



Hi3861 V100 / Hi3861L V100 Device Driver

Development Guide

Issue	01
Date	2020-04-30

Copyright © HiSilicon (Shanghai) Technologies Co., Ltd. 2020. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of HiSilicon (Shanghai) Technologies Co., Ltd.

Trademarks and Permissions



HISILICON, and other HiSilicon icons are trademarks of HiSilicon Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between HiSilicon and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided “AS IS” without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

HiSilicon (Shanghai) Technologies Co., Ltd.

Address: New R&D Center, 49 Wuhe Road,
Bantian, Longgang District,
Shenzhen 518129 P. R. China

Website: <https://www.hisilicon.com/en/>

Email: support@hisilicon.com



About This Document

Purpose

This document describes the device driver development of Hi3861 V100, including the working principles, API usage methods based on scenarios, and precautions.

Related Versions

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

Product Name	Version
Hi3861	V100
Hi3861L	V100



Intended Audience

The document is intended for:




- Technical support engineers
- Software development engineers

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.



Symbol	Description
 CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
 NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	<p>This issue is the first official release.</p> <ul style="list-style-type: none">• In 1.2 Function Description, the description of <code>hi_uart_quit_read</code> is added.• In 3.4 Precautions, the description of sending failure is added.• In 5.3 Development Guidance, the description of data formats is deleted. The code sample is updated.• In 5.4 Precautions, the description of the <code>delay_cnt</code> parameter in the <code>hi_adc_read</code> API is updated.• In 6.2 Function Description, the descriptions of the <code>hi_gpio_get_dir</code>, <code>hi_gpio_get_output_val</code>, <code>hi_gpio_set_isr_mask</code>, and <code>hi_gpio_set_isr_mode</code> are added. The names of <code>hi_gpio_register_isr_function</code> and <code>hi_gpio_unregister_isr_function</code> are updated.• In 6.4 Precautions, the precautions for <code>hi_gpio_set_isr_mask</code> and <code>hi_gpio_set_isr_mode</code> are added.• 9.4 Precautions is updated.



Issue	Date	Change Description
00B05	2020-03-06	<ul style="list-style-type: none">• In 1.2 Function Description, the description of hi_uart_lp_save and hi_uart_lp_restore is added.• In 1.4 Precautions, the precautions about hi_uart_lp_restore and hi_uart_lp_save are added.
00B04	2020-02-12	<ul style="list-style-type: none">• In 1.2 Function Description, the description of hi_uart_is_buf_empty and hi_uart_is_busy is added.• 1.4 Precautions is updated.• In 7.1 Overview, the description of the 256-bit fields is added.• In 7.2 Function Description, Table 7-1 is updated.• In 7.3 Development Guidance, the description of sample 1 and the comments of sample 2 are updated.• 9 SDIO is added.• In 10.2 Function Description, the description of hi_sensor_code_to_temperature and hi_sensor_temperature_to_code is added.• In 11.2 Function Description, the name of hi_dma_link_list_transfer is updated and the description of hi_dma_is_init is added.
00B03	2020-01-15	<ul style="list-style-type: none">• In 1.2 Function Description and 2.2 Function Description, the description of supporting DMA data transmission and reception is added.• In 2.4 Precautions, the precautions for using the Microwire frame protocol are added.• 4 I2S is added.• In 5 LSADC, the APIs are updated. 5.2 Function Description, 5.3 Development Guidance, and 5.4 Precautions are updated.• In 11.2 Function Description, the description of using DMA to transfer data in the SPI/UART is added.



Issue	Date	Change Description
00B02	2019-12-19	<ul style="list-style-type: none">• In 2.4 Precautions, a precaution description is added.• 7.2 Function Description and Table 7-1 are updated.• 7.3 Development Guidance is updated.
00B01	2019-11-15	This issue is the first draft release.



Contents

About This Document.....	i
1 UART.....	1
1.1 Overview.....	1
1.2 Function Description.....	1
1.3 Development Guidance.....	2
1.4 Precautions.....	3
2 SPI.....	4
2.1 Overview.....	4
2.2 Function Description.....	4
2.3 Development Guidance.....	5
2.4 Precautions.....	6
3 I²C.....	7
3.1 Overview.....	7
3.2 Function Description.....	7
3.3 Development Guidance.....	8
3.4 Precautions.....	8
4 I²S.....	9
4.1 Overview.....	9
4.2 Function Description.....	9
4.3 Development Guidance.....	10
4.4 Precautions.....	12
5 LSADC.....	13
5.1 Overview.....	13
5.2 Function Description.....	13
5.3 Development Guidance.....	13
5.4 Precautions.....	14
6 IO/GPIO.....	15
6.1 Overview.....	15
6.2 Function Description.....	15
6.3 Development Guidance.....	16
6.4 Precautions.....	16



7 eFuse	17
7.1 Overview	17
7.2 Function Description	17
7.3 Development Guidance	21
7.4 Precautions	22
8 Flash	24
8.1 Overview	24
8.2 Function Description	24
8.3 Development Guidance	24
8.4 Precautions	25
9 SDIO	26
9.1 Overview	26
9.2 Function Description	26
9.3 Development Guidance	27
9.4 Precautions	31
10 Tsensor	32
10.1 Overview	32
10.2 Function Description	32
10.3 Development Guidance	33
10.4 Precautions	35
11 DMA	36
11.1 Overview	36
11.2 Function Description	36
11.3 Development Guidance	37
11.4 Precautions	38
12 SysTick	39
12.1 Overview	39
12.2 Function Description	39
12.3 Development Guidance	39
12.4 Precautions	40
13 PWM	41
13.1 Overview	41
13.2 Function Description	41
13.3 Development Guidance	41
13.4 Precautions	42
14 WDG	43
14.1 Overview	43
14.2 Function Description	43
14.3 Development Guidance	43



14.4 Precautions.....	44
-----------------------	----



1 UART

- [1.1 Overview](#)
- [1.2 Function Description](#)
- [1.3 Development Guidance](#)
- [1.4 Precautions](#)

1.1 Overview

The universal asynchronous receiver transmitter (UART) is an asynchronous serial communication interface. It is used to connect to the UART interface of an external chip for inter-communication. The chip provides three UART units.

1.2 Function Description

NOTE

If the UART driver needs to support DMA data transmission and reception, ensure that the DMA driver has been initialized.

The UART module provides the following APIs:

- `hi_uart_init`: Initializes the UART.
- `hi_uart_read`: Reads data.
- `hi_uart_write`: Writes data.
- `hi_uart_deinit`: Deinitializes the UART.
- `hi_uart_set_flow_ctrl`: Sets the hardware flow control of the UART.
- `hi_uart_write_immediately`: Writes data in polling mode.
- `hi_uart_get_attribute`: Obtains UART configuration parameters.
- `hi_uart_is_buf_empty`: Queries whether the UART FIFO and software buffer are empty.
- `hi_uart_is_busy`: Queries whether the UART is busy.
- `hi_uart_quit_read`: Quits data read in blocking mode.



- `hi_uart_lp_save`: Saves the content of the UART registers before deep sleep.
- `hi_uart_lp_restore`: Restores the content of the UART registers after deep sleep wakeup.

1.3 Development Guidance

The following describes the data transmit (TX) and receive (RX) process by using UART2 as an example.

- Step 1** Configure I/O multiplexing, that is, multiplex the corresponding I/O as the TX, RX, RTS, or CTS function of the UART.

If hardware flow control is not required, configure only TX and RX.

```
hi_void usr_uart_io_config()
{
    /* The following I/O multiplexing configurations can be performed in the app_io_init function
    of the SDK. */
    hi_io_set_func(HI_IO_NAME_GPIO_11, HI_IO_FUNC_GPIO_11_UART2_TXD); /* uart2 tx */
    hi_io_set_func(HI_IO_NAME_GPIO_12, HI_IO_FUNC_GPIO_12_UART2_RXD); /* uart2 rx */
}
```

- Step 2** Configure the UART attributes such as the baud rate and data bits, and enable the UART to initialize the UART.

```
hi_u32 usr_uart_io_config()
{
    hi_u32 ret;
    static hi_uart_attribute g_demo_uart_cfg = {115200, 8, 1, 2, 0};
    ret = hi_uart_init(HI_UART_IDX_2, &g_demo_uart_cfg, HI_NULL);
    if (ret != HI_ERR_SUCCESS) {
        printf("uart init fail\r\n");
    }
    return ret;
}
```

- Step 3** Call the UART read/write data API to transmit or receive UART data.

```
hi_void usr_uart_read_data()
{
    hi_s32 len;
    hi_u8 ch[64] = { 0 };
    len = hi_uart_read(HI_UART_IDX_2, ch, 64);
    if (len > 0) {
        /* process data */
    }
}

hi_u32 usr_uart_write_data(hi_u8 *data, hi_u32 data_len)
{
    hi_u32 offset = 0;
    hi_s32 len = 0;
    while (offset < data_len) {
        len = hi_uart_write(HI_UART_IDX_2, data + offset, (hi_u32)(data_len - offset));
        if ((len < 0) || (0 == len)) {
            return -1;
        }
        offset += (hi_32)len;
        if (offset >= data_len) {
            break;
        }
    }
}
```



```
    return HI_ERR_SUCCESS;  
}
```

----End

1.4 Precautions

- In the SDK, UART1 is used as the AT command channel by default, and GPIO5/6 is multiplexed as the TX/RX function of the UART.
- In the SDK, UART0 is used as the data channel for program burning, maintenance, and test by default, and GPIO3/4 is multiplexed as the TX/RX function of the UART. To use UART0 as a different function, the **hi_diag_init** function in **app_main** can be shielded.
- UART0 does not support hardware flow control.
- **hi_uart_lp_restore** and **hi_uart_lp_save** are used in the deep sleep wakeup and sleep processes to ensure that the UART can restore the configuration before sleep after wakeup.



2 SPI

[2.1 Overview](#)

[2.2 Function Description](#)

[2.3 Development Guidance](#)

[2.4 Precautions](#)

2.1 Overview

The serial peripheral interface (SPI) can act as the master or slave to perform synchronous serial communication with peripheral devices (which must support the SPI frame format). The chip provides two SPI units. SPI0 has a TX/RX FIFO with the bit width of 16 bits x 256, and SPI1 has a TX/RX FIFO with the bit width of 16 bits x 64.

2.2 Function Description

NOTE

If the SPI driver needs to support DMA data transmission and reception, ensure that the DMA driver has been initialized.

The SPI module provides the following APIs:

- `hi_spi_init`: Initializes the SPI, including setting the master/slave device, polarity, phase, frame protocol, transfer frequency, and transfer bit width.
- `hi_spi_deinit`: Deinitializes the SPI module, such as disabling the corresponding SPI unit to release resources.
- `hi_spi_set_basic_info`: Sets the SPI parameters, such as polarity, phase, frame protocol, transfer bit width, and frequency.
- `hi_spi_host_writeread`: Transmits and receives data in full-duplex mode when the SPI acts as the master.
- `hi_spi_host_read`: Receives data in half-duplex mode when the SPI acts as the master.



- `hi_spi_host_write`: Transmits data in half-duplex mode when the SPI acts as the master.
- `hi_spi_slave_read`: Receives data in half-duplex mode when the SPI acts as the slave.
- `hi_spi_slave_write`: Transmits data in half-duplex mode when the SPI acts as the slave.
- `hi_spi_set_irq_mode`: Sets whether to transfer data in interrupt mode. If the interrupt mode is not configured when the SPI acts as the master, the polling mode is used by default. When the SPI acts as the slave, data is transferred in interrupt mode by default.
- `hi_spi_set_dma_mode`: Sets whether to transfer data in DMA mode for the slave device.
- `hi_spi_register_usr_func`: Registers the user preparation/restoration function.
- `hi_spi_set_loop_back_mode`: Sets whether to enable the loopback test mode.

2.3 Development Guidance

The SPI is used to connect to the device that supports the SPI protocol. The SPI unit can be used as the master or slave device. The following describes how to write data by using the SPI unit as the master device:

Step 1 Multiplex the required I/O pins as the SPI function.

- SPI0 provides two groups of multiplexed pins.
- SPI1 provides one group of multiplexed pins.

Step 2 Initialize the SPI resources by calling **`hi_spi_init`**, select the SPI functional unit, and configure SPI parameters.

```
hi_u32 usr_spi_init()
{
    hi_u32 ret;
    /* Select an SPI unit. */
    hi_spi_idx id = 0;
    /* I/O multiplexing */
    hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_SPI0_CSN);
    hi_io_set_func(HI_IO_NAME_GPIO_6, HI_IO_FUNC_GPIO_6_SPI0_CK);
    hi_io_set_func(HI_IO_NAME_GPIO_7, HI_IO_FUNC_GPIO_7_SPI0_RXD);
    hi_io_set_func(HI_IO_NAME_GPIO_8, HI_IO_FUNC_GPIO_8_SPI0_TXD);
    hi_io_set_driver_strength(HI_IO_NAME_GPIO_8, HI_IO_DRIVER_STRENGTH_0);
    hi_spi_cfg_init_param init_param;
    /* Set the master mode. */
    init_param.is_slave = HI_FALSE;
    hi_spi_cfg_basic_info spi_cfg;
    /* Configure SPI parameters. */
    spi_cfg.cpha = 0; /* Polarity */
    spi_cfg.cpol = 0;
    spi_cfg.data_width = 7; /* Data bit width */
    spi_cfg.endian = 0; /* Little endian */
    spi_cfg.fram_mode = 0; /* Frame protocol */
    spi_cfg_basic_info.freq = 8000000; /* Frequency */
    ret = hi_spi_init(id, init_param, &spi_cfg);
    return ret;
}
```

Step 3 Write to the SPI master device by calling **`hi_spi_host_write`**.



For example, send data to the master device.

```
hi_u32 demo_spi_host_write_task()
{
    hi_u32 ret;
    hi_spi_idx id = 0; /* SPI unit select */
    hi_u8 send_buf[BUF_LEN]; /* Data to be transmitted */
    /* Process data. */
    ret = hi_spi_host_write(id, send_buf, BUF_LEN);
    /* Judge and process errors. */
    return ret;
}
```

----End

2.4 Precautions

- If the rate of a slave device is low when the chip functions as a master device, the master device delays each time the read/write API is called to avoid data errors caused by slow data read/write of the slave device.
- When the SPI is no longer used, **hi_spi_deinit** must be called to release resources. Otherwise, the error code **HI_ERR_SPI_REINIT** is returned during initialization.
- When **hi_spi_set_basic_info** is used to reset parameters, the returned status code must be checked. The parameters cannot be reset when the SPI is transferring data. Otherwise, the error code **HI_ERR_SPI_BUSY** is returned.
- When the SPI unit works as a slave device, the data transfer does not support the polling mode. When the SPI unit works as a master device, the data transfer does not support the DMA mode.
- When the SPI unit works as a slave device, the data transfer supports only the half-duplex mode.
- When the Microwire protocol is used, the master device can transmit only 8-bit data due to the protocol restrictions.



3 I²C

[3.1 Overview](#)

[3.2 Function Description](#)

[3.3 Development Guidance](#)

[3.4 Precautions](#)

3.1 Overview

The I²C module is a slave device on the Advanced Peripheral Bus (APB) while a master device on the I²C bus. It is used by the CPU to read and write data from a slave device on the I²C bus.

3.2 Function Description

The I²C module provides the following APIs:

- `hi_i2c_init`: Initializes the I²C module, including configuring interrupts and setting SCI high or low.
- `hi_i2c_deinit`: Deinitializes the I²C module, including clearing interrupts and resetting the I²C status.
- `hi_i2c_register_reset_bus_func`: Registers the I²C callback function, which is used for future extension.
- `hi_i2c_set_baudrate`: Sets the I²C baud rate.
- `hi_i2c_write`: Transmits I²C data.
- `hi_i2c_read`: Receives I²C data.
- `hi_i2c_writeread`: Transmits and receives I²C data in dual-line mode.



3.3 Development Guidance

The I²C module can read and write the electrically erasable programmable read-only memory (EEPROM) and the ADC-DAC mono audio decoder chip ES8311. The following describes how to read the EEPROM.

Step 1 Multiplex required I/O pins as the I²C function.

Step 2 Initialize the I²C resources by calling **hi_i2c_init**, select the I²C hardware device, and configure the baud rate.

```
hi_void sample_i2c_init(hi_i2c_idx id)
{
    /* I/O multiplexing */
    /* ... */

    hi_i2c_init(id, baudrate);
}
```

Step 3 Enable the master device to transmit data to the slave device EEPROM by calling **hi_i2c_write**. In addition, set the device address length to 16 bits so that 10-bit components may be supported in the future.

```
hi_u32 sample_i2c_write(hi_i2c_idx id, hi_16 device_addr, hi_u32 send_len)
{
    hi_u32 status;
    hi_i2c_data sample_data = { 0 };
    sample_data.send_buf = sample_send_buf;
    sample_data.send_len = send_len;
    status = hi_i2c_write(id, eeprom_addr, &sample_data);
    if (status != HI_ERR_SUCCESS) {
        return status;
    }
    return HI_ERR_SUCCESS;
}
```

----End

3.4 Precautions

- The **hi_set_baudrate** function is called after initialization to set the baud rate. If this function is directly called without the initialization of the I²C module, an error will be returned.
- Ensure that the data TX pointer **send_buf** and data RX pointer **receive_buf** are not null.
- If the data to be sent is beyond the acceptable range of the peer device, the sending fails. In addition, if the I²C device is switched to another I²C device to continue sending after an error is sent, the bus is suspended, and all I²C devices cannot correctly send data.



4 I²S

- [4.1 Overview](#)
- [4.2 Function Description](#)
- [4.3 Development Guidance](#)
- [4.4 Precautions](#)

4.1 Overview

The I²S module is used to transfer data between audio devices.

4.2 Function Description

The I²S module has the following features:

- Supports TX and RX only in I²S master mode. The master clock (MCLK) of I²S supports only 12.288 MHz (the error does not exceed 10 Hz).
- Complies with the Philips I²S protocol and supports only the normal I²S mode.
- Supports 16-bit or 24-bit sampling.
- Supports the sampling rates of 8 kHz, 16 kHz, 32 kHz, and 48 kHz (SCK frequency = 2 x Sampling frequency x Number of sampling bits).
- Supports audio-left and audio-right channels.
- Supports the DMA mode.

NOTE

Ensure that the DMA driver has been initialized before using the APIs related to the I²S module.

The I²S module provides the following APIs:

- `hi_i2s_init`: Initializes the I²S.
- `hi_i2s_deinit`: Deinitializes the I²S.
- `hi_i2s_write`: Transmits data.



- `hi_i2s_read`: Receives data.

4.3 Development Guidance

The I²S interface is used for data transmission between audio devices. The following describes how to use the I²S interface by using the audio codec ES8311 to play audio as an example:

- Step 1** Set the corresponding pins to the I²S and I²C functions through the I/O multiplexed interface. The I²C is used to initialize the audio codec ES8311.
- Step 2** Initialize the ES8311 by calling `hi_i2c_write`.
- Step 3** Initialize the I²S resources by calling `hi_i2s_init` and configure the I²S sampling rate and sampling precision.
- Step 4** Copy the audio data from the flash memory to the RAM by calling `hi_flash_read`.
- Step 5** Transmit the audio data in the RAM to the ES8311 by calling `hi_i2s_write`, and then play the audio by using the headset.

----End

Sample:

```
#define SAMPLE_RATE_8K      8
#define SAMPLE_RESOLUTION_16BIT 16
#define SAMPLE_PLAY_AUDIO   0
#define AUDIO_PLAY_BUF_SIZE 4096

typedef struct {
    hi_u32 flash_start_addr;
    hi_u32 data_len;
} sample_audio_map;

typedef struct {
    hi_u8 *play_buf;
    hi_u8 *record_buf;
} sample_audio_attr;

/* Burn the audio data less than 100 KB to the 0x001A1000 address of the flash memory in advance. */
sample_audio_map g_sample_audio_map[2] = {
    {0x001A1000, 100 * 1024},
    {0x001CE000, 204800},
};

sample_audio_attr g_audio_sample;

hi_u32 es8311_init(hi_codec_attribute *codec_attr)
{
    hi_u32 ret;

    if (codec_attr == HI_NULL) {
        return HI_ERR_FAILURE;
    }

    /*In this sample, I2C1 is used to initialize the ES8311. During I/O multiplexing configuration,
    select the pins that support the I2C1 function.*/
```



```
ret = hi_i2c_init(HI_I2C_IDX_1, 100000); /* 100000:baudrate */
if (ret != HI_ERR_SUCCESS) {
    printf("===ERROR=== failed to init i2c!!, err = : %X\r\n", ret);
    return ret;
}

ret = hi_codec_init(codec_attr);
if (ret != HI_ERR_SUCCESS) {
    printf("===ERROR=== failed to init es8311!!, err = : %X\r\n", ret);
    return ret;
}

printf("----SUCCESS---init codec=====\r\n");
return HI_ERR_SUCCESS;
}

hi_u32 sample_audio_play(hi_u32 map_index)
{
    hi_u32 ret;
    hi_u32 play_addr = g_sample_audio_map[map_index].flash_start_addr;
    hi_u32 total_play_len = g_sample_audio_map[map_index].data_len;
    hi_u32 send_len = 0;
    hi_u32 time_out = HI_SYS_WAIT_FOREVER;
    hi_u32 start_time, end_time;

    g_audio_sample.play_buf = (hi_u8 *) hi_malloc(HI_MOD_ID_DRV, AUDIO_PLAY_BUF_SIZE);
    if (g_audio_sample.play_buf == HI_NULL) {
        hi_i2s_deinit();
        printf("===ERROR== Failed to malloc play buf!!\n");
        return HI_ERR_MALLOC_FAILUE;
    }
    memset_s(g_audio_sample.play_buf, AUDIO_PLAY_BUF_SIZE, 0, AUDIO_PLAY_BUF_SIZE);

    while (total_play_len > 0) {
        send_len = hi_min(total_play_len, AUDIO_PLAY_BUF_SIZE);
        ret = hi_flash_read(play_addr, send_len, g_audio_sample.play_buf);
        if (ret != HI_ERR_SUCCESS) {
            printf("===ERROR== Falied to read flash!! err = %X\n", ret);
        }

        ret = hi_i2s_write(g_audio_sample.play_buf, send_len, time_out);
        if (ret != HI_ERR_SUCCESS) {
            printf("Failed to write data\n");
        }

        play_addr += send_len;
        total_play_len -= send_len;
    }
    printf("Play over....\n");

    hi_free(HI_MOD_ID_DRV, g_audio_sample.play_buf);
    return HI_ERR_SUCCESS;
}

hi_void sample_i2s(hi_void)
{
    hi_u32 ret;
    hi_i2s_attribute i2s_cfg;
    hi_codec_attribute codec_cfg;
    hi_u32 sample_rate = SAMPLE_RATE_8K;
    hi_u32 resolution = SAMPLE_RESOLUTION_16BIT;
```



```
codec_cfg.sample_rate = (hi_codec_sample_rate) sample_rate;
codec_cfg.resolution = (hi_codec_resolution) resolution;
i2s_cfg.sample_rate = (hi_i2s_sample_rate) sample_rate;
i2s_cfg.resolution = (hi_i2s_resolution) resolution;

ret = es8311_init(&codec_cfg);
if (ret != HI_ERR_SUCCESS) {
    return;
}

ret = hi_i2s_init(&i2s_cfg);
if (ret != HI_ERR_SUCCESS) {
    printf("Failed to init i2s!\n");
    return;
}
printf("I2s init success!\n");

ret = sample_audio_play(SAMPLE_PLAY_AUDIO);
if (ret != HI_ERR_SUCCESS) {
    printf("Failed to play audio!\n");
}
}
```

4.4 Precautions

When using the RX and TX APIs, shorten the interval between the previous and next calling of **hi_i2s_write** or **hi_i2s_read** if possible.



5 LSADC

[5.1 Overview](#)

[5.2 Function Description](#)

[5.3 Development Guidance](#)

[5.4 Precautions](#)

5.1 Overview

The low-speed analog-to-digital converter (LSADC) is used to convert external analog signals into digital values in a certain proportion for measurement. It can be used for power detection and keypress detection. The scanning frequency of the LSADC is less than or equal to 166 kbit/s.

5.2 Function Description

The LSADC module provides the following APIs:

- `hi_adc_read`: Reads data from a specified ADC channel.

5.3 Development Guidance

The chip provides one LSADC with eight independent channels. It reads a piece of data from a specified channel by calling `hi_adc_read`. Channel 7 provides the reference voltage and cannot be used for ADC conversions.

Conversion relationship between the code word and the input voltage (V): Code word = $V/4/1.8 \times 4096$

For example, to read the voltage of channel 7 of the LSADC, perform the following steps:

Step 1 Read the voltage of channel 7 by calling `hi_adc_read`.

----End

Sample:



```
hi_u32 adc_test(hi_void)
{
    hi_u32 ret;
    hi_u16 data = 0;
    hi_float voltage = 0.0;
    ret = hi_adc_read(HI_ADC_CHANNEL_7, &data, HI_ADC_EQU_MODEL_4,
HI_ADC_CUR_BAIS_DEFAULT, 0);
    if (ret == HI_ERR_SUCCESS) {
        voltage = hi_adc_convert_to_voltage(data);
    }
    return ret;
}
```

5.4 Precautions

- Before starting LSADC scanning, ensure that the previous scanning has been stopped.
- LSADC channel 7 detects only internal VBAT voltage rather than input voltage through pins.
- The value range of the **delay_cnt** parameter in the **hi_adc_read** interface is [0, 0xFF0]. From sampling configuration to sampling start, the count is 334 ns.



6 IO/GPIO

[6.1 Overview](#)

[6.2 Function Description](#)

[6.3 Development Guidance](#)

[6.4 Precautions](#)

6.1 Overview

The I/O driver supports the configuration of the I/O drive strength, multiplexing I/O pins as peripheral pins, and setting the pull-up/pull-down status.

The GPIO driver supports the functions such as setting the direction of the GPIO pin, setting the output level, and reporting interrupts.

6.2 Function Description

The I/O module provides the following APIs:

- `hi_io_set_pull`: Sets the pull-up or pull-down of an I/O.
- `hi_io_get_pull`: Obtains the I/O pull-up or pull-down of an I/O.
- `hi_io_set_func`: Sets the multiplexing function of an I/O.
- `hi_io_get_func`: Obtains the multiplexing function of an I/O.
- `hi_io_set_driver_strength`: Sets the drive strength of an I/O.
- `hi_io_get_driver_strength`: Obtains the drive strength of an I/O.

The GPIO module provides the following APIs:

- `hi_gpio_init`: Initializes the GPIO module.
- `hi_gpio_deinit`: Deinitializes the GPIO module.
- `hi_gpio_set_dir`: Sets the direction of a GPIO pin.
- `hi_gpio_get_dir`: Obtains the direction of a GPIO pin.
- `hi_gpio_set_output_val`: Sets the output level of a GPIO pin.



- `hi_gpio_get_output_val`: Obtains the output level of a GPIO pin.
- `hi_gpio_get_input_val`: Obtains the input level of a GPIO pin.
- `hi_gpio_register_isr_function`: Enables the interrupt function of a GPIO pin.
- `hi_gpio_unregister_isr_function`: Disables the interrupt function of a GPIO pin.
- `hi_gpio_set_isr_mask`: Sets the interrupt mask enable status of a GPIO pin.
- `hi_gpio_set_isr_mode`: Sets the interrupt trigger mode of a GPIO pin.

6.3 Development Guidance

The I/O interface and GPIO interface should be used independently. Perform the following steps:

- Step 1** Determine whether a pin is used as GPIO.
- Step 2** If a pin is used as GPIO, set the direction of the GPIO pin and the rising/falling edge interrupt, as shown in **sample_gpio**.
- Step 3** If a pin is multiplexed as a peripheral drive pin, set the multiplexing function and drive strength (optional), as shown in **sample_io**.

----End

Sample:

```
hi_void sample_gpio
{
    hi_gpio_init();
    hi_gpio_dir(sample_gpio_id, sample_gpio_dir_in);
    hi_gpio_register_isr_func(sample_gpio_id, sample_type_level, sample_level_high,
sample_func);
    hi_gpio_deinit();
}

hi_void sample_io
{
    hi_io_set_func(HI_IO_NAME_GPIO_9, HI_IO_FUNC_GPIO_9_SPIO_TXD);
    hi_io_set_driver_strength(HI_IO_NAME_GPIO_9, HI_IO_DRIVER_STRENGTH_0);
}
```

6.4 Precautions

- When configuring the I/O multiplexing function, ensure that the I/O has not been multiplexed as other functions to avoid impacts on existing functions.
- **`hi_gpio_set_isr_mask`** and **`hi_gpio_set_isr_mode`** are valid only after **`hi_gpio_register_isr_function`**. If they are used before **`hi_gpio_register_isr_function`**, the corresponding configurations will be overwritten by the configurations in **`hi_gpio_register_isr_function`**.



7 eFuse

[7.1 Overview](#)

[7.2 Function Description](#)

[7.3 Development Guidance](#)

[7.4 Precautions](#)

7.1 Overview

The eFUSE is a programmable storage unit. Since it can be programmed only once, it is used to store the chip ID, key, or other one-time data in most cases. Hi3861 provides a 2 KB eFUSE (offset address: 0x000–0x7FF). The eFUSE is divided into several fields based on the usage. One 256-bit field and one 64-bit field are reserved for users and can be further subdivided.

7.2 Function Description

The eFUSE provides two sets of read and write APIs:

- The eFUSE fields are numbered sequentially by start address to facilitate indexing.
 - `hi_efuse_get_id_size`: Obtains the length (in bits) of an eFUSE field by field ID.
 - `hi_efuse_read`: Reads data from an eFUSE field by field ID.
 - `hi_efuse_write`: Writes data to an eFUSE field by field ID.
 - `hi_efuse_lock`: Locks an area in the eFUSE by using the lock ID. After the area is locked, data cannot no longer be written to it.
 - `hi_efuse_get_lockstat`: Obtains the lock status of the eFUSE and queries which areas are locked.
- Read/Write mode of the eFUSE start address, which is used for reserved areas:
 - `hi_efuse_usr_read`: Reads the eFUSE from a specified start address.
 - `hi_efuse_usr_write`: Writes the eFUSE into a specified start address.



For details about the eFUSE registers, see section 5.7 "eFUSE" in the *Hi3861 V100/Hi3861L V100/Hi3881 V100 Wi-Fi Chip Data Sheet*. **Table 7-1** describes the mapping between the eFUSE ID, lock ID, and eFUSE field in software.

Table 7-1 Mapping between eFUSE fields and eFUSE IDs

Field	eFUSE ID in Software	Lock ID in Software
chip_id	0	-
die_id	1	-
pmu_fuse1	2	-
pmu_fuse2	3	-
flash_encpt_cnt[7:6]	4	-
flash_encpt_cnt[9:8]	5	-
flash_encpt_cnt[11:10]	6	-
secure_boot	52	-
deep_sleep_flag	7	-
PG36	-	36
PG37	-	37
PG38	-	38
PG39	-	39
PG40	-	40
root_pubkey	8	-
root_key	9	-
customer_rsvd0	10	-
subkey_cat	11	-
encrypt_flag	12	-
rsim	13	-
start_type	14	-
jtm	15	-
utm0	16	-
utm1	17	-
utm2	18	-
sdc	19	-
rsvd0	20	-



Field	eFUSE ID in Software	Lock ID in Software
kdf2ecc_huk_disable	21	-
SSS_corner	22	-
uart_halt_interval	23	-
ts_trim	24	-
chip_id2	25	-
ipv4_mac_addr	26	-
ipv6_mac_addr	27	-
pa2gccka0_trim0	28	-
pa2gccka1_trim0	29	-
nvrkam_pa2ga0_trim0	30	-
nvrkam_pa2ga1_trim0	31	-
pa2gccka0_trim1	32	-
pa2gccka1_trim1	33	-
nvrkam_pa2ga0_trim1	34	-
nvrkam_pa2ga1_trim1	35	-
pa2gccka0_trim2	36	-
pa2gccka1_trim2	37	-
nvrkam_pa2ga0_trim2	38	-
nvrkam_pa2ga1_trim2	39	-
tee_boot_ver	40	-
tee_firmware_ver	41	-
tee_salt	42	-
flash_encpt_cnt[1:0]	43	-
flash_encpt_cnt[3:2]	44	-
flash_encpt_cnt[5:4]	45	-
flash_encpt_cfg	46	-
flash_scramble_en	47	-
user_flash_ind	48	-
rf_pdbuffer_gcal	49	-
customer_rsvd1	50	-



Field	eFUSE ID in Software	Lock ID in Software
die_id2	51	-
PG0	-	0
PG1	-	1
PG2	-	2
PG3	-	3
PG4	-	4
PG5	-	5
PG6	-	6
PG7	-	7
PG8	-	8
PG9	-	9
PG10	-	10
PG11	-	11
PG12	-	12
PG13	-	13
PG14	-	14
PG15	-	15
PG16	-	16
PG17	-	17
PG18	-	18
PG19	-	19
PG20	-	20
PG21	-	21
PG22	-	22
PG23	-	23
PG24	-	24
PG25	-	25
PG26	-	26
PG27	-	27
PG28	-	28



Field	eFUSE ID in Software	Lock ID in Software
PG29	-	29
PG30	-	30
PG31	-	31
PG32	-	32
PG33	-	33
PG34	-	34
PG35	-	35

7.3 Development Guidance

The eFUSE is used to store recovery data or chip information. Perform the following steps:

Step 1 Write data to the eFUSE by calling **hi_efuse_write** or **hi_efuse_usr_write**.

Step 2 Read data from the eFUSE by calling **hi_efuse_read** or **hi_efuse_usr_read**.

Step 3 Lock an area in the eFUSE by calling **hi_efuse_lock** or **hi_efuse_usr_write**. After the area is locked, data cannot be written to it.

----End

Sample 1: ID20 corresponds to the read, write, and lock operations on eFUSE fields.

```
hi_u32 sample_efuse(void)
{
    hi_u8 read_data;
    hi_u8 write_data = 0x55;
    hi_u32 ret;
    hi_efuse_idx efuse_id = HI_EFUSE_RSVD0_RW_ID;
    hi_efuse_lock_id lock_id = HI_EFUSE_LOCK_RSVD0_ID;

    ret = hi_efuse_write(efuse_id, &write_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to write EFUSE!\n");
        return ret;
    }

    ret = hi_efuse_read(efuse_id, &read_data, (hi_u8)sizeof(read_data));
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to read EFUSE!\n");
        return ret;
    }

    printf("data = 0x%02X\n", data);

    ret = hi_efuse_lock(lock_id);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to lock EFUSE!\n");
    }
}
```



```
        return ret;
    }

    return HI_ERR_SUCCESS;
}
```

Sample 2: Read, write, and lock operations in the reserved area

```
#define EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN 2 /* The length of customer_rsvd1 is 64 bits.
Two 32-bit spaces are required for storing the read and write data. */
hi_u32 sample_usr_efuse(void)
{
    hi_u32 ret;
    hi_u32 read_data[EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN] = {0};
    hi_u32 write_data[EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN] = {
        0x55555555,
        0x55555555,
    };
    hi_u8 lock_data = 0x1;
    hi_u16 start_bit = 0x75C; /* The offset address of customer_rsvd1 is 0x75C. */
    hi_u16 rw_bits = 64; /* The length of customer_rsvd1 is 64 bits. */
    hi_u16 lock_start_bit = 0x7FD; /* The lock offset address of customer_rsvd1 is 0x7FD. */
    hi_u16 lock_bits = 1; /* The lock length of customer_rsvd1 is 1 bit. */

    ret = hi_efuse_usr_write(start_bit, rw_bits, (hi_u8 *)write_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to write EFUSE!\n");
        return ret;
    }

    ret = hi_efuse_usr_read(start_bit, rw_bits, (hi_u8 *)read_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to read EFUSE!\n");
    }

    printf("read_data = %08X %08X\n", read_data[1], read_data[0]);

    ret = hi_efuse_usr_write(lock_start_bit, lock_bits, &lock_data);
    if (ret != HI_ERR_SUCCESS) {
        printf("Failed to lock EFUSE!\n");
        return ret;
    }

    return HI_ERR_SUCCESS;
}
```

7.4 Precautions

- When the eFuse is read, the argument **data** is used to store the read data. Ensure that the **data** space is larger than the size of the data to be read.
- It is recommended that data be read from or written to a random start address of the eFuse based on the address in the user-reserved area. For other areas, data should be written based on the ID.
- When **hi_efuse_usr_read** is used, the input start address must be 8-bit aligned. If you input the number of bits to be read that is not 8-bit aligned, the function internally handles it as 8-bit aligned. After data is read, the data needs to be shifted rightwards. For example, if you input 10 as the number of



bits to be read, the read data is the data after being shifted rightwards by six bits.

- The lock PG31 (offset address: 0x7FB, length: 1 bit) and its corresponding eFuse field (offset address: 0x73F, length: 11 bits) take effect immediately after being written. Other eFuse areas take effect only after successful data writing and restart.



8 Flash

[8.1 Overview](#)

[8.2 Function Description](#)

[8.3 Development Guidance](#)

[8.4 Precautions](#)

8.1 Overview

The flash module supports read, write, and erase operations on the flash memory.

8.2 Function Description

The flash module provides the following APIs:

- `hi_flash_init`: Initializes the flash module (typically during system boot).
- `hi_flash_deinit`: Deinitializes the flash module.
- `hi_flash_ioctl`: Obtains flash information.
- `hi_flash_erase`: Erases the data in a specified flash partition.
- `hi_flash_write`: Writes data to a specified flash partition.
- `hi_flash_read`: Reads flash data to a specified cache.

8.3 Development Guidance

The flash module is used to read and write the flash memory. The following flash memories are supported: W25Q16JL, W25Q16JW, GD25WQ16, EN25S16, and EN25QH16. The following describes how to read data from and write data to the flash memory by using the flash module as an example:

Step 1 Write data to the flash memory by calling **`hi_flash_write`**. Read data from the flash memory by calling **`hi_flash_read`**.

```
hi_u32 flash_demo()  
{  
    hi_u32 ret;
```



```
hi_bool do_erase = HI_FLASE; /* Whether to erase the flash */
hi_u32 flash_offset = 0x10000; /* Flash offset address */
hi_u32 size = 32; /* Length of the data to be read from or written to the flash memory */
hi_u8 ram_data[32]; /* Buffer of the read/write data */

/* Fill ram_data and flash_offset size do_erase. */
/* Write the flash memory. */
ret = hi_flash_write(flash_offset, size, ram_data, do_erase);
if (ret != HI_ERR_SUCCESS) {
    /* Handle an error. */
}
memset_s(ram_data, DATA_LEN, 0, DATA_LEN); /* Clear the buffer. */
/* Read the flash memory. */
ret = hi_flash_read(flash_offset, size, ram_data);
if (ret != HI_ERR_SUCCESS) {
    /* Handle an error. */
}

return ret;
}
```

----End

8.4 Precautions

When **hi_flash_erase** is used, the erase address and length must be 4 KB aligned.



9 SDIO

[9.1 Overview](#)

[9.2 Function Description](#)

[9.3 Development Guidance](#)

[9.4 Precautions](#)

9.1 Overview

As a slave device, the SDIO module is used to transmit data or messages with the SDIO master device.

9.2 Function Description

The SDIO module has the following features:

- The working clock of the SDIO module of the master device must be less than or equal to 50 MHz.
- DMA must be used for data transmission.

The SDIO module provides the following APIs:

- `hi_sdio_init`: Initializes the SDIO module.
- `hi_sdio_reinit`: Re-initializes the SDIO module.
- `hi_sdio_register_callback`: Registers the SDIO interrupt handling function. The interrupt types are as follows:
 - Detect that the host initiates a data read.
 - Detect that the host has finished a data read.
 - Detect that the host read data is incorrect.
 - Detect that the host initiates a data write.
 - Detect that the host has finished a data write.
 - Receive a message sent by the host.
 - Receive a soft reset interrupt initiated by the host.



- `hi_sdio_soft_reset`: Soft resets the SDIO module.
- `hi_sdio_complete_send`: Stops transmitting data. 4-byte data is padded into the ADMA linked list.
- `hi_sdio_set_pad_admatab`: Pads data of a specified length into the ADMA linked list.
- `hi_sdio_write_extinfo`: Writes the SDIO extension configuration information.
- `hi_sdio_send_data`: Sends data of a specified length. The data needs to be padded into the ADMA linked list when a read data interrupt initiated by the host is received.
- `hi_sdio_set_admatable`: Pads specified data into the ADMA linked list.
- `hi_sdio_sched_msg`: Sends a message suspended in the message queue.
- `hi_sdio_send_sync_msg`: Adds a specified message to a message queue and sends the message.
- `hi_sdio_send_msg_ack`: Sends a specified message immediately. The message that is being sent will be overwritten.
- `hi_sdio_process_msg`: Clears a specified message suspended in the message queue, adds a new message to the message queue, and sends the message.
- `hi_sdio_get_extend_info`: Obtains the SDIO extension information.
- `hi_sdio_is_pending_msg`: Specifies whether a message is in a suspended message queue.
- `hi_sdio_is_sending_msg`: Specifies whether a message is being sent.
- `hi_sdio_register_notify_message_callback`: Registers the callback function for sending data or messages. For example, the host can be notified through the GPIO when data or messages are sent.
- `hi_sdio_set_powerdown_when_deep_sleep`: Sets whether the SDIO is powered down in deep sleep mode. The default setting is power-down.

9.3 Development Guidance

The following describes the process of using the SDIO interface by taking simple data or message transmission and reception as an example:

- Step 1** In `app_io_init`, configure the corresponding pins as the SDIO function by using the I/O multiplexed interface. For example, multiplex I/O9 to I/O14 as the DATA, CMD, and CLK signals of the SDIO module, and pull the DATA signal high.
- Step 2** Initialize the SDIO module by calling `hi_sdio_init`.
- Step 3** Register the SDIO interrupt handling callback function by calling `hi_sdio_register_callback`.
- Step 4** Send data by calling `hi_sdio_send_data`. When the interrupt handling function that the host starts to read data is detected, the address of the data to be sent is padded in the ADMA linked list.
- Step 5** Receive data sent by the host. When the interrupt handling function that the host starts to write data is detected, the address of the received data is padded in the ADMA linked list. When the completion of data write by the host is detected, the host finishes data transmission.



Step 6 Send a message by calling `hi_sdio_send_sync_msg`.

Step 7 Receive a message sent by the host. When the interrupt handling function that a message sent by the host is detected, the message is processed based on the message content.

----End

Sample:

```
#define DATA_BLOCK      32768 /* sdio data block size:32768 */
#define SEND_RCV_DATA_SIZE 1024 /* send/recv 1024 byte per cycle */
hi_u32 g_sdio_send_data[SEND_RCV_DATA_SIZE] = {0}; /* data array of data to be send */
hi_u32* g_sdio_send_data_addr = NULL;
hi_u32 g_receive_data[SEND_RCV_DATA_SIZE] = {0}; /* data array to store receive data */
hi_u32* g_receive_data_addr = NULL;
hi_u32 g_receive_data_len = 0;
hi_s32 app_demo_sdio_read_over_callback(hi_void)
{
    printf("app_demo_sdio_read_over_callback\r\n");
    return HI_ERR_SUCCESS;
}

hi_s32 app_demo_sdio_read_start_callback(hi_u32 len, hi_u8* admatable)
{
    hi_watchdog_feed();

    hi_u32 i;
    hi_u32 remain;
    hi_u32 index = 0;
    hi_u32* addr = NULL;
    g_sdio_send_data_addr = &g_sdio_send_data[0];

    for (i = 0; i < (len / DATA_BLOCK); i++) {
        addr = g_sdio_send_data_addr + ((DATA_BLOCK >> 2) * i); /* 2 bits for g_download_addr is
        hi_u32 */
        if (hi_sdio_set_admatable(admatable, index++, addr, DATA_BLOCK) != 0) {
            return HI_ERR_FAILURE;
        }
    }

    remain = len % DATA_BLOCK;
    if (remain != 0) {
        addr = g_sdio_send_data_addr + ((DATA_BLOCK >> 2) * i); /* 2 bits for g_download_addr is
        hi_u32 */
        if (hi_sdio_set_admatable(admatable, index++, addr, remain) != 0) {
            return HI_ERR_FAILURE;
        }
    }

    if (hi_sdio_complete_send(admatable, index) != 0) {
        return HI_ERR_FAILURE;
    }

    hi_cache_flush();
    return (hi_s32) index;
}

hi_s32 app_demo_sdio_write_start_callback(hi_u32 len, hi_u8* admatable)
{
    printf("app_demo_sdio_write_start_callback,len: %d\r\n", len);
    hi_watchdog_feed();
}
```



```
g_receive_data_addr = &g_receive_data[0];
g_recevei_data_len = len;

hi_u32 i;
hi_u32 remain;
hi_u32 index = 0;
hi_u32* addr = NULL;

for (i = 0; i < (len / DATA_BLOCK); i++) {
    addr = g_receive_data_addr + ((DATA_BLOCK >> 2) * i); /* shift 2bits is for hi_u32* reason. */
    if (hi_sdio_set_admatable(admatable, index++, addr, DATA_BLOCK) != 0) {
        return HI_ERR_FAILURE;
    }
}

remain = len % DATA_BLOCK;

if (remain != 0) {
    addr = g_receive_data_addr + ((DATA_BLOCK >> 2) * i); /* shift 2bits is for hi_u32* reason. */
    if (hi_sdio_set_admatable(admatable, index++, addr, remain) != 0) {
        return HI_ERR_FAILURE;
    }
}
if (hi_sdio_complete_send(admatable, index) != 0) {
    return HI_ERR_FAILURE;
}
hi_cache_flush();
return (hi_s32) index;
}

hi_s32 app_demo_sdio_write_over_callback(hi_void)
{
    printf("app_demo_sdio_write_over_callback, len:%d\n", g_recevei_data_len);

    hi_u8* received_data = (hi_u8*)&g_receive_data[0];
    for (hi_u32 i = 0; i < g_recevei_data_len; i++) {
        if (i % 8 == 0) { /* 8:Newline */
            printf ("\r\n");
        }
        printf("0x%x ", received_data[i]);
    }
    return HI_ERR_SUCCESS;
}

hi_void app_demo_sdio_receive_msg_callback(hi_u32 msg)
{
    printf("app_demo_sdio_receive_msg_callback:0x%x\n", msg);
}

hi_void app_demo_sdio_read_err_callback(hi_void)
{
    printf("app_demo_sdio_read_err_callback\n");
}

hi_void app_demo_sdio_soft_rst_callback(hi_void)
{
    printf("app_demo_sdio_soft_rst_callback\r\n");
}

hi_void app_demo_sdio_send_data(hi_void)
{
    printf("app demo sdio start send data\r\n");
}
```



```
hi_u8* send_data = (hi_u8*)&g_sdio_send_data[0];
hi_cipher_trng_get_random_bytes(send_data, SEND_RCV_DATA_SIZE);
hi_sdio_send_data(SEND_RCV_DATA_SIZE);

return;
}

hi_void app_demo_sdio_callback_init(hi_void)
{
hi_sdio_intcallback callback;

callback.rdover_callback = app_demo_sdio_read_over_callback;
callback.rdstart_callback = app_demo_sdio_read_start_callback;
callback.wrstart_callback = app_demo_sdio_write_start_callback;
callback.wrover_callback = app_demo_sdio_write_over_callback;
callback.processmsg_callback = app_demo_sdio_receive_msg_callback;
callback.rderr_callback = app_demo_sdio_read_err_callback;
callback.soft_rst_callback = app_demo_sdio_soft_rst_callback;
(hi_void)hi_sdio_register_callback(&callback);

printf("sdio_slave_test_init success\r\n");
}

hi_void app_demo_sdio_send_msg(hi_void)
{
hi_sdio_send_sync_msg(0);
return;
}

hi_void app_demo_sdio(hi_void)
{
hi_u32 ret;
/* init sdio */
/* should config io in app_io_init first. */
ret = hi_sdio_init();
if (ret != HI_ERR_SUCCESS) {
printf("app demo sdio init fail\r\n");
return;
}

/* register sdio interrupt callbak. */
app_demo_sdio_callback_init();

/* sdio send msg */
app_demo_sdio_send_msg();

/* sdio receive msg */
/* when host send msg to device, device will receive msg in
app_demo_sdio_receive_msg_callback*/

/* sdio send data */
app_demo_sdio_send_data();

/* sdio receive data */
/* when host send data to device, device will receive data in
app_demo_sdio_write_start_callback, app_demo_sdio_write_over_callback
*/
}
```



9.4 Precautions

- The DCache is enabled by default during system startup, and the SDIO uses the DMA for data transfer. If the data in the memory is inconsistent with that in the cache, the DMA transfer is abnormal. Therefore, you need to flush the DCache before data transfer, or disable the DCache and then enable the SDIO data RX and TX after the system is started.
- If the SDIO host device does not pull the DATA signal high, you need to configure the DATA to high when configuring the I/O multiplexing.



10 Tsensor

[10.1 Overview](#)

[10.2 Function Description](#)

[10.3 Development Guidance](#)

[10.4 Precautions](#)

10.1 Overview

The Tsensor detects the CPU junction temperature, which ranges from -40°C to $+140^{\circ}\text{C}$. After trimming and adjustment, the absolute temperature detection granularity is $\pm 3^{\circ}\text{C}$.

10.2 Function Description

The Tsensor module provides the following APIs:

- `hi_tsensor_start`: Starts the Tsensor module.
- `hi_tsensor_stop`: Stops Tsensor temperature collection.
- `hi_tsensor_destroy`: Disables the interrupt, deletes the registered interrupt callback function, and stops the Tsensor module.
- `hi_tsensor_read_temperature`: Reads the Tsensor temperature in non-interrupt mode.
- `hi_tsensor_set_temp_trim`: Trims the Tsensor temperature by using the register or eFUSE. By default, the temperature is trimmed by using the eFUSE.
- `hi_tsensor_code_to_temperature`: Converts a temperature code into a temperature value.
- `hi_tsensor_temperature_to_code`: Converts a temperature value into a temperature code.
- `hi_tsensor_set_outtemp_threshold`: Sets the high/low temperature threshold and registers the over-temperature interrupt callback function.
- `hi_tsensor_set_overtemp_threshold`: Sets the over-temperature threshold and registers the over-temperature interrupt callback function.



- `hi_tsensor_set_pdtemp_threshold`: Sets the over-temperature power-off protection threshold.
- `hi_tsensor_register_temp_collect_finish_int_callback`: Registers the interrupt callback function for temperature collection completion.

You can set the trim code in the eFUSE or register to trim the internal temperature of the Tsensor. [Table 10-1](#) describes the trimming relationship.

Table 10-1 Configuration table for Tsensor temperature trimming

Trim Code (Binary)	Temperature Code Trim Value	Temperature Trim Value (°C)
0000	default	0.000
0001	+2	1.410
0010	+4	2.820
0011	+6	4.230
0100	+8	5.640
0101	+10	7.050
0110	+12	8.460
0111	+14	9.870
1000	0	0.000
1001	-2	-1.410
1010	-4	-2.820
1011	-6	-4.230
1100	-8	-5.640
1101	-10	-7.050
1110	-12	-8.460
1111	-14	-9.870

Note: The trim code **1000** is a trimming boundary between positive and negative temperatures.

10.3 Development Guidance

The Tsensor monitors the chip temperature and provides hierarchical protection against over-temperature, under-temperature, and power failures. Perform the following steps:

- Step 1** Trim the temperature by calling `hi_tsensor_set_temp_trim`. Skip this step if you trim the temperature by using the eFuse or do not trim the temperature at all.



- Step 2** Set the over-temperature power-off protection threshold by calling **hi_tsensor_set_pdtemp_threshold**.
- Step 3** Set the high/low temperature threshold and register the over-temperature interrupt callback function by calling **hi_tsensor_set_outtemp_threshold**. This step is optional.
- Step 4** Set the over-temperature threshold by calling **hi_tsensor_set_overtemp_threshold** and register the over-temperature interrupt callback function. This step is optional.
- Step 5** Select the temperature reporting mode and start the Tsensor module by calling **hi_tsensor_start**.

----End

Sample:

```
/* The following macro definitions are for reference only. You can define them based on product requirements. */
#define TSENSOR_PERIOD_VALUE 500
#define TSENSOR_TRIM_CODE 0x1
#define LOW_TEMP_THRESHOLD (-40)
#define HIGH_TEMP_THRESHOLD 100
#define OVER_TEMP_THRESHOLD 110
#define PD_TEMP_THRESHOLD 125

hi_void user_outtemp_callback(hi_s16 temperature)
{
    hi_tsensor_stop();

    /* High and low temperature alarm handling */
}

hi_void user_overtemp_callback(hi_s16 temperature)
{
    hi_tsensor_destroy();

    /* Over-temperature alarm handling */
}

hi_u32 sample_tsensor(hi_void)
{
    hi_u32 ret;

    ret = hi_tsensor_set_temp_trim(TSENSOR_TRIM_CODE, 1);
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

    ret = hi_tsensor_set_outtemp_threshold(LOW_TEMP_THRESHOLD, HIGH_TEMP_THRESHOLD,
user_outtemp_callback);
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

    ret = hi_tsensor_set_overtemp_threshold(OVER_TEMP_THRESHOLD, user_overtemp_callback);
    if (ret != HI_ERR_SUCCESS) {
        return ret;
    }

    ret = hi_tsensor_start(HI_TSENSOR_MODE_16_POINTS_SINGLE, TSENSOR_PERIOD_VALUE);
```



```
if (ret != HI_ERR_SUCCESS) {  
    return ret;  
}  
  
return HI_ERR_SUCCESS;  
}
```

10.4 Precautions

- When the Tsensor works in single-point cyclic report mode, only the collection completion interrupt takes effect.
- When the Wi-Fi service is started, the Tsensor is enabled for over-temperature protection. If the Tsensor needs to be used for other purposes, evaluate whether the over-temperature protection function of the Wi-Fi service is affected.
- When the temperature is read in interrupt mode, you are advised to stop the Tsensor by calling **hi_tsensor_stop** in the user interrupt callback. This prevents frequent interrupts from occupying too many CPU resources and preventing other services from being scheduled.
- To use over-temperature protection, stop the over-temperature interrupt by calling **hi_tsensor_destroy** in the interrupt callback. Then, read the temperature in non-interrupt mode to check whether the temperature is too high. If the temperature is too high, generate an alarm and disable some services.



11 DMA

[11.1 Overview](#)

[11.2 Function Description](#)

[11.3 Development Guidance](#)

[11.4 Precautions](#)

11.1 Overview

DMA, short for direct memory access, refers to an operating mode in which I/O data is exchanged by the hardware. In this manner, the direct memory access controller (DMAC) directly transfer data between the memory and a peripheral, between peripherals, and between memories, so as to reduce interference and overheads of the processor. The DMA mode applies to the high-speed transfer of grouped data. After receiving a DMA request, the DMAC enables the master bus controller based on the channel settings configured by the CPU and transfers address and control signals to memories and peripherals. The DMAC also counts the transferred data segments and reports the data transfer status (that is, whether the data transfer is complete or faulty) to the CPU in interrupt mode.

11.2 Function Description

NOTE

If the DMA is used to transmit data in the SPI or UART, or the I²S driver is used, the DMA needs to be initialized during system startup.

The DMA module provides the following APIs:

- `hi_dma_create_link_list`: Creates a DMA transfer linked list.
- `hi_dma_add_link_list_item`: Adds a node to the end of the transfer linked list.
- `hi_dma_link_list_transfer`: Starts the DMA linked list transfer.
- `hi_dma_mem2mem_transfer`: Starts the DMA memory data transfer.
- `hi_dma_ch_close`: Disables a specified DMA channel.



- `hi_dma_init`: Initializes the DMA.
- `hi_dma_deinit`: Deinitializes the DMA.
- `hi_dma_is_init`: Initializes the DMA module.

The APIs for linked list transfer are mainly used to transfer multiple pieces of discontinuous memory data. After the linked list transfer is complete, the specified DMA channel can be disabled based on the allocated channel ID.

11.3 Development Guidance

The DMA interface only provides the function of copying data from memory to memory (other copying modes are integrated into the corresponding peripheral driver). Perform the following steps:

Step 1 Initialize the DMA module by calling `hi_dma_init`.

Step 2 Start the DMA data transfer by calling `hi_dma_mem2mem_transfer`.

Step 3 Deinitialize the DMA module by calling `hi_dma_deinit`, such as releasing the DMA resources. This API can be called only when the DMA function is no longer used.

----End

```
hi_void dma_callback(hi_u32 irq_type)
{
    printf("This is callback,irq type is %d\r\n", irq_type);
}
hi_void user_dma_sample()
{
    hi_u32 ret;
    hi_u32 *src_addr = HI_NULL;
    hi_u32 *dst_addr = HI_NULL;
    hi_32 data_size = 0x1000;
    ret = hi_dma_init();
    if (ret != HI_ERR_SUCCESS) {
        return;
    }

    src_addr = hi_malloc(HI_MOD_ID_DRV_DMA, data_size);
    dst_addr = hi_malloc(HI_MOD_ID_DRV_DMA, data_size);
    memset_s(src_addr, ts, 0x5, data_size);
    memset_s(dst_addr, ts, 0x6, data_size);
    ret = hi_dma_mem2mem_transfer((hi_u32)dst_addr, (hi_u32)src_addr, data_size, HI_TRUE,
dma_callback);
    if (ret != HI_ERR_SUCCESS) {
        printf("dma copy fail, ret = %x\n", ret);
    }

    hi_free(HI_MOD_ID_DRV_DMA, src_addr);
    hi_free(HI_MOD_ID_DRV_DMA, dst_addr);
    hi_dma_deinit();
}
```



11.4 Precautions

- The DMA interrupt callback function is executed in the interrupt context. To call the callback function, you must comply with the programming precautions of the interrupt context.
- In non-blocking mode, you are advised to use the DMA for data copying. In blocking mode, you are advised to use **memcpy_s** for data copying.



12 SysTick

[12.1 Overview](#)

[12.2 Function Description](#)

[12.3 Development Guidance](#)

[12.4 Precautions](#)

12.1 Overview

The SysTick is a system tick timer and provides the following functions:

- Queries the current value of the SysTick counter.
- Clears the SysTick counter.

12.2 Function Description

The SysTick module provides the following APIs:

- `hi_systick_get_cur_tick`: Obtains the current value of the SysTick counter.
- `hi_systick_clear`: Clears the SysTick counter.

12.3 Development Guidance

You can obtain the current count value of the SysTick by calling **`hi_systick_get_cur_tick`**. Each time value is determined by the SysTick clock source. The SysTick clock is 32 kHz, and a tick is 1/32000 seconds. In addition, the latency API is called. Therefore, do not call **`hi_systick_get_cur_tick`** in the interrupt context. **`hi_systick_clear`** clears the count value of the SysTick. However, after the API returns, it waits for three SysTick clock cycles to complete the clearing operation.



12.4 Precautions

None



13 PWM

[13.1 Overview](#)

[13.2 Function Description](#)

[13.3 Development Guidance](#)

[13.4 Precautions](#)

13.1 Overview

The pulse width modulator (PWM) is used to output waveforms and has six ports. I/O multiplexing is required when the PWM port is called.

13.2 Function Description

The PWM module provides the following APIs:

- `hi_pwm_init`: Initializes the PWM module.
- `hi_pwm_deinit`: Deinitializes the PWM module.
- `hi_pwm_set_clock`: Sets the clock type of the PWM module.
- `hi_pwm_start`: Starts the signal output of the PWM module.
- `hi_pwm_stop`: Stops the signal output of the PWM module.

13.3 Development Guidance

The PWM controls the analog circuit by using the digital output of the microprocessor. Perform the following steps:

- Step 1** Initialize the PWM by calling **`hi_pwm_init`** based on the input port number. The PWM can use different clock sources. The working clock is 160 MHz. The external crystal can be 24 MHz or 40 MHz. Different PWM ports need to multiplex different GPIO pins.
- Step 2** Set the clock type of the PWM module by calling **`hi_pwm_set_clock`**. By default, the 150 MHz clock is used.



Step 3 Start the PWM module by calling **hi_pwm_start**, output the PWM signal based on the configured input parameters. The duty cycle of the signal is (duty/freq), and the frequency is (Clock source frequency/freq).

----End

Sample:

```
hi_void sample_pwm(hi_void)
{
    hi_pwm_init(SAMPLE_PWM_PORT);
    hi_pwm_set_clock(SAPMLE_PWM_CLK);
    hi_pwm_start(SAMPLE_PWM_PORT, sample_duty, sample_freq);
}
```

13.4 Precautions

In the PWM, **hi_pwm_stop** cannot be called in an interrupt.



14 WDG

[14.1 Overview](#)

[14.2 Function Description](#)

[14.3 Development Guidance](#)

[14.4 Precautions](#)

14.1 Overview

The watchdog is used to recover the abnormal system. It generates a system reset signal at an interval (which is programmable and can be up to 26s) when not updated, but will not generate a reset signal if the working clock is disabled and the timer is updated.

14.2 Function Description

The watchdog module provides the following APIs:

- `hi_watchdog_enable`: Enables the watchdog.
- `hi_watchdog_feed`: Resets the watchdog timer (sometimes called "kicking" or "feeding" the dog).
- `hi_watchdog_disable`: Disables the watchdog.

14.3 Development Guidance

The watchdog is used to check whether the system is down. If the dog is not kicked after the waiting time, a system reset is generated. Perform the following steps to "kick" the dog:

Step 1 Enable the watchdog module by calling `hi_watchdog_enable`.

Step 2 Kick the dog by calling `hi_watchdog_feed`.



Step 3 Disable the watchdog by calling **hi_watchdog_disable**.

----End

Sample:

```
hi_void test_wdg()
{
    /* Enable the watchdog. */
    hi_watchdog_enable();
    hi_udelay(5000000); /* delay 5000000 us */
    /* Kick the dog. */
    hi_watchdog_feed();
    /* Disable the watchdog. */
    hi_watchdog_disable();
}
```

14.4 Precautions

Ensure that the watchdog has been enabled in the software development kit (SDK) and watchdog feeding exists.