FreeRTOS操作系统培训文档

目录

[一、嵌入式操作系统简介 2](#_Toc117753460)

[1.1什么是嵌入式操作系统 2](#_Toc117753461)

[1.2为什么要在单片机上使用嵌入式操作系统 2](#_Toc117753463)

[1.3常见的嵌入式操作系统 2](#_Toc117753466)

[二、 FreeRTOS任务及任务管理器 2](#_Toc117753467)

[2.1任务调度器 2](#_Toc117753468)

[2.2任务特性 3](#_Toc117753469)

[2.3任务状态 3](#_Toc117753470)

[2.4 API函数使用 4](#_Toc117753471)

[三、系统中的钩子函数 6](#_Toc117753472)

[3.1空闲任务中的钩子函数 6](#_Toc117753473)

[3.2 C语言中使用钩子函数 7](#_Toc117753474)

# 一、嵌入式操作系统简介

**1.1什**么是嵌入式操作系统

一般是指运行在嵌入式系统中的操作系统，一般常见于处理器上使用。可以使系统运行更流畅，响应更快，且利于开发。

### 1.2为什么要在单片机上使用嵌入式操作系统

1. 跑系统的单片机可以更有效地支持多任务，完成更加复杂的任务。
2. 系统运行过程更流畅，响应更快。

### 1.3常见的嵌入式操作系统

1. FreeRTOS:开源免费
2. Ucoss：收费
3. RT\_Thread：免费

# FreeRTOS任务及任务管理器

在这里，通常将线程统称为任务。如果处理器核心仅为一个核心的话，在同一时刻也只会执行一个任务。

### 2.1任务调度器

哪个任务先运行，哪个任务后运行。完成这个功能的东西在 RTOS 系统中叫做任务调度器。不同的系统其任务调度器的实现方法也不同，比如 FreeRTOS 是一个抢占式的实时多任务系统，其任务调度器也是抢占式的。运行状态如下图2-1。

图中高优先级的任务可以打断低优先级任务的运行而取得 CPU 的使用权，这样保证了那些紧急任务的运行。这样我们就可以为那些对实时性要求高的任务设置一个很高的优先级，高优先级的任务执行完成以后重新把 CPU 的使用权归还给低优先级的任务，这个就是抢占式多任务系统的基本原理。

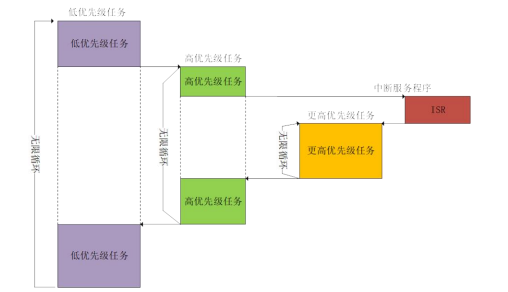


图2-1系统运行图

### 2.2任务特性

1、简单

2、没有使用限制

3、支持抢占

4、支持优先级

5、每个任务都拥有堆栈导致了 RAM 使用量增大

### 2.3任务状态

FreeRTOS 中的任务永远处于下面几个状态中的某一个

1.运行态

当一个任务正在运行时，那么就说这个任务处于运行态，处于运行态的任务就是当前正在使用处理器的任务。如果使用的是单核处理器的话那么不管在任何时刻永远都只有一个任务处于运行态。

2.就绪态

处于就绪态的任务是那些已经准备就绪(这些任务没有被阻塞或者挂起)，可以运行的任务，但是处于就绪态的任务还没有运行，因为有一个同优先级或者更高优先级的任务正在运行。

3.阻塞态

如果一个任务当前正在等待某个外部事件的话就说它处于阻塞态，比如说如果某个任务调用了函数 vTaskDelay()的话就会进入阻塞态，直到延时周期完成。任务在等待队列、信号量、事件组、通知或互斥信号量的时候也会进入阻塞态。任务进入阻塞态会有一个超时时间，当超过这个超时时间任务就会退出阻塞态，即使所等待的事件还没有来临

4.挂起态

像阻塞态一样，任务进入挂起态以后也不能被调度器调用进入运行态，但是进入挂起态的任务没有超时时间。任务进入和退出挂起态通过调用函数 vTaskSuspend()和 xTaskResume()。

系统任务状态转换如图2-2所示

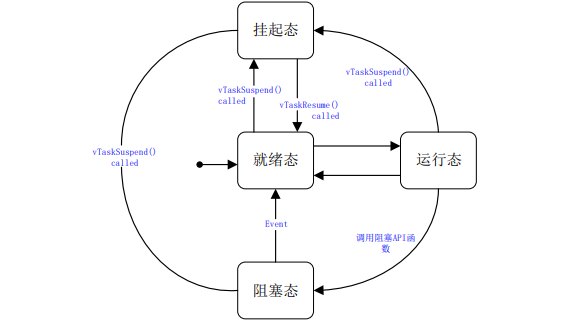


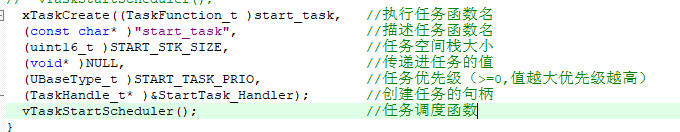
图2-2任务状态之间的转换

### 2.4 API函数使用

#### 2.4.1 任务创建函数

xTaskCreate() 动态任务创建函数（RAM自动从FreeRTOS堆中分配，必须提供内存管理文件）

xTaskCreateStatic()静态任务创建函数（用户提供RAM,一般自定义数组，并提供数组大小为堆栈大小）



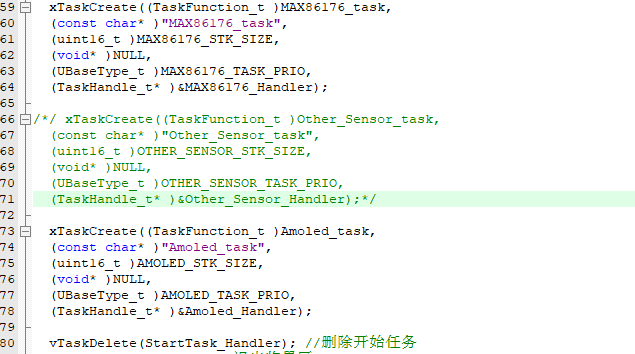
#### 2.4.2任务调度函数

vTaskStartScheduler() 任务调度函数，创建任务完成后使用该函数，便开始执行任务。

#### 2.4.3删除任务

函数原型：vTaskDelete(TaskHandle\_t xTaskToDelete)

提供创建任务时的任务句柄即可删除已创建任务



此处用为在开始任务函数内创建完成其他任务后删除开始任务

#### 2.4.4任务挂起函数

此函数用于将某个任务设置为挂起态，进入挂起态的任务永远都不会进入运行态。退出挂起态的唯一方法就是调用任务恢复函数 vTaskResume()或xTaskResumeFromISR()。函数原型如下：

void vTaskSuspend(TaskHandle\_t xTaskToSuspend)

提供创建任务时的任务句柄就可以挂起该句柄的任务，如果传入参数为NULL则表示挂起当前任务。

#### 2.4.5任务恢复函数

函数原型：void vTaskResume(TaskHandle\_t xTaskToResume) 通过传入被挂起的任务句柄使其恢复为就绪态。

函数原型：BaseType\_t xTaskResumeFromISR( TaskHandle\_t xTaskToResume)在中断里传入被挂起的任务句柄已恢复任务函数。

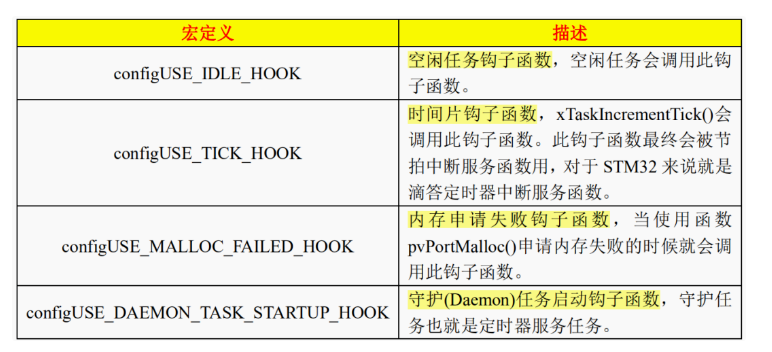
返回参数：

PdTRUE

PdFALSE

# 三、系统中的钩子函数

### 3.1空闲任务中的钩子函数



通过空闲任务钩子函数，或称回调，可以直接在空闲任务中添加应用程序相关的功能。空闲任务钩子函数会被空闲任务每循环一次就自动调用一次。当有更高优先级任务来临时，会打断空闲任务并执行主任务。

空闲任务钩子函数一般被用作以下几个地方：

1.执行低优先级，后台或需要不停处理的功能代码

2.进入低功耗模式，在没有其他任何任务需要处理时，系统自动进入低功耗模式

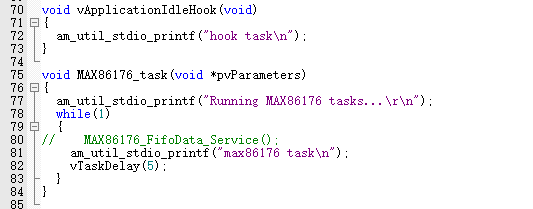
空闲任务钩子函数原型如下：  
void vApplicationIdleHook( void )

{

//需要实现的任务功能，系统会在进入空闲任务时自动调用

}

程序示例：



最终输出结果：



### 3.2 C语言中使用钩子函数

#### 3.2.1 C语言中使用钩子函数的意义

1.用于封装API函数

在封装库的时候只需提供接口，用户直接将实现函数或功能函数挂钩即可，不需要再去弄处理流程。

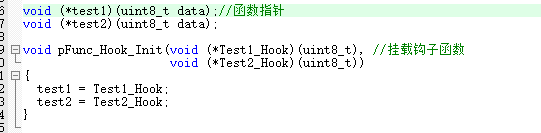
2.处理系统消息

通过设置“钩子”，应用程序可以在系统级对所有消息、事件进行过滤，访问在正常情况下无法访问的消息。钩子的本质是一段用以处理系统消息的程序，通过系统调用，把它挂入系统。

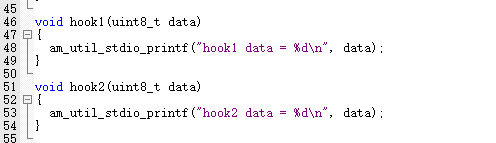
#### 3.2.2钩子函数的实现方式

钩子函数可以理解为函数指针，使用指针指向函数，后调用指针实现挂钩函数功能。

挂载钩子函数



定义两个函数



挂载钩子函数



运行钩子函数



代码运行结果



# GITHUB代理访问

1.首先用sourcetree创建本地仓库

2.配置github的地址

3.在仓库.github/config文件下加入

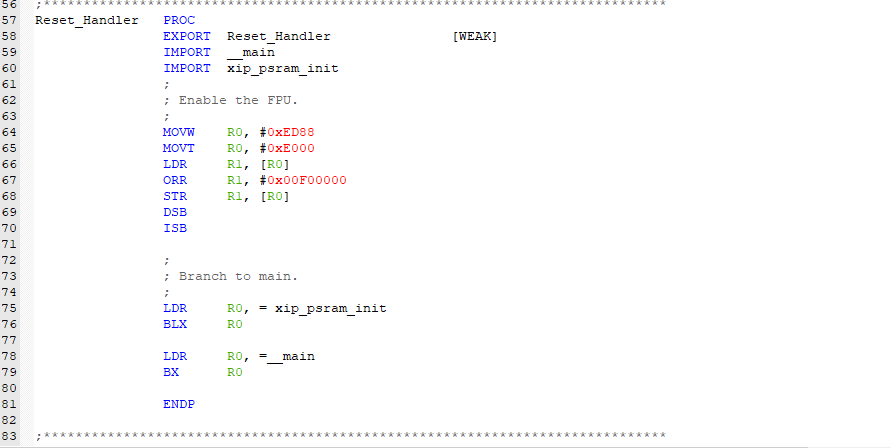
[http]

proxy = socks5://127.0.0.1:7890  
4.其中127.0.0.1:7890是你的电脑代理地址

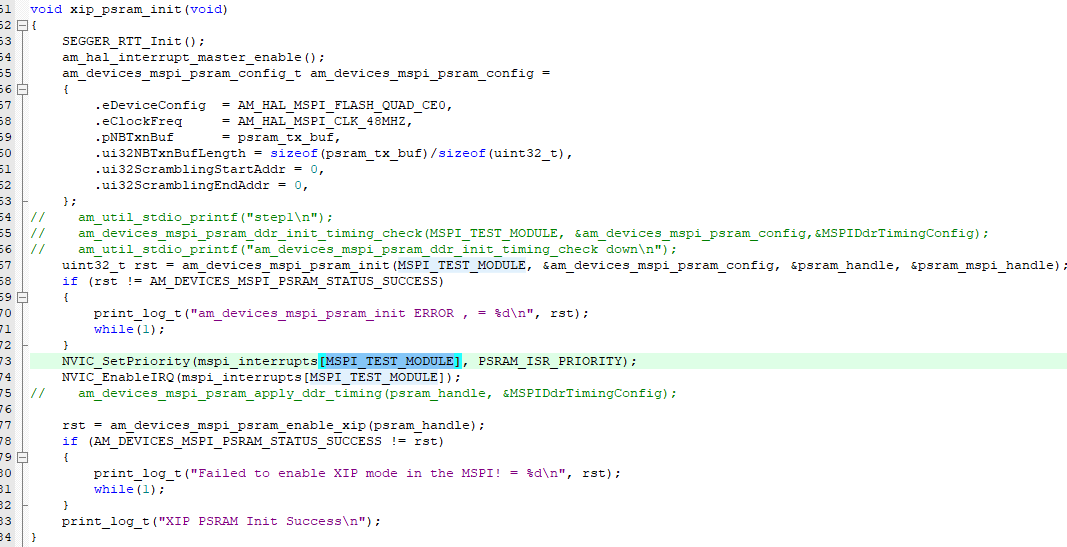
# PSRAM XIP用作堆使用的方法

1 PSRAM驱动移植并打开XIP模式

2 PSRAM的驱动以及XIP初始化的函数需要再main函数启动前进行初始化

3 即.s启动文件中初始化：如下  
 

4 因在main函数前的初始化不能使用全局变量，所以psram相关的初始化得用局部变量进行初始化，否则代码不运行

5 启动后需要在main里面再次进行一次psram驱动（驱动以及xip模式）的初始化，否则内存池可能会错误  


# 打印宏定义的办法

Void Log\_RTOS(char \*string)

{

SEGGER\_RTT\_printf(0, "%s", string);

}

#define LOG\_LENTH (128)

#define print\_log\_t(...) do {\

char chDebugStr[LOG\_LENTH] = {0};\

snprintf(chDebugStr, LOG\_LENTH, \

"[PpgAfDetect DEBUG]:" \_\_VA\_ARGS\_\_);\

Log\_RTOS(chDebugStr);\ //Log\_RTOS(char \*string)

}while(0)

#define print\_log\_func\_line(fm, ...) print\_log\_t("func=%s, Line=%d---" fm, \_\_func\_\_, \_\_LINE\_\_, ##\_\_VA\_ARGS\_\_)

#define LOGS\_I(abuf, len)  do{ \

                                LOG\_I("Buf\_Start\n"); \

                                uint8\_t i=0; \

                                for(i=0; i<len; i++) \

                                { \

                                    LOG("0x%x ", abuf[i]); \

                                } \

                                LOG("\r\n"); \

                                LOG\_I("Buf\_End\r\n"); \

                            } while (0);

# 打印嵌入式的堆占用情况

#include "stdlib.h" //包含标准C库头，此处使用该函数时不能使用micro lib c

void Printf\_Heap\_Size(void \*param, char const \*format, ...) //写内存打印函数

{

char buffer[1024];

va\_list args;

va\_start(args, format);

vsnprintf(buffer, sizeof(buffer), format, args);

va\_end(args);

am\_util\_stdio\_printf("%s", buffer);

}

Int main（void）

{

\_\_heapstats(Printf\_Heap\_Size, NULL); //调用标准C库查看当前堆所剩空间，并指向打印函数

int \*p = new int[16];//实际内存占用总是会多8个字节

\_\_heapstats(Printf\_Heap\_Size, NULL); //再次查看并验证内存使用情况

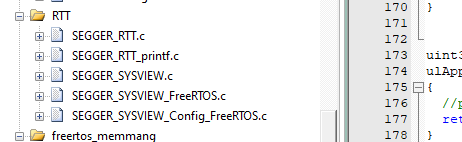
}

# SEGGER SYSTEMVIEW的使用方法

工具下载地址：

<https://www.segger.com/products/development-tools/systemview/technology/supported-rtos/>

1.下载后程序内有需要移植代码，将src目录直接移植过来在keil内添加.c文件如下：



3.Freertos系统需要在Freertos.h添加定义如下（如果Freertos\_config.h下有该宏定义需将宏定义设置为1）

#ifndef INCLUDE\_xTaskGetIdleTaskHandle

#define INCLUDE\_xTaskGetIdleTaskHandle 1

#endif

#ifndef INCLUDE\_pxTaskGetStackStart

#define INCLUDE\_pxTaskGetStackStart 1

#endif

Freertos\_config.h头文件在末尾需要添加调用#include "SEGGER\_SYSVIEW\_FreeRTOS.h"

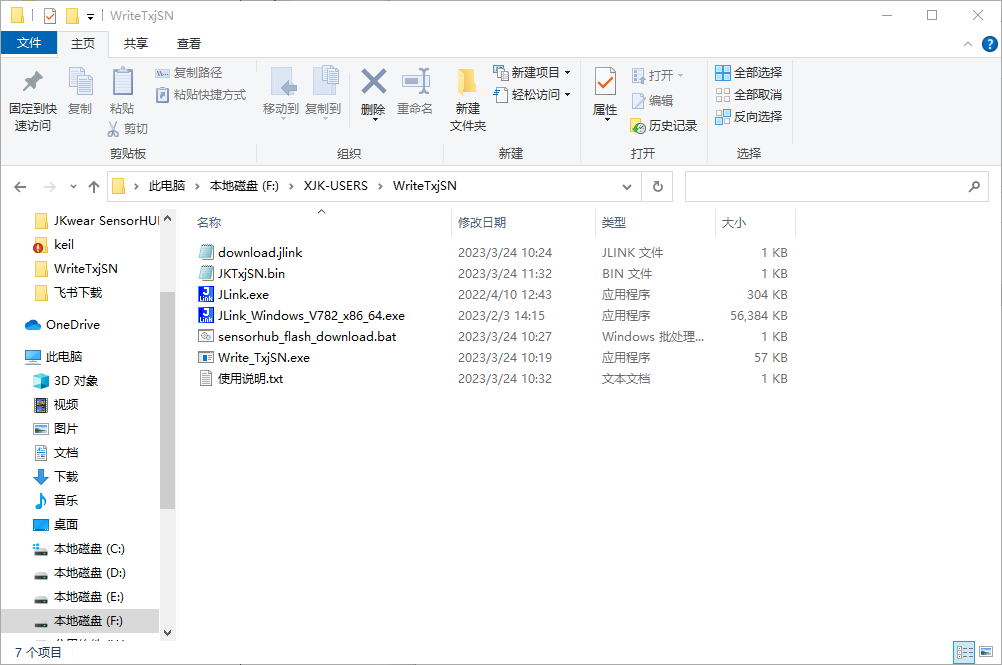
4.在main下添加调用#include "SEGGER\_SYSVIEW.h"

5.在rtos初始化前调用systmeview初始化SEGGER\_SYSVIEW\_Conf();

6.配备有打印函数SEGGER\_SYSVIEW\_Print（const char\* s）需要重写一下，否则不能传入数值参数

# 用bat脚本批量烧录Jlink程序

1.首先安装JLINK驱动，后将Jlink\_command.exe或jlink.exe复制到当前文件如下：

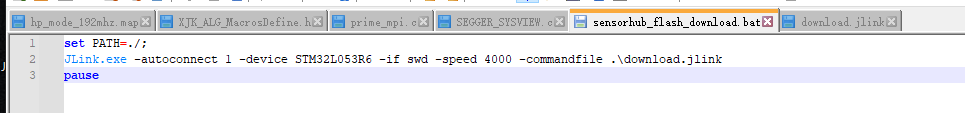


2.创建.bat脚本输入：

set PATH=./;

JLink.exe -autoconnect 1 -device STM32L053R6 -if swd -speed 4000 -commandfile .\download.jlink

Pause



4.创建.jlink文件输入：

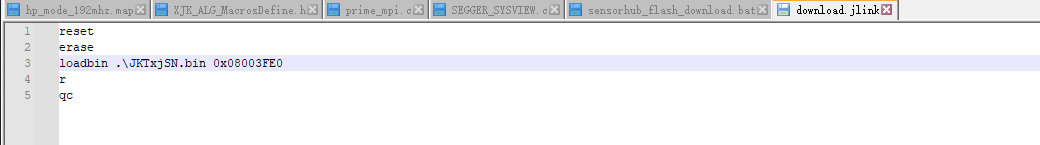
reset

erase

loadbin .\JKTxjSN.bin 0x08003FE0

r

qc



5.可以使用C语言生成EXE循环调用.bat脚本并记录烧录次数

# 常用的数据解析封装方法

static void Max86176\_Only\_Open\_PPGGreen(void);

static void Max86176\_Only\_Open\_PPGIr(void);

static void Max86176\_Open\_PPGGreenIR(void);

typedef struct

{

uint8\_t cmd\_id;

void (\*probe)();

}write\_cmd\_func\_t;

static write\_cmd\_func\_t write\_cmd\_func[] =

{

{.cmd\_id = ONLY\_OPEN\_PPG\_GREEN, .probe = Max86176\_Only\_Open\_PPGGreen,},

{.cmd\_id = ONLY\_OPEN\_PPG\_GREEN, .probe = Max86176\_Only\_Open\_PPGIr,},

{.cmd\_id = ONLY\_OPEN\_PPG\_GREEN, .probe = Max86176\_Open\_PPGGreenIR,},

};

static void Max86176\_Only\_Open\_PPGGreen(void)

{

uint8\_t tx\_buf[2]={0};

tx\_buf[0] = 0x11;

tx\_buf[1] = 0x01;

os64\_writeReg(tx\_buf);

}

static void Max86176\_Only\_Open\_PPGIr(void)

{

uint8\_t tx\_buf[2]={0};

tx\_buf[0] = 0x11;

tx\_buf[1] = 0x02;

os64\_writeReg(tx\_buf);

}

static void Max86176\_Open\_PPGGreenIR(void)

{

uint8\_t tx\_buf[2]={0};

tx\_buf[0] = 0x11;

tx\_buf[1] = 0x03;

os64\_writeReg(tx\_buf);

}

void Max86176\_Write\_Fun(Write\_fun fun)

{

for (uint8\_t i=0; i<(sizeof(write\_cmd\_func)/(sizeof(write\_cmd\_func[0]))); i++)

{

if(fun == write\_cmd\_func[i].cmd\_id)

{

write\_cmd\_func[i].probe();

}

}

}

void Max86176\_Read\_Data(max86176\_read\_data data)

{

}

void MAX86176\_Func\_Init(Max86176\_func\_test \*func)

{

func->open = Max86176\_Init;

func->write = Max86176\_Write\_Fun;

func->read = Max86176\_Read\_Data;

// func->realease //

}

# 重定义printf的方式

#ifndef \_\_GNUC\_\_

int fputc(int ch, FILE \*f)

{

  HAL\_UART\_Transmit(&huart1, (uint8\_t \*)&ch, 1, 0xFFFF);

  return ch;

}

#else

int \_write(int fd, char \*pBuffer, int size)

{

    HAL\_UART\_Transmit(&huart1, (uint8\_t \*)pBuffer, size, 0xff);

    return size;

}

#endif

# 写一个初始化注册函数的方式

全局.h头文件的宏定义方式

typedef int (\*initcall\_t)(void);

#define \_\_INITCALL\_NAME(fn, level)  \_\_initcall\_##fn##level

#define INIT\_CALL\_SECTION(level)  \_\_attribute\_\_((\_\_used\_\_,\_\_section\_\_(".initcall." #level ".init")))

#define \_DEFINE\_INITCALL(fn,level)  static initcall\_t INIT\_CALL\_SECTION(level) \_\_INITCALL\_NAME(fn,level) = fn

#define DRIVER\_INIT\_EXPORT(fn) \_DEFINE\_INITCALL(fn,0)

#define DEVICE\_INIT\_EXPORT(fn) \_DEFINE\_INITCALL(fn,1)

Main函数内遍历链表后进行初始化

extern uint32\_t  \_\_initcall\_start;

extern uint32\_t  \_\_initcall\_end;

static void Driver\_Init(void)

{

  initcall\_t fc;

  uint32\_t \*fn\_entry;

  for (fn\_entry=&\_\_initcall\_start; fn\_entry<&\_\_initcall\_end; fn\_entry++)

  {

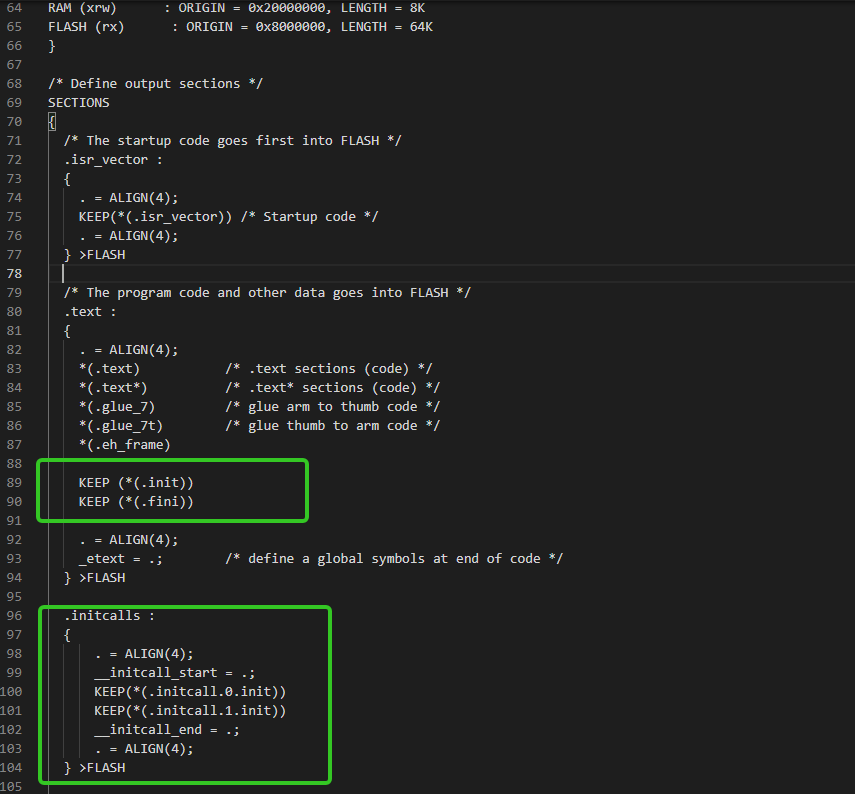
    fc = (initcall\_t \*)(\*fn\_entry);

    fc();

  }

}

修改ld/sct链表文件



# Makefile文件规则

1.批量添加源文件方法（C文件）

SRC += $(notdir $(wildcard ../userfile/tma525c/\*.c))  #添加tma525c文件目录下所有C文件，wildcard通用符意味取出所有C文件路径及名称，notdir意味只保留C文件名称

# MTK2601

## 生成差分包

1.1首先使用编译指令编译此次版本，假设为V3.1.0.0

make -j16 XJK\_BUILD\_ID=v3.1.0.0 2>&1 |tee bu1ild.log

1.2生成全量包

make otapackage

1.3全量包位置

ubuntu16.04\home\ncy\source\2601source\out\target\product\sac2601\_l\obj\PACKAGING\target\_files\_intermediates

1.4进入全量包位置目录并将全量包复制到ubuntu16.04\home\ncy\source\2601source目录下

cd out/target/product/sac2601\_l/obj/PACKAGING/target\_files\_intermediates/

cp full\_sac2601\_l-target\_files-1684809762.zip ~/source/2601source/

1.5重命名全量包

mv full\_sac2601\_l-target\_files-1684809762.zip V3100.zip

2.1在内核代码中产生了变更，假设变更版本为V3.1.0.1

make -j16 XJK\_BUILD\_ID=v3.1.0.0 2>&1 |tee bu1ild.log

2.2生成全量包

make otapackage

2.3进入全量包位置目录并将全量包复制到ubuntu16.04\home\ncy\source\2601source目录下

cd out/target/product/sac2601\_l/obj/PACKAGING/target\_files\_intermediates/

cp full\_sac2601\_l-target\_files-1684809762.zip ~/source/2601source/

2.4重命名全量包

mv full\_sac2601\_l-target\_files-1684809762.zip V3101.zip

3.1 生成差分包

./build/tools/releasetools/ota\_from\_target\_files -block -k device/mediatek/common/security/sac2601\_l/releasekey -i V3100.zip V3101.zip V3100-V3101.zip

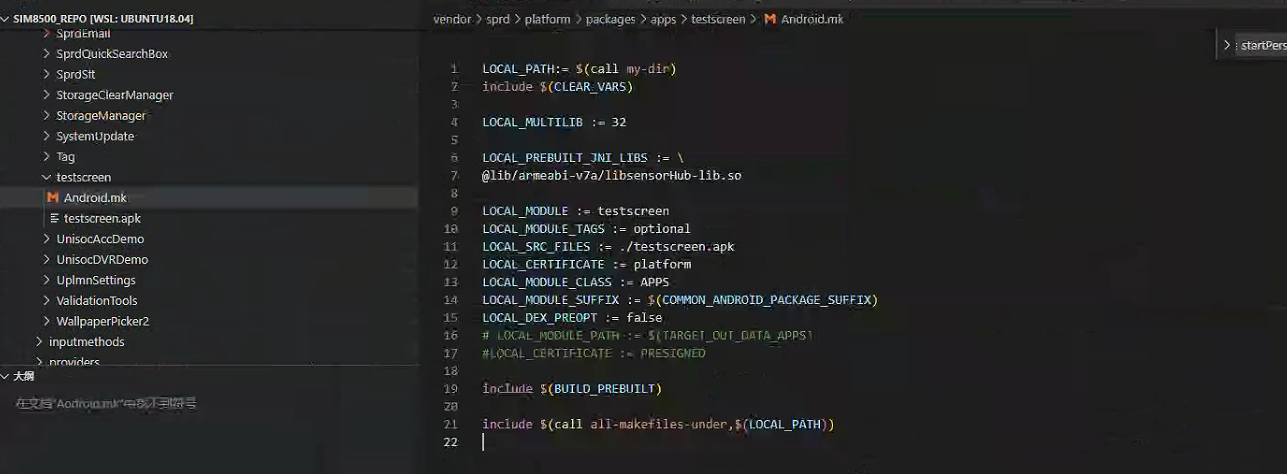
3.2差分包所在位置

ubuntu16.04\home\ncy\source\2601source

# SIM8500

## 预装APP

1. 在文件目录为vendor/sprd/platform/packages/apps/.目录下创建一个your app的目录如我的为testscreen，该目录应该包含一个Android.mk文件以及apk文件。如下图



1. Android.mk代码为：

LOCAL\_PATH:= $(call my-dir) /\*这为必须包含的路径

include $(CLEAR\_VARS) \*/

LOCAL\_MULTILIB := 32 //so库编译为32位则为32，64位则为64

LOCAL\_PREBUILT\_JNI\_LIBS := \

@lib/armeabi-v7a/libsensorHub-lib.so //包含的SO库

LOCAL\_MODULE := testscreen //创建的app module名称(在系统中此app名称)

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES := ./testscreen.apk //目录下的apk文件

LOCAL\_CERTIFICATE := platform //进行了系统签名的app

LOCAL\_MODULE\_CLASS := APPS

LOCAL\_MODULE\_SUFFIX := $(COMMON\_ANDROID\_PACKAGE\_SUFFIX)

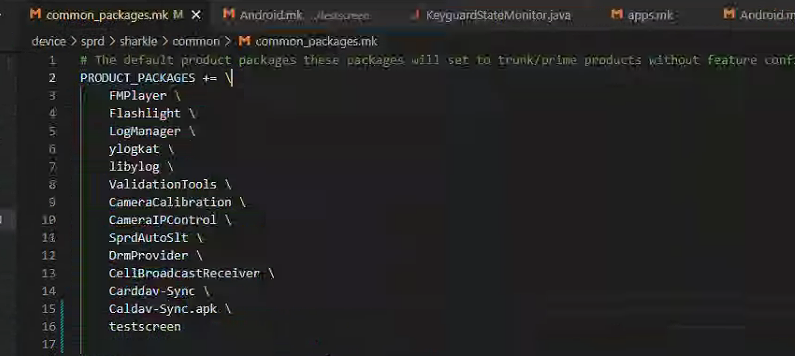
LOCAL\_DEX\_PREOPT := false

# LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_DATA\_APPS)

#LOCAL\_CERTIFICATE := PRESIGNED

include $(BUILD\_PREBUILT) /\*必须包含的路径

include $(call all-makefiles-under,$(LOCAL\_PATH)) \*/

1. 完成上面两步后，编译时并不会将apk编译进out/system/app中。还需要再目录为device/sprd/sharkle/common/common\_packages.mk文件中添加你的app module名称，如下

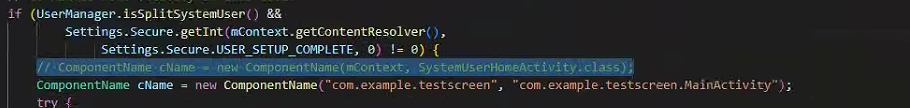
注意一定要将SO库导入完全，否则系统会处于恢复模式

## 系统启动后自动启动APP

1. 确保你的app以及被预装进系统，即编译后app文件会处于out/system/app目录下
2. 在目录为

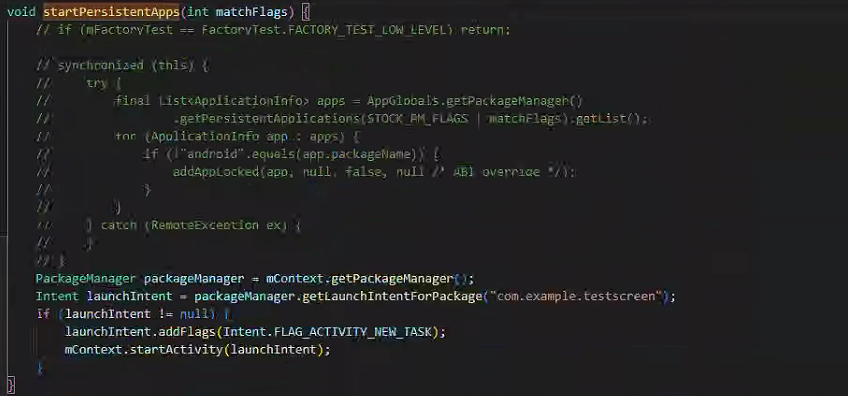


frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java文件下的public void systemReady函数中更改代码，如下图：



将你的app包名，以及app内的主活动名称替换进去

1. 再次更改该文件下的持久性app为你的app包名，函数名称为void startPersistentApps，更改如下：



# linux系统迁移（从电脑A迁移至电脑B）

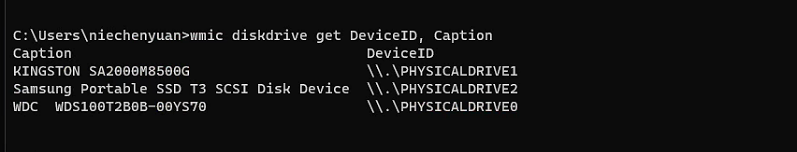
使用一个移动硬盘将其格式化为EXT4文件系统

方法：[(28条消息) 24.Ubuntu新磁盘挂载\_ubuntu挂载硬盘\_其木王·王子的博客-CSDN博客](https://blog.csdn.net/u014217137/article/details/128057593)

在windows终端（cmd）上使用指令

wmic diskdrive get DeviceID, Caption

可以查看当前电脑所有插入硬盘的盘符



# 安卓系统打包签名文件

Android系统签名证书的目录是“build/target/product/security”。

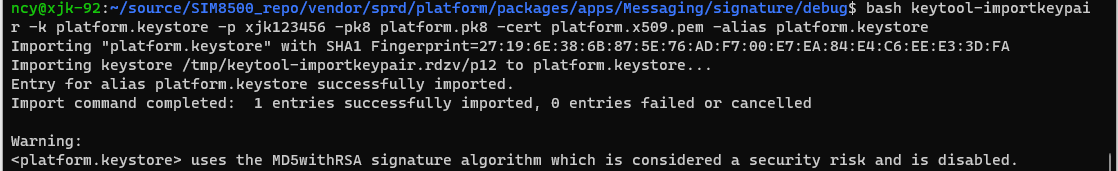
JKwear Mt2601的有效签名文件目录是” device\mediatek\common\security\sac2601\_l”

一 在github上下载工具

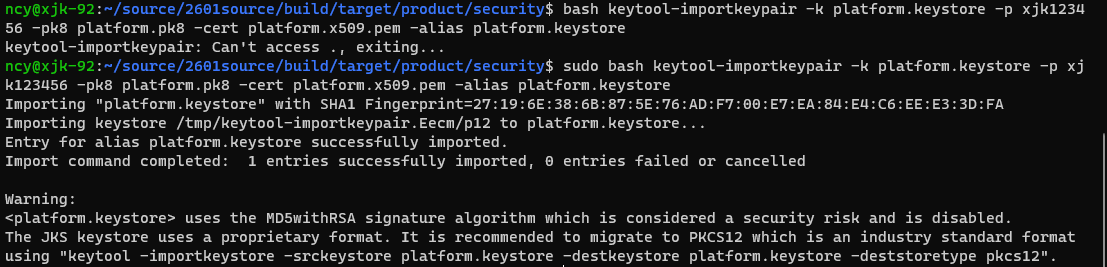
https://github.com/getfatday/keytool-importkeypair  
二 将工具在Linux环境下解压或者解压后Copy到Linux下，运行如下命令  
keytool-importkeypair -k ~/android/debug.keystore -p android -pk8 ~/android/platform.pk8 -cert ~/android/platform.x509.pem -alias anyname

其中 -k:表示要生成的Eclipse下的签名的keystore的名字，包括路径名，这里存到主目录的android目录下，名字为debug.keystore。你可以随便取名  
-p 表示新生成的keystore的密码是什么，这里为android  
-pk8 表示要导入的pk8文件的名称，可以包括路径，这里是主目录的Android目录下，我们的pk8文件和pem文件都在这个目录下  
-cert 表示要导入的证书文件,和pk8文件在同一个目录  
-alias 表示给生成的debug.keystore取一个别名，这个名字只有我们在签名的时候才用的到，生成在文件名是debug.keystore。这个名字，随便取

例1：



例2：



# 系统层和应用层通信的方法

1创建文件流

2 实现文件的接口函数open release write read unlocked\_ioclt

3 文件流初始化如下

static const struct file\_operations sensor\_fops = {

    .owner = THIS\_MODULE,

    .open = sensor\_open,

    .release = sensor\_release,

    .write = sensor\_write,

    .read = sensor\_read,

    .unlocked\_ioctl = sensor\_ioctl

};

static struct miscdevice sensorhub\_misc = {

    .minor = MISC\_DYNAMIC\_MINOR,

    .name = "xjkbuffer",

    .fops = &sensor\_fops};

int xjktest\_init(void \*bus\_dev)

{

    int error;

    //注册字符设备

    send\_start\_flag = 0;

    error = misc\_register(&real\_sensorhub\_misc);

    error = misc\_register(&sensorhub\_misc);

    wake\_lock\_init(&data\_sync\_wakelock, WAKE\_LOCK\_SUSPEND, "sensorh\_hub\_lock");//初始化锁

    return error;

}

4 文件流需要selinux权限，并且系统需要提供签名文件给应用层