泰芯半导体
TaiXin Semiconductor

TXW81x API 手册

珠海泰芯半导体有限公司
Zhuhai Taixin Semiconductor Co.,Ltd

珠海市高新区港湾一号科创园港 11 栋 3 楼

责任与版权

责任限制

由于产品版本升级或者其他原因，本文档会不定期更新。除非另行约定，泰芯半导体有限公司对本文档所有内容不提供任何担保或授权。

客户应在遵守法律、法规和安全要求的前提下进行产品设计，并做充分验证。泰芯半导体有限公司对应用帮助或客户产品设计不承担任何义务。客户应对其使用泰芯半导体有限公司的产品和应用自行负责。

在适用法律允许的范围内，泰芯半导体有限公司在任何情况下，都不对因使用本文档相关内容及本文档描述的产品而产生的损失和损害进行超过购买支付价款的赔偿（除在涉及人身伤害的情况中根据适用的法律规定的损害赔偿外）。

版权申明

泰芯半导体有限公司保留随时修改本文档中任何信息的权利，无需提前通知且不承担任何责任。

修订记录

日期	版本	描　　述	修订人
2023-12-15	V1.0	初始版本	TX

目录

1. GPIO 接口说明

本章节主要介绍 GPIO 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

1.1. Enum

1.1.1. gpio_pin_direction

枚举量	说明
GPIO_DIR_INPUT	设置 GPIO PIN 为普通输入模式
GPIO_DIR_OUTPUT	设置 GPIO PIN 为普通输出模式

1.1.2. gpio_pin_mode

枚举量	说明
GPIO_PULL_NONE	设置 GPIO PIN 为无上下拉模式
GPIO_PULL_UP	设置 GPIO PIN 为上拉模式
GPIO_PULL_DOWN	设置 GPIO PIN 为下拉模式
GPIO_OPENDRAIN_PULL_NONE	设置 GPIO PIN 为开漏无上拉模式
GPIO_OPENDRAIN_PULL_UP	设置 GPIO PIN 为开漏上拉模式
GPIO_OPENDRAIN_DROP_NONE	设置 GPIO PIN 为开漏无下拉模式
GPIO_OPENDRAIN_DROP_DOWN	设置 GPIO PIN 为开漏下拉模式

1.1.3. gpio_irq_event

枚举量	说明
GPIO_IRQ_EVENT_NONE	设置无中断

GPIO_IRQ_EVENT_RISE	设置上升沿中断
GPIO_IRQ_EVENT_FALL	设置下降沿中断
GPIO_IRQ_EVENT_ALL	设置电平中断

1.1.4. gpio_ioctl_cmd

枚举量	说明
GPIO_INPUT_LAG	设置 GPIO PIN 输入迟滞 @Note: 输入迟滞：界定 0 和 1 的电压值，从固定值变为范围值（在固定值上下的范围）。例如，判 0 和 1 的电压固定值为 2V，开启迟滞后，变为 1.8V~2.2V。故电压大于 2.2V 为 1，电压小于 1.8V 为 0。
GPIO_DIR_ATOMIC	设置 GPIO PIN 方向的原子操作
GPIO_VALUE_ATOMIC	设置 GPIO PIN 输出值的原子操作
GPIO_LOCK	锁定 GPIO，配置不能更改，除非复位 GPIO 模块
GPIO_DEBUNCE	开启 GPIO PIN 输入的滤波功能
GPIO_OUTPUT_TOGGLE	对 GPIO PIN 输出值取反
GPIO_GENERAL_ANALOG	设置 GPIO PIN 为普通模拟模式

1.1.5. gpio_pull_level

枚举量	说明
GPIO_PULL_LEVEL_NONE	设置 GPIO PIN 的上/下拉阻值为 0
GPIO_PULL_LEVEL_4_7K	设置 GPIO PIN 的上拉阻值为 4.7K
GPIO_PULL_LEVEL_100K	设置 GPIO PIN 的上/下拉阻值为 100K

1.1.6. pin_driver_strength

枚举量	说明
GPIO_DS_4MA	设置 GPIO PIN 的驱动能力为 4mA
GPIO_DS_12MA	设置 GPIO PIN 的驱动能力为 12mA
GPIO_DS_20MA	设置 GPIO PIN 的驱动能力为 20mA
GPIO_DS_28MA	设置 GPIO PIN 的驱动能力为 28mA

1.1.7. gpio_afio_set

枚举量	说明
GPIO_AF_0	设置 GPIO PIN 的 AFIO 值为 0
GPIO_AF_1	设置 GPIO PIN 的 AFIO 值为 1
GPIO_AF_2	设置 GPIO PIN 的 AFIO 值为 2
GPIO_AF_3	设置 GPIO PIN 的 AFIO 值为 3

1.1.8. gpio_iomap_out_func

枚举量	说明
略	请参考 SDK\include\chip\txw81x\io_function.h

1.1.9. gpio_iomap_in_func

枚举量	说明
略	请参考 SDK\include\chip\txw81x\io_function.h

1.2. Define

1.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

1.3. Structure

1.3.1. gpio_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
mode	int32 函数指针	-
dir	int32 函数指针	-
set	int32 函数指针	-
get	int32 函数指针	-
request_pin_irq	int32 函数指针	-
release_pin_irq	int32 函数指针	-
ioctl	int32 函数指针	-

1.4. Function

1.4.1. gpio_set_mode()

设置 GPIO PIN 的工作模式，以及该模式下的上拉或下拉阻值。

- **函数原型**

```
int32 gpio_set_mode(  uint32               pin,
                      enum gpio_pin_mode   mode,
                      enum gpio_pull_level level
                    )
```

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
mode	enum gpio_pin_mode	设置 GPIO PIN 的模式，见枚举 gpio_pin_mode
level	enum gpio_pull_level	设置 GPIO PIN 的上拉或下拉的阻值，见枚举 gpio_pull_level

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 模式及阻值配置成功
RET_ERR	int32	GPIO PIN 模式及阻值配置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 模式及阻值配置失败

- **代码示例**

```
/* 配置 PA1 为上拉模式，上拉阻值为 4.7K */
gpio_set_mode(PA_1, GPIO_PULL_UP, GPIO_PULL_LEVEL_4_7K);
```

1.4.2. gpio_set_dir()

设置 GPIO PIN 的方向。

- **函数原型**

  int32 gpio_set_dir(uint32 pin, enum gpio_pin_direction direction)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
direction	enum gpio_pin_direction	设置 GPIO PIN 的方向，见枚举 gpio_pin_direction

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的方向设置成功
RET_ERR	int32	GPIO PIN 的方向设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的设置方向失败

- **代码示例**

```
/* 配置 PA1 为普通输出模式 */
gpio_set_dir(PA_1, GPIO_DIR_OUTPUT);
```

1.4.3. gpio_set_val()

设置 GPIO PIN 的值。在调用此函数之前，须将引脚设置为普通输出模式，见函数 gpio_set_sir()。

- **函数原型**

  int32 gpio_set_val(uint32 pin, int32 value)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
value	int32	设置 GPIO PIN 的值

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的值设置成功
RET_ERR	int32	GPIO PIN 的值设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的值设置失败

- **代码示例**

```
/* 配置 PA1 为普通输出模式 */
gpio_set_dir(PA_1, GPIO_DIR_OUTPUT);
/* 配置 PA1 输出 1（高电平） */
gpio_set_val(PA_1, 1);
```

1.4.4. gpio_get_val()

获取 GPIO PIN 的当前值。在调用此函数之前，须将引脚设置为普通输入模式，见函数 gpio_set_sir()。

- **函数原型**

  int32 gpio_set_val(uint32 pin, int32 value)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
value	int32	设置 GPIO PIN 的值

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的值设置成功
RET_ERR	int32	GPIO PIN 的值设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的值设置失败

- **代码示例**

```
/* 配置 PA1 为普通输出模式 */
gpio_set_dir(PA_1, GPIO_DIR_OUTPUT);
/* 配置 PA1 输出 1（高电平） */
gpio_set_val(PA_1, 1);
```

1.4.5. gpio_driver_strength()

设置 GPIO PIN 的驱动能力。

- **函数原型**

  int32 gpio_driver_strength(uint32 pin, enum pin_driver_strength strength)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
strength	enum pin_driver_strength	设置 GPIO PIN 的驱动能力，见枚举 pin_driver_strength

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的驱动能力设置成功
RET_ERR	int32	GPIO PIN 的驱动能力设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的驱动能力设置失败

- **代码示例**

```
/* 配置 PA1 的驱动能力为 28mA */
gpio_driver_strength(PA_1, GPIO_DS_28MA);
```

1.4.6. gpio_set_altnt_func()

设置 GPIO PIN 的 AFIO，根据 GPIO 的 AFIO 表格进行配置。

- **函数原型**

  int32 gpio_set_altnt_func(uint32 pin, enum gpio_afio_set afio)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
afio	enum gpio_afio_set	设置GPIO PIN的AFIO,见枚举gpio_afio_set

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的 AFIO 设置成功
RET_ERR	int32	GPIO PIN 的 AFIO 设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的 AFIO 设置失败

- **代码示例**

```
/* 配置 PA1 的 AFIO 为 0 */
gpio_set_altnt_func(PA_1, GPIO_AF_0);
```

1.4.7. gpio_iomap_output()

设置 GPIO PIN 的 IOMAP OUTPUT 的功能，根据所需要的功能进行配置。

- **函数原型**

  int32 gpio_iomap_output(uint32 pin, enum gpio_iomap_out_func func_sel)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
func_sel	enum gpio_iomap_out_func	设置 GPIO PIN 的 iomap output 功能，见枚举 gpio_iomap_out_func

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的 iomap output 功能设置成功
RET_ERR	int32	GPIO PIN 的 iomap output 功能设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的 iomap output 功能设置失败

- **代码示例**

```
/* 配置 PA1 的 iomap_output 的功能为 UART0 的 TX */
/* 即 PA1 作为 UART0 的 TX 功能 */
gpio_iomap_output(PA_1, GPIO_IOMAP_OUT_UART0_OUT);
```

1.4.8. gpio_iomap_input()

设置 GPIO PIN 的 IOMAP INPUT 的功能，根据所需要的功能进行配置。

- **函数原型**

  int32 gpio_iomap_input(uint32 pin, enum gpio_iomap_in_func func_sel)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
func_sel	enum gpio_iomap_in_func	设置 GPIO PIN 的 iomap input 功能，见枚举 gpio_iomap_in_func

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的 iomap input 功能设置成功
RET_ERR	int32	GPIO PIN 的 iomap input 功能设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的 iomap input 功能设置失败

- **代码示例**

```
/* 配置 PA1 的 iomap_output 的功能为 UART0 的 RX */
/* 即 PA1 作为 UART0 的 RX 功能 */
gpio_iomap_output(PA_1, GPIO_IOMAP_IN_UART0_IN);
```

1.4.9. gpio_iomap_inout()

设置 GPIO PIN 的 IOMAP INOUT 的功能，根据所需要的功能进行配置。例如，SPI 模块的 I/O 都是要求支持输入与输出，故要使用此函数配置该 GPIO PIN 为 INOUT 的功能。

- **函数原型**

```
int32 gpio_iomap_inout( uint32 pin,
                        enum gpio_iomap_in_func  in_func_sel,
                        enum gpio_iomap_out_func out_func_sel
                      )
```

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
in_func_sel	enum gpio_iomap_in_func	设置 GPIO PIN 的 iomap input 功能，见枚举 gpio_iomap_in_func
out_func_sel	enum gpio_iomap_out_func	设置 GPIO PIN 的 iomap output 功能，见枚举 gpio_iomap_out_func

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的 iomap inout 功能设置成功
RET_ERR	int32	GPIO PIN 的 iomap inout 功能设置失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的 iomap inout 功能设置失败

- **代码示例**

```
/* 配置 PA1 的 iomap_inout 的功能为 SPI0 的 CLK */
/* 即 PA1 作为 SPI0 的 CLK 功能 */
gpio_iomap_inout( PA_1,
                  GPIO_IOMAP_IN_SPI0_SCK_IN,
                  GPIO_IOMAP_OUT_SPI0_SCK_OUT);
```

1.4.10. gpio_request_pin_irq()

申请 GPIO PIN 中断。

- **函数原型**

```
int32 gpio_request_pin_irq( uint32              pin,
                            gpio_irq_hdl        handler,
                            uint32              data,
                            enum gpio_irq_event evt
                          )
```

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
handler	gpio_irq_hdl	中断句柄，中断产生后执行
data	uint32	中断句柄的参数
evt	enum gpio_irq_event	申请的中断类型，参考枚举 gpio_irq_event

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的申请中断成功
RET_ERR	int32	GPIO PIN 的申请中断失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的申请中断失败

- **代码示例**

```
void pin_irq_test_handle(int32 id, enum gpio_irq_event evt) {
    __NOP();
}
/* PA1 申请上升沿中断 */
gpio_request_pin_irq( PA_1,
                      (gpio_irq_hdl)pin_irq_test_handle,
                      0,
                      GPIO_IRQ_EVENT_RISE);
```

1.4.11. gpio_release_pin_irq()

释放 GPIO PIN 中断。

- **函数原型**

  int32 gpio_release_pin_irq(uint32 pin, enum gpio_irq_event evt)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
evt	enum gpio_irq_event	释放的中断类型,参考枚举 gpio_irq_event

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO PIN 的释放中断成功
RET_ERR	int32	GPIO PIN 的释放中断失败
-EINVAL	int32	因函数传参有误，故 GPIO PIN 的释放中断失败

- **代码示例**

```
/* PA1 释放上升沿中断 */
gpio_release_pin_irq(PA_1, GPIO_IRQ_EVENT_RISE);
```

1.4.12. gpio_ioctl()

依据 enum gpio_ioctl_cmd 中的命令，调用此函数对 GPIO 进行相关配置。

- **函数原型**

  int32 gpio_ioctl(uint32 pin, int32 cmd, int32 param1, int32 param2)

- **函数参数**

参数	类型	说明
pin	uint32	GPIO PIN，目前支持 GPIOA、GPIOB、GPIOC 的引脚
cmd	int32	GPIO 模块的配置命令，见枚举 gpio_ioctl_cmd
param1	uint32	配置参数 1，依据配置命令而定
param2	uint32	配置参数 2，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	GPIO 模块配置成功
RET_ERR	int32	GPIO 模块配置失败
-EINVAL	int32	因函数传参有误，故 GPIO 配置失败

- **代码示例**

```
/* 配置 PA1 为普通输出模式 */
gpio_set_dir(PA_1, GPIO_DIR_OUTPUT);
/* 配置 PA1 输出值进行一次翻转 */
gpio_ioctl(PA_1, GPIO_OUTPUT_TOGGLE, 1, 0);
```

2. UART 接口说明

本章节主要介绍 UART 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

2.1. Enum

2.1.1. uart_mode

enum uart_mode 描述了 UART 的工作模式。

枚举量	说明
UART_MODE_DUPLEX	UART 工作在全双工模式，即在同一时间可以接收和发送数据
UART_MODE_SIMPLEX_TX	UART 工作在单发送模式，即 UART 只执行数据发送的动作，忽视数据接收 @Note: 此模式下，发送使用 TX 引脚，忽视 RX 引脚上的数据变化
UART_MODE_SIMPLEX_RX	UART 工作在单接收模式，即 UART 只执行数据接收的动作，忽视数据发送 @Note: 此模式下，接收使用 RX 引脚，忽视 TX 引脚上的数据变化

2.1.2. uart_parity

enum uart_parity 描述了 UART 的校验模式。

枚举量	说明
UART_PARITY_NONE	UART 工作在无校验模式
UART_PARITY_ODD	UART 工作在奇校验模式。每个字节传送整个过程中 bit 为 1 的个数是奇数个，则校验位为 1，否则为 0
UART_PARITY_EVEN	UART 工作在偶校验模式。每个字节传送整个过程中 bit 为 1 的个数是偶数个，则校验位为 1，否则为 0

2.1.3. uart_stop_bit

enum uart_stop_bit 描述了 UART 停止位的 bit 数。

枚举量	说明
UART_STOP_BIT_1	UART 的停止位为 1bit
UART_STOP_BIT_2	UART 的停止位为 2bit

2.1.4. uart_data_bit

enum uart_data_bit 描述了 UART 停止位的 bit 数。

枚举量	说明
UART_DATA_BIT_8	UART 的数据位为 8bit
UART 的数据位为 9bit	UART 的数据位为 9bit @Note: 在此模式下，UART 校验模式必须为无校验模式

2.1.5. uart_irq_flag

enum uart_irq_flag 描述了 UART 可申请的中断类型，通过 uart_request_irq()和 uart_release_irq()函数使用。

枚举量	说明
UART_IRQ_FLAG_TX_BYTE	UART 发送完一帧数据，则产生中断 @Note: 若 UART 配置数据位为 8bit，一帧数据为 8bit； 若数据位为 9bit，一帧数据为 9bit
UART_IRQ_FLAG_TIME_OUT	UART 超过设定的超时时间后，还未接收到数据，则产生中断 @Note: 此中断只用于 UART 的接收

UART_IRQ_FLAG_DMA_TX_DONE	UART 使用 DMA 发送完成，则产生中断 @Note: 配置该中断之前，需要使用 uart_ioctl 开启 DMA
UART_IRQ_FLAG_DMA_RX_DONE	UART 使用 DMA 接收完成，则产生中断 @Note: 配置该中断之前，需要使用 uart_ioctl 开启 DMA
UART_IRQ_FLAG_FRAME_ERR	UART 接收一帧数据错误，则产生中断 @Note: 1.若 UART 配置数据位为 8bit，一帧数据为 8bit；若数据位为 9bit，一帧数据为 9bit 2.一帧数据错误，通常是指一帧数据的停止位 bit 数和设定 bit 数不符合
UART_IRQ_FLAG_RX_BYTE	UART 接收完一帧数据，则产生中断 @Note: 若 UART 配置数据位为 8bit，一帧数据为 8bit；若数据位为 9bit，一帧数据为 9bit

2.1.6. uart_ioctl_cmd

enum uart_ioctl_cmd 描述了 UART 的配置命令，通过调用 uart_ioctl()进行配置。

枚举量	说明
UART_IOCTL_CMD_SET_BAUDRATE	设置 UART 的波特率
UART_IOCTL_CMD_SET_DATA_BIT	设置 UART 的数据位 bit 数 @Note: 参考 enum uart_data_bit
UART_IOCTL_CMD_SET_PARITY	设置 UART 的校验模式 @Note: 参考 enum uart_parity
UART_IOCTL_CMD_SET_STOP_BIT	设置 UART 的停止位 bit 数 @Note: 参考 enum uart_stop_bit

UART_IOCTL_CMD_SET_TIME_OUT	设置 UART 接收的超时时间
UART_IOCTL_CMD_USE_DMA	设置 UART 是否使用 DMA 发送 @Note: 设置使用 DMA 传输，仅对 uart_puts()和 uart_gets()函数生效
UART_IOCTL_CMD_SET_WORK_MODE	设置 UART 的工作模式 @Note: 参考 enum uart_mode
UART_IOCTL_CMD_DATA_RDY	询问 UART 是否空闲

2.2. Define

2.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

2.3. Structure

2.3.1. uart_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-

putc	int32 函数指针	-
getc	int32 函数指针	-
puts	int32 函数指针	-
gets	int32 函数指针	-
ioctl	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

2.4. Function

2.4.1. uart_open()

初始化 UART。在使用 UART 模块之前，必须先调用此函数。UART 初始化成功后，默认配置为：工作模式为全双工模式、校验模式为无校验模式、数据位 8bit、停止位 1bit。若需要改变配置，可以在初始化完成后，使用 uart_ioctl 函数进行更改。

- **函数原型**

  int32 uart_open(struct uart_device *uart, uint32 baudrate)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
baudrate	uint32	UART 波特率的值

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块初始化成功
RET_ERR	int32	UART 模块初始化失败

- **代码示例**

```
struct uart_device    *uart_test;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
```

2.4.2. uart_close()

关闭 UART。调用此函数后，UART 将无法正常接发数据，所有的配置（包含中断相关配置）都会失效。

- **函数原型**

  int32 uart_close(struct uart_device *uart)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块关闭成功
RET_ERR	int32	UART 模块关闭失败

- **代码示例**

```
struct uart_device    *uart_test;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，关闭串口 */
uart_close((struct uart_device *)uart_test);
```

2.4.3. uart_putc()

UART 发送一帧数据。若 UART 配置数据位为 8bit，一帧数据为 8bit；若数据位为 9bit，一帧数据为 9bit。

- **函数原型**

  int32 uart_putc(struct uart_device *uart, int8 value)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
value	int8	UART 要发送的数据，单位为 1 帧

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块初始化成功
RET_ERR	int32	UART 模块初始化失败

- **代码示例**

```
struct uart_device    *uart_test;
uint8  tx_data = 0x55;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 发送一帧数据（8bit） */
uart_putc((struct uart_device *)uart_test, tx_data);
```

2.4.4. uart_getc()

UART 接收一帧数据。若 UART 配置数据位为 8bit，一帧数据为 8bit；若数据位为 9bit，一帧数据为 9bit。

- **函数原型**

  uint8 uart_getc(struct uart_device *uart)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
value	uint8	UART 模块接收数据成功，返回数据值

- **代码示例**

```
struct uart_device    *uart_test;
uint8  rx_data;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 接收一帧数据（8bit） */
rx_data = uart_getc((struct uart_device *)uart_test);
```

2.4.5. uart_puts()

UART 依据 DATA BUFFER 的地址和数据个数，发送数据。UART 发送数据的默认方式为 CPU 发送，若要使用 DMA 发送，需要使用 uart_ioctl()函数进行配置。

- **函数原型**

  int32 uart_puts(struct uart_device *uart, uint8* buf, uint32 len)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
buf	uint8	DATA BUFFER 的起始地址
len	uint32	要发送的数据个数，单位默认为 8bit

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块发送数据成功
RET_ERR	int32	UART 模块发送数据失败

- **代码示例**

```
struct uart_device    *uart_test;
uint8  tx_data[3] = {1, 2, 3};
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 发送数据（8bit），发送 3byte */
uart_puts((struct uart_device *)uart_test, tx_data, 3);
```

2.4.6. uart_gets()

UART 依据 DATA BUFFER 的地址和数据个数，接收数据。UART 接收数据的默认方式为 CPU 接收，若要使用 DMA 接收，需要使用 uart_ioctl()函数进行配置。

- **函数原型**

  int32 uart_gets(struct uart_device *uart, uint8* buf, uint32 len)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
buf	uint8	DATA BUFFER 的起始地址
len	uint32	要接收的数据个数，单位默认为 8bit

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块发送数据成功
RET_ERR	int32	UART 模块发送数据失败

- **代码示例**

```
struct uart_device    *uart_test;
uint8  rx_data[3] = {0};
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 发送数据（8bit），接收 3byte */
uart_gets((struct uart_device *)uart_test, rx_data, 3);
```

2.4.7. uart_ioctl()

依据 enum uart_ioctl_cmd 中的命令，调用此函数对 UART 模块进行相关配置。

- **函数原型**

```
int32 uart_ioctl( struct uart_device  *uart,
                  enum uart_ioctl_cmd  ioctl_cmd,
                  uint32               param1,
                  uint32               param2
                )
```

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
ioctl_cmd	enum uart_ioctl_cmd	UART 模块的配置命令，见枚举 uart_ioctl_cmd
param1	uint32	配置参数 1，依据配置命令而定
param2	uint32	配置参数 2，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块配置成功
RET_ERR	int32	UART 模块配置失败

- **代码示例**

```
struct uart_device    *uart_test;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 重新设置波特率：9600 */
uart_ioctl((struct uart_device *)uart_test, UART_IOCTL_CMD_SET_BAUDRATE,
9600, 0);
```

2.4.8. uart_request_irq()

依据 enum uart_irq_flag 的中断类型，调用此函数申请 UART 模块的中断。

- **函数原型**

```
int32 uart_request_irq( struct uart_device  *uart,
                        uart_irq_hdl         irq_hdl,
                        uint32               irq_flag,
                        uint32               irq_data
                      )
```

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
irq_hdl	uart_irq_hdl	中断句柄，中断产生后执行
irq_flag	uint32	申请的中断类型，参考枚举 uart_irq_flag
irq_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块申请中断成功
RET_ERR	int32	UART 模块申请中断失败

- **代码示例**

```
void uart_interrupt_func(int32 data) {
  __NOP();
}
struct uart_device    *uart_test;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 申请接收超时中断，中断句柄为 uart_interrupt_func，参数为 0 */
uart_request_irq((struct uart_device *)uart_test, uart_interrupt_func,
UART_IRQ_FLAG_TIME_OUT, 0);
```

2.4.9. uart_release_irq()

依据 enum uart_irq_flag 的中断类型，调用此函数释放 UART 模块的中断。

- **函数原型**

  int32 uart_release_irq(struct uart_device *uart, uint32 irq_flag)

- **函数参数**

参数	类型	说明
uart	struct uart_device	UART 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	释放的中断类型，参考枚举 uart_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	UART 模块关闭释放成功
RET_ERR	int32	UART 模块关闭释放失败

- **代码示例**

```
struct uart_device    *uart_test;
uart_test = (struct uart_device*)dev_get(HG_UART0_DEVID);
/* 通过 dev_get()获取 UART 的句柄，并配置波特率 115200，打开串口 */
uart_open((struct uart_device *)uart_test, 115200);
/* 关闭接收超时中断 */
uart_release_irq((struct uart_device *)uart_test,\
UART_IRQ_FLAG_TIME_OUT);
```

3. I2S 接口说明

本章节主要介绍 I2S 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

3.1. Enum

3.1.1. i2s_sample_bits

i2s_sample_bits 描述了音频数据的位数。

枚举量	说明
I2S_SAMPLE_BITS_8BITS	I2S 输出 8 位音频数据
I2S_SAMPLE_BITS_16BITS	I2S 输出 16 位音频数据
I2S_SAMPLE_BITS_24BITS	I2S 输出 24 位音频数据

3.1.2. i2s_sample_freq

i2s_sample_freq 描述了采样频率，即 WSCLK 的频率。

枚举量	说明
I2S_SAMPLE_FREQ_8K	I2S 采样频率为 8KHz
I2S_SAMPLE_FREQ_16K	I2S 采样频率为 16KHz
I2S_SAMPLE_FREQ_44_1K	I2S 采样频率为 44.1KHz
I2S_SAMPLE_FREQ_48K	I2S 采样频率为 48KHz

3.1.3. i2s_channel

i2s_channel 描述了 I2S 的声道模式。

枚举量	说明
I2S_CHANNEL_MONO	I2S 工作在单声道模式

I2S_CHANNEL_STEREO	I2S 工作在双声道模式

3.1.4. i2s_data_fmt

i2s_data_fmt 描述了 I2S 的数据格式。

枚举量	说明
I2S_DATA_FMT_I2S	I2S 的数据格式为标准 I2S 格式
I2S_DATA_FMT_MSB	I2S 的数据格式为左对齐格式
I2S_DATA_FMT_LSB	I2S 的数据格式为右对齐格式
I2S_DATA_FMT_PCM	I2S 的数据格式为 PCM 格式

3.1.5. i2s_mode

i2s_mode 描述了 I2S 的工作模式。

枚举量	说明
I2S_MODE_MASTER	I2S 工作在主机模式
I2S_MODE_SLAVE	I2S 工作在从机模式

3.1.6. i2s_ioctl_cmd

i2s_ioctl_cmd 描述了 I2S 的配置命令，通过调用 i2s_ioctl()进行配置。

枚举量	说明
I2S_IOCTL_CMD_SET_WSCLK_POL	设置 I2S 的 WSCLK 极性
I2S_IOCTL_CMD_SET_SAMPLE_BITS	设置 I2S 的音频数据的 bit 位数 @Note: 参考 enum i2s_sample_bits
I2S_IOCTL_CMD_SET_CHANNEL	设置 I2S 的声道模式 @Note:

	参考 enum i2s_channel
I2S_IOCTL_CMD_SET_DATA_FMT	设置 I2S 的数据格式 @Note: 参考 enum i2s_data_fmt
I2S_IOCTL_CMD_SET_DEBOUNCE	设置 I2S 的滤波功能 @Note: 此滤波功能仅作用于 I2S 从机；开启滤波时，模块会对 SCLK、WSCLK、DATA 进行滤波

3.1.7. i2s_irq_flag

i2s_irq_flag 描述了 I2S 可申请的中断类型，通过 i2s_request_irq() 和 i2s_release_irq()函数使用。

枚举量	说明
I2S_IRQ_FLAG_HALF	I2S 完成一半传输中断
I2S_IRQ_FLAG_FULL	I2S 完成全部传输中断

3.2. Define

3.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

3.3. Structure

3.3.1. i2s_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-
ioctl	int32 函数指针	-
read	int32 函数指针	-
write	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

3.4. Function

3.4.1. i2s_open()

初始化 I2S。在使用 I2S 模块之前，必须先调用此函数。I2S 初始化成功后，其中的默认配置为：数据格式为 I2S 格式；关闭滤波功能；I2S_WSCLK 时钟左通道为低电平，右通道为高电平；声道模式为立体声。若需要改变配置，可以在初始化完成后，使用 i2s_ioctl 函数进行更改。

- **函数原型**

```
int32 i2s_open( struct i2s_device    *i2s,
                enum i2s_mode         mode,
                enum i2s_sample_freq  frequency,
                enum i2s_sample_bits  bits
```

)

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄
mode	enum i2s_mode	I2S 的工作模式，参考枚举 i2s_mode
frequency	enum i2s_sample_freq	I2S 的采样频率，参考枚举 i2s_sample_freq
bits	enum i2s_sample_bits	I2S 的音频数据位数，参考枚举 i2s_sample_bits

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块初始化成功
RET_ERR	int32	I2S 模块初始化失败
-EINVAL	int32	I2S 模块初始化失败，传入参数有误

- **代码示例**

```
struct i2s_device *i2s_test = NULL;
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，并配置主机模式，采样率 44.1k，
音频数据位数为 16bit */
i2s_open(i2s_test, I2S_MODE_MASTER, I2S_SAMPLE_FREQ_44_1K, \
I2S_SAMPLE_BITS_16BITS);
```

3.4.2. i2s_close()

关闭 I2S。调用此函数后，I2S 将无法正常工作，所有的配置（包含中断相关配置）都会失效。

- **函数原型**

  int32 i2s_close(struct i2s_device *i2s)

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块关闭成功
RET_ERR	int32	I2S 模块关闭失败

- **代码示例**

```
struct i2s_device *i2s_test = NULL;
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，关闭 I2S0*/
i2s_close(i2s_test);
```

3.4.3. i2s_write()

I2S 依据 DATA BUFFER 的地址和数据个数，发送数据。用户可在 I2S 模块的完成一半传输中断时，切换下一次要发送的 DATA BUFFER 的地址和数据个数。I2S 模块会在本次传输完成后，自动切换成下一次的 DATA BUFFER 和数据个数。若用户在 I2S 模块的完成一半传输中断时，未进行切换地址和长度。I2S 模块会在本次传输完成后，重新载入本次的 DATA BUFFER 和数据个数。

注意：用户只能在 I2S 模块的完成一半传输中断时切换下一次的地址和数据，否则会导致发送异常。

- **函数原型**

  int32 i2s_write(struct i2s_device *i2s, const void* buf, uint32 len)

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄
buf	const void*	DATA BUFFER 的起始地址
len	uint32	要发送的数据个数，单位为 byte，要求 4byte 对齐

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块写入 BUFFER 地址和长度成功
RET_ERR	int32	I2S 模块写入 BUFFER 地址和长度失败

- **代码示例**

```
uint16 data[256];
struct i2s_device *i2s_test = NULL;
/* 随意初始化数据，只为举例理解，无实际意义 */
memset((void *)data, 0x55, sizeof(data)/sizeof(data[0]));
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，并配置主机模式，采样率 44.1k，
音频数据位数为 16bit */
i2s_open(i2s_test, I2S_MODE_MASTER, I2S_SAMPLE_FREQ_44_1K, \
I2S_SAMPLE_BITS_16BITS);
/* 发送 256*2byte */
i2s_write(i2s_test, (void *)data, sizeof(data)/sizeof(data[0]));
```

3.4.4. i2s_read()

I2S 依据 DATA BUFFER 的地址和数据个数，接收数据。用户可在 I2S 模块的完成一半传输中断时，切换下一次要接收的 DATA BUFFER 的地址和数据个数。I2S 模块会在本次传输完成后，自动切换成下一次的 DATA BUFFER 和数据个数。若用户在 I2S 模块的完成一半传输中断时，未进行切换地址和长度。I2S 模块会在本次传输完成后，重新载入本次的 DATA BUFFER

和数据个数。

注意：用户只能在 I2S 模块的完成一半传输中断时切换下一次的地址和数据，否则会导致发送异常。

- **函数原型**

  int32 i2s_read(struct i2s_device *i2s, void* buf, uint32 len)

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄
buf	void*	DATA BUFFER 的起始地址
len	uint32	要接收的数据个数，单位为 byte，要求 4byte 对齐

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块写入 BUFFER 地址和长度成功
RET_ERR	int32	I2S 模块写入 BUFFER 地址和长度失败

- **代码示例**

```
uint16 data[256];
struct i2s_device *i2s_test = NULL;
/* 随意初始化数据，只为举例理解，无实际意义 */
memset((void *)data, 0x00, sizeof(data)/sizeof(data[0]));
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，并配置主机模式，采样率 44.1kHz，
音频数据位数为 16bit */
i2s_open(i2s_test, I2S_MODE_MASTER, I2S_SAMPLE_FREQ_44_1K, \
I2S_SAMPLE_BITS_16BITS);
/* 接收 256*2byte */
i2s_read(i2s_test, (void *)data, sizeof(data)/sizeof(data[0]));
```

3.4.5. i2s_ioctl()

依据 enum i2s_ioctl_cmd 中的命令，调用此函数对 I2S 模块进行相关配置。

- **函数原型**

```
int32 i2s_ioctl(  struct i2s_device  *i2s,
                  uint32             cmd,
                  uint32             param,
                )
```

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄
cmd	uint32	I2S 模块的配置命令，见枚举 i2s_ioctl_cmd
param	uint32	配置参数，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块配置成功
RET_ERR	int32	I2S 模块配置失败

- **代码示例**

```
struct i2s_device *i2s_test = NULL;
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，并配置主机模式，采样率 44.1kHz，
音频数据位数为 16bit */
i2s_open(i2s_test, I2S_MODE_MASTER, I2S_SAMPLE_FREQ_44_1K, \
I2S_SAMPLE_BITS_16BITS);
/* 通过 i2s_ioctl，配置 I2S 的声道模式为单声道 */
i2s_ioctl(i2s_test,I2S_IOCTL_CMD_SET_CHANNEL, I2S_CHANNEL_MONO);
```

3.4.6. i2s_request_irq()

依据 enum i2s_irq_flag 的中断类型，调用此函数申请 I2S 模块的中断。

- **函数原型**

```
int32 i2s_request_irq(  struct i2s_device  *i2s,
                        uint32             irq_flag,
                        i2s_irq_hdl        irq_hdl,
                        uint32             irq_data
                      )
```

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	申请的中断类型，参考枚举 i2s_irq_flag
irq_hdl	i2s_irq_hdl	中断句柄，中断产生后执行
irq_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块申请中断成功
RET_ERR	int32	I2S 模块申请中断失败

- **代码示例**

```
void i2s_interrupt_func(uint32 irq, uint32 irq_data) {
  __NOP();
}
struct i2s_device *i2s_test = NULL;
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，并配置主机模式，采样率 44.1kHz，
音频数据位数为 16bit */
i2s_open(i2s_test, I2S_MODE_MASTER, I2S_SAMPLE_FREQ_44_1K, \
I2S_SAMPLE_BITS_16BITS);
```

```
/* I2S 申请完成一半传输中断，中断句柄为 i2s_interrupt_func，参数为 0 */
i2s_request_irq(i2s_test, I2S_IRQ_FLAG_HALF,\
(i2s_irq_hdl)i2s_interrupt_func, 0);
```

3.4.7. i2s_release_irq()

依据 enum i2s_irq_flag 的中断类型，调用此函数释放 I2S 模块的中断。

- **函数原型**

  int32 i2s_release_irq(struct i2s_device *i2s, uint32 irq_flag)

- **函数参数**

参数	类型	说明
i2s	struct i2s_device	I2S 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	释放的中断类型，参考枚举 i2s_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	I2S 模块中断关闭成功
RET_ERR	int32	I2S 模块中断关闭失败

- **代码示例**

```
struct i2s_device    *i2s_test;
i2s_test = (struct i2s_device*)dev_get(HG_IIS0_DEVID);
/* 通过 dev_get()获取 I2S0 的句柄，并配置主机模式，采样率 44.1kHz，
音频数据位数为 16bit */
i2s_open(i2s_test, I2S_MODE_MASTER, I2S_SAMPLE_FREQ_44_1K, \
I2S_SAMPLE_BITS_16BITS);
/* 关闭完成一半传输中断 */
i2s_release_irq(i2s_test, I2S_IRQ_FLAG_HALF);
```

4. PDM 接口说明

本章节主要介绍 PDM 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

4.1. Enum

4.1.1. pdm_sample_freq

pdm_sample_freq 描述了采样频率。

枚举量	说明
PDM_SAMPLE_FREQ_16K	PDM 采样频率为 16KHz
PDM_SAMPLE_FREQ_32K	PDM 采样频率为 32KHz
PDM_SAMPLE_FREQ_48K	PDM 采样频率为 48KHz

4.1.2. pdm_channel

pdm_channel 描述了工作模式选择。

枚举量	说明
PDM_CHANNEL_LEFT	PDM 工作在左声道模式
PDM_CHANNEL_RIGHT	PDM 工作在左声道模式
PDM_CHANNEL_STEREO	PDM 工作在立体声模式

4.1.3. pdm_irq_flag

pdm_irq_flag 描述了 PDM 可申请的中断类型，通过 pdm_request_irq() 和 pdm_release_irq()函数使用。

枚举量	说明
PDM_IRQ_FLAG_DMA_HF	PDM 完成一半传输中断
PDM_IRQ_FLAG_DMA_OV	PDM 完成全部传输中断

4.1.4. pdm_ioctl_cmd

pdm_ioctl_cmd 描述了 PDM 的配置命令，通过调用 pdm_ioctl()进行配置。

枚举量	说明
PDM_IOCTL_CMD_LR_CHANNEL_INTERCHANGE	PDM 左右声道互换

4.2. Define

4.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

4.3. Structure

4.3.1. pdm_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-

write	int32 函数指针	-
close	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-
ioctl	int32 函数指针	-

4.4. Function

4.4.1. pdm_open()

初始化 PDM。在使用 PDM 模块之前，必须先调用此函数。PDM 初始化成功后，其中的默认配置为：降频率比（fs）= 100；不互换左右声道。若需要改变配置，可以在初始化完成后，使用 pdm_ioctl 函数进行更改。

- **函数原型**

```
int32 pdm_open( struct pdm_device    *pdm,
                enum pdm_sample_freq  freq,
                enum pdm_channel      channel
                )
```

- **函数参数**

参数	类型	说明
pdm	struct pdm_device	PDM 的句柄。通常使用 dev_get()获取句柄
freq	enum pdm_sample_freq	PDM 的采样频率，参考枚举 pdm_sample_freq
channel	enum pdm_channel	PDM 的工作模式，参考枚举 pdm_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	PDM 模块初始化成功
RET_ERR	int32	PDM 模块初始化失败

- **代码示例**

```
struct pdm_device *pdm_test = NULL;
pdm_test = (struct pdm_device*)dev_get(HG_PDM0_DEVID);
/* 通过 dev_get()获取 PDM 的句柄，并配置立体声，采样率 16k */
pdm_open(pdm_test, PDM_SAMPLE_FREQ_16K, PDM_CHANNEL_STEREO);
```

4.4.2. pdm_read()

填入 DATA BUFFER 地址和长度，使能 PDM 开始传输。用户可在 PDM 模块的完成一半传输中断时，切换下一次要接收的 DATA BUFFER 的地址和数据个数。PDM 模块会在本次传输完成后，自动切换成下一次的 DATA BUFFER 和数据个数。若用户在 PDM 模块的完成一半传输中断时，未进行切换地址和长度。PDM 模块会在本次传输完成后，重新载入本次的 DATA BUFFER 和数据个数。

注意：用户只能在 PDM 模块的完成一半传输中断时切换下一次的地址和数据，否则会导致接收异常。

- **函数原型**

  int32 pdm_read(struct pdm_device *pdm, void *buf, uint32 len)

- **函数参数**

参数	类型	说明
pdm	struct pdm_device	PDM 的句柄。通常使用 dev_get()获取句柄
buf	void*	DATA BUFFER 的起始地址
len	uint32	要接收的数据个数，单位为 byte

- **返回值**

返回值	类型	说明
RET_OK	int32	PDM 模块写入 BUFFER 地址和长度成功
RET_ERR	int32	PDM 模块写入 BUFFER 地址和长度成功

- **代码示例**

```
uint16 data[256];
struct pdm_device *pdm_test = NULL;
/* 随意初始化数据，只为举例理解，无实际意义 */
memset((void *)data, 0x00, sizeof(data)/sizeof(data[0]));
pdm_test = (struct pdm_device*)dev_get(HG_PDM0_DEVID);
/* 通过 dev_get()获取 PDM 的句柄，并配置立体声，采样率 16k */
pdm_open(pdm_test, PDM_SAMPLE_FREQ_16K, PDM_CHANNEL_STEREO);
/* 接收 256*2byte */
pdm_read(pdm_test, (void *)data, sizeof(data)/sizeof(data[0]));
```

4.4.3. pdm_close()

关闭PDM。调用此函数后,PDM将无法正常工作,需要重新使用pdm_open()函数进行open。

- **函数原型**

  int32 pdm_close(struct pdm_device *pdm)

- **函数参数**

参数	类型	说明
pdm	struct pdm_device	PDM 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	PDM 模块关闭成功
RET_ERR	int32	PDM 模块关闭失败

- **代码示例**

```
struct pdm_device *pdm_test = NULL;
pdm_test = (struct pdm_device*)dev_get(HG_PDM0_DEVID);
/* 通过 dev_get()获取 PDM 的句柄，关闭 PDM*/
pdm_close(pdm_test);
```

4.4.4. pdm_request_irq()

依据 enum pdm_irq_flag 的中断类型，调用此函数申请 PDM 模块的中断。

- **函数原型**

```
int32 pdm_request_irq(  struct pdm_device *pdm,
                        enum pdm_irq_flag  flag,
                        pdm_irq_hdl        irq_hdl,
                        uint32             data
                        )
```

- **函数参数**

参数	类型	说明
pdm	struct pdm_device	PDM 的句柄。通常使用 dev_get()获取句柄
flag	enum pdm_irq_flag	申请的中断类型，参考枚举 pdm_irq_flag
irq_hdl	pdm_irq_hdl	中断句柄，中断产生后执行
data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	PDM 模块申请中断成功
RET_ERR	int32	PDM 模块申请中断失败

- **代码示例**

```
void pdm_interrupt_func(uint32 irq, uint32 irq_data) {
  __NOP();
}
struct pdm_device *pdm_test = NULL;
pdm_test = (struct pdm_device*)dev_get(HG_PDM0_DEVID);
/* 通过 dev_get()获取 PDM 的句柄，并配置立体声，采样率 16k */
pdm_open(pdm_test, PDM_SAMPLE_FREQ_16K, PDM_CHANNEL_STEREO);
/* 申请 PDM 模块的完成一半传输中断 */
pdm_request_irq(pdm_test, PDM_IRQ_FLAG_DMA_HF, pdm_interrupt_func, 0);
```

4.4.5. pdm_release_irq()

依据 enum pdm_irq_flag 的中断类型，调用此函数释放 PDM 模块的中断。

- **函数原型**

  int32 pdm_release_irq(struct pdm_device *pdm, enum pdm_irq_flag irq_flag)

- **函数参数**

参数	类型	说明
pdm	struct pdm_device	PDM 的句柄。通常使用 dev_get()获取句柄
irq_flag	enum pdm_irq_flag	释放的中断类型，参考枚举 pdm_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	PDM 模块中断关闭成功
RET_ERR	int32	PDM 模块中断关闭失败

- **代码示例**

```
struct pdm_device *pdm_test = NULL;
pdm_test = (struct pdm_device*)dev_get(HG_PDM0_DEVID);
/* 通过 dev_get()获取 PDM 的句柄，并配置立体声，采样率 16k */
pdm_open(pdm_test, PDM_SAMPLE_FREQ_16K, PDM_CHANNEL_STEREO);
/* 关闭 PDM 模块的完成一半传输中断 */
pdm_release_irq(pdm_test, PDM_IRQ_FLAG_DMA_HF);
```

4.4.6. pdm_ioctl()

依据 enum pdm_ioctl_cmd 中的命令，调用此函数对 PDM 模块进行相关配置。

- **函数原型**

```
int32 pdm_ioctl(struct pdm_device  *pdm,
                enum pdm_ioctl_cmd  cmd,
                uint32              param
                )
```

- **函数参数**

参数	类型	说明
pdm	struct pdm_device	PDM 的句柄。通常使用 dev_get()获取句柄
cmd	enum pdm_ioctl_cmd	PDM 模块的配置命令,见枚举 pdm_ioctl_cmd
param	uint32	配置参数，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	PDM 模块配置成功
RET_ERR	int32	PDM 模块配置失败

- **代码示例**

```
struct pdm_device *pdm_test = NULL;
pdm_test = (struct pdm_device*)dev_get(HG_PDM0_DEVID);
/* 通过 dev_get()获取 PDM 的句柄，并配置立体声，采样率 16k */
pdm_open(pdm_test, PDM_SAMPLE_FREQ_16K, PDM_CHANNEL_STEREO);
/* 配置左右声道互换 */
pdm_ioctl(pdm_test, PDM_IOCTL_CMD_LR_CHANNEL_INTERCHANGE, 1);
```

5. ADC 接口说明

本章节主要介绍 ADC 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

5.1. Enum

5.1.1. adc_irq_flag

adc_irq_flag 描述了 ADC 可申请的中断类型，通过 adc_request_irq() 和 adc_release_irq()函数使用。

枚举量	说明
ADC_IRQ_FLAG_SAMPLE_DONE	ADC 采样完成中断

5.1.2. adc_ioctl_cmd

adc_ioctl_cmd 描述了 ADC 的配置命令，通过调用 adc_ioctl()进行配置。

枚举量	说明
RESERVE	保留，供后续使用

5.1.3. adc_voltage_type

adc_voltage_type 描述了 ADC 可以采集的芯片内部电压类型，通过调用 adc_add_channel()添加采样通路。

注：

1. 除了采样芯片内部电压，ADC 也支持添加采样 I/O 电压的通路，通路名字采用 PA_0-PA_15，PB_6-PB_15，PC_0-PC_15 枚举值。

2. adc_voltage_type.h 中的芯片内部电压类型的枚举值必须从 0x101 开始。

3. 该枚举量位于“...sdk\include\chip\txw80x\adc_voltage_type.h”。

枚举量	说明
ADC_CHANNEL_RF_TEMPERATURE	ADC 采样 RF 温度通路

5.2. Define

5.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

5.3. Structure

5.3.1. adc_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	
add_channel	int32 函数指针	-
delete_channel	int32 函数指针	-

get_value	int32 函数指针	-
ioctl	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	

5.4. Function

5.4.1. adc_open()

初始化 ADC。在使用 ADC 模块之前，必须先调用此函数。

- **函数原型**

  int32 adc_open(struct adc_device *adc)

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块初始化成功
RET_ERR	int32	ADC 模块初始化失败

- **代码示例**

```
struct adc_device *adc_test = NULL;
adc_test = (struct adc_device*)dev_get(HG_ADC_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
```

5.4.2. adc_close()

关闭ADC。调用此函数后,ADC将无法正常工作,需要重新使用adc_open()函数进行open。

- **函数原型**

  int32 adc_close(struct adc_device *adc)

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块关闭成功
RET_ERR	int32	ADC 模块关闭失败

- **代码示例**

```
struct adc_device *adc_test = NULL;
adc_test = (struct adc_device*)dev_get(HG_ADC0_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
/* 通过 dev_get()获取 ADC 的句柄，关闭 ADC */
adc_close(adc_test);
```

5.4.3. adc_add_channel()

添加 ADC 采样通路。ADC 的采样通路支持芯片内部电压和 I/O 电压，具体见 enum adc_voltage_type。

注：

1. 无法添加重复的采样通道。

- **函数原型**

  int32 adc_add_channel(struct adc_device *adc, uint32 channel)

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄
channel	uint32	ADC 的采样通道，具体见枚举 adc_voltage_type

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块添加采样通道成功
RET_ERR	int32	ADC 模块添加采样通道失败

- **代码示例**

```
struct adc_device *adc_test = NULL;
adc_test = (struct adc_device*)dev_get(HG_ADC0_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
/* 添加采样 PA_0 电压的通路 */
adc_add_channel(adc_test, PA_0);
/* 添加采样芯片内部的 RF 温度通路 */
adc_add_channel(adc_test, ADC_CHANNEL_RF_TEMPERATURE);
```

5.4.4. adc_delete_channel()

删除已添加的 ADC 采样通路。删除通道后，该通道将无法进行 ADC 采样。

注：

1. 不能删除不存在的通道

- **函数原型**

  int32 adc_delete_channel(struct adc_device *adc, uint32 channel)

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄
channel	uint32	ADC 的采样通道，具体见枚举 adc_voltage_type

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块删除采样通道成功
RET_ERR	int32	ADC 模块删除采样通道失败

- **代码示例**

```
struct adc_device *adc_test = NULL;
adc_test = (struct adc_device*)dev_get(HG_ADC_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
/* 添加采样 PA_0 电压的通路 */
adc_add_channel(adc_test, PA_0);
/* 添加采样芯片内部的 RF 温度通路 */
adc_add_channel(adc_test, ADC_CHANNEL_RF_TEMPERATURE);
/* 删除已添加的采样 PA_0 电压的通路 */
adc_delete_channel(adc_test, PA_0);
```

5.4.5. adc_get_value()

获取当前采样通道的 ADC 采样值。

注：

1. ADC 是 12bit 转换精度，故 ADC 采样值最大值为 2^12 = 4095。
2. 不能采集未添加的采样通道的电压。
3. 通过此函数获取 ADC 采样值后，通过公式计算得到当前的电压值，如下：

   ADC 采样值=2048；参考电压=3.3V；

电压值 = (2048/4095)*3.3 = 1.65V

4. 如果采样通道为 ADC_CHANNEL_RF_TEMPERATURE，则获取的值为当前温度值，并非 ADC 采样值。

- **函数原型**

```
int32 adc_get_value(struct adc_device *adc,
                    uint32             channel,
                    uint32            *raw_data
                    )
```

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄
channel	uint32	ADC 的采样通道，具体见枚举 adc_voltage_type
raw_data	uint32*	ADC 采样完成后，返回的采样值

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块采样成功
RET_ERR	int32	ADC 模块采样失败

- **代码示例**

```
struct adc_device *adc_test = NULL;
uint32 adc_raw_data = 0;
uint32 voltage = 0;
adc_test = (struct adc_device*)dev_get(HG_ADC0_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
/* 添加采样 PA_0 电压的通路 */
adc_add_channel(adc_test, PA_0);
/* 获取 PA_0 的电压，通过 adc_raw_data 返回 ADC 采样值 */
adc_get_value(adc_test, PA_0, &adc_raw_data);
```

```
/* 计算此时 PA_0 上的电压值 */
voltage = (adc_raw_data / 4095) * 3.3;
```

5.4.6. adc_ioctl()

依据 enum adc_ioctl_cmd 中的命令，调用此函数对 ADC 模块进行相关配置。

- **函数原型**

```
int32 adc_ioctl(struct adc_device  *adc,
                enum adc_ioctl_cmd  ioctl_cmd,
                uint32              param1,
                uint32              param2
                )
```

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄
ioctl_cmd	enum adc_ioctl_cmd	ADC 模块的配置命令，见枚举 adc_ioctl_cmd
param1	uint32	配置参数，依据配置命令而定
param2	uint32	配置参数，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块配置成功
RET_ERR	int32	ADC 模块配置失败

- **代码示例**

```
/* ADC 模块暂未设置 ioctl_cmd，故无例程 */
```

5.4.7. adc_request_irq()

依据 enum adc_irq_flag 的中断类型，调用此函数申请 ADC 模块的中断。

- **函数原型**

```
int32 adc_request_irq(struct adc_device  *adc,
                      enum adc_irq_flag   irq_flag,
                      adc_irq_hdl         irq_hdl,
                      uint32              irq_data
                      )
```

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄
irq_flag	enum adc_irq_flag	申请的中断类型，参考枚举 adc_irq_flag
irq_hdl	adc_irq_hdl	中断句柄，中断产生后执行
irq_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块申请中断成功
RET_ERR	int32	ADC 模块申请中断失败

- **代码示例**

```
void adc_interrupt_func(uint32 irq, uint32 channel, uint32 irq_data) {
  __NOP();
}
struct adc_device *adc_test = NULL;
adc_test = (struct adc_device*)dev_get(HG_ADC0_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
/* 添加采样 PA_0 电压的通路 */
adc_add_channel(adc_test, PA_0);
/* 申请 ADC 模块的采样完成中断 */
adc_request_irq(adc_test, ADC_IRQ_FLAG_SAMPLE_DONE, \
adc_interrupt_func, 0);
```

5.4.8. adc_release_irq()

依据 enum adc_irq_flag 的中断类型，调用此函数释放 ADC 模块的中断。

- **函数原型**

  int32 adc_release_irq(struct adc_device *adc, enum adc_irq_flag irq_flag)

- **函数参数**

参数	类型	说明
adc	struct adc_device	ADC 的句柄。通常使用 dev_get()获取句柄
irq_flag	enum adc_irq_flag	释放的中断类型，参考枚举 adc_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	ADC 模块中断关闭成功
RET_ERR	int32	ADC 模块中断关闭失败

- **代码示例**

```
struct adc_device *adc_test = NULL;
adc_test = (struct adc_device*)dev_get(HG_ADC0_DEVID);
/* 通过 dev_get()获取 ADC 的句柄，打开 ADC */
adc_open(adc_test);
/* 添加采样 PA_0 电压的通路 */
adc_add_channel(adc_test, PA_0);
/* 关闭 ADC 模块的采样完成中断 */
adc_release_irq(adc_test, ADC_IRQ_FLAG_SAMPLE_DONE);
```

6. DVP 接口说明

本章节主要介绍 DVP 接口函数，枚举，宏，结构体的作用，它位于“SDK\include\hal”文件目录下。

6.1. Enum

6.1.1. dvp_ioctl_cmd

dvp_ioctl_cmd 描述了 DVP 接口的 ioctl 的配置命令，使用 dvp_device 结构体中的 ioctl 实现。

枚举量	说明
DVP_IOCTL_CMD_SET_FORMAT	设置 DVP 的输入格式（YUV,RGB,JPG）
DVP_IOCTL_CMD_RGB_2_YUV	设置 RGB 输入的情况下，将 RGB 数据转成 YUV
DVP_IOCTL_CMD_SET_SIZE	设置图片输入的大小
DVP_IOCTL_CMD_SET_ADR_1	设置 DVP 的数据接收地址 BUF1
DVP_IOCTL_CMD_SET_ADR_2	设置 DVP 的数据接收地址 BUF2
DVP_IOCTL_CMD_SET_SCEN	设置图片数据缩小，长宽都缩小一倍
DVP_IOCTL_CMD_SET_HSYNC_VAILD	HSYNC 的有效电平配置
DVP_IOCTL_CMD_SET_VSYNC_VAILD	VSYNC 的有效电平配置
DVP_IOCTL_CMD_SET_ONE_SAMPLE	设置单次图像输出，此配置打开后只会捕获一次 dvp 图像
DVP_IOCTL_CMD_SET_DEBOUNCE	PCLK 滤波，配置滤波周期
DVP_IOCTL_CMD_SET_YCBCR_MODE	YUV 格式下，输入源的亮度与色度排布顺序
DVP_IOCTL_CMD_DIS_UV_MODE	YUV 格式下，启动只捕获亮度（Y）数据
DVP_IOCTL_CMD_SET_FRAME_RATE	设置帧率控制，25%/50%/75%/100%捕获效率
DVP_IOCTL_CMD_SET_THRESHOLD	设置 Y,UV 数据的上下范围（小于取最低值，

	大于取最大值）
DVP_IOCTL_CMD_EX_D5_D6	DVP 的 D5,D6 数据引脚是否需要交换，与硬件接线相关
DVP_IOCTL_CMD_SET_JPEG_LEN	当输入格式配置为 JPG 后，设置 BUF 数据满中断的数据长度

6.2. Define

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

6.3. Structure

6.3.1. dvp_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
init	int32 函数指针	开启 dvp 时钟
baudrate	int32 函数指针	Mclk 的频率配置
open	int32 函数指针	Dvp 功能打开
close	int32 函数指针	Dvp 功能关闭
ioctl	int32 函数指针	Dvp 配置信息，功能调用
Request_irq	int32 函数指针	中断申请

Release_irq	int32 函数指针	中断释放

6.4. Function

6.4.1. dvp_init()

dvp 启动前配置，时钟打开

- **函数原型**

  int32 dvp_init(struct dvp_device *p_dvp)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	dvp 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块初始化成功
RET_ERR	int32	dvp 模块初始化失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 初始化 dvp
 */
dvp_init(dvp_test);
```

6.4.2. dvp_open()

dvp 功能启动

- **函数原型**

  int32 dvp_open(struct dvp_device *p_dvp)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	dvp 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块打开成功
RET_ERR	int32	dvp 模块打开失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 打开 dvp 模块
 */
dvp_open(dvp_test);
```

6.4.3. dvp_close()

dvp 功能关闭

- **函数原型**

  int32 dvp_close(struct dvp_device *p_dvp)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	dvp 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块关闭成功
RET_ERR	int32	dvp 模块关闭失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 关闭 dvp 模块
 */
dvp_close(dvp_test);
```

6.4.4. dvp_set_baudrate()

配置 dvp 输出时钟频率

- **函数原型**

  int32 dvp_set_baudrate(struct dvp_device *p_dvp,uint32 mclk)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
mclk	uint32	DVP 的输出 mclk 时钟频率

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 配置 dvp 的 mclk 时钟输出, 时钟为 24M
 */
dvp_set_baudrate(dvp_test,24000000);
```

6.4.5. dvp_set_size()

配置 dvp 配置图像显示坐标

- **函数原型**

  int32 dvp_set_size(struct dvp_device *p_dvp, uint32 x_s, uint32 y_s, uint32 x_e, uint32 y_e)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
x_s	uint32	图像启始横坐标
y_s	uint32	图像启始纵坐标
x_e	uint32	图像结束横坐标-1
y_e	uint32	图像结束纵坐标-1

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄，配置输入图像的分辨率，设置 VGA 的情况下，横坐标地址为 0 到 640-1，纵坐标地址为 0 到 480-1
 */
dvp_set_size(dvp_test,0,0,640-1,480-1);
```

6.4.6. dvp_set_addr1()/dvp_set_addr2()

配置 dvp 数据缓存位置

- **函数原型**

  int32 dvp_set_addr1(struct dvp_device *p_dvp, uint32 yuv_addr)

  int32 dvp_set_addr2(struct dvp_device *p_dvp, uint32 yuv_addr)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
yuv_addr	uint32	Buf 空间的地址

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
uint8 yuvbuf[2][IMAGE_W*8*2+IMAGE_W*8+16];
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 设置 dvp 的 buf 地址
 */
dvp_set_addr1(dvp_test,ybuf[0]);
dvp_set_addr2(dvp_test,ybuf[1]);
```

6.4.7. dvp_set_rgb2yuv()

配置 dvp 数据从 rgb565 转换成 yuv

- **函数原型**

  int32 dvp_set_rgb2yuv(struct dvp_device *p_dvp, uint8en)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
en	uint8	是否启动 rgb565 到 yuv 的转换

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 使能输入源从 rgb565 转换成 yuv 数据
 */
dvp_set_rgb2yuv(dvp_test,1);
```

6.4.8. dvp_set_format()

配置 dvp 输入源配置

- **函数原型**

  int32 dvp_set_format(struct dvp_device *p_dvp, uint8format)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
format	uint8	DVP 输入源格式 0:YUV 1:RGB565 2:JPEG

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 配置 dvp 输入源图像格式为 YUV
 */
dvp_set_format(dvp_test,0);
```

6.4.9. dvp_set_half_size()

配置 dvp 长宽倍缩

- **函数原型**

  int32 dvp_set_half_size(struct dvp_device *p_dvp, uint8en)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
en	uint8	DVP 输出长宽各缩小一半

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 配置输入图像的分辨率缩半功能,长宽各缩一半,
即 VGA 变 QVGA
 */
dvp_set_half_size(dvp_test,1);
```

6.4.10. dvp_set_vsync_polarity()/dvp_set_hsync_polarity()

配置 dvp 输入的 vs,hs 有效电平配置

- **函数原型**

  int32 dvp_set_vsync_polarity(struct dvp_device *p_dvp, uint8 high_valid)

  int32 dvp_set_hsync_polarity(struct dvp_device *p_dvp, uint8 high_valid)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
high_valid	uint8	DVP 的 Hsync 与 Vsync 的有效电平设置，1 为高有效

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 配置 vsync/hsync 的有效电平为高电平
 */
dvp_set_vsync_polarity(dvp_test,1);
dvp_set_hsync_polarity(dvp_test,1);
```

6.4.11. dvp_set_once_sampling()

配置 dvp 的单次捕获

- **函数原型**

  int32 dvp_set_once_sampling(struct dvp_device *p_dvp, uint8en)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
en	uint8	DVP 启动单次捕获，模块只会捕获一次图片

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 配置只采一帧图像
 */
dvp_set_once_sampling(dvp_test,1);
```

6.4.12. dvp_debounce_enable()

配置 dvp 的 pclk 滤波功能

- **函数原型**

  int32 dvp_debounce_enable(struct dvp_device *p_dvp, uint8en, uint8pixel)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
en	uint8	DVP 滤波功能使能
pixel	uint8	滤波周期

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 使能 dvp 的 pclk 滤波, 滤波周期为 2
 */
dvp_debounce_enable(dvp_test,1,2);
```

6.4.13. dvp_set_ycbcr()

当输入格式为 YUV，配置 dvp 的 YUV 数据排布

- **函数原型**

  int32 dvp_set_ycbcr(struct dvp_device *p_dvp, uint8mode)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
mode	uint8	YUV 数据排布 0:YUYV 1:YVYU 2:UYVY 3:VYUY

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 输入源为 yuv 情况下, yuv 的输入格式为 yuyv
 */
dvp_set_ycbcr(dvp_test,0);
```

6.4.14. dvp_unload_uv()

配置 dvp 的纯亮度采集

- **函数原型**

  int32 dvp_unload_uv(struct dvp_device *p_dvp, uint8en)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
en	uint8	使能纯亮度采集，DVP 无效化色彩数据的收集

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄,配置只采集亮度数据
 */
dvp_unload_uv(dvp_test,1);
```

6.4.15. dvp_frame_load_precent()

配置 dvp 的帧采样频率

- **函数原型**

  int32 dvp_frame_load_precent(struct dvp_device *p_dvp, uint8precent)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
precent	uint8	DVP 帧采样频率 0:100% 1:75% 2:50% 3:25%

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 配置采样率为 50%, 每两张图片只采一张
 */
dvp_frame_load_precent(dvp_test,2);
```

6.4.16. dvp_low_high_threshold()

配置 dvp 的 yuv 高低阈值

- **函数原型**

  int32 dvp_low_high_threshold(struct dvp_device *p_dvp,bool low_high)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
low_high	bool	DVP 使能高低阈值功能，函数内可再配置阈值范围 Byte[0]:Y 值 Byte[1]:U 值 Byte[2]:V 值

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 开启 yuv 数据阈值功能
 */
dvp_low_high_threshold(dvp_test,1);
```

6.4.17. dvp_set_exchange_d5_d6()

配置 dvp 的 d5 与 d6 引脚功能交换

- **函数原型**

  int32 dvp_set_exchange_d5_d6(struct dvp_device *p_dvp, uint8en)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
en	uint8	使能 d5 与 d6 的功能交换,与芯片的走线相关

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, dvp 的 IO 引脚，当 IO5->PB11,IO6->PB10，d5/d6
引脚不用交换，当 IO5->PB10,IO6->PB11，引脚交换功能使能
 */
dvp_set_exchange_d5_d6(dvp_test,1);
```

6.4.18. dvp_jpeg_mode_set_len()

当 dvp 的输入格式为 jpeg 时，配置 jpeg 的 buf 长度

- **函数原型**

  int32 dvp_jpeg_mode_set_len(struct dvp_device *p_dvp, uint32len)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
len	uint32	Jpeg 的 buf 长度，采够 buf 长度后，会产生 buf 满中断

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄，dvp 输入为 jpg 时，配置接收数据为 2048 时产生一个 buf 满中断
 */
dvp_jpeg_mode_set_len(dvp_test,2048);
```

6.4.19. dvp_request_irq()

dvp 的中断注册函数

- **函数原型**

```
int32 dvp_request_irq(struct dvp_device *p_dvp,
                      uint32irq_flag,
                      dvp_irq_hdl irq_hdl,
                      uint32 irq_data
                      )
```

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	子中断号： 0:hsync 中断 1:vsync 中断 2:帧完成中断，当输入源为 jpeg 时，为 buf 满中断 3:FIFO 半满中断 4:FIFO 溢出中断 5:SYNC 失效中断 6:JPEG 完成中断
irq_hdl	dvp_irq_hdl	注册的中断执行函数
irq_data	uint32	中断执行函数的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 接受到 vsync 有效信号后，执行中断处理函数
dvp_vsie_isr,传参为 dvp_test
 */
dvp_request_irq (dvp_test,1, (dvp_irq_hdl )&dvp_vsie_isr,dvp_test);
```

6.4.20. dvp_release_irq()

当 dvp 的输入格式为 jpeg 时，配置 jpeg 的 buf 长度

- **函数原型**

  int32 dvp_release_irq(struct dvp_device *p_dvp, uint32irq_flag)

- **函数参数**

参数	类型	说明
p_dvp	struct dvp_device	DVP 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	释放中断号

- **返回值**

返回值	类型	说明
RET_OK	int32	dvp 模块配置成功
RET_ERR	int32	dvp 模块配置失败

- **代码示例**

```
struct dvp_device *dvp_test = NULL;
dvp_test = (struct dvp_device*)dev_get(HG_DVP_DEVID);
/*
通过 dev_get()获取 dvp 的句柄, 将 vsync 的中断注销
 */
dvp_release_irq (dvp_test,1);
```

7. JPG 接口说明

本章节主要介绍 JPG 接口函数，枚举，宏，结构体的作用，它位于“SDK\include\hal”文件目录下。

7.1. Enum

7.1.1. jpg_ioctl_cmd

jpg_ioctl_cmd 描述了 JPG 接口的 ioctl 的配置命令，使用 jpg_device 结构体中的 ioctl 实现。

枚举量	说明
JPG_IOCTL_CMD_SET_ADR	设置 JPG 的数据地址
JPG_IOCTL_CMD_SET_QT	设置 QT 表的微调，每个质量 table 表能细分 16 个等级
JPG_IOCTL_CMD_SET_SIZE	设置 JPEG 的输入分辨率
JPG_IOCTL_CMD_UPDATE_QT	更换质量 table 表

7.2. Define

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

7.3. Structure

7.3.1. jpg_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
init	int32 函数指针	开启 jpg 时钟,并初始化各 table 表
open	int32 函数指针	打开 jpg 功能
close	int32 函数指针	关闭 jpg 功能
ioctl	int32 函数指针	Jpeg 配置信息，功能调用
request_irq	int32 函数指针	中断申请
release_irq	int32 函数指针	中断释放

7.4. Function

7.4.1. jpg_init()

jpg 初始化，jpg 时钟打开，霍夫曼表配置，初始量化表配置

- **函数原型**

  int32 jpg_init(struct jpg_device *p_jpg , uint32 table_idx, uint32 qt)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
table_idx	uint32	使用哪份量化表，默认有 6 份 quality_table
qt	uint32	对使用的量化表进行微调，微调有 16 个等级

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块初始化成功
RET_ERR	int32	jpg 模块初始化失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄, 打开 jpg,使用第 2 份默认 quality_table 表,微调到第 4 个等级, 等级 8 为不改变质量
 */
jpg_init(jpg_test,2,4);
```

7.4.2. jpg_open()

jpg 模块功能打开

- **函数原型**

  int32 jpg_open(struct jpg_device *p_jpg)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块打开成功
RET_ERR	int32	jpg 模块打开失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);

jpg_init(jpg_test,2,4);
。。。
。。。
。。。
/*
通过 dev_get()获取 jpg 的句柄，初始化 jpg，并配置好对应功能后，打开 jpg 功能
 */
jpg_open(jpg_test);
```

7.4.3. jpg_close()

jpg 模块功能关闭

- **函数原型**

  int32 jpg_close(struct jpg_device *p_jpg)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块关闭成功
RET_ERR	int32	jpg 模块关闭失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄,关闭 jpg 功能
 */
jpg_close(jpg_test);
```

7.4.4. jpg_updata_dqt()

jpg 模块量化表更新

- **函数原型**

  int32 jpg_updata_dqt(struct jpg_device *p_jpg,uint32 *dqtbuf)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
dqtbuf	uint32 *	要更新的量化表

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块更新量化表成功
RET_ERR	int32	jpg 模块更新量化表失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
extern char quality_tab[6][128];
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄,更新 quality_tab 的 6 份量化表里的第 0 份量化表
 */
jpg_updata_dqt(jpg_test, quality_tab[0]);
```

7.4.5. jpg_set_qt()

jpg 模块量化表微调

- **函数原型**

  int32 jpg_set_qt(struct jpg_device *p_jpg,uint32 qt)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
qt	uint32	微调等级，0~15，8 为量化表不变，向上减少压缩等级，向下提高压缩等级

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块细调量化表成功
RET_ERR	int32	jpg 模块细调量化表失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
extern char quality_tab[6][128];
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
jpg_updata_dqt(jpg_test, quality_tab[0]);
/*
通过 dev_get()获取 jpg 的句柄,针对当前选用的 0 号量化表进行质量微调，提高一个等级的图像质量
 */
jpg_set_qt(jpg_test,7);
```

7.4.6. jpg_set_size()

jpg 模块修改输入源分辨率

- **函数原型**

  int32 jpg_set_size(struct jpg_device *p_jpg,uint32 h,uint32 w)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
h	uint32	输入的图像高度
w	uint32	输入的图像宽度

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块设置 size 成功
RET_ERR	int32	jpg 模块设置 size 失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄,配置 jpg 模块的输入源分辨率为 VGA
 */
jpg_set_size(jpg_test,480,640);
```

7.4.7. jpg_set_addr()

jpg 模块配置输出 buf 的位置，和触发满中断的数据量

函数原型

int32 jpg_set_addr(struct jpg_device *p_jpg,uint32 addr,uint32 buflen)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
addr	uint32	Jpg 模块生成的数据存放位置
buflen	uint32	Jpg 模块数据满中断的数据量配置

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块设置成功
RET_ERR	int32	jpg 模块设置失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
uint8  jpgbuf[2048];
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄,配置 jpg 模块的输出 buf 位置为 jpgbuf，数据满中断长度为 2048
 */
jpg_set_size(jpg_test, jpgbuf ,2048);
```

7.4.8. jpg_request_irq()

jpg 模块中断请求函数

- **函数原型**

```
int32 jpg_request_irq(struct jpg_device *p_jpg,
                      jpg_irq_hdlirq_hdl,
                      uint32 irq_flags,
                      void *irq_data
                      )
```

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
irq_hdl	jpg_irq_hdl	Jpg 对应中断处理函数
irq_flags	uint32	Jpg 模块子中断号 BIT（0）:帧完成中断 BIT（1）：帧 buf 数据满中断

		BIT（2）：帧错误中断
irq_data	void *	中断处理函数的传参

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块子中断注册成功
RET_ERR	int32	jpg 模块子中断注册失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
uint8  jpgbuf[2048];
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄,配置 jpg 模块的中断注册函数，在帧结束执行中断函数 jpg_done_isr,传参为 jpg_test
 */
jpg_request_irq(jpg_test, (jpg_irq_hdl )&jpg_done_isr,BIT(0),jpg_test);
```

7.4.9. jpg_release_irq()

jpg 模块中断释放函数

- **函数原型**

  int32 jpg_release_irq(struct jpg_device *p_jpg,uint32 irq_flags)

- **函数参数**

参数	类型	说明
p_jpg	struct jpg_device	JPG 的句柄。通常使用 dev_get()获取句柄
irq_flags	uint32	Jpg 模块子中断号 BIT（0）:帧完成中断 BIT（1）：帧 buf 数据满中断 BIT（2）：帧错误中断

- **返回值**

返回值	类型	说明
RET_OK	int32	jpg 模块中断释放成功
RET_ERR	int32	jpg 模块中断释放失败

- **代码示例**

```
struct jpg_device *jpg_test = NULL;
extern char quality_tab[6][128];
jpg_test = (struct jpg_device*)dev_get(HG_JPG_DEVID);
/*
通过 dev_get()获取 jpg 的句柄,释放帧结束中断
 */
jpg_release_irq(jpg_test,BIT(0));
```

8. CRC 接口说明

本章节主要介绍 CRC 的函数、枚举、宏、结构体的作用。它们位于“sdk\include\hal”文件目录下。

8.1. Enum

8.1.1. CRC_DEV_FLAGS

enum CRC_DEV_FLAGS 描述了 CRC 的工作模式。

枚举量	说明
CRC_DEV_FLAGS_CONTINUE_CALC	CRC 工作在持续运算模式，主要用于多个非连续的数据 buffer 计算总 CRC @Note: 此模式下，CRC 多项式配置会沿用之前配置的多项式

8.1.2. CRC_DEV_TYPE

enum CRC_DEV_TYPE 描述了 CRC 的校验模式。

枚举量	说明	
CRC_TYPE_CRC5_USB	CRC-5/USB	x5+x2+1
CRC_TYPE_CRC7_MMC	CRC-7/MMC	x7+x3+1
CRC_TYPE_CRC8_MAXIM	CRC-8/MAXIM	x8+x5+x4+1
CRC_TYPE_CRC8	CRC-8	x8+x2+x+1
CRC_TYPE_CRC16	CRC-16/MAXIM	x16+x15+x2+1
CRC_TYPE_CRC16_CCITT	CRC-16/CCITT	x16+x12+x5+1
CRC_TYPE_CRC16_MODBUS	CRC-16/MODBUS	x16+x15+x2+1
CRC_TYPE_CRC32_WINRAR	CRC-32 x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5	

	+x4+x2+x+1
CRC_TYPE_TCPIP_CHKSUM	计算 TCP/IP 包校验和

8.2. Define

8.2.1. MACRO

8.3. Structure

8.3.1. crc_dev

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
hold	int32 函数指针	此函数会锁住 CRC 资源，通常在需要连续多次计算 CRC 时使用，使用完后需要主动解除 hold
calc	int32 函数指针	此函数用于计算 CRC

8.4. Function

8.4.1. crc_dev_hold()

锁住 CRC 资源，通常在需要连续多次计算 CRC 时使用，使用完后需要主动解除 hold.

- **函数原型**

  int32 crc_dev_hold(struct crc_dev *dev, uint16 *cookie, uint8 hold)

- **函数参数**

参数	类型	说明
Dev	struct uart_device	CRC 的句柄。通常使用 dev_get()获取句柄

cookie	uint16 *	Cookie 相同才能解除 hold
Hold	uint8	1： hold， 0：unHold

- **返回值**

返回值	类型	说明
RET_OK	int32	成功
RET_ERR	int32	失败

8.4.2. crc_dev_calc()

计算 CRC 或者 CheckSum。

- **函数原型**

int32 crc_dev_calc(struct crc_dev *dev, struct crc_dev_req *req, uint32 *crc_val, uint32 flags)

- **函数参数**

参数	类型	说明
Dev	struct uart_device	CRC 的句柄。通常使用 dev_get()获取句柄
req	struct crc_dev_req *	配置 CRC 校验请求：buffer 地址、长度、CRC 类型
crc_val	uint32 *	用于返回 CRC 结果
flags	uint32	enum CRC_DEV_FLAGS

- **返回值**

返回值	类型	说明
RET_OK	int32	成功
RET_ERR	int32	失败

- **代码示例**

```
uint32 crc = 0xffff;
struct crc_dev_req req;
struct crc_dev *crcdev = (struct crc_dev *)dev_get(HG_CRC_DEVID);
if (crcdev) {
req.type = CRC_TYPE_CRC16_MODBUS;
req.data = data;
req.cookie = *data;
req.len  = len;
crc_dev_hold(crcdev, &req.cookie, 1);
crc_dev_calc(crcdev, &req, &crc, 0);
crc_dev_calc(crcdev, &req, &crc, CRC_DEV_FLAGS_CONTINUE_CALC);
crc_dev_calc(crcdev, &req, &crc, CRC_DEV_FLAGS_CONTINUE_CALC);
crc_dev_hold(crcdev, &req.cookie, 0);
} else {
os_printf("no crc dev\r\n");
}
```

9. AES 接口说明

本章节主要介绍 AES 的函数、枚举、宏、结构体的作用。它们位于“sdk\include\hal”文件目录下。

9.1. Enum

9.1.1. SYSAES_KEY_LEN

enum SYSAES_KEY_LEN 描述了 AES 密钥长度。

枚举量	说明
AES_KEY_LEN_BIT_128	AES 密钥长度:128 bit @Note:
AES_KEY_LEN_BIT_192	AES 密钥长度:192 bit @Note:
AES_KEY_LEN_BIT_256	AES 密钥长度:256bit @Note:

9.1.2. SYSAES_MODE

enum SYSAES_MODE 描述了 SYSAES_MODE 的加解密模式。

枚举量	说明
AES_MODE_ECB	
AES_MODE_CBC	

9.2. Define

9.2.1. MACRO

9.3. Structure

9.3.1. sysaes_para

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
mode	enum SYSAES_MODE	AES 加解密模式
key_len	enum SYSAES_KEY_LEN	AES 密钥长度
src	Int8 指针	数据源地址
Dest	Int8 指针	数据目的地址
block_num	uint32	AES Block 个数(1 一个 Block 等于 16 Byte)
Key	Int8 指针	AES 密钥地址
Iv	Int8 指针	AES IV 地址（AES CBC 模式使用）

9.3.2. sysaes_dev

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
encrypt	int32 函数指针	AES 加密
decrypt	int32 函数指针	AES 解密

9.4. Function

9.4.1. sysaes_encrypt（）

AES 加密。

- **函数原型**

  int32(*encrypt)(struct sysaes_dev *dev, struct sysaes_para *para);

- **函数参数**

参数	类型	说明
Dev	struct sysaes_dev *	AES 的句柄。通常使用 dev_get()获取句柄
para	struct sysaes_para *	AES 加密数据地址、密钥地址、加密模式等配置

- **返回值**

返回值	类型	说明
RET_OK	int32	成功
RET_ERR	int32	失败

9.4.2. sysaes_decrypt()

AES 解密。

- **函数原型**

  int32(*decrypt)(struct sysaes_dev *dev, struct sysaes_para *para);

- **函数参数**

参数	类型	说明
Dev	struct sysaes_dev *	AES 的句柄。通常使用 dev_get()获取句柄
para	struct sysaes_para *	AES 解密数据地址、密钥地址、加密模式等配置

- **返回值**

返回值	类型	说明
RET_OK	int32	成功
RET_ERR	int32	失败

- **代码示例**

```
struct sysaes_para para;
   uint8 *src = os_malloc(ALIGN(len, 16));
   uint8 *dest = os_malloc(ALIGN(len, 16));
   uint8 *k = os_malloc(ALIGN(len, 16));
   struct sysaes_dev *aes = (struct sysaes_dev
           *)dev_get(HG_HWAES_DEVID);
   if (aes && key && src && dest) {
       para.mode = AES_MODE_ECB;
       para.src = src;
       para.dest = dest;
       para.block_num = (len + 15) / 16; //round up
       para.key = k;
       ret = (en ? sysaes_encrypt(aes, &para) : sysaes_decrypt(aes,
           &para));
       if (ret == RET_OK) {
           os_memcpy(out, dest, len);
       }
   }
   if(src) os_free(src);
   if(dest) os_free(dest);
   if(k) os_free(k);
```

10. SDHOST 接口说明

10.1. Enum

10.2. Define

10.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

10.3. Structure

10.3.1. sdh_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
init	int32 函数指针	开启 jpg 时钟,并初始化各 table 表
open	int32 函数指针	打开 sdhost 功能
close	int32 函数指针	关闭 sdhost 功能
ioctl	int32 函数指针	sdhost 配置信息，功能调用
cmd	int32 函数指针	sdhost 命令发送
write	int32 函数指针	DMA 发送数据到 sd 从机
read	int32 函数指针	从 sd 从机中读取数据

freq_min	uint32	最低速率，目前为 400K，用于初始化
freq_max	uint32	最高速率
valid_ocr	uint32	电压范围
iocfg	rt_mmcsd_io_cfg 结构	Sdhost 配置信息
card_capacity	uint32	卡容量
new_lba	uint32	最新的读写操作地址
sd_opt	uint8	读写状态标识
data	rt_mmcsd_data 结构	读写操作的数据信息
dat_sema	os_semaphore	用于读写操作完成唤醒

10.4. Function

10.4.1. sdhost_io_func_init()

sd host 的 io 功能初始化

- **函数原型**

  void sdhost_io_func_init(uint32 req)

- **函数参数**

参数	类型	说明
req	uint32	0:1 线模式 IO 初始化 1:4 线模式 IO 初始化

- **代码示例**

```
struct jpg_device *sdh_test = NULL;
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, 判断 sdh 是否配置为 4 线模式,并使能对应的 IO
 */
sdhost_io_func_init(sdh_test->flags&MMCSD_BUSWIDTH_4);
```

10.4.2. sdhost_cmd

sdhost 命令发送操作

- **函数原型**

int32 (*cmd)(struct sdh_device *sdhost,struct rt_mmcsd_cmd *cmd);

- **函数参数**

参数	类型	说明
sdhost	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄
cmd	rt_mmcsd_cmd	命令的参数结构，命令指令值，应答值，参数值

- **返回值**

返回值	类型	说明
RET_OK	int32	Sdhost 命令发送成功
RET_ERR	int32	Sdhost 命令发送失败

- **代码示例**

```
/*发送 cmd7*/
uint32 send_select_card(struct sdh_device * host)
{
    struct rt_mmcsd_cmd cmd;
     int ret = 0;
    memset(&cmd, 0, sizeof(struct rt_mmcsd_cmd));
    cmd.cmd_code = SELECT_CARD;
    if (host->rca)
    {
           cmd.arg = host->rca << 16;
            cmd.flags = RESP_R1 | CMD_AC;
     }
    else
    {
            cmd.arg = 0;
            cmd.flags = RESP_NONE | CMD_AC;
```

```
    }
      if(host->cmd)
        ret = host->cmd(host,&cmd);
    return ret;
}
```

10.4.3. sd_multiple_read()

sd host 发送读命令，进行数据读取

- **函数原型**

```
int sd_multiple_read(struct sdh_device * host,
                     uint32 lba,
                     uint32 len,
                     uint8* buf
                     )
```

- **函数参数**

参数	类型	说明
host	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄
lba	uint32	读取的卡地址
len	uint32	要读取的数据长度
buf	uint8 *	数据读取到的位置，此位置不能设置为 psram

- **返回值**

返回值	类型	说明
RET_OK	int32	Sdhost 读取数据成功
RET_ERR	int32	Sdhost 读取数据失败

- **代码示例**

```
struct sd_device *sdh_test = NULL;
uint8 buf[1024];
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, 读取第 4 号 sector 起的数据，读取 1024，存放到 buf 里
 */
sd_multiple_read((struct sdh_device*)sdh_test,4,1024,buf);
```

10.4.4. sd_multiple_write()

sd host 发送写命令，进行数据写入

- **函数原型**

```
int sd_multiple_write(struct sdh_device * host,
                      uint32 lba,
                      uint32 len,
                      uint8* buf
                      )
```

- **函数参数**

参数	类型	说明
host	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄
lba	uint32	写入的卡地址
len	uint32	要写入的数据长度
buf	uint8 *	要写入数据的源数据位置,此位置不能设置为 psram

- **返回值**

返回值	类型	说明
RET_OK	int32	Sdhost 写数据成功
RET_ERR	int32	Sdhost 写数据失败

- **代码示例**

```
struct sd_device *sdh_test = NULL;
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, 写第 4 号 sector 起的数据,写入数据量为 1024,
源数据在 source_buf 里
 */
sd_multiple_write((struct sdh_device*)sdh_test,4,1024,source_buf);
```

10.4.5. sd_set_clk()

设置 sd host 的输出时钟

- **函数原型**

```
void sd_set_clk(struct sdh_device * host,uint32_t clk)
```

- **函数参数**

参数	类型	说明
host	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄
clk	uint32_t	配置 sd host 时钟频率

- **代码示例**

```
struct sd_device *sdh_test = NULL;
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, 配置 sd host 的输出时钟为 24M
 */
sd_set_clk((struct sdh_device*)sdh_test,24000000);
```

10.4.6. sd_set_bus_width()

配置 sd host 的总线为 1 线还是 4 线

- **函数原型**

void sd_set_bus_width(struct sdh_device * host,uint32_t width)

- **函数参数**

参数	类型	说明
host	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄
width	uint32_t	1 线：MMCSD_BUS_WIDTH_1 4 线：MMCSD_BUS_WIDTH_4

- **代码示例**

```
struct sd_device *sdh_test = NULL;
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, 配置 sd host 的 1 线模式
 */
sd_bus_width((struct sdh_device*)sdh_test, MMCSD_BUS_WIDTH_1);
```

10.4.7. sd_open

sd host 模块功能打开

- **函数原型**

int32 (*open)(struct sdh_device *sdhost,uint8 bus_w);

- **函数参数**

参数	类型	说明
sdhost	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄
bus_w	uint8	总线配置，1 为 1 线，4 为 4 线

- **返回值**

返回值	类型	说明
RET_OK	int32	Sdhost 打开成功
RET_ERR	int32	Sdhost 打开失败

- **代码示例**

```
struct jpg_device *sdh_test = NULL;
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, sdh 模块打开，使能为 1 线
 */
sdh_test->open(sdh_test,1)
```

10.4.8. sd_close

sd host 模块功能打开

- **函数原型**

int32 (*close)(struct sdh_device *sdhost);

- **函数参数**

参数	类型	说明
sdhost	struct sdh_device	SDHOST 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	Sdhost 关闭成功
RET_ERR	int32	Sdhost 关闭失败

- **代码示例**

```
struct jpg_device *sdh_test = NULL;
sdh_test = (struct sdh_device*)dev_get(HG_SDC_HOST_DEVID);
/*
通过 dev_get()获取 SDH 的句柄, sdh 模块关闭
 */
sdh_test->close(sdh_test)
```

11. I2C 接口说明

本章节主要介绍 I2C 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

11.1. Enum

11.1.1. i2c_mode

i2c_mode 描述了工作模式。

枚举量	说明
IIC_MODE_MASTER	I2C 工作在主机模式
IIC_MODE_SLAVE	I2C 工作在从机模式

11.1.2. i2c_addr_mode

i2c_mode 描述了地址格式。在 I2C 主机模式，表示从机的地址格式；在 I2C 从机模式，表示自身的地址格式。

枚举量	说明
IIC_ADDR_7BIT	I2C 地址格式为 7bit
IIC_ADDR_10BIT	I2C 地址格式为 10bit

11.1.3. i2c_ioctl_cmd

i2c_ioctl_cmd 描述了 I2C 的配置命令，通过调用 i2c_ioctl()进行配置。

枚举量	说明
IIC_SDA_OUTPUT_DELAY	I2C 的 SDA 延时输出数据，仅在 I2C 主机模式下使用

IIC_STRONG_OUTPUT	I2C 开启或关闭强输出模式 @Note: 不能将同为强输出模式的 I2C 设备相连接
IIC_FILTERING	I2C 开启或关闭滤波
IIC_SET_DEVICE_ADDR	设置地址。在主机模式下，设置从机的地址； 在从机模式下，设置自身的地址。

11.1.4. i2c_irq_flag

i2c_irq_flag 描述了 I2C 可申请的中断类型，通过 i2c_request_irq() 和 i2c_release_irq()函数使用。

枚举量	说明
I2C_IRQ_FLAG_TX_DONE	I2C 的发送完成中断
I2C_IRQ_FLAG_RX_DONE	I2C 的接收完成中断
I2C_IRQ_FLAG_RX_NACK	接收到 NACK 中断
I2C_IRQ_FLAG_RX_ERROR	接收 BUFFER 溢出中断，阈值为 40bit
I2C_IRQ_FLAG_DETECT_STOP	检测到总线上有 STOP 信号中断

11.2. Define

11.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

11.3. Structure

11.3.1. i2c_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-
send_stop	int32 函数指针	-
baudrate	int32 函数指针	-
read	int32 函数指针	-
write	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

11.4. Function

11.4.1. i2c_open()

初始化 I2C。在使用 I2C 模块之前，必须先调用此函数。I2C 初始化成功后，其中的默认配置为：不开启强输出功能。若需要改变配置，可以在初始化完成后，使用 i2c_ioctl 函数进行更改。

- **函数原型**

```
int32 i2c_open( struct i2c_device  *i2c,
                enum i2c_mode       mode,
                enum i2c_addr_mode  addr_mode,
                uint32              addr )
```

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
mode	enum i2c_mode	I2C 的工作模式，参考枚举 i2c_mode
addr_mode	enum i2c_addr_mode	I2C 的地址格式，参考枚举 i2c_addr_mode
addr	uint32	I2C 设备的地址 @Note: I2C 主机模式，表示从机的地址格式；I2C 从机模式，表示自身的地址格式

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块初始化成功
RET_ERR	int32	I2C 模块初始化失败

- **代码示例**

```
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
```

11.4.2. i2c_close()

关闭I2C。调用此函数后,I2C将无法正常工作,需要重新使用i2c_open()函数进行open。

- **函数原型**

  int32 i2c_close(struct i2c_device *i2c)

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块关闭成功
RET_ERR	int32	I2C 模块关闭失败

- **代码示例**

```
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 关闭 I2C */
i2c_close(i2c_test);
```

11.4.3. i2c_set_baudrate()

设置 I2C 的波特率。I2C 模块必须要配置波特率，才能够正常使用。

- **函数原型**

  int32 i2c_set_baudrate(struct i2c_device *i2c, uint32 baudrate)

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
baudrate	uint32	I2C 设置的波特率值

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块设置波特率成功
RET_ERR	int32	I2C 模块设置波特率失败

- **代码示例**

```
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 设置波特率为 300KHz */
i2c_set_baudrate(i2c_test, 300000);
```

11.4.4. i2c_ioctl()

依据 enum adc_ioctl_cmd 中的命令，调用此函数对 I2C 模块进行相关配置。

- **函数原型**

  int32 i2c_ioctl(struct i2c_device *i2c, uint32 cmd, uint32 param)

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
cmd	enum i2c_mode	I2C 模块的配置命令，见枚举 i2c_ioctl_cmd
param	uint32	配置参数，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块配置成功
RET_ERR	int32	I2C 模块配置失败

- **代码示例**

```
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 设置波特率为 300KHz */
i2c_set_baudrate(i2c_test, 300000);
/* 设置成强输出模式 */
i2c_ioctl(i2c_test, IIC_STRONG_OUTPUT, 1);
```

11.4.5. i2c_write()

I2C 依据发送数据的相关配置，进行发送。

注：

1. 主机模式下，I2C 模块发送数据之前，会先发送从机设备地址（从机设备地址由 i2c_open()函数配置）。

- **函数原型**

```
int32 i2c_write(struct i2c_device *i2c,
                int8              *addr,
                uint32             addr_len,
                int8              *buf,
                uint32             buf_len
                )
```

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
addr	int8*	要访问的地址，仅针对主机模式有效
addr_len	uint32	地址长度，单位为 byte，仅针对主机模式有效
buf	int8*	要发送的数据 BUFFER 的起始地址
buf_len	uint32	数据长度，单位为 byte

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块发送成功
RET_ERR	int32	I2C 模块发送失败

- **代码示例**

```
uint8 tx_buf[256];
uint32 i = 0;
uint8 addr = 0x30;
/* 初始化 tx_buf，数值无实际意义 */
for (i = 0; i < sizeof(tx_buf)/sizeof(tx_buf[0]); i++) {
    tx_buf[i] = 0x55;
}
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 设置波特率为 300KHz */
i2c_set_baudrate(i2c_test, 300000);
/* 向器件地址为 0x80 的 I2C 从设备的 0x30 地址开始，写 256byte 数据 */
i2c_write(i2c_test, &addr, 1, tx_buf, sizeof(tx_buf)/sizeof(tx_buf[0]));
```

11.4.6. i2c_read()

I2C 依据接收数据的相关配置，进行接收。

注：

1. 主机模式下，I2C 模块接收数据之前，会先发送从机设备地址（从机设备地址由 i2c_open()函数配置）。

- **函数原型**

```
int32 i2c_read( struct i2c_device *i2c,
                int8              *addr,
                uint32             addr_len,
                int8              *buf,
                uint32             buf_len
                )
```

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
addr	int8*	要访问的地址，仅针对主机模式有效
addr_len	uint32	地址长度，单位为 byte，仅针对主机模式有效
buf	int8*	要接收的数据 BUFFER 的起始地址
buf_len	uint32	数据长度，单位为 byte

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块接收成功
RET_ERR	int32	I2C 模块接收失败

- **代码示例**

```
uint8 rx_buf[256];
uint32 i = 0;
uint8 addr = 0x30;
/* 初始化 rx_buf = 0 */
for (i = 0; i < sizeof(rx_buf)/sizeof(rx_buf[0]); i++) {
    rx_buf[i] = 0x00;
}
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备
```

```
地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 设置波特率为 300KHz */
i2c_set_baudrate(i2c_test, 300000);
/* 向器件地址为 0x80 的 I2C 从设备的 0x30 地址开始，读 256byte 数据 */
i2c_read(i2c_test, &addr, 1, rx_buf, sizeof(rx_buf)/sizeof(rx_buf[0]));
```

11.4.7. i2c_request_irq()

依据 enum i2c_irq_flag 的中断类型，调用此函数申请 I2C 模块的中断。

- **函数原型**

```
int32 i2c_request_irq(struct i2c_device *i2c,
                      i2c_irq_hdl       handle,
                      uint32            irq_data,
                      uint32            irq_flag
                      )
```

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
handle	i2c_irq_hdl	中断句柄，中断产生后执行
irq_data	uint32	中断句柄的参数
irq_flag	uint32	申请的中断类型，参考枚举 i2c_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块申请中断成功
RET_ERR	int32	I2C 模块申请中断失败

- **代码示例**

```
void i2c_irq_hdl(uint32 irq, uint32 irq_data, uint32 param) {
    __NOP();
}
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 设置波特率为 300KHz */
i2c_set_baudrate(i2c_test, 300000);
/* 申请 I2C 发送完成中断 */
i2c_request_irq(i2c_test, i2c_irq_hdl, 0, I2C_IRQ_FLAG_TX_DONE);
```

11.4.8. i2c_release_irq()

依据 enum i2c_irq_flag 的中断类型，调用此函数释放 I2C 模块的中断。

- **函数原型**

  i2c_release_irq(struct i2c_device *i2c, uint32 irq_flag)

- **函数参数**

参数	类型	说明
i2c	struct i2c_device	I2C 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	释放的中断类型，参考枚举 i2c_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	I2C 模块中断关闭成功
RET_ERR	int32	I2C 模块中断关闭失败

- **代码示例**

```
struct i2c_device *i2c_test = NULL;
i2c_test = (struct i2c_device*)dev_get(HG_I2C2_DEVID);
/* 通过 dev_get()获取 I2C2 的句柄，并配置主机模式，7 位地址模式，从机设备地址为 0x80 */
i2c_open(i2c_test, IIC_MODE_MASTER, IIC_ADDR_7BIT, 0x80);
/* 设置波特率为 300KHz */
i2c_set_baudrate(i2c_test, 300000);
/* 释放 I2C 发送完成中断 */
i2c_release_irq(i2c_test, I2C_IRQ_FLAG_TX_DONE);
```

12. SPI 接口说明

本章节主要介绍 SPI 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

12.1. Enum

12.1.1. spi_work_mode

spi_mode 描述了工作模式。

枚举量	说明
SPI_MASTER_MODE	SPI 工作在主机模式
SPI_SLAVE_MODE	SPI 工作在从机模式
SPI_SLAVE_FSM_MODE	SPI 工作在从机状态机模式

12.1.2. spi_wire_mode

spi_mode 描述了工作的线模式。

枚举量	说明
SPI_WIRE_SINGLE_MODE	SPI 工作在 1 根 DATA 线模式 @Note: 此时 SPI 包含：CLK; CS; DATA 线。一个 CLK 周期传输 1bit 数据
SPI_WIRE_NORMAL_MODE	SPI 工作在 2 根 DATA 线模式，即标准 SPI 模式 @Note: 此时 SPI 包含：CLK; CS; DATA0(MOSI); DATA1(MISO)线。一个 CLK 周期传输 1bit
SPI_WIRE_DUAL_MODE	SPI 工作在 2 根 DATA 线模式 @Note: 此时 SPI 包含：CLK; CS; DATA0; DATA1 线。一个 CLK 周期传输 2bit

SPI_WIRE_QUAD_MODE	SPI 工作在 4 根 DATA 线模式 @Note: 此时 SPI 包含：CLK; CS; DATA0; DATA1; DATA2; DATA3 线。一个 CLK 周期传输 4bit

12.1.3. spi_clk_mode

spi_clk_mode 描述了工作的时钟模式。

枚举量	说明
SPI_CPOL_0_CPHA_0	SPI 工作在上升沿采样，下降沿发送，空闲时 CLK 为低电平 @Note: SPI_CLK_MODE_0 = SPI_CPOL_0_CPHA_0
SPI_CPOL_0_CPHA_1	SPI 工作在下降沿采样，上升沿发送，空闲时 CLK 为低电平 @Note: SPI_CLK_MODE_1 = SPI_CPOL_0_CPHA_1
SPI_CPOL_1_CPHA_0	SPI 工作在下降沿采样，上升沿发送，空闲时 CLK 为高电平 @Note: SPI_CLK_MODE_2 = SPI_CPOL_1_CPHA_0
SPI_CPOL_1_CPHA_1	SPI 工作在上升沿采样，下降沿发送，空闲时 CLK 为高电平 @Note: SPI_CLK_MODE_3 = SPI_CPOL_1_CPHA_1

12.1.4. spi_ioctl_cmd

spi_ioctl_cmd 描述了 SPI 的配置命令，通过调用 spi_ioctl()进行配置。

枚举量	说明
SPI_WIRE_MODE_SET	SPI 设置工作的线模式
SPI_WIRE_MODE_GET	获取当前 SPI 工作的线模式

SPI_SAMPLE_DELAY	SPI 采样延时，仅对 SPI 主机有效
SPI_SET_FRAME_SIZE	设置 SPI 一帧的数据长度，单位为 bit
SPI_SET_LEN_THRESHOLD	设置 SPI 使用 DMA 发送的数据长度阈值，单位为 byte @Note: 默认的阈值为 16byte

12.1.5. spi_irq_flag

spi_irq_flag 描述了 SPI 可申请的中断类型，通过 spi_request_irq() 和 spi_release_irq()函数使用。

枚举量	说明
SPI_IRQ_FLAG_TX_DONE	SPI 发送数据完成中断
SPI_IRQ_FLAG_RX_DONE	SPI 接收数据完成中断
SPI_IRQ_FLAG_FIFO_OVERFLOW	SPI 的硬件 FIFO 溢出中断，FIFO 的容量为 40bit
SPI_IRQ_FLAG_CS_RISING	检测到 CS 的上升沿中断，仅对 SPI 从机有效
SPI_IRQ_FLAG_SLAVE_FSM_READ_STATUS	SPI 收到读状态指令中断，仅对 SPI 从机状态机有效

12.2. Define

12.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		

参数 2		

12.3. Structure

12.3.1. spi_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-
ioctl	int32 函数指针	-
read	int32 函数指针	-
write	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

12.4. Function

12.4.1. spi_open()

初始化 SPI。在使用 SPI 模块之前，必须先调用此函数。SPI 初始化成功后，其中的默认配置为：数据高位先发；CS 输出低电平有效；一帧的长度为 8bit。若需要改变配置，可以在初始化完成后，使用 spi_ioctl 函数进行更改。

- **函数原型**

```
int32 spi_open( struct spi_device *p_spi,
                uint32             clk_freq,
                uint32             work_mode,
```

```
    uint32              wire_mode,
    uint32              clk_mode
    )
```

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
clk_freq	uint32	SPI 的工作时钟频率配置
work_mode	uint32	SPI 的工作模式，参考枚举 spi_work_mode
wire_mode	uint32	SPI 的工作的线模式，参考枚举 spi_wire_mode
clk_mode	uint32	SPI 的工作的时钟模式，参考枚举 spi_clk_mode

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块初始化成功
RET_ERR	int32	SPI 模块初始化失败

- **代码示例**

```
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
```

12.4.2. spi_close()

关闭SPI。调用此函数后,SPI将无法正常工作,需要重新使用spi_open()函数进行open。

- **函数原型**

int32 spi_close(struct spi_device *p_spi)

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块关闭成功
RET_ERR	int32	SPI 模块关闭失败

- **代码示例**

```
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* 关闭 SPI */
spi_close(spi_test);
```

12.4.3. spi_ioctl()

依据 enum spi_ioctl_cmd 中的命令，调用此函数对 SPI 模块进行相关配置。

- **函数原型**

```
int32 spi_ioctl(struct spi_device *p_spi,
                uint32             cmd,
                uint32             param1,
                uint32             param2
```

)

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
cmd	uint32	SPI 模块的配置命令
param1	uint32	配置参数 1，依据配置命令而定
param2	uint32	配置参数 2，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块配置成功
RET_ERR	int32	SPI 模块配置失败

- **代码示例**

```
uint8 rx_buf[100];
memset((void *)rx_buf, 0, sizeof(rx_buf)/sizeof(rx_buf[0]));
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* 设置 frame size 为 16bit */
spi_ioctl(spi_test, SPI_SET_FRAME_SIZE, 16, 0);
```

12.4.4. spi_read()

SPI 依据接收数据的相关配置，进行接收。

注：

1. 主机模式下，调用此函数后，立即产生 CLK 去接收从机的数据。

2. 从机模式下，调用此函数后，从机等到主机的 CLK，才会接收。

- **函数原型**

  int32 spi_read(struct spi_device *p_spi, void *buf, uint32 size)

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
buf	void*	要接收的数据 BUFFER 的起始地址
size	uint32	数据长度,单位为 byte @Note: 数据长度最大值为：4095

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块接收成功
RET_ERR	int32	SPI 模块接收失败

- **代码示例**

```
uint8 rx_buf[100];
memset((void *)rx_buf, 0, sizeof(rx_buf)/sizeof(rx_buf[0]));
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* SPI 主机，接收 100byte */
spi_read(spi_test, (void *)rx_buf, 100);
```

12.4.5. spi_write()

SPI 依据发送数据的相关配置，进行发送。

注：

1. 主机模式下，调用此函数后，立即产生 CLK 去发送数据。
2. 从机模式下，调用此函数后，从机等到主机的 CLK，才会发送。

- **函数原型**

```
int32 spi_write(struct spi_device *p_spi, const void *buf, uint32 size)
```

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
buf	const void*	要发送的数据 BUFFER 的起始地址
size	uint32	数据长度,单位为 byte @Note: 数据长度最大值为：4095

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块发送成功
RET_ERR	int32	SPI 模块发送失败

- **代码示例**

```
uint8 tx_buf[100];
/* 初始化 tx_buf，无实际意义 */
memset((void *)tx_buf, 0x55, sizeof(tx_buf)/sizeof(tx_buf[0]));
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* SPI 主机，发送 100byte */
spi_write(spi_test, (void *)tx_buf, 100);
```

12.4.6. spi_set_cs()

SPI 设置 CS 的电平值，仅限于主机模式下，片选中 SPI 从机。

- **函数原型**

  int32 spi_set_cs(struct spi_device *p_spi, uint32 cs, uint32 value)

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
cs	uint32	保留，可填任意值 @Note: 可用于 SPI 总线多从机的扩展：指定某一个 SPI 从机被选中
value	uint32	CS 要设置的电平值

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块设置 CS 电平成功
RET_ERR	int32	SPI 模块设置 CS 电平失败

- **代码示例**

```
uint8 tx_buf[100];
/* 初始化 tx_buf，无实际意义 */
memset((void *)tx_buf, 0x55, sizeof(tx_buf)/sizeof(tx_buf[0]));
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* SPI 主机，设置 CS 电平为 0 */
spi_set_cs(spi_test, 0, 0);
/* SPI 主机，发送 100byte */
spi_write(spi_test, (void *)tx_buf, 100);
/* SPI 主机，设置 CS 电平为 1 */
spi_set_cs(spi_test, 0, 1);
```

12.4.7. spi_request_irq()

依据 enum spi_irq_flag 的中断类型，调用此函数申请 SPI 模块的中断。

- **函数原型**

```
int32 spi_request_irq(struct spi_device *p_spi,
                      uint32            irq_flag,
                      spi_irq_hdl       irqhdl,
                      uint32            irq_data
                      )
```

- **函数参数**

参数	类型	说明
p_spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	申请的中断类型，参考枚举 spi_irq_flag
irqhdl	spi_irq_hdl	中断句柄，中断产生后执行
irq_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块申请中断成功
RET_ERR	int32	SPI 模块申请中断失败

- **代码示例**

```
void spi_irq_hdl(uint32 irq, uint32 irq_data){
    __NOP();
}
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* 申请 SPI 发送完成中断 */
spi_request_irq(spi_test, SPI_IRQ_FLAG_TX_DONE, \
spi_irq_hdl, 0);
```

12.4.8. spi_release_irq()

依据 enum spi_irq_flag 的中断类型，调用此函数释放 SPI 模块的中断。

- **函数原型**

```
int32 spi_release_irq(struct spi_device *spi, uint32 irq_flag);
```

- **函数参数**

参数	类型	说明
spi	struct spi_device	SPI 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	释放的中断类型，参考枚举 spi_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	SPI 模块释放中断成功
RET_ERR	int32	SPI 模块释放中断失败

- **代码示例**

```
void spi_irq_hdl(uint32 irq, uint32 irq_data){
    __NOP();
}
struct spi_device *spi_test = NULL;
spi_test = (struct spi_device*)dev_get(HG_SPI0_DEVID);
/* 通过 dev_get()获取 SPI0 的句柄，并配置主机模式，CLK=10MHz，
线模式为标准 SPI 模式，时钟模式为 MODE 0
 */
spi_open(spi_test, 10000000, SPI_MASTER_MODE, \
SPI_WIRE_NORMAL_MODE, SPI_CPOL_0_CPHA_0);
/* 释放 SPI 发送完成中断 */
spi_request_irq(spi_test, SPI_IRQ_FLAG_TX_DONE);
```

13. TIMER 接口说明

本章节主要介绍 TIMER 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

13.1. Enum

13.1.1. timer_type

timer_type 描述了工作模式。

枚举量	说明
TIMER_TYPE_ONCE	TIMER 工作在单次计数模式 @Note: TIMER 计数完成一个周期后，将会停止计数
TIMER_TYPE_PERIODIC	TIMER 工作在循环计数模式 @Note: TIMER 计数完成一个周期后,将会重新从 0 计数

13.1.2. timer_irq_flag

timer_irq_flag 描述了 TIMER 可申请的中断类型。

注：

1. TIMER 的驱动程序，自动开启计数到达周期值中断。
2. 用户使用 timer_request_irq() 和 timer_release_irq()申请和释放中断无效。
3. 用户使用 timer_start()中的 cb 和 cb_data 作为中断回调函数。

枚举量	说明
TIMER_INTR_PERIOD	TIMER 计数到周期值中断

13.1.3. timer_ioctl_cmd

timer_ioctl_cmd 在本芯片中不支持，故不描述。

13.2. Define

13.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

13.3. Structure

13.3.1. timer_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-
start	int32 函数指针	-
stop	int32 函数指针	-
ioctl	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

13.4. Function

13.4.1. timer_device_open()

初始化 TIMER。在使用 TIMER 模块之前，必须先调用此函数。

- **函数原型**

```
int32 timer_device_open(struct timer_device *timer,
                        enum timer_type      mode,
                        uint32               flags
                        )
```

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER 的句柄。通常使用 dev_get()获取句柄
mode	enum timer_type	TIMER 的工作模式选择
flags	uint32	保留，未使用，可填任意值。

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER 模块初始化成功
RET_ERR	int32	TIMER 模块初始化失败

- **代码示例**

```
struct timer_device *timer_test = NULL;
timer_test = (struct timer_device*)dev_get(HG_TIMER0_DEVID);
/* 通过 dev_get()获取 TIMER0 的句柄，并配置单次计数模式
 */
timer_device_open(timer_test, TIMER_TYPE_ONCE, 0);
```

13.4.2. timer_device_close()

关闭TIMER。调用此函数后,TIMER将无法正常工作,需要重新使用timer_device_open()函数进行open。

- **函数原型**

  int32 timer_device_close(struct timer_device *timer)

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER的句柄。通常使用dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER模块关闭成功
RET_ERR	int32	TIMER模块关闭失败

- **代码示例**

```
struct timer_device *timer_test = NULL;
timer_test = (struct timer_device*)dev_get(HG_TIMER0_DEVID);
/* 通过dev_get()获取TIMER0的句柄，并配置单次计数模式
 */
timer_device_open(timer_test, TIMER_TYPE_ONCE, 0);
/* 关闭TIMER模块 */
timer_device_close(timer_test);
```

13.4.3. timer_device_start()

启动TIMER计时。

- **函数原型**

```
int32 timer_device_start( struct timer_device *timer,
                          uint32              tmo_us,
                          timer_cb_hdl        cb,
                          uint32              cb_data
                          )
```

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER 的句柄。通常使用 dev_get()获取句柄
tmo_us	uint32	TIMER 设置的计时时间，单位为 us
cb	timer_cb_hdl	TIMER 计数到周期值中断的回调函数
cb_data	uint32	回调函数的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER 模块启动计时成功
RET_ERR	int32	TIMER 模块启动计时失败

- **代码示例**

```
void timer_cb_hdl(uint32 cb_data, uint32 irq_flag) {
    __NOP();
}
struct timer_device *timer_test = NULL;
timer_test = (struct timer_device*)dev_get(HG_TIMER0_DEVID);
/* 通过 dev_get()获取 TIMER0 的句柄，并配置单次计数模式
 */
timer_device_open(timer_test, TIMER_TYPE_ONCE, 0);
/* 启动 TIMER 计时 1000us，并配置回调函数 */
timer_device_start(timer_test, 1000, timer_cb_hdl, 0);
```

13.4.4. timer_device_stop()

关闭 TIMER 计时。

- **函数原型**

  int32 timer_device_stop(struct timer_device *timer)

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER 模块关闭计时成功
RET_ERR	int32	TIMER 模块关闭计时失败

- **代码示例**

```
void timer_cb_hdl(uint32 cb_data, uint32 irq_flag) {
    __NOP();
}
struct timer_device *timer_test = NULL;
timer_test = (struct timer_device*)dev_get(HG_TIMER0_DEVID);
/* 通过 dev_get()获取 TIMER0 的句柄，并配置单次计数模式
 */
timer_device_open(timer_test, TIMER_TYPE_ONCE, 0);
/* 启动 TIMER 计时 1000us，并配置回调函数 */
timer_device_start(timer_test, 1000, timer_cb_hdl, 0);
/* 关闭 TIMER 计时 */
timer_device_stop(timer_test);
```

13.4.5. timer_device_ioctl()

驱动暂时还未定义可用的 cmd，调用无效。

- **函数原型**

```
int32 timer_device_ioctl( struct timer_device *timer,
                          uint32               cmd,
                          uint32               param1,
                          uint32               param2
                          )
```

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER 的句柄。通常使用 dev_get()获取句柄
cmd	uint32	TIMER 模块的配置命令
param1	uint32	配置参数 1，依据配置命令而定
param2	uint32	配置参数 2，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER 模块配置成功
RET_ERR	int32	TIMER 模块配置失败

- **代码示例**

13.4.6. timer_device_request_irq()

中断相关内容通过 timer_device_start()进行配置，该接口调用无效。

- **函数原型**

```
int32 timer_request_irq(struct timer_device *timer,
                        uint32              irq_flag,
                        timer_cb_hdl        cb,
                        uint32              cb_data
                        )
```

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	申请的中断类型，参考枚举 timer_irq_flag
cb	timer_cb_hdl	中断句柄，中断产生后执行
cb_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER 模块申请中断成功
RET_ERR	int32	TIMER 模块申请中断失败

- **代码示例**

```
```

13.4.7. timer_device_release_irq()

中断相关内容通过 timer_device_start()进行配置，该接口调用无效。

- **函数原型**

```
int32 timer_release_irq(struct timer_device *timer, uint32 irq_flag)
```

- **函数参数**

参数	类型	说明
timer	struct timer_device	TIMER 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	释放的中断类型，参考枚举 timer_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	TIMER 模块释放中断成功
RET_ERR	int32	TIMER 模块释放中断失败

- **代码示例**

14. PWM 接口说明

本章节主要介绍 PWM 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

14.1. Enum

14.1.1. pwm_channel

pwm_channel 描述了 PWM 的产生通道。

枚举量	说明
PWM_CHANNEL_0 ~ 10	PWM 的产生通道 @Note: 用户须在 device.c 文件中，绑定通道到相应的 TIMER
TIMER_TYPE_PERIODIC	TIMER 工作在循环计数模式 @Note: TIMER 计数完成一个周期后，将会重新从 0 计数

14.1.2. pwm_irq_flag

pwm_irq_flag 描述了 PWM 可申请的中断类型。

枚举量	说明
PWM_IRQ_FLAG_COMPARE	PWM 通道绑定的 TIMER 计数到比较值中断
PWM_IRQ_FLAG_PERIOD	PWM 通道绑定的 TIMER 计数到周期值中断

14.1.3. pwm_ioctl_cmd

pwm_ioctl_cmd 描述了 PWM 的配置命令，通过调用 pwm_ioctl()进行配置。

枚举量	说明
PWM_IOCTL_CMD_SET_PERIOD_DUTY	PWM 通道绑定的 TIMER 计数到比较值中断
PWM_IRQ_FLAG_PERIOD	PWM 通道绑定的 TIMER 计数到周期值中断
PWM_IOCTL_CMD_SET_SINGLE_INCREAM	设置为单调递增模式，仅限绑定特殊 TIMER 使用
PWM_IOCTL_CMD_SET_INCREAM_DECREASE	设置为先增后减模式，仅限绑定特殊 TIMER 使用

14.2. Define

14.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

14.3. Structure

14.3.1. pwm_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
init	int32 函数指针	-
deinit	int32 函数指针	-
start	int32 函数指针	-

stop	int32 函数指针	-
ioctl	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

14.4. Function

14.4.1. pwm_init()

初始化 PWM 通道。在使用 PWM 模块通道之前，必须先调用此函数。PWM 通道初始化成功后，其中的默认配置为：PWM 通道绑定的 TIMER 默认使用系统时钟进行计数。

- **函数原型**

```
int32 pwm_init( struct pwm_device *pwm,
                enum pwm_channel   channel,
                uint32             period_us,
                uint32             h_duty_us
                )
```

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel
period_us	uint32	PWM 的周期值，单位为 us @Note: 周期值的倒数，则为 PWM 波形频率
h_duty_us	uint32	PWM 波形中高电平所占的时间 @Note: 其与周期值的比值，即高电平占空比

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道初始化成功
RET_ERR	int32	PWM 通道初始化失败

- **代码示例**

```
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
```

14.4.2. pwm_deinit()

关闭 PWM 通道。调用此函数后，PWM 通道将无法正常工作，需要重新使用 pwm_init()函数进行初始化。

- **函数原型**

  int32 pwm_deinit(struct pwm_device *pwm, enum pwm_channel channel)

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道关闭成功
RET_ERR	int32	PWM 通道关闭失败

- **代码示例**

```
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
/* 关闭 PWM 通道 0 */
pwm_deinit(pwm_test, PWM_CHANNEL_0);
```

14.4.3. pwm_start()

启动 PWM 通道输出 PWM 波形。启动输出后，PWM 的 I/O 引脚，默认输出高电平，持续 pwm_init()配置的高电平时间。

- **函数原型**

  int32 pwm_start(struct pwm_device *pwm, enum pwm_channel channel)

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道启动成功
RET_ERR	int32	PWM 通道启动失败

- **代码示例**

```
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
/* 启动 PWM 通道 0 输出 PWM 波形 */
pwm_start(pwm_test, PWM_CHANNEL_0);
```

14.4.4. pwm_stop()

停止 PWM 通道输出 PWM 波形。

- **函数原型**

  int32 pwm_stop(struct pwm_device *pwm, enum pwm_channel channel)

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道停止成功
RET_ERR	int32	PWM 通道停止失败

- **代码示例**

```
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
/* 启动 PWM 通道 0 输出 PWM 波形 */
```

```
pwm_start(pwm_test, PWM_CHANNEL_0);
/* 停止 PWM 通道 0 输出 PWM 波形 */
pwm_stop(pwm_test, PWM_CHANNEL_0);
```

14.4.5. pwm_ioctl()

依据 enum pwm_ioctl_cmd 中的命令，调用此函数对 PWM 通道进行相关配置。

- **函数原型**

```
int32 pwm_ioctl(struct pwm_device  *pwm,
                enum pwm_channel    channel,
                enum pwm_ioctl_cmd  ioctl_cmd,
                uint32              param1,
                uint32              param2
                )
```

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel
ioctl_cmd	enum pwm_ioctl_cmd	PWM 通道的配置命令，参考枚举 pwm_ioctl_cmd
param1	uint32	配置参数 1，依据配置命令而定
param2	uint32	配置参数 2，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道配置成功
RET_ERR	int32	PWM 通道配置失败

- **代码示例**

```
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
/* 修改周期值和高电平的持续时间 */
pwm_stop(pwm_test, PWM_CHANNEL_0, 100, 10);
/* 启动 PWM 通道 0 输出 PWM 波形 */
pwm_start(pwm_test, PWM_CHANNEL_0);
```

14.4.6. pwm_request_irq()

依据 enum pwm_irq_flag 的中断类型，调用此函数申请 PWM 通道的中断。

- **函数原型**

```
int32 pwm_request_irq(struct pwm_device *pwm,
                      enum pwm_channel   channel,
                      enum pwm_irq_flag  irq_flag,
                      pwm_irq_hdl        irq_hdl,
                      uint32             data
                      )
```

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel
irq_flag	enum pwm_irq_flag	申请的中断类型，参考枚举 pwm_irq_flag
irq_hdl	pwm_irq_hdl	中断句柄，中断产生后执行
data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道申请中断成功
RET_ERR	int32	PWM 通道申请中断失败

- **代码示例**

```
void pwm_irq_hdl(uint32 irq, uint32 irq_data) {
    __NOP();
}
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
/* 申请 PWM 通道 0 绑定的 TIMER 计数到周期值中断 */
pwm_request_irq(pwm_test, PWM_CHANNEL_0, PWM_IRQ_FLAG_COMPARE, \
pwm_irq_hdl, 0);
/* 启动 PWM 通道 0 输出 PWM 波形 */
pwm_start(pwm_test, PWM_CHANNEL_0);
```

14.4.7. pwm_release_irq()

依据 enum pwm_irq_flag 的中断类型，调用此函数释放 PWM 通道的所有中断。

- **函数原型**

  int32 pwm_release_irq(struct pwm_device *pwm, enum pwm_channel channel)

- **函数参数**

参数	类型	说明
pwm	struct pwm_device	PWM 的句柄。通常使用 dev_get()获取句柄
channel	enum pwm_channel	PWM 通道，参考枚举 pwm_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	PWM 通道释放中断成功
RET_ERR	int32	PWM 通道释放中断失败

- **代码示例**

```
struct pwm_device *pwm_test = NULL;
pwm_test = (struct pwm_device*)dev_get(HG_PWM0_DEVID);
/* 通过 dev_get()获取 PWM 的句柄，使用 channel 0，
  配置周期值 100us(10KHz 频率)和高电平时间(90us)
 */
pwm_init(pwm_test, PWM_CHANNEL_0, 100, 90);
/* 申请 PWM 通道 0 绑定的 TIMER 计数到周期值中断 */
pwm_request_irq(pwm_test, PWM_CHANNEL_0, PWM_IRQ_FLAG_COMPARE, \
pwm_irq_hdl, 0);
/* 释放 PWM 通道 0 的所有中断 */
pwm_release_irq(pwm_test, PWM_CHANNEL_0);
/* 启动 PWM 通道 0 输出 PWM 波形 */
pwm_start(pwm_test, PWM_CHANNEL_0);
```

15. CAPTURE 接口说明

本章节主要介绍 CAPTURE 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

15.1. Enum

15.1.1. capture_channel

capture_channel 描述了 CAPTURE 捕获通道。

枚举量	说明
CAPTURE_CHANNEL_0 ~ 3	CAPTURE 的捕获通道 @Note: 用户须在 device.c 文件中,绑定通道到相应的 TIMER

15.1.2. capture_mode

capture_mode 描述了捕获模式。

枚举量	说明
CAPTURE_MODE_RISE	CAPTURE 捕获上升沿
CAPTURE_MODE_FALL	CAPTURE 捕获下降沿
CAPTURE_MODE_ALL	CAPTURE 捕获双边沿

15.1.3. capture_irq_flag

capture_irq_flag 描述了 CAPTURE 可申请的中断类型。

枚举量	说明
CAPTURE_IRQ_FLAG_CAPTURE	CAPTURE 通道捕获信号成功中断

CAPTURE_IRQ_FLAG_OVERFLOW	CAPTURE 通道绑定的 TIMER 计数到周期值中断，即在 TIMER 周期值内未成功捕获信号

15.1.4. capture_ioctl_cmd

capture_ioctl_cmd 暂未定义有用的 cmd，故不描述。

15.2. Define

15.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

15.3. Structure

15.3.1. capture_device

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
init	int32 函数指针	-
deinit	int32 函数指针	-
start	int32 函数指针	-
stop	int32 函数指针	-
ioctl	int32 函数指针	-

request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

15.4. Function

15.4.1. capture_init()

初始化 CAPTURE 通道。在使用 CAPTURE 模块通道之前，必须先调用此函数。CAPTURE 通道初始化成功后，其中的默认配置为：CAPTURE 通道绑定的 TIMER 默认使用系统时钟进行计数；CAPTURE 通道捕获信号成功后，绑定的 TIMER 重新从 0 开始计数。

- **函数原型**

```
int32 capture_init( struct capture_device *capture,
                    enum capture_channel   channel,
                    enum capture_mode      mode
                    )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel
mode	enum capture_mode	CAPTURE 通道捕获模式，参考枚举 capture_mode

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道初始化成功
RET_ERR	int32	CAPTURE 通道初始化失败

- **代码示例**

```
struct capture_device *capture_test = NULL;
capture_test = (struct capture_device*)dev_get(HG_CAPTURE0_DEVID);
/* 通过 dev_get()获取 CAPTURE 的句柄，使用 channel 0，
  配置捕获上升沿
 */
capture_init(capture_test, CAPTURE_CHANNEL_0, CAPTURE_MODE_RISE);
```

15.4.2. capture_deinit()

关闭 CAPTURE 通道。调用此函数后，CAPTURE 通道将无法正常工作，需要重新使用 capture_init()函数进行初始化。

- **函数原型**

```
int32 capture_deinit( struct capture_device *capture,
                      enum capture_channel   channel
                      )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道关闭成功
RET_ERR	int32	CAPTURE 通道关闭失败

- **代码示例**

```
struct capture_device *capture_test = NULL;
capture_test = (struct capture_device*)dev_get(HG_CAPTURE0_DEVID);
/* 通过 dev_get()获取 CAPTURE 的句柄，使用 channel 0，
  配置捕获上升沿
 */
capture_init(capture_test, CAPTURE_CHANNEL_0, CAPTURE_MODE_RISE);
/* 关闭 CAPTURE channel 0 */
capture_deinit(capture_test, CAPTURE_CHANNEL_0);
```

15.4.3. capture_start()

启动 CAPTURE 通道进行捕获。

注：

1. 捕获值需要用户申请 CAPTURE 通道捕获信号成功中断，在回调值中获取。

- **函数原型**

```
int32 capture_start(struct capture_device *capture,
                    enum capture_channel   channel
                    )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道启动成功
RET_ERR	int32	CAPTURE 通道启动失败

- **代码示例**

```
struct capture_device *capture_test = NULL;
capture_test = (struct capture_device*)dev_get(HG_CAPTURE0_DEVID);
/* 通过 dev_get()获取 CAPTURE 的句柄，使用 channel 0，
  配置捕获上升沿
 */
capture_init(capture_test, CAPTURE_CHANNEL_0, CAPTURE_MODE_RISE);
/* 启动 CAPTURE channel 0 进行捕获 */
capture_start(capture_test, CAPTURE_CHANNEL_0);
```

15.4.4. capture_stop()

停止 CAPTURE 通道进行捕获。

- **函数原型**

```
int32 capture_stop( struct capture_device *capture,
                    enum capture_channel   channel
                    )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道停止成功
RET_ERR	int32	CAPTURE 通道停止失败

- **代码示例**

```
struct capture_device *capture_test = NULL;
capture_test = (struct capture_device*)dev_get(HG_CAPTURE0_DEVID);
/* 通过 dev_get()获取 CAPTURE 的句柄，使用 channel 0，
  配置捕获上升沿
 */
capture_init(capture_test, CAPTURE_CHANNEL_0, CAPTURE_MODE_RISE);
/* 启动 CAPTURE channel 0 进行捕获 */
capture_start(capture_test, CAPTURE_CHANNEL_0);
/* 停止 CAPTURE channel 0 进行捕获 */
capture_stop(capture_test, CAPTURE_CHANNEL_0);
```

15.4.5. capture_ioctl()

依据 enum capture_ioctl_cmd 中的命令，调用此函数对 CAPTURE 通道进行相关配置。

注：

1. 暂时未设置相应的 ioctl cmd，故调用此函数无效。

- **函数原型**

```
int32 capture_ioctl(struct capture_device *capture,
                    enum capture_channel   channel,
                    enum capture_ioctl_cmd cmd,
                    uint32                 param1,
                    uint32                 param2
                    )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel
cmd	enum capture_ioctl_cmd	CAPTURE 通道的配置命令，参考枚举 capture_ioctl_cmd
param1	uint32	配置参数 1，依据配置命令而定

param2	uint32	配置参数 2，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道配置成功
RET_ERR	int32	CAPTURE 通道配置失败

- **代码示例**

```
```

15.4.6. capture_request_irq()

依据 enum capture_irq_flag 的中断类型，调用此函数申请 CAPTURE 通道的中断。

- **函数原型**

```
int32 capture_request_irq(struct capture_device *capture,
                          enum capture_channel   channel,
                          enum capture_irq_flag  irq_flag,
                          capture_irq_hdl        irq_hdl,
                          uint32                 data
                          )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel

irq_flag	enum capture_irq_flag	申请的中断类型，参考枚举 capture_irq_flag
irq_hdl	capture_irq_hdl	中断句柄，中断产生后执行
data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道申请中断成功
RET_ERR	int32	CAPTURE 通道申请中断失败

- **代码示例**

```
void capture_irq_hdl(uint32 irq, uint32 irq_data) {
    __NOP();
}
struct capture_device *capture_test = NULL;
capture_test = (struct capture_device*)dev_get(HG_CAPTURE0_DEVID);
/* 通过 dev_get()获取 CAPTURE 的句柄，使用 channel 0，
  配置捕获上升沿
 */
capture_init(capture_test, CAPTURE_CHANNEL_0, CAPTURE_MODE_RISE);
/* 申请 CAPTURE channel 0 捕获成功中断 */
capture_request_irq(capture_test, CAPTURE_CHANNEL_0,\
CAPTURE_IRQ_FLAG_CAPTURE, capture_irq_hdl, 0);
/* 启动 CAPTURE channel 0 进行捕获 */
capture_start(capture_test, CAPTURE_CHANNEL_0);
```

15.4.7. capture_release_irq()

依据 enum capture_irq_flag 的中断类型，调用此函数释放 CAPTURE 通道的所有中断。

- **函数原型**

```
int32 capture_release_irq(struct capture_device *capture,
                          enum capture_channel   channel
                          )
```

- **函数参数**

参数	类型	说明
capture	struct capture_device	CAPTURE 的句柄。通常使用 dev_get()获取句柄
channel	enum capture_channel	CAPTURE 通道，参考枚举 capture_channel

- **返回值**

返回值	类型	说明
RET_OK	int32	CAPTURE 通道释放中断成功
RET_ERR	int32	CAPTURE 通道释放中断失败

- **代码示例**

```
void capture_irq_hdl(uint32 irq, uint32 irq_data) {
    __NOP();
}
struct capture_device *capture_test = NULL;
capture_test = (struct capture_device*)dev_get(HG_CAPTURE0_DEVID);
/* 通过 dev_get()获取 CAPTURE 的句柄，使用 channel 0，
  配置捕获上升沿
 */
capture_init(capture_test, CAPTURE_CHANNEL_0, CAPTURE_MODE_RISE);
/* 申请 CAPTURE channel 0 捕获成功中断 */
capture_request_irq(capture_test, CAPTURE_CHANNEL_0,\
CAPTURE_IRQ_FLAG_CAPTURE, capture_irq_hdl, 0);
/* 释放 CAPTURE channel 0 的所有中断 */
capture_release_irq(capture_test, CAPTURE_CHANNEL_0);
/* 启动 CAPTURE channel 0 进行捕获 */
capture_start(capture_test, CAPTURE_CHANNEL_0);
```

16. SHA 接口说明

本章节主要介绍 sha 的函数、枚举、宏、结构体的作用。它们位于“sdk\include\hal”文件目录下。

16.1. Enum

16.1.1. sha_calc_flags

enum sha_calc_flags 提供了 SHA 的工作方式。

枚举量	说明
SHA_CALC_LAST_DATA	指明最后一次计算，这时 SHA 才会产生输出数据，使用时与工作方式相或： SHA_CALC_LAST_DATA | SHA_CALC_SHA256
SHA_CALC_SHA256	使用 SHA256 运算类型
SHA_CALC_SHA384	使用 SHA384 运算类型，尚未支持
SHA_CALC_SHA512	使用 SHA512 运算类型，尚未支持

16.2. Define

16.2.1. MACRO

16.3. Structure

16.3.1. sha_dev

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
dev	struct dev_obj	描述了操作系统相关内容
calc	int32 函数指针	
read	int32 函数指针	

16.4. Function

16.4.1. sha_calc()

进行 SHA 运算，通常可以通过多次不带有 SHA_CALC_LAST_DATA 的 sha_calc()调用，再加上最后一次带有 SHA_CALC_LAST_DATA 的 sha_calc 调用来进行分包运算。也可以在第一次时，就调用带有 SHA_CALC_LAST_DATA 的 sha_calc(),这会直接完成一次 sha 运算。

需要注意的是，分包运算时除了最后一包，其余包的数据长度需要为 64 的整数倍。

当输出产生后，需要用 sha_read()拿取输出后才可以继续使用 sha_calc()。

- **函数原型**

```
int32 sha_calc(struct sha_dev *dev, uint8 input[], uint32 len,
                    enum sha_calc_flags flags)
```

- **函数参数**

参数	类型	说明
dev	sha_dev *	SHA 的句柄。通常使用 dev_get()获取句柄
input[]	uint8[]	输入数据 buf
len	uint32	输入数据长度
flags	enum sha_calc_flags	工作方式

- **返回值**

返回值	类型	说明
RET_OK	int32	成功
RET_ERR	int32	失败

16.4.2. sha_read()

拿取 sha 的输出结果

- **函数原型**

  int32 sha_read(struct sha_dev *dev, uint8 output[], uint32 timeout);

- **函数参数**

参数	类型	说明
dev	struct sha_dev	SHA 的句柄。通常使用 dev_get()获取句柄
Output[]	uint8[]	输出结果要存放的位置
timeout	uint32	超时时间

- **返回值**

返回值	类型	说明
RET_OK	int32	成功
RET_ERR	int32	失败

- **代码示例**

```
struct sha_dev *shadev= (struct sha_dev *)dev_get(HG_SHA_DEVID);
extern uint8 idat[];
extern uint32 ilen;
uint8 odat[32];
sha_calc(shadev, idat, ilen,SHA_CALC_SHA256|SHA_CALC_LAST_DATA);
sha_read(shadev, odat, 500);
```

17. AUADC 接口说明

本章节主要介绍audio adc 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

17.1. Enum

17.1.1. auadc_sample_rate

auadc_sample_rate 描述了音频采样率。

枚举量	说明
AUADC_SAMPLE_RATE_8K	AUDIO ADC 采样率为 8K
AUADC_SAMPLE_RATE_16K	AUDIO ADC 采样率为 16K
AUADC_SAMPLE_RATE_44_1K	AUDIO ADC 采样率为 44.1K
AUADC_SAMPLE_RATE_48K	AUDIO ADC 采样率为 48K

17.1.2. auadc_irq_flag

auadc_irq_flag 描述了 AUDIO ADC 可供申请的中断类型。

枚举量	说明
AUADC_IRQ_FLAG_HALF	AUDIO ADC 半满中断
AUADC_IRQ_FLAG_FULL	AUDIO ADC 全满中断

17.1.3. auadc_ioctl_cmd

auadc_ioctl_cmd 描述了 AUDIO ADC 的配置命令，通过调用 auadc_ioctl()进行配置。

枚举量	说明
AUADC_IOCTL_CMD_SET_SAMPLE_RATE	设置 AUDIO ADC 的采样率

AUADC_IOCTL_CMD_SET_SOUND_CHANNEL	设置 AUDIO ADC 的声道
AUADC_IOCTL_CMD_SET_DIGITAL_GAIN	设置 AUDIO ADC 的数字增益
AUADC_IOCTL_CMD_SET_ANALOG_GAIN	设置 AUDIO ADC 的模拟增益

17.2. Define

17.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

17.3. Structure

17.3.1. auadc_hal_ops

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
ops	struct devobj_ops	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-
ioctl	int32 函数指针	-
read	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

17.4. Function

17.4.1. auadc_open()

初始化 AUDIO ADC。在使用 AUDIO ADC 模块之前，必须先调用此函数。AUDIO ADC 初始化成功后，其中的默认配置为：数据位宽为 16bit。若需要改变配置，可以在初始化完成后，使用 auadc_ioctl 函数进行更改。

- **函数原型**

```
int32 auadc_open(struct auadc_device *auadc,
                 enum auadc_sample_rate sample_rate
                 )
```

- **函数参数**

参数	类型	说明
auadc	struct auadc_device	auadc 的句柄。通常使用 dev_get()获取句柄
sample_rate	enum auadc_sample_rate	AUDIO ADC 的采样率，参考枚举 auadc_sample_rate

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDIO ADC 模块初始化成功
RET_ERR	int32	AUDIO ADC 模块初始化失败

- **代码示例**

```
struct auadc_device *auadc_test = NULL;
auadc_test = (struct auadc_device*)dev_get(HG_AUADC_DEVID);
/* 通过 dev_get()获取 AUADC 的句柄，采样率 44.1k */
auadc_open(auadc_test, AUADC_SAMPLE_RATE_44_1K);
```

17.4.2. auadc_close()

关闭 AUDIO ADC。调用此函数后，AUDIO ADC 将无法正常工作，所有的配置（包含中断相关配置）都会失效。

- **函数原型**

  int32 auadc_close(struct auadc_device *auadc)

- **函数参数**

参数	类型	说明
auadc	struct auadc_device	AUADC 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	AUADC 模块关闭成功
RET_ERR	int32	AUADC 模块关闭失败

- **代码示例**

```
struct auadc_device *auadc_test = NULL;
auadc_test = (struct auadc_device*)dev_get(HG_AUADC_DEVID);
/* 通过 dev_get()获取 AUADC 的句柄，关闭 AUADC*/
auadc_close(auadc_test);
```

17.4.3. auadc_read()

AUDIO ADC 依据 DATA BUFFER 的地址和数据个数，接收数据。用户可在 AUDIO ADC 模块的完成一半传输中断时，切换下一次要接收的 DATA BUFFER 的地址和数据个数。AUDIO ADC 模块会在本次传输完成后，自动切换成下一次的 DATA BUFFER 和数据个数。若用户在 AUDIO ADC 模块的完成一半传输中断时，未进行切换地址和长度。AUDIO ADC 模块会在本次传输完成后，重新载入本次的 DATA BUFFER 和数据个数。

注意：用户只能在 AUDIO ADC 模块的完成一半传输中断时切换下一次的地址和数据，否则会导致发送异常。

- **函数原型**

  int32 auadc_read(struct auadc_device *auadc, void* buf, uint32 bytes)

- **函数参数**

参数	类型	说明
auadc	struct auadc_device	AUADC 的句柄。通常使用 dev_get()获取句柄
buf	void*	DATA BUFFER 的起始地址
bytes	uint32	要接收的数据个数，单位为 byte，要求 4byte 对齐

- **返回值**

返回值	类型	说明
RET_OK	int32	AUADC 模块写入 BUFFER 地址和长度成功
RET_ERR	int32	AUADC 模块写入 BUFFER 地址和长度失败

- **代码示例**

```
uint16 data[256];
struct auadc_device *auadc_test = NULL;
/* 随意初始化数据，只为举例理解，无实际意义 */
memset((void *)data, 0x00, sizeof(data)/sizeof(data[0]));
auadc_test = (struct auadc_device*)dev_get(HG_AUADC_DEVID);
/* 通过 dev_get()获取 AUADC 的句柄，并配置采样率 44.1kHz */
auadc_open(auadc_test, AUADC_SAMPLE_RATE_44_1K);
/* 接收 256byte */
auadc_read(auadc_test, (void *)data, sizeof(data)/sizeof(data[0]));
```

17.4.4. auadc_ioctl()

依据 enum auadc_ioctl_cmd 中的命令，调用此函数对 AUDIO ADC 模块进行相关配置。

- **函数原型**

```
int32 auadc_ioctl(struct auadc_device *auadc,
                  enum auadc_ioctl_cmd ioctl_cmd,
                  uint32 param1,
                  uint32 param2)
```

- **函数参数**

参数	类型	说明
auadc	struct auadc_device	AUADC 的句柄。通常使用 dev_get()获取句柄
cmd	uint32	AUADC 模块的配置命令，见枚举 auadc_ioctl_cmd
param1	uint32	配置参数，依据配置命令而定
param2	uint32	配置参数，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	AUADC 模块配置成功
RET_ERR	int32	AUADC 模块配置失败

- **代码示例**

```
struct auadc_device *auadc_test = NULL;
auadc_test = (struct auadc_device*)dev_get(HG_AUADC_DEVID);
/* 通过 dev_get()获取 AUADC 的句柄，并配置采样率 44.1kHz */
auadc_open(auadc_test, auadc_test, AUADC_SAMPLE_RATE_44_1K);
/* 通过 auadc_ioctl，配置 AUADC 的数字增益 */
auadc_ioctl(auadc_test,AUADC_IOCTL_CMD_SET_DIGITAL_GAIN, 50, 0);
```

17.4.5. auadc_request_irq()

依据 enum auadc_irq_flag 的中断类型，调用此函数申请 AUDIO ADC 模块的中断。

- **函数原型**

```
int32 auadc_request_irq(struct auadc_device *auadc,
                        enum auadc_irq_flag irq_flag,
                        auadc_irq_hdl irq_hdl,
                        uint32 irq_data
                        )
```

- **函数参数**

参数	类型	说明
auadc	struct auadc_device	AUADC 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	申请的中断类型，参考枚举 auadc_irq_flag
irq_hdl	auadc_irq_hdl	中断句柄，中断产生后执行
irq_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	AUADC 模块申请中断成功
RET_ERR	int32	AUADC 模块申请中断失败

- **代码示例**

```
void auadc_interrupt_func(uint32 irq, uint32 irq_data) {
  __NOP();
}
struct auadc_device *auadc_test = NULL;
auadc_test = (struct auadc_device*)dev_get(HG_AUADC_DEVID);
/* 通过 dev_get()获取 AUADC 的句柄，并配置采样率 44.1kHz */
auadc_open(auadc_test, AUADC_SAMPLE_RATE_44_1K);
/*AUADC 申请完成一半传输中断，中断句柄为 auadc_interrupt_func，参数为 0*/
auadc_request_irq(auadc_test, AUADC_IRQ_FLAG_HALF,\
(auadc_irq_hdl)auadc_interrupt_func, 0);
```

17.4.6. auadc_release_irq()

依据 enum auadc_irq_flag 的中断类型，调用此函数释放 AUDIO ADC 模块的中断。

- **函数原型**

```
int32 auadc_release_irq(struct auadc_device *auadc,
                        enum auadc_irq_flag irq_flag
                        )
```

- **函数参数**

参数	类型	说明
auadc	struct auadc_device	AUADC 的句柄。通常使用 dev_get()获取句柄
irq_flag	enum auadc_irq_flag	释放的中断类型，参考枚举 auadc_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	AUADC 模块中断关闭成功
RET_ERR	int32	AUADC 模块中断关闭失败

- **代码示例**

```
struct auadc_device *auadc_test = NULL;
auadc_test = (struct auadc_device*)dev_get(HG_AUADC_DEVID);
/* 通过 dev_get()获取 AUADC 的句柄，并配置采样率 44.1kHz */
auadc_open(auadc_test, AUADC_SAMPLE_RATE_44_1K);
/* 关闭完成一半传输中断 */
auadc_release_irq(auadc_test, AUADC_IRQ_FLAG_HALF);
```

18. AUDAC 接口说明

本章节主要介绍audio dac 的函数、枚举、宏、结构体的作用。它们位于“SDK\include\hal”文件目录下。

18.1. Enum

18.1.1. audac_sample_rate

audac_sample_rate 描述了音频采样率。

枚举量	说明
AUDAC_SAMPLE_RATE_8K	AUDIO DAC 采样率为 8K
AUDAC_SAMPLE_RATE_16K	AUDIO DAC 采样率为 16K
AUDAC_SAMPLE_RATE_44_1K	AUDIO DAC 采样率为 44.1K
AUDAC_SAMPLE_RATE_48K	AUDIO DAC 采样率为 48K

18.1.2. audac_irq_flag

audac_irq_flag 描述了 AUDIO DAC 可供申请的中断类型。

枚举量	说明
AUDAC_IRQ_FLAG_HALF	AUDIO DAC 半满中断
AUDAC_IRQ_FLAG_FULL	AUDIO DAC 全满中断

18.1.3. audac_ioctl_cmd

audac_ioctl_cmd 描述了 AUDIO DAC 的配置命令，通过调用 audac_ioctl()进行配置。

枚举量	说明
AUDAC_IOCTL_CMD_SET_SAMPLE_RATE	设置 AUDIO DAC 的采样率

AUDAC_IOCTL_CMD_SET_SOUND_CHANNEL	设置 AUDIO DAC 的声道
AUDAC_IOCTL_CMD_SET_DIGITAL_GAIN	设置 AUDIO DAC 的数字增益
AUDAC_IOCTL_CMD_SET_ANALOG_GAIN	设置 AUDIO DAC 的模拟增益

18.2. Define

18.2.1. MACRO

描述宏的功能，如果宏需要传参数，则在下面表格说明；如果没有要传的参数，则在下面表格填无。（注意：宏的名字采用字母大写）

参数	类型	说明
参数 1		
参数 2		

18.3. Structure

18.3.1. audac_hal_ops

该结构体描述了函数指针以及操作系统的相关内容，不建议用户修改。

数据元素	类型	说明
ops	struct devobj_ops	描述了操作系统相关内容
open	int32 函数指针	-
close	int32 函数指针	-
ioctl	int32 函数指针	-
write	int32 函数指针	-
request_irq	int32 函数指针	-
release_irq	int32 函数指针	-

18.4. Function

18.4.1. audac_open()

初始化 AUDIO DAC。在使用 AUDIO DAC 模块之前，必须先调用此函数。AUDIO DAC 初始化成功后，其中的默认配置为：数据位宽为 16bit。若需要改变配置，可以在初始化完成后，使用 audac_ioctl 函数进行更改。

- **函数原型**

```
int32 audac_open(struct audac_device *audac,
                 enum audac_sample_rate sample_rate
                 )
```

- **函数参数**

参数	类型	说明
audac	struct audac_device	audac 的句柄。通常使用 dev_get() 获取句柄
sample_rate	enum audac_sample_rate	AUDIO DAC 的采样率，参考枚举 audac_sample_rate

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDIO DAC 模块初始化成功
RET_ERR	int32	AUDIO DAC 模块初始化失败

- **代码示例**

```
struct audac_device *audac_test = NULL;
audac_test = (struct audac_device*)dev_get(HG_AUDAC_DEVID);
/* 通过 dev_get()获取 AUDAC 的句柄，采样率 44.1k */
audac_open(audac_test, AUDAC_SAMPLE_RATE_44_1K);
```

18.4.2. audac_close()

关闭 AUDIO DAC。调用此函数后，AUDIO DAC 将无法正常工作，所有的配置（包含中断相关配置）都会失效。

- **函数原型**

  int32 audac_close(struct audac_device *audac)

- **函数参数**

参数	类型	说明
audac	struct audac_device	AUDAC 的句柄。通常使用 dev_get()获取句柄

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDAC 模块关闭成功
RET_ERR	int32	AUDAC 模块关闭失败

- **代码示例**

```
struct audac_device *audac_test = NULL;
audac_test = (struct audac_device*)dev_get(HG_AUDAC_DEVID);
/* 通过 dev_get()获取 AUDAC 的句柄，关闭 AUDAC*/
audac_close(audac_test);
```

18.4.3. audac_write()

AUDIO DAC 依据 DATA BUFFER 的地址和数据个数，接收数据。用户可在 AUDIO DAC 模块的完成一半传输中断时，切换下一次要接收的 DATA BUFFER 的地址和数据个数。AUDIO DAC 模块会在本次传输完成后，自动切换成下一次的 DATA BUFFER 和数据个数。若用户在 AUDIO DAC 模块的完成一半传输中断时，未进行切换地址和长度。AUDIO DAC 模块会在本次传输完成后，重新载入本次的 DATA BUFFER 和数据个数。

注意：用户只能在 AUDIO DAC 模块的完成一半传输中断时切换下一次的地址和数据，否则会导致发送异常。

- **函数原型**

  int32 audac_write(struct audac_device *audac, void* buf, uint32 bytes)

- **函数参数**

参数	类型	说明
audac	struct audac_device	AUDAC 的句柄。通常使用 dev_get()获取句柄
buf	void*	DATA BUFFER 的起始地址
bytes	uint32	要接收的数据个数，单位为 byte，要求 4byte 对齐

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDAC 模块写入 BUFFER 地址和长度成功
RET_ERR	int32	AUDAC 模块写入 BUFFER 地址和长度失败

- **代码示例**

```
uint16 data[256];
struct audac_device *audac_test = NULL;
/* 随意初始化数据，只为举例理解，无实际意义 */
memset((void *)data, 0x00, sizeof(data)/sizeof(data[0]));
audac_test = (struct audac_device*)dev_get(HG_AUDAC_DEVID);
/* 通过 dev_get()获取 AUDAC 的句柄，并配置采样率 44.1kHz */
audac_open(audac_test, AUDAC_SAMPLE_RATE_44_1K);
/* 接收 256byte */
audac_write(audac_test, (void *)data, sizeof(data)/sizeof(data[0]));
```

18.4.4. audac_ioctl()

依据 enum audac_ioctl_cmd 中的命令，调用此函数对 AUDIO DAC 模块进行相关配置。

- **函数原型**

```
int32 audac_ioctl(struct audac_device *audac,
                  enum audac_ioctl_cmd ioctl_cmd,
                  uint32 param1,
                  uint32 param2)
```

- **函数参数**

参数	类型	说明
auadc	struct audac_device	AUDAC 的句柄。通常使用 dev_get()获取句柄
cmd	uint32	AUDAC 模块的配置命令，见枚举 audac_ioctl_cmd
param1	uint32	配置参数，依据配置命令而定
param2	uint32	配置参数，依据配置命令而定

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDAC 模块配置成功
RET_ERR	int32	AUDAC 模块配置失败

- **代码示例**

```
struct audac_device *audac_test = NULL;
audac_test = (struct audac_device*)dev_get(HG_AUDAC_DEVID);
/* 通过 dev_get()获取 AUDAC 的句柄，并配置采样率 44.1kHz */
audac_open(audac_test, audac_test, AUDAC_SAMPLE_RATE_44_1K);
/* 通过 auadc_ioctl，配置 AUDAC 的数字增益 */
audac_ioctl(audac_test,AUDAC_IOCTL_CMD_SET_DIGITAL_GAIN, 50, 0);
```

18.4.5. audac_request_irq()

依据 enum audac_irq_flag 的中断类型，调用此函数申请 AUDIO DAC 模块的中断。

- **函数原型**

```
int32 audac_request_irq(struct audac_device *audac,
                        enum audac_irq_flag irq_flag,
                        audac_irq_hdl irq_hdl,
                        uint32 irq_data
                        )
```

- **函数参数**

参数	类型	说明
audac	struct audac_device	AUDAC 的句柄。通常使用 dev_get()获取句柄
irq_flag	uint32	申请的中断类型，参考枚举 audac_irq_flag
irq_hdl	audac_irq_hdl	中断句柄，中断产生后执行
irq_data	uint32	中断句柄的参数

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDAC 模块申请中断成功
RET_ERR	int32	AUDAC 模块申请中断失败

- **代码示例**

```
void audac_interrupt_func(uint32 irq, uint32 irq_data) {
  __NOP();
}
struct audac_device *audac_test = NULL;
audac_test = (struct audac_device*)dev_get(HG_AUDAC_DEVID);
/* 通过 dev_get()获取 AUDAC 的句柄，并配置采样率 44.1kHz */
audac_open(audac_test, AUDAC_SAMPLE_RATE_44_1K);
/* AUDAC 申请完成一半传输中断，中断句柄为 audac_interrupt_func，参数为 0 */
audac_request_irq(audac_test, AUDAC_IRQ_FLAG_HALF,\
```

```
(audac_irq_hdl)audac_interrupt_func, 0);
```

18.4.6. audac_release_irq()

依据 enum audac_irq_flag 的中断类型，调用此函数释放 AUDIO DAC 模块的中断。

- **函数原型**

```
int32 audac_release_irq(struct audac_device *audac,
                        enum audac_irq_flag irq_flag
                        )
```

- **函数参数**

参数	类型	说明
audac	struct audac_device	AUDAC 的句柄。通常使用 dev_get()获取句柄
irq_flag	enum audac_irq_flag	释放的中断类型，参考枚举 audac_irq_flag

- **返回值**

返回值	类型	说明
RET_OK	int32	AUDAC 模块中断关闭成功
RET_ERR	int32	AUDAC 模块中断关闭失败

- **代码示例**

```
struct audac_device *audac_test = NULL;
audac_test = (struct audac_device*)dev_get(HG_AUDAC_DEVID);
/* 通过 dev_get()获取 AUDAC 的句柄，并配置采样率 44.1kHz */
audac_open(audac_test, AUDAC_SAMPLE_RATE_44_1K);
/* 关闭完成一半传输中断 */
audac_release_irq(audac_test, AUDAC_IRQ_FLAG_HALF);
```