Peripheral Driver

Operation Guide

Issue **04**

Date **2016-11-18**

HiSilicon Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: http://www.hisilicon.com

Email: support@hisilicon.com

About This Document

Purpose

This document describes how to manage the peripherals connecting to the modules that have the gigabit media access control (GMAC), universal serial bus (USB) 2.0 Host/Device, and SD/MMC driver installed. It covers the following topics, namely, preparations, operation procedures, precautions to be taken during operation, and operation instances.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3516A	V100
Hi3516D	V100

Intended Audience

This document is intended for:

- Technical support engineers
- Software development engineers

Change History

Changes between document issues are cumulative. Therefore, the latest document issue contains all changes made in previous issues.

Issue 04 (2016-11-18)

This issue is the forth official release, which incorporates the following changes:

Chapter 4 Operation Guide to the I^2C

Section 4.4.2 and section 4.4.3 are modified.

Issue 03 (2015-06-04)

This issue is the third official release, which incorporates the following changes:

Chapter 1 Operation Guide to the GMAC Module

Section 1.4 and section 1.5 are added.

Issue 02 (2015-02-10)

This issue is the second official release, which incorporates the following changes:

Chapter 4 Operation Guide to the I^2C

Section 4.3 is added.

Chapter 5 Operation Guide to the SPI

The descriptions in sections 5.3.2 and 5.3.3 are updated.

Issue 01 (2014-12-20)

This issue is the first official release, which incorporates the following changes:

The contents related to the Hi3516D are added.

Issue 00B04 (2014-11-10)

This issue is the fourth draft release, which incorporates the following changes:

Chapter 5 Operation Guide to the SPI

Chapter 5 "Operation Guide to the SPI" is added.

Issue 00B03 (2014-10-19)

This issue is the third draft release, which incorporates the following changes:

Chapter 2 Operation Guide to the USB 2.0 Host/Device Module

The description in sections 2.2 and 2.3.4 is modified.

Chapter 4 Operation Guide to the I^2C

Chapter 4 is added.

Issue 00B02 (2014-09-14)

This issue is the second draft release.

Issue 00B01 (2014-07-25)

This issue is the first draft release.

Contents

Figures

1 Operation Guide to the GMAC Module

NOTE

- Unless otherwise specified, this document applies to the Hi3516A and Hi3516D.
- The following addresses are examples only. You need to configure the addresses as required.

1.1 Operation Instance

NOTE

The GMAC drivers for the Hi3516A are compiled in the kernel by default. Therefore, you can configure IP addresses without loading GMAC drivers.

Note the following when using the network interface under the kernel:

- The gigabit media access control (GMAC) module supports the TCP segmentation offload (TSO) function, and the TSO function is enabled by default. You can use the ethtool to disable this function. To enable or disable the TSO function, run the following commands:
  - Disable TSO: **./ethtool –K eth0 tx off**
  - Enable TSO: **./ethtool –K eth0 tx on**

  Function description of TSO:
  - TSO is a technology to split large data packets and reduce the CPU usage by using the network adapter. If the data packets can only be TCP packets, the technology is called TSO. Otherwise, it is called large segmentation offload (LSO). If the TSO function is supported by the hardware, the TCP checksum calculation and scatter-gather functions are also required. The TSO function is implemented by using the software and hardware. The hardware splits large data packets and attaches headers to each fragment.
  - When using the TSO function, the Hi3516A delivers part of the CPU tasks to the network adapter to reduce the CPU usage and improve performance.
- You can configure the IP address and subnet mask by running the following command:

  ```
  ifconfig eth0 xxx.xxx.xxx.xxx netmask xxx.xxx.xxx.xxx up
  ```

- You can set the default gateway by running the following command:

  ```
  route add default gw xxx.xxx.xxx.xxx
  ```

- You can mount to the NFS by running the following command:

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

- You can upload or download files over TFTP in the shell.

  Ensure that the TFTP service software is running on the server.

  - To download a file, run the **tftp -r XX.file serverip –g** command.

    Where, **XX.file** is the file to be downloaded, and **serverip** is the IP address of the server where the file to be downloaded is located.

  - To upload a file, run the **tftp -l xx.file remoteip -p** command.

    Where, **xx.file** is the file to be uploaded, and **remoteip** is the IP address of the server that the file is uploaded to.

1.2 IPv6

The IPv6 function is disabled in the release package by default. If the IPv6 function is required, you need to modify the kernel options and recompile the kernel as follows:

```
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
```

NOTE

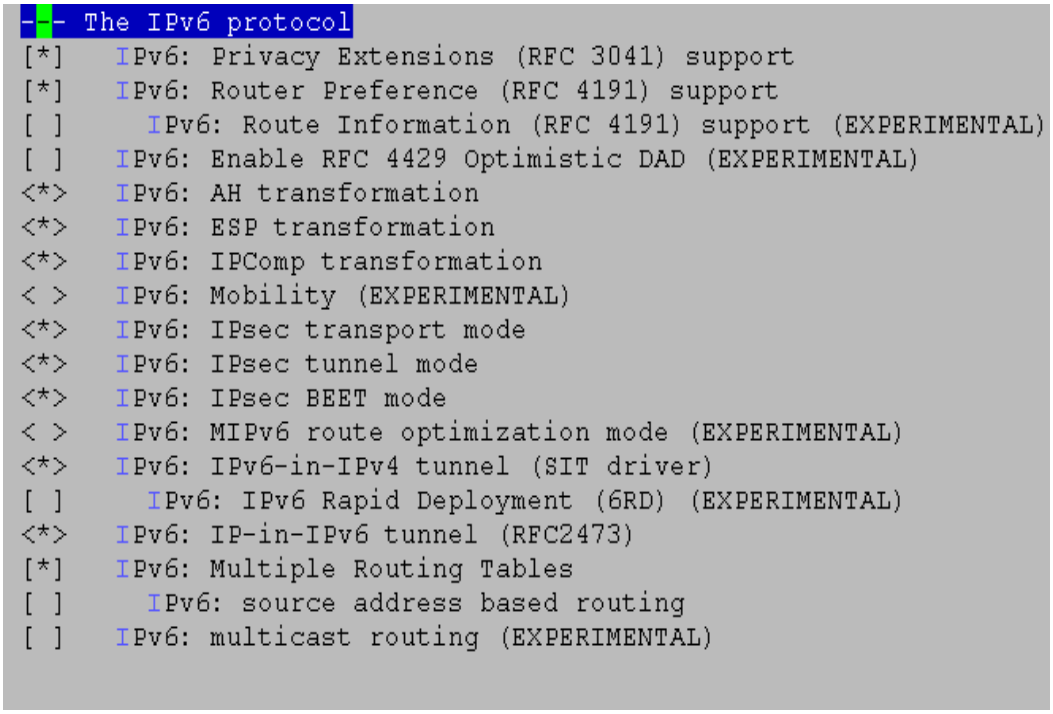
There are two cases for CROSS_COMPILE=arm-hisiXXX-linux-:

- Hi3516A_V100R001C01SPC*xxx* corresponds to uClibc. CROSS_COMPILE=arm-hisiv300-linux- indicates the uClibc tool chain.
- Hi3516A_V100R001C02SPC*xxx* corresponds to glibc. CROSS_COMPILE=arm-hisiv400-linux- indicates the glibc tool chain.

Go to the following directories and configure the options, as shown in Figure 1-1.

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

Figure 1-1 Configuration options of the IPv6 protocol



The IPv6 configurations are as follows:

- Configure the IP address and the default gateway by running the following command:
  `hisilicon$ ip -6 addr add <ipv6address>/<ipv6_prefixlen> dev <port>`

  Example: `ip -6 addr add 2001:da8:207::9402/64 dev eth0`
- Ping a website by running the following command:
  `hisilicon$ ping -6 <ipv6address>`

  Example: `ping -6 2001:da8:207::9403`

1.3 Configuring the PHY Address

On the Hi3516A demo board, the default PHY address is 1. When different PHY addresses are selected, the PHY address configuration must be modified under the U-boot or the kernel.

- Configuring the PHY address under the U-boot.

  To configure the PHY address under the U-boot, change the value of **CONFIG_HIGMAC_PHY1_ADDR** macro definition in the configuration files of U-boot. The U-boot of the Hi3516A contains the following two configuration files, and they must be modified at the same time:

  - **include/configs/hi3516a.h**
  - **include/configs/hi3516a_spinand.h**
- Configuring the PHY address under the kernel.

  To configure the PHY address under the kernel, use the following kernel configuration options:

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

1.4 Configuring the PHY Interface Mode

The Hi3516A GMAC module supports the reduced gigabit media independent interface (RGMII), reduced media independent interface (RMII), and media independent interface (MII) PHY interface modes. The default PHY interface mode in the SDK is RGMII. If you want to set the mode to RMII or MII, you need to modify the configuration under the U-boot and kernel.

- To configure the mode under the U-boot, set the environment variable **setenv mdio_intf** to **rmii** or **mii**.
- To configure the mode under the kernel, use the following kernel configuration options in **menuconfig**.

  ```
  Device Drivers
  [*] Network device support  --->
  [*] Ethernet driver support  --->
  <*> hieth gmac family network device support  --->
  (6) hieth-gmac phy0 interface mode
  ```

  where the values **1**, **5**, and **6** indicate MII, RMII, and RGMII respectively. If the pin multiplexing relationship is changed, you need to reconfigure it.

1.5 Configuring the IEEE 802.3x Flow Control Function

1.5.1 Function Description

The GMAC network supports the flow control function defined in IEEE 802.3x, and it can transmit pause frames as well as receive and process pause frames sent by the peer end.

- Transmitting the pause frame

  At the RX end, when the space of the current descriptor RX queue is insufficient, some of the received data packets may fail to be transmitted to software. In this case, a pause frame is sent to the peer end to instruct the peer end to stop sending packets for a specified period.
- Receiving the pause frame

  When a pause frame is received, the GMAC delays the transmission based on the flow control time field in the frame, and restarts the transmission when the flow control period expires or when the GMAC receives the pause frame with zero flow control time from the peer end during waiting.

1.5.2 Configuring the Flow Control Function by Using the Kernel menuconfig

Six configurable parameters are provided for controlling pause frame transmission and reception. You can set these parameters by running **menuconfig** during kernel compilation. The parameters of the network driver are configured as follows by running **menuconfig**:

Figure 1-2 Configuring parameters by running menuconfig

```
--- 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
```

The configurable flow control parameters are described as follows:

- **rx flow ctrl supported**

  Pause frame RX enable

- **tx flow ctrl supported**

  Pause frame TX enable

- **tx flow ctrl pause time**

  Pause time value carried in the transmitted pause frame. The unit is the time required for transmitting 512 bits. The maximum value and the default value are 0xFFFF.

- **tx flow ctrl pause interval**

  Interval for retransmitting pause frames when the conditions are met. The unit is the time required for transmitting 512 bits. The maximum value is 0xFFFF, and the default value is 0x002F. **tx flow ctrl pause interval** must be less than **tx flow ctrl pause time**.

- **tx flow ctrl active threshold**

  Threshold for activating the process of transmitting pause frames. When the number of available descriptors in the RX queue is less than **tx flow ctrl active threshold**, the process of transmitting pause frames by the logic is activated.

- **tx flow ctrl deactive threshold**

Threshold for deactivating the process of transmitting pause frames. When the number of available descriptors in the RX queue is greater than or equal to **tx flow ctrl deactive threshold** and flow control is enabled, current flow control is disabled.

NOTE

The threshold for activating the process of transmitting pause frames must be less than the threshold for deactivating the process of transmitting pause frames.

1.5.3 Configuring the Flow Control Function by Using the ethtool

You can enable flow control by using the standard ethtool.

You can view the flow control status of the eth0 port by running **ethtool –a eth0**. The following information is displayed:

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

As shown in the preceding displayed information, both RX flow control and TX flow control are enabled.

You can enable or disable TX/RX flow control by running the following commands:

```
# ./ethtool -A eth0 rx off (Disable RX flow control)
# ./ethtool -A eth0 rx on (Enable RX flow control)
# ./ethtool -A eth0 tx off (Disable TX flow control)
# ./ethtool -A eth0 tx on (Enable TX flow control)
```

2 Operation Guide to the USB 2.0 Host/Device Module

NOTE

The USB 2.0 of the Hi3516A supports the host mode and the device mode.

2.1 Preparations

Before using the USB 2.0 host/Device module, ensure that the following items are available:

- U-boot and Linux kernel released in the SDK
- NFS or the local file system Yaffs2, Jffs2, or Cramfs (Jffs2 is recommended)

2.2 Procedure

The operation procedure is as follows:

Step 1 Start the board, and load the NFS or the file system Yaffs2, Jffs2, or Cramfs.

By default, all drivers related to the USB 2.0 module are compiled in the kernel. Therefore, you do not need to load the drivers.

Step 2 Insert a USB device such as the USB flash drive, mouse, or keyboard, and then perform operations on the USB device. For details, see section 2.3 "Operation Instances."

The drivers related to USB are as follows:

- Drivers related to the file system and storage devices
  - vfat
  - scsi_mod
  - sd_mod
  - nls_ascii
  - nls_iso8859-1
- Drivers related to the keyboard
  - evdev

  - usbhid
- Drivers related to the mouse
  - mousedev
  - usbhid
  - evdev
- Drivers related to the USB 2.0 module
  - ohci-hcd
  - ehci-hcd
  - usb-storage
  - hiusb-hi3516A

----End

Procedure of the USB 2.0 Device

Step 1 Start the board, and load the NFS or the file system Yaffs2, Jffs2, or Cramfs.

Step 2 When the board acts as a device, **dwc_otg** and **g_file_storage** need to be loaded so that the board can be identified by the host as the USB storage media. **dwc_otg** and **g_file_storage** are not compiled in the kernel by default. Therefore, you need to load the driver. For details, see section 2.3 "Operation Instances."

The drivers related to the USB 2.0 device are as follows:

- Drivers related to the file system and storage devices, which are the same as those of the host.
- USB 2.0 device drivers
  - **hi_hs_device**
  - **g_file_storage**

----End

2.3 Operation Instances

2.3.1 Operation Instance Related to the USB Flash Drive

Inserting and Detecting a USB Flash Drive

Insert a USB flash drive, and then check whether it can be detected.

If the USB 2.0 host module works properly, the following information is displayed over the serial port:

```
~ # 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, does not
support DPO or FUA
sda: sda1
sd 2:0:0:0: [sda] Attached SCSI removable disk
```

Where, **sda1** is the first partition of the USB flash drive or the portable drive. If there are multiple partitions, information such as sda1, sda2, sda3, …, and sdaN is displayed.

Using the USB Flash Drive

Perform the following initialization steps after the related drivers are loaded:

NOTE

In **sd*XY***, X indicates the disk ID, and Y indicates the partition ID. You need to change them as required.

- The **sd*X*** device node is partitioned by running the partition command such as **$ fdisk /dev/sda**.
- The **sd*XY*** partition is formatted by running the **mkdosfs** command such as **~ $ mkdosfs –F 32 /dev/sda1**.
- The **sd*XY*** partition is mounted by running the mount command such as **~ $ mount -t vfat /dev/sda1 /mnt**.

Step 1 Check whether the USB flash drive is partitioned.

- Run **ls /dev** to view system device files. If **sd*XY*** is not displayed, the USB flash drive is not partitioned. In this case, run the **~ $ fdisk /dev/sda** command to partition it. For details, see section 6.1 "Partitioning a Storage Device.", and then go to Step 2.
- If **sd*XY*** is displayed, the USB flash drive is detected and partitioned. In this case, go to Step 2.

Step 2 Check whether the USB flash drive is formatted.

- If it is not formatted, run the **~ $ mkdosfs –F 32 /dev/sdaX command** to format it. For details, see section 6.2 "Formatting a Partition."
- If it is formatted, go to Step 3.

Step 3 Check whether a directory is mounted.

- If no directory is mounted, run the **~ $ mount -t vfat /dev/sdaX /mnt** command to mount a directory. For details, see section 6.3 "Mounting a Directory."
- If a directory is mounted, go to Step 4.

Step 4 Read/write to the USB flash drive. For details, see section 6.4 "Reading/Writing to a File."

----End

2.3.2 Using the Keyboard

Before you can use the keyboard, you need to perform the following steps:

Step 1 Load the drivers related to the keyboard.

After the drivers related to the keyboard are loaded, the **event0** node is generated in **/dev/**.

Step 2 Receive the keyboard inputs by running the following command:

```
cat  /dev/ event0
```

Step 3 Press any keys on the keyboard.

If no error occurs, the content that you entered is displayed on the screen.

----End

2.3.3 Using the Mouse

Before you can use the mouse, perform the following steps:

Step 1 Load the drivers related to the mouse.

After the drivers related to the mouse are loaded, the **mouse0** node is generated in **/dev/**.

Step 2 Run a standard test program (mev recommended) of the gpm tool.

Step 3 Click randomly on the screen or move the pointer.

If the mouse functions properly, the corresponding code value is displayed.

----End

2.3.4 Using the USB Device

When the board acts as the device, it supports the flash memory and the secure digital (SD) card. To use the USB device, perform the following steps:

Step 1 Load the drivers related to the USB device.

- To use the flash memory as the USB device virtual storage media, run the following command:

  insmod dwc_otg.ko

  insmod g_file_storage.ko file=/dev/mtdblockX luns=1 stall=0 removable=1

  The **mtdblockX** partition is partition X of the flash memory. You can select the partition as required.
- To use the SD card as the USB device virtual storage media, run the following command:

  insmod dwc_otg.ko

  insmod g_file_storage.ko file=/dev/mmcblk0pX luns=1 stall=0 removable=1

  The **mmcblk0pX** partition is partition X of the SD card. You can select the partition as required.

NOTE

dwc_otg.ko is stored in **drivers/usb/gadget/hi_hs_dev/dwc_otg.ko** and **g_file_storage.ko** is stored in **drivers/usb/gadget/g_file_storage.ko** under the kernel.

Step 2 Connect the board to the host over the USB port. The board is identified by the host as the USB storage device, and the corresponding device node is generated in **/dev**.

Step 3 Partition, format, read, and write to the board. The board is identified as a common USB storage device by the host. See 2.3.1 "Operation Instance Related to the USB Flash Drive."

----End

2.4 Precautions

Note the following when performing the operations related to the USB 2.0 module:

- You need to run the **mount** command, operate a file, and then run the **umount** command in sequence each time. This avoids exceptions in the file system.
- The drivers of the keyboard and mouse must work with the upper layer. For example, mouse events are displayed on the graphical user interface (GUI) of the upper layer. You only need to operate the keyboard by accessing the event node in **/dev**. However, standard libraries are required for mouse operations.
- Mouse application libgpm libraries are provided in Linux. If you need to use the mouse, these libraries must be compiled. You are recommended to use the standard kernel interface gpm-1.20.5 that has passed the test.

  In addition, a set of test programs (such as mev) are provided in the gpm tool. You can perform encoding by using the test programs, making the development easier.

  The drivers related to the USB device must be loaded before the board is connected to the host. Otherwise the host cannot identify the USB device, and the misoperation information is displayed repeatedly.

3 Operation Guide to the SD/MMC Card

3.1 Preparations

Before using the SD card or multi-media card (MMC), ensure that the following items are available:

- The U-boot and Linux use the U-boot and kernel released by the SDK
- File systems

You can use the local file system Yaffs2, Jffs2, or SquashFS released in the SDK or mounted to the NFS.

3.2 Procedure

To use the SD card or MMC, perform the following steps:

Step 1 Start the board, and load the local file system Yaffs2, Jffs2, or SquashFS or mount the local file system to the NFS.

Step 2 Load the kernel. By default, all drivers related to the SD card or MMC are compiled in the kernel. Therefore, you do not need to load the drivers. The drivers related to the SD card or MMC are as follows:

- Drivers related to the file system and storage devices
  - nls_base
  - nls_cp437
  - fat
  - vfat
  - msdos
  - nls_iso8859-1
  - nls_ascii
- SD or MMC drivers
  - mmc_core
  - himci
  - mmc_block

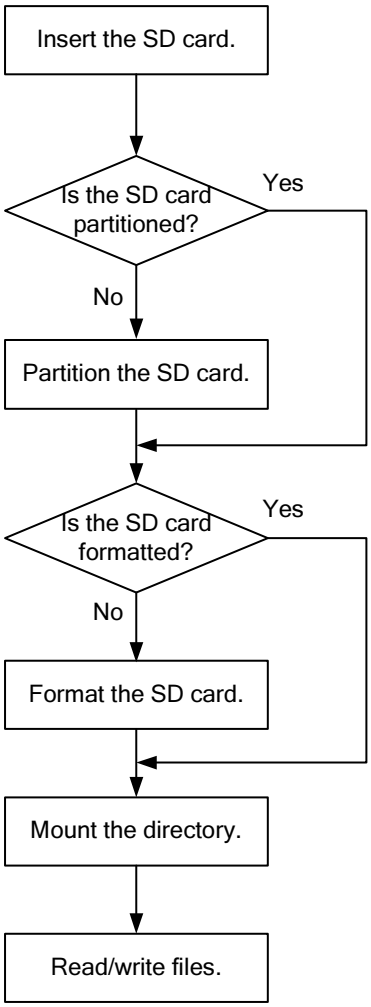
Step 3 Insert the SD card or MMC. Then you can perform operations on the SD card or MMC. For details, see section 3.3 "Operation Instances."

----End

3.3 Operation Instances

The following instance describes how to read/write to an SD card by using the SDIO interface. The procedure for reading/writing to an MMC is similar to that of an SD card and therefore not described in this document. Figure 3-1 shows the read/write flowchart.

Figure 3-1 Read/write flowchart



After initialization and application, and the SD card or MMC is inserted, perform the following steps:

NOTE

In the following commands, **X** indicates the partition number that is assigned when you partition an SD card or MMC by running the **fdisk** command.

- The directory in which the **fdisk** command is executed must be changed to ~ **$ fdisk /dev/mmcblk0**.
- The directory formatted by running the **mkdosfs** command must be changed to ~ **$ mkdosfs –F 32 /dev/mmcblk0pX**.
- The mount directory must be changed to ~ **$ mount -t vfat /dev/mmcblk0pX /mnt**.

Step 1 Check whether the SD card or MMC is partitioned.

- If **p1** is not displayed, the SD card or MMC is not partitioned. In this case, run the ~ **$ fdisk /dev/mmcblk0** command to partition the SD card or MMC. For details, see section 4.1.
- If **p1** is displayed, the SD card or MMC is detected and partitioned. In this case, go to 3.3 Step 2.

Step 2 Check whether the SD card or MMC is formatted.

- If the SD card or MMC is not formatted, format it by following section 6.2 "Formatting a Partition."
- If it is formatted, go to Step 3.

Step 3 Mount the directory by following section 6.3 "Mounting a Directory."

Step 4 Read/write to the SD card or MMC. For details, see section 6.3 "Mounting a Directory."

----End

3.4 Precautions During Operations

Note the following during operation:

- Ensure that the metal sheet of the card is in good contact with the card slot; otherwise, the card fails to be detected or an error occurs in reading/writing to the card. When testing a thin MMC, hold the communication end of the slot if necessary.
- Before reading data from or writing data to the SD card, ensure that the SD card is partitioned and formatted to the vfat file system by running the **fdisk** and **mkdosfs** commands. For details, see 3.3 "Operation Instances."
- You need to run the **mount** command to mount an SD card to a file system so that you can read/write to the SD card. After you remove an SD card, you need to release the mount node by running the **umount** command; otherwise, the partitions of the SD card fail to be detected after the card is inserted again.
- If an SD card is removed, you need to release the mount node by running the **umount** command. You are advised to run the **umount** command, and remove the SD card in sequence. If an SD card is removed incorrectly, you also need to release the mount node by running the **umount** command; otherwise, the partitions of the SD card fail to be detected after it is inserted again.

Do not perform the following operations:

- Do not remove an SD card during read/write. Otherwise, exception information is displayed, and the files in the SD card or the file system may be damaged.
- When the current directory is the mount directory such as **/mnt**, the mount node cannot be released.

- If the process of reading or writing the mount directory in the system does not end, do not run the **umount** command. Wait till the process ends.

Take the following measures if exceptions occur during operation:

- If a card is removed incorrectly during a cyclic test, press **Ctrl**+**C** to roll back to the shell; otherwise, misoperation information is displayed repeatedly
- If you reseat a card rapidly, the card may fail to be detected. This is because the registration and deregistration processes take a period of time.
- If a card is removed incorrectly, run the **umount** command to release the mount node. Otherwise, the mount node directory, **/mnt** for example, cannot be read or written and misoperation information is displayed.
- If an SD card has multiple partitions, you can run the **mount** command to switch to another partition for mounting. Ensure that the times of running the **umount** command are the same as those of running the **mount** command. In this way, all mounted partitions can be released.
- If the file system is damaged due to read/write operations or other exceptions, the file system may be panic after you reseat the card, mount a folder, and then read/write to the card. In this case, run the **umount** command, reseat the SD card, and then run the **mount** command.

4 Operation Guide to the I²C

4.1 Preparations

Before using the I²C, ensure that the following items are available:

- Linux kernel released in the SDK
- File systems

You can use the local file system Yaffs2, Jffs2, or SquashFS released in the SDK or mounted to the NFS.

4.2 Procedure

To use the I²C, perform the following steps:

Step 1 Start the board, and load the local file system Yaffs2, Jffs2, or SquashFS or mount the local file system to the NFS.

Step 2 Load the kernel. By default, all drivers related to the I²C module are compiled in the kernel. Therefore, you do not need to load the drivers.

Step 3 Run I²C read and write commands on the console or compile I²C read and write programs in kernel mode or user mode to read/write to the peripherals mounted on the I²C controller. For details, see section 4.4 "Operation Instances."

----End

4.3 Interface Rate Configuration

The default interface rate is 100 KHz in the SDK. To change the interface rate, you need to modify the kernel options and recompile the kernel as follows:

```
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
```

NOTE

There are two cases for CROSS_COMPILE=arm-hisi*XXX*-linux-:

- Hi3516A_V100R001C01SPC*xxx* corresponds to uClibc. CROSS_COMPILE=arm-hisiv300-linux- indicates the uClibc tool chain.
- Hi3516A_V100R001C02SPC*xxx* corresponds to glibc. CROSS_COMPILE=arm-hisiv400-linux- indicates the glibc tool chain.

Go to the following directories and configure the options, as shown in Figure 4-1.

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

Figure 4-1 Configuration options of the interface rate

```
<*> 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 Operation Instances

4.4.1 I²C Read and Write Commands

The following instances describe how to read and write to I²C peripherals by running I²C read and write commands:

- To read I²C peripherals on the console, run **i2c_read**:

  ```
  ~ $ i2c_read <i2c_num> <device_addr> <reg_addr> <end_reg_addr>
  <reg_width> <data_width> <reg_step>
  ```

- For example, to read the register (address of 0x8) of the sil9024 device mounted on I²C controller 2, run the following command:

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

NOTE

- **i2c_num**: I²C controller ID (corresponding to I²C controller 0, I²C controller1, and I²C controller 2 in the *Hi3516A HD IP Camera SoC Data Sheet*)
- **device_addr**: peripheral address (the Hi3516A supports only 7-bit device address)
- **reg_addr**: start address for reading peripheral registers
- **end_reg_addr**: end address for reading peripheral registers
- **reg_width**: bit width of peripheral registers (the Hi3516A supports 8-bit or 16-bit width)
- **data_width**: data width of peripherals (the Hi3516A supports 8-bit or 16-bit data width)
- **reg_step**: incremental step when peripheral registers are read consecutively. The default value is 1, indicating that two or more registers are read consecutively. This parameter is not used when only one register is read.

- To write to I²C peripherals on the console, run **i2c_write**:

  ```
  ~ $ i2c_write <i2c_num> <device_addr> <reg_addr> <value> <reg_width>
  <data_width>
  ```

  For example, to write 0xa5 to the register (address of 0x8) of the sil9024 device mounted on I²C controller 2, run the following command:

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

NOTE

- **i2c_num**: I²C controller ID (corresponding to I²C controller 0, I²C controller1, and I²C controller 2 in the *Hi3516A HD IP Camera SoC Data Sheet*)
- **device_addr**: peripheral address (the I²C controller of the Hi3516A supports only 7-bit device address)
- **reg_addr**: address for writing to peripheral registers
- **value**: data written to peripheral registers
- **reg_width**: bit width of peripherals (the I²C controller of the Hi3516A supports 8-bit or 16-bit width)
- **data_width**: data bit width of peripherals (the I²C controller of the Hi3516A supports 8-bit or 16-bit data width)

4.4.2 I²C Read and Write Programs in Kernel Mode

The following instance describes how to read/write to I²C peripherals by compiling I²C read and write programs in kernel mode:

Step 1 Call functions at the I²C core layer. i2c_adap is obtained, which is a structure of the I²C controller.

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

NOTE

Assume that the new peripheral is mounted on I²C controller 2, the parameter value of **i2c_get_adapter** can be set to **2**.

Step 2 Associate the I²C controller with the new I²C peripheral. hi_client is obtained, which is a data structure that describes the client of I²C peripherals.

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

NOTE

The hi_info structure provides the address of the I²C peripheral.

Step 3 Call the standard read and write functions provided by the I²C core layer to read/write to peripherals.

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

NOTE

- The **client** parameter is the hi_client structure obtained in step 2, which is a data structure that describes the client of I²C peripherals.
- The **buf** parameter is the register or data needs to be read or written.
- The **count** parameter is the length of **buf**.

The code instance is as follows:

NOTE

The following code instance is a sample program, which only serves as a reference for customers when they develop kernel-mode I²C peripheral drivers. The code has no actual functions.

For details about kernel-mode I²C peripheral drivers, see **tlv320aic31** in **mpp/extdrv/tlv320aic31/tlv320aic31.c** of the SDK.

//Information list for I²C peripherals

static struct i2c_board_info hi_info = {

//Each I2C_BOARD_INFO structure represents a supported I²C peripheral. The device name is hi_test and the device address is 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;
```

//Set the flag according to the register bit width parameter.

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

//Set the flag according to the data bit width parameter.

//If the data length is greater than or equal to 3 bytes, the I²C DMA mode needs to be used for the write operation, and the DMA operation flag is required. Note that I2C_M_DMA and I2C_M_16BIT_DATA cannot be configured simultaneously.

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

//Call the I²C standard write function provided by the kernel to perform the write operation.

//The last parameter (**count**) in the i2c_master_send function indicates the total byte length of the transmitted register address and data.

```
        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;
```

//Set the flag according to the register bit width parameter.

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

//Set the flag according to the data bit width parameter.

//If the data length is greater than or equal to 3 bytes, the I²C DMA mode needs to be used for the read operation, and the DMA operation flag is required. Note that I2C_M_DMA and I2C_M_16BIT_DATA cannot be configured simultaneously.

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

//Call the I²C standard read function provided by the kernel to perform the read operation.

//The last parameter (**count**) in the i2c_master_recv function indicates the byte length of the data to be read.

```
        ret = i2c_master_recv(client, buf, count);
        return ret;
}
static int hi_dev_init(void)
{
```

//Allocate an I²C controller pointer.

```
        struct i2c_adapter *i2c_adap;
```

//Call functions at I²C core layer to obtain i2c_adap, which is a structure of the I²C controller. Assume that the new peripheral is mounted on I²C controller 2.

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

//Associate the I²C controller with the new I²C peripheral by mounting the I²C peripheral on I²C controller 2. The address is 0x72. In this way, hi_client is formed.

```
        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);
}
```

----End

4.4.3 I²C Read and Write Programs in User Mode

The following instance describes how to read/write to I²C peripherals by compiling I²C read and write programs in user mode:

Step 1 Open the device file corresponding to the I²C bus to obtain the file descriptor.

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

Step 2 Run the following ioctl commands to configure the peripheral address as well as the bit width and data length of peripheral registers.

```
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);
```

NOTE

- For the register bit width and data length, when the third parameter of ioctl is set to 0, the bit width is 8 bits. When the third parameter of ioctl is set to 1, the bit width is 16 bits.
- If the third parameter of ioctl is 0, the non-DMA mode is used; if the parameter value is 1, the DMA mode is used.
- I2C_DMA and I2C_16BIT_DATA cannot be configured simultaneously.

Step 3 Call the read and write functions to read/write the data.

When the read operation is performed in non-DMA mode, the last parameter of the read function indicates the register bit width.

```
read(fd, recvbuf, reg_width);
When the read operation is performed in DMA mode, the last parameter of
the read function indicates the byte length of the data to be read.
read(fd, recvbuf, data_len);
During the write operation, the last parameter of the write function
indicates the total byte length of the register address and data to be
written.
write(fd, buf, (reg_width + data_width));
```

The code instance is as follows:

NOTE

The following code instance is a sample program, which only serves as a reference for customers when they develop user-mode I^2C peripheral drivers. The code has no actual functions.

For details about user-mode I^2C peripheral drivers, see **i2c_ops** in **osdrv/tools/board/reg-tools-1.0.0/source/tools/i2c_ops.c** of the SDK.

```
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;
}
```

----End

5 Operation Guide to the SPI

5.1 Preparations

Before using the serial peripheral interface (SPI), ensure that the following items are available:

- Linux kernel released in the SDK
- File systems

You can use the local file system Yaffs2, Jffs2, or SquashFS released in the SDK or mounted to the NFS.

5.2 Procedure

To use the SPI, perform the following steps:

Step 1 Start the board, and load the local file system Yaffs2, Jffs2, or SquashFS or mount the local file system to the NFS.

Step 2 Load the kernel. By default, all drivers related to the SPI module are compiled in the kernel. Therefore, you do not need to load the drivers.

Step 3 Run SPI read and write commands on the console or compile SPI read and write programs in kernel mode or user mode to read/write to the peripherals mounted on one of the chip selects (CSs) of the SPI controller. For details, see section 5.3 "Operation Instances."

----End

5.3 Operation Instances

5.3.1 SPI Read and Write Commands

The following instances describe how to read and write to SPI peripherals by running SPI read and write commands:

- To read SPI peripherals on the console, run **spi_read**:

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

[num_reg] can be omitted and its default value is 1 (indicating reading one register). **[dev_width]**, **[reg_width]**, and **[data_width]** can be omitted and their default values are 1 (indicating one byte).

For example, to read the register (address of 0x0) of the device (address of 0x2) mounted on CS 0 of the SPI controller 0, run the following command:

```
~ $ ssp_read 0x0 0x0 0x2 0x0 0x10 0x1 0x1 0x1
```

NOTE

- **spi_num**: SPI controller ID (corresponding to SPI controller 0 and SPI controller 1 in the *Hi3516A HD IP Camera SoC Data Sheet*)
- **csn**: CS ID (SPI controller 0 and SPI controller 1 of the Hi3516A have one CS and three CSs respectively)
- **dev_addr**: peripheral address
- **reg_addr**: address of the first peripheral register to be read among multiple peripheral registers whose addresses are consecutive
- **num_reg**: number of peripheral registers to be read
- **dev_width**: address bit width of peripherals (8 bits)
- **reg_width**: address bit width of peripheral registers (8 bits)
- **data_width**: data bit width of peripherals (8 bits)

- To write to SPI peripherals on the console, run **spi_write**:

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

  [dev_width], **[reg_width]**, and **[data_width]** can be omitted and their default values are 1 (indicating one byte).

  For example, to write 0x65 to the register (address of 0x0) of the device (address of 0x2) mounted on CS 0 of the SPI controller 0, run the following command:

  ```
  ~ $ ssp_write 0x0 0x0 0x2 0x0 0x65 0x1 0x1 0x1
  ```

NOTE

- **spi_num**: SPI controller ID (corresponding to SPI controller 0 and SPI controller 1 in the *Hi3516A HD IP Camera SoC Data Sheet*)
- **csn**: CS ID (SPI controller 0 and SPI controller 1 of the Hi3516A have one CS and three CSs respectively)
- **dev_addr**: peripheral address
- **reg_addr**: peripheral register address
- **data**: data written to peripheral registers
- **dev_width**: address bit width of peripherals (8 bits)
- **reg_width**: address bit width of peripheral registers (8 bits)
- **data_width**: data bit width of peripherals (8 bits)

NOTE

The SPI read and write commands can be used only to read and write to sensors.

5.3.2 SPI Read and Write Programs in Kernel Mode

The following instance describes how to read/write to SPI peripherals by compiling SPI read and write programs in kernel mode:

Step 1 Call functions at the SPI core layer. A structure of the SPI controller is obtained.

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

NOTE

- **bus_num** is the ID of the SPI controller connected to the SPI peripherals to be read or written to.
- **hi_master** is a pointer to the spi_master structure that describes the SPI controller.

Step 2 Call functions at the SPI core layer by name of each SPI CS at the core layer to obtain a structure that describes an SPI peripheral mounted to one CS of an SPI controller.

```
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);
```

NOTE

- **spi_bus_type** is a pointer to the bus_type structure that describes the SPI bus and is defined at the core layer.
- **hi_spi** is a pointer to the spi_device structure that describes SPI peripherals.

Step 3 Configure members in the spi_transfer structure, call functions at the SPI core layer, and add spi_transfer to the queue of spi_message.

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

NOTE

- Parameter **t** is a pointer to the spi_transfer structure that describes a transferred message frame.
- Parameter **m** is a pointer to the spi_message structure that describes a transferred message queue.

Step 4 Call the standard read and write functions provided by the SPI core layer to read/write to peripherals.

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

NOTE

- Parameter **spi** is a pointer to the **spi_device** structure that describes SPI peripherals.
- The spi_async function is used to perform asynchronous SPI read and write operations.
- The spi_sync function is used to perform synchronous SPI read and write operations.

The code instance is as follows:

NOTE

- The following code instance is a sample program for asynchronously reading and writing to the SPI peripheral imx185. The code only serves as a reference for customers when they develop kernel-mode SPI peripheral drivers and has no actual functions.
- For details about kernel-mode SPI peripheral drivers, see **sensor_spi** in **mpp/extdrv/sensor_spi/sensor_spi.c** of the SDK.

//Module parameters. The transferred SPI controller ID is the SPI bus ID and the SPI CS ID.

```
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");
```

//A structure that describes the SPI controller

```
struct spi_master  *hi_master;
```

//A structure that describes SPI peripherals

```
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);
```

//After the transfer of the message queue is complete, the **state** member of spi_message is set to **null** at the core layer.

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

//Configure the SPI transfer mode.

```
    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;
```

//Initialize and configure the SPI transfer queue.

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

NOTE

When the function is returned after calling, the SPI read and write operations may not be complete because the operations are asynchronous. Therefore, the address space pointed by the transferred parameters must be a local static variable or a global variable to prevent the address space transferred to spi_async from being released when the function is returned. The prototype of the spi_async function is int spi_async(struct spi_device *spi, struct spi_message *message). **m** and **t** must be static variables and the buffer pointed by **t** must be static.

```
    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);
```

//After the transfer of the message queue is complete, the **state** member of spi_message is set to **null** at the core layer.

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

//Configure the SPI transfer mode.

```
    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;
```

//Initialize and configure the SPI transfer queue.

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

NOTE

When the function is returned after calling, the SPI read and write operations may not be complete because the operations are asynchronous. Therefore, the address space pointed by the transferred parameters must be a local static variable or a global variable to prevent the address space transferred to spi_async from being released when the function is returned. The prototype of the spi_async function is int spi_async(struct spi_device *spi, struct spi_message *message). **m** and **t** must be static variables and the buffer pointed by **t** must be static.

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

//External reference declaration for the spi_bus_type structure that describes the SPI bus and is defined at the SPI core layer.

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

//The structure that describes the SPI controller is obtained by using the SPI controller ID.

```
    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);
```

//The pointer to the **device** member of spi_device is obtained by name of each CS at the SPI core layer.

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

//The structure that describes SPI peripherals is obtained by using the pointer to the **device** member of spi_device.

```
        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;
}
```

----End

5.3.3 SPI Read and Write Programs in User Mode

The following instance describes how to read/write to SPI peripherals mounted on CS 0 of SPI controller 0 by compiling SPI read and write programs in user mode:

Step 1 Open the device file corresponding to the SPI bus to obtain the file descriptor.

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

Step 2 Configure the SPI transfer mode by using ioctl.

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

NOTE

- **SPI_MODE_3** indicates the mode in which the SPI clock and phase are both 1.

- **SPI_LSB_FIRST** indicates that the format of each data segment is big endian during the SPI transfer.

CAUTION

For details about macros **SPI_MODE_3** and **SPI_LSB_FIRST**, see **include/linux/spi/spi.h**. When these two macros are used in user mode, the code in user mode can include **osdrv/tools/board/reg-tools-1.0.0/include/common/hi_spi.h** in the SDK.

For details about the clock, phase, endian mode of the SPI, see the *Hi3516A HD IP Camera SoC Data Sheet*.

Step 3 Use ioctl to read and write data.

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

NOTE

- **mesg** indicates the start address of the spi_ioc structure array that transfers a message frame. The total length of the data that is read or written each time cannot be greater than 4 KB.
- **SPI_IOC_MESSAGE(n)** indicates commands used to read and write *n* message frames in full-duplex mode.

The code instance is as follows:

NOTE

- The following code instance is a sample program for synchronously reading and writing to IMX185 (an SPI peripheral). The code only serves as a reference for customers when they develop the operation program for user-mode SPI peripherals and has no actual functions.
- For details about user-mode SPI peripheral drivers, see **ssp_rw** in **osdrv/tools/board/reg-tools-1.0.0/source/tools/ssp_rw.c** of the SDK.

```
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;
}
```

----End

6 Appendix

6.1 Partitioning a Storage Device

You can check the current partition status of a device by following section 6.1.1 "Checking the Current Partition Status of a Storage Device." If the device is not partitioned, you need to partition it as follows:

- If partitions exist, skip the operations in this section and go to section 6.2 "Formatting a Partition."
- If partitions do not exist, enter the **fdisk** command at the command prompt of the console:

  ```
  ~ $ fdisk device node
  ```

  Press **Enter** and enter the **m** command. Continue with the operations based on the help information.

  The device node depends on the type of the actual storage device. For details, see the operation instances in preceding chapters.

6.1.1 Checking the Current Partition Status of a Storage Device

To check the current partition status of a storage device, enter the **p** command at the command prompt of the console:

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

The partition status similar to the following is displayed:

```
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
```

The preceding information indicates that the device is not partitioned. In this case, you need to partition the device by following section 6.1.2 "Creating Partitions for a Storage Device.", and save the partition information by following section 6.1.3 "Saving the Partition Information."

6.1.2 Creating Partitions for a Storage Device

Perform the following steps:

Step 1 Create partitions.

Under the prompt, enter the **n** command to create partitions.

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

The following information is displayed on the console:

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

Step 2 Create the primary partition.

Enter the **p** command to select the primary partition:

```
p
```

Step 3 Select the number of partitions.

In this example, set the number to **1**, that is, enter **1**.

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

The following information is displayed on the console:

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

Step 4 Select the start cylinder.

This example takes the default value **1**. Press **Enter**.

```
Using default value 1
```

Step 5 Select the end cylinder.

This example takes the default value **970**. Press **Enter**.

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

Step 6 Select a data format for the storage device.

The default value is **Linux**. This example takes Win95 FAT; therefore, run the **t** command to change the value.

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

Then run the following command:

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

Run the **l** command to view the details of all the partitions of the fdisk.

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

Step 7 Check the status of partitions.

Enter the **p** command to view the partition information:

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

If the partition information is displayed on the console, the partitioning is successful.

----End

6.1.3 Saving the Partition Information

To save the partition information, run the following command:

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

If the following information is displayed on the console, the partition information is successfully saved:

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

6.2 Formatting a Partition

Perform the following operations:

- If partitions are formatted, skip the operations in this section and go to section 6.3 "Mounting a Directory."
- If partitions are not formatted, run the **mkdosfs** command to format the partitions:

  ```
  ~ $ mkdosfs -F 32 Partition name
  ```

  The name of a partition depends on the type of the actual storage device. For details, see the operation instances in preceding chapters.

  If no error information is displayed on the console, a partition is formatted successfully:

  ```
  ~ $
  ```

6.3 Mounting a Directory

Run the **mount** command to mount the partitions to the **/mnt** directory. Then you can read and write files.

```
~ $ mount -t vfat Partition name /mnt
```

The name of a partition depends on the type of the actual storage device. For details, see the operation instances in the preceding chapters.

6.4 Reading/Writing to a File

There are various read and write operations. This section takes the **cp** command as an example to read and write to a file.

To write to a file, copy the **test.txt** file in the current directory to the **mnt** directory of a storage device by running the following command:

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