统文件 system.img：

```
USER-NAME@MACHINE-NAME:~/Android$ make snod
```

这样，重新打包后的 system.img 文件就包含刚才编译好的 hello 可执行文件了。

七. 运行模拟器，使用/system/bin/hello 可执行程序来访问 Linux 内核驱动程序：

USER-NAME@MACHINE-NAME:~/Android$ emulator -kernel ./kernel/common/arch/arm/boot/zImage &

USER-NAME@MACHINE-NAME:~/Android$ adb shell

root@android:/ # cd system/bin

root@android:/system/bin # ./hello

Read the original value:

0.

Write value 5 to /dev/hello.

Read the value again:

5.

看到这个结果，就说编写的 C 可执行程序可以访问编写的 Linux 内核驱动程序了。

介绍完了如何使用 C 语言编写的可执行程序来访问 Linux 内核驱动程序，读者可能会问 ,能不能在 Android 的 Application Frameworks 提供 Java 接口来访问 Linux 内核驱动程序呢？可以的 ,接下来的几节中 ,将介绍如何在 Android 的 Application Frameworks 中，增加 Java 接口来访问 Linux 内核驱动程序，敬请期待。

三 硬件抽象层（HAL）

程序位置: Android 系统

一. 参照第一节所示，准备好示例内核驱动序。完成这个内核驱动程序后，便可以在 Android 系统中得到三个文件，分别是/dev/hello、/sys/class/hello/hello/val 和 /proc/hello。在本文中，将通过设备文件/dev/hello 来连接硬件抽象层模块和 Linux 内核驱动程序模块。

二. 进入到在 hardware/libhardware/include/hardware 目录 ,新建 hello.h 文件：

USER-NAME@MACHINE-NAME:~/Android$ cd hardware/libhardware/include/hardware

USER-NAME@MACHINE-NAME:~/Android/hardware/libhardware/include/hardware$ vi hello.h

hello.h 文件的内容如下：

```
#ifndef ANDROID_HELLO_INTERFACE_H
#define ANDROID_HELLO_INTERFACE_H
#include <hardware/hardware.h>

__BEGIN_DECLS

/*定义模块 ID*/
#define HELLO_HARDWARE_MODULE_ID "hello"

/*硬件模块结构体*/
struct hello_module_t {
    struct hw_module_t common;
};

/*硬件接口结构体*/
struct hello_device_t {
    struct hw_device_t common;
    int fd;
    int (*set_val)(struct hello_device_t* dev, int val);
    int (*get_val)(struct hello_device_t* dev, int* val);
};
```

```
__END_DECLS

#endif
```

这里按照 Android 硬件抽象层规范的要求，分别定义模块 ID、模块结构体以及硬件接口结构体。在硬件接口结构体中，fd 表示设备文件描述符，对应将要处理的设备文件"/dev/hello"，set_val 和 get_val 为该 HAL 对上提供的函数接口。

三. 进入到 hardware/libhardware/modules 目录 ,新建 hello 目录 ,并添加 hello.c 文件。 hello.c 的内容较多，分段来看。

首先是包含相关头文件和定义相关结构：

```
#define LOG_TAG "HelloStub"

#include <hardware/hardware.h>
#include <hardware/hello.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>

#define DEVICE_NAME "/dev/hello"
#define MODULE_NAME "Hello"
#define MODULE_AUTHOR "shyluo@gmail.com"

/*设备打开和关闭接口*/
```

```
static int hello_device_open(const struct hw_module_t* module, const char* name, struct hw_device_t** device);

static int hello_device_close(struct hw_device_t* device);


/*设备访问接口*/

static int hello_set_val(struct hello_device_t* dev, int val);

static int hello_get_val(struct hello_device_t* dev, int* val);


/*模块方法表*/

static struct hw_module_methods_t hello_module_methods = {

    open: hello_device_open

};


/*模块实例变量*/

struct hello_module_t HAL_MODULE_INFO_SYM = {

    common: {

        tag: HARDWARE_MODULE_TAG,

        version_major: 1,

        version_minor: 0,

        id: HELLO_HARDWARE_MODULE_ID,

        name: MODULE_NAME,

        author: MODULE_AUTHOR,

        methods: &hello_module_methods,

    }

};
```

这里，实例变量名必须为 HAL_MODULE_INFO_SYM，tag 也必须为 HARDWARE_MODULE_TAG，这是 Android 硬件抽象层规范规定的。

定义 hello_device_open 函数：

```
static int hello_device_open(const struct hw_module_t* module, const char*
name, struct hw_device_t** device) {

    struct hello_device_t* dev;dev = (struct
hello_device_t*)malloc(sizeof(struct hello_device_t));


    if(!dev) {

        LOGE("Hello Stub: failed to alloc space");

        return -EFAULT;

    }


    memset(dev, 0, sizeof(struct hello_device_t));

    dev->common.tag = HARDWARE_DEVICE_TAG;

    dev->common.version = 0;

    dev->common.module = (hw_module_t*)module;

    dev->common.close = hello_device_close;

    dev->set_val = hello_set_val;dev->get_val = hello_get_val;


    if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {

        LOGE("Hello Stub: failed to open /dev/hello -- %s.",
strerror(errno));free(dev);

        return -EFAULT;

    }


    *device = &(dev->common);

    LOGI("Hello Stub: open /dev/hello successfully.");


    return 0;

}
```

DEVICE_NAME 定义为"/dev/hello"。由于设备文件是在内核驱动里面通过 device_create 创建的，而 device_create 创建的设备文件默认只有 root 用户可读写，而 hello_device_open 一般是由上层 APP 来调用的，这些 APP 一般不具有 root 权限，这时候就导致打开设备文件失败：

Hello Stub: failed to open /dev/hello -- Permission denied.

解决办法是类似于 Linux 的 udev 规则，打开 Android 源代码工程目录下，进入到 system/core/rootdir 目录，里面有一个名为 ueventd.rc 文件，往里面添加一行：

/dev/hello 0666 root root

定义 hello_device_close、hello_set_val 和 hello_get_val 这三个函数：

```
static int hello_device_close(struct hw_device_t* device) {
    struct hello_device_t* hello_device = (struct hello_device_t*)device;

    if(hello_device) {
        close(hello_device->fd);
        free(hello_device);
    }

    return 0;
}

static int hello_set_val(struct hello_device_t* dev, int val) {
    LOGI("Hello Stub: set value %d to device.", val);
```

```
    write(dev->fd, &val, sizeof(val));

    return 0;
}

static int hello_get_val(struct hello_device_t* dev, int* val) {
    if(!val) {
        LOGE("Hello Stub: error val pointer");
        return -EFAULT;
    }

    read(dev->fd, val, sizeof(*val));

    LOGI("Hello Stub: get value %d from device", *val);

    return 0;
}
```

四. 继续在 hello 目录下新建 Android.mk 文件：

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

include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_PRELINK_MODULE := false

LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw

LOCAL_SHARED_LIBRARIES := liblog

LOCAL_SRC_FILES := hello.c
```

LOCAL_MODULE := hello.default

include $(BUILD_SHARED_LIBRARY)

注意，LOCAL_MODULE 的定义规则，hello 后面跟有 default，hello.default 能够保证模块总能被硬象抽象层加载到。

五. 编译：

USER-NAME@MACHINE-NAME:~/Android$ mmm hardware/libhardware/modules/hello

编译成功后，就可以在 out/target/product/generic/system/lib/hw 目录下看到 hello.default.so 文件了。

六. 重新打包 Android 系统镜像 system.img：

USER-NAME@MACHINE-NAME:~/Android$ make snod

重新打包后，system.img 就包含定义的硬件抽象层模块 hello.default 了。

虽然在 Android 系统为自己的硬件增加了一个硬件抽象层模块 ,但是现在 Java 应用程序还不能访问到硬件。还必须编写 JNI 方法和在 Android 的 Application Frameworks 层增加 API 接口，才能让上层 Application 访问硬件。在接下来的部分，还将完成这一系统过程，使得能够在 Java 应用程序中访问自己定制的硬件。

测试成功后.为了以后编译都会自动加入些模块,还需要修改 hardware/libhardware/Android.mk 文件,添加 hello 目录.

四 硬件访问服务

1 JNI部分

程序位置:　　Android 系统

一. 参照第三节，准备好硬件抽象层模块，确保 Android 系统镜像文件 system.img 已经包含 hello.default 模块。

二. 进入到 frameworks/base/services/jni 目录，新建 com_android_server_HelloService.cpp 文件：

USER-NAME@MACHINE-NAME:~/Android$ cd frameworks/base/services/jni

USER-NAME@MACHINE-NAME:~/Android/frameworks/base/services/jni$ vi com_android_server_HelloService.cpp

在 com_android_server_HelloService.cpp 文件中，实现 JNI 方法。注意文件的命令方法，com_android_server 前缀表示的是包名，表示硬件服务 HelloService 是放在 frameworks/base/services/java 目录下的 com/android/server 目录的 ,即存在一个命令为 com.android.server.HelloService 的类。这里，暂时略去 HelloService 类的描述，在下一节中，将回到 HelloService 类来。简单地说，HelloService 是一个提供 Java 接口的硬件访问服务类。

首先是包含相应的头文件：

```
#define LOG_TAG "HelloService"
#include "jni.h"
#include "JNIHelp.h"
```

```
#include "android_runtime/AndroidRuntime.h"
#include <utils/misc.h>
#include <utils/Log.h>
#include <hardware/hardware.h>
#include <hardware/hello.h>
#include <stdio.h>
```

接着定义 hello_init、hello_getVal 和 hello_setVal 三个 JNI 方法：

```
namespace android
{
    /*在硬件抽象层中定义的硬件访问结构体，参考<hardware/hello.h>*/
        struct hello_device_t* hello_device = NULL;
    /*通过硬件抽象层定义的硬件访问接口设置硬件寄存器 val 的值*/
        static void hello_setVal(JNIEnv* env, jobject clazz, jint value) {
        int val = value;
        LOGI("Hello JNI: set value %d to device.", val);
        if(!hello_device) {
            LOGI("Hello JNI: device is not open.");
            return;
        }

        hello_device->set_val(hello_device, val);
    }
        /*通过硬件抽象层定义的硬件访问接口读取硬件寄存器 val 的值*/
    static jint hello_getVal(JNIEnv* env, jobject clazz) {
        int val = 0;
        if(!hello_device) {
            LOGI("Hello JNI: device is not open.");
            return val;
```

```
        }

        hello_device->get_val(hello_device, &val);


        LOGI("Hello JNI: get value %d from device.", val);


        return val;

    }

        /*通过硬件抽象层定义的硬件模块打开接口打开硬件设备*/

    static inline int hello_device_open(const hw_module_t* module, struct
hello_device_t** device) {

        return module->methods->open(module,
HELLO_HARDWARE_MODULE_ID, (struct hw_device_t**)device);

    }

        /*通过硬件模块 ID 来加载指定的硬件抽象层模块并打开硬件*/

    static jboolean hello_init(JNIEnv* env, jclass clazz) {

        hello_module_t* module;


        LOGI("Hello JNI: initializing.....");

        if(hw_get_module(HELLO_HARDWARE_MODULE_ID, (const struct
hw_module_t**)&module) == 0) {

            LOGI("Hello JNI: hello Stub found.");

            if(hello_device_open(&(module->common), &hello_device) == 0) {

                LOGI("Hello JNI: hello device is open.");

                return 0;

            }

            LOGE("Hello JNI: failed to open hello device.");

            return -1;

        }

        LOGE("Hello JNI: failed to get hello stub module.");

        return -1;

    }
```

```
    /*JNI 方法表*/
static const JNINativeMethod method_table[] = {
    {"init_native", "()Z", (void*)hello_init},
    {"setVal_native", "(I)V", (void*)hello_setVal},
    {"getVal_native", "()I", (void*)hello_getVal},
};
    /*注册 JNI 方法*/
int register_android_server_HelloService(JNIEnv *env) {
        return jniRegisterNativeMethods(env,
"com/android/server/HelloService", method_table, NELEM(method_table));
}
};
```

注意，在 hello_init 函数中，通过 Android 硬件抽象层提供的 hw_get_module 方法来加载模块 ID 为 HELLO_HARDWARE_MODULE_ID 的硬件抽象层模块，其中，HELLO_HARDWARE_MODULE_ID 是在<hardware/hello.h>中定义的。Android 硬件抽象层会根据 HELLO_HARDWARE_MODULE_ID 的值在 Android 系统的/system/lib/hw 目录中找到相应的模块，然后加载起来，并且返回 hw_module_t 接口给调用者使用。在 jniRegisterNativeMethods 函数中，第二个参数的值必须对应 HelloService 所在的包的路径，即 com.android.server.HelloService。

三. 修改同目录下的 onload.cpp 文件，首先在 namespace android 增加 register_android_server_HelloService 函数声明：

```
namespace android {
.....
int register_android_server_HelloService(JNIEnv *env);
};
```

在 JNI_onLoad 增加 register_android_server_HelloService 函数调用：

```
extern "C" jint JNI_onLoad(JavaVM* vm, void* reserved)
{
  ..........................................................................
  register_android_server_HelloService(env);
  ..........................................................................
}
```

这样，在 Android 系统初始化时，就会自动加载该 JNI 方法调用表。

四. 修改同目录下的 Android.mk 文件，在 LOCAL_SRC_FILES 变量中增加一行：

```
LOCAL_SRC_FILES:= \
com_android_server_AlarmManagerService.cpp \
com_android_server_BatteryService.cpp \
com_android_server_InputManager.cpp \
com_android_server_LightsService.cpp \
com_android_server_PowerManagerService.cpp \
com_android_server_SystemServer.cpp \
com_android_server_UsbService.cpp \
com_android_server_VibratorService.cpp \
com_android_server_location_GpsLocationProvider.cpp \
com_android_server_HelloService.cpp \
onload.cpp
```

五. 编译和重新找亿 system.img：

```
USER-NAME@MACHINE-NAME:~/Android$ mmm
frameworks/base/services/jni
```

```
USER-NAME@MACHINE-NAME:~/Android$ make snod
```

这样，重新打包的 system.img 镜像文件就包含刚才编写的 JNI 方法了，也就是可以通过 Android 系统的 Application Frameworks 层提供的硬件服务 HelloService 来调用这些 JNI 方法，进而调用低层的硬件抽象层接口去访问硬件了。

2 Java部分

程序位置:　　Android 系统

一. 参照上节所示，为硬件抽象层模块准备好 JNI 方法调用层。

二. 在 Android 系统中，硬件服务一般是运行在一个独立的进程中为各种应用程序提供服务。因此，调用这些硬件服务的应用程序与这些硬件服务之间的通信需要通过代理来进行。为此，要先定义好通信接口。进入到 frameworks/base/core/java/android/os 目录，新增 IHelloService.aidl 接口定义文件：

USER-NAME@MACHINE-NAME:~/Android$ cd frameworks/base/core/java/android/os

USER-NAME@MACHINE-NAME:~/Android/frameworks/base/core/java/android/os$ vi IHelloService.aidl

IHelloService.aidl 定义了 IHelloService 接口：

```
package android.os;


interface IHelloService {
    void setVal(int val);
```

```
    int getVal();
}
```

IHelloService 接口主要提供了设备和获取硬件寄存器 val 的值的功能，分别通过 setVal 和 getVal 两个函数来实现。

三.返回到 frameworks/base 目录，打开 Android.mk 文件，修改 LOCAL_SRC_FILES 变量的值，增加 IHelloService.aidl 源文件：

```
## READ ME:
############################################################
    ##
    ## When updating this list of aidl files, consider if that aidl is
    ## part of the SDK API. If it is, also add it to the list below that
    ## is preprocessed and distributed with the SDK. This list should
    ## not contain any aidl files for parcelables, but the one below should
    ## if you intend for 3rd parties to be able to send those objects
    ## across process boundaries.
    ##
    ## READ ME:
############################################################
    LOCAL_SRC_FILES += /
    ..........................................................
    core/java/android/os/IVibratorService.aidl /
    core/java/android/os/IHelloService.aidl /
    core/java/android/service/urlrenderer/IUrlRendererService.aidl /
    ..........................................................
```

四. 编译 IHelloService.aidl 接口：

```
USER-NAME@MACHINE-NAME:~/Android$ mmm frameworks/base
```

这样，就会根据 IHelloService.aidl 生成相应的 IHelloService.Stub 接口。

五.进入到 frameworks/base/services/java/com/android/server 目录，新增 HelloService.java 文件：

```
package com.android.server;
import android.content.Context;
import android.os.IHelloService;
import android.util.Slog;
public class HelloService extends IHelloService.Stub {
    private static final String TAG = "HelloService";
    HelloService() {
        init_native();
    }
    public void setVal(int val) {
        setVal_native(val);
    }
    public int getVal() {
        return getVal_native();
    }

    private static native boolean init_native();
        private static native void setVal_native(int val);
    private static native int getVal_native();
};
```

HelloService 主要是通过调用 JNI 方法 init_native、setVal_native 和 getVal_native(见第四节)来提供硬件服务。

六. 修改同目录的 SystemServer.java 文件，在 ServerThread::run 函数中增加加载 HelloService 的代码：

```
    @Override
    public void run() {
    ..........................................................
.......................
          try {
               Slog.i(TAG, "DiskStats Service");
               ServiceManager.addService("diskstats", new
DiskStatsService(context));
          } catch (Throwable e) {
               Slog.e(TAG, "Failure starting DiskStats Service",
e);
          }
          try {
               Slog.i(TAG, "Hello Service");
               ServiceManager.addService("hello", new
HelloService());
          } catch (Throwable e) {
               Slog.e(TAG, "Failure starting Hello Service", e);
          }
    ..........................................................
.......................
    }
```

七. 编译 HelloService 和重新打包 system.img：

USER-NAME@MACHINE-NAME:~/Android$ mmm frameworks/base/services/java

USER-NAME@MACHINE-NAME:~/Android$ make snod

这样，重新打包后的 system.img 系统镜像文件就在 Application Frameworks 层中包含了自定义的硬件服务 HelloService 了，并且会在系统启动的时候，自动加载 HelloService。这时，应用程序就可以通过 Java 接口来访问 Hello 硬件服务了。

3 导出服务管理类

为了开放给外部 APP 使用,可以将服务导出为一个类.

在 frameworks/base/core/java/android/os 目录下添加 HelloManager.java 文件,内容包括:

1 构造函数

完成连接到服务,获取 IHelloService 接口对象.

2 功能函数:设置/获取值

调用服务接口设置获取值.

这样,外部就可以直接调用HelloManager类来与驱动交互,而不必关心驱动实现的细节,如驱动的名称,驱动的挂接等.

五 测试硬件服务

程序位置: Android 系统

一. 参照第五节，在 Application Frameworks 层定义好自己的硬件服务 HelloService，并提供 IHelloService 接口提供访问服务。

二. 为了方便开发，可以在 IDE 环境下使用 Android SDK 来开发 Android 应用程序。开发完成后，再把程序源代码移植到 Android 源代码工程目录中。使用 Eclipse 的 Android 插件 ADT 创建 Android 工程很方便，这里不述，可以参考网上其它资料。工程名称为 Hello，下面主例出主要文件：

主程序是 src/shy/luo/hello/Hello.java：

```
package shy.luo.hello;

import shy.luo.hello.R;
import android.app.Activity;
import android.os.ServiceManager;
import android.os.Bundle;
import android.os.IHelloService;
import android.os.RemoteException;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

public class Hello extends Activity implements OnClickListener {
    private final static String LOG_TAG = "shy.luo.renju.Hello";

    private IHelloService helloService = null;
```

```
    private EditText valueText = null;
    private Button readButton = null;
    private Button writeButton = null;
    private Button clearButton = null;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

    helloService = IHelloService.Stub.asInterface(
        ServiceManager.getService("hello"));

        valueText = (EditText)findViewById(R.id.edit_value);
        readButton = (Button)findViewById(R.id.button_read);
        writeButton = (Button)findViewById(R.id.button_write);
        clearButton = (Button)findViewById(R.id.button_clear);

    readButton.setOnClickListener(this);
    writeButton.setOnClickListener(this);
    clearButton.setOnClickListener(this);

        Log.i(LOG_TAG, "Hello Activity Created");
    }

    @Override
    public void onClick(View v) {
```

```
        if(v.equals(readButton)) {
        try {
                int val = helloService.getVal();
                String text = String.valueOf(val);
                valueText.setText(text);
        } catch (RemoteException e) {
            Log.e(LOG_TAG, "Remote Exception while reading value from
device.");
        }
        }
        else if(v.equals(writeButton)) {
        try {
                String text = valueText.getText().toString();
                int val = Integer.parseInt(text);
            helloService.setVal(val);
        } catch (RemoteException e) {
            Log.e(LOG_TAG, "Remote Exception while writing value to
device.");
        }
        }
        else if(v.equals(clearButton)) {
            String text = "";
            valueText.setText(text);
        }
    }
}
```

程序通过 ServiceManager.getService("hello")来获得 HelloService，接着通过 IHelloService.Stub.asInterface 函数转换为 IHelloService 接口。其中，服务名字“hello”是系统启动时加载 HelloService 时指定的，而 IHelloService 接口定义在

android.os.IHelloService 中，具体可以参考第五节。这个程序提供了简单的读定自定义硬件有寄存器 val 的值的功能，通过 IHelloService.getVal 和 IHelloService.setVal 两个接口实现。

界面布局文件 res/layout/main.xml：

```
<?xml version="1.0" encoding="utf-8"?>
    <LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:gravity="center">
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/value">
            </TextView>
            <EditText
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:id="@+id/edit_value"
                android:hint="@string/hint">
            </EditText>
        </LinearLayout>
        <LinearLayout
```

```
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal"
            android:gravity="center">
            <Button
                android:id="@+id/button_read"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/read">
            </Button>
            <Button
                android:id="@+id/button_write"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/write">
            </Button>
            <Button
                android:id="@+id/button_clear"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/clear">
            </Button>
        </LinearLayout>
    </LinearLayout>
```

字符串文件 res/values/strings.xml：

```
<?xml version="1.0" encoding="utf-8"?>
    <resources>
        <string name="app_name">Hello</string>
```

```
        <string name="value">Value</string>
        <string name="hint">Please input a value...</string>
        <string name="read">Read</string>
        <string name="write">Write</string>
        <string name="clear">Clear</string>
    </resources>
```

程序描述文件 AndroidManifest.xml：

```
<?xml version="1.0" encoding="utf-8"?>
    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="shy.luo.hello"
      android:versionCode="1"
      android:versionName="1.0">
      <application android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:name=".Hello"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
      </application>
    </manifest>
```

三. 将 Hello 目录拷贝至 packages/experimental 目录，新增 Android.mk 文件：

USER-NAME@MACHINE-NAME:~/Android/packages/experimental$

vi Android.mk

Android.mk 的文件内容如下：

LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

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

LOCAL_PACKAGE_NAME := Hello

include $(BUILD_PACKAGE)

四. 编译：

USER-NAME@MACHINE-NAME:~/Android$ mmm

packages/experimental/Hello

编译成功后，便可以在 out/target/product/generic/system/app 目录下看到 Hello.apk 文件了。

五. 重新打包系统镜像文件 system.img：

USER-NAME@MACHINE-NAME:~/Android$ make snod

重新打包后的 system.img 文件就内置了 Hello.apk 文件了。

六 总结

1 驱动层修改

添加目录 **Android_kernel/drivers/hello**,其下添加文件
hello.c 驱动实现文件
hello.h 驱动头文件
Kconfig 驱动配置文件
Makefile 驱动工程文件
修改:drivers/kconfig 添加对 hello/Kconfig 的引用
修改 Kernel 配置: **make menuconfig**,启用 hello driver
修改 drivers/Makefile,添加对 hello 的 Makefile 的引用.

驱动实现的结构:
基本有三大块:
1) /dev/hello 设备的读写
2) /proc/hello 进程文件的读写
3) /sys/class/hello/hello 文件的读写

2 驱动测试程序修改
添加目录 **Android_OS/external/hello**,其下添加文件
hello.c 测试实现文件
Android.mk 工程文件
需要手动编译: **mmm ./external/hello**

3 编写 HAL
添加 HAL 头文件:
Android_OS/hardware/libhardware/include/hardware/hello.h
添加 HAL 实现文件:
添加目录 **Android_OS/hardware/libhardware/modules/hello**,其下添加文件
hello.c HAL 实现文件
Android.mk 工程文件
编译:手动编译,或修改 **Android_OS/hardware/libhardware/Android.mk**,自动编译

4 编写服务
1) 创建接口描述文件:
Android_OS/frameworks/base/core/java/android/os/IHelloService.aidl
如果需要使用四节 3 中方法导出服务管理类,在些目录创建:HelloManager.java
[Forlinx]
作法是,创建 **Android_OS/frameworks/base/core/java/android/forlinx** 目录,在其下放置这两个文件
修改 frameworks/base/Android.mk,添加对 IHelloService.aidl 的引用(Java 文件是自动参与编译,不需要配置).
2)添加服务实现类:
Android_OS/frameworks/base/services/java/com/android/server/HelloService.java
此类有 native 方法用于访问设备,添加 JNI 实现:
Android_OS/frameworks/base/services/jni/com_android_server_HelloService.cpp
修改 **Android_OS/frameworks/base/services/jni/onload.cpp**,添加注册 HelloService 函数的代码.

[Forlinx]

作法是，在 **Android_OS/frameworks/base/services/** 目录下，直接创建 forlinux_xxx_server 目录用于存放 java 文件, forlinux_xxx_jni 目录用于存放 JNI 文件.]

5 编写测试程序

使用 Eclipse 编写后,清理下工程目录,后放置到 **Android_OS/packages/experimental/** 目录下,需要手动编译.

6 修正 could load ‘clearsilver-jin’

移除 build-android 脚本中的 export java 路径的部分.