外围设备驱动

操作指南

文档版本　　00B04

发布日期　　2018-10-26

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、USB3.0 DRD 和 SD/MMC/eMMC 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 Hi3519AV100 描述为例，未有特殊说明，Hi3556AV100 与 Hi3519AV100 一致。

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
Hi3519A	V100
Hi3556A	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
2018-10-26	00B04	1.2.2.2 小节的步骤 1 和步骤 3，图 1-3、图 1-4 涉及修改 1.2.3 小节涉及修改
2018-07-30	00B03	补充 Hi3556AV100 的相关内容
2018-07-09	00B02	1.1.2、1.1.3、1.2.2 小节涉及修改 新增 1.2.3.6 小节 1.4.3 小节，图 1-7 涉及修改
2018-06-15	00B01	第 1 次临时版本发布

目 录

插图目录

表格目录

1 Linux

1.1 GMAC 操作指南

注意

- Hi3556AV100 平台没有 GMAC 模块。
- 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。

1.1.1 操作示例

说明

Hi3519AV100 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：

- 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。

1.1.2 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：

```
cd kernel/linux-4.9.y
cp arch/arm/configs/hi3519av100_xxx_defconfig .config
make ARCH=arm CROSS_COMPILE=arm-himix100-linux- menuconfig
```

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

```
-*- The IPv6 protocol
[ ]   IPv6: Router Preference (RFC 4191) support (NEW)
[ ]   IPv6: Enable RFC 4429 Optimistic DAD (NEW)
< >   IPv6: AH transformation (NEW)
< >   IPv6: ESP transformation (NEW)
< >   IPv6: IPComp transformation (NEW)
< >   IPv6: Mobility (NEW)
<*>   IPv6: IPsec transport mode (NEW)
<*>   IPv6: IPsec tunnel mode (NEW)
<*>   IPv6: IPsec BEET mode (NEW)
< >   IPv6: MIPv6 route optimization mode (NEW)
< >   Virtual (secure) IPv6: tunneling (NEW)
<*>   IPv6: IPv6-in-IPv4 tunnel (SIT driver) (NEW)
[ ]     IPv6: IPv6 Rapid Deployment (6RD) (NEW)
< >   IPv6: IP-in-IPv6 tunnel (RFC2473) (NEW)
[ ]   IPv6: Multiple Routing Tables (NEW)
[ ]   IPv6: multicast routing (NEW)
```

IPv6 环境配置如下：

- 配置 ip 地址及缺省网关

  ```
  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

  ```
  ping -6  <ipv6address>
  ```

  示例：`ping -6  2001:da8:207::9403`

1.1.3 PHY 地址配置

Hi3519AV100 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。

- 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_PHY0_ADDR 的值来配置不同的 PHY 地址。Hi3519AV100 的 U-boot 配置文件如下：

  - include/configs/hi3519av100.h

- Kernel 下配置方式

  在 Kernel 下可通过修改目录 arch/arm/boot/dts 下的 dts 配置文件配置 PHY 地址。

  对于 Hi3519AV100，配置文件为：hi3519av100.dts 和 hi3519av100-smp.dts；

  如图 1-2 所示，“reg = <1>”中的数值 1 表示 PHY 地址。

图1-2 PHY 地址配置节点示意图

```
&mdio {
        ethphy: ethernet-phy@1 {
                reg = <1>;
        };
};
```

1.1.4 网络加速特性

MAC 支持以下网卡硬件加速特性：

- TXCOE（Tx checksum offload engine）：发送数据包 checksum 计算。
- RXCOE（Rx checksum offload engine）：接收数据包 checksum 计算。
- SG（Scatter-Gather）：发送分散聚合特性。
- TSO（TCP Segmentation Offload）：TCP 发送硬件分片功能。
- UFO（UDP Fragmentation Offload）：UDP 发送硬件分片功能。

默认配置下，上述所有硬件特性开启，以达到节省 cpu 处理的目的。

可以通过标准 ethtool 工具查看和修改以上网络特性。

1.1.4.1 ethtool 查看网络加速特性

执行如下命令：

./ethtool –k eth0

查看打印的网络加速特性信息如下：

```
~ # ./ethtool -k eth0
Features for eth0:
rx-checksumming: on
```

```
tx-checksumming: on
        tx-checksum-ipv4: on
        tx-checksum-ip-generic: off [fixed]
        tx-checksum-ipv6: on
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp-ecn-segmentation: off [fixed]
        tx-tcp6-segmentation: on
```

udp-fragmentation-offload: on

1.1.4.2 ethtool 修改网络加速特性

修改时，只需要配置相关特性是 on 还是 off。

```
./ethtool -K eth0 tx off（关闭TXCOE特性）
./ethtool -K eth0 tx on（打开TXCOE特性）
./ethtool -K eth0 rx off（关闭RXCOE特性）
./ethtool -K eth0 rx on（打开RXCOE特性）
./ethtool -K eth0 sg off（关闭SG特性）
./ethtool -K eth0 sg on（打开SG特性）
./ethtool -K eth0 tso off（关闭TSO特性）
./ethtool -K eth0 tso on（打开TSO特性）
./ethtool -K eth0 ufo off（关闭UFO特性）
./ethtool -K eth0 ufo on（打开UFO特性）
```

注意

相关特性有依赖关系，TSO、UFO 特性都依赖 TXCOE 和 SG 特性，关闭 TXCOE 或者 SG 特性时，TSO 和 UFO 特性会自动关闭。

建议客户不要关闭这些加速特性，因为这些特性对降低 cpu 占用率是有效果的。

1.2 USB 3.0 DRD 操作指南

说明

- Hi3519AV100 的 USB3.0 DRD (Dule Role Device)模块支持 USB3.0 Host 和 USB3.0 Device 两种模式。
- Hi3519AV100 的 USB 3.0 只能 Host 模式或 Device 模式二选一使用。

1.2.1 操作准备

USB 3.0 DRD 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、ext4 或 cramfs，也可以使用 NFS，建议使用 jffs2。

1.2.2 操作过程

1.2.2.1 USB Host 操作过程

步骤 1. 启动单板，加载 yaffs2、jffs2、ext4 或 cramfs 文件系统，也可以使用 NFS。

步骤 2. 默认 USB Host 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操作。具体操作请参见“1.2.3 操作示例”。下面列出所有 USB Host 相关驱动：

- 文件系统和存储设备相关模块
  - vfat
  - ext4
  - 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
- USB3.0 Host 模块
  - xhci-hcd
  - usb-storage
  - phy-hisi-usb3

----结束

1.2.2.2 USB Device 操作过程

USB 3.0 Device 操作过程

步骤 1. 编译 USB 3.0 Device 相关的内核驱动模块。

- 进入 menuconfig 的如下路径，并 USB3.0 device 作为复合设备（网口/串口）、u 盘的配置如下。

```
Device Drivers  --->
    [*] USB support  --->
        <M>   DesignWare USB3 DRD Core Support
            DWC3 Mode Selection (Gadget only mode)  --->
        <*>   USB Gadget Support  --->
            <M>   USB Gadget Drivers
            < >     Ethernet Gadget (with CDC Ethernet support)
            <M>     Mass Storage Gadget
            < >     Serial Gadget (with CDC ACM and CDC OBEX support)
            <M>     Multifunction Composite Gadget
            [*]     RNDIS + CDC Serial + Storage configuration (NEW)
    PHY Subsystem  --->
        <*> HISI USB3 PHY Driver
        <*> HISI USB2 PHY Driver
```

说明

port0 默认配置为 device 模式，port1 默认配置为 host 模式，如需进行切换，可在 dtsi 中打开相应设备节点。

图1-3 USB DTS 节点配置示意图

```
#if 0
        xhci_0@0x04110000 {
            compatible = "generic-xhci";
            reg = <0x04110000 0x10000>;
            interrupts = <0 111 4>;
        };
#endif
#if 1
        xhci_1@0x04120000 {
            compatible = "generic-xhci";
            reg = <0x04120000 0x10000>;
            interrupts = <0 112 4>;
        };
#endif
#if 1
        hidwc3_0@0x04110000 {
            compatible = "snps,dwc3";
            reg = <0x04110000 0x10000>;
            interrupts = <0 111 4>;
            interrupt-names = "peripheral";
            maximum-speed = "super-speed";
            dr_mode = "peripheral";
        };
#endif
#if 0
        hidwc3_1@0x04120000 {
            compatible = "snps,dwc3";
            reg = <0x04120000 0x10000>;
            interrupts = <0 112 4>;
            interrupt-names = "peripheral";
            maximum-speed = "high-speed";
            dr_mode = "peripheral";
        };
#endif
```

图1-4 内核模块主要选项图

```
--- USB Gadget Support
[ ]   Debugging messages (DEVELOPMENT)
[ ]   Debugging information files (DEVELOPMENT)
[ ]   Debugging information files in debugfs (DEVELOPMENT)
(2)   Maximum VBUS Power usage (2-500 mA)
(2)   Number of storage pipeline buffers
      USB Peripheral Controller  --->
< >   USB functions configurable through configfs
<M>   USB Gadget Drivers
< >     Gadget Zero (DEVELOPMENT)
< >     Ethernet Gadget (with CDC Ethernet support)
< >     Network Control Model (NCM) support
< >     Gadget Filesystem
< >     Function Filesystem
<M>     Mass Storage Gadget
< >     Serial Gadget (with CDC ACM and CDC OBEX support)
< >     Printer Gadget
< >     CDC Composite Device (Ethernet and ACM)
< >     CDC Composite Device (ACM and mass storage)
<M>     Multifunction Composite Gadget
[*]       RNDIS + CDC Serial + Storage configuration
[ ]       CDC Ethernet + CDC Serial + Storage configuration
< >     HID Gadget
```

```
<M>   DesignWare USB3 DRD Core Support
        DWC3 Mode Selection (Gadget only mode)  --->
        *** Platform Glue Driver Support ***
< >     PCIe-based Platforms
< >     Generic OF Simple Glue Layer
```

```
-*- PHY Core
< > Marvell USB HSIC 28nm PHY Driver
< > Marvell USB 2.0 28nm PHY Driver
< > Broadcom Kona USB2 PHY Driver
<*> HISI USB2 PHY Driver
<*> HISI USB3 PHY Driver
```

- 编译内核模块，生成.ko 文件。

  ```
  make ARCH=arm CROSS_COMPILE=arm-himix200-linux- modules
  ```

- 注意：在编译模块时，要先编译内核，编译内核命令为：

  ```
  make ARCH=arm CROSS_COMPILE=arm-himix200-linux- uImage
  ```

步骤 2. 启动单板，加载 yaffs2、jffs2、ext4 或 cramfs 文件系统，也可以使用 NFS。

步骤 3. 单板作为 Device 时，须加载 USB 3.0 Device 模块才能在 Host 端被识别成 USB 存储介质。具体操作请参见“1.2.3 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

- USB 存储介质相关驱动

  dwc3.ko

  libcomposite.ko

  usb_f_mass_storage.ko

  g_mass_storage.ko

- USB 复合设备相关驱动

  dwc3.ko

  libcomposite.ko

  u_ether.ko

  u_serial.ko

  usb_f_acm.ko

  usb_f_rndis.ko

  g_multi.ko

----结束

1.2.3 操作示例

1.2.3.1 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 3.0 Host 插入高速 u 盘正常情况下串口打印为：

```
~ # usb 1-1: new high-speed USB device number 2 using hiusb-xhci
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
```

- USB 3.0 Host 插入超速 u 盘正常情况下串口打印为：

```
usb 4-1: new SuperSpeed USB device number 2 using xhci-hcd
scsi1 : usb-storage 4-1:1.0
usbdev42 -> /dev/usbdev4.2
scsi 1:0:0:0: Direct-Access     SanDisk  Extreme        0001 PQ: 0 ANSI: 6
sd 1:0:0:0: [sda] 31277232 512-byte logical blocks: (16.0 GB/14.9 GiB)
sd 1:0:0:0: Attached scsi generic sg0 type 0
sd 1:0:0:0: [sda] Write Protect is off
sd 1:0:0:0: [sda] Mode Sense: 33 00 00 08
sd 1:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't
```

```
support DPO or FUA
 sda: sda1
sd 1:0:0:0: [sda] Attached SCSI disk
udisk4110 -> /dev/sda
udisk4110p1 -> /dev/sda1
```

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“1.7.1 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“1.7.2 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“1.7.3 挂载目录”。

步骤 4. 对硬盘进行读写操作，请参见“1.7.4 读写文件”。

----结束

1.2.3.2 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/input 目录下生成 event0 节点。

步骤 2. 接收键盘输入。

执行命令：cat /dev/input/event0

然后在 USB 键盘上敲击，可以看到屏幕有输出。

----结束

1.2.3.3 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/input 目录下生成 mouse0 节点。

步骤 2. 接收鼠标输入。

执行命令：cat /dev/input/mouse0

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

1.2.3.4 USB Device 存储介质操作示例

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 dwc3.ko
insmod libcomposite.ko
insmod usb_f_mass_storage.ko
insmod g_mass_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 dwc3.ko
insmod libcomposite.ko
insmod usb_f_mass_storage.ko
insmod g_mass_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

USB Device 相关模块在 kernel 下的路径分别为：

- drivers/usb/dwc3/dwc3.ko
- drivers/usb/gadget/libcomposite.ko
- drivers/usb/gadget/function/usb_f_mass_storage.ko
- drivers/usb/gadget/legacy/g_mass_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“1.2.3.1 U 盘操作示例”。

----结束

1.2.3.5 USB Device 复合设备操作示例

单板作为 Device 时作复合设备功能，操作过程如下：

步骤 1. 插入模块。

```
insmod dwc3.ko
insmod libcomposite.ko
insmod u_ether.ko
insmod u_serial.ko
insmod usb_f_acm.ko
insmod usb_f_rndis.ko
insmod g_multi.ko
```

说明

USB Device 作为复合设备相关模块在 kernel 下的路径分别为：

- drivers/usb/dwc3/dwc3.ko
- drivers/usb/gadget/libcomposite.ko
- drivers/usb/gadget/function/u_ether.ko
- drivers/usb/gadget/function/u_serial.ko
- drivers/usb/gadget/function/usb_f_acm.ko
- drivers/usb/gadget/function/usb_f_rndis.ko
- drivers/usb/gadget/legacy/g_multi.ko

步骤 2. 在单板端，把 dwc3.ko、libcomposite.ko、u_ether.ko、u_serial.ko、usb_f_acm.ko、usb_f_rndis.ko、g_multi.ko 下载到/root/路径。

步骤 3. 执行 vi/etc/init.d/rcS，把以下命令添加进文件中，然后保存退出。

- USB2.0 device 执行如下命令：

```
cd /root/
insmod dwc3.ko
insmod libcomposite.ko
insmod u_ether.ko
insmod u_serial.ko
insmod usb_f_acm.ko
insmod usb_f_rndis.ko
insmod g_multi.ko
```

步骤 4. 在单板端，进行如下操作：

```
vi /etc/inittab
#::respawn:/sbin/getty -L ttyS000 115200 vt100 -n root -I "Auto login as
root ..."
::respawn:/sbin/getty -L ttyGS0 115200 vt100 -n root -I "Auto login as
root ..."
```

步骤 5. 通过 USB 数据线将单板与 Host pc 端相连，pc 端会自动加载驱动，第一次可能会失败，需要自行安装驱动，方法为：

- 右击计算机，进入管理界面；
- 打开设备管理器；
- 点击端口（COM 和 LPT），会看到 Gadget Serial(COMx)，双击；
- 点击网络适配器 会看到 Linux USB Ethernet /RNDIS Gadget #x 双击；
- 打开驱动程序界面，点击更新驱动程序，进入浏览计算机以查找驱动程序软件(R)；
- 把路径指向 linux-cdc-acm.inf 和 linux.inf 所在的目录，点击下一步，计算机会自动进行安装驱动程序，安装成功后，关闭界面。

步骤 6. 在单板端配置 IP，命令为 ifconfig usb0 xx.xx.xx.xx netmask 255.255.xxx.0;route add default gw xx.xx.xx.xx。

步骤 7. 当单板和 pc 通过 USB 数据线相连时，会在 PC 端生成 USB 网络节点，具体位置：打开网络和共享中心→更改适配器设置→Linux USB Ethernet /RNDIS Gadget #x。

步骤 8. 建立网桥：a、右键 Linux USB E thernet /RNDIS Gadget #x 节点，点击添加到桥选项；b、PC 端连接大网的本地连接节点右键，选择添加到桥选项；c、等待网桥建立完成，如图 1-5 所示。

图1-5 成功建立网桥

步骤 9. 此时在单板端，就可以正常访问大网，USB 可作为真正的网口操作使用，至此复合设备可以正常使用。

----结束

1.2.4 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。

- 作为 USB device 网口功能时，在每次重启单板后，请删除以前网桥并重新建立新的网桥。

1.3 SD/MMC/eMMC 卡操作指南

1.3.1 操作准备

SD/MMC/EMMC 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、ext4 或 squashFS，也可以通过本地文件系统再挂载到 NFS。

1.3.2 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、ext4 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/EMMC 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/EMMC 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/EMMC 相关模块
  - mmc_core
  - himci
  - mmc_block

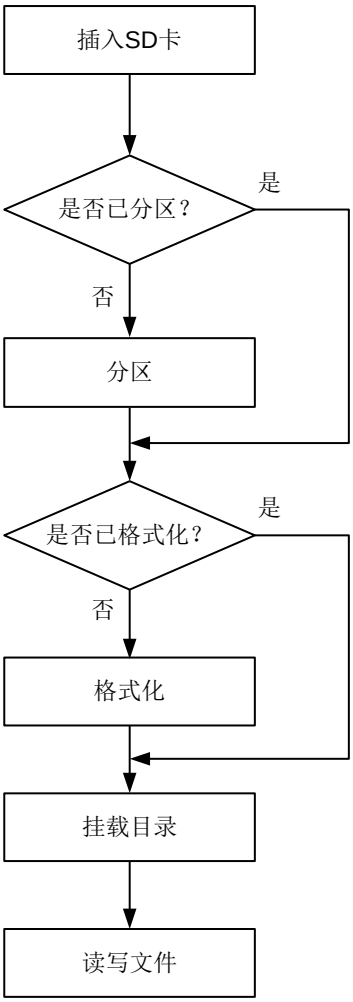
步骤 3. 插入 SD/MMC/EMMC 卡，就可以对 SD/MMC/EMMC 卡进行相关的操作。具体操作请参见“1.3.3 操作示例”。

----结束

1.3.3 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，而 eMMC 卡不支持热插拔，其读写操作也类似，这里不再举例。在控制台下实现读写 SD 卡的操作示例如图 1-6 所示。

图1-6 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/EMMC 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“1.7.1 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“1.7.2 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“1.7.3 挂载目录”。

步骤 4. 对 SD/MMC/EMMC 卡进行读写操作，请参见“1.7.4 读写文件”。

----结束

1.3.4 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“1.3.3 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。

1.4 I2C 操作指南

1.4.1 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、ext4 或 squashFS，也可以通过本地文件系统再挂载到 NFS。

1.4.2 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、ext4 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“1.4.4 操作示例”。

----结束

1.4.3 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 arch/arm/boot/dts/hi3519av100.dtsi，并重新编译内核。具体操作如下：

i2c_bus0 节点中的 clock-frequency 属性的值，如图 1-7 所示。

图1-7 接口速率配置示意图

```
i2c_bus0: i2c@04560000 {
    compatible = "hisilicon,hibvt-i2c";
    reg = <0x04560000 0x1000>;
    clocks = <&clock HI3519AV100_I2C0_CLK>;
    clock-frequency = <100000>;
    status = "disabled";
};
```

1.4.4 操作示例

1.4.4.1 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。

- 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 0 上、设备地址为 0x72 设备的 0x8 寄存器：

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

说明

i2c_num：I2C 控制器序号（对应《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 I2C 控制器编号）

device_addr：外围设备地址（Hi3519AV100 支持标准地址（7bit）和扩展地址（10bit））

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（Hi3519AV100 支持 8/16/24/32bit）

data_width：外围设备的数据位宽（Hi3519AV100 支持 8/16/24/32bit）

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

- 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 0 上、设备地址为 0x72 设备的 0x8 寄存器写入数据 0xa5：

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

说明

i2c_num：I2C 控制器编号（对应《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 I2C 控制器编号）

device_addr：外围设备地址（Hi3519AV100 支持标准地址（7bit）和扩展地址（10bit））

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（Hi3519AV100 支持 8/16/24/32bit）

data_width：外围设备的数据位宽（Hi3519AV100 支持 8/16/24/32bit）

1.4.4.2 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(0);
```

说明

假设我们已经知道新增的器件挂载在 I2C 控制器 0 上，直接设置 i2c_get_adapter 的参数为 0。

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. 在非中断上下文中，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_transfer(client->adapter, msg, 2);
```

在中断上下文中，调用 I2C 驱动层的读写函数对外围器件进行读写：

```
ret = hi_i2c_master_send(client, buf, count);
ret = hi_i2c_transfer (client->adapter, msg, 2);
```

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。
- 参数 msg 为读操作时的两个 i2c_msg 的首地址。

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。

//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", 0x39),
};
```

说明

在代码调用中，器件地址不能包含读写位，但命令行中需要包含读写位。

```
static struct i2c_client *hi_client = NULL;

int hi_i2c_write(unsigned int reg_addr, unsigned int reg_addr_num,
                    unsigned int data, unsigned int data_byte_num)
{
    unsigned char buf[8];
    int ret = 0;
    int idx = 0;
    struct i2c_client *client;
```

```
    client = hi_client;

    /* reg_addr config */
    if (reg_addr_num == 2)
        buf[idx++]  = (reg_addr >> 8);
    buf[idx++] = reg_addr;

    /* data config */
    if (data_byte_num == 2)
        buf[idx++] = data >> 8;
    buf[idx++] = data;
```

//在非中断上下文中，调用内核提供的 I2C 标准写函数进行写操作

```
    ret = i2c_master_send(client, buf, idx);
```

//在中断上下文中，调用驱动层提供的写函数进行写操作

```
    ret = hi_i2c_master_send(client, buf, idx);
    return ret;
}
static struct i2c_msg g_msg[2];

int hi_i2c_read(unsigned int reg_addr, unsigned int reg_addr_num,
        unsigned int data_byte_num)
{
    unsigned char buf[4];
    int ret = 0;
    int ret_data = 0xFF;
    int idx = 0;
    struct i2c_client *client;
    struct i2c_msg *msg;

    client = hi_client;

    msg = &g_msg[0];

    memset(msg, 0x0, sizeof(struct i2c_msg) * 2);

    msg[0].addr = hi_client->addr;
    msg[0].flags = client->flags & I2C_M_TEN;
    msg[0].len = reg_addr_num;
    msg[0].buf = buf;

    /* reg_addr config */
    if (reg_addr_num == 2)
```

```
        buf[idx++] = reg_addr >> 8;
    buf[idx++] = reg_addr;

    msg[1].addr = hi_client->addr;
    msg[1].flags = client->flags & I2C_M_TEN;
    msg[1].flags |= I2C_M_RD;
    msg[1].len = data_byte_num;
    msg[1].buf = buf;
```

//sensor 的读操作一般需要先写一个寄存器地址，才能读出指定寄存器的数据。

//在非中断上下文中，需要调内核提供的 i2c_transfer 传两个 msg 进行先写后读。

```
    ret = i2c_transfer(client->adapter, msg, 2);
```

//在中断上下文中，需要调内核提供的 hi_i2c_transfer 传两个 msg 进行先写后读。

```
    ret = hi_i2c_transfer(client->adapter, msg, 2);
    if (ret == 2) {
        if (data_byte_num == 2)
            ret_data = buf[1] | (buf[0] << 8);
        else
            ret_data = buf[0];
    } else
        ret_data = -EIO;

    return ret_data;
}
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

1.4.4.3 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-0", O_RDWR);
```

步骤 2. 进行数据读写：

```
ioctl(fd, I2C_RDWR, &rdwr);
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 dev_addr, unsigned int
reg_addr,
        unsigned int reg_addr_end, unsigned int reg_width,
        unsigned int data_width, unsigned int reg_step)
{
    int retval = 0;
    int fd = -1;
    char file_name[0x10];
    unsigned char buf[4];
    int cur_addr;
    static struct i2c_rdwr_ioctl_data rdwr;
    static struct i2c_msg msg[2];
    unsigned int data;

    memset(buf, 0x0, 4);

   printf("i2c_num:0x%x, dev_addr:0x%x; reg_addr:0x%x; reg_addr_end:0x%x;
reg_width: %d; data_width: %d. \n\n",
           i2c_num, dev_addr << 1, reg_addr, reg_addr_end, reg_width,
data_width);

    sprintf(file_name, "/dev/i2c-%u", i2c_num);
    fd = open(file_name, O_RDWR);
    if (fd < 0) {
        printf("Open %s error!\n",file_name);
        return -1;
```

```
    }

    retval = ioctl(fd, I2C_SLAVE_FORCE, dev_addr);
    if (retval < 0) {
        printf("CMD_SET_I2C_SLAVE error!\n");
        retval = -1;
        goto end;
    }
    msg[0].addr = dev_addr;
    msg[0].flags = 0;
    msg[0].len = reg_width;
    msg[0].buf = buf;
```

说明

在代码调用中，器件地址不能包含读写位，但命令行中需要包含读写位。

```
    msg[1].addr = dev_addr;
    msg[1].flags = 0;
    msg[1].flags |= I2C_M_RD;
    msg[1].len = data_width;
    msg[1].buf = buf;

    rdwr.msgs = &msg[0];
    rdwr.nmsgs = (__u32)2;
    for (cur_addr = reg_addr; cur_addr <= reg_addr_end; cur_addr +=
reg_step) {
        if (reg_width == 2) {
            buf[0] = (cur_addr >> 8) & 0xff;
            buf[1] = cur_addr & 0xff;
        } else
            buf[0] = cur_addr & 0xff;

        retval = ioctl(fd, I2C_RDWR, &rdwr);
        if (retval != 2) {
            printf("CMD_I2C_READ error!\n");
            retval = -1;
            goto end;
        }

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

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

```
    retval = 0;

end:
    close(fd);
    return retval;
}

int i2c_write(unsigned int i2c_num, unsigned int dev_addr,
        unsigned int reg_addr, unsigned int data,
        unsigned int reg_width, unsigned int data_width)
{
    int retval = 0;
    int fd = -1;
    int index = 0;
    char file_name[0x10];
    unsigned char buf[4];

    printf("i2c_num:0x%x, dev_addr:0x%x; reg_addr:0x%x; data:0x%x;
reg_width: %d; data_width: %d.\n",
           i2c_num, dev_addr << 1, reg_addr, data, reg_width, data_width);

    sprintf(file_name, "/dev/i2c-%u", i2c_num);
    fd = open(file_name, O_RDWR);
    if (fd<0) {
        printf("Open %s error!\n", file_name);
        return -1;
    }

    retval = ioctl(fd, I2C_SLAVE_FORCE, dev_addr);
    if(retval < 0) {
        printf("set i2c device address error!\n");
        retval = -1;
        goto end;
    }

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

```
    }

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

    retval = write(fd, buf, (reg_width + data_width));
    if(retval < 0) {
        printf("i2c write error!\n");
        retval = -1;
        goto end;
    }

    retval = 0;

end:
    close(fd);
    return retval;
}
```

----结束

1.5 SPI 操作指南

1.5.1 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、ext4 或 squashFS，也可以通过本地文件系统再挂载到 NFS。

1.5.2 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、ext4 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 1.5.3 “操作示例”。

----结束

1.5.3 操作示例

1.5.3.1 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《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 SPI 控制器编号）

csn：片选号（对应《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 SPI 控制器片选数，比如：控制器有 2 个片选，则片选号为 0、1）

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 8/16bit）

reg_width：外围设备寄存器地址位宽（支持 8/16bit）

data_width：外围设备的数据位宽（支持 8/16bit）

- 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《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 SPI 控制器编号）

csn：片选号（对应《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 SPI 控制器片选数，比如：控制器有 2 个片选，则对应片选号为 0、1）

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 8/16bit）

reg_width：外围设备寄存器地址位宽（支持 8/16bit）

data_width：外围设备的数据位宽（支持 8/16bit）

说明

此 SPI 读写命令仅支持 sensor 的读写操作。

----结束

1.5.3.2 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;
```

//初始化并设置 SPI 传输队列

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

说明

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;
```

说明

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

1.5.3.3 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《Hi3519AV100 4K Smart 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_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] = 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

1.6 GPIO 操作指南

1.6.1 操作准备

GPIO 的操作准备如下：

- Linux 内核使用 SDK 发布的 kernel。
- 文件系统。

说明
可以使用 SDK 发布的本地文件系统 yaffs2、jffs2、Ext4 或 SquashFS，也可以通过本地文件系统再挂载到 NFS。

1.6.2 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、Ext4 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 GPIO 相关模块已全部编入内核，不需要再执行加载命令。

步骤 3. 配置管脚复用。

步骤 4. 在控制台下运行相关命令或者自行在内核态或者用户态下编写 GPIO 操作程序，就可以对 GPIO 进行输入输出操作。具体操作请参见 1.6.3 “操作示例”。

说明
操作 GPIO 前，对应的管脚需要复用到 GPIO 功能。

----结束

1.6.3 操作示例

1.6.3.1 GPIO 操作命令示例

此操作示例通过命令实现对 GPIO 的读写操作。

步骤 1. 在控制台使用 echo 命令将要操作的 GPIO 编号 export：

```
echo N > /sys/class/gpio/export
```

说明
每组 GPIO 有 8 个 GPIO 管脚。

N 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 4 * 8 + 2 = 34。

export 之后就会生成/sys/class/gpio/gpioN 目录

例如：exportGPIO4_2：

```
echo 34 > /sys/class/gpio/export
```

步骤 2. 在控制台使用 echo 命令设置 GPIO 方向：

- 对于输入：`echo in > /sys/class/gpio/gpioN/direction`
- 对于输出：`echo out > /sys/class/gpio/gpioN/direction`

例如：设置 GPIO4_2 方向

- 对于输入：`echo in > /sys/class/gpio/gpio34/direction`

- 对于输出：`echo out > /sys/class/gpio/gpio34/direction`

说明

- GPIO 方向只有 out 和 in 两种。
- 可使用 cat 命令查看 GPIO 方向：cat /sys/class/gpio/gpioN/direction，例如查看 GPIO4_2 方向：cat /sys/class/gpio/gpio34/direction

步骤 3. 在控制台使用 cat 或 echo 命令查看 GPIO 输入值或设置 GPIO 输出值：

查看输入值：`cat /sys/class/gpio/gpioN/value`

输出低：`echo 0 > /sys/class/gpio/gpioN/value`

输出高：`echo 1 > /sys/class/gpio/gpioN/value`

说明

GPIO 的电平值只有 0 和 1。0 为低电平；1 为高电平。

步骤 4. 在控制台使用 echo 命令将操作的 GPIO 编号 unexport：

```
echo N > /sys/class/gpio/unexport
```

----结束

1.6.3.2 内核态 GPIO 操作示例

此操作示例在内核态下实现对 GPIO 的读写操以及 GPIO 中断操作。

内核态 GPIO 读写操作示例

操作示例如下：

步骤 1. 注册 GPIO：

```
gpio_request(gpio_num, NULL);
```

说明

- 每组 GPIO 有 8 个 GPIO 管脚。
- 参数 gpio_num 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 4 * 8 + 2 = 34

步骤 2. 设置 GPIO 方向：

对于输入：`gpio_direction_input(gpio_num)`

对于输出：`gpio_direction_output(gpio_num, gpio_out_val)`

说明

如果是输出，需要设置一个输出的初始值，参数 gpio_out_val 为输出时的初始值。

步骤 3. 查看 GPIO 输入值或设置 GPIO 输出值：

查看输入值：`gpio_get_value(gpio_num);`

输出低：`gpio_set_value(gpio_num, 0);`

输出高：`gpio_set_value(gpio_num, 1);`

说明
输入值为 gpio_get_value(gpio_num)的返回值。

步骤 4. 释放注册的 GPIO 编号：

```
gpio_free(gpio_num);
```

----结束

内核态 GPIO 中断操作示例

注意

默认使用内核标准 GPIO。若客户自己实现了 GPIO 的驱动及中断注册，而非使用内核标准 GPIO，则需关闭内核标准 GPIO。否则两套驱动之间会产生冲突，影响使用。

关闭内核标准 GPIO 方法如下：

- 打开 dts 文件：./arch/arm/boot/dts/hi3519av100.dts，将相应的 gpio 状态从“okay”改成“disabled”，如图 1-8 所示。

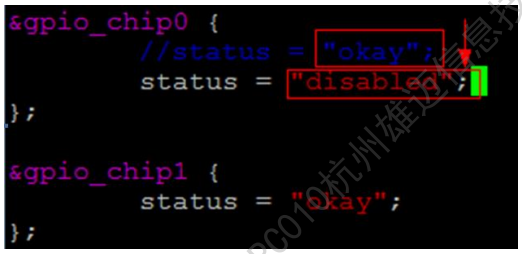
  vi ./arch/arm/boot/dts/hi3519av100.dts
- 修改后保存退出，重新编译内核。

  cp ./arch/arm/configs/hi3519av100_xxx_defconfig .config

  make ARCH=arm CROSS_COMPILE=arm-himix100-linux - menuconfig

  make ARCH=arm CROSS_COMPILE=arm-himix100-linux- uImage
- Hi3519AV100 和 Hi3556AV100 对应的 dts 文件和配置文件不同，使用中请选择对应的 dts 文件及配置文件。

图1-8 关闭内核标准 GPIO 示例



步骤 1. 注册 GPIO：

```
gpio_request(gpio_num, NULL);
```

说明
- 每组 GPIO 有 8 个 GPIO 管脚。
- 参数 gpio_num 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 4 * 8 + 2 = 34。

步骤 2. 设置 GPIO 方向：

```
gpio_direction_input(gpio_num)
```

说明

对于要作为中断源的 GPIO 引脚，方向必须配置为输入。

步骤 3. 映射操作的 GPIO 编号对应的中断号：

```
irq_num = gpio_to_irq(gpio_num);
```

说明

中断号为 gpio_to_irq(gpio_num)的返回值。

步骤 4. 注册中断：

```
request_irq(irq_num, gpio_dev_test_isr, irqflags, "gpio_dev_test",
&gpio_irq_type))
```

说明

Irqflags 为需要注册的中断类型，常用类型有：

- IRQF_SHARED ：共享中断；
- IRQF_TRIGGER_RISING ：上升沿触发；
- IRQF_TRIGGER_FALLING ：下降沿触发；
- IRQF_TRIGGER_HIGH ：高电平触发；
- IRQF_TRIGGER_LOW ：低电平触发。

步骤 5. 结束时释放注册的中断和 GPIO 编号：

```
free_irq(gpio_to_irq(gpio_num), &gpio_irq_type);
gpio_free(gpio_num);
```

代码示例如下：

说明

此代码为 GPIO 操作的示例程序，仅为客户开发内核态的 GPIO 操作程序提供参考，不提供实际应用功能。

```
#include <linux/delay.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/module.h>
//模块参数，GPIO组号、组内偏移、方向、输出时的输出初始值
static unsigned int gpio_chip_num = 4;
module_param(gpio_chip_num, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_chip_num, "gpio chip num");

static unsigned int gpio_offset_num = 2;
module_param(gpio_offset_num, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_offset_num, "gpio offset num");

static unsigned int gpio_dir = 1;
module_param(gpio_dir, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_dir, "gpio dir");
```

```
static unsigned int gpio_out_val = 1;
module_param(gpio_out_val, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_out_val, "gpio out val");

//模块参数，中断触发类型
/*
 * 0 - disable irq
 * 1 - rising edge triggered
 * 2 - falling edge triggered
 * 3 - rising and falling edge triggered
 * 4 - high level triggered
 * 8 - low level triggered
 */
static unsigned int gpio_irq_type = 0;
module_param(gpio_irq_type, uint, S_IRUGO);
MODULE_PARM_DESC(gpio_irq_type, "gpio irq type");

spinlock_t lock;

static int gpio_dev_test_in(unsigned int gpio_num)
{
        //设置方向为输入
       if (gpio_direction_input(gpio_num)) {
              pr_err("[%s %d]gpio_direction_input fail!\n",
                           __func__, __LINE__);
              return -EIO;
       }
        //读出GPIO输入值
       pr_info ("[%s %d]gpio%d_%d in %d\n", __func__, __LINE__,
                    gpio_num / 8, gpio_num % 8,
                    gpio_get_value(gpio_num));

       return 0;
}
//中断处理函数
static irqreturn_t gpio_dev_test_isr(int irq, void *dev_id)
{
       pr_info("[%s %d]\n", __func__, __LINE__);

       return IRQ_HANDLED;
}
```

```
static int gpio_dev_test_irq(unsigned int gpio_num)
{
	unsigned int irq_num;
	unsigned int irqflags = 0;
	 //设置方向为输入
	if (gpio_direction_input(gpio_num)) {
		pr_err("[%s %d]gpio_direction_input fail!\n",
				__func__, __LINE__);
		return -EIO;
	}

	switch (gpio_irq_type) {
		case 1:
			irqflags = IRQF_TRIGGER_RISING;
			break;
		case 2:
			irqflags = IRQF_TRIGGER_FALLING;
			break;
		case 3:
			irqflags = IRQF_TRIGGER_RISING |
				IRQF_TRIGGER_FALLING;
			break;
		case 4:
			irqflags = IRQF_TRIGGER_HIGH;
			break;
		case 8:
			irqflags = IRQF_TRIGGER_LOW;
			break;
		default:
			pr_info("[%s %d]gpio_irq_type error!\n",
					__func__, __LINE__);
			return -1;
	}

	pr_info("[%s %d]gpio_irq_type = %d\n", __func__, __LINE__,
gpio_irq_type);
	irqflags |= IRQF_SHARED;
	 //根据GPIO编号映射中断号
	irq_num = gpio_to_irq(gpio_num);
	 //注册中断
	if (request_irq(irq_num, gpio_dev_test_isr, irqflags,
				"gpio_dev_test", &gpio_irq_type)) {
		pr_info("[%s %d]request_irq error!\n", __func__, __LINE__);
```

```
                return -1;
        }

        return 0;
}

static void gpio_dev_test_irq_exit(unsigned int gpio_num)
{
        unsigned long flags;

        pr_info("[%s %d]\n", __func__, __LINE__);
         //释放注册的中断
        spin_lock_irqsave(&lock, flags);
        free_irq(gpio_to_irq(gpio_num), &gpio_irq_type);
        spin_unlock_irqrestore(&lock, flags);
}
static int gpio_dev_test_out(unsigned int gpio_num, unsigned int
gpio_out_val)
{
         //设置方向为输出，并输出一个初始值
        if (gpio_direction_output(gpio_num, !!gpio_out_val)) {
                pr_err("[%s %d]gpio_direction_output fail!\n",
                                __func__, __LINE__);
                return -EIO;
        }

        pr_info("[%s %d]gpio%d_%d out %d\n", __func__, __LINE__,
                                gpio_num / 8, gpio_num % 8, !!gpio_out_val);
        return 0;
}

static int __init gpio_dev_test_init(void)
{
        unsigned int gpio_num;
        int status = 0;

        pr_info("[%s %d]\n", __func__, __LINE__);

        spin_lock_init(&lock);

        gpio_num = gpio_chip_num * 8 + gpio_offset_num;
         //注册要操作的GPIO编号
        if (gpio_request(gpio_num, NULL)) {
```

```
                pr_err("[%s %d]gpio_request fail! gpio_num=%d \n", __func__,
__LINE__, gpio_num);
                return -EIO;
        }

        if (gpio_dir) {
                status = gpio_dev_test_out(gpio_num, gpio_out_val);
        } else {
                if (gpio_irq_type)
                        status = gpio_dev_test_irq(gpio_num);
                else
                        status = gpio_dev_test_in(gpio_num);
        }

        if (status)
                gpio_free(gpio_num);

        return status;
}
module_init(gpio_dev_test_init);

static void __exit gpio_dev_test_exit(void)
{
        unsigned int gpio_num;

        pr_info("[%s %d]\n", __func__, __LINE__);

        gpio_num = gpio_chip_num * 8 + gpio_offset_num;

        if (gpio_irq_type)
                gpio_dev_test_irq_exit(gpio_num);
         //释放注册的GPIO编号
        gpio_free(gpio_num);
}

module_exit(gpio_dev_test_exit);

MODULE_DESCRIPTION("GPIO device test Driver sample");
MODULE_LICENSE("GPL");
```

----结束

1.6.3.3 用户态 GPIO 操作程序示例

此操作示例在用户态下实现对 GPIO 的读写操作。

步骤 1. 将要操作的 GPIO 编号 export：

```
fp = fopen("/sys/class/gpio/export", "w");
fprintf(fp, "%d", gpio_num);
fclose(fp);
```

说明

- 每组 GPIO 有 8 个 GPIO 管脚。
- 参数 gpio_num 为要操作的 GPIO 编号，该编号等于 GPIO 组号 * 8 + 组内偏移号，例如 GPIO4_2 的编号为 4 * 8 + 2 = 34。

步骤 2. 设置 GPIO 方向：

```
fp = fopen("/sys/class/gpio/gpio%d/direction", "rb+");
```

对于输入：`fprintf(fp, "in");`

对于输出：`fprintf(fp, "out");`

```
fclose(fp);
```

步骤 3. 查看 GPIO 输入值或设置 GPIO 输出值：

```
fp = fopen("/sys/class/gpio/gpio%d/value", "rb+");
```

查看输入值：`fread(buf, sizeof(char), sizeof(buf) - 1, fp);`

输出低：

```
strcpy(buf,"0");
fwrite(buf, sizeof(char), sizeof(buf) - 1, fp);
```

输出高：

```
strcpy(buf,"1");
fwrite(buf, sizeof(char), sizeof(buf) - 1, fp);
```

步骤 4. 将操作的 GPIO 编号 unexport：

```
fp = fopen("/sys/class/gpio/unexport", "w");
fprintf(fp, "%d", gpio_num);
fclose(fp);
```

代码示例如下：

说明

此代码为 GPIO 操作示例程序，仅为客户开发用户态的 GPIO 操作程序提供参考，不提供实际应用功能。

```
#include <stdio.h>
#include <string.h>
```

```
int gpio_test_in(unsigned int gpio_chip_num, unsigned int gpio_offset_num)
{
        FILE *fp;
        char file_name[50];
        unsigned char buf[10];
        unsigned int gpio_num;

        gpio_num = gpio_chip_num * 8 + gpio_offset_num;

        sprintf(file_name, "/sys/class/gpio/export");
        fp = fopen(file_name, "w");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
        fprintf(fp, "%d", gpio_num);
        fclose(fp);

        sprintf(file_name, "/sys/class/gpio/gpio%d/direction", gpio_num);
        fp = fopen(file_name, "rb+");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
        fprintf(fp, "in");
        fclose(fp);

        sprintf(file_name, "/sys/class/gpio/gpio%d/value", gpio_num);
        fp = fopen(file_name, "rb+");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
        memset(buf, 0, 10);
        fread(buf, sizeof(char), sizeof(buf) - 1, fp);
        printf("%s: gpio%d_%d = %d\n", __func__,
                        gpio_chip_num, gpio_offset_num, buf[0]-48);
        fclose(fp);
        sprintf(file_name, "/sys/class/gpio/unexport");
        fp = fopen(file_name, "w");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
```

```
        fprintf(fp, "%d", gpio_num);
        fclose(fp);

        return (int)(buf[0]-48);
}

int gpio_test_out(unsigned int gpio_chip_num, unsigned int
gpio_offset_num,
                unsigned int gpio_out_val)
{
        FILE *fp;
        char file_name[50];
        unsigned char buf[10];
        unsigned int gpio_num;

        gpio_num = gpio_chip_num * 8 + gpio_offset_num;

        sprintf(file_name, "/sys/class/gpio/export");
        fp = fopen(file_name, "w");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
        fprintf(fp, "%d", gpio_num);
        fclose(fp);

        sprintf(file_name, "/sys/class/gpio/gpio%d/direction", gpio_num);
        fp = fopen(file_name, "rb+");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
        fprintf(fp, "out");
        fclose(fp);

        sprintf(file_name, "/sys/class/gpio/gpio%d/value", gpio_num);
        fp = fopen(file_name, "rb+");
        if (fp == NULL) {
                printf("Cannot open %s.\n", file_name);
                return -1;
        }
        if (gpio_out_val)
                strcpy(buf,"1");
        else
```

```
        strcpy(buf,"0");

    fwrite(buf, sizeof(char), sizeof(buf) - 1, fp);
    printf("%s: gpio%d_%d = %s\n", __func__,
                gpio_chip_num, gpio_offset_num, buf);
    fclose(fp);

    sprintf(file_name, "/sys/class/gpio/unexport");
    fp = fopen(file_name, "w");
    if (fp == NULL) {
        printf("Cannot open %s.\n", file_name);
        return -1;
    }
    fprintf(fp, "%d", gpio_num);
    fclose(fp);

    return 0;

}
```

----结束

1.7 附录

1.7.1 用 fdisk 工具分区

通过“1.7.1.1 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“1.7.2 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。

1.7.1.1 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“1.7.1.2 创建新的分区”和“1.7.1.3 保存分区信息”的描述对设备进行分区。

1.7.1.2 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

1.7.1.3 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.
.....
~ $
```

1.7.2 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“1.7.3 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：

~ $

1.7.3 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。

1.7.4 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
```

2 Huawei LiteOS

说明
Hi3519AV100 SMP 版本不支持该章节内容。

2.1 I²C 操作指南

2.1.1 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。

2.1.2 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 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。

2.1.3 使用示例

2.1.3.1 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《Hi3519AV100_PINOUT_CN》中管脚控制寄存器页签。

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 2.2.3.2 及 2.2.3.3 。

----结束

2.1.3.2 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. 在非中断上下文中，调用 I²C 提供的标准读写函数对外围器件进行读写：

```
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);
```

在中断上下文中，调用 I2C 驱动层的读写函数对外围器件进行读写：

```
读：ret = hi_i2c_transfer(struct i2c_adapter *adapter, struct i2c_msg
*msgs, int count);
写：ret = hi_i2c_master_send(struct i2c_client *client, const char *buf,
int 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。
- 参数 buf 为需要读写的寄存器和数据。
- 参数 count 为 buf 的长度。
- 参数 msg 为读操作时的两个 i2c_msg 的首地址。

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;
    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;
}
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;
}
```

----结束

2.1.3.3 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/ i2c.h 头文件中。

步骤 3. 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，使用 ioctl 进行读写的相关宏定义在 drivers/i2c/include/i2c-dev.h 头文件中。

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。

----结束

2.1.4 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《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 I²C 控制器编号）。

- 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（支持 8/16bit，2: 16bit/1:8bit）。
- data_width：外围设备的数据位宽（支持 8/16bit，2: 16bit/1:8bit）。
- addr_width：外围设备地址位宽（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《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 I²C 控制器编号）。

- 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（I²C 控制器支持 8/16bit）。
- data_width：外围设备的数据位宽（I²C 控制器支持 8/16bit）。

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

2.1.5 API 参考

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

参数名称	描述	输入/输出
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

返回值	描述
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
其它	操作失败

2.1.6 数据类型

i2c_client：I²C 从设备结构体。

i2c_client

【说明】

I²C 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
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

2.2 SPI 操作指南

2.2.1 功能介绍

SPI 模块的作用是访问 SPI 总线上连接的从设备。

2.2.2 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 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。

2.2.3 使用示例

2.2.3.1 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《Hi3519AV100_PINOUT_CN》中管脚控制寄存器页签。

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 2.2.3.2 及 2.2.3.3 。

----结束

2.2.3.2 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

2.2.3.3 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《Hi3519AV100 4K Smart 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

2.2.4 shell 命令

2.2.4.1 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《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 SPI 控制器编号）。
- csn：片选号
- 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。

2.2.4.2 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《Hi3519AV100 4K Smart IP Camera SoC 用户指南》中的 SPI 控制器编号）。
- csn：片选号。
- 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。

2.2.5 API 参考

spi_dev_set：用于发起对 SPI 设备的读写操作。

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 2.2.6 。	输入

【返回值】

返回值	描述
非负值	读写长度
负值	读写失败

2.2.6 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
rx_buf	读缓冲区地址
len	读、写长度
cs_change	片选改变标志
speed	速率

2.3 UART 操作指南

2.3.1 功能介绍

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

2.3.2 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 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。

2.3.3 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 UART 驱动注册。

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《Hi3519AV100_PINOUT_CN》中管脚控制寄存器页签。

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

2.3.4 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 控制器编号）。
- 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 2-1 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。

2.3.5 API 参考

- uart_dev_init：UART 设备初始化。
- uart_suspend：UART 设备挂起。
- uart_resume：UART 设备唤醒。

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

2.3.5.1 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 2-1 所示。

表2-1 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_PRIV	0x110	自定义	驱动自定命令

命令号	命令码	参数	说明
ATE			

2.3.5.2 举例

例程请参考文件：

```
./drivers/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。

2.4 GPIO 操作指南

2.4.1 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。

2.4.2 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 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。

2.4.3 使用示例

2.4.3.1 模块初始化

在对 GPIO 操作之前需要调用初始化函数：

```
gpio_dev_init();
```

2.4.3.2 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

2.4.4 API 参考

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