



Hi3861V100 / Hi3861LV100 设备驱动

开发指南

文档版本 02

发布日期 2020-07-03

版权所有 © 上海海思技术有限公司2020。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HISILICON、海思和其他海思商标均为海思技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

上海海思技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： <https://www.hisilicon.com/cn/>

客户服务邮箱： support@hisilicon.com



前言

概述

本文档主要介绍Hi3861V100的设备驱动开发相关内容，包括工作原理、按场景描述接口使用方法和注意事项。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3861	V100
Hi3861L	V100



读者对象

本文档主要适用于以下工程师：




- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	表示如不可避免则将会导致死亡或严重伤害的具有高等级风险的危害。
 警告	表示如不可避免则可能导致死亡或严重伤害的具有中等级风险的危害。



符号	说明
 注意	表示如不可避免则可能导致轻微或中度伤害的具有低等级风险的危害。
 须知	用于传递设备或环境安全警示信息。如不可避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
02	2020-07-03	在“ 6.4 注意事项 ”中新增关于SFC相关的IO复用和配置的注意事项。
01	2020-04-30	第一次正式版本发布。 <ul style="list-style-type: none">在“1.2 功能描述”中新增hi_uart_quit_read接口说明。在“3.4 注意事项”中新增关于发送失败的说明。在“5.3 开发指引”中删除关于数据格式的说明；更新代码示例。在“5.4 注意事项”中更新关于hi_adc_read接口中delay_cnt参数值的注意说明。在“6.2 功能描述”中新增hi_gpio_get_dir、hi_gpio_get_output_val、hi_gpio_set_isr_mask、hi_gpio_set_isr_mode接口说明；更新hi_gpio_register_isr_function、hi_gpio_unregister_isr_function接口名称。在“6.4 注意事项”中新增关于hi_gpio_set_isr_mask和hi_gpio_set_isr_mode接口的注意说明。更新“9.4 注意事项”。
00B05	2020-03-06	<ul style="list-style-type: none">在“1.2 功能描述”中新增hi_uart_lp_save、hi_uart_lp_restore接口说明。在“1.4 注意事项”中新增关于hi_uart_lp_restore接口与hi_uart_lp_save接口的注意说明。



文档版本	发布日期	修改说明
00B04	2020-02-12	<ul style="list-style-type: none">在“1.2 功能描述”中新增hi_uart_is_buf_empty、hi_uart_is_busy接口说明。更新“1.4 注意事项”。在“7.1 概述”中新增关于256bit大小字段的描述。更新“7.2 功能描述”中的“表7-1”。在“7.3 开发指引”中更新示例1的说明，更新示例2的注释。新增“9 SDIO”章节。在“10.2 功能描述”中新增hi_tsensor_code_to_temperature、hi_tsensor_temperature_to_code接口说明。在“11.2 功能描述”中更新hi_dma_link_list_transfer接口名称，新增hi_dma_is_init接口说明。
00B03	2020-01-15	<ul style="list-style-type: none">在“1.2 功能描述”、“2.2 功能描述”中新增如果支持DMA数据收发的说明。在“2.4 注意事项”中新增使用microwire帧协议时的注意事项。新增“4 I2S”章节。更新“5 LSADC”提供的接口，并更新相应的“5.2 功能描述”、“5.3 开发指引”、“5.4 注意事项”。在“11.2 功能描述”中新增如果需要在SPI/UART中使用DMA传输数据的说明。
00B02	2019-12-19	<ul style="list-style-type: none">在“2.4 注意事项”中增加关于使用SPI单元的从模式时只支持半双工数据传输方式的说明。更新“7.2 功能描述”中关于EFUSE寄存器等详细信息的说明，更新“表7-1”。更新“7.3 开发指引”中ID51对应EFUSE字段的读、写、锁操作的代码示例。
00B01	2019-11-15	第一次临时版本发布。



目录

前言.....	i
1 UART.....	1
1.1 概述.....	1
1.2 功能描述.....	1
1.3 开发指引.....	2
1.4 注意事项.....	3
2 SPI.....	4
2.1 概述.....	4
2.2 功能描述.....	4
2.3 开发指引.....	5
2.4 注意事项.....	6
3 I2C.....	7
3.1 概述.....	7
3.2 功能描述.....	7
3.3 开发指引.....	7
3.4 注意事项.....	8
4 I2S.....	9
4.1 概述.....	9
4.2 功能描述.....	9
4.3 开发指引.....	10
4.4 注意事项.....	12
5 LSADC.....	13
5.1 概述.....	13
5.2 功能描述.....	13
5.3 开发指引.....	13
5.4 注意事项.....	14
6 IO/GPIO.....	15
6.1 概述.....	15
6.2 功能描述.....	15
6.3 开发指引.....	16
6.4 注意事项.....	16



7 EFUSE	17
7.1 概述	17
7.2 功能描述	17
7.3 开发指引	21
7.4 注意事项	22
8 Flash	23
8.1 概述	23
8.2 功能描述	23
8.3 开发指引	23
8.4 注意事项	24
9 SDIO	25
9.1 概述	25
9.2 功能描述	25
9.3 开发指引	26
9.4 注意事项	29
10 Tsensor	30
10.1 概述	30
10.2 功能描述	30
10.3 开发指引	31
10.4 注意事项	32
11 DMA	34
11.1 概述	34
11.2 功能描述	34
11.3 开发指引	35
11.4 注意事项	35
12 SYSTICK	36
12.1 概述	36
12.2 功能描述	36
12.3 开发指引	36
12.4 注意事项	36
13 PWM	37
13.1 概述	37
13.2 功能描述	37
13.3 开发指引	37
13.4 注意事项	38
14 WDG	39
14.1 概述	39
14.2 功能描述	39
14.3 开发指引	39



14.4 注意事项.....	40
----------------	----



1 UART

1.1 概述

1.2 功能描述

1.3 开发指引

1.4 注意事项

1.1 概述

通用异步收发器UART（Universal Asynchronous Receiver Transmitter）是一个异步串行的通信接口，主要功能是和外芯片的UART进行对接，从而实现两芯片间的通信。芯片提供3个UART单元。

1.2 功能描述

说明

如果UART驱动需要支持DMA数据收发，需要确保DMA驱动已经初始化。

UART模块提供以下接口：

- hi_uart_init：UART初始化。
- hi_uart_read：读数据。
- hi_uart_write：写数据。
- hi_uart_deinit：去初始化UART。
- hi_uart_set_flow_ctrl：配置UART硬件流程控制功能。
- hi_uart_write_immediately：轮询写数据。
- hi_uart_get_attribute：获取UART配置参数。
- hi_uart_is_buf_empty：查询UART FIFO与软件BUF是否为空。
- hi_uart_is_busy：查询UART是否忙。
- hi_uart_quit_read：退出阻塞读数据。
- hi_uart_lp_save：深睡前保存UART寄存器内容。



- hi_uart_lp_restore: 深睡唤醒后恢复UART寄存器内容。

1.3 开发指引

以应用UART2为例，数据收发流程如下：

步骤1 配置IO复用：将对应的IO复用为UART的TX、RX、RTS、CTS功能。

如果不需要支持硬件流控，仅配置TX、RX即可。

```
hi_void usr_uart_io_config()
{
    /* 如下IO复用配置，也可集中在SDK中的app_io_init函数中进行配置 */
    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 */
}
```

步骤2 UART初始化：配置UART的波特率、数据位等属性，并使能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;
}
```

步骤3 UART数据收发：调用UART读写数据接口，进行数据收发。

```
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;
}
```

----结束



1.4 注意事项

- SDK中，UART1默认用作AT命令通道，复用GPIO5/6 为UART的TX、RX功能。
- SDK中，UART0默认用作程序烧写和维测数据通道，复用GPIO3/4 为UART的TX、RX功能，用户如果使用UART0作为其他功能，可屏蔽app_main中的hi_diag_init函数。
- UART0不支持硬件流控功能。
- hi_uart_lp_restore接口与hi_uart_lp_save接口用于深睡唤醒与睡眠流程中，以保证唤醒后UART能够恢复睡前配置。



2 SPI

2.1 概述

2.2 功能描述

2.3 开发指引

2.4 注意事项

2.1 概述

SPI可以作为Master或Slave与外部设备进行同步串行通信（外围设备必须支持SPI帧格式），芯片提供2个SPI单元，SPI0具有收/发分开的位宽为16bit×256的FIFO，SPI1具有收/发分开的位宽为16bit×64的FIFO。

2.2 功能描述

说明

如果SPI驱动需要支持DMA数据收发，需要确保DMA驱动已经初始化。

SPI模块提供以下接口：

- hi_spi_init：SPI初始化（包括：主从设备、极性、相性、帧协议、传输频率、传输位宽等设定）。
- hi_spi_deinit：SPI模块去初始化（关闭相应的SPI单元，释放资源）。
- hi_spi_set_basic_info：配置SPI参数（例如：极性、相性、帧协议、传输位宽、频率等）。
- hi_spi_host_writeread：SPI主模式全双工收发数据。
- hi_spi_host_read：SPI主模式半双工接收数据。
- hi_spi_host_write：SPI主模式半双工发送数据。
- hi_spi_slave_read：SPI从模式半双工接收数据。
- hi_spi_slave_write：SPI从模式半双工发送数据。
- hi_spi_set_irq_mode：设置是否使用中断方式传输数据，主模式如果不配置中断方式，则传输数据默认使用轮询模式；从模式默认使用中断方式传输数据。



- hi_spi_set_dma_mode: 从设备设置是否使用DMA方式传输数据。
- hi_spi_register_usr_func: 注册用户准备/恢复函数。
- hi_spi_set_loop_back_mode: 设置是否为回环测试模式。

2.3 开发指引

SPI用于对接支持SPI协议的设备，SPI单元可以作为主设备或从设备，以SPI单元作为主设备为例，写数据操作如下：

步骤1 通过IO复用，复用SPI功能用到的管脚为SPI功能。其中：

- SPI0提供2组复用管脚。
- SPI1提供1组复用管脚。

步骤2 调用hi_spi_init，初始化SPI资源，选择SPI功能单元以及配置SPI参数。

```
hi_u32 usr_spi_init()
{
    hi_u32 ret;
    /* SPI 单元选择 */
    hi_spi_idx id = 0;
    /* IO 复用 */
    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;
    /* 主模式设置 */
    init_param.is_slave = HI_FALSE;
    hi_spi_cfg_basic_info spi_cfg;
    /* spi cfg 参数设置 */
    spi_cfg.cpha = 0; /* 极性 相性 */
    spi_cfg.cpol = 0;
    spi_cfg.data_width = 7; /* 数据位宽 */
    spi_cfg.endian = 0; /* 小端 */
    spi_cfg.fram_mode = 0; /* 帧协议 */
    spi_cfg_basic_info.freq = 8000000; /* 频率 */
    ret = hi_spi_init(id, init_param, &spi_cfg);
    return ret;
}
```

步骤3 调用hi_spi_host_write，进行SPI主设备写操作。

以主设备发送数据为例：

```
hi_u32 demo_spi_host_write_task()
{
    hi_u32 ret;
    hi_spi_idx id = 0; /* SPI单元选择 */
    hi_u8 send_buf[BUF_LEN]; /* 要传输的数据 */
    /* 数据处理 */
    ret = hi_spi_host_write(id, send_buf, BUF_LEN);
    /* 错误判断 并处理 */
    return ret;
}
```

----结束



2.4 注意事项

- 芯片作为主设备时，如果从设备速率较慢，主设备在每次调用读写接口后进行适当延时，避免从设备因读写数据太慢导致数据出错。
- 当不再使用SPI时，必须调用hi_spi_deinit进行资源释放，否则在进行初始化时将返回HI_ERR_SPI_REINIT错误。
- 使用hi_spi_set_basic_info接口进行参数重新设置时，必须判断返回的状态码，在SPI正在传输时不能进行参数重新设置，否则返回HI_ERR_SPI_BUSY错误。
- 使用SPI单元的从模式时，数据传输方式不支持轮询模式；使用SPI单元的主模式时，数据传输方式不支持DMA模式。
- 使用SPI单元的从模式时，只支持半双工数据传输方式。
- 使用microwire帧协议时，由于microwire帧协议限制，主设备只能发送8bit位宽数据。



3 I2C

3.1 概述

3.2 功能描述

3.3 开发指引

3.4 注意事项

3.1 概述

I2C模块是APB总线上的从设备，是I2C总线上的主设备，用于完成CPU对I2C总线上从设备的数据读写。

3.2 功能描述

I2C模块提供以下接口：

- hi_i2c_init：I2C初始化（配置中断、SCI信号高低电平等）。
- hi_i2c_deinit：I2C去初始化（清除中断、复位I2C状态等）。
- hi_i2c_register_reset_bus_func：注册I2C回调函数，用于扩展。
- hi_i2c_set_baudrate：修改I2C波特率。
- hi_i2c_write：I2C发送数据。
- hi_i2c_read：I2C接收数据。
- hi_i2c_writeread：I2C发送与接收数据双线传输。

3.3 开发指引

I2C模块可以读写EEPROM，同时也支持对单声道ADC-DAC音频解码器ES8311的读写，下面以读写EEPROM为例：

步骤1 通过IO复用，复用I2C功能用到的管脚为I2C功能。

步骤2 调用hi_i2c_init，初始化I2C资源，配置I2C硬件设备的选择和波特率。

```
hi_void sample_i2c_init(hi_i2c_idx id)
{
```



```
/* IO复用 */  
/* ... */  
  
hi_i2c_init(id, baudrate);  
}
```

步骤3 调用hi_i2c_write，实现主设备发送、从设备EEPROM接收，同时为了将来对10bit地址器件扩展的支持，设备地址长度为16bit。

```
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;  
}
```

----结束

3.4 注意事项

- 函数hi_set_baudrate为初始化后调用，方便修改波特率，如果I2C未经过初始化直接调用hi_set_baudrate会返回错误。
- 需要主动确保数据发送指针send_buf和数据接收指针receive_buf不可传入空指针。
- 当发送的数据大于对接设备的可接受范围，发送会失败；并且如果在错误发送后再切换另一个I2C设备继续发送，会造成总线挂死，所有I2C设备都无法正确发送。



4 I2S

4.1 概述

4.2 功能描述

4.3 开发指引

4.4 注意事项

4.1 概述

I2S模块用于完成音频设备之间的数据传输。

4.2 功能描述

I2S模块特性：

- 仅支持I2S Master收发，I2S主时钟（MCLK）仅支持12.288MHz（误差不超过10Hz）。
- 满足飞利浦I2S协议规范，仅支持norm I2S。
- 采样位数16bit或24bit。
- I2S的采样率8kHz、16kHz、32kHz、48kHz（SCK的频率=2×采样频率×采样位数）。
- 支持左右声道。
- 支持DMA模式。

说明

需确保在使用I2S模块相关接口前，DMA驱动已经初始化。

I2S模块提供以下接口：

- hi_i2s_init：I2S初始化。
- hi_i2s_deinit：I2S去初始化。
- hi_i2s_write：I2S发送数据。



- hi_i2s_read: I2S接收数据。

4.3 开发指引

I2S主要用于音频设备之间的数据传输，下面以音频解码器ES8311播放音频为例介绍I2S接口的使用：

- 步骤1** 通过IO复用接口设置相应管脚为I2S、I2C（I2C用于初始化音频解码器ES8311）功能。
- 步骤2** 调用hi_i2c_write，初始化ES8311。
- 步骤3** 调用hi_i2s_init，初始化I2S资源，配置I2S采样率和采样精度。
- 步骤4** 调用hi_flash_read，将音频数据从Flash复制到RAM中。
- 步骤5** 调用hi_i2s_write，将RAM中的音频数据传给ES8311，然后通过耳机播放出来。

----结束

示例：

```
#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;

/* 需要事先将小于100K的音频数据烧录到Flash的0x001A1000地址上 */
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;
    }

    /* 此示例使用I2C1初始化es8311，IO复用设置时需选择支持I2C1功能的管脚 */
    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 注意事项

使用收发接口时，应尽量缩短2次调用hi_i2s_write或hi_i2s_read之间的时间间隔。



5 LSADC

5.1 概述

5.2 功能描述

5.3 开发指引

5.4 注意事项

5.1 概述

LSADC (Low Speed ADC) 实现对外部模拟信号转换成一定比例的数字值，从而实现对模拟信号的测量，可应用于电量检测、按键检测等，LSADC的扫描频率 $\leq 166\text{K/s}$ 。

5.2 功能描述

LSADC模块提供以下接口：

- hi_adc_read：从指定的ADC通道读取数据。

5.3 开发指引

芯片提供1个LSADC、8个独立通道。通过hi_adc_read接口读取指定通道的一个数据；通道7为参考电压，不能用作其他ADC转换。

码字与输入电压V的转换关系： $\text{码字} = V/4/1.8 \times 4096$

以LSADC读取通道7的电压为例，操作步骤如下：

步骤1 调用hi_adc_read读取通道7的电压值。

----结束

示例：

```
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 注意事项

- LSADC扫描启动前需确认上次扫描已停止。
- LSADC通道7为内部VBAT电压检测通道，非管脚输入。
- hi_adc_read接口中delay_cnt参数值取值范围为[0,0xFF0]，从配置采样到启动采样的延时时间计数，一次计数是334ns，其值需在0~0xFF0之间。



6 IO/GPIO

6.1 概述

6.2 功能描述

6.3 开发指引

6.4 注意事项

6.1 概述

IO驱动支持配置IO驱动能力、复用IO为外设管脚、设置上下拉状态等功能。

GPIO驱动支持设置GPIO管脚方向、设置输出电平状态、中断上报等功能。

6.2 功能描述

IO模块提供以下接口：

- hi_io_set_pull：设置某个IO上下拉功能。
- hi_io_get_pull：获取某个GPIO上下拉状态。
- hi_io_set_func：设置某个IO的复用功能。
- hi_io_get_func：获取某个IO的复用功能。
- hi_io_set_driver_strength：设置某个IO驱动能力。
- hi_io_get_driver_strength：获取某个IO驱动能力。

GPIO模块提供以下接口：

- hi_gpio_init：GPIO模块初始化。
- hi_gpio_deinit：GPIO模块去初始化。
- hi_gpio_set_dir：设置某个GPIO管脚方向。
- hi_gpio_get_dir：获取某个GPIO管脚方向。
- hi_gpio_set_output_val：设置单个GPIO管脚输出电平状态。
- hi_gpio_get_output_val：获取单个GPIO管脚输出电平状态。



- hi_gpio_get_input_val: 获取某个GPIO管脚输入电平状态。
- hi_gpio_register_isr_function: 使能某个GPIO的中断功能。
- hi_gpio_unregister_isr_function: 去使能某个GPIO的中断功能。
- hi_gpio_set_isr_mask: 设置某个GPIO中断屏蔽使能。
- hi_gpio_set_isr_mode: 设置某个GPIO中断触发模式。

6.3 开发指引

IO接口和GPIO接口独立使用，遵循如下操作步骤：

步骤1 确定相应管脚是否用作GPIO。

步骤2 如果管脚用作GPIO，可对GPIO管脚方向进行设置，可设置上升沿/下降沿中断，如sample_gpio所示。

步骤3 如果管脚复用为其他外设驱动管脚，需对其进行复用功能设置、驱动能力设置（可选）等，如sample_io所示。

----结束

示例：

```
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 注意事项

- 配置IO复用功能时，应关注该IO是否已经被复用为其他功能，避免影响既有功能。
- hi_gpio_set_isr_mask和hi_gpio_set_isr_mode在hi_gpio_register_isr_function之后使用才有意义，在hi_gpio_register_isr_function之前使用，对应配置会被hi_gpio_register_isr_function中的配置覆盖。
- SFC相关的IO复用和配置，仅供SDK内部使用，切勿复用为其它功能，否则将导致系统异常。



7 EFUSE

7.1 概述

7.2 功能描述

7.3 开发指引

7.4 注意事项

7.1 概述

EFUSE是一种可编程的存储单元，由于其只可编程一次的特征，多用于芯片保存Chip ID、密钥或其他一次性存储数据。Hi3861的EFUSE大小为2KB（偏移地址0x000 ~ 0x7FF），按照用途分割为若干字段，其中有1个256bit大小的字段和1个64bit大小的字段预留给用户分配，用户可根据用途将这2个字段再进行细分。

7.2 功能描述

EFUSE模块提供两套不同的读写接口：

- 将分割出的各个EFUSE字段按照首地址顺序从小到大排ID，通过ID号索引到对应的EFUSE功能字段：
 - hi_efuse_get_id_size：获取ID号对应的EFUSE字段的长度（单位：bit）。
 - hi_efuse_read：从ID号对应的EFUSE字段中读取数据。
 - hi_efuse_write：写数据到ID号对应的EFUSE字段。
 - hi_efuse_lock：通过锁ID加锁EFUSE中的某个区域，加锁后该区域无法再写入。
 - hi_efuse_get_lockstat：获取EFUSE的锁状态，查询哪些区域已锁定。
- 指定EFUSE起始地址的读写方式，主要针对用户预留区使用：
 - hi_efuse_usr_read：从指定的起始地址读EFUSE。
 - hi_efuse_usr_write：从指定的起始地址写EFUSE。

EFUSE的寄存器等详细信息请参见《Hi3861V100 / Hi3861LV100 / Hi3881V100 WiFi 芯片 用户指南》的“5.7 EFUSE”小节，软件EFUSE ID和锁ID与EFUSE字段的对应关系如表7-1所示。



表 7-1 EFUSE 字段与 ID 对应关系表

字段	软件EFUSE ID	软件锁ID
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	-
kdf2ecc_huk_disable	21	-
SSS_corner	22	-



字段	软件EFUSE ID	软件锁ID
uart_halt_interval	23	-
ts_trim	24	-
chip_id2	25	-
ipv4_mac_addr	26	-
ipv6_mac_addr	27	-
pa2gccka0_trim0	28	-
pa2gccka1_trim0	29	-
nvrām_pa2ga0_trim0	30	-
nvrām_pa2ga1_trim0	31	-
pa2gccka0_trim1	32	-
pa2gccka1_trim1	33	-
nvrām_pa2ga0_trim1	34	-
nvrām_pa2ga1_trim1	35	-
pa2gccka0_trim2	36	-
pa2gccka1_trim2	37	-
nvrām_pa2ga0_trim2	38	-
nvrām_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	-
die_id2	51	-
PG0	-	0



字段	软件EFUSE ID	软件锁ID
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
PG29	-	29
PG30	-	30



字段	软件EFUSE ID	软件锁ID
PG31	-	31
PG32	-	32
PG33	-	33
PG34	-	34
PG35	-	35

7.3 开发指引

EFUSE通常用于存储修复数据或芯片信息，操作步骤如下：

- 步骤1** 调用hi_efuse_write或hi_efuse_usr_write，写数据到EFUSE中。
- 步骤2** 调用hi_efuse_read或hi_efuse_usr_read，从EFUSE中读取数据。
- 步骤3** 调用hi_efuse_lock或hi_efuse_usr_write，加锁EFUSE中的某个区域，加锁后该区域无法再写入数据。

----结束

示例1：ID20对应EFUSE字段的读、写、锁操作。

```
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;
}
```



示例2：用户预留区的读、写、锁操作。

```
#define EFUSE_USR_RW_SAMPLE_BUFF_MAX_LEN 2 /* customer_rsvd1的长度为64bits, 读写数据
需要2个32bit空间来存储 */
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; /* customer_rsvd1的偏移地址为0x75C */
    hi_u16 rw_bits = 64; /* customer_rsvd1的长度为64bits */
    hi_u16 lock_start_bit = 0x7FD; /* customer_rsvd1锁的偏移地址为0x7FD */
    hi_u16 lock_bits = 1; /* customer_rsvd1锁的长度为1bit */

    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 注意事项

- 读取EFUSE时，data用于存储读取的数据，用户须确保data的空间>要读取数据的大小。
- 指定EFUSE起始地址的读写方式支持用户写入任意地址数据，但建议用户使用用户预留区，其他区域的写入建议使用ID号索引的方式。
- 使用hi_efuse_usr_read时，输入起始地址必须8bit对齐，输入待读取的bit位数，如果不是8bit对齐则函数内部会处理为8bit对齐，用户读取数据后需右移处理，如输入读取位数为10，则读取到的数据右移6位后才是真正要读取的数据。
- 锁PG31（偏移地址：0x7FB，长度：1bit）及其对应的EFUSE字段（偏移地址：0x73F，长度：11bit）写后立即生效，其他EFUSE区域写成功后需要重启才能生效。



8 Flash

8.1 概述

8.2 功能描述

8.3 开发指引

8.4 注意事项

8.1 概述

Flash模块提供对Flash存储器的读写擦操作。

8.2 功能描述

Flash模块提供以下接口：

- hi_flash_init：Flash模块初始化（一般在系统启动时调用）。
- hi_flash_deinit：去初始化Flash设备。
- hi_flash_ioctl：获取Flash信息。
- hi_flash_erase：将指定的Flash区域数据擦除。
- hi_flash_write：将数据写入指定的Flash区域。
- hi_flash_read：读出Flash数据到指定缓存区域。

8.3 开发指引

Flash模块用于Flash存储器的读写，支持的Flash器件：W25Q16JL、W25Q16JW、GD25WQ16、EN25S16、EN25QH16。以Flash模块对Flash器件读写数据为例，操作步骤如下：

步骤1 调用hi_flash_write，对Flash器件进行写数据；调用hi_flash_read，对Flash器件进行读数据。

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



```
hi_bool do_erase = HI_FLASE; /* 是否擦除Flash */
hi_u32 flash_offset = 0x10000; /* Flash的偏移地址 */
hi_u32 size = 32; /* 读写Flash的数据长度 */
hi_u8 ram_data[32]; /*读写数据的BUF */

/* 填充 ram_data, flash_offset size do_erase */
/* 写 Flash */
ret = hi_flash_write(flash_offset, size, ram_data, do_erase);
if (ret != HI_ERR_SUCCESS) {
    /* 错误处理 */
}
memset_s(ram_data, DATA_LEN, 0, DATA_LEN); /* 清空BUF */
/* 读Flash */
ret = hi_flash_read(flash_offset, size, ram_data);
if (ret != HI_ERR_SUCCESS) {
    /* 错误处理 */
}

return ret;
}
```

----结束

8.4 注意事项

hi_flash_erase接口使用时，要求擦除地址和长度必须是4KB对齐。



9 SDIO

9.1 概述

9.2 功能描述

9.3 开发指引

9.4 注意事项

9.1 概述

SDIO模块作为从设备，用于同SDIO主设备进行数据和消息的传输。

9.2 功能描述

SDIO模块特性：

- 要求主设备SDIO模块工作时钟 $\leq 50\text{MHz}$ 。
- 使用DMA进行数据传输。

SDIO模块提供以下接口：

- hi_sdio_init：SDIO初始化。
- hi_sdio_reinit：SDIO重新初始化。
- hi_sdio_register_callback：注册SDIO中断处理函数，包括如下几种中断类型：
 - 感知到HOST发起了读数据操作
 - 感知到HOST读数据结束
 - 感知到HOST读数据错误
 - 感知到HOST发起了写数据操作
 - 感知到HOST写数据结束
 - 接收到HOST发送的消息
 - 接收到HOST发起的软复位中断
- hi_sdio_soft_reset：SDIO模块软复位。
- hi_sdio_complete_send：结束发送数据（在ADMA链表中，补充一个4byte的数据）。



- hi_sdio_set_pad_admatab: 补充指定长度的数据到ADMA链表中。
- hi_sdio_write_extinfo: 写入SDIO扩展配置信息。
- hi_sdio_send_data: 发送指定长度的数据, 数据内容需要在接收到HOST发起读数据中断时, 填充到ADMA链表中。
- hi_sdio_set_admatable: 填充指定数据到ADMA链表中。
- hi_sdio_sched_msg: 将挂起在消息队列中的消息发送。
- hi_sdio_send_sync_msg: 将指定消息加入消息队列并发送。
- hi_sdio_send_msg_ack: 立即发送指定消息, 当前正在发送的消息将被覆盖。
- hi_sdio_process_msg: 清除消息队列中挂起的指定消息, 将新消息加入消息队列并发送。
- hi_sdio_get_extend_info: 获取SDIO扩展信息。
- hi_sdio_is_pending_msg: 指定消息是否在挂起的消息队列中。
- hi_sdio_is_sending_msg: 指定消息是否正在发送。
- hi_sdio_register_notify_message_callback: 注册发送数据消息时的回调函数(例如: 可再发送数据或消息时通过GPIO通知HOST)。
- hi_sdio_set_powerdown_when_deep_sleep: 设置系统深睡模式下, SDIO是否掉电(默认为掉电)。

9.3 开发指引

以SDIO简单数据和消息收发为例, SDIO接口使用流程如下:

- 步骤1** 在app_io_init中, 通过IO复用接口配置相应管脚为SDIO功能(例如: 将IO9~14复用为SDIO的DATA、CMD、CLK, 同时配置DATA上拉)。
- 步骤2** 调用hi_sdio_init, 初始化SDIO。
- 步骤3** 调用hi_sdio_register_callback, 注册SDIO中断处理回调函数。
- 步骤4** 调用hi_sdio_send_data, 发送数据, 在感知到HOST开始读数据的中断处理函数时, 将需要发送的数据地址填充到ADMA链表。
- 步骤5** 接收HOST发送的数据(在感知到HOST开始写数据的中断处理函数时, 将接收数据的地址填充到ADMA链表; 当感知到HOST写数据结束时, 表示HOST发送数据完成)。
- 步骤6** 调用hi_sdio_send_sync_msg, 发送消息。
- 步骤7** 接收HOST发送的消息(在接收到HOST发送消息的中断处理函数时, 根据消息内容处理对应的消息)。

----结束

示例:

```
#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\n", len);
hi_watchdog_feed();
g_receive_data_addr = &g_receive_data[0];
g_receive_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_receive_data_len);

    hi_u8* received_data = (hi_u8*)&g_receive_data[0];
    for (hi_u32 i = 0; i < g_receive_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 注意事项

- 由于系统启动默认开启DCache，且SDIO内部采用DMA进行数据传输，如果内存中数据与Cache中数据不一致，将导致DMA传输出现异常。因此，需要在数据传输前flush Dcache，或在系统启动后，直接关闭DCache后再启动SDIO数据收发。
- 如果SDIO HOST设备没有拉高DATA，则需要在IO复用配置的同时，配置DATA上拉。



10 Tsensor

10.1 概述

10.2 功能描述

10.3 开发指引

10.4 注意事项

10.1 概述

Tsensor检测CPU结温，检测范围-40℃ ~ 140℃，校准修调后温度检测绝对精度为 $\pm 3^{\circ}\text{C}$ 。

10.2 功能描述

Tsensor模块提供以下接口：

- hi_tsensor_start：启动Tsensor模块。
- hi_tsensor_stop：停止Tsensor温度采集。
- hi_tsensor_destroy：关中断，删除注册的中断回调，停止Tsensor模块。
- hi_tsensor_read_temperature：非中断方式读取Tsensor温度。
- hi_tsensor_set_temp_trim：Tsensor温度校准，可设置通过寄存器或EFUSE方式校准，默认为通过EFUSE校准。
- hi_tsensor_code_to_temperature：温度码字转换为温度值。
- hi_tsensor_temperature_to_code：温度值转换为温度码字。
- hi_tsensor_set_outtemp_threshold：设置高温门限和低温门限并注册用户过温中断回调。
- hi_tsensor_set_overtemp_threshold：设置超高温门限并注册用户过温中断回调。
- hi_tsensor_set_pdtemp_threshold：设置过温掉电保护门限。
- hi_tsensor_register_temp_collect_finish_int_callback：注册温度采集完成中断回调。



通过在EFUSE或寄存器中设置校准码，可对Tsensor内部的温度值进行校准，对应的校准关系如表10-1所示。

表 10-1 Tsensor 温度校准配置表

校准码（二进制）	温度码修调值	温度修调值(°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

注：校准码1000为正负温度校准分界线。

10.3 开发指引

Tsensor监控芯片节温，实现高低温警报，超高温警报及过温掉电多级保护。操作步骤如下：

- 步骤1** 调用hi_tsensor_set_temp_trim，进行温度校准。如果采用EFUSE方式校准温度或不进行温度校准，则略过此步骤。
- 步骤2** 调用hi_tsensor_set_pdtemp_threshold，设置过温掉电保护门限。
- 步骤3** 调用hi_tsensor_set_outtemp_threshold，设置高低温门限并注册用户过温中断回调。此步骤可省略。
- 步骤4** 调用hi_tsensor_set_overtemp_threshold，设置超高温门限并注册用户超高温中断回调。此步骤可省略。



步骤5 调用hi_tsensor_start，选择上报温度模式并启动Tsensor模块。

----结束

示例：

```
/* 以下宏定义仅供参考，用户应根据产品需求自行定义 */
#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();

    /* 高低温警报处理 */
}

hi_void user_overtemp_callback(hi_s16 temperature)
{
    hi_tsensor_destroy();

    /* 超高温警报处理 */
}

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 注意事项

- Tsensor工作在单点循环上报模式时，仅采集完成中断能够生效。



- WiFi业务启动时会启用Tsensor进行过温保护，当WiFi业务启动时如果需要将Tsensor作他用，请结合WiFi业务评估是否会影响WiFi业务的过温保护功能。
- 当使用中断方式读取温度时，建议在用户中断回调中调用hi_tsensor_stop停止Tsensor，避免中断频繁占用过多CPU资源而导致其他业务无法得到调度。
- 使用超高温保护功能，需在用户中断回调中调用hi_tsensor_destroy停止超高温中断，然后在通过非中断方式读取温度确认是否确实过温，再采取警报，关闭部分业务等相应措施进行处理。



11 DMA

- [11.1 概述](#)
- [11.2 功能描述](#)
- [11.3 开发指引](#)
- [11.4 注意事项](#)

11.1 概述

直接存储器访问（DMA，Directory Memory Access）方式是一种完全由硬件执行I/O交换的工作方式。在这种方式中，直接存储器访问控制器（DMAC，Directory Memory Access Controller）直接在存储器和外设、外设和外设、存储器和存储器之间进行数据传输，减少处理器的干涉和开销。DMA方式一般用于高速传输成组的数据。DMAC在收到DMA传输请求后根据CPU对通道的配置启动总线主控制器，向存储器和外设发出地址和控制信号，对传输数据的个数计数，并且以中断方式向CPU报告传输操作的结束或错误。

11.2 功能描述

说明

如果需要在SPI/UART中使用DMA传输数据，或者需要使用I2S驱动，需要在系统启动时进行DMA初始化。

DMA模块提供以下接口：

- hi_dma_create_link_list：创建DMA传输链表。
- hi_dma_add_link_list_item：传输链表末尾添加节点。
- hi_dma_link_list_transfer：启动DMA链表传输。
- hi_dma_mem2mem_transfer：启动DMA内存数据传输。
- hi_dma_ch_close：关闭DMA指定通道。
- hi_dma_init：DMA初始化。
- hi_dma_deinit：DMA去初始化。



- hi_dma_is_init: DMA模块是否初始化。

其中，链表传输相关接口多用于多块不连续内存数据的传输。当链表传输完成后，可根据分配到的通道号关闭DMA指定通道。

11.3 开发指引

DMA接口仅对外提供存储器到存储器的拷贝功能（其他拷贝方式已经集成到相应的外设驱动中），操作步骤如下：

步骤1 调用hi_dma_init，初始化DMA模块

步骤2 调用hi_dma_mem2mem_transfer，启动DMA数据传输。

步骤3 调用hi_dma_deinit，去初始化DMA模块（释放DMA资源，可仅在不再使用DMA功能时调用该接口）。

----结束

```
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 注意事项

- DMA中断回调函数执行于中断上下文，调用回调函数时需要遵守中断上下文的编程注意事项。
- 建议仅在需要非阻塞进行数据拷贝的场景下使用DMA，阻塞场景下，仍建议使用memcpy_s进行数据拷贝。



12 SYSTICK

12.1 概述

12.2 功能描述

12.3 开发指引

12.4 注意事项

12.1 概述

SYSTICK是系统节拍定时器，提供2个功能：

- 获取SYSTICK当前计数
- 将SYSTICK计数值清0

12.2 功能描述

SYSTICK模块提供以下接口：

- hi_systick_get_cur_tick：获取当前SYSTICK计数值。
- hi_systick_clear：将SYSTICK计数值清0。

12.3 开发指引

调用hi_systick_get_cur_tick可以获取当前SYSTICK计数值，每个值的时间由SYSTICK时钟源决定。SYSTICK时钟为32kHz，一个TICK值为1/32000秒，同时接口内调用了延时接口，所以禁止在中断上下文调用该接口。而hi_systick_clear是将SYSTICK清0，但是接口返回后需要等待3个SYSTICK的时钟周期才会完成清0操作。

12.4 注意事项

无



13 PWM

13.1 概述

13.2 功能描述

13.3 开发指引

13.4 注意事项

13.1 概述

PWM用于输出波形，共有6个端口，调用PWM端口时需要进行IO复用。

13.2 功能描述

PWM模块提供以下接口：

- hi_pwm_init：初始化PWM。
- hi_pwm_deinit：去初始化PWM。
- hi_pwm_set_clock：设置PWM模块时钟类型。
- hi_pwm_start：启动PWM信号输出。
- hi_pwm_stop：停止PWM信号输出。

13.3 开发指引

PWM利用微处理器的数字输出对模拟电路进行控制，操作步骤如下：

步骤1 PWM可以使用不同的时钟源，工作时钟为160M，外部晶体可以使用24M或40M，PWM不同端口需要复用不同的GPIO，通过传入端口号调用hi_pwm_init对PWM进行初始化。

步骤2 调用hi_pwm_set_clock，设置PWM模块时钟类型（默认为150M时钟）。

步骤3 调用hi_pwm_start，按配置输入参数输出PWM信号，信号占空比为duty/freq，频率为时钟源频率/freq。

----结束



示例:

```
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 注意事项

PWM调用hi_pwm_stop时不支持在中断中调用。



14_{WDG}

14.1 概述

14.2 功能描述

14.3 开发指引

14.4 注意事项

14.1 概述

WatchDog用于系统异常恢复，如果未得到更新则隔一定时间（可编程，最大时间间隔26s）产生一个系统复位信号，当WatchDog在此之前关闭工作时钟或更新计数器，复位信号不会产生。

14.2 功能描述

WatchDog模块提供以下接口：

- hi_watchdog_enable：使能看门狗。
- hi_watchdog_feed：喂狗，重新启动计数器，即踢狗。
- hi_watchdog_disable：关闭看门狗。

14.3 开发指引

WatchDog一般用于检测是否死机，如果超过踢狗等待的时间没有进行踢狗操作，则产生一个系统复位。在需要踢狗的任务场景下按照以下步骤操作：

步骤1 调用hi_watchdog_enable，使能看门狗模块。

步骤2 调用hi_watchdog_feed，进行踢狗操作。

步骤3 调用hi_watchdog_disable，关闭看门狗。

----结束

示例：

```
hi_void test_wdg()  
{
```



```
/* 使能看门狗 */  
hi_watchdog_enable();  
hi_udelay(5000000); /* delay 5000000 us */  
/* 踢狗 */  
hi_watchdog_feed();  
/* 关闭看门狗 */  
hi_watchdog_disable();  
}
```

14.4 注意事项

看门狗在SDK中已经使能且已存在喂狗动作。