外围设备驱动

操作指南

文档版本　04

发布日期　2016-11-18

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 GMAC、USB 2.0 Host/Device 和 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。

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

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
2016-11-18	04	第 4 次正式版本发布。 4.4.2 和 4.4.3 小节涉及修改

修订日期	版本	修订说明
2015-06-04	03	新增 1.4 和 1.5 小节
2015-02-10	02	5.3.2 和 5.3.3 小节的注意有添加相关内容，新增了 4.3 小节
2014-12-20	01	添加 Hi3516D 的相关内容
2014-11-10	00B04	新增第 5 章 “SPI 操作指南”
2014-10-19	00B03	第 3 次版本发布 2.2 和 2.3.4 章节涉及修改，新增第 4 章“I2C 操作指南”
2014-09-14	00B02	第 2 次版本发布
2014-07-15	00B01	第 1 次版本发布

目 录

插图目录

1 GMAC 操作指南

说明

- 本文未做特殊说明，Hi3516D 与 Hi3516A 完全一致
- 以下设置的地址只是一个举例说明，具体的地址设置要根据具体使用的地址来设置。

1.1 操作示例

说明

Hi3516A 默认相关 GMAC 模块已全部编入内核，不需要执行加载操作，请直接跳至配置 IP 地址步骤。

内核下使用网口的操作涉及到以下几个方面：

- GMAC 模块支持 TSO 功能且默认是打开的，如果用户希望关闭 TSO 功能，可通过工具 ethtool 将其关闭。开关 TSO 功能的方法如下：
  - 关闭 TSO：./ethtool –K eth0 tx off
  - 打开 TSO：./ethtool –K eth0 tx on

  TSO（TCP Segment Offload）功能简介：
  - TSO (TCP Segmentation Offload)是一种利用网卡分割大数据包，减小 CPU 负荷的一种技术，也被叫做 LSO (Large segment offload)，如果数据包的类型只能是 TCP，则被称之为 TSO，如果硬件支持 TSO 功能的话，也需要同时支持硬件的 TCP 校验计算和分散-聚集 (Scatter Gather) 功能。TSO 的实现，其实是由软件和硬件结合起来完成的，具体说来，硬件能够对大的数据包进行分片，并对每个分片附着相关的头部。
  - Hi3516A 芯片使用 TSO 时，会把一部分由 CPU 处理的工作转移到由网卡来处理，减轻 CPU 的压力，提高性能。
- 配置 ip 地址和子网掩码

  ```
  ifconfig eth0 xxx.xxx.xxx.xxx netmask xxx.xxx.xxx.xxx up
  ```
- 设置缺省网关

  ```
  route add default gw xxx.xxx.xxx.xxx
  ```
- mount nfs

```
mount -t nfs -o nolock xxx.xxx.xxx.xxx:/your/path /mount-dir
```

- shell 下使用 tftp 上传下载文件

  前提是在 server 端有 tftp 服务软件在运行。

  - 下载文件：tftp -r XX.file serverip -g

    其中：XX.file 为需要下载的文件，serverip 需要下载的文件所在的 server 的 ip 地址。

  - 上传文件：tftp -l xx.file remoteip –p

    其中，xx.file 为需要上传的文件，remoteip 文件需要上传到的 server 的 ip 地址。

1.2 IPv6 说明

发布包中默认关闭 IPv6 功能。如果要支持 IPv6，需要修改内核选项，并重新编译内核。具体操作如下：

```
hisilicon$cd kernel/linux-3.4.y
hisilicon$cp arch/arm/configs/hi3516a_full_defconfig .config
hisilicon$make ARCH=arm CROSS_COMPILE=arm-hisivXXX-linux- menuconfig
```

说明

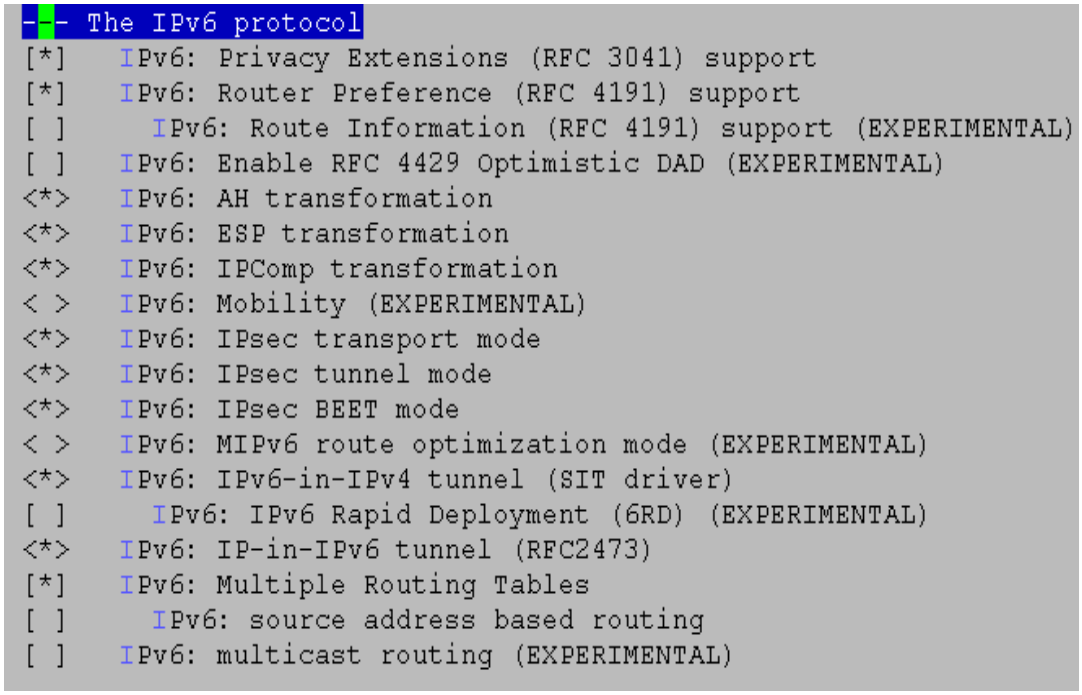
CROSS_COMPILE=arm-hisiXXX-linux-中 XXX 表示两种情况。

- Hi3516A_V100R001C01SPCxxx 对应 uclibc，使用 uclibc 工具链时，CROSS_COMPILE=arm-hisiv300-linux-。
- Hi3516A_V100R001C02SPCxxx 对应 glibc，使用 glibc 工具链时，CROSS_COMPILE=arm-hisiv400-linux-。

进入如下目录，将该页面选项配置如图 1-1 所示。

```
[*] Networking support  --->
    Networking options  --->
        <*>   The IPv6 protocol  --->
```

图1-1 IPv6 Protocol 配置示意图



IPv6 环境配置如下：

- 配置 ip 地址及缺省网关

  ```
  hisilicon$ ip -6  addr add <ipv6address>/<ipv6_prefixlen> dev <port>
  ```

  示例：`ip -6 addr add 2001:da8:207::9402/64 dev eth0`

- Ping 某个- IPv6 地址

  ```
  hisilicon$ ping -6  <ipv6address>
  ```

  示例：`ping -6  2001:da8:207::9403`

1.3 PHY 地址配置

Hi3516A DMEB 板上 PHY 地址默认为 1，当选用不同的 PHY 地址时须在 U-boot 和 Kernel 下更改 PHY 地址配置。

a. U-boot 下配置方式

   U-boot 下可通过更改 U-boot 配置文件中宏定义 CONFIG_HIGMAC_PHY1_ADDR 的值来配置不同的 PHY 地址。Hi3516A 的 U-boot 包含以下两个配置文件，两者须同时更改。

   - include/configs/hi3516a.h
   - include/configs/hi3516a_spinand.h

b. Kernel 下配置方式

   在 Kernel 下可通过以下内核配置选项配置 PHY 地址。

   ```
   Device Drivers
   ```

```
[*] Network device support  --->
[*] Ethernet driver support  --->
<*> hieth gmac family network device support  --->
(1) hieth-gmac phy0 addr
```

1.4 PHY 接口模式配置

Hi3516A 的 GMAC 模块支持 PHY 接口模式有 rgmii、rmii 和 mii，发布包中默认配置为 rgmii，若需配置成 rmii 或 mii，需要在 boot 和 kernel 下修改配置

a. U-boot 下通过环境变量设置

   setenv mdio_intf mii 或者 setenv mdio_intf rmii

b. Kernel 下通过 menuconfig 配置

   Device Drivers

   [*] Network device support --->

   [*] Ethernet driver support --->

   <*> hieth gmac family network device support --->

   (6) hieth-gmac phy0 interface mode

其中，1 代表 mii，5 代表 rmii，6 代表 rgmii，若管脚复用有变化，请重新配置管脚复用关系。

1.5 IEEE 802.3x 流控功能配置

1.5.1 流控功能描述

GMAC 网络支持 IEEE 802.3x 定义的流控功能，能够发送流控帧和接收处理对端的流控帧。

a. 流控帧发送功能：

   在接收方向，若当前接收描述子队列出现紧张，可能无法满足已收到的数据包全部送达软件，则会发送流控帧至对端，告知对端暂停一定时间不发包；

b. 流控帧的接收功能：

   当接收到流控帧，GMAC 会根据帧内的流控时间字段进行延迟发送，等待计时到达流控时间后，则会再次启动发送，或在等待过程中收到了对端发送的流控时间为 0 的流控帧，同样会再次启动发送；

1.5.2 内核 menuconfig 配置接口

发送和接收流控帧功能提供 6 个参数供用户配置，用户通过编译内核时配置 menuconfig 进行配置。网络驱动的 menuconfig 如下：

```
--- hieth gmac family network device support
(0x726d6d73) hieth-gmac misc tag
(0x726d6d73) hieth-gmac mac address tag
(0x10090000) hieth-gmac IO address
(57)  hieth-gmac irq number
(1)   hieth-gmac phy0 addr
(6)   hieth-gmac phy0 interface mode
[*]   higmac reset helper
(0x20140000) higmac reset helper on which gpio group
(1)     higmac reset helper on gpio bit
(0)     higmac reset helper on gpio value
[*]   rx flow ctrl supported
[*]   tx flow ctrl supported
(0xFFFF) tx flow ctrl pause time
(0x002f) tx flow ctrl pause interval
(16)  tx flow ctrl active threshold
(32)  tx flow ctrl deactive threshold
```

用户可配置的流控参数如下：

a. rx flow ctrl supported

接收流控帧功能是否使能；

b. tx flow ctrl supported：

发送流控帧功能是否使能；

c. tx flow ctrl pause time

发送的 pause 帧中携带的 pause time 值；单位是传输 512bit 所需要的时间，最大为 0xFFFF，默认值为 0xFFFF；

d. tx flow ctrl pause interval

当满足发送流控帧条件时，重复发送流控帧的时间间隔；单位是传输 512bit 所需要的时间，最大为 0xFFFF，默认值为 0x002F；该值必须小于 tx flow ctrl pause time；

e. tx flow ctrl active threshold

发送流控激活水线，当接收队列可用描述子个数小于该值，会启动逻辑发送流控帧的流程；

f. tx flow ctrl deactive threshold

发送流控撤销水线，当接收队列可用描述子个数大于或者等于该值同时正处于流控状态时，解除当前流控状态。

说明

流控功能激活水线值必须小于撤销水线。

1.5.3 ethtool 配置接口

用户可以通过标准 ethtool 工具接口进行流控功能的使能。

ethtool –a eth0 命令查看 eth0 口流控功能状态；打印如下：

```
# ./ethtool -a eth0
Pause parameters for eth0:
Autonegotiate:    on
RX:               on
TX:               on
```

其中，RX 流控是打开的，TX 流控是打开的；

用户可以通过以下命令打开或关闭流控：

./ethtool -A eth0 rx off（关闭 RX 流控）

./ethtool -A eth0 rx on（打开 RX 流控）

./ethtool -A eth0 tx off（关闭 TX 流控）

./ethtool -A eth0 tx on（打开 TX 流控）

2 USB 2.0 Host/ Device 操作指南

说明

Hi3516A 的 USB 2.0 支持 Host 和 Device 两种模式。

2.1 操作准备

USB 2.0 Host/ Device 的操作准备如下：

- U-boot 和 Linux 内核使用 SDK 发布的 U-boot 和 kernel
- 文件系统

  可以使用本地文件系统 yaffs2、jffs2 或 cramfs，也可以使用 NFS，建议使用 jffs2。

2.2 操作过程

操作过程如下：

步骤 1. 启动单板，加载 yaffs2、jffs2 或 cramfs 文件系统，也可以使用 NFS。

步骤 2. 默认 USB 相关模块已经全部编入内核，不需要再执行加载命令，就可以对 U 盘、鼠标或者键盘进行相关的操作了。具体操作请参见“2.3 操作示例”。下面列出所有 USB 相关驱动：

- 文件系统和存储设备相关模块
  - vfat
  - scsi_mod
  - sd_mod
  - nls_ascii
  - nls_iso8859-1
- 键盘相关模块
  - evdev
  - usbhid

- 鼠标相关模块
  - mousedev
  - usbhid
  - evdev
- USB2.0 模块
  - ohci-hcd
  - ehci-hcd
  - usb-storage
  - hiusb-hi3516A

----结束

USB 2.0 Device 操作过程

步骤 1. 启动单板，加载 yaffs2、jffs2 或 cramfs 文件系统，也可以使用 NFS。

步骤 2. 单板作为 Device 时，须加载模块 dwc_otg 和 g_file_storage 才能在 Host 端被识别成 USB 存储介质。dwc_otg 和 g_file_storage 模块默认不编入内核，需要另行加载。具体操作请参见“2.3 操作示例”。

下面列出所有 USB Device 相关驱动：

- 文件系统和存储设备相关模块与 Host 相同
- USB Device 模块
  - hi_hs_device
  - g_file_storage

----结束

2.3 操作示例

2.3.1 U 盘操作示例

插入检测

直接插入 U 盘，观察是否枚举成功。

USB 2.0 Host 正常情况下串口打印为：

```
~ # usb 1-1: new high-speed USB device number 7 using hiusb-ehci
scsi2 : usb-storage 1-1:1.0
scsi 2:0:0:0: Direct-Access     Kingston DT 101 G2       1.00 PQ: 0 ANSI:
4
sd 2:0:0:0: [sda] 15131636 512-byte logical blocks: (7.74 GB/7.21 GiB)
sd 2:0:0:0: [sda] Write Protect is off
sd 2:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't
support DPO or FUA
```

```
 sda: sda1
sd 2:0:0:0: [sda] Attached SCSI removable disk
```

其中：sda1 表示 U 盘或移动硬盘上的第一个分区，当存在多个分区时，会出现 sda1、sda2、sda3 等字样。

初始化及应用

模块插入完成后，进行如下操作：

说明

sdXY 中 X 代表磁盘号，Y 代表分区号，请根据具体系统环境进行修改。

- 分区命令操作的具体设备节点为 sdX，示例：$ fdisk /dev/sda
- 用 mkdosfs 工具格式化的具体分区为 sdXY：~ $ mkdosfs –F 32 /dev/sda1
- 挂载的具体分区为 sdXY：~ $ mount -t vfat /dev/sda1 /mnt

步骤 1. 查看分区信息。

- 运行命令“ls /dev”查看系统设备文件，若没有分区信息 sdXY，表示还没有分区，请参见“6.1 用 fdisk 工具分区”进行分区后，进入步骤 2。
- 若有分区信息 sdXY，则已经检测到 U 盘，并已经进行分区，进入步骤 2。

步骤 2. 查看格式化信息。

- 若没有格式化，请参见“6.2 用 mkdosfs 工具格式化”进行格式化后，进入步骤 3。
- 若已格式化，进入步骤 3。

步骤 3. 挂载目录，请参见“6.3 挂载目录”。

步骤 4. 对硬盘进行读写操作，请参见“6.4 读写文件”。

----结束

2.3.2 键盘操作示例

键盘操作过程如下：

步骤 1. 插入模块。

插入键盘相关模块后，键盘会在/dev/目录下生成 event0 节点。

步骤 2. 接收键盘输入。

执行命令：cat /dev/ event0

然后在 USB 键盘上敲击，可以看到屏幕有输出。

----结束

2.3.3 鼠标操作示例

鼠标操作过程如下：

步骤 1. 插入模块。

插入鼠标相关模块后，鼠标会在/dev/目录下生成 mouse0 节点。

步骤 2. 运行 gpm 中提供的标准测试程序（建议使用 mev）。

步骤 3. 进行鼠标操作（点击、滑动等），可以看到串口打印出相应码值。

----结束

2.3.4 USB Device 操作示例

单板作为 Device 时支持 Flash 和 SD 卡两种存储介质，操作过程如下：

步骤 1. 插入模块。

- 将 Flash 虚拟为 Device 存储介质，操作为：
  insmod dwc_otg.ko
  insmod g_file_storage.ko file=/dev/mtdblockX luns=1 stall=0 removable=1
  其中，mtdblockX 为 Flash 的第 X 个分区，请用户根据具体情况选择。
- 将 SD 卡虚拟为 Device 存储介质，操作为：
  insmod dwc_otg.ko
  insmod g_file_storage.ko file=/dev/mmcblk0pX luns=1 stall=0 removable=1
  其中，mmcblk0pX 为 SD 卡的第 X 个分区，请用户根据具体情况选择。

说明

dwc_otg.ko 和 g_file_storage.ko kernel 下路径分别为：

drivers/usb/gadget/hi_hs_dev/dwc_otg.ko

drivers/usb/gadget/g_file_storage.ko

步骤 2. 通过 USB 将单板与 Host 端相连，即可在 Host 端将单板识别成 USB 存储设备，并在/dev 目录下生成相应的设备节点。

步骤 3. 在 Host 端可将单板当成一个普通的 USB 存储设备，对其进行分区、格式化、读写等相关操作，详见“2.3.1 U 盘操作示例”。

----结束

2.4 操作中需要注意的问题

操作中需要注意的问题如下：

- 在操作时请尽量按照完整的操作顺序进行操作（mount→操作文件→umount），以免造成文件系统的异常。

- 目前键盘和鼠标的驱动要和上层结合使用，比如鼠标事件要和上层的 GUI 结合。对键盘的操作只需要对/dev 下的 event 节点读取即可，而鼠标则需要标准的库支持。
- 在 Linux 系统中提供了一套标准的鼠标应用接口 libgpm，如果需要使用鼠标客户可自行编译此库。在使用时建议使用内核标准接口 gpm。

  已测试通过的标准接口版本：gpm-1.20.5。

  另外在 gpm 中还提供了一整套的测试工具源码（如：mev 等），用户可根据这些测试程序进行编码等操作，降低开发难度。

  USB Device 单板要插入模块后，再与 Host 端相连，否则 Host 端将不识别 Device 设备，并循环打印错误信息。

3 SD/MMC 卡操作指南

3.1 操作准备

SD/MMC 卡的操作准备如下：

- U-boot 和 Linux 内核使用 SDK 发布的 U-boot 和 kernel。
- 文件系统。

可以使用 SDK 发布的本地文件系统 yaffs2、jffs2 或 SquashFS，也可以通过本地文件系统再挂载到 NFS。

3.2 操作过程

操作过程如下：

步骤 1. 启动单板，加载本地文件系统 yaffs2、jffs2 或 SquashFS，也可以通过本地文件系统进一步挂载到 NFS。

步骤 2. 加载内核。默认 SD/MMC 相关模块已全部编入内核，不需要再执行加载命令。下面列出 SD/MMC 所有相关驱动：

- 文件系统和存储设备相关模块
  - nls_base
  - nls_cp437
  - fat
  - vfat
  - msdos
  - nls_iso8859-1
  - nls_ascii
- SD/MMC 相关模块
  - mmc_core
  - himci
  - mmc_block

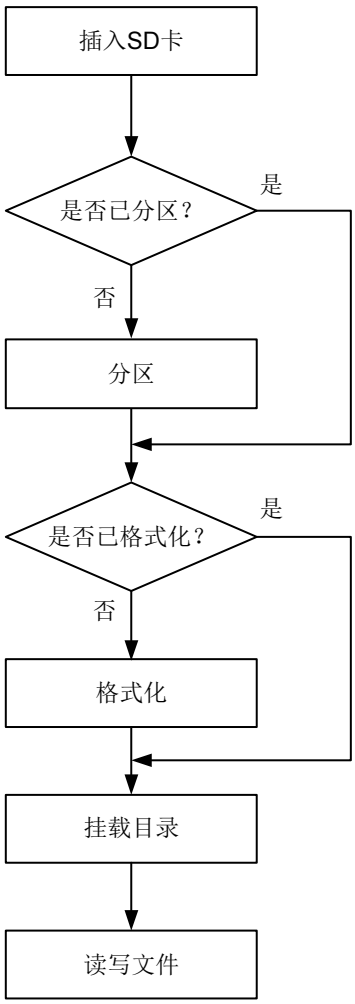
步骤 3. 插入 SD/MMC 卡，就可以对 SD/MMC 卡进行相关的操作。具体操作请参见“3.3 操作示例”。

----结束

3.3 操作示例

此操作示例通过 SDIO 接口实现对 SD 卡的读写操作，MMC 卡的读写操作和 SD 卡类似，这里不再举例。在控制台下实现读写 SD 卡的操作示例如图 3-1 所示。

图3-1 在控制台下实现读写 SD 卡的操作示例



初始化及应用，待 SD/MMC 卡插入后，进行如下操作：

说明

其中 X 为分区号，由 fdisk 工具分区时决定。

- 命令 fdisk 操作的具体目录需改为：~ $ fdisk /dev/mmcblk0
- 用 mkdosfs 工具格式化的具体目录需改为：~ $ mkdosfs –F 32 /dev/mmcblk0pX

- 挂载的具体目录需改为：~ $ mount -t vfat /dev/mmcblk0pX /mnt

步骤 1. 查看分区信息。

- 若没有显示出 p1，表示还没有分区，请参见“6.1 用 fdisk 工具分区”进行分区后，进入步骤 2。
- 若有分区信息 p1，则 SD/MMC 卡已经检测到，并已经进行分区，进入步骤 2。

步骤 2. 查看格式化信息。

- 若没有格式化，请参见“6.2 用 mkdosfs 工具格式化”进行格式化后，进入步骤 3。
- 若已格式化，进入步骤 3。

步骤 3. 挂载目录，请参见“6.3 挂载目录”。

步骤 4. 对 SD/MMC 卡进行读写操作，请参见“6.4 读写文件”。

----结束

3.4 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：

- 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。
- 每次需要读写 SD 卡时，必须确保 SD 卡已经创建分区，并将该分区格式化为 vfat 文件系统（通过 fdisk 和 mkdosfs 命令，具体过程参见“3.3 操作示例”）。
- 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；如果 SD 卡已经挂载到文件系统，拔卡后，必须做一次 umount 操作，否则，再次插入卡时就会找不到 SD 卡的分区。
- 正常拔卡后需要 umount 挂载点（建议正常的操作顺序是先 umount，再拔卡），异常拔卡后，也需要 umount 挂载点，否则再次插卡时就会找不到 SD 卡的分区。

在正常操作过程中不能进行的操作：

- 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。
- 当前目录是挂载目录如/mnt 时，不能 umount 操作，必须转到其它目录下才能 umount 操作。
- 系统中读写挂载目录的进程没有完全退出时，不能 umount 操作，必须完全结束操作挂载目录的任务才能正常 umount 操作。

在操作过程中出现异常时的操作：

- 如果在循环测试过程中异常拔卡，需要按 ctrl+c 回退出到 shell 下，否则会一直不停地打印异常操作信息。
- 拔卡后，再极其快速地插入卡时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。

- 异常拔卡后，必须执行 umount 操作，否则不能读写挂载点目录如/mnt，并会打印异常信息。
- SD 有多分区时，可以通过 mount 操作切换挂载不同的分区，但最后 umount 操作次数与 mount 操作次数相等时，才会完全 umount 所有的挂载分区。
- 如果由于读写数据或其它异常原因，导致文件系统破坏，重新插卡并挂载，读写卡时可能会出现文件系统 panic，这时，需要 umount 操作，拔卡，再次插卡并 mount，才能正常读写 SD 卡。

4 I2C 操作指南

4.1 操作准备

I2C 的操作准备如下：

- Linux 内核使用 SDK 发布的 kernel。
- 文件系统。

可以使用 SDK 发布的本地文件系统 yaffs2、jffs2 或 SquashFS，也可以通过本地文件系统再挂载到 NFS。

4.2 操作过程

操作过程如下：

步骤 1. 启动单板，加载本地文件系统 yaffs2、jffs2 或 SquashFS，也可以通过本地文件系统进一步挂载到 NFS。

步骤 2. 加载内核。默认 I2C 相关模块已全部编入内核，不需要再执行加载命令。

步骤 3. 在控制台下运行 I2C 读写命令或者自行在内核态或者用户态编写 I2C 读写程序，就可以对挂载在 I2C 控制器上的外围设备进行读写操作。具体操作请参见“4.4 操作示例”。

----结束

4.3 接口速率设置说明

发布包中默认接口速率是 100K。如果要更改接口速率，需要修改内核选项，并重新编译内核。具体操作如下：

```
hisilicon$cd kernel/linux-3.4.y
hisilicon$cp arch/arm/configs/hi3516a_full_defconfig .config
hisilicon$make ARCH=arm CROSS_COMPILE=arm-hisivXXX-linux- menuconfig
```

说明

CROSS_COMPILE=arm-hisiXXX-linux-中 XXX 表示两种情况。

- Hi3516A_V100R001C01SPCxxx 对应 uclibc，使用 uclibc 工具链时，CROSS_COMPILE=arm-hisiv300-linux-。
- Hi3516A_V100R001C02SPCxxx 对应 glibc，使用 glibc 工具链时，CROSS_COMPILE=arm-hisiv400-linux-。

进入如下目录，将该页面选项配置如图 4-1 所示。

```
Device Drivers  --->
    <*> I2C support  --->
        I2C Hardware Bus support  --->
                (100000) hi i2c0 clock limit
                (100000) hi i2c1 clock limit
                (100000) hi i2c2 clock limit
```

图4-1 接口速率配置示意图

```
<*> Hisilicon I2C Controller support
(0x200d0000) hi i2c0 register base address
(0x1000) hi i2c0 register size
(0x20240000) hi i2c1 register base address
(0x1000) hi i2c1 register size
(0x20250000) hi i2c2 register base address
(0x1000) hi i2c2 register size
(0x1) hi i2c retry times
(0x8) hi i2c tx fifo
(0x8) hi i2c rx fifo
(100000) hi i2c0 clock limit
(100000) hi i2c1 clock limit
(100000) hi i2c2 clock limit
```

4.4 操作示例

4.4.1 I2C 读写命令示例

此操作示例通过 I2C 读写命令实现对 I2C 外围设备的读写操作。

a. 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> <reg_step>
```

例如读挂载在 I2C 控制器 2 上的 sil9024 设备的 0x8 寄存器：

```
~ $ i2c_read 2 0x72 0x8 0x8 0x1 0x1
```

说明

i2c_num：I2C 控制器序号（对应《Hi3516A 专业型 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（Hi3516A 只支持 7bit 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）

data_width：外围设备的数据位宽（Hi3516A 支持 8/16bit）

reg_step：连续读外围设备寄存器操作时递增幅值，默认为 1，即连续读寄存器，读取单个寄存器时不使用该参数

b. 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> <value> <reg_width>
<data_width>
```

例如向挂载在 I2C 控制器 2 上的 sil9024 设备的 0x8 寄存器写入数据 0xa5：

```
~ $ i2c_write 2 0x72 0x8 0xa5 0x1 0x1
```

说明

i2c_num：I2C 控制器编号（对应《Hi3516A 专业型 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（Hi3516A 的 I2C 控制器只支持 7bit 设备地址）

reg_addr：写外围设备寄存器操作的地址

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）

4.4.2 内核态 I2C 读写程序示例：

此操作示例在内核态下通过 I2C 读写程序实现对 I2C 外围设备的读写操作。

步骤 1. 调用 I2C 核心层的函数，获得描述一个 I2C 控制器的结构体 i2c_adap：

```
i2c_adap = i2c_get_adapter(2);
```

说明

假设我们已经知道新增的器件挂载在 I2C 控制器 2 上，直接设置 i2c_get_adapter 的参数为 2。

步骤 2. 把 I2C 控制器和新增的 I2C 外围设备关联起来，得到描述 I2C 外围设备的客户端结构体 hi_client：

```
hi_client = i2c_new_device(i2c_adap, &hi_info);
```

说明

hi_info 结构体提供了 I2C 外围设备的设备地址

步骤 3. 调用 I2C 核心层提供的标准读写函数对外围器件进行读写：

```
ret = i2c_master_send(client, buf, count);
ret = i2c_master_recv(client, buf, count);
```

说明

参数 client 为步骤 2 得到的描述 I2C 外围设备的客户端结构体 hi_client。

参数 buf 为需要读写的寄存器和数据。

参数 count 为 buf 的长度。

代码示例如下：

说明

此代码为示例程序，仅为客户开发内核态的 I2C 外围设备驱动程序提供参考，不提供实际应用功能。

内核态具体 I2C 外围设备驱动程序可以参考 SDK 发布包中的 tlv320aic31 程序，具体路径为：mpp/extdrv/tlv320aic31/tlv320aic31.c

//I2C 外围设备信息列表

```
static struct i2c_board_info hi_info = {
```

//一项 I2C_BOARD_INFO 代表一个支持的 I2C 外围设备，它的名字叫做"hi_test”，设备地址是 0x72

```
        I2C_BOARD_INFO("hi_test", 0x72),
};

static struct i2c_client *hi_client;

static int i2c_drv_write(char *buf, unsigned int count,
              unsigned int reg_addr_num, unsigned int data_byte_num)
{
       int ret;
       char *tmp;
       struct i2c_client *client = hi_client;
```

//根据寄存器位宽参数设置标志

```
       if (reg_addr_num == 2)
              client->flags |= I2C_M_16BIT_REG;
```

//根据数据位宽参数设置标志

//如果数据的长度大于等于 3 字节，则需要使用 i2c 的 dma 模式进行写操作，需要 dma 操作的标志，I2C_M_DMA 和 I2C_M_16BIT_DATA 不能同时设置。

```
       if (data_byte_num == 2)
              client->flags |= I2C_M_16BIT_DATA;
       if (data_byte_num >= 3)
              client->flags |= I2C_M_DMA;
```

//调用内核提供的 I2C 标准写函数进行写操作

// i2c_master_send 函数中最后一个参数 count 代表发送的寄存器地址和数据的总字节长度

```
        ret = i2c_master_send(client, buf, count);
        return ret;
}

static int i2c_drv_read(char *buf, unsigned int count,
                unsigned int reg_addr_num, unsigned int data_byte_num)
{
        int ret;
        struct i2c_client *client = hi_client;
```

//根据寄存器位宽参数设置标志

```
        if (reg_addr_num == 2)
                client->flags |= I2C_M_16BIT_REG;
```

//根据数据位宽参数设置标志

//如果数据的长度大于等于 3 字节，则需要使用 i2c 的 dma 模式进行读操作，需要 dma 操作的标志，I2C_M_DMA 和 I2C_M_16BIT_DATA 不能同时设置。

```
        if (data_byte_num == 2)
                client->flags |= I2C_M_16BIT_DATA;
        if (data_byte_num == 2)
                client->flags |= I2C_M_DMA;
```

//调用内核提供的 I2C 标准读函数进行读操作

// i2c_master_recv 函数中的最后一个参数 count 代表要读的数据的字节长度

```
        ret = i2c_master_recv(client, buf, count);


        return ret;
}
static int hi_dev_init(void)
{
```

//分配一个 I2C 控制器指针

```
        struct i2c_adapter *i2c_adap;
```

//调用 core 层的函数，获得描述一个 I2C 控制器的结构体 i2c_adap。假设我们已经知道新增的外围设备挂载在编号为 I2C 控制器 2 上

```
        i2c_adap = i2c_get_adapter(2);
```

//把 I2C 控制器和新增的 I2C 外围设备关联起来，I2C 外围设备挂载在 I2C 控制器 2，地址是 0x72，就组成了一个客户端 hi_client。

```
        hi_client = i2c_new_device(i2c_adap, &hi_info);
        i2c_put_adapter(i2c_adap);
        return 0;
}
static void hi_dev_exit(void)
{
         i2c_unregister_device(hi_client);
}
```

----结束

4.4.3 用户态 I2C 读写程序示例：

此操作示例在用户态下通过 I2C 读写程序实现对 I2C 外围设备的读写操作。

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-2", O_RDWR);
```

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长度：

```
ret = ioctl(fd, I2C_SLAVE_FORCE, device_addr);
ioctl(fd, I2C_16BIT_REG, 0);
ioctl(fd, I2C_16BIT_DATA, 0);
ioctl(fd, I2C_DMA, 0);
```

说明

- **设置寄存器位宽和数据长度时**，ioctl **的第三个参数为** 0 **表示** 8bit **位宽**，**为** 1 **表示** 16bit **位宽。**
- **设置** dma **模式是**，ioctl **的第三个参数为** 0 **表示非** dma **模式**，**为** 1 **表示** dma **模式。**
- I2C_DMA **和** I2C_16BIT_DATA **不能被同时设置。**

步骤 3. 使用 read/wite 进行数据读写：

读操作为非 dma 模式时，read 函数的最后一个参数表示寄存器位宽：

```
read(fd, recvbuf, reg_width);
```

读操作为 dma 模式时，read 函数的最后一个参数表示要读的数据的字节长度：

```
read(fd, recvbuf, data_len);
```

写操作操作时，write 函数的最后一个参数表示要写的寄存器地址和数据的总字节长度：

```
write(fd, buf, (reg_width + data_width));
```

代码示例如下：

说明

此代码为示例程序，仅为客户开发用户态的 I2C 外围设备驱动程序提供参考，不提供实际应用功能。

用户态具体 I2C 外围设备驱动程序可以参考 SDK 发布包中的 i2c_ops 程序，具体路径为：osdrv/tools/board/reg-tools-1.0.0/source/tools/i2c_ops.c

```
int i2c_read(unsigned int i2c_num, unsigned int device_addr, unsigned int reg_addr, unsigned int reg_addr_end, char data, unsigned int reg_width, unsigned int data_width, unsigned int reg_step)
{
    int fd = -1;
    int ret;
    char recvbuf[4];
    int cur_addr;

    memset(recvbuf, 0x0, 4);

    fd = open("/dev/i2c-2", O_RDWR);

    if (fd<0)
    {
        printf("Open i2c dev error!\n");
        return -1;
    }

    ret = ioctl(fd, I2C_SLAVE_FORCE, device_addr);

    if (reg_width == 2)
        ret = ioctl(fd, I2C_16BIT_REG, 1);
    else
        ret = ioctl(fd, I2C_16BIT_REG, 0);
    if (ret < 0) {
        printf("CMD_SET_REG_WIDTH error!\n");
        close(fd);
        return -1;
    }

    if (data_width == 2)
        ret = ioctl(fd, I2C_16BIT_DATA, 1);
    else
        ret = ioctl(fd, I2C_16BIT_DATA, 0);
    if (ret < 0) {
        printf("CMD_SET_DATA_WIDTH error!\n");
        close(fd);
        return -1;
```

```
        }

    for (cur_addr = reg_addr; cur_addr < reg_addr_end + reg_width;
cur_addr += reg_step)
    {
        if (reg_width == 2) {
            recvbuf[0] = cur_addr & 0xff;
            recvbuf[1] = (cur_addr >> 8) & 0xff;
        } else
            recvbuf[0] = cur_addr & 0xff;

        ret = read(fd, recvbuf, reg_width);
        if (ret < 0) {
            printf("CMD_I2C_READ error!\n");
            close(fd);
            return -1;
            }

        if (data_width == 2) {
            data = recvbuf[0] | (recvbuf[1] << 8);
        } else
            data = recvbuf[0];

        printf("0x%x 0x%x\n", cur_addr, data);
    }

    close(fd);
    return 0;
}

int i2c_write(unsigned int i2c_num, unsigned int device_addr, unsigned
int reg_addr, unsigned int reg_value, unsigned int reg_width, unsigned
int data_width)
{
    int fd = -1;
    int ret =0, index = 0;
    char buf[4];

    fd = open("/dev/i2c-2", O_RDWR);
    if(fd < 0)
    {
        printf("Open i2c dev error!\n");
        return -1;
    }
```

```
    ret = ioctl(fd, I2C_SLAVE_FORCE, device_addr);

    if (reg_width == 2)
        ret = ioctl(fd, I2C_16BIT_REG, 1);
    else
        ret = ioctl(fd, I2C_16BIT_REG, 0);

    if (data_width == 2)
        ret = ioctl(fd, I2C_16BIT_DATA, 1);
    else
        ret = ioctl(fd, I2C_16BIT_DATA, 0);

    if (reg_width == 2) {
        buf[index] = reg_addr & 0xff;
        index++;
        buf[index] = (reg_addr >> 8) & 0xff;
        index++;
    } else {
        buf[index] = reg_addr & 0xff;
        index++;
    }

    if (data_width == 2) {
        buf[index] = reg_value & 0xff;
        index++;
        buf[index] = (reg_value >> 8) & 0xff;
        index++;
    } else {
        buf[index] = reg_value & 0xff;
        index++;
    }

    write(fd, buf, (reg_width + data_width));
    if(ret < 0)
    {
        printf("I2C_WRITE error!\n");
        return -1;
    }

    close(fd);
    return 0;
}
int i2c_dma_read(unsigned int i2c_num, unsigned int device_addr, unsigned
```

```
int reg_addr, char *recvbuf, unsigned int reg_width, unsigned int
data_len)
{
    int fd = -1;
    int ret;

    memset(recvbuf, 0x0, 4);

    fd = open("/dev/i2c-2", O_RDWR);

    if (fd<0)
    {
        printf("Open i2c dev error!\n");
        return -1;
    }

    ret = ioctl(fd, I2C_SLAVE_FORCE, device_addr);
    if (ret < 0) {
         printf("I2C_SLAVE_FORCE error!\n");
         close(fd);
         return -1;
    }
    if (reg_width == 2)
        ret = ioctl(fd, I2C_16BIT_REG, 1);
    else
        ret = ioctl(fd, I2C_16BIT_REG, 0);

    if (ret < 0) {
        printf("CMD_SET_REG_WIDTH error!\n");
        close(fd);
        return -1;
    }

    if (dma)
           ret = ioctl(fd, I2C_DMA, 1);
    else
           ret = ioctl(fd, I2C_DMA, 0);

     if (ret < 0) {
           printf("CMD_SET_DMA error!\n");
           close(fd);
           return -1;
    }
```

```
    if (reg_width == 2) {
        recvbuf[0] = cur_addr & 0xff;
        recvbuf[1] = (cur_addr >> 8) & 0xff;
    } else
        recvbuf[0] = cur_addr & 0xff;

    ret = read(fd, recvbuf, data_len);
    if (ret < 0) {
        printf("CMD_I2C_READ error!\n");
        close(fd);
        return -1;
    }

   printf("reg_addr: 0x%x:\n\r", reg_addr);
   for (i = 0; i < data_len; i++)
       printf("  0x%x\n", recv_buf[i]);
    printf("\n\r");

    close(fd);
    return 0;
}

HI_RET i2c_dma_write(int argc , char* argv[])
int i2c_dma_write(unsigned int i2c_num, unsigned int device_addr, char
*buf, unsigned int reg_width, unsigned int data_len)

{
    int fd = -1;
    int ret =0;

    fd = open("/dev/i2c-2", O_RDWR);
    if(fd < 0)
    {
        printf("Open i2c dev error!\n");
        return -1;
    }

    ret = ioctl(fd, I2C_SLAVE_FORCE, device_addr);
if (ret < 0) {
         printf("I2C_SLAVE_FORCE error!\n");
         close(fd);
         return -1;
    }
```

```
    if (reg_width == 2)
        ret = ioctl(fd, I2C_16BIT_REG, 1);
    else
        ret = ioctl(fd, I2C_16BIT_REG, 0);

    if (ret < 0) {
        printf("CMD_SET_REG_WIDTH error!\n");
        close(fd);
        return -1;
    }

    if (dma)
         ret = ioctl(fd, I2C_DMA, 1);
    else
         ret = ioctl(fd, I2C_DMA, 0);

    if (ret < 0) {
         printf("CMD_SET_DMA error!\n");
         close(fd);
         return -1;
    }

    write(fd, buf, (reg_width + data_len));
    if(ret < 0)
    {
        printf("I2C_WRITE error!\n");
        return -1;
    }

    close(fd);
    return 0;
}
```

----结束

5 SPI 操作指南

5.1 操作准备

SPI 的操作准备如下：

Linux 内核使用 SDK 发布的 kernel。

文件系统。

可以使用 SDK 发布的本地文件系统 yaffs2、jffs2 或 SquashFS，也可以通过本地文件系统再挂载到 NFS。

5.2 操作过程

操作过程如下：

步骤 1. 启动单板，加载本地文件系统 yaffs2、jffs2 或 SquashFS，也可以通过本地文件系统进一步挂载到 NFS。

步骤 2. 加载内核。默认 SPI 相关模块已全部编入内核，不需要再执行加载命令。

步骤 3. 在控制台下运行 SPI 读写命令或者自行在内核态或者用户态编写 SPI 读写程序，就可以对挂载在某个 SPI 控制器某个片选上的外围设备进行读写操作。具体操作请参见“5.3 操作示例”。

----结束

5.3 操作示例

5.3.1 SPI 读写命令示例

此操作示例通过 SPI 读写命令实现对 SPI 外围设备的读写操作。

- 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《Hi3516A 专业型 HD IP Camera Soc 用户指南》中的 SPI 控制器 0、1）

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）

- 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 1Byte）。

  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《Hi3516A 专业型 HD IP Camera Soc 用户指南》中的 SPI 控制器 0、1）

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）

NOTE

此 SPI 读写命令仅支持 sensor 的读写操作。

----结束

5.3.2 内核态 SPI 读写程序示例

此操作示例在内核态下通过 SPI 读写程序实现对 SPI 外围设备的读写操作。

步骤 1. 调用 SPI 核心层的函数，获得描述一个 SPI 控制器的结构体：

```
hi_master = spi_busnum_to_master(bus_num);
```

说明

参数 bus_num 为要读写的 SPI 外围设备所连接的 SPI 控制器号。

hi_master 为描述 SPI 控制器的 spi_master 结构体类型指针变量。

步骤 2. 通过每个 spi 片选在核心层的名称调用 SPI 核心层函数得到挂载在某个 spi 控制器某个片选上描述 SPI 外围设备的结构体：

```
sprintf(spi_name, "%s.%u", dev_name(&hi_master->dev),csn);
d = bus_find_device_by_name(&spi_bus_type, NULL, spi_name);
hi_spi = to_spi_device(d);
```

说明

spi_bus_type 为核心层定义的描述 spi 总线的 bus_type 结构体类型变量。

hi_spi 为描述 SPI 外围设备的 spi_device 结构体类型指针变量。

步骤 3. 设置 spi_transfer 结构体中的成员，调用 SPI 核心层函数将 spi_transfer 添加到 spi_message 的队列当中。

```
spi_message_init(&m);
spi_message_add_tail(&t, &m);
```

说明

参数 t 为描述传输一桢消息的 spi_transfer 结构体类型变量。

参数 m 为描述传输一个消息队列的 spi_message 结构体类型变量。

步骤 4. 然后调用 SPI 核心层提供的标准读写函数对外围器件进行读写：

```
status = spi_async(spi, &m);
```

status = spi_sync(spi, &m);

说明

参数 spi 为描述 SPI 外围设备的 spi_device 结构体类型指针变量。

spi_async 函数进行 spi 异步读写操作。

spi_sync 函数进行 spi 同步读写操作。

代码示例如下：

说明

此代码为异步读写 SPI 外围设备 imx185 的示例程序，仅为客户开发内核态的 SPI 外围设备驱动程序提供参考，不提供实际应用功能。

内核态具体 SPI 外围设备驱动程序可以参考 SDK 发布包中的 sensor_spi 程序，具体路径为：mpp/extdrv/sensor_spi/sensor_spi.c

//模块参数，传入 spi 控制器号即 spi 总线号和 spi 片选号

```
static unsigned bus_num = 0;
static unsigned csn = 0;
module_param(bus_num, uint, S_IRUGO);
MODULE_PARM_DESC(bus_num, "spi bus number");
module_param(csn, uint, S_IRUGO);
MODULE_PARM_DESC(csn, "chip select number");
```

//描述 SPI 控制器的结构体

```
struct spi_master  *hi_master;
```

//描述 SPI 外围设备的结构体

```
struct spi_device  *hi_spi;

int ssp_write_alt(unsigned char devaddr,unsigned char addr, unsigned char
data)
{
    struct spi_master       *master = hi_master;
    struct spi_device      *spi = hi_spi;
    static struct spi_transfer  t;
    static struct spi_message      m;
    static unsigned char       buf[4];
    int               status = 0;
    unsigned long           flags;

    /* check spi_message is or no finish */
    spin_lock_irqsave(&master->queue_lock, flags);
```

//该消息队列传输完成之后，在核心层会将 spi_message 的 state 成员设为空指针。

```
    if (m.state != NULL) {
        spin_unlock_irqrestore(&master->queue_lock, flags);
        return -EFAULT;
    }
    spin_unlock_irqrestore(&master->queue_lock, flags);
```

//设置 SPI 传输模式

```
    spi->mode = SPI_MODE_3 | SPI_LSB_FIRST;

    memset(buf, 0, sizeof buf);
    buf[0] = devaddr;
    buf[0] &= (~0x80);
    buf[1] = addr;
    buf[2] = data;
```

```
    t.tx_buf = buf;
    t.rx_buf = buf;
    t.len = 3;
    t.cs_change = 1;
    t.bits_per_word = 8;
    t.speed_hz = 2000000;
```

//初始化化并设置 SPI 传输队列

```
    spi_message_init(&m);
    spi_message_add_tail(&t, &m);
    m.state = &m;
```

NOTE

进行异步的 spi 读写操作，由于是异步操作，因此该调用返回时，spi 读写操作不一定完成，因此往该调用传的参数所指的地址空间必须是局部静态变量或全局变量，以防止函数返回时将传给 spi_async 的地址空间释放掉。spi_async 函数的原型为 int spi_async(struct spi_device *spi, struct spi_message *message)，则在这里变量 m 和 t 都必须为静态变量，并且 t 中所指的 buf 也必须是静态的。

```
    status = spi_async(spi, &m);
    return status;
}

int ssp_read_alt(unsigned char devaddr,unsigned char addr, unsigned char
*data)
{
    struct spi_master       *master = hi_master;
    struct spi_device      *spi = hi_spi;
    static struct spi_transfer  t;
    static struct spi_message      m;
    static unsigned char       buf[4];
    int             status = 0;
    unsigned long       flags;

    /* check spi_message is or no finish */
    spin_lock_irqsave(&master->queue_lock, flags);
```

//该消息队列传输完成之后，在核心层会将 spi_message 的 state 成员设为空指针。

```
    if (m.state != NULL) {
        spin_unlock_irqrestore(&master->queue_lock, flags);
        return -EFAULT;
    }
    spin_unlock_irqrestore(&master->queue_lock, flags);
```

//设置 SPI 传输模式

```
    spi->mode = SPI_MODE_3 | SPI_LSB_FIRST;

    memset(buf, 0, sizeof buf);
    buf[0] = devaddr;
    buf[0] |= 0x80;
    buf[1] = addr;
    buf[2] = 0;

    t.tx_buf = buf;
    t.rx_buf = buf;
    t.len = 3;
    t.cs_change = 1;
    t.bits_per_word = 8;
    t.speed_hz = 2000000;
```

//初始化化并设置 SPI 传输队列

```
    spi_message_init(&m);
    spi_message_add_tail(&t, &m);
    m.state = &m;
```

NOTE

进行异步的 spi 读写操作，由于是异步操作，因此该调用返回时，spi 读写操作不一定完成，因此往该调用传的参数所指的地址空间必须是局部静态变量或全局变量，以防止函数返回时将传给 spi_async 的地址空间释放掉。spi_async 函数的原型为 int spi_async(struct spi_device *spi, struct spi_message *message)，则在这里变量 m 和 t 都必须为静态变量，并且 t 中所指的 buf 也必须是静态的。

```
    status = spi_async(spi, &m);
    *data = buf[2];
    return status;
}
```

//外部引用声明 SPI 核心层定义的表示 spi 总线的 spi_bus_type

```
extern struct bus_type spi bus_type;

static int __init sspdev_init(void)
{
    int         status = 0;
    struct device       *d;
    char            *spi_name;
```

//通过 spi 控制器号得到描述 spi 控制器的结构体

```
    hi_master = spi_busnum_to_master(bus_num);
    if (hi_master) {
        spi_name = kzalloc(strlen(dev_name(&hi_master->dev)) + 10 ,
```

```
GFP_KERNEL);
        if (!spi_name) {
            status = -ENOMEM;
            goto end0;
        }
        sprintf(spi_name, "%s.%u", dev_name(&hi_master->dev),csn);
```

//通过每个片选在 SPI 核心层的名称得到指向 spi_device 的 device 成员的指针

```
        d = bus_find_device_by_name(&spi_bus_type, NULL, spi_name);
        if (d == NULL) {
            status = -ENXIO;
            goto end1;
        }
```

//通过指向 spi_device 的 device 成员的指针得到描述 SPI 外围设备的结构体

```
        hi_spi = to_spi_device(d);
        if(hi_spi == NULL) {
            status = -ENXIO;
            goto end2;
        }
    } else {
        status = -ENXIO;
        goto end0;
    }

    status = 0;
end2:
    put_device(d);
end1:
    kfree(spi_name);
end0:
    return status;
}
```

----结束

5.3.3 用户态 SPI 读写程序示例

此操作示例在用户态下实现对挂载在 SPI 控制器 0 片选 0 上的 SPI 外围设备的读写操作。

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);
```

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

SPI_MODE_3 和 SPI_LSB_FIRST 的含义可参考内核代码 include/linux/spi/spi.h，用户态下使用该宏可包含 SDK 发布包中的 osdrv/tools/board/reg-tools-1.0.0/include/common/hi_spi.h 头文件。

SPI 的时钟、相位、大小端结束模式可参考《Hi3516A 专业型 HD IP Camera Soc 用户指南》。

步骤 3. 使用 ioctl 进行数据读写：

```
ret = ioctl(fd, SPI_IOC_MESSAGE(1), mesg);
```

说明

mesg 表示传输一帧消息的 spi_ioc 结构体数组首地址，并且一次读写的数据总长度不超过 4KB。

SPI_IOC_MESSAGE(n)表示全双工读写 n 帧消息的命令。

代码示例如下：

说明

此代码为同步读写 SPI 外围设备 imx185 的示例程序，仅为客户开发用户态的 SPI 外围设备操作程序提供参考，不提供实际应用功能。

用户态具体 SPI 外围设备驱动程序可以参考 SDK 发布包中的 ssp_rw 程序，具体路径为：osdrv/tools/board/reg-tools-1.0.0/source/tools/ssp_rw.c

```
int sensor_write_register(unsigned int addr, unsigned char data)
{
    int fd = -1;
    int ret;
    unsigned int    value;
    struct spi_ioc_transfermesg[1];
    unsigned char       tx_buf[4];
    unsigned char   rx_buf[4];
    char    file_name[] = "/dev/spidev0.0";
    fd = open(file_name, 0);
    if (fd < 0) {
        return -1;
    }
    memset(tx_buf, 0, sizeof tx_buf);
```

```
    memset(rx_buf, 0, sizeof rx_buf);
    tx_buf[0] = (addr & 0xff00) >> 8;
    tx_buf[0] &= (~0x80);
    tx_buf[1] = addr & 0xff;
    tx_buf[2] = data;

    memset(mesg, 0, sizeof mesg);
    mesg[0].tx_buf = (__u32)tx_buf;
    mesg[0].rx_buf = (__u32)rx_buf;
    mesg[0].len = 3;
    mesg[0].speed_hz = 2000000;
    mesg[0].bits_per_word = 8;
    mesg[0].cs_change = 1;

    value = SPI_MODE_3 | SPI_LSB_FIRST;
    ret = ioctl(fd, SPI_IOC_WR_MODE, &value);
    if (ret < 0) {
        close(fd);
        return -1;
    }

    ret = ioctl(fd, SPI_IOC_MESSAGE(1), mesg);
    if (ret != mesg[0].len) {
        close(fd);
        return -1;
    }
    close(fd);
    return 0;
}

int sensor_read_register(unsigned int addr,unsigned char *data)
{
    int fd = -1;
    int ret = 0;
    unsigned int    value;
    struct spi_ioc_transfer    mesg[1];
    unsigned char  tx_buf[4];
    unsigned char  rx_buf[4];
    char    file_name[] = "/dev/spidev0.0";

    fd = open(file_name, 0);
    if (fd < 0) {
        return -1;
    }
```

```
    memset(tx_buf, 0, sizeof tx_buf);
    memset(rx_buf, 0, sizeof rx_buf);
    tx_buf[0] = (addr & 0xff00) >> 8;
    tx_buf[0] |= 0x80;
    tx_buf[1] = addr & 0xff;
    tx_buf[2] = 0;

    memset(mesg, 0, sizeof mesg);
    mesg[0].tx_buf = (__u32)tx_buf;
    mesg[0].rx_buf = (__u32)rx_buf;
    mesg[0].len = 3;
    mesg[0].speed_hz = 2000000;
    mesg[0].bits_per_word = 8;
    mesg[0].cs_change = 1;
    value = SPI_MODE_3 | SPI_LSB_FIRST;
    ret = ioctl(fd, SPI_IOC_WR_MODE, &value);
    if (ret < 0) {
        close(fd);
        return -1;
    }
    ret = ioctl(fd, SPI_IOC_MESSAGE(1), mesg);
    if (ret != mesg[0].len) {
        close(fd);
        return -1;
    }
    *data = rx_buf[2];
    close(fd);
    return 0;
}
```

----结束

6 附录

6.1 用 fdisk 工具分区

通过“6.1.1 查看当前状态”，对应以下情况选择操作：

- 若已有分区，本操作可以跳过，直接到“6.2 用 mkdosfs 工具格式化”。
- 若没有分区，则在控制台的提示符下，输入命令 fdisk，具体格式如下：

  ```
  ~ $ fdisk 设备节点
  ```

回车后，输入命令 m，根据帮助信息继续进行以下的操作。

其中设备节点与实际接入的设备类型有关，具体名称在以上各章节的“操作示例”中均有说明。

6.1.1 查看当前状态

在控制台的提示符下，输入命令 p，查看当前分区状态：

```
Command (m for help): p
```

控制台显示出分区状态信息：

```
Disk /dev/mmc/blk1/disc: 127 MB, 127139840 bytes
8 heads, 32 sectors/track, 970 cylinders
Units = cylinders of 256 * 512 = 131072 bytes
Device Boot Start End Blocks Id System
```

上面信息表明设备没有分区，需要按照“6.1.2 创建新的分区”和“6.1.3 保存分区信息”的描述对设备进行分区。

6.1.2 创建新的分区

创建新的分区步骤如下：

步骤 1. 创建新的分区。

在提示符下输入命令 n，创建新的分区：

```
Command (m for help): n
```

控制台显示出如下信息：

```
Command action
e extended
p primary partition (1-4)
```

步骤 2. 建立主分区。

输入命令 p，选择主分区：

```
p
```

步骤 3. 选择分区数。

本例中选择为 1，输入数字 1：

```
Partition number (1-4): 1
```

控制台显示出如下信息：

```
First cylinder (1-970, default 1):
```

步骤 4. 选择起始柱面。

本例选择默认值 1，直接回车：

```
Using default value 1
```

步骤 5. 选择结束柱面。

本例选择默认值 970，直接回车：

```
Last cylinder or +size or +sizeM or +sizeK (1-970, default 970):
Using default value 970
```

步骤 6. 选择系统格式。

由于系统默认为 Linux 格式，本例中选择 Win95 FAT 格式，输入命令 t 进行修改：

```
Command (m for help): t
Selected partition 1
```

输入命令 b，选择 Win95 FAT 格式：

```
Hex code (type L to list codes): b
```

输入命令 l，可以查看 fdisk 所有分区的详细信息：

```
Changed system type of partition 1 to b (Win95 FAT32)
```

步骤 7. 查看分区状态。

输入命令 p，查看当前分区状态：

```
Command (m for help): p
```

控制台显示出当前分区状态信息，表示成功分区。

----结束

6.1.3 保存分区信息

输入命令 w，写入并保存分区信息到设备：

```
Command (m for help): w
```

控制台显示出当前设备信息，表示成功写入分区信息到设备：

```
The partition table has been altered!
Calling ioctl() to re-read partition table.
.....
~ $
```

6.2 用 mkdosfs 工具格式化

存在以下情况选择操作：

- 若已格式化，本操作可以跳过，直接到“6.3 挂载目录”。
- 若没有格式化，则输入命令 mkdosfs 进行格式化：

  ```
  ~ $ mkdosfs -F 32 设备分区名
  ```

其中设备分区名与实际接入的设备类型有关，具体名称在以上各章节的“操作示例”中均有说明。

控制台没有显示错误提示信息，表示成功格式化：

~ $

6.3 挂载目录

使用命令 mount 挂载到 mnt 目录下，就可以进行读写文件操作：

```
~ $ mount -t vfat 设备分区名 /mnt
```

其中设备分区名与实际接入的设备类型有关，具体名称在以上各章节的“操作示例”中均有说明。

6.4 读写文件

读写操作的具体情况很多，在本例中使用命令 cp 实现读写操作。

使用命令 cp 拷贝当前目录下的 test.txt 文件到 mnt 目录下，即拷贝至设备，实现写操作，如：

```
~ $ cp ./test.txt /mnt
```