Huawei LiteOS 应用开发指南

文档版本　00B03

发布日期　2016-09-01

商标声明

 、HISILICON、海思和其他海思商标均为深圳市海思半导体有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

深圳市海思半导体有限公司

地址：　深圳市龙岗区坂田华为基地华为电气生产中心　邮编：518129

网址：　http://www.hisilicon.com

客户服务电话：　+86-755-28788858

客户服务传真：　+86-755-28357515

客户服务邮箱：　support@hisilicon.com

前 言

概述

本文档为开发入门引导文档，站在产品工程师开发的角度，对使用 Huawei LiteOS 开发产品的过程中遇到的各种问题进行引导说明。

具体的相关说明还请按指引到对应文件查找相关内容。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3516A	V100
Hi3516D	V100
Hi3518E	V200
Hi3518E	V201
Hi3516C	V200
Hi3519	V100
Hi3519	V101
Hi3516C	V300

修订记录

修订记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

文档版本 00B03 (2016-09-01)

第 3 次临时版本发布。

2.5.1 和 2.6 小节涉及修改；4.3.1 和 4.3.2 小节涉及修改。

文档版本 00B02 (2016-07-10)

第 2 次临时版本发布。删除 6.4 小节。

文档版本 00B01 (2016-04-08)

第 1 次临时版本发布。

目 录

插图目录

表格目录

1 Huawei LiteOS 开发与 Linux 开发的异同点

1.1 功能上的异同点

表1-1 Huawei LiteOS 与 Linux 功能上的异同点表

特性	Huawei LiteOS 与 Linux 的相同点	Huawei LiteOS 与 Linux 的差异点
fastboot	一致	-
uboot	基本一致	不支持 uboot 向 kernel 的参数传递
SDK	1、功能一致 2、接口基本一致 3、SDK 在 proc 文件系统注册的调试节点，在 OS 和 Linux 下使用方法一致	获取码流的接口在 Linux 下使用了虚拟地址到物理地址映射的特性解决物理内存 ringbuffer 回绕的问题，Huawei LiteOS 上需要上层应用处理；
文件系统	1、支持 jffs2，Nor Flash 文件系统 2、支持 yaffs2，Nand Flash 文件系统 3、支持 fat32，SD 卡文件系统	1、不支持 ext2/3/4 文件系统 2、不支持 cramfs、squashfs 文件系统 3、不支持 UBI 文件系统
网络协议栈	支持 tcp/ip 协议栈	1、不支持 IPV6 2、socket 支持的选项略有区别
Shell	支持命令行查看系统运行、任务、内存、proc 等信息	1、Huawei LiteOS 的命令行仅用于调试，不能像 Linux 那样可以在应用中调用 2、Huawei LiteOS 的命令需要注册命令的处理函数，不能像 Linux 那样是独立的可执行程序
动态库	-	Huawei LiteOS 暂不支持动态库
调试手段	1、支持通过串口输出打印信息 2、支持通过串口执行命令	不支持 gdb 调试应用

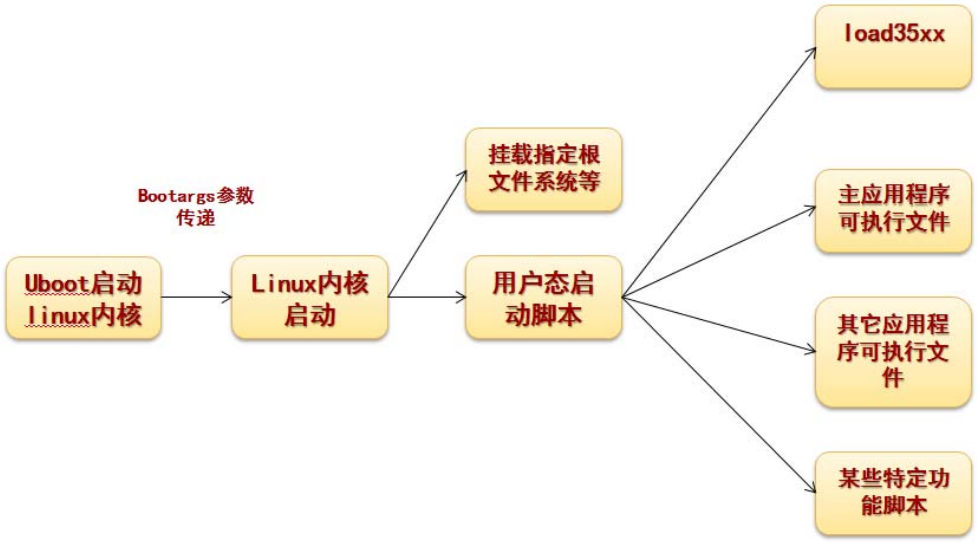
特性	Huawei LiteOS 与 Linux 的相同点	Huawei LiteOS 与 Linux 的差异点
	3、支持仿真器调试 4、异常死机输出调试信息	
内存	支持 MMZ	1、Huawei LiteOS 没有虚拟地址概念，代码中用到虚拟地址的地方实际上和物理地址一一对应 2、不支持 mmap 一个文件 3、Huawei LiteOS 的地址是不隔离的，可以访问任意地址空间
编译器	都使用 hisiv500 编译器	Huawei LiteOS 上编译时需要使用 Huawei LiteOS 的头文件，不能使用编译器自带的头文件
内核模块	驱动可以编译成独立的二进制文件，Huawei LiteOS 是.a，Linux 是.ko	1、Linux 通过 insmod 加载驱动的 ko，Huawei LiteOS 需要显式调用驱动的初始化函数 2、Linux 在 insmod 时可以传入模块参数，Huawei LiteOS 需要通过驱动初始化函数的参数设置 3、Linux 可以通过 shell 脚本加载内核模块，Huawei LiteOS 只能通过代码编程调用驱动的初始化函数
库函数	1、支持标准 C 语言库函数 2、支持标准数学函数库 3、支持 posix 标准函数库	1、Huawei LiteOS 不支持全部的 Linux 的库函数 2、Huawei LiteOS 支持的标准库函数，少数函数的规格比标准有差异 具体参考《Huawei LiteOS Kernel Deverloper Guide(zh)》中“8 标准库”列表说明。
C++	支持 C++	1、Huawei LiteOS 的 C++需要调用 LOS_CppSystemInit 进行 C++的初始化 2、Huawei LiteOS 不支持 C++的异常机制
进程/线程	1、支持线程模型 2、支持 Posix 线程库	1、Huawei LiteOS 不支持进程 2、Huawei LiteOS 不支持信号机制 3、Huawei LiteOS 不支持进程间通信方式
Flash 布局	1、uboot、bootargs 2、参数区	Huawei LiteOS 的 OS、驱动和应用编成一个镜像，不像 Linux 中分为 kernel 镜像、ko 文件、可执行文件等

1.2 启动和运行异同点

1.2.1 Linux 系统的启动和运行

Linux 系统的启动和运行示意图如图 1-1 所示。

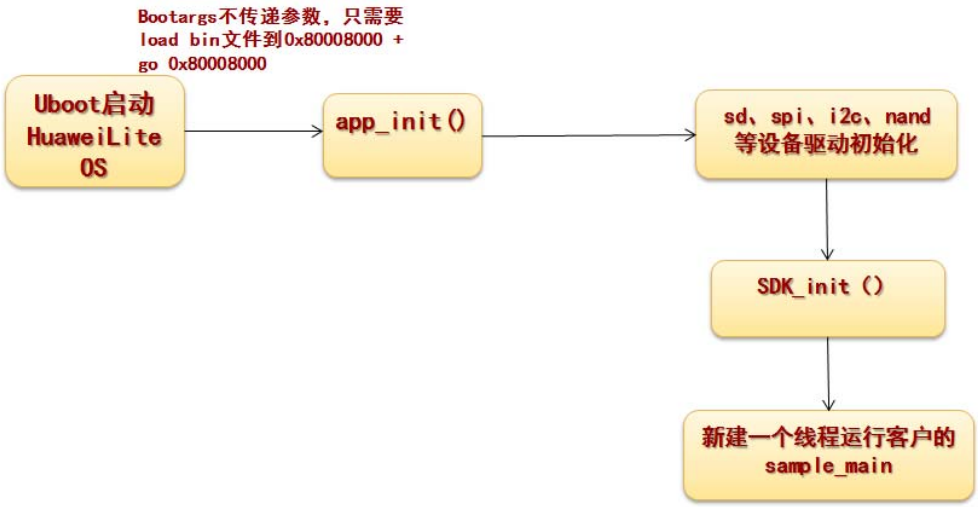
图1-1 Linux 系统的启动和运行示意图



1.2.2 Huawei LiteOS 系统的启动和运行

Huawei LiteOS 系统的启动和运行示意图如图 1-2 所示。

图1-2 Huawei LiteOS 系统的启动和运行示意图



2 开发环境设置

2.1 开发环境准备

Huawei LiteOS 不支持 Windows 编译，需要在 linux 服务器（或 linux 虚拟机）上进行编译。

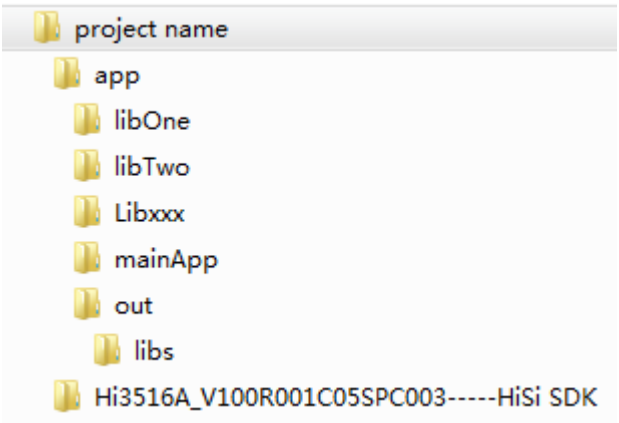
说明

Linux 服务器编译工具链的安装 请参考 SDK 包中/osdrv/Readme.txt

2.2 推荐的开发目录结构

开发目录结构推荐按如下设计。

图2-1 推荐的开发目录结构示意图



项目名称目录下，APP 路径和 SDK 并列，APP 和 SDK 完全分离，方便 SDK 升级。APP 下由各个功能 Lib 和入口 mainApp 组成，编译输出到 out 目录下，方便 APP 的库和 SDK 的库链接成一个 app.bin 文件。

2.3 Makefile 的设计

APP 的 Makefile 有几个关键点如下：

1. 必须使用 SDK 包中/osdrv/opensource/liteos/liteos /config.mk。其目的是：
   a. 获取当前编译器类型来定义 gcc、ar、ld、g++等编译命令。
      例如：CROSS_COMPILE = arm-hisiv500-linux-
      CC=$(CROSS_COMPILE)gcc
   b. 编译在对应芯片上运行代码所需的关键编译选项。
      - C 文件对应的编译选项为 LITEOS_COPTS
      - C++文件对应的编译选项为 LITEOS_CXXOPTS
      - 汇编文件对应的编译选项为 LITEOS_ASOPTS
   c. 编译所需的 OS 头文件的路径信息
      - LITEOS_USR_INCLUDE：应用层可见 OS 头文件
      - LITEOS_OSDRV_INCLUDE：驱动可见 OS 和驱动头文件
      - LITEOS_CXXINCLUDE：C++相关头文件
   d. 编译所需的 OS 宏
      LITEOS_MACRO：会影响 OS 头文件的编译，必须加入到编译选项中。
   e. 最终链接 bin 文件所需的 OS 库和库所在路径信息（LITEOS_LIBS 和 LITEOS_LDFLAGS）
   f. 最终链接 bin 文件所需的 ld 文件（LITEOS_LDFLAGS）
2. 最终链接 bin 文件所需的 SDK 的库文件和 sensor 库文件，参考 /mpp/sample/Makefile.param 中 SDK_LIB 的定义

以下是一份 Makefile.param 文件的样例：

```
# Hisilicon Hi35xx sample Makefile.param
# Get the local SDK_ROOT path or RELEASE_ROOT path by PARAM_FILE.
# PARAM_FILE is configed at Makefile before include this file
export DEV_PATH=/home/$(LOGNAME)/code/huaweiliteos_0924
export
SDK_PATH=$(DEV_PATH)/Hi3516A_V100R001C05SPC004/01.software/board/Hi3516A_
SDK_V5.0.0.4
export MPP_PATH=$(SDK_PATH)/mpp
export DRV_ROOT=$(SDK_PATH)/drv
################################################################################
export KERNEL_NAME=liteos
export LINUX_ROOT?=$(SDK_PATH)/osdrv/opensource/liteos/$(KERNEL_NAME)
export LITEOSTOPDIR?=$(LINUX_ROOT)
include $(LITEOSTOPDIR)/config.mk
################################################################################
INC_FLAGS = -I$(SDK_PATH)/mpp/component/acodec -I$(SDK_PATH)/mpp/include
-I$(SDK_PATH)/mpp/extdrv/sensor_spi
```

```
CFLAGS = $(LIBS_CFLAGS)
CFLAGS += $(LITEOS_COPTS)
CFLAGS += $(LITEOS_OSDRV_INCLUDE) $(LITEOS_USR_INCLUDE)
CPPFLAGS += $(LITEOS_CXXOPTS) -D__LITEOS__
CPPFLAGS += $(LITEOS_OSDRV_INCLUDE) $(LITEOS_USR_INCLUDE)
$(LITEOS_CXXINCLUDE)
################################################################################
LIB_OUT ?= $(LITEOSTOPDIR)/out/lib/
REL_DIR=$(SDK_PATH)/mpp
export REL_LIB := $(REL_DIR)/lib
SDK_LIB_PATH := -L$(REL_LIB) -L$(REL_LIB)/extdrv
SDK_LIB := -lsns_ov4689 -l_cmoscfg -l_iniparser -l_hiae -l_hiawb -l_hiaf
-l_hidefog -lmpi -lmmz -lhi3516a_sys -lhi3516a_viu -lhi3516a_vpss -
lhi3516a_vou -lhi_mipi \
        -lsensor_i2c -lsensor_spi -lhi3516a_isp -lisp -lpwm -
lhi3516a_vda -lhi3516a_vgs -lhi3516a_venc -lhi3516a_rc -lhi3516a_chnl\
         -lhi3516a_h264e -lhi3516a_h265e -lhi3516a_jpege -
lhi3516a_region -lsil9024 -ltde -lhi3516a_tde -lhi3516a_base\
         -lupvqe -ldnvqe -lVoiceEngine -lhifb -lhi3516a_ive -live -
lhi3516a_adec -lhi3516a_aenc -lhi3516a_aio -lhi3516a_ai -lacodec -
lhi3516a_ao
SDK_LIB += $(SENSOR_LIBS)
LIBDEP = $(LITEOS_LIBS)
CFLAGS += $(SCFLAGS)
LDFLAGS  = -nostartfiles -static -L$(LIB_OUT) $(SDK_LIB_PATH)
$(LITEOS_LDFLAGS)
```

在 app 路径下的 Makefile 文件包含该 Makefile.param，即可访问 SDK 对应的各种头文件和 lib 库。

app 路径下 Makefile 中的 TARGET 推荐按如下定义：

```
$(TARGET): $(EXEC_OBJS)
    $(LD) $(LDFLAGS) -Map=$(MAP) -o $(TARGET) $(EXEC_OBJS) --start-group
$(LIBS) $(SDK_LIB) $(LIBDEP) --end-group
    $(OBJCOPY) -O binary $(TARGET) $(EXEC_BIN)
    $(OBJDUMP) -d $@ > $@.asm
```

同时输出.asm 文件可以在系统出现异常的时候定位出是哪个函数出了问题。

2.4 下载编译好的 bin 文件到板端运行

请参考《Hi35xx SDK 安装以及升级使用说明.txt》将编译好的 bin 文件烧写到板端运行。

2.5 编程语言

Huawei LiteOS 支持 C 和 C++。

2.5.1 C++程序开发

C++初始化

在系统第一次调用 C++代码前，需要调用如下函数保证 C++程序的构造函数的运行：

```
int LOS_CppSystemInit(unsigned long start, unsigned long end, int flag);
```

具体信息参考《Huawei LiteOS Kernel Deverloper Guide(zh)》中“4.6 C++支持”的说明。

2.6 客户原有代码移植 Huawei LiteOS 的一些建议

- 将各个进程和脚本，都变为线程实现。
- 有线、无线网络初始化，Huawei LiteOS 有自己的初始化接口，无需继续保留 Linux 下的相关实现。
- 所有 system()替换为代码实现。
- Huawei LiteOS 不依赖文件系统，上电初始化不挂载文件系统可以提高开机速度，因此应用可以把配置文件保存在 Flash 上的一个分区，直接对 Flash 读写，对于可写分区，为了安全其见，可以做 Flash 备份。
- 请特别留意局部变量的使用，Huawei LiteOS 线程栈大小默认只有 24KB。可以根据需要使用 pthread 接口改大指定线程的栈大小。
- Huawei LiteOS 的 SDK 媒体接口及概念与 Linux 版本媒体接口及概念基本一致，原有海思 Linux 版本代码可直接移植使用；少数涉及内存映射的代码需要修改，例如获取编码数据时需要处理环回，具体参考《Hi35xx Huawei LiteOS 与 Linux 开发包差异说明》。

3 系统启动及系统级配置

3.1 系统启动流程

3.1.1 应用启动函数

Huawei LiteOS 应用启动的第一个函数是 app_init()，请勿改动该函数名。

app_init()是 OS 默认启动的线程。

参考/mpp/sample/common/app_init.c 中 app_init()的实现，该函数中会进行一些系统应用使用到的 OS、驱动、SDK 模块的初始化，最后会启动一个线程（sample_command）进行上层应用自己的模块初始化。

注意

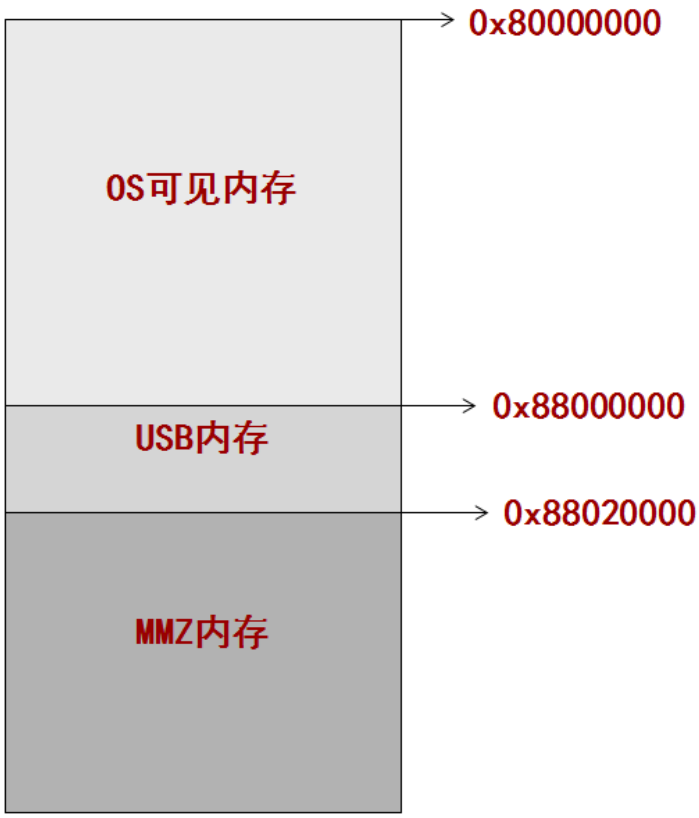
- **客户自行开发的应用模块初始化建议维持示例中在 sample_command 启动的线程中进行，从而避免 app_init 所在栈空间不够，发生栈溢出需要修改 OS 内部代码调整。**
- **示例代码中 app_init()中的 OS、驱动、SDK 模块初始化函数，可根据实际应用需求保留、删除或调整顺序。**

3.2 系统内存配置

Huawei LiteOS 内存分系统 OS 内存、MMZ 内存、USB 内存三大块。

3.2.1 ～3.2.2 的示例以 DDR 大小为 128MB 进行内存配置，系统内存的分配如图 3-1 所示。

图3-1 系统内存分配示例图



3.2.1 OS 可见内存和 USB 内存的配置

通过定义下面的函数实现：

```
void board_config(void)
{
    g_sys_mem_addr_end = 0x88000000;
    g_usb_mem_addr_start= g_sys_mem_addr_end;
    g_usb_mem_size= 0x20000; //recommend 128K nonCache for usb
}
```

注意

- 不可以修改 void board_config(void)的声明，若应用层不定义 void board_config(void)函数，Huawei LiteOS 会按默认值分配 OS 可见内存大小。建议自行根据实际情况进行定义。
- g_sys_mem_addr_end 配置的是 OS 可见内存的结束位置。
- g_usb_mem_addr_start 配置的是 USB 内存的起始位置；g_usb_mem_size 配置的是 USB 内存的大小。
- 若系统完全不支持 usb 相关功能，则可以不配置 g_usb_mem_addr_start 和 g_usb_mem_size，配置为 0 即可。

3.2.2 MMZ 内存配置

在/mpp/ko/sdk_init.c 中如下函数中定义：

```
static HI_S32 MMZ_init(void)
{
    extern int media_mem_init(void * pArgs);
    MMZ_MODULE_PARAMS_S stMMZ_Param;
    snprintf(stMMZ_Param.mmz, MMZ_SETUP_CMDLINE_LEN, "anonymous,0,
0x88020000,63M");
    stMMZ_Param.anony = 1;
    return media_mem_init(&stMMZ_Param);
}
```

4 应用开发

4.1 媒体业务相关

4.1.1 基本概念

Huawei LiteOS 的 SDK 媒体接口及概念与 Linux 版本媒体接口及概念一致，已有 linux 版本的相关代码，可直接移植使用。

少数涉及内存映射的代码需要修改，例如获取编码数据时需要处理环回，具体参考《Hi35xx Huawei LiteOS 与 Linux 开发包差异说明》。

原本在 load35xx 脚本中调用的媒体相关初始化脚本，由/mpp/ko/sdk_init.c 中 SDK_init()实现。

若无海思平台开发经验，需要重新编码，请参考《HiMPP IPC V3.0 媒体处理软件开发参考》和/mpp/sample 来进行开发。

4.1.2 涉及相关函数

1. Sensor 控制接口相关总线的初始化，例如：i2c_dev_init()/hi_spi_init()
2. Load35xx 的替代者：SDK_init()

   原来 linux 下插入 ko 时加载的参数，比如 hi35xx_h264e.ko 模块参数：H264eMiniBufMode。在 Huawei LiteOS 中使用 HI_MPI_VENC_SetModParam 函数来替代实现，具体请参考《HiMPP IPC V3.0 媒体处理软件开发参考》。
3. MPP 媒体处理接口，与 linux 版本一致，具体请参考《HiMPP IPC V3.0 媒体处理软件开发参考》和《Huawei LiteOS 与 Linux 开发包差异说明》。

4.2 芯片外设相关

外围设备的使用，具体请参考《外围设备驱动操作指南》，这里对常用的外设驱动做引导说明。

4.2.1 SD

SD 驱动提供 mmc、sdio 的支持，需调用 SD_MMC_Host_init()初始化 SD 驱动。

具体参考《外围设备驱动操作指南》。

注意

SD_MMC_Host_init()初始化后不是立刻就可以识别到卡，因此 mount 前须检查对应设备节点是否存在。

4.2.2 USB

Huawei LiteOS 支持读写 U 盘（下文简称为 USB2.0 Host），也支持作为 USB 大容量存储设备被 PC 读写（下文简称为 USB2.0 Device）。

需调用 usb_init（）初始化 USB 驱动。

- USB2.0 Device 功能，应用需通过 fmass_register_notify(void(*nofify)(void* context, int status), void* context)函数注册 notify 回调，驱动会在识别到接入和拔出 USB2.0 Host 时调用此回调函数，从而实现 USB2.0 Device 相关拔插的处理。
- USB2.0 Host 功能，驱动识别到 U 盘设备的插入后，会在/dev 目录下生成对应的设备节点/dev/sda 和/dev/sdap0，应用对 U 盘进行格式化、挂载等操作后，即可对 U 盘进行读写访问。

具体参考《外围设备驱动操作指南》。

注意

USB2.0 Device 是将 SD 卡交由 USB Host 处理，因此在 USB2.0 Device 识别到 USB2.0 Host 后，会通过回调函数通知 APP 软件模块，APP 应该将所有录像抓拍等操作 sd 卡的业务停下，并 umount sd 卡分区，待 usb host 离线后再 mount 和恢复相关业务。

4.2.3 UART

需调用 hi_uartdev_init（）初始化 UART 驱动。

使用某个串口与外设进行串口通信，使用示例如下：

```
fd = open("dev/uartdev-1", O_RDWR);      //打开串口设备
ret = ioctl(fd, CFG_BAUDRATE, 115200);   //配置串口波特率属性
if (ret < 0) {
printf("config error \n");
ret = -2;
```

```
}else{
ret = read(fd, str, len);                    //读串口接收的数据
if (ret < 0) {
        printf("read err %d.\n", ret);
        ret = -2;
}
}
ret = close(fd);                             //关闭串口设备
```

具体参考《外围设备驱动操作指南》

4.2.4 控制台

- 控制台提供串口输出日志信息功能。
- 控制台功能依赖 UART 驱动实现串口输出日志信息，因此依赖 UART 驱动的初始化。通过调用 system_console_init(TTY_DEVICE);初始化控制台功能，其中 TTY_DEVICE 为“/dev/uartdev-0”。
- 控制台初始化后，printf 函数方可使用。

4.2.5 I2C

1. 开发者需要根据设备硬件特性自行配置相关管脚复用，请参考《Hi351x 专业型 HD IP Camera Soc 用户指南》第 2.3 章节——管脚复用控制寄存器（Hi3519V10x 请参考《Hi3519V10x_PINOUT_CN》表单）。
2. 调用 i2c_dev_init()函数初始化 i2c 总线。
3. 用户可根据需要自行调用模块的读写函数对设备进行访问。

具体参考《外围设备驱动操作指南》。

4.3 网络开发相关

4.3.1 Socket 编程

Huawei LiteOS 采用的是 lwip 协议栈。

socket 编程基本与 linux 一致，可延续使用。Socket 相关的选项有些和 Linux 协议栈定义的不一致，在移植程序的时候需要关注。具体参考《Huawei LiteOS LwIP Developer Guide》中“1.5 Huawei LiteOS LwIP-BSD Compatibility”说明。

注意

网络 socket 的描述符是独立操作的，不支持和文件或设备的描述符一起组合操作，比如针对 select 系统调用，不支持网络 socket 描述符和设备描述符一起组合。

4.3.2 以太网

1. 要确保以太网初始化前，先初始化 tcpip

```
STlwIPSecFuncSsp stlwIPSspCbk= {0};
stlwIPSspCbk.pfMemset_s = Stub_MemSet;
stlwIPSspCbk.pfMemcpy_s = Stub_MemCpy;
stlwIPSspCbk.pfStrNCpy_s = Stub_StrnCpy;
stlwIPSspCbk.pfStrNCat_s = Stub_StrnCat;
stlwIPSspCbk.pfStrCat_s = Stub_StrCat;
stlwIPSspCbk.pfMemMove_s = Stub_MemMove;
stlwIPSspCbk.pfSnprintf_s = Stub_Snprintf;
stlwIPSspCbk.pfRand = rand;
ret = lwIPRegSecSspCbk(&stlwIPSspCbk);
tcpip_init(NULL,NULL)
```

2. 在 tcpip 初始化后调用以太网驱动初始化接口：

```
higmac_init()
```

3. 启动以太网设备：

```
extern struct los_eth_driver higmac_drv_sc;
struct netif *pnetif;
pnetif = &(higmac_drv_sc.ac_if);
netif_set_up(pnetif);
```

4. 关闭以太网设备

```
netif_set_down(pnetif);
```

5. 有线网络下 DHCP 的启停接口和判断是否已获取到 IP 地址的接口

   a. netifapi_dhcp_start(struct netif *netif);启动 DHCP

   b. netifapi_dhcp_is_bound(struct netif *netif);判断 DHCP 是否获取到 IP 地址，返回值 0 为获取到 IP 地址，其它为错误码

   c.netifapi_dhcp_stop(struct netif *netif);停止 dhcp

   相关头文件在 osdrv\opensource\liteos\liteos\net\lwip\include\lwip\下，名字为 netifapi.h

   示例代码：

```
#include "netifapi.h"
extern struct los_eth_driver  hisi_eth_drv_sc;
struct netif *pnetif;
pnetif = &(hisi_eth_drv_sc.ac_if);
netifapi_dhcp_start(pnetif);  //启动dhcp
netifapi_dhcp_stop(pnetif);停止dhcp
```

6. ip 地址、netmask、gateway 的获取和配置，和 linux 的 socket 编程一致：
   - 查询有线网络的状态
   - 使用如下宏定义查询有线网络的状态，返回 1 是网线连上，0 没连上

```
/** Ask if a link is up */
#define netif_is_link_up(netif) (((netif)->flags &
NETIF_FLAG_LINK_UP) ? (u8_t)1 : (u8_t)0)
```

使用示例：

```
#include "netif.h"
extern struct los_eth_driver   hisi_eth_drv_sc;
struct netif *pnetif;
pnetif = &(hisi_eth_drv_sc.ac_if);
ret=netif_is_link_up(pnetif)
if(ret==1)
{
   连上
}
else
{
   没连上
}
```

4.4 文件系统

Huawei LiteOS 支持的文件系统有 JFFS2(针对 SPI Nor FLASH 器件)、Yaffs2(针对 Nand FLASH 器件)、FAT32（SD 卡、eMMC 等大容量存储外设）、PROC 文件系统、VFS。

4.4.1 VFS

VFS 虚拟文件系统（Virtual File System）。VFS 为系统应用访问文件系统提供了统一的抽象接口。系统通过 VFS 层调用统一文件操作函数实现对 FAT、YAFFS2、JFFS2 等文件系统类型的访问。

VFS 通过在内存中的树结构来实现的，树的每个结点都是一个 inode 结构体。设备注册和文件系统挂载后会根据路径在树中生成相应的结点。

VFS 最主要是两个功能：

1. 查找节点
2. 统一调用（标准）

通过 VFS 层，可以使用标准的 Unix 文件操作函数（如 open、read、write 等）来实现对不同介质上不同文件系统的访问。

VFS 框架内存中的 inode 树结点有三种类型。

- 虚拟结点：作为 vfs 框架的虚拟文件，保持树的连续性，如/bin,/bin/vs
- 设备结点：/dev 目录下，对应一个设备，如/dev/mmc0

- 挂载点：调用 mount 函数后生成，如/bin/vs/sd,/ramfs,/yaffs

VFS 为文件系统的框架，一般的应用层开发者不会涉及。

具体参考《Huawei LiteOS Kernel Deverloper Guide(zh)》中“5.2 VFS”的说明。

4.4.2 JFFS2 文件系统

JFFS2 是 Journalling Flash File System Version 2(日志文件系统)的缩写。它的功能是管理在设备上实现的日志型文件系统。主要应用于对 NOR_FLASH 闪存的文件管理。Huawei LiteOS 的 JFFS2 支持多分区。

JFFS2 的使用方法如下：

步骤 1. 调用 spinor_init()初始化 NorFlash 器件。

步骤 2. 在 NorFlash 上划分物理分区

```
int add_mtd_partition( char *type, uint32_t start_addr, uint32_t length,
uint32_t partition_num);
```

type：定义包括“nand”和“spinor”。JFFS2 为“spinor”

partition_num：由 APP 指定，假定是 0 分区，增加分区之后，会在/dev 目录下生成：/dev/spinorblk0 设备节点。

步骤 3. 设备节点生成之后，就可以调用 mount 系统例程挂载 JFFS2 文件系统了：

```
int mount(FAR const char *source, FAR const char *target,
        FAR const char *filesystemtype, unsigned long mountflags,
        FAR const void *data);
```

其中的 filesystemtype 包括：vfat，romfs，yaffs，jffs。例如：

```
   uwRet=mount(“/dev/spinorblk0”, JFFS_DIR, "jffs", 0, NULL);
```

这样就可以挂载 JFFS2 分区到 JFFS_DIR 指定的目录上了。

----结束

具体参考《Huawei LiteOS Kernel Deverloper Guide(zh)》中“5.4 JFFS2”的说明。

4.4.3 YAFFS2 文件系统

YAFFS 是 Yet Another Flash File System 的简称，是一种开源的、针对 Nand flash 的嵌入式文件系统。适用于大容量的存储设备，同时也使得 Nand flash 具有高效性和健壮性 YAFFS2 为文件系统提供了损耗平衡和掉电保护，保证数据在系统对文件系统修改的过程中发生意外而不损坏。Huawei LiteOS 的 YAFFS2 支持多分区。

YAFFS2 的使用方法如下：

步骤 1. 调用 nand_init()初始化 NandFlash 器件。

步骤 2. 与 JFFS 文件系统类似，通过 add_mtd_paratition 函数在 NandFlash 上划分物理分区。只是指定的文件系统类形为 yaffs，以 0 号分区为例，生成的块设备名称为 "/dev/nandblk0"。

步骤 3. 设备节点生成之后，就可以调用 mount 系统例程挂载 yaffs2 文件系统了。

```
uwRet=mount(“/dev/nandblk0”,YAFFS_DIR, "yaffs", 0, NULL);
```

----结束

具体参考《Huawei LiteOS Kernel Deverloper Guide(zh)》中“5.6 YAFFS2”的说明。

4.4.4 FAT 文件系统

FAT 文件系统是 File Allocation Table（文件配置表）的简称，有 FAT12、FAT16、FAT32 这几种类型。在可移动存储介质(U 盘、SD 卡、移动硬盘等)上多使用 FAT 文件系统，使设备与 Windows、Linux 等桌面系统之间保持很好的兼容性。

FAT 文件系统最大可以支持 64GB SD 卡分区。

FAT 的使用方法如下：

步骤 1. FAT 文件系统所在的大容量存储设备驱动初始化。

```
SD_MMC_Host_init()：SD卡或者eMMC设备的SD驱动初始化
usb_init()：USB驱动初始化
```

步骤 2. SD 卡识别后在/dev 目录下产生设备节点和分区设备节点，其设备名称规则如下。

设备名称是在 SD 卡枚举的时候指定的，一般是“/dev/mmcblk0”，表示这个块设备是接在 MMC 控制器 0 上的，如果硬件上是接到 MMC 控制器 1 上，则设备名是“/dev/mmcblk1”。

驱动依据 SD 卡上分区的情况产生分区对应设备节点，每增加一个分区分区号递增 1。例如：“/dev/mmcblk0p0”、“/dev/mmcblk0p1”。

步骤 3. eMMC 设备的设备节点和分区设备节点。

在内置的 eMMC 也用于镜像文件的存储时，会分三个物理分区用于存储 uboot.bin/sample.bin/内置 FAT 分区。因此需要通过如下接口配置物理分区。

```
UINT32 add_mmc_partition(const char *dev_name, UINT32 start_block, UINT32
blocks);
```

需要注意，这里是用 block 为单位指定分区的起始地址和长度的，每个 block 固定是 512 字节。设备名称是在 eMMC 设备枚举的时候指定的，一般是“/dev/mmcblk0”，表示这个块设备是接在 MMC 控制器 0 上的，如果硬件上是接到 MMC 控制器 1 上，则设备名是“/dev/mmcblk1”。增加分区之后，每增加一个分区分区号递增 1，对应的设备节点名称加 1。

步骤 4. U 盘识别后在/dev 目录下产生设备节点和分区设备节点，其设备名称规则如下。

设备名称是在 U 盘枚举的时候指定的，一般是“/dev/sda”。

驱动依据 U 盘上分区的情况产生分区对应设备节点，每增加一个分区分区号递增 1。例如：“/dev/sdap0”、 “/dev/sdap1”。

步骤 5. 分区设备节点生成之后，就可以调用 mount 系统例程挂载 fat 文件系统了。

对于 SD 卡设备的 0 号分区，挂载命令为：

```
uwRet = mount("/dev/mmcblk0p0", "/mnt", "vfat", 0, 0);
```

表示把 MMC 控制器 0 上的块设备分区 0，挂载到目录/mnt 下，文件系统类形是 FAT32。

----结束

4.4.5 PROC 文件系统

PROC 文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

4.4.5.1 PROC 文件系统的使用方法

PROC 文件系统的使用方法如下：

步骤 1. 通过调用 proc_fs_init()函数初始化 PROC 文件系统。

步骤 2. 运行 cat 命令查看节点信息；运行 writeproc 命令修改节点信息。

----结束

具体参考《Huawei LiteOS Kernel Deverloper Guide(zh)》中“5.8 PROC”的说明。

4.4.5.2 在 PROC 文件系统下增加 Proc 文件节点

增加 Proc 文件节点调用接口包括如下：

```
struct proc_dir_entry *create_proc_entry(const char *name, umode_t mode,
                    struct proc_dir_entry *parent);
```

参数说明：

- name：待创建的目录项树节点的名字；
- mode：文件的访问权限(模式);该参数无效；
- parent：指定该文件的上层 proc 目录项树节点，如果为 NULL，表示创建在 /proc 根目录树节点下；
- 返回值：成功时返回 proc_dir_entry 实例指针；失败时返回 NULL。

如下为示例：

```
    struct proc_dir_entry *driverDir;
    driverDir = proc_mkdir("driver", NULL);
    if(driverDir == NULL)
    {
```

```
        printk(KERN_ERR "mipi: can't create proc dir driver.\n");
        goto MIPI_INIT_FAIL2;
    }
    mipi_entry = create_proc_entry(MIPI_PROC_NAME, 0644, driverDir);
    if(mipi_entry == NULL)
    {
        printk(KERN_ERR "mipi: can't create %s.\n", MIPI_PROC_NAME);
        goto MIPI_INIT_FAIL3;
    }
mipi_entry->read_proc = mipi_proc_show;
```

4.5 系统时间

4.5.1 系统启动时间的量测

- 在 APP 中如果需要精确计算系统的启动时间，可以按如下方式实现。

  ```
  extern unsigned int hi_getmsclock(void);
  printf("\033[0;31m %s:%d: %dms\n \033[0m", __FUNCTION__, __LINE__,
  hi_getmsclock());
  ```

  hi_getmsclock 例程返回从系统启动到执行的时候的毫秒数，但是这个不包括 uboot 时间。

- uboot 的时间可以在 uboot 中调用 READ_TIMER 实现，样例代码如下。

  ```
  void hi_getcurtime()
  {
      /* stage 1 timing */
      unsigned long time_0 = READ_TIMER;
      time_0 = 0xffffffff - time_0;
      time_0 = time_0 * 256;
      time_0 = time_0 / 49500;
      printf("----------stage n clock : %d ms\n", time_0);
  }
  ```

4.5.2 系统 Tick

系统 Tick 设置为 10ms 一个，是 gettimeofday，sleep，msleep 等系统调用的基础，Tick 时间可以通过以下例程获取。

```
OS_SEC_TEXT_MINOR UINT64 LOS_TickCountGet (VOID); //获取系统当前Tick值
OS_SEC_TEXT_MINOR UINT32 LOS_MilliSec2Tick(UINT32 millisec);//毫秒转换成
Tick
OS_SEC_TEXT_MINOR UINT32 LOS_Tick2MilliSec(UINT32 tick);//Tick转换成毫秒
```

4.5.3 时区设置

配置时区通过 void settimezone (char *buff)函数配置时区信息，配置后再通过 localtime 获取本地时间将是配置后结果。

void settimezone (char *buff)函数传参为一个固定格式的字符串，其格式如下：

```
STD[-/+]hour:0:0
```

只有+/-和 hour 可配，其余不可改变。

示例如下：

```
#include “time.h”
settimezone (“STD-7:0:0”);    //代表东7区
settimezone (“STD+7:0:0”);   //代表西7区
```

4.5.4 Ntp client

Huawei LiteOS 支持 ntp client 功能，具体使用方法参见《Huawei LiteOS lwIP API Reference》中 sntp 相关说明。

5 驱动开发

5.1 驱动实现

一个驱动一般包括一个初始化函数，注册 DEV 设备节点，提供该节点的读写和 ioctl 配置和控制调用接口。

注册驱动的接口如下：

```
struct file_operations_vfs {
  int     (*open)(FAR struct file *filep);
  int     (*close)(FAR struct file *filep);
  ssize_t (*read)(FAR struct file *filep, FAR char *buffer, size_t
buflen);
  ssize_t (*write)(FAR struct file *filep, FAR const char *buffer, size_t
buflen);
  off_t   (*seek)(FAR struct file *filep, off_t offset, int whence);
  int     (*ioctl)(FAR struct file *filep, int cmd, unsigned long arg);
  int     (*poll)(FAR struct file *filep, poll_table *fds);
  int     (*unlink)(FAR struct inode *inode);
};
int register_driver(FAR const char *path, FAR const struct
file_operations_vfs *fops,
                    mode_t mode, FAR void *priv)
```

register_driver 函数的参数说明如下：

- path：指设备节点的路径，如：/dev/hi_rtc；
- fops：是可以对这个节点的各种操作的执行函数；
- mode：是配置该节点文件的可读写，可执行等模式，应用读写该驱动节点权限设置，暂时无效，后续或提供支持；
- priv：驱动节点注册过程需要传入的参数。

调用示例如下：

```
register_driver("/dev/cw201x_bat", &cw_bat_fops, 0666, 0);
```

具体参考《Huawei LiteOS Kernel Deverloper Guide(zh) 》中“6 驱动开发指导”说明。

5.2 Select 实现

struct file_operations_vfs 中的 poll 接口用于 select 系统调用，如果需要用 select 系统调用等待某个条件成立，需要实现 poll 接口。在 poll 的实现中可以调用 poll_wait 函数来等待系统中断唤醒，或者其他任务唤醒该 poll 动作。poll_wait 的声明如下：

```
void poll_wait(struct file *filp, wait_queue_head_t *wait_address,
poll_table *p);
```

poll_wait 的 wait_address 参数需要调用宏 init_waitqueue_head(p_wait)初始化，通过调用 wake_up_interruptible(p_wait)唤醒，具体参见 compat/linux/include/linux/wait.h 头文件中的定义。使用示例：

```
//定义驱动操作函数数据结构
const static struct file_operations_vfs venc_fops =
{
    .open = VencOpen,
    .close = VencClose,
    .ioctl = VENC_Ioctl,
    .poll = VencPoll,
};
//驱动初始化函数初始化等待队列头
HI_S32 VENC_Init(void *p)
{
….
    init_waitqueue_head(&g_stVencChn[i].waitStream);
}
//中断处理函数里唤醒等待队列
HI_VOID VENC_FrameOverNotify(VENC_CHN VeChn)
{
….
    wake_up_interruptible(&pstVencChn->waitStream);
….
}
//在select函数调用的时候VencPoll阻塞等待中断事件
static unsigned int VencPoll(struct file * file, poll_table *wait)
{
….
    poll_wait(file,&pstVencChn->waitStream,wait);
….
}
```

6 其它开发

6.1 命令行

命令行分系统命令行和自定义命令行，通过调用 osShellInit()来初始化 shell 功能。

系统启动后可以通过在串口敲击“help”命令，然后回车查看当前系统支持的命令有哪些。

6.1.1 系统命令行

通过调用 shell_cmd_register()来注册系统命令行，此函数为 OS 提供的默认的系统命令行注册函数。

系统默认带的 shell 命令说明在如下三份文档中予以说明：

- 《Huawei LiteOS Kernel Developer Guide》说明 OS 相关的 shell 命令
- 《外围设备驱动 操作指南》说明外围驱动相关的 shell 命令
- 《HiMPP IPC V3.0 媒体处理软件开发参考》说明通过 cat 命令可以查看的 sdk 相关信息

6.1.2 自定义命令行

通过调用如下函数注册自定义命令

```
UINT32 osCmdReg(UINT32 uwCmdType,CHAR*pscCmdKey,UINT32
uiParaNum,CMD_CBK_FUNC pfnCmdProc)
```

调用示例如下：

```
osCmdReg(CMD_TYPE_EX,”sample”,0,(CMD_CBK_FUNC)app_sample);
```

执行完此示例函数后，可以在 shell 中执行“sample”命令，具体执行的命令内容为 UINT32 app_sample(UINT32 argc, UINT32 **argv)函数。

6.2 分散加载

分散加载使用指南请参考《Huawei LiteOS 分散加载 Sample 说明》。

6.3 有关升级功能的开发

升级功能涉及两个方面，一是如何将升级包下载到目标机，二是如何将升级包写入FLASH。

- 升级包的制作和下载，这里不做特别说明。
- 升级包写入到 FLASH 请调用如下函数：

1. SPI Nor Flash：
   使用时请保证要 start 和 size 都要块对齐（块大小根据 FLASH 器件而定）
   int hispinor_erase(unsigned long start, unsigned long size)
   int hispinor_write(void* memaddr, unsigned long start, unsigned long size)
   int hispinor_read(void* memaddr, unsigned long start, unsigned long size)
2. Nand Flash 和 spi nandflash：
   使用时请保证要 start 和 size 都要块对齐（块大小根据 FLASH 器件而定）
   int hinand_erase(unsigned long start, unsigned long size)
   int hinand _write(void* memaddr, unsigned long start, unsigned long size)
   int hinand _read(void* memaddr, unsigned long start, unsigned long size)
3. Emmc：
   void emmc_raw_write(char * buffer, unsigned int start_sector, unsigned int nsectors)
   void emmc_raw_read(char * buffer, unsigned int start_sector, unsigned int nsectors)

7 问题定位

7.1 定位死机问题

一般死机都是出现异常导致，可能是指令异常，PC 指针跑到非指令区，或者 PC 指针在指令区没有 4 字节对齐，可能是数据访问异常，访问 0 地址，或者非 DDR 内存区地址，又不是寄存器地址。对于异常产生，Huawei LiteOS 捕获到该异常，会打印出函数调用栈和当时的寄存器值，通过函数调用栈，结合链接输出的 asm 文件或者 map 文件，可以定位出异常出现在哪个函数里，以及函数的调用轨迹，能查出是哪个模块，哪个任务，哪个函数出现了问题。在使用该功能之前，记住，在编译 APP 应用模块的时候，应该增加如下编译选项：

```
-fno-omit-frame-pointer
```

死机时打出的异常 debug 如图 7-1 所示：

图7-1 死机异常日志示例图

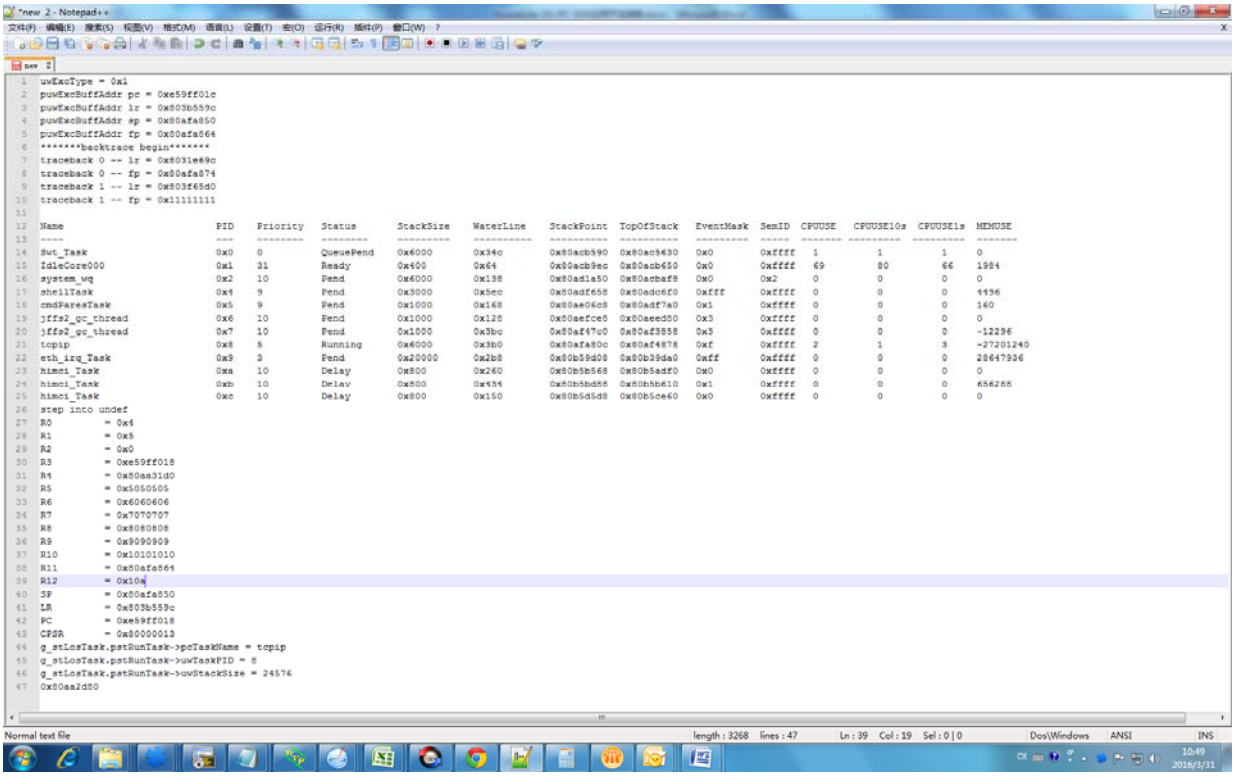


图 7-1 中，第一行“uwExcType = 0x1”代表死机类型，uwExcType 的值对应的死机类型定义在 los_exc.h 中，本示例中 0x1 代表死机类型为 OS_EXCEPT_UNDEF_INSTR 类型，说明汇编指令不支持导致的死机。

图7-2 死机类型定义图

```
/**
 *@ingroup los_exc
 *ARM7 exception type: undefined instruction exception.
 */
#define  OS_EXCEPT_UNDEF_INSTR                    1
/**
 *@ingroup los_exc
 *ARM7 exception type: software interrupt.
 */
#define  OS_EXCEPT_SWI                            2
/**
 *@ingroup los_exc
 *ARM7 exception type: prefetch abort exception.
 */
#define  OS_EXCEPT_PREFETCH_ABORT                 3
/**
 *@ingroup los_exc
 *ARM7 exception type: data abort exception.
 */
#define  OS_EXCEPT_DATA_ABORT                     4
/**
 *@ingroup los_exc
 *ARM7 exception type: FIQ exception.
 */
#define  OS_EXCEPT_FIQ                            5
```

1. “puwExcBuffAddr pc = 0xe59ff01c”为死机的点，配合对应 bin 文件的反汇编文件可以查到执行哪一句的时候死机的。但由于踩内存等原因，这个 pc 值有时候是无效值，不能在反汇编文件里查到对应语句。

图7-3 死机信息--PC 寄存器示例图

```
puwExcBuffAddr pc = 0xe59ff01c
puwExcBuffAddr lr = 0x803b559c
puwExcBuffAddr sp = 0x80afa850
puwExcBuffAddr fp = 0x80afa864
```

2. 从“*******backtrace begin*******”开始下方的 backtrace 代表的是死机时的调用栈信息。

图7-4 死机信息--调用栈信息示例图

```
*******backtrace begin*******
traceback 0 -- lr = 0x8031e69c
traceback 0 -- fp = 0x80afa874
traceback 1 -- lr = 0x803f65d0
traceback 1 -- fp = 0x11111111
```

- traceback 0 为死机所在函数的调用函数信息；
- traceback 1 为 traceback 0 所示函数的调用函数信息；
- lr 为函数的起始地址

3. 死机时的任务列表信息

图7-5 死机信息—任务列表信息示例图

Name	PID	Priority	Status	StackSize	WaterLine	StackPoint	TopOfStack	EventMask	SemID	CPUUSE	CPUUSE10s	CPUUSE1s	MEMUSE
----	---	--------	--------	---------	---------	----------	----------	---------	-----	-------	---------	---------	-------
Swt_Task	0x0	0	QueuePend	0x6000	0x34c	0x80acb590	0x80ac9630	0x0	0xffff	1	1	1	0
IdleCore000	0x1	31	Ready	0x400	0x64	0x80acb9ec	0x80acb650	0x0	0xffff	69	80	66	1984
system_wq	0x2	10	Pend	0x6000	0x138	0x80ad1a50	0x80acbaf8	0x0	0x2	0	0	0	0
shellTask	0x4	9	Pend	0x3000	0x5ec	0x80adf658	0x80adc6f0	0xfff	0xffff	0	0	0	4496
cmdParesTask	0x5	9	Pend	0x1000	0x168	0x80ae06c8	0x80adf7a0	0x1	0xffff	0	0	0	160
jffs2_gc_thread	0x6	10	Pend	0x1000	0x128	0x80aefce8	0x80aeed80	0x3	0xffff	0	0	0	0
jffs2_gc_thread	0x7	10	Pend	0x1000	0x3bc	0x80af47c0	0x80af3858	0x3	0xffff	0	0	0	-12296
tcpip	0x8	5	Running	0x6000	0x3b0	0x80afa80c	0x80af4878	0xf	0xffff	2	1	3	-27201240
eth_irq_Task	0x9	3	Pend	0x20000	0x2b8	0x80b59d08	0x80b39da0	0xff	0xffff	0	0	0	28647936
himci_Task	0xa	10	Delay	0x800	0x260	0x80b5b568	0x80b5adf0	0x0	0xffff	0	0	0	0
himci_Task	0xb	10	Delay	0x800	0x434	0x80b5bd88	0x80b5b610	0x1	0xffff	0	0	0	656288
himci_Task	0xc	10	Delay	0x800	0x150	0x80b5d5d8	0x80b5ce60	0x0	0xffff	0	0	0	0

4. 挂死现场信息

图7-6 死机信息—挂死现场信息示例图

```
R0          = 0x4
R1          = 0x5
R2          = 0x0
R3          = 0xe59ff018
R4          = 0x80aa31d0
R5          = 0x5050505
R6          = 0x6060606
R7          = 0x7070707
R8          = 0x8080808
R9          = 0x9090909
R10         = 0x10101010
R11         = 0x80afa864
R12         = 0x10a
SP          = 0x80afa850
LR          = 0x803b559c
PC          = 0xe59ff018
CPSR        = 0x80000013
```

5. 死机时运行的任务信息

图7-7 死机信息—死机时运行的任务信息示例图

```
g_stLosTask.pstRunTask->pcTaskName = tcpip
g_stLosTask.pstRunTask->uwTaskPID = 8
g_stLosTask.pstRunTask->uwStackSize = 24576
```

7.2 定位踩内存问题

踩内存是 RTOS 中比较常见但是却是最难定位的问题，踩内存引起的问题的现象五花八门，各种现象都有可能，但是问题不一定都是踩内存引起的。因此如果怀疑到是踩

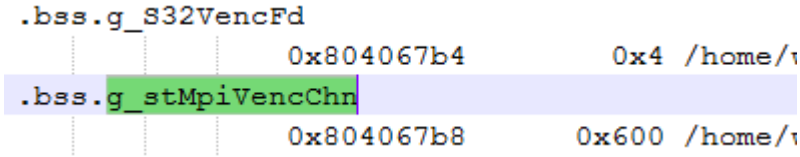
内存了，可以首先确定下内存分区是否合理，可以先调用 shell 中的 free 命令，查看系统内存剩余情况，同时查看 MMZ 的使用情况，确定 MMZ 内存、OS 内存、USB 内存的分配是合理，足够并且没有交叉的。

下面介绍几种踩内存的定位方法：

7.2.1 全局变量定位踩内存

在.map 文件中找到该全局变量所在的地址。并且特别注意该地址之前最近被使用的变量，有极大概率，是前面的变量在使用的过程中内存越界，踩掉了这个全局变量。如图 7-8 中的 g_stMpiVencChn 被踩。

图7-8 map 文件中 g_stMpiVencChn 信息图



在.map 中找到该全局变量在 bss 段对应的位置。若已经表现的现象为 g_stMpiVencChn 被踩，则需要特别注意分析 g_S32VencFd 这个全局变量的使用情况，观察是否存在某处，对全局变量 g_S32VencFd 进行了越界操作。

7.2.2 堆踩内存定位方法

堆内存即通过 malloc 得到的内存块被踩，这种踩内存有可能会破坏堆内存的管理信息（链表信息、魔数等）。因此当怀疑某一段代码有踩内存的可能时，可以调用 shCmd_memcheck 来检测该段代码执行完后是否有踩内存造成的管理信息的破坏。

可通过二分法在所有可能踩内存的操作前后添加 shCmd_memcheck 来定位踩内存的地方。

例如，有如下结构的代码：

```
extern UINT32 shCmd_memcheck(int argc, char *argv[]);
VOID sample_func(VOID)
{
    //1
    Operation1();
//2
    Operation2();
//3
    Operation3();
    //4
}
```

首先在'//1'、'//4'处添加 shCmd_memcheck(0,NULL)，对比前后 shCmd_memcheck 函数的 log，判断以上 operation1~3 是否造成节点损坏。若确定此处有造成节点损坏，则用二分法进一步定位更加准确位置。

7.2.3 用 shell 指令'task'，查看任务栈使用状态

查看 task 内存状态，分析是否踩内存（这里其实是判断任务栈是否溢出）。

操作：运行业务后，在串口中运行 task 指令。可以查看当前系统所有任务的状态。stackSize、WaterLine、StackPoint、Top0fStack 将作为一个指标判断任务栈是否踩内存.

图7-9 任务列表示例图

```
HuaweiLite OS # task
Name            PID  Priority  Status     StackSize  WaterLine  StackPoint  TopOfStack  EventMask  SemID  CPUUSE  CPUUSE10s  CPUUSE1s  MEMUSE
----            ---  --------  --------   ---------  ---------  ----------  ----------  ---------  -----  ------  ---------  --------  ------
Swt_Task        0x0  0         QueuePend  0x6000     0x360      0x80cfbad0  0x80cf5c70  0x0        0xffff  1       1          1         -120080
IdleCore000     0x1  31        Ready      0x400      0x22c      0x80cfbf2c  0x80cfbc90  0x0        0xffff  77      78         78        -7566584
system_wq       0x2  10        Pend       0x6000     0x23c      0x80d01f90  0x80cfc138  0x0        0x2     0       0          0         0
app_Task        0x3  10        Delay      0x6000     0x3450     0x80d07f38  0x80d02150  0x1        0xffff  0       0          0         21010728
shellTask       0x4  9         Running    0x3000     0x2810     0x80d10084  0x80d0d768  0xffff     0xffff  0       0          0         -12148
cmdParesTask    0x5  9         Pend       0x1000     0x26c      0x80d11640  0x80d10818  0x1        0xffff  0       0          0         216
pth01           0x6  10        Delay      0x6000     0x26b0     0x80d176c8  0x80d11868  0x0        0xffff  0       0          0         -340
RcvAndDispatch  0x7  10        Ready      0x6000     0x25c      0x80df8d18  0x80df2ee0  0x0        0xffff  0       0          0         -2704
SavePara2Flash  0x8  10        Ready      0x6000     0x234      0x80dfee60  0x80df9000  0x0        0xffff  0       0          0         -340
ProcessEncStreamThread  0x9  10  Ready    0x10000    0xc40      0x80e11440  0x80e016b8  0x1        0xffff  3       4          4         8593536
UpdateTimeOsdThread  0xa  10   Delay      0x6000     0x860      0x80e174d8  0x80e116f0  0x0        0xffff  0       0          0         3008
IspRun          0xb  10        Ready      0x6000     0x1604     0x80e206d8  0x80e1a9f0  0x1        0xffff  11      9          10        -485952
aenc_get        0xc  10        Pend       0x12000    0x3f4      0x80e9fad8  0x80e8e238  0x1        0xffff  0       0          0         0
SendAFrameToAencThread  0xd  10  Ready     0x6000     0x5a3c     0x80ea5f70  0x80ea0250  0x0        0xffff  4       4          4         -345056
WORKSTATE_CHECK  0xe  10        Ready      0x6000     0x234      0x81cf23b0  0x81cee550  0x0        0xffff  0       0          0         -3136
himci_Task      0xf  10        Ready      0x800      0x4d4      0x81cf4268  0x81cf3c20  0x1        0xffff  0       0          0         653876
himci_Task      0x10 10        Ready      0x800      0x460      0x81cf5ad8  0x81cf5490  0x0        0xffff  0       0          0         -40
TIMER_CHECK     0x11 10        Delay      0x6000     0x7cc      0x81cfbaf0  0x81cf5cb0  0x1        0xffff  0       0          0         -1060
```

如图 7-9 所示结果：举例，任务名为 shellTask

`StackSize = 0x3000`（创建该任务时分配的栈大小）

`WaterLine = 0x2810`（水线，目前为止该任务栈已经被使用的内存大小）

`StackPoint = 0x80d10084` （任务栈指针， 指向该任务当前的地址）

`Top0fStack = 0x80d0d768`（栈顶）

`MaxStackPoint = Top0fStack + StackSize = 0x80d10768`(得到该任务栈最大的可访问地址)

对比`WaterLine`和`StackSize`， 若`WaterLine`大于`StackSize`，则说明该任务踩内存

观察`StackPoint` ，理论上 `StackPoint<= MaxStackPoint && StackPoint >= Top0fStack`,

若不在这个范围内，则说明任务栈踩内存。