Peripheral Driver

Operation Guide

Issue	**00B02**
Date	**2017-11-20**

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 Ethernet (ETH) and universal serial bus (USB) 2.0 Host driver installed. It covers the following topics, namely, preparations, operation procedures, precautions to be taken during operation, and operation instances.

Related Version

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

Product Name	Version
Hi3536D	V100

Intended Audience

This document is intended for:

- Technical support personnel
- Software development engineers

Change History

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

Issue 00B02 (2017-11-20)

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

Chapter 1 Operation Guide to the ETH Module

In section 1.1, figure 1-1 is modified.

Section 1.4.2 is deleted.

Chapter 2 Operation Guide to the USB 2.0 Host Module

The description in section 2.3.3 is modified.

Issue 00B01 (2017-09-08)

This issue is the first draft release.

Contents

Figures

1 Operation Guide to the ETH Module

NOTE

The following addresses are examples only. You need to configure the addresses as required.

1.1 Operation Instance

NOTE

The ETH drivers for the Hi3536D V100 are compiled in the kernel by default. Therefore, you can configure IP addresses without loading ETH drivers.

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

- The ETH 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 Hi3536D V100 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:

```
cd kernel/linux-4.9.y
cp arch/arm/configs/hi3536dv100_full_defconfig .config
make ARCH=arm CROSS_COMPILE=arm-hisivXXX-linux- menuconfig
```

NOTE

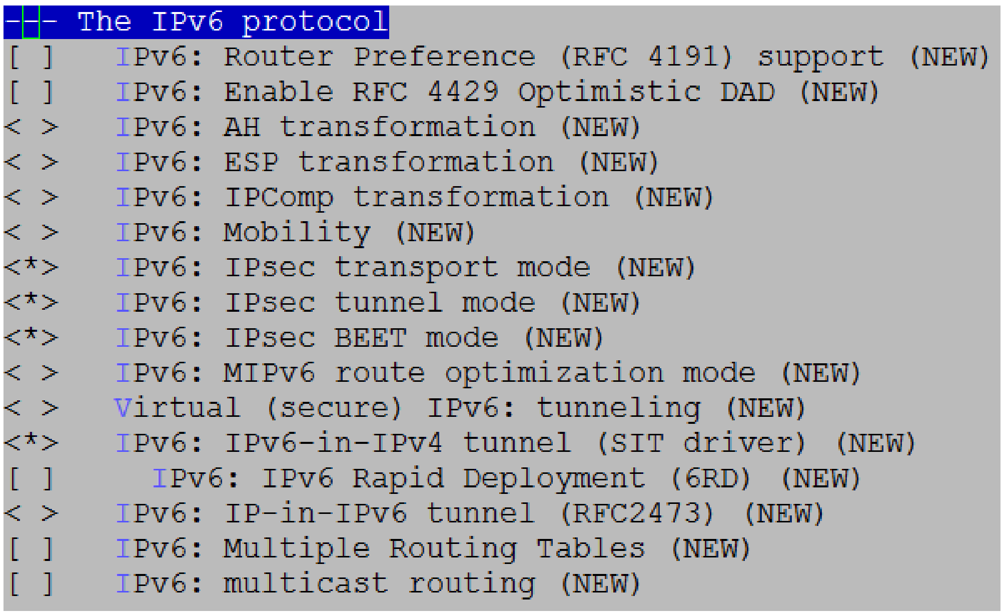
There are two cases for CROSS_COMPILE=arm-hisi*XXX*-linux-:

- Hi3536D_V100R001C01SPCxxx corresponds to Uclibc. CROSS_COMPILE=arm-hisiv510-linux- indicates the Uclibc tool chain.
- Hi3536D_V100R001C02SPCxxx corresponds to Glibc. CROSS_COMPILE=arm-hisiv610-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 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)
```

The IPv6 configurations are as follows:

- Configure the IP address and the default gateway by running the following command:
```
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:
```
ping -6  <ipv6address>
```
  Example: `ping -6  2001:da8:207::9403`

1.3 Configuring the PHY Address

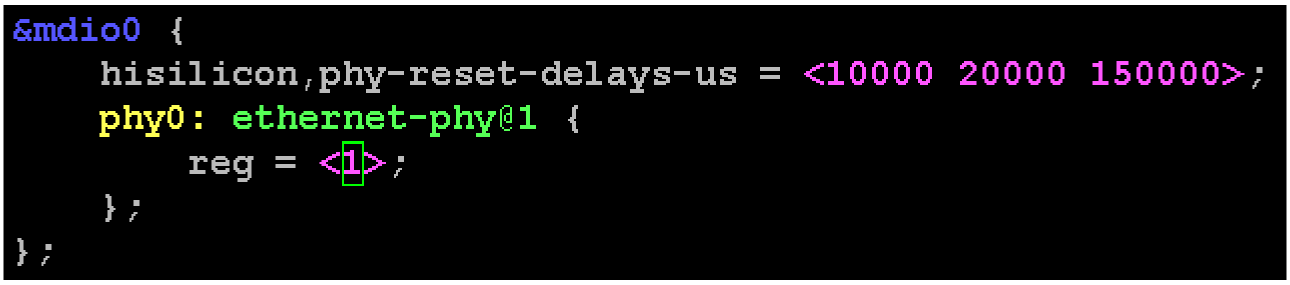
On the Hi3536D V100 demo board, the default PHY address is 1. When different PHY addresses are selected, the PHY address configuration must be modified under the U-boot or the kernel.

- Configuring the PHY address under the U-boot.

  To configure the PHY address under the U-boot, change the value of **HISFV_PHY_U** macro definition in the configuration files of U-boot. The U-boot of the Hi3536D V100 contains **the following configuration file:**
  - **include/configs/hi3536dv100.h**
- Configuring the PHY address under the kernel.

  To configure the PHY address, modify the **arch/arm/boot/dts/hi3536dv100-demb.dts** file: As shown in Figure 1-2, the value **1** in "reg = <1>" indicates the PHY address.

Figure 1-2 Diagram of node for configuring the PHY address


```
&mdio0 {
    hisilicon,phy-reset-delays-us = <10000 20000 150000>;
    phy0: ethernet-phy@1 {
        reg = <1>;
    };
};
```

1.4 Configuring the IEEE 802.3x Flow Control Function

1.4.1 Function Description

The ETH 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 ETH 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 ETH receives the pause frame with zero flow control time from the peer end during waiting.

1.4.2 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 and disable TX flow control by running the following commands:

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

RX flow control is enabled by default and cannot be disabled. Therefore, RX flow control cannot be configured by using the ethtool.

2 Operation Guide to the USB 2.0 Host Module

NOTE

The USB 2.0 of the Hi3536D V100 supports only the host mode.

2.1 Preparations

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

- U-boot and Linux kernel released in the SDK
- File systems

  You can use the local file system Yaffs2, ubifs, Jffs2, SquashFS, or Cramfs (Jffs2 is recommended) released in the SDK or mounted to the NFS.

2.2 Procedure

The operation procedure is as follows:

Step 1 Start the board, and load the NFS or the file system Yaffs2, ubifs, Jffs2, SquashFS, 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 Instance Related to the USB Flash Drive."

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-hi3536dv100

----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 **sdXY**, 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/input**.

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

```
cat  /dev/input 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** and **event0** nodes are generated in **/dev/input**.

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

2.4 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/input**. 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) is provided in the gpm tool. You can perform encoding by using the test programs, making the development easier.

3 Operation Guide to the SATA

3.1 Preparations

Before using the I^{2}C, ensure that the following items are available:

- Standard serial advanced technology attachment (SATA) hard disk
- U-boot and Linux kernel released in the SDK
- File systems

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

3.2 Procedure

Before testing an SATA hard disk, perform the following steps:

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

Step 2 Load the drivers. By default, all drivers related to the SATA module are compiled in the kernel, you do not need to run the command for loading drivers. For details, see section 3.3 "Operation Instances."

- Drivers related to the file system and storage devices
  - `nls_base`
  - `nls_cp437`
  - `fat`
  - `vfat`
  - `msdos`
  - `nls_iso8859-1`
  - `nls_ascii`
  - `scsi_mod`
  - `sd_mod`
- Drivers related to hard disks
  - `libata`
  - `ahci`

----End

3.3 Operation Instances

Note the following:

NOTE

In the following commands, *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 partitioning command such as **$ fdisk /dev/sda** or **$ parted /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 mounting command such as **$ mount -t vfat /dev/sda1 /mnt**.

Perform the following steps:

Step 1 Check whether the SATA hard disk is partitioned.

- Run **ls /dev** to view the system device files. If partition information sd*XY* is not displayed, the SATA hard disk has not been partitioned yet. In this case, partition the SATA hard disk by following section 6.1 "Partitioning a Storage Device."
- If sd*XY* is displayed, the hard disk is detected and partitioned. In this case, go to step 2.

Step 2 Check whether the SATA hard disk is formatted.

- If the SATA hard disk 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 SATA hard disk by following section 6.4 "Reading/Writing to a File."

----End

3.4 Precautions

The Hi3536D V100 SATA drivers support hot plug. After the hot plug, you need to unmount the nodes mounted to the hard disk. Otherwise, the device nodes of the SATA hard disk change after the SATA hard disk is reinserted.

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, Ext4, 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, Ext4, 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."

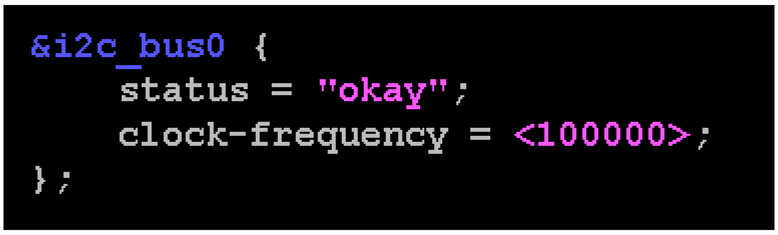
----End

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 **arch/arm/boot/dts/hi3536dv100-demb.dts** and recompile the kernel. Perform the following operation:

Change the value of the **clock-frequency** attribute in the i2c_bus0 node, as shown in Figure 4-1.

Figure 4-1 Configuration options of the interface rate



```
&i2c_bus0 {
    status = "okay";
    clock-frequency = <100000>;
};
```

4.4 Operation Instances

4.4.1 I²C Read and Write Commands

The following instances describe how to read and write to I^{2}C peripherals by running I^{2}C read and write commands:

- To read I^{2}C 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 I^{2}C controller 0, run the following command:

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

NOTE

- **i2c_num**: I^{2}C controller ID (corresponding to I^{2}C controller in the *Hi3536DV100 H.265/H.264 Decoder Processor Data Sheet)*
- **device_addr**: peripheral address (The read and write bits should be added to the device address in the command line.)
- **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 Hi3536D V100 supports 8-bit or 16-bit width)
- **data_width**: data width of peripherals (the Hi3536D V100 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 I^{2}C 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 I^{2}C controller 0, run the following command:

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

NOTE

- **i2c_num**: I^{2}C controller ID (corresponding to I^{2}C controller in the *Hi3536DV100 H.265/H.264 Decoder Processor Data Sheet*)
- **device_addr**: peripheral address (The read and write bits should be added to the device address in the command line.)

- **reg_addr**: address for writing to peripheral registers
- **value**: data written to peripheral registers
- **reg_width**: bit width of peripherals (the I^{2}C controller of the Hi3536D V100 supports 8-bit or 16-bit width)
- **data_width**: data bit width of peripherals (the I^{2}C controller of the Hi3536D V100 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 I^{2}C peripherals by compiling I^{2}C read and write programs in kernel mode:

Step 1 Call functions at the I^{2}C core layer. i2c_adap is obtained, which is a structure of the I^{2}C controller.

```
i2c_adap = i2c_get_adapter(0);
```

NOTE

Assume that the new peripheral is mounted on I^{2}C controller 0, the parameter value of **i2c_get_adapter** can be set to **0**.

Step 2 Associate the I^{2}C controller with the new I^{2}C peripheral. hi_client is obtained, which is a data structure that describes the client of I^{2}C peripherals.

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

NOTE

The hi_info structure provides the address of the I^{2}C peripheral.

Step 3 Call the standard read function or transfer function provided by the I^{2}C core layer to read/write to peripherals.

```
ret = i2c_master_send(client, buf, count);
ret = i2c_transfer(client->adapter, msg, 2);
```

NOTE

- The **client** parameter is the hi_client structure obtained in step 2, which is a data structure that describes the client of I^{2}C peripherals.
- The **buf** parameter is the data to be written to the register.
- The **count** parameter is the length of **buf**.
- The **msg** parameter is the initial address of two **i2c_msg** during the read operation.

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 I^{2}C peripheral drivers. The code has no actual functions.

//Information list for I^{2}C peripherals

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

//Each I2C_BOARD_INFO structure represents a supported I^{2}C peripheral. The device name is hi_test and the device address is 0x72.

```
        I2C_BOARD_INFO("hi_test", 0x39),
};
```

NOTE

When the code is called, the read and write bits cannot be included in the device address, but need to be included in the command line.

```
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;
```

//Call the I^2C standard write function provided by the kernel to perform the write operation.

```
    ret = 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;
```

//When the sensor performs the read operation, a register address should be written first to read the data of the specified register. Therefore, the **i2c_transfer** provided by the kernel should be called to transfer two **msg** to perform the write operation and then the read operation.

```
    ret = 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)
{
```

//Allocate an I^{2}C controller pointer.

```
    struct i2c_adapter *i2c_adap;
```

//Call functions at I^{2}C core layer to obtain i2c_adap, which is a structure of the I^{2}C controller. Assume that the new peripheral is mounted on I^{2}C controller 0.

```
    i2c_adap = i2c_get_adapter(0);
```

//Associate the I^{2}C controller with the new I^{2}C peripheral by mounting the I^{2}C peripheral on I^{2}C controller 0. The address is 0x39. 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 I^{2}C peripherals by compiling I^{2}C read and write programs in user mode:

Step 1 Open the device file corresponding to the I^{2}C bus to obtain the file descriptor.

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

Step 2 Call the read and write functions to read/write the data.

```
ioctl(fd, I2C_RDWR, &rdwr);
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^{2}C peripheral drivers. The code has no actual functions.

For details about user-mode I^{2}C 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 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;
```

NOTE

When the code is called, the read and write bits cannot be included in the device address, but need to be included in the command line.

```
    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;
    }
```

----End

5 Operation Guide to the GPIO

5.1 Preparations

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

- Linux kernel released in the SDK
- File systems

NOTE

You can use the local file system Ubifs, Jffs2, Ext4, or SquashFS released in the SDK or mounted to the NFS.

5.2 Procedure

To use the GPIO, perform the following steps:

Step 1 Start the board, and load the local file system Ubifs, Jffs2, Ext4, or SquashFS or mount the local file system to the NFS.

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

Step 3 Configure the pin multiplexing function.

Step 4 Run related commands on the console or compile GPIO operation programs in kernel mode or user mode to implement GPIO input and output operations. For details, see section 5.3 "Operation Instances."

NOTE

Before the GPIO operations are performed, the corresponding pins need be multiplexed as the GPIO function.

----End

5.3 Operation Instances

5.3.1 Example of GPIO Operation Commands

The following example implements GPIO read and write operations by using commands:

Step 1 Export the number of the GPIO to be operated by using the echo command on the console.

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

NOTE

Each GPIO group contains eight GPIO pins.

N indicates the number of the GPIO to be operated, and equals (GPIO group number x 8 + Offset within the group). For example, the number of GPIO4_2 is 34 (4 x 8 + 2).

After the GPIO number is exported, the **/sys/class/gpio/gpioN** directory is generated.

For example, to export the number of GPIO4_2, run the following command:

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

Step 2 Set the GPIO direction using the echo command on the console.

- For the input: `echo in > /sys/class/gpio/gpioN/direction`
- For the output: `echo out > /sys/class/gpio/gpioN/direction`

For example, to set the direction of GPIO4_2, run either of the following commands:

- For the input: `echo in > /sys/class/gpio/gpio34/direction`
- For the output: `echo out > /sys/class/gpio/gpio34/direction`

NOTE

- There are two GPIO directions: in and out.
- You can run the cat command **cat /sys/class/gpio/gpioN/direction** to check the GPIO direction. For example, run **cat /sys/class/gpio/gpio34/direction** to check the direction of GPIO4_2.

Step 3 Check the GPIO input value or set the GPIO output value by using the cat or echo command on the console.

Check the input value: cat /sys/class/gpio/gpioN/value

Set the output to low level: echo 0 > /sys/class/gpio/gpioN/value

Set the output to high level: echo 1 > /sys/class/gpio/gpioN/value

NOTE

The value of the GPIO level can only be 0 or 1. The value 0 indicates low level, and 1 indicates high level.

Step 4 Unexport the number of the operated GPIO by using the echo command on the console.

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

----End

5.3.2 GPIO Operation Example in Kernel Mode

The following operation examples implement the GPIO read, write, and interrupt operations in kernel mode:

5.3.2.1 Example of GPIO Read and Write Operations in Kernel Mode

The operation example is as follows:

Step 1 Register the GPIO.

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

NOTE

- Each GPIO group contains eight GPIO pins.
- **gpio_num** indicates the number of the GPIO to be operated, and equals (GPIO group number x 8 + Offset within the group). For example, the number of GPIO4_2 is 34 (4 x 8 + 2).

Step 2 Set the GPIO direction.

- For the input: gpio_direction_input(gpio_num)
- For the output: gpio_direction_output(gpio_num, gpio_out_val)

NOTE

If the direction is output, the parameter **gpio_out_val**, which indicates an initial output value, needs to be set.

Step 3 Check the GPIO input value or set the GPIO output value.

Check the input value: gpio_get_value(gpio_num);

Set the output to low level: gpio_set_value(gpio_num, 0);

Set the output to high level: gpio_set_value(gpio_num, 1);

NOTE

The input value is the return value of gpio_get_value(gpio_num).

Step 4 Release the registered GPIO number.

```
gpio_free(gpio_num);
```

----End

5.3.2.2 Example of GPIO Interrupt Operations in Kernel Mode

CAUTION

The built-in GPIO driver of the standard kernel is used by default. If customers choose to use their own GPIO driver to register the interrupt instead of the built-in GPIO driver of the standard kernel, the built-in GPIO driver of the kernel should be disabled. Otherwise, driver conflicts will occur and the operations will be affected.

Perform the following steps to disable the built-in GPIO driver of the standard kernel:

- Open the DTS file **arch/arm/boot/dts/hi3536dv100-demb.dts**. Change the GPIO status from "okay" to "disabled", as shown in Figure 5-1:

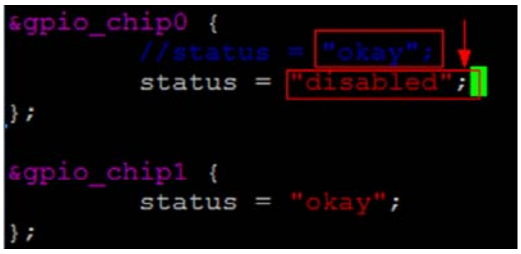
  vi arch/arm/boot/dts/hi3536dv100-demb.dts
- Save and exit. Recompile the kernel.

  cp arch/arm/configs/hi3536dv100_full_defconfig .config

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

  make ARCH=arm CROSS_COMPILE=arm-hisivXXX-linux- uImage

Figure 5-1 Example of disabling the build-in GPIO of the kernel



```
&gpio_chip0 {
        //status = "okay";
        status = "disabled";
};

&gpio_chip1 {
        status = "okay";
};
```

Step 1 Register the GPIO.

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

NOTE

- Each GPIO group contains eight GPIO pins.
- **gpio_num** indicates the number of the GPIO to be operated, and equals (GPIO group number x 8 + Offset within the group). For example, the number of GPIO4_2 is 34 (4 x 8 + 2).

Step 2 Set the GPIO direction.

```
gpio_direction_input(gpio_num)
```

NOTE

The direction of the GPIO pin that is used as the interrupt source must be configured as input.

Step 3 Map the number of the GPIO to be operated to the number of the corresponding interrupt.

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

NOTE

The interrupt number is the return value of gpio_to_irq(gpio_num).

Step 4 Register the interrupt.

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

NOTE

Irqflags indicates the type of the interrupt to be registered. The common interrupt types are as follows:

- IRQF_SHARED: shared interrupt
- IRQF_TRIGGER_RISING: rising-edge-triggered interrupt
- IRQF_TRIGGER_FALLING: falling-edge-triggered interrupt
- IRQF_TRIGGER_HIGH: high-level-triggered interrupt
- IRQF_TRIGGER_LOW: low-level-triggered interrupt

Step 5 Release the registered interrupt number and GPIO number.

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

The code instance is as follows:

NOTE

The following code instance is a sample program of the GPIO operation, which only serves as a reference for customers when they develop kernel-mode GPIO operation programs. The code has no actual application function.

```
#include <linux/delay.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/module.h>
//Module parameters, including the GPIO group number, offset within the
group, direction, and initial output value
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");

//Module parameter, which indicates the interrupt trigger type
/*
 * 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)
{
         //Set the direction to input.
        if (gpio_direction_input(gpio_num)) {
                pr_err("[%s %d]gpio_direction_input fail!\n",
                                __func__, __LINE__);
                return -EIO;
        }
         //Read the GPIO input value.
        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;
}
//Interrupt handler function
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;
         //Set the direction to input.
        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;
         //Map the interrupt number based on the GPIO number.
        irq_num = gpio_to_irq(gpio_num);
         //Register the interrupt.
        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__);
         //Release the registered interrupt.
        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)
{
         //Set the direction to output and output an initial value.
        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;
         //Register the number of the GPIO to be operated.
        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);
         //Release the registered GPIO number.
        gpio_free(gpio_num);
}

module_exit(gpio_dev_test_exit);

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

----End

5.3.2.3 GPIO Operation Example in User Mode

The following operation example implements the GPIO read and write operations in user mode:

Step 1 Export the number of the GPIO to be operated.

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

NOTE

- Each GPIO group contains eight GPIO pins.
- **gpio_num** indicates the number of the GPIO to be operated, and equals (GPIO group number x 8 + Offset within the group). For example, the number of GPIO4_2 is 34 (4 x 8 + 2).

Step 2 Set the GPIO direction.

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

For the input: `fprintf(fp, "in");`

For the output: `fprintf(fp, "out");`

```
fclose(fp);
```

Step 3 Check the GPIO input value or set the GPIO output value.

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

Check the input value: `fread(buf, sizeof(char), sizeof(buf) - 1, fp);`

Set the output to low level:

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

Set the output to high level:

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

Step 4 Unexport the number of the operated GPIO.

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

The code instance is as follows:

NOTE

The following code instance is a sample program of the GPIO operation, which only serves as a reference for customers when they develop user-mode GPIO operation programs. The code has no actual application function.

```
#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;

}
```

----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
```