外围设备驱动

操作指南

文档版本　　00B07

发布日期　　2017-04-28

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

本文档主要是指导使用以太网、SD/MMC 卡等驱动模块的相关人员，通过一定的步骤和方法对和这些驱动模块相连的外围设备进行控制，主要包括操作准备、操作过程、操作中需要注意的问题以及操作示例。

说明

- 本文以 Hi3516A 为例，未有特殊说明，Hi3519V100/Hi3519V101、Hi3518EV20X、Hi3516CV300 与 Hi3516A 一致。
- 文档中的路径指的是 Huawei LiteOS 源代码根目录下的相对路径。
- 应用通过 config.mk 中的 LITEOS_LIBS 来链接对应的驱动库。

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

读者对象

本文档（本指南）主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

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。

修订日期	版本	修订说明
2017-04-28	00B07	新增 2.9 小节 3.2、3.5、4.3、4.5、4.6、5.5 和 8.4 都涉及修改
2017-01-20	00B06	2.1、3.3.2 和 3.3.3 小节涉及修改
2016-08-25	00B05	第 5 次临时版本发布。 增加 Hi3516CV300 内容。
2016-06-20	00B04	第 4 次临时版本发布。 增加 Hi3519V101 内容。
2016-05-10	00B03	第 3 次临时版本发布。 补充第 1 章的相关内容 2.3 小节，5.2.1 至 5.2.3 小节有修改。
2016-02-26	00B02	第 2 次临时版本发布。 2.2、2.3 和 3.2 小节涉及修改；5.2.3 小节添加相关内容，新增 5.2.4 小节；6.1 和 7.1 小节涉及修改。
2015-12-10	00B01	第 1 次版本发布。

目 录

表目录

1 以太网操作指南

1.1 功能介绍

Hi35xx 支持以太网交换接口。

GMAC 是 Hi3516A、Hi3519V100/Hi3519V101 的千兆以太网交换接口模块。GMAC 模块支持 PHY 接口模式有 rgmii、rmii 和 mii。

HIETH-SF 是 Hi3518EV200、Hi3518EV201、Hi3516CV200、Hi3516CV300 的百兆以太网交换接口模块。HIETH-SF 模块支持 PHY 接口模式有 rmii 和 mii。

1.2 重要概念

1.2.1 PHY 接口模式

以太网模块支持 PHY 接口模式有 rgmii、rmii 和 mii，发布包中默认配置为 rmii，若需配置成 rgmii 或 mii，需要重新配置 u-boot 环境变量和 Huawei LiteOS 源码。

a. U-boot 下通过环境变量设置：

```
setenv mdio_intf rgmii 或者 setenv mdio_intf mii
```

b. Huawei LiteOS 中通过调用以下函数接口修改 PHY 接口：

```
hisi_eth_set_phy_mode("rmii");
```

说明

- 用户可参考源码目录下 sample/sample_osdrv/sample_hi35xx.c 关于 PHY 接口模式配置的用法示例。
- Hi3516A、Hi3519V100/ Hi3519V101 支持 PHY 接口模式有 rgmii、rmii 和 mii
- Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300 支持 PHY 接口模式有 rmii 和 mii

1.2.2 PHY 地址配置

Huawei LiteOS 可通过以下函数接口修改 Phy 地址：

hisi_eth_set_phy_addr(0);

说明
用户可参考源码目录下 sample/sample_osdrv/sample_hi35xx.c 关于 PHY 地址修改的用法示例。

1.3 模块编译

以太网模块源码路径为 drivers/higmac 和 drivers/hieth-sf。Huawei LiteOS 发布时已经根据发布的芯片指定好对应的以太网模块，用户在 Huawei LiteOS 根目录下执行 make 即可编译出对应的以太网模块库。编译成功后，out/hi35xx/lib 目录下会生成名为 libhigmac.a 或 libhieth-sf.a 的库文件。

说明
应用通过 config.mk 中的 LITEOS_LIBS 来链接对应的以太网库：

- libhigmac.a（Hi3516A、Hi3519V100/Hi3519V101）
- libhieth-sf.a（Hi3518EV200、Hi3518EV201、Hi3516CV200、Hi3516CV300）

1.4 以太网模块使用示例

1.4.1 以太网初始化示例

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

hisi_eth_set_phy_mode("rmii");   //设置 PHY 接口模式

hisi_eth_set_phy_addr(0);        //设置 PHY 地址

tcpip_init(NULL, NULL);          //初始化 tcpip 协议栈

higmac_init();                   // Hi3516A、Hi3519V100/ Hi3519V101 时初始化 GMAC 模块

netif_set_up(pnetif);
```

说明

- Hi3516A、Hi3519V100/Hi3519V101 的以太网模块初始化接口为 higmac_init()函数
- Hi3518EV200、Hi3518EV201、Hi3516CV200、Hi3516CV300 的以太网模块初始化接口为 hisi_eth_init()函数

1.5 网络相关的 shell 命令

1.5.1 ifconfig 命令

ifconfig 原是 linux 中用于显示或配置网络设备的命令，英文全称是 network interfaces configuring。通过 ifconfig 命令可以方便地对网络设备进行快速设置。ifconfig 命令用法与常用设置参数：

- 不带参数：当 ifconfig 命令不带参数时，输出为当前网络设置。
- 带参数用法：

- 格式：

  ```
  ifconfig [网卡名称] [需设置对象] [设置值]
  ```

- 将指定网卡的 IP 地址设置为 192.168.0.1（格式：ifconfig 网络设备名 IP 地址）：

  ```
  ifconfig eth0 192.168.0.1
  ```

- 将指定网卡的子网掩码设置为 255.255.255.0：

  ```
  ifconfig eth0 netmask 255.255.255.0
  ```

- 修改 gateway:

  ```
  ifconfig eth0 gateway 192.168.1.4
  ```

- 修改 MAC 地址:

  ```
  ifconfig eth0 hw ether xx:xx:xx:xx:xx:xx
  ```

- 暂时关闭或启用网卡：

  关闭网卡：ifconfig eth0 down

  启用网卡：ifconfig eth0 up

1.5.2 PING 命令

PING (Packet Internet Groper)，因特网包探索器，用于测试网络连接量的程序。Ping 发送一个 ICMP(Internet Control Messages Protocol）即因特网信报控制协议；回声请求消息给目的地并报告是否收到所希望的 ICMP echo （ICMP 回声应答）。ping 也属于一个通信协议，是 TCP/IP 协议的一部分。

- 格式：ping [ip] [发送次数]
- 用法：ping 192.168.1.1 12

  ping 命令的用法比较简单。用户需要确认调用以太网模块初始化函数完成 GMAC 模块初始化后，按照编译、烧写步骤下载到单板后，重启单板，系统进入 shell 界面后，可以在命令行执行 ping 命令检查单板与服务器端网络连接情况。

1.6 API 参考

该功能模块提供以下接口：

- higmac_init：初始化 Hi3516A、Hi3519V100/ Hi3519V101 以太网模块。
- hisi_eth_init：初始化 Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300 以太网模块。
- hisi_eth_set_phy_mode：配置 PHY 接口模式。
- hisi_eth_set_phy_addr：配置 PHY 地址。

higmac_init

【描述】

初始化 Hi3516A、Hi3519V100/ Hi3519V101 以太网模块。

【语法】

```
void higmac_init(void)
```

【参数】

无

【返回值】

无

【需求】

- 头文件：higmac.h
- 库文件：libhigmac.a

【注意】

PHY 接口模式的配置和 PHY 地址的配置，需要在以太网模块初始化前配置。

【举例】

参考以太网模块使用示例。

【相关主题】

- hisi_eth_set_phy_mode
- hisi_eth_set_phy_addr

hisi_eth_init

【描述】

初始化 Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300 以太网模块。

【语法】

```
void hisi_eth_init(void)
```

【参数】

无

【返回值】

无

【需求】

- 头文件：无
- 库文件：libhieth-sf.a

【注意】

PHY 接口模式的配置和 PHY 地址的配置，需要在以太网模块初始化前配置。

【举例】

参考以太网模块使用示例。

【相关主题】

- hisi_eth_set_phy_mode
- hisi_eth_set_phy_addr

hisi_eth_set_phy_mode

【描述】

配置 PHY 接口模式。

【语法】

```
int hisi_eth_set_phy_mode(const char *phy_mode);
```

【参数】

参数名称	描述	输入/输出
phy_mode	PHY 接口模式。 取值范围： • Hi3516A、Hi3519V100/ Hi3519V101：字符串 mii、rmii、rgmii • Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300：字符串 mii、rmii	输入

【返回值】

返回值	描述
0	成功
-1	失败

【需求】

- 头文件：eth_drv.h
- 库文件：libhigmac.a（Hi3516A、Hi3519V100/ Hi3519V101）
  libhieth-sf.a（Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300）

【注意】

PHY 接口模式的配置，需要在以太网模块初始化前配置。

【举例】

参考以太网模块使用示例。

【相关主题】

- higmac_init
- hisi_eth_init

hisi_eth_set_phy_addr

【描述】

配置 PHY 地址。

【语法】

```
int hisi_eth_set_phy_addr(unsigned int phy_addr);
```

【参数】

参数名称	描述	输入/输出
phy_addr	PHY 地址。 取值范围： Hi3516A、Hi3519V100/ Hi3519V101：0~MAX_PHY_ADDR Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300：0~MAX_PHY_ADDR	输入

说明

- Hi3516A、Hi3519V100/ Hi3519V101：MAX_PHY_ADDR 值的定义在 drivers/higmac/include/eth_phy.h
- Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300：MAX_PHY_ADDR 值的定义在 drivers/hieth-sf/include/eth_phy.h

【返回值】

返回值	描述
0	成功
-1	失败

【需求】

- 头文件：eth_drv.h
- 库文件：libhidmac.a（Hi3516A、Hi3519V100/ Hi3519V101）
  libhieth-sf.a（Hi3518EV200、Hi3518EV201、Hi3516CV200 和 Hi3516CV300）

【注意】

PHY 地址的配置，需要在以太网模块初始化前配置。

【举例】

参考以太网模块使用示例。

【相关主题】

- higmac_init
- hisi_eth_init

2 SD/MMC/SDIO 操作指南

2.1 功能介绍

SD/MMC 控制器（以下简称 MMC）用于处理对 SD/MMC 卡（以下统称 SD 卡）的读写等操作，并通过 SDIO 协议实现对扩展外设（如蓝牙、WiFi 等）的支持。

- Hi3516A 提供 2 个 MMC 控制器。可支持 SD 卡，MMC 卡和 SDIO 接口。
- Hi3518EV200 提供 2 个 MMC 控制器。MMC 0 通过提供不同 MMC 驱动库，支持 SD 卡（SDIO 接口）或 eMMC 存储器，不可同时支持 SD 卡和 eMMC；MMC 1 支持 SD 卡（SDIO 接口）。
- Hi3519V100/ Hi3519V101 提供 3 个 MMC 控制器。其中，MMC 0、MMC 1 支持 SD 卡（SDIO 接口），MMC 2 支持 eMMC 存储器。
- Hi3516CV300 提供 4 个 MMC 控制器。其中 MMC 0、MMC 1、MMC3 支持 SD 卡(SD mem-version2.0)、SDIO 接口(SDIO-version 2.0)，MMC 2 支持 eMMC 存储器，默认 MMC3 控制器不使能。

2.2 重要概念

分区

目前 Huawei LiteOS 不支持用户自定义分区 SD 卡，eMMC 可通过以下方法自定义分区：

步骤 1. 声明外部 eMMC 分区信息结构体

```
extern struct disk_divide_info emmc;
```

步骤 2. 自定义分区地址并增加分区

```
size_t part0_start_sector = 16   * (0x100000/512);
size_t part0_count_sector = 1024 * (0x100000/512);
add_mmc_partition(&emmc, part0_start_sector, part0_count_sector);
```

说明

对 eMMC 进行分区操作，务必在初始化 MMC 模块前，关于 MMC 的初始化详见 2.4.1 。

----结束

挂载

用户可在程序中通过调用挂载函数实现对 SD 卡的挂载操作。使用挂载函数前，务必确定设备节点已生成。否则会挂载失败。

挂载函数原型为：

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

使用示例（示例把“/dev/mmcblk0p0”分区以 vfat 格式挂载到“/sd”目录下）：

```
mount("/dev/mmcblk0p0", "/sd", "vfat", 0, 0);
```

卸载

用户挂载 SD 卡分区成功后，可通过调用卸载函数实现 SD 卡的卸载操作。

卸载函数原型为：

```
int umount(const char *target)
```

使用示例（示例将“/dev/mmcblk0p0”分区挂载的“/sd”目录卸载）：

```
umount("/sd");
```

格式化

用户需要在程序中对分区运行格式化时，可调用系统提供的格式化函数，其函数原型如下：

```
int format(consst char *dev, int sectors, int option)
```

使用示例（示例把“/dev/mmcblk0p0”分区格式化）：

```
format("/dev/mmcblk0p0", 128, 2);
```

其中，option 的传值如下：

1：把设备格式化成 FAT16 类型

2：把设备格式化成 FAT32 类型

4：把设备格式化成 exFAT 类型

其它值：由系统自动选择类型

2.3 模块编译

源码路径为 drivers/mmc。用户需要对 MMC 设备进行访问操作时，首先要在编译脚本里指定 MMC 源码路径与头文件路径。编译成功后，out/hi35xx/lib 目录下会生成名为 libmmc.a 的库文件。链接时需要通过-lmmc 参数指定该库文件。

对于 Hi3518EV200，MMC0 控制器可支持 SD 或 eMMC。可以根据需要，编译链接时，选择支持 SD 的 libhi3518ev200_sd.a 库或支持 eMMC 的 libhi3518ev200_emmc.a 库实现。

2.4 使用示例

2.4.1 模块初始化

在初始化函数中调用以下接口实现 MMC 驱动初始化：

```
SD_MMC_Host_init();
```

2.4.2 访问 SD 卡

编辑、编译源码，下载可执行文件至单板。系统重启后，完成 MMC 驱动初始化。如果有 SD 卡插入，根据 Huawei LiteOS/dev 目录下生成的节点信息，可通过命令 mount、format 对生成的结点进行挂载或格式化操作。

节点分区信息 mount 后，可对分区进行读写操作。

2.5 相关的 shell 命令

mount

对 SD 卡设备分区挂载操作。

- 格式：mount　[节点信息]　[目录]　[文件系统格式]
  注：目前[文件系统格式]仅支持 vfat 格式
- 用法示例：mount /dev/mmcblk0p0 /sd0 vfat
  将 mmcblk0p0 节点 挂载到/sd0 目录。

statfs

对已挂载的 SD 卡分区查看分区信息。

- 格式：statfs　[目录]
- 用法示例：statfs /sd0
  注：如果没有/sd0 目录没有 mount 过，查看的分区信息无效

umount

对已挂载的分区卸载。

- 格式：umount [目录]
- 用法示例：umount /sd0
  将/sd0 目录卸载，如果没有/sd0 目录没有 mount 过，卸载操作会失败

format

对 SD 卡的分区节点格式化。

- 格式：format [设备节点] [每簇的扇区数] [选项]
  选项取以下值：
  1：把设备格式化成 FAT16 类型
  2：把设备格式化成 FAT32 类型
  4：把设备格式化成 exFAT 类型
  其它值：由系统自动选择类型
- 用法示例：format /dev/mmcblk0p0 128 4
  将 mmcblk0p0 节点格式化成 exFAT 格式。

2.6 操作须知

- 正常操作过程中需要遵守的事项：
  - 保证卡的金属片与卡槽硬件接触充分良好（如果接触不好，会出现检测错误或读写数据错误），测试薄的 MMC 卡，必要时可以用手按住卡槽的通讯端测试。
  - 每次插入 SD 卡后，需要做一次 mount 操作挂载文件系统，才能读写 SD 卡。
- 不能进行的操作：
  - 读写 SD 卡时不要拔卡，否则会打印一些异常信息，并且可能会导致卡中文件或文件系统被破坏。
  - 格式化 SD 卡时禁止拔卡，否则可能会造成 SD 卡永久性损坏。
- 可能出现异常的操作：
  拔卡后，再极其快速地插入卡可能会出现检测不到卡的现象，因为卡的检测注册/注销过程需要一定的时间。此时可再拔出卡，过几秒插入即可。

2.7 API 参考

2.7.1 通过文件系统读写 SD 卡

步骤 1. 挂载设备结点：

```
mount("/dev/mmcblk0p0", "/sd0p0", "vfat", 0, 0);
```

说明
如未完成设备文件的注册工作，可调用 SD_MMC_Host_init（）函数注册设备文件。

步骤 2. 创建文件

```
fd = open("file_test", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
```

步骤 3. 读写文件

```
write(fd, w_buf, 64);
read(fd, r_buf, 64);
```

说明
r_buf 与 w_buf 为用户读写准备的 buf 地址。

步骤 4. 关闭文件

```
close(fd);
```

----结束

2.7.2 裸读写 eMMC

【描述】

对 eMMC 存储介质进行直接读写操作。Hi3516A 平台不支持该接口，Hi3518EV200、Hi3519V100/ Hi3519V101 和 Hi3516CV300 平台均支持 eMMC。

【语法】

```
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);
```

【参数】

参数名称	描述	输入/输出
buffer	读写数据保存的 buffer 地址	输入
start_sector	读写的起始块地址	输入
nsectors	写入块数量	输入

2.8 数据类型

disk_divide_info

【说明】

eMMC 多分区信息结构体。

【定义】

```
struct disk_divide_info{
    unsigned int   sector_count;
    unsigned int   part_count;
    struct partition_info part[MAX_DIVIDE_PART_PER_DISK];
};
```

【成员】

成员名称	描述
sector_count	分区块个数
part_count	分区号
part	分区信息结构

2.9 SDIO 对外接口使用说明

Huawei LiteOS 提供了一套完整的 SDIO 对外接口。SDIO 扩展外设通过调用这套接口实现对其操作，包括发送命令，收发数据等。对 Huawei LiteOS 的 SDIO 扩展外设开发步骤如下：

步骤 1. 包含 SDIO 对外接口必要的头文件。

```
#include "mmc/sdio_func.h"
```

此头文件声明了 SDIO 对外接口，定义了一个 struct sdio_func 用于 SDIO 外设操作 SDIO 设备。

步骤 2. 调用 sdio_get_func（接口说明如表 2-1，下同）获得一个 sdio_func，后续的调用都依赖此 sdio_func。

步骤 3. 调用 sdio_en_func 使能此 sdio_func。

步骤 4. 调用 sdio_require_irq 注册 SDIO 中断，SDIO 中断会在 SDIO 驱动收到 SDIO 外设中断时触发。（此步骤根据用户需要选做）。

步骤 5. sdio_func 中有一成员 void *data，可由用户赋值使用。（此步骤根据用户需要选做）。

表2-1 SDIO 对外接口说明

接口	传参	说明
sdio_get_func	uint32_t func_num uint32_t manufacturer_id uint32_t device_id	获得一个 sdio_func 此 sdio_func 在驱动到 SDIO 设备识别后产生，通过 func 号、厂商 id、设备 id（此三个参数根据 SDIO 外设获

接口	传参	说明
		得）唯一标识
sdio_en_func	struct sdio_func *func	使能此 sdio_func CMD52，CCCR IOEx 地址和 CCCR IORx 地址
sdio_dis_func	struct sdio_func *func	禁能 sdio_func CMD52，CCCR IOEx 地址
sdio_require_irq	struct sdio_func *func sdio_irq_handler_t *handler	注册 sdio 中断 SDIO 中断触发后会调用此注册 handler（此接口如 void (sdio_irq_handler_t)(struct sdio_func *); 由用户定义）
sdio_release_irq	struct sdio_func *func	释放 sdio 中断
sdio_read_byte	struct sdio_func *func uint32_t addr int *err_ret	按给定地址 addr 读取一字节数据 CMD52，返回读取的数据
sdio_read_byte_ext	struct sdio_func *func uint32_t addr int *err_ret unsigned in	按给定地址 addr 读取一字节数据 CMD52，增加一个 in 添加到 CMD 参数中，返回读取的数据
sdio_read_incr_block	struct sdio_func *func void *dst uint32_t addr int count	按给定地址 addr 读取指定长度 count 数据到 dst 中 CMD53，读取的设备地址依次增长(比如设备的内存地址)
sdio_read_fifo_block	struct sdio_func *func void *dst uint32_t addr int count	按给定地址 addr 读取指定长度 count 数据到 dst 中 CMD53，读取的设备地址固定(比如设备的 FIFO)
sdio_write_incr_block	struct sdio_func *func uint32_t addr void *src int count	将以 src 地址，长度 count 的数据写到给定设备地址 addr 中 CMD53，写入的设备地址依次增长(比如设备的内存地址)
sdio_write_fifo_block	struct sdio_func *func uint32_t addr void *src int count	将以 src 地址，长度 count 的数据写到给定设备地址 addr 中 CMD53，写入的设备地址固定(比如设备的 FIFO)
sdio_write_byte	struct sdio_func *func uint8_t byte	按给定地址 addr 写入一字节数据 byte CMD52

接口	传参	说明
	uint32_t addr int *err_ret	
sdio_write_byte_raw	struct sdio_func *func uint8_t write_byte uint32_t addr int *err_ret	按给定地址 addr 写入一字节数据 byte CMD52，写完后读回（设置 RAW Flag）
sdio_func0_read_byte	struct sdio_func *func uint32_t addr int *err_ret	按给定地址 addr 读取一字节数据 CMD52，读取 func0，返回读取的数据
sdio_func0_write_byte	struct sdio_func *func unsigned char byte uint32_t addr int *err_ret	按给定地址 addr 写入一字节数据 byte CMD52，写入 func0
sdio_set_cur_blk_size	struct sdio_func *func uint32_t blksz	配置 SDIO 当前块大小，此块小大不应该大于 512 配置后应该调用 sdio_enable_blksz_for_byte_mode 使能 byte 模式
sdio_enable_blksz_for_byte_mode	struct sdio_func *func uint32_t enable	使能 byte 模式

3 I²C 操作指南

3.1 功能介绍

I²C 模块的作用是完成 CPU 对 I²C 总线上连接的从设备的读写。

3.2 模块编译

源码路径为 drivers/i2c。用户需要对 I²C 设备进行访问操作时，首先要在编译脚本里指定 I²C 源码路径与头文件路径。编译成功后，out/hi35xx/lib 目录下会生成名为 libi2c.a 的库文件。链接时需要通过-li2c 参数指定该库文件。

说明
文档中的路径指的是 Huawei LiteOS 源代码根目录或其相对路径。

3.3 使用示例

3.3.1 模块初始化

此操作示例介绍如何初始化 I²C 驱动。

步骤 1. 驱动初始化，调用如下接口：

```
i2c_dev_init();
```

步骤 2. 开发者需要根据设备硬件特性配置相关的管脚复用。

具体请参考如下：

- 《Hi3516A/Hi3516D 专业型 HD IP Camera Soc 用户指南》和《Hi3518EV20X/Hi3516CV200 经济型 HD IP Camera Soc 用户指南》中管脚复用控制寄存器章节；
- Hi3519V100/ Hi3519V101 请参考《Hi3519V100_PINOUT_CN》/《Hi3519V101_PINOUT_CN》中管脚复用寄存器页签。
- Hi3516CV300 请参考《Hi3516CV300_PINOUT_CN》中管脚复用寄存器页签。

步骤 3. 用户可根据需要调用模块的读写函数对设备进行访问。参考 3.3.2 及 3.3.3 。

----结束

3.3.2 通过调用驱动函数访问 I²C 设备

步骤 1. 定义一个 I²C 设备描述结构体。

```
struct i2c_client client;
```

步骤 2. 调用 client_attach 把 client 关系到对应的控制器上。

函数原型：

```
int client_attach(struct i2c_client * client, int adapter_index)
```

adapter_index:被关联的i2c总线号的值，例如需要操作i2c0，则该值为0。

步骤 3. 调用 I²C 提供的标准读写函数对外围器件进行读写。

Hi3516CV300 平台调用以下接口：

```
读：ret = i2c_transfer(struct i2c_adapter *adapter, struct i2c_msg *msgs,
int count);
写：ret = i2c_master_send(struct i2c_client *client, const char *buf, int
count);
```

Hi3516A、Hi3519V100/Hi3519V101 和 Hi3518EV20X 平台则调用以下接口：

```
读：i2c_master_recv(const struct i2c_client *client, char *buf, int count)
写：i2c_master_send(struct i2c_client *client, const char *buf, int count)
```

代码示例如下：

说明

此代码为读写 I²C 外围设备的示例程序，仅为客户调用 I²C 驱动程序访问外围设备提供参考，不提供实际应用功能。

```
struct  i2c_client  i2c_client_obj; //i2c控制结构体
#define SLAVE_ADDR 0x34  //i2c设备地址
#define SLAVE_REG_ADDR 0x300f //i2c设备寄存器地址
/* client 初始化 */
int i2c_client_init(void)
{
    int ret = 0;
    struct i2c_client * i2c_client0 = &i2c_client_obj;
    i2c_client0->addr = SLAVE_ADDR >> 1;  //外围器件地址，其中Hi3516A与
Hi3518EV200不需要做右移操作
    ret = client_attach(i2c_client0, 0);
    if(ret) {
        dprintf("Fail to attach client!\n");
```

```
        return -1;
    }
    return 0;
}
UINT32 sample_i2c_write(void)
{
    int ret;
    struct i2c_client * i2c_client0 = & i2c_client_obj;
    char buf[4] = {0};
    i2c_client_init();

    buf[0] = SLAVE_REG_ADDR & 0xff;
    buf[1] = (SLAVE_REG_ADDR >> 8) & 0xff;
    buf[2] = 0x03; //往i2c设备写入的值

    //调用I2C驱动标准写函数进行写操作
    ret = i2c_master_send(i2c_client0, &buf, 3);
    return ret;
}
```

Hi3516CV300 读使用示例：

```
UINT32 sample_i2c_read(void)
{
    int ret;
    struct i2c_client *i2c_client0 = & i2c_client_obj;
    struct i2c_rdwr_ioctl_data rdwr;
    struct i2c_msg msg[2];
    unsigned char recvbuf[4];

    memset(recvbuf, 0x0 ,4);
    i2c_client_init();

    msg[0].addr = SLAVE_ADDR >> 1;
    msg[0].flags = 0;
    msg[0].len = 2;
    msg[0].buf = recvbuf;

    msg[1].addr = SLAVE_ADDR >> 1;
    msg[1].flags = 0;
    msg[1].flags |= I2C_M_RD;
    msg[1].len = 1;
    msg[1].buf = recvbuf;

    rdwr.msgs = &msg[0];
```

```
    rdwr.nmsgs = 2;

    recvbuf[0] = SLAVE_REG_ADDR & 0xff;
    recvbuf[1] = (SLAVE_REG_ADDR >> 8) & 0xff;

    i2c_transfer(i2c_client0->adapter, msg, rdwr.nmsgs);
    dprintf("val = 0x%x\n",recvbuf[0]); //buf[0] 保存着从i2c设备读写的值
return ret;
}
```

Hi3516A、Hi3519V100/Hi3519V101 和 Hi3518EV20X 平台读写使用示例：

```
UINT32 sample_i2c_read (void)
{
    int ret;
    struct i2c_client * i2c_client0 = & i2c_client_obj;
    char buf[4] = {0};
    i2c_client_init();

    buf[0] = SLAVE_REG_ADDR & 0xff;
    buf[1] = (SLAVE_REG_ADDR >> 8) & 0xff;

    ret = i2c_master_recv (i2c_client0, &buf, 3);
    return ret;
}
```

----结束

3.3.3 通过设备节点操作访问 I²C 设备

步骤 1. 打开 I²C 总线对应的设备文件，获取文件描述符：

```
fd = open("/dev/i2c-0",O_RDWR);
```

说明

如未完成设备文件的注册工作，可调用 i2c_dev_init（）函数注册设备文件。

步骤 2. 通过 ioctl 设置外围设备地址、外围设备寄存器位宽和数据位宽：

```
ret = ioctl(fd, I2C_SLAVE_FORCE, device_addr);
ioctl(fd, I2C_16BIT_REG, 0);
ioctl(fd, I2C_16BIT_DATA, 0);
```

注意

相关宏定义在 drivers/i2c/include/ hi_i2c.h 头文件中。

步骤 3. Hi3516A、Hi3519V100、Hi3519V101 和 Hi3518EV20X 使用 read/wite 进行数据读写：

```
read(fd, buf, count);
write(fd, buf, count);
```

Hi3516CV300 使用以下函数进行数据读写：

```
ioctl(fd, I2C_RDWR, &rdwr);
write(fd, buf, count);
```

说明

- 步骤 2 中，设置寄存器位宽和数据位宽时，ioctl 的第三个参数为 0 表示 8bit 位宽，为 1 表示 16bit 位宽。
- 步骤 3 中，Hi3516CV300 使用 ioctl 进行读写的相关宏定义在 drivers/i2c/include/i2c-dev.h 头文件中。

Hi3516A、Hi3519V100、Hi3519V101、Hi3518EV20X 和 Hi3516CV300 代码示例请参考文件：

```
drivers/i2c/src/i2c_shell.c
```

说明

- 此代码为示例程序，仅为客户通过文件系统访问 I²C 外围设备操作提供参考，不提供实际应用功能。
- 用户调用 read、write 接口读写 i2c 操作时，buf 包含了寄存器地址与需要操作的数据字节，count 为寄存器地址所占字节数与需要操作数据字节数的总和。
- 其中，示例代码中使用“LOSCFG_HOST_TYPE_HIBVT”宏定义包括的用法为 Hi3516CV300 平台的实现。

----结束

3.4 shell 命令

i2c_read 命令

在控制台使用 i2c_read 命令对 I²C 外围设备进行读操作：

```
i2c_read <i2c_num> <device_addr> <reg_addr> <end_reg_addr>  [reg_width]
[data_width] [addr_width]
```

例如读挂载在 I²C 控制器 0 上的 IMX178 设备的 0x3000 到 0x3010 寄存器：

```
i2c_read 0x0 0x34 0x3000 0x3010 2 1 7
```

说明

i2c_num：I²C 控制器序号（对应《Hi35xx xx HD IP Camera SoC 用户指南》中的 I²C 控制器 0、1、2）。

- device_addr：外围设备地址。
- reg_addr：读外围设备寄存器操作的开始地址。
- end_reg_addr：读外围设备寄存器操作的结束地址。
- reg_width：外围设备的寄存器位宽（Hi3516A 支持 8/16bit，2: 16bit/1:8bit）。
- data_width：外围设备的数据位宽（Hi3516A 支持 8/16bit，2: 16bit/1:8bit）。

- addr_width：外围设备地址位宽（Hi3516A 支持 7/10 位位宽，7:7bit/10:10bit)。

i2c_write 命令

在控制台使用 i2c_write 命令对 I²C 外围设备进行写操作：

```
i2c_write <i2c_num> <device_addr> <reg_addr> <reg_value> [reg_width]
[data_width] [addr_width]
```

例如向挂载在 I²C 控制器 0 上的 IMX178 设备的 0x300f 寄存器以 400K 速率写入数据 0x10：

```
i2c_write 0x0 0x34 0x300f 0x00 2 1 7
```

说明

i2c_num：I²C 控制器编号（对应《Hi35xx xx HD IP Camera SoC 用户指南》中的 I²C 控制器 0、1、2）。

- device_addr：外围设备地址。
- reg_addr：写外围设备寄存器操作的地址。
- reg_value：写外围设备寄存器操作的数据。
- reg_width：外围设备的寄存器位宽（Hi3516A 的 I²C 控制器支持 8/16bit）。
- data_width：外围设备的数据位宽（Hi3516A 的 I²C 控制器支持 8/16bit）。
- addr_width：外围设备地址位宽（Hi3516A 支持 7/10 位位宽），7:7bit/10:10bit。

3.5 API 参考

3.5.1 API 需求

【需求】

- 头文件：hi_i2c.h
- 库文件：libi2c.a

该功能模块提供以下接口：

- i2c_dev_init：用于初始化。
- i2c_master_recv：用于读取 I²C 数据的函数接口。
- i2c_master_send：用于写入 I²C 数据的函数接口。
- i2c_transfer：用于 I²C 传输的函数接口。
- client_attach：用于关联 client 与 adapter。
- client_deinit：用于去关联 client 与 adapter。

i2c_dev_init

【描述】

I²C 设备初始化。

【语法】

```
int i2c_dev_init(void);
```

【参数】

无

【返回值】

返回值	描述
0	操作成功
其它	操作失败

i2c_master_recv

【描述】

用于读取 I²C 数据的函数接口。

【语法】

```
int i2c_master_recv(const struct i2c_client *client, char *buf, int
count);
```

【参数】

参数名称	描述	输入/输出
client	I²C 设备描述结构体	输入
buf	数据保存 buffer	输出
count	传输的字节数	输入

【返回值】

返回值	描述
非负值	读写长度
负值	读写失败

i2c_master_send

【描述】

用于写入 I²C 数据的函数接口。

【语法】

```
int i2c_master_send(const struct i2c_client *client, char *buf, int
count);
```

【参数】

参数名称	描述	输入/输出
client	I²C 设备描述结构体	输入
buf	数据保存 buffer	输入
count	传输的字节数	输入

【返回值】

返回值	描述
非负值	读写长度
负值	读写失败

i2c_transfer

【描述】

用于写入 I²C 数据的函数接口。

【语法】

```
int i2c_transfer(struct i2c_adapter *adapter, struct i2c_msg *msgs, int
count);
```

【参数】

参数名称	描述	输入/输出
adapter	I²C 控制器 adatper	输入
msgs	待发送 msg 数组	输入
count	需要被发送的 msg 个数	输入

【返回值】

返回值	描述
非负值	被发送的 msg 个数
负值	读写失败

client_attach

【描述】

用于关联 client 与 adapter。

【语法】

```
int client_attach(struct i2c_client * client, int adapter_index);
```

【参数】

参数名称	描述	输入/输出
client	client 结构体	输入
adapter_index	关联的 host 序号	输入

【返回值】

无

返回值	描述
0	操作成功
其它	操作失败

client_deinit

【描述】

用于去关联 client 与 adapter。

【语法】

```
int client_deinit(struct i2c_client * client);
```

【参数】

参数名称	描述	输入/输出
client	client 结构体	输入

【返回值】

返回值	描述
0	操作成功

返回值	描述
其它	操作失败

3.5.2 芯片差异

芯片类型	差异
Hi3516A	3 个 I^2C 控制器
Hi3518E	3 个 I^2C 控制器
Hi3519V100/ Hi3519V101	4 个 I^2C 控制器
Hi3516CV300	3 个 I^2C 控制器

3.6 数据类型

i2c_client

【说明】

I^2C 从设备结构体。

【定义】

```
struct i2c_client {
    unsigned short flags;
    unsigned short addr;
    char name[I2C_NAME_SIZE];
    struct i2c_adapter *adapter;
    struct i2c_driver *driver;
    struct device dev;
    int irq;
    struct list_head detected;
};
```

【成员】

成员名称	描述
flags	标志信息
addr	地址
name	名称

成员名称	描述
adapter	适配层结构体指针
driver	驱动层结构体指针
dev	设备结构体
irq	中断号
detected	链表头

4 SPI 操作指南

4.1 功能介绍

SPI 模块的作用是访问 SPI 总线上连接的从设备。

4.2 模块编译

源码路径为 drivers/spi，要在编译脚本里指定 SPI 源码路径与头文件路径，编译成功后，out/hi35xx/lib 目录下会生成名为 libspi.a 的库文件，链接时需要通过-lspi 参数指定该库文件。

说明
文档中的路径指的是 Huawei LiteOS 源代码根目录下的相对路径。

4.3 使用示例

4.3.1 模块初始化

此操作示例介绍如何初始化 SPI 驱动。

步骤 1. 驱动初始化，调用如下接口：

```
spi_dev_init();
```

步骤 2. 开发者需要根据设备硬件特性配置相关的管脚复用。

具体请参考如下：

- 《Hi3516A/Hi3516D 专业型 HD IP Camera Soc 用户指南》、《Hi3518EV20X/Hi3516CV200 经济型 HD IP Camera Soc 用户指南》中管脚复用控制寄存器章节；
- Hi3519V100/Hi3519V101 请参考《Hi3519V100_PINOUT_CN》/《Hi3519V101_PINOUT_CN》中管脚复用寄存器页签。
- Hi3516CV300 请参考《Hi3516CV300_PINOUT_CN》中管脚复用寄存器页签。

步骤 3. 用户可根据需要调用模块的读写函数对设备进行访问。参考 4.3.2 及 4.3.3 。

----结束

4.3.2 通过调用驱动函数访问 SPI 设备

步骤 1. 定义一个 SPI 数据传输结构体：

```
struct spi_ioc_transfer transfer[1];
```

步骤 2. 初始化 spi_ioc_transfer：

```
transfer[0].tx_buf = (uint32_t)buf;
transfer[0].rx_buf = (uint32_t)buf;
transfer[0].len = DEV_WIDTH + REG_WIDTH + DATA_WIDTH;
transfer[0].cs_change = 1;
```

说明

- SPI_NUM 为需要操作的 spi 总线号。
- cs_change 为是否在发送完成后失效片选，值为 1，则片选会在发送完成后失效设备，否则不失效。

步骤 3. 调用 spidev_transfer 函数访问 spi 设备：

```
retval = spi_dev_set(bus_num, csn, &transfer[0]);
```

代码示例如下：

说明

此代码为读写 SPI 外围设备的示例程序，仅为客户调用 SPI 驱动程序访问外围设备提供参考，不提供实际应用功能。

```
#define DEV_WIDTH  1
#define REG_WIDTH 2
#define DATA_WIDTH 1
#define SPI_NUM 0
#define SPI_DEVICE_ADDR 0x82
#define SPI_REG_ADDR 0x0

int ssp_func_read(unsigned char *data)
{
    unsigned char buf[0x10];
    struct spi_ioc_transfer transfer[1];
    int retval = 0;

    transfer[0].tx_buf = buf;
    transfer[0].rx_buf = buf;
    transfer[0].len = DEV_WIDTH + REG_WIDTH + DATA_WIDTH;
    transfer[0].cs_change = 1;
    transfer[0].speed = 20000;
    memset(buf, 0, sizeof(buf));
```

```
    /*请根据spi的发送方式配置是高位在前或低位在前配置读写位*/
    buf[0] = (SPI_DEVICE_ADDR & 0xff) | 0x80 ;
    buf[1] = 0x0;
    buf[2] = (SPI_REG_ADDR & 0xff);

    retval = spi_dev_set(0,0,&transfer[0]);
    *data =  buf[DEV_WIDTH + REG_WIDTH];
    return retval ;
}

int ssp_func_write(void)
{
    unsigned char buf[0x10];
    struct spi_ioc_transfer transfer[1];
    int retval = 0;

    transfer[0].tx_buf = buf;
    transfer[0].rx_buf = buf;
    transfer[0].len = DEV_WIDTH + REG_WIDTH + DATA_WIDTH;
    transfer[0].cs_change = 1;
    transfer[0].speed = 20000;
memset(buf, 0, sizeof(buf));

    /*请根据spi的发送方式配置是高位在前或低位在前配置读写位*/
    buf[0] = (SPI_DEVICE_ADDR & 0xff) &(~0x80) ;
    buf[1] = 0x0;
    buf[2] = (SPI_REG_ADDR & 0xff);
    buf[3] = 0x1; //需要写入的数据
    retval = spi_dev_set(0,0,&transfer[0]);
    return retval;
}
```

----结束

4.3.3 通过设备节点操作访问 SPI 设备

步骤 1. 打开 SPI 总线对应的设备文件，获取文件描述符：

```
fd = open("/dev/spidev0.0", O_RDWR);
```

说明

如未完成设备文件的注册工作，可调用 spidev_init 函数注册设备文件。

步骤 2. 通过 ioctl 设置 SPI 传输模式：

```
value = SPI_MODE_3 | SPI_LSB_FIRST;
ret = ioctl(fd, SPI_IOC_WR_MODE, &value);
```

说明

- SPI_MODE_3 表示 SPI 的时钟和相位都为 1 的模式。
- SPI_LSB_FIRST 表示 SPI 传输时每个数据的格式为大端结束。

CAUTION

相关宏定义在 drivers/spi/include/spidev.h 头文件。

SPI 的时钟、相位、大小端模式可参考《Hi35xx xx HD IP Camera SoC 用户指南》。

步骤 3. 使用 ioctl 进行读写：

```
ret = ioctl(fd, SPI_IOC_MESSAGE, mesg);
```

说明

- mesg 表示传输一帧消息的 struct spi_ioc_transfer 结构体数组首地址。
- SPI_IOC_MESSAGE 表示全双工读写消息的命令。

代码示例如下：

说明

此代码为示例程序，仅为客户通过文件系统访问 SPI 外围设备操作提供参考，不提供实际应用功能。

```
#define SPI_DEV_ADDR 0x02  //器件地址
INT32  spi_dev_read(unsigned char reg_addr , unsigned char * data )
{
    struct file * fd = NULL;
    int ret = 0;
    unsigned char buf[0x10];
    struct spi_ioc_transfer transfer[1];

    fd = open("/dev/spidev0.0", O_RDWR);
    if ( fd == 0 ) {
        dprintf("open /dev/spidev0.0 fail! \n");
        return -1;
    }
    dprintf("open success !\n \n");

    value = SPI_MODE_3 | SPI_LSB_FIRST;
    ret = ioctl(fd, SPI_IOC_WR_MODE, &value);

    memset(transfer, 0, sizeof transfer);
    transfer[0].tx_buf = (uint32_t)buf;
    transfer[0].rx_buf = (uint32_t)buf;
```

```
    transfer[0].len = 3;
    transfer[0].cs = 0;

    buf[0] = SPI_DEV_ADDR | 0x80;    //dev addr
    buf[1] = reg_addr & 0xff ; //reg_addr
    buf[2] = 0;
    ret = ioctl(fd, SPI_IOC_MESSAGE, transfer);
    if (ret < 0)
    {
        dprintf("ioctl spi fail ! \n");
        close(fd);
        return -1;
    }
    * data = buf[2];
    close(fd);
    return 0 ;
}

INT32  spi_dev_write(unsigned char reg_addr , unsigned char reg_val)
{
    struct file * fd;
    int ret = 0;
    unsigned char buf[0x10];
    struct spi_ioc_transfer transfer[1];
    fd = open("/dev/spidev0.0", O_RDWR);
    if ( fd == 0 ) {
        dprintf("open /dev/spidev0.0fail! \n");
        return -1;
    }
    dprintf("open success !\n \n");
    value = SPI_MODE_3 | SPI_LSB_FIRST;
    ret = ioctl(fd, SPI_IOC_WR_MODE, &value);

    memset(transfer, 0, sizeof transfer);
    transfer[0].tx_buf = (uint32_t)buf;
    transfer[0].rx_buf = (uint32_t)buf;
    transfer[0].len = 3;
    transfer[0].cs = 0;

    buf[0] = SPI_DEV_ADDR & (~0x80);   //dev addr
    buf[1] = reg_addr & 0xff;       //reg_addr
    buf[2] = reg_val & 0xff;

ret = ioctl(fd, SPI_IOC_MESSAGE, transfer);
```

```
    if (ret < 0)
    {
        dprintf("ioctl spi fail ! \n");
        close(fd);
        return -1;
    }
    close(fd);
    return 0;
}
```

----结束

4.4 shell 命令

4.4.1 ssp_read 命令

在控制台使用 spi_read 命令对 SPI 外围设备进行读操作：

```
ssp_read <spi_num> <csn> <dev_addr> <reg_addr> [num_reg] [dev_width]
[reg_width] [data_width]
```

其中[num_reg] 可以省略，缺省值是 1（表示读 1 个寄存器）。

[dev_width] [reg_width] [data_width]可以省略，缺省值都是 1（表示 1Byte）。

例如读挂载在 SPI 控制器 0 片选 0 上设备地址为 0x2 的设备的 0x0 寄存器：

```
ssp_read 0x0 0x0 0x2 0x0 0x10 0x1 0x1 0x1
```

说明

- spi_num：SPI 控制器号（对应《Hi35xx xx HD IP Camera SoC 用户指南》中的 SPI 控制器编号）。
- csn：片选号（Hi3516A 的 SPI 控制器 0 有 1 个片选、控制器 1 有 3 个片选）
- dev_addr：外围设备地址。
- reg_addr：外围设备寄存器开始地址。
- num_reg：读外围设备寄存器个数。
- dev_width：外围设备地址位宽（支持 8 位）。
- reg_width：外围设备寄存器地址位宽（支持 8 位）。
- data_width：外围设备的数据位宽（支持 8 位）。

说明

此 SPI 读写命令仅支持 sensor 的读写操作。

4.4.2 ssp_rwrite 命令

在控制台使用 spi_write 命令对 SPI 外围设备进行写操作：

```
ssp_write <spi_num> <csn> <dev_addr> <reg_addr> <data> [dev_width]
[reg_width] [data_width]
```

其中[dev_width] [reg_width] [data_width]可以省略，缺省值都是 1（表示 1 Byte）。例如向挂载在 SPI 控制器 0 片选 0 上设备地址为 0x2 的设备的 0x0 寄存器写入数据 0x65：

```
ssp_write 0x0 0x0 0x2 0x0 0x65 0x1 0x1 0x1
```

说明

- spi_num：SPI 控制器序号（对应《Hi35xx xx HD IP Camera SoC 用户指南》中的 SPI 控制器编号）。
- csn：片选号（Hi3516A 的 SPI 控制器 0 有 1 个片选、控制器 1 有 3 个片选）。
- dev_addr：外围设备地址。
- reg_addr：外围设备寄存器地址。
- data：写外围设备寄存器的数据。
- dev_width：外围设备地址位宽（支持 8 位）。
- reg_width：外围设备寄存器地址位宽（支持 8 位）。
- data_width：外围设备的数据位宽（支持 8 位）。

说明

此 SPI 读写命令仅支持 sensor 的读写操作。

4.5 API 参考

spi_dev_set

【描述】

用于发起对 SPI 设备的读写操作。

【语法】

```
int hi_spidev_set(int host_no, int cs_no, struct spi_ioc_transfer
*transfer);
```

【需求】

- 头文件：spi.h hisoc/spi.h
- 库文件：libspi.a

【参数】

参数名称	描述	输入/输出
host_no	操作的控制器序号	输入
cs_no	片选	输入
transfer	传输控制结构体指针。 取值：用户定义并填充成员，参考 4.6 。	输入

【返回值】

返回值	描述
非负值	读写长度
负值	读写失败

【芯片差异】

芯片类型	差异
Hi3516A	2 个 SPI 控制器，SPI 控制器 0 有 1 个片选、控制器 1 有 3 个片选
HI3518E	2 个 SPI 控制器，SPI 控制器 0 有 1 个片选、控制器 1 有 2 个片选
HI3519V100	3 个 SPI 控制器，SPI 控制器 0 有 1 个片选、控制器 1 有 2 个片选、控制器 2 有 1 个片选
HI3519V101	4 个 SPI 控制器，SPI 控制器 0 有 1 个片选、控制器 1 有 2 个片选、控制器 2 有 1 个片选、控制器 3 有 1 个片选
Hi3516CV300	2 个 SPI 控制器，SPI 控制器 0 有 1 个片选、控制器 1 有 2 个片选

4.6 数据类型

spi_ioc_transfer：SPI 传输控制结构体。

spi_ioc_transfer

【说明】

SPI 传输控制结构体。

【定义】

```
struct spi_ioc_transfer {
    const char      *tx_buf;
    char            *rx_buf;
    unsigned        len;
    unsigned        cs_change;
    unsigned int    speed;
};
```

【成员】

成员名称	描述
tx_buf	写缓冲区地址

成员名称	描述
rx_buf	读缓冲区地址
len	读、写长度
cs_change	片选改变标志
speed	速率

5 UART 操作指南

5.1 功能介绍

UART 是一个异步串行的通信接口。UART 模块实现 Hi35xx 与其连接的其它 UART 设备(串行终端，MCU 等)收发通信。

5.2 模块编译

源码路径为 drivers/uart。用户需要对 UART 设备进行访问操作时，首先要在编译脚本里指定 UART 源码路径与头文件路径。编译成功后，out/hi35xx/lib 目录下会生成名为 libuart.a 的库文件。链接时需要通过-luart 参数指定该库文件。

5.3 使用示例

步骤 1. 在初始化函数中调用以下接口实现 UART 驱动注册：

```
uart_dev_init();
```

由于 shell 多用 UART0 作为通信交互与调试信息打印。应该完成以上 UAR 驱动注册，再进行 console，Shell 等与 UART 有依赖关系的模块初始化。

如果启用 dma 方式接收数据，要进行 dmac 初始化，初始化函数中调用：

```
hi_dmac_init();
```

步骤 2. 开发者根据设备硬件特性配置相关的管脚复用。

具体请参考如下：

- 《Hi3516A/Hi3516D 专业型 HD IP Camera Soc 用户指南》、《Hi3518EV20X/Hi3516CV200 经济型 HD IP Camera Soc 用户指南》中管脚复用控制寄存器章节；
- Hi3519V100/ Hi3519V101 请参考《Hi3519V100_PINOUT_CN》/《Hi3519V101_PINOUT_CN》中管脚复用寄存器页签。
- Hi3516CV300 请参考《Hi3516CV300_PINOUT_CN》中管脚复用寄存器页签。

步骤 3. 通过/dev/ uartdev-x 节点调用 open 打开指定 UART。

步骤 4. 打开 UART 后可调用 ioctl 配置。

步骤 5. 打开 UART 后可调用 read，write 读取数据，与发送数据，可采用 select 阻塞 read。

步骤 6. 不使用 UART 时，调用 close 关闭。关闭后 UART 控制器不会再接收串口接收线上的数据。

----结束

5.4 Shell 命令

包括 uart_read、uart_write、uart_config、uart_close 分别对 UART 读取、发送、配置和关闭操作。

uart_read

从 UART 接收缓存里读取指定长度数据。

- 建立线程接收数据，格式：uart_read <num> <len>
- 退出接收线程，停止读取数据，uart_read -q

说明

- num：UART 控制器号（对应 UART 控制器 0、1、2、3）。
- len：一次读取接收缓存的长度。

- 示例：uart_read 2 100
  从 UART 2 中读取 100 个字节数据长度。

uart_write

通过 UART 控制器发送指定一段数据（字符）。

- 格式：uart_write <num> <buf>

说明

- num：UART 控制器号。
- buf：发送数据缓存，ascii 模式。

- 示例：uart_wirte 2 abcdefghijklmnopqrstuvwxy
  通过 UART 2 发送一段字符串。

uart_config

配置 UART 波特率、阻塞读取、DMA 接收、数据位等。

- 格式：uart_config <num> <cmd> <arg>

说明

- num：UART 控制器号。
- cmd：命令号，0x101—配置通信波特率，具体配置可查看表 5-1 UART 配置说明。
- arg：命令号对应的参数，如配置波特率可为 2400、9600、115200 等。

- 示例：uart_config 2 0x101 9600
  配置 UART2 波特率为 9600

uart_close

关闭已打开的 UART 控制器，在调用 uart_open、uart_write、uart_cofing 时已对 UART 打开。

格式：uart_close <num>

说明

UART 在关闭后，不会再接收数据。下次再次 read 或 write 时会重新打开，并配置成上一次关闭前串口的工作状态。

5.5 API 参考

uart_dev_init

【描述】

UART 设备初始化。

【语法】

```
int uart_dev_init(void);
```

【参数】

参数名称	描述	输入/输出
无	无	无

【返回值】

返回值	描述
0	操作成功
其它	操作失败

uart_suspend

【描述】

UART 设备挂起。

【语法】

```
int uart_suspend(void *data);
```

【参数】

参数名称	描述	输入/输出
data	未使用（保留），传入 NULL 即可	无

【返回值】

返回值	描述
0	操作成功
其它	操作失败

uart_resume

【描述】

UART 设备唤醒。

【语法】

```
int uart_resume(void *data);
```

【参数】

参数名称	描述	输入/输出
data	未使用（保留），传入 NULL 即可	无

【返回值】

返回值	描述
0	操作成功
其它	操作失败

5.5.1 ioctl 配置说明

打开 UART 后，通过 ioctl 配置 UART 波特率，dma 接收，阻塞读取，线控等。如不配置，采用默认值配置。例如，配置波特率为：

```
ret = ioctl(fd, CFG_BAUDRATE, 9600);
```

说明

相关宏定义在 drivers/uart/include/uart.h 头文件中，配置说明请查看表 5-1 所示。

表5-1 UART 配置说明

命令号	命令码	参数	说明
UART_CFG_BAUDRATE	0x101	波特率	配置波特率，UART0 默认波特率为 115200；UART1、UART2、UART3 为 9600 支持最大波特率为 921600。
UART_CFG_DMA_RX	0x102	0、1	0：配置为中断接收方式； 1：配置为 DMA 接收方式默认为中断方式
UART_CFG_DMA_TX	0x103	0、1	暂未支持
UART_CFG_RD_BLOCK	0x104	0、1	0：配置为非阻塞方式 read； 1：配置为事件阻塞方式 read 默认为阻塞方式；
UART_CFG_ATTR	0x105	&uart_attr	配置校验位，数据位，停止位，FIFO，CTS/RTS 等 默认值为：无校验位，8 位数据位，1 位停止位，禁能 CTS/RTS。 参考头文件 struct uart_attr
UART_CFG_PRIVATE	0x110	自定义	驱动自定命令

5.5.2 举例

例程请参考文件：

```
./drive0072s/uart/src/uart_shell.c
```

此例程为 shell 命令 uart_open、uart_write、uart_cofing 的具体实现。仅供参考。

6 USB 2.0 Host/Device 操作指南

6.1 功能介绍

Hi35xx 支持的 USB 相关功能有：

- 支持挂载到系统的 SD/MMC 卡当做 U 盘(简称 USB2.0 DeviceDisk)供 USB Host 端（下文以 PC 端为例）进行读写访问等操作
- 支持对 U 盘的读写访问等操作

说明

- 目前 USB2.0 Device 不支持在 PC 端执行  弹出 MassStorageDevice 的安全退出功能
- 另外在多卡情况下(系统同时挂载 2 张卡以上)，USB2.0 Device 默认绑定第一张卡设备节点，例如同时存在 /dev/mmcblk0p0,/dev/mmcblk1p0，USB2.0 Device 会绑定/dev/mmcblk0p0，Host 端因此通过 USB 口只能访问到一张 SD/MMC 卡。
- USB2.0 Device 连接 PC 端后，PC 端可以通过 USB 端口访问到系统存储介质的文件。在 PC 端访问存储介质过程，系统端请勿对 MMC/SD Card 存储介质进行读写、格式化等操作，避免 PC 端访问存储介质异常。
- 当前 USB2.0 Host 仅支持 EHCI，尚不支持 OHCI、XHCI，由此出现全速 USB 设备不能识别的现象是正常的；另外 USB3.0 设备运行在 USB2.0 的现象也是正常的。

6.2 重要概念

6.2.1 USB 模块内存配置

USB 模块的正常运行需要一段 uncached 内存，需要单独配置给 USB 模块。

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= 0x40000; //recommend 256K 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。

6.2.2 USB2.0 Device 运行的必须动作

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 必须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。

应用层通过 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。

请在 USB 模块初始化后，调用 fmass_register_notify()实现 USB2.0 Device 回调函数的注册。

注意

Nofify 回调函数中，在 usb device 链接的情况下，需通过 fmass_partition_startup()将需要被 PC 端识别的 SD 卡分区节点启动。该函数的使用示例见 USB 模块使用示例。

6.2.3 操作中需要注意的问题

- 在正常操作过程中需要遵守的事项：

  PC 端进行读写访问过程，不要拔出单板的 SD/MMC 卡，否则会打印一些异常信息，并且可能会导致卡中文件或文件系统被破坏。

- 在操作过程中出现异常时的操作：

  拔出 USB2.0 DeviceDisk 盘后，再极其快速地插入 USB2.0 DeviceDisk 时可能会出现 PC 端检测不到 Udisk 的现象，因为单板 USB2.0 DeviceDisk 的检测注册/注销过程需要一定的时间。

6.3 模块编译

USB 驱动源码路径为 drivers/usb 。用户在 Huawei LiteOS 根目录下执行 make 即可编译出对应的 USB 模块库。编译成功后，会在 out/hi3516a/lib 目录下生成名为 libliteos_usb.a 的库文件。

6.4 USB 模块使用示例

6.4.1 USB 模块初始化示例

```
extern int fmass_register_notify(void(*notify)(void* context, int status),
void* context);
extern int fmass_partition_startup(char* path);
void fmass_app_notify(void* conext, int status)
{
    char *path = "/dev/mmcblk0p0";
    if(status == 1)/*usb device connect*/
    {
        /*停止录像抓拍等对/dev/mmcblk0p0分区的读写操作，并umount该分区*/
        //startup fmass access patition
        fmass_partition_startup(path);
}
else
{
    /* mount该分区,并通知其它模块该分区可以进行读写操作*/
}
}
void app_init()
{
    g_usb_mem_addr_start= g_sys_mem_addr_end;
    g_usb_mem_size= 0x40000; //recommend 256K nonCache for usb
    uwRet = usb_init();
    if(!uwRet)
          fmass_register_notify(fmass_app_notify,NULL);
    //usb_init must after SD_MMC_Host_init
    SD_MMC_Host_init();
}
```

6.4.2 USB2.0 Host 读写 U 盘

初始化完毕后，若 USB 插入 U 盘， USBHost 枚举 U 盘完成后，会在/dev 目录下生成/dev/sda 节点。对该设备节点挂载正确的文件系统后，即可对 U 盘进行读写操作。

```
#define WRITE_BUFFER_SIZE 40
```

```
#define READ_BUFFER_SIZE  40
INT32 Udisk_fs_ctrl(void)
{
    int fd = 0;
    int ret = 0;
    char w_buf[WRITE_BUFFER_SIZE] = "Hello,this is a test!
Monday,Tuesday";
    char r_buf[READ_BUFFER_SIZE];
    char * charp = NULL;
    unsigned int file_length = 0;
    int i = 0;

    memset(r_buf, 0, READ_BUFFER_SIZE);

    dprintf("1.open file:\n");
    fd = open("/sd0/testfile",O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if (-1 == fd) {
        dprintf("open/create Testfile fail ! \n");
        return -1;
    }
    dprintf("open file success ! fd = %d \n\n",fd);

    dprintf("2.write file:");
    ret = write(fd, w_buf, WRITE_BUFFER_SIZE);
    if (-1 == ret ) {
        dprintf("file write fail! \n");
        close(fd);
        return -1;
    }
    dprintf("write file success!\n\n");

    ret = lseek(fd, 0, SEEK_SET);
    if (-1 == ret) {
        dprintf("lseek fail!\n");
        close(fd);
        return -1;
    }
    dprintf("lseek file success!\n\n");

    dprintf("4.read file:");
    if (file_length > READ_BUFFER_SIZE) {
        file_length = READ_BUFFER_SIZE;
    }
    ret = read(fd, r_buf, file_length);
```

```
    if (-1 ==  ret) {
        dprintf("file read fail! \n");
        close(fd);
        return -1;
    }
    charp = r_buf;
    for ( i = 0; i < ret; i++) {
        dprintf("%c",*charp);
        charp++;
    }
    dprintf("\n");
    charp = NULL;
    close(fd);
    return 0;
}
void app_init()
{
    g_usb_mem_addr_start= g_sys_mem_addr_end;
     g_usb_mem_size= 0x40000; //recommend 256K nonCache for usb
    uwRet = usb_init();

    /*等待U盘插入，驱动注册/dev/sdap0节点*/
    mount("/dev/sdap0", "/sd0", "vfat", 0, 0);
    Udisk_fs_ctrl();
}
```

6.5 API 参考

该功能模块提供以下接口：

- usb_init：初始化 USB 模块。
- fmass_register_notify：注册 USB2.0 Device notify 回调函数。
- fmass_partition_startup：将需要被 PC 端识别的 SD 卡分区节点启动

usb_init

【描述】

初始化 USB 模块。

【语法】

```
UINT32 usb_init(void)
```

【参数】

无

【返回值】

返回值	描述
0	成功
-1	失败

【需求】

- 头文件：无
- 库文件：libliteos_usb.a

【注意】

支持 USB2.0 Device 功能时，则 usb_init 必须在 SD_MMC_Host_init 之后调用。

【举例】

参考 USB 模块使用示例。

【相关主题】

fmass_register_notify

fmass_register_notify

【描述】

注册 USB2.0 Device notify 回调函数。

【语法】

```
int fmass_register_notify(void(*notify)(void* context, int status), void* context)
```

【参数】

参数名称	描述	输入/输出
notify	Notify 回调函数指针	输入
context	需要传入 Notify 回调函数的私有数据指针，该指针会传入给 Notify 回调函数的 context 参数	输出

【返回值】

返回值	描述
0 ～ MAX_NOFIFY_NUM-1	成功

返回值	描述
-1	失败

【需求】

- 头文件：无
- 库文件：libliteos_usb.a

【注意】

允许注册多个 notify 回调函数，USB2.0 Device 拔插事件发生时驱动会调用每一个 notify 函数。

【举例】

参考 USB 模块使用示例。

【相关主题】

fmass_partition_startup

fmass_partition_startup

【描述】

将需要被 PC 端识别的 SD 卡分区节点启动。

【语法】

```
int fmass_partition_startup(char* path)
```

【参数】

参数名称	描述	输入/输出
path	SD 卡分区节点。	输入

【返回值】

返回值	描述
0	成功
-1	失败

【需求】

- 头文件：None
- 库文件：libliteos_usb.a

【注意】

无

【举例】

参考 USB 模块使用示例。

【相关主题】

fmass_register_notify

7 Flash 操作指南

7.1 功能介绍

芯片集成 flash 控制器，支持对 NAND flash、SPINOR flash 的擦除、读、写等操作。

7.2 模块编译

源码路径为 drivers/mtd，在编译脚本里指定源码路径与头文件路径，编译成功后，out/hi35xx/lib 目录下会生成名为 libnand_flash.a 和 libspinor_flash.a 的库文件，链接时通过-lnand_flash 和-lspinor_flash 指定对应库文件。

说明

- 文档中的路径指的是 Huawei LiteOS 源代码根目录下的相对路径。
- 针对 Hi3516A/Hi3516D 及 Hi3519V100/Hi3519V101，默认使用并口 NAND FLASH，如果使用 SPI NAND FLASH，需要在主目录下执行 make FLASH_TYPE=spinand 命令。

7.3 重要概念

7.3.1 NAND flash

按照接口类型可以分为：并口 NAND flash 和 SPI NAND flash；

- Hi3516A/Hi3516D 支持并口 NAND flash 和 SPI NAND flash，但两者不能同时使用；
- Hi3518EV200/Hi3518EV201/Hi3516CV200/Hi3516CV300 仅支持 SPI NAND flash；
- Hi3519V100/Hi3519V101 支持并口 NAND flash 和 SPI NAND flash，但两者不能同时使用；

擦除：在对 NAND flash 写之前必须先对其擦除，擦除的单位为一个块(block)；

读：读的单位为一个页(page)；

写：或称为编程，其操作单位为一个页(page)；

7.3.2 SPI NOR flash

擦除：在对 SPI NOR flash 写之前必须先对其擦除，擦除的单位为一个块(block)；

读：读的单位为一个页(page)；

写：或称为编程，其操作单位为一个页(page)；

7.4 使用示例

7.4.1 模块初始化

在对 flash 操作之前需要根据 flash 类型调用相应的初始化函数：

```
NAND flash:
   nand_init();
SPINOR flash:
   spinor_init();
```

7.4.2 通过文件系统访问 flash

步骤 1. 创建分区：

```
NAND flash:
   add_mtd_partition("nand", 分区起始地址, 分区大小, 分区号);
SPINOR flash:
   add_mtd_partition("spinor",分区起始地址, 分区大小, 分区号);
```

说明

分区起始地址和分区大小必须块对齐。

步骤 2. mount 分区：

```
NAND flash:
   mount("/dev/nandblk0", 挂载点名称, "yaffs", 0, NULL);
SPINOR flash:
   mount("/dev/spinorblk0", 挂载点名称, "jffs", 0, NULL)
```

步骤 3. 通过 VFS 接口操作对应的挂载点：

参考 OS VFS 接口描述。

----结束

7.4.3 通过驱动提供的接口操作 flash

擦除：

```
NAND flash:
```

```
int hinand_erase(loff_t start, size_t size)
```

SPINOR flash:

```
int hispinor_erase(loff_t start, size_t size)
```

说明

对 NAND 和 SPI NOR 擦除，起始地址和大小必须块对齐。

读：

NAND flash:

```
int hinand_read(void* memaddr, loff_t start, size_t size)
```

SPINOR flash:

```
int hispinor_read(void* memaddr, loff_t start, size_t size)
```

写：

NAND flash:

```
int hinand_write(void* memaddr, loff_t to, size_t size)
```

说明

对 NAND 写，起始地址和大小必须页对齐。

SPINOR flash:

```
int hispinor_write (void* memaddr, loff_t to, size_t size)
```

7.5 shell 命令

7.5.1 nd_bad 命令

命令格式：nd_bad

功能：查看 NAND flash 坏块信息。

7.6 API 参考

7.6.1 擦除接口

【描述】

擦除 flash 设备。

【语法】

NAND flash:

```
int hinand_erase(loff_t start, size_t size)
```

SPINOR flash:

```
int hispinor_erase(loff_t start, size_t size)
```

【参数】

参数名称	描述	输入/输出
start	擦除起始地址； 取值：块对齐。	输入
size	擦除大小； 取值：块对齐。	输入

【返回值】

返回值	描述
0	擦除成功
负值	擦除失败

7.6.2 读接口

【描述】

读 flash 设备

【语法】

```
NAND flash:
    int hinand_read(void* memaddr, loff_t start, size_t size);
SPI NOR flash:
    int hispinor_read(void* memaddr, loff_t start, size_t size);
```

【参数】

参数名称	描述	输入/输出
memaddr	读数据存放的内存起始地址	输入
start	flash 起始地址	输入
size	读长度	输入

【返回值】

返回值	描述
0	读成功

返回值	描述
负值	读失败

7.6.3 写接口

【描述】

写 flash 设备

【语法】

```
NAND flash:
    int hinand_write(void* memaddr, loff_t to, size_t size);
SPINOR flash:
    int hispinor_write(void* memaddr, loff_t to, size_t size);
```

【参数】

参数名称	描述	输入/输出
memaddr	写数据存放的内存起始地址	输入
to	flash 起始地址：对于 NAND flash 要页对齐	输入
size	写长度：对于 NAND flash 要页对齐	输入

【返回值】

返回值	描述
0	写成功
负值	写失败

8 GPIO 操作指南

8.1 功能介绍

GPIO 可配置为输入或者输出，可用于生成特定应用的输出信号或采集特定应用的输入信号。

8.2 模块编译

源码路径为 drivers/gpio，在编译脚本里指定源码路径与头文件路径，编译成功后，out/hi35xx/lib 目录下会生成名为 libgpio.a 的库文件，链接时通过-lgpio 指定对应库文件。

说明
文档中的路径指的是 Huawei LiteOS 源代码根目录下的相对路径。

8.3 使用示例

8.3.1 模块初始化

在对 GPIO 操作之前需要调用初始化函数：

```
gpio_dev_init();
```

8.3.2 通过文件系统访问 GPIO

步骤 1. 打开 GPIO 总线对应的设备文件，获取文件描述符：

```
fd = open("/dev/gpio", O_RDWR);
```

说明
如未完成设备文件的注册工作，可调用 gpio_dev_init 函数注册设备文件。

步骤 2. 定义 GPIO 状态结构体，并初始化：

```
gpio_groupbit_info group_bit_info;
```

```
group_bit_info.groupnumber = 1;
group_bit_info.bitnumber =1;
使用ioctl获得GPIO信息
ioctl(fd, GPIO_GET_DIR, &group_bit_info);
```

注意

以上示例的作用是获得 GPIO 的输入输出状态。更多操作的宏定义在 drivers/gpio/include/hi_gpio.h 头文件。

----结束

8.4 API 参考

【需求】

- 头文件：gpio.h
- 库文件：libgpio.a

该功能模块提供以下接口：

- gpio_chip_init：GPIO 初始化接口
- gpio_chip_deinit：GPIO 去初始化接口
- gpio_get_direction：获取 GPIO 方向
- gpio_direction_input：设置 GPIO 方向为输入
- gpio_direction_output：设置 GPIO 方向为输出
- gpio_get_value：获取 GPIO 值
- gpio_set_value：设置 GPIO 值
- gpio_irq_register：注册 GPIO 中断
- gpio_set_irq_type：设置 GPIO 中断类型
- gpio_irq_enable：使能 GPIO 中断
- gpio_get_irq_status：获取中断状态
- gpio_clear_irq：清除 GPIO 寄存器中断状态

gpio_chip_init

【描述】

GPIO 初始化接口。

【语法】

```
int gpio_chip_init(struct gpio_descriptor *gd);
```

【参数】

参数名称	描述	输入/输出
gd	全局变量，定义于 drivers/gpio/src/gpio_dev.c 中	输入

【注意】

开发者可参考 drivers/gpio/src/gpio_dev.c 中对该接口的调用。

gpio_chip_deinit

【描述】

GPIO 去初始化接口。

【语法】

```
int gpio_chip_deinit(struct gpio_descriptor *gd);
```

【参数】

参数名称	描述	输入/输出
gd	全局变量，定义于 drivers/gpio/src/gpio_dev.c 中	输入

【注意】

开发者可参考 drivers/gpio/src/gpio_dev.c 中对该接口的调用。

gpio_get_direction

【描述】

获取 GPIO 方向。

【语法】

```
int gpio_get_direction(gpio_groupbit_info * gpio_info);;
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 与 bitnumber 成员 获取的方向值将保存在 direction 成员中。	输入

gpio_direction_input

【描述】

设置 GPIO 方向为输入。

【语法】

```
int gpio_direction_input(gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 与 bitnumber 成员	输入

gpio_direction_output

【描述】

设置 GPIO 方向为输出。

【语法】

```
int gpio_direction_output(gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 与 bitnumber 成员	输入

gpio_get_value

【描述】

获取 GPIO 值。

【语法】

```
int gpio_get_value (gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 与 bitnumber 成员 获取的值将保存在 value 成员里	输入

gpio_set_value

【描述】

设置 GPIO 值。

【语法】

```
int gpio_set_value (gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 和 bitnumber 成员 并将设置的值保存在 value 成员里	输入

gpio_irq_register

【描述】

注册 GPIO 中断。

【语法】

```
int gpio_irq_register (gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 和 bitnumber 成员。 将设置的中断类型保存在 irq_type 成员里。 将设置的中断回调函数地址保存在 irq_handler 成员里。 如有私有数据需要传递到回调函数，将数据地址保存在 data 成员里，如无则不需要初始化。	输入

gpio_set_irq_type

【描述】

设置 GPIO 中断类型。

【语法】

```
int gpio_set_irq_type(gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 和 bitnumber 成员 并将中断类型保存在 irq_type 成员里。相关值可参考 drivers/gpio/include/gpio.h 文件中 gpio_groupbit_info 结构体中定义的宏。	输入

gpio_irq_enable

【描述】

使能 GPIO 中断。

【语法】

```
int gpio_irq_enable(gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 和 bitnumber 成员 打开中断，则初始化 irq_enableGPIO_IRQ_ENABLE 否则为 GPIO_IRQ_DISABLE	输入

gpio_get_irq_status

【描述】

获取 GPIO 中断状态。

【语法】

```
int gpio_get_irq_status (gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 和 bitnumber 成员 获取的中断状态将保存在 irq_status 成员里	输入

gpio_clear_irq

【描述】

清除 GPIO 中断寄存器状态。

【语法】

```
int gpio_clear_irq (gpio_groupbit_info * gpio_info);
```

【参数】

参数名称	描述	输入/输出
gpio_info	操作的 GPIO 信息。 作输入时必须初始化 groupnumber 和 bitnumber 成员	输入