
# Ingenic SDK 使用说明

文档历史：

| 版本 | 作者 | 注释 |
| --- | --- | --- |
| 1.0 | | |

目录

# 1 GPIO

该套 GPIO 的接口实现基于 libgpio，GPIO 中断回调基于 linux 线程调度器实现（最高优先级）。

## 1.1 源码文件

头 文 件: sdk/include/gpio/gpio_manager.h

源 文 件:sdk/gpio/gpio_manager.c

测试程序: sdk/examples/gpio/

## 1.2 注意事项

请注意该接口调用非线程安全，请避免多个线程同时调用一个 API 接口。

## 1.3 get_gpio_manager

函数原型: struct gpio_manager *get_gpio_manager(void);

函数功能: 获取 gpio_manager 操作指针, 以操作 gpio_manager 内部方法

返 回 值: 返回 gpio_manager 结构体指针

其　他: 通过该结构体指针访问 gpio_manager 内部提供的方法

## 1.4 gpio_init

函数原型: int32_t (*init)(void);

函数功能: GPIO 库资源初始化

返 回 值:  0:成功;  -1: 失败;

其　他: 使用 gpio_manager 内部方法必须先调用此函数先初始化资源

## 1.5 gpio_deinit

函数原型: void (*deinit)(void);

函数功能: GPIO 库资源释放

返 回 值: 无

其　他:　与 gpio_init 相对应，会释放所有 GPIO 资源包括 GPIO 中断。

请确认无需使用 GPIO 后才调用，释放资源后之前操作的 GPIO 状态会恢复默认状态（上电时状态）

## 1.6 gpio_open

函数原型: int32_t (*open)(uint32_t gpio);

函数功能: 打开某个 GPIO 功能

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

返 回 值:  0: 成功;  -1: 失败

其　他:　操作某个 GPIO 功能之前必须先打开 GPIO

## 1.7 gpio_close

函数原型: void (*close)(uint32_t gpio);

函数功能: 关闭某个 GPIO 功能

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

返 回 值: 无

其　他:　关闭 GPIO 后中断也会关闭，GPIO 恢复默认状态（上电时状态）

## 1.8 gpio_get_direction

函数原型: int32_t (*get_direction)(uint32_t gpio, gpio_direction *dir);

函数功能: 获取 GPIO 的输入输出模式

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

dir: 获取功能状态 输入或输出

参数：GPIO_IN or GPIO_OUT

注意：dir 参数是 gpio_direction 指针

返 回 值: 0:成功; -1: 失败;

## 1.9 gpio_set_direction

函数原型: int32_t (*set_direction)(uint32_t gpio, gpio_direction dir);

函数功能: 设置 GPIO 的输入输出模式

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

dir: 设置功能状态 输入或输出

参数：GPIO_IN or GPIO_OUT

返 回 值: 0:成功; -1: 失败;

## 1.10 gpio_get_value

函数原型: int32_t (*get_value)(uint32_t gpio, gpio_value *value);

函数功能: 获取 GPIO 的电平状态

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

value: 获取电平状态 低电平或高电平

参数：GPIO_LOW or GPIO_HIGH

注意：value 参数是 gpio_vlaue 指针

返 回 值: 0:成功; -1: 失败;

## 1.11 gpio_set_value

函数原型: int32_t (*set_value)(uint32_t gpio, gpio_value value);

函数功能: 设置 GPIO 的电平状态

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

value: 设置电平状态 低电平或高电平

参数：GPIO_LOW or GPIO_HIGH

注意：输入模式下禁止设置电平状态

返 回 值: 0:成功; -1: 失败;

## 1.12 gpio_set_irq_func

函数原型: void (*set_irq_func)(gpio_irq_func func);

函数功能: 设置 GPIO 中断回调函数

函数参数:

func: GPIO 中断回调函数

typedef void (*irq_work_func)(int);

无返回值和整型参数（GPIO 的编号）的函数

注意：所有 GPIO 对应一个中断函数，回调函数参数为触发中断的 GPIO 编号

返 回 值: 无

## 1.13 gpio_enable_irq

函数原型: uint32_t (*enable_irq)(uint32_t gpio, gpio_irq_mode mode);

函数功能: 使能某个 GPIO 中断

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

mode: 设置中断触发方式

参数：GPIO_RISING, 上升沿触发

GPIO_FALLING, 下降沿触发

GPIO_BOTH, 双边沿沿触发

注意：使能前必须设置中断回调函数 set_irq_func，GPIO 引脚必须为输入模式

返 回 值: 0:成功; -1: 失败;

## 1.14 gpio_disable_irq

函数原型: void (*disable_irq)(uint32_t gpio);

函数功能: 关闭某个 GPIO 中断

函数参数:

gpio: 需要操作的 GPIO 编号

例如: GPIO_PA(n) GPIO_PB(n) GPIO_PC(n) GPIO_PD(n)

返 回 值: 无

# 2 timer

该套定时器的接口实现基于 linux timerfd 系统调用。timerfd 的定时精度在微秒级别，由于本定时器基于 linux 线程调度器实现，线程切换精度在几十个微秒，导致定时器的误差在 1 毫秒内，下述实现的定时器封装接口最小精度都限定在 1 毫秒。

## 2.1 源码文件

头 文 件: sdk/include/timer/timer_manager.h

源 文 件:sdk/timer/timer_manager.c

测试程序: sdk/examples/timer/

## 2.2 配置参数

TIMER_DEFAULT_MAX_CNT: 表示系统支持的最大定时器个数，默认设置为 5

## 2.3 注意事项

请注意该接口调用非线程安全，请避免多个线程同时调用一个 API 接口。

## 2.4 get_timer_manager

函数原型: struct timer_manager *get_timer_manager(void);

函数功能: 获取 timer_manager 操作指针, 以操作 timer_manager 内部方法

返 回 值: 返回 timer_manager 结构体指针

其　他: 通过该结构体指针访问 timer_manager 内部提供的方法

## 2.5 timer_init

函数原型: int32_t (*init)(int32_t id, uint32_t interval, uint8_t is_one_time,

func_handle routine, void *arg);

函数功能: 定时器初始化

函数参数:

id: 指定分配的 id 号,可选配置有以下两类

id=-1: 自动分配定时器 id

id>=1: 固定分配 id，范围[1,TIMER_DEFAULT_MAX_CNT]

interval: 定时周期,单位:ms

is_one_time: 是否是一次定时, 大于 0 为一次定时,否则周期定时

routine: 定时器处理函数

arg: 定时器处理函数参数

注意: arg 为指针,sdk 中只是传递指针, 指针指向的内容请用户注意保护

返 回 值: >=1:返回成功分配的 id 号; -1: 失败

其　他: 支持的最大定时器数目由宏定义 TIMER_DEFAULT_MAX_CNT 决定

## 2.6 timer_deinit

函数原型: int32_t (*deinit)(uint32_t id);

函数功能: 定时器释放

函数参数:

id: 定时器 id 号，可配置范围[1,TIMER_DEFAULT_MAX_CNT]

返 回 值: 0: 成功; -1: 失败

其　他: 与 timer_init 相对应

## 2.7 timer_start

函数原型: int32_t (*start)(uint32_t id);

函数功能: 定时器开启, 调用成功后定时器执行定时计数

函数参数:

id: 定时器 id 号，可配置范围[1,TIMER_DEFAULT_MAX_CNT]

返 回 值: 0: 成功; -1: 失败

其　他: 与 timer_init 相对应

## 2.8 timer_stop

函数原型: int32_t (*stop)(uint32_t id);

函数功能: 定时器停止, 调用成功后定时器停止定时计数

函数参数:

id: 定时器 id 号，可配置范围[1,TIMER_DEFAULT_MAX_CNT]

返 回 值: 0: 成功; -1: 失败

其　他: 与 timer_start 相对应, 调用 stop 后定时器被终止,下次调用 start 时,定时器按照 timer_init 时设置的参数重新定时计数

## 2.9 timer_get_counter

函数原型: int64_t (*get_counter)(uint32_t id);

函数功能: 返回本次定时剩余时间,单位:ms

函数参数:

id: 定时器 id 号，可配置范围[1,TIMER_DEFAULT_MAX_CNT]

返 回 值: >=0:返回本次定时剩余时间; -1: 失败

# 3 watchdog

该套看门狗接口是基于芯片的硬件看门狗实现的，最小的 timeout 时间为一秒，详细使用方法看 API 接口的说明以及看门狗的测试代码。

## 3.1 源码文件

头 文 件: sdk/include/watchdog/watchdog_manager.h

源 文 件: sdk/watchdog/watchdog_manager.c

测试程序: sdk/examples/watchdog/

## 3.2 get_watchdog_manager

函数原型: watchdog_manager *get_watchdog_manager(void);

函数功能: 获取 watchdog_manager 句柄

函数参数: 无

返 回 值: 返回 watchdog_manager 结构体指针

其　他: 通过该结构体指针访问 watchdog_manager 内部提供的方法

## 3.3 watchdog_init

函数原型: int32_t watchdog_init(uint32_t timeout);

函数功能: 看门狗初始化

函数参数:

timeout:看门狗超时的时间, 以秒为单位, 其值必须大于零

返 回 值: 0: 成功; -1: 失败

其　他: 必须优先调用 init 函数初始化看门狗和设置 timeout, 可被多次调用

## 3.4 watchdog_deinit

函数原型: void watchdog_deinit(void);

函数功能: 看门狗释放

函数参数: 无

返 回 值: 无

其　他: 对应 init 函数, 不再使用看门狗时调用, 该函数将关闭看门狗, 释放设备

## 3.5 watchdog_reset

函数原型: int32_t watchdog_reset(void);

函数功能: 看门狗喂狗

函数参数: 无

返 回 值:  0: 成功;  -1: 失败

其　他: 使能看门狗后, 在 timeout 时间内不调用此函数, 系统将复位

## 3.6 watchdog_enable

函数原型: int32_t watchdog_enable(void);

函数功能: 看门狗使能

函数参数: 无

返 回 值:  0:成功;  -1:失败

其　他: 在 init 函数初始化或 disable 函数关闭看门狗之后, 调用此函数启动看门狗

## 3.7 watchdog_disable

函数原型: int32_t watchdog_disable(void);

函数功能: 看门狗关闭

函数参数: 无

返 回 值:  0: 成功;  -1: 失败

其　他: 对应 enable 函数, 区别 deinit 函数在于, 调用此函数之后, 能通过 enable 函数重新启动

# 4 power

该套电源管理接口是基于内核标准的接口来实现的，详细使用方法看 API 接口的说明以及 power 的测试代码。

## 4.1 源码文件

头 文 件: sdk/include/power/power_manager.h

源 文 件: sdk/power/power_manager.c

测试程序: sdk/examples/power/

## 4.2 get_power_manager

函数原型: power_manager *get_power_manager(void);

函数功能: 获取 power_manager 句柄

函数参数: 无

返 回 值: 返回 power_manager 结构体指针

其　他: 通过该结构体指针访问 power_manager 内部提供的方法

## 4.3 pm_power_off

函数原型: int32_t pm_power_off(void);

函数功能: 关机

函数参数: 无

返 回 值:  -1: 失败; 成功将关机

## 4.4 pm_reboot

函数原型: int32_t pm_reboot(void);

函数功能: 进入休眠

函数参数: 无

返 回 值:  -1: 失败; 成功将重启系统

## 4.5 pm_sleep

函数原型: int32_t pm_sleep(void);

函数功能: 进入休眠

函数参数: 无

返 回 值:  0: 成功;  -1: 失败

# 5 pwm

该套接口是基于 JZ PWM generic drivers 实现的，最大支持 5 路 PWM 输出，详细使用方法看 API 接口的说明以及 PWM 的测试代码。

## 5.1 源码文件

头 文 件: sdk/include/pwm/pwm_manager.h

源 文 件: sdk/pwm/pwm_manager.c

测试程序: sdk/examples/pwm/

## 5.2 get_pwm_manager

函数原型: pwm_manager *get_pwm_manager(void);

函数功能: 获取 pwm_manager 句柄

函数参数: 无

返 回 值: 返回 pwm_manager 结构体指针

其　他: 通过该结构体指针访问 pwm_manager 内部提供的方法

## 5.3 pwm_init

函数原型: int32_t pwm_init(enum pwm id, enum pwm_active level);

函数功能: PWM 通道初始化

函数参数:

id: PWM 通道 id, 其值必须小于 PWM_CHANNEL_MAX

level: PWM 通道工作的有效电平, 例如: PWM 控制 LED, 当低电平 LED 亮, 即这个参数的值是 ACTIVE_LOW

返 回 值: 0: 成功; -1: 失败

其　他: 在使用每路 PWM 通道之前，必须优先调用 pwm_init 函数进行初始化

## 5.4 pwm_deinit

函数原型: void pwm_deinit(enum pwm id);

函数功能: PWM 通道释放

函数参数: 无

返 回 值: 无

其　他: 对应于 init 函数, 不再使用 PWM 某个通道时,应该调用此函数释放

## 5.5 pwm_setup_freq

函数原型: int32_t pwm_setup_freq(enum pwm id, uint32_t freq);

函数功能: 设置 PWM 通道的频率，实际上是设置周期

函数参数:

id: PWM 通道 id, 其值必须小于 PWM_CHANNEL_MAX

freq: 周期值, 单位为 ns, 其值在[PWM_FREQ_MIN, PWM_FREQ_MAX]之间

返 回 值: 0: 成功; -1: 失败

其　他: 此函数可以不调用, 即使用默认频率: 30000ns

## 5.6 pwm_setup_duty

函数原型: int32_t pwm_setup_duty(enum pwm id, uint32_t duty);

函数功能: 设置 PWM 通道的占空比

函数参数:

id: PWM 通道 id, 其值必须小于 PWM_CHANNEL_MAX

duty: 占空比, 其值为 0 ~ 100 区间

返 回 值: 0: 成功; -1: 失败

其　他: 这里不用关心 IO 输出的有效电平

## 5.7 pwm_setup_state

函数原型: int32_t pwm_setup_state(enum pwm id, enum pwm_state state);

函数功能: 设置 PWM 通道的工作状态

函数参数:

id: PWM 通道 id, 其值必须小于 PWM_CHANNEL_MAX

state: 指定 PWM 的工作状态, 为 0: disable, 非 0: enable

返 回 值: 0: 成功; -1: 失败

其　他: 此函数不需要在 setup_freq 或 setup_duty 之前调用, 主要用于暂停/开始 PWM 的工作。
再重新开始工作时, PWM 保持之前的 freq 和 duty 继续工作

# 6 uart

该套接口是基于内核标准的 uart 设备应用编程方法实现的，最大支持 3 个 uart 通道。

## 6.1 源码文件

头 文 件: sdk/include/uart/uart_manager.h

源 文 件: sdk/uart/uart_manager.c

测试程序: sdk/examples/uart/

## 6.2 配置参数

UART_MAX_CHANNELS 表示系统支持的最大 UART 通道数，默认设置为 3。

## 6.3 get_uart_manager

函数原型: uart_manager *get_uart_manager(void);

函数功能: 获取 uart_manager 句柄

函数参数: 无

返 回 值: 返回 uart_manager 结构体指针

其　他: 通过该结构体指针访问 uart_manager 内部提供的方法

## 6.4 uart_init

函数原型: int32_t (*init)(char* devname, uint32_t baudrate, uint8_t date_bits,
uint8_t parity, uint8_t stop_bits);

函数功能: 串口初始化

函数参数:

devname: 串口设备名称

例如: 普通串口设备/dev/ttySX， usb 转串口/dev/ttyUSBX; X 为设备序号

baudrate: 波特率 单位:bis per second

波特率取值范围 1200~3000000

date_bits: 数据位宽

stop_bits: 停止位宽

parity_bits: 奇偶校验位

可选设置 UART_PARITY_NONE, 无校验

UART_PARITY_ODD, 奇校验
UART_PARITY_EVEN, 偶校验

UART_PARITY_MARK, 校验位总为 1
UART_PARITY_SPACE 校验位总为 0

返 回 值: 0: 成功; -1: 失败

其 他: 每个通道在使用前必须优先调用 uart_init，默认流控不开启

## 6.5 uart_deinit

函数原型: void_t uart_deinit(char* devname);

函数功能: 串口释放

函数参数:

devname: 串口设备名称

例如: 普通串口设备/dev/ttySX， usb 转串口/dev/ttyUSBX; X 为设备序号

返 回 值: 无

## 6.6 uart_flow_control

函数原型: int32_t (*flow_control)(char* devname, uint8_t flow_ctl);

函数功能: 串口流控设置

函数参数:

devname: 串口设备名称

例如: 普通串口设备/dev/ttySX， usb 转串口/dev/ttyUSBX; X 为设备序号

flow_ctl: 流控选项

UART_FLOWCONTROL_NONE: 无流控

UART_FLOWCONTROL_XONXOFF: 软件流控使用 XON/XOFF 字符

UART_FLOWCONTROL_RTSCTS: 硬件流控使用 RTS/CTS 信号

UART_FLOWCONTROL_DTRDSR: 硬件流控使用 DTR/DSR 信号

返 回 值: 0: 成功; -1: 失败

## 6.7 uart_read

函数原型: int32_t (*read)(char* devname, const void* buf, uint32_t count,
uint32_t timeout_ms);

函数功能: 串口读取数据

函数参数:

devname: 串口设备名称

例如: 普通串口设备/dev/ttySX， usb 转串口/dev/ttyUSBX; X 为设备序号

buf: 存储读取数据的缓存区指针, 不能是空指针

count: 读取的字节数

timeout_ms 读取超时时间，单位 ms

返 回 值: 大于 0: 成功读取到的字节数; -1: 失败

其 他: 无

## 6.8 uart_write

函数原型: int32_t (*write)(char* devname, const void* buf, uint32_t count,
uint32_t timeout_ms);

函数功能: 串口写入数据

函数参数:

devname: 串口设备名称

例如: 普通串口设备/dev/ttySX， usb 转串口/dev/ttyUSBX; X 为设备序号

buf： 指向存储待写入数据的缓存区指针, 不能是空指针

count: 写入的字节数

timeout_ms 读取超时时间，单位 ms

返 回 值: 大于 0: 成功写入的字节数; -1: 失败

# 7 i2c

该套接口是基于内核标准的 i2c 设备应用编程方法实现的，最大支持 3 条 i2c 总线，详细使用方法看 API 接口的说明以及 i2c 的测试代码。

## 7.1 源码文件

头 文 件: sdk/include/i2c/i2c_manager.h

源 文 件: sdk/i2c/i2c_manager.c

测试程序: sdk/examples/i2c/

## 7.2 配置参数

I2C_DEV_ADDR_LENGTH: 读写 i2c 设备所发送的地址的长度, 以 BIT 为单位, 有 8BIT 或 16BIT, 根据实际使用的 i2c 设备修改此宏值，默认是 8BIT

I2C_CHECK_READ_ADDR: 对设备的这个地址进行读操作, 以检测 i2c 总线上有没有 chip_addr 这个从设备, 可根据实际修改该宏值

I2C_ACCESS_DELAY_US: 对设备一次读写操作后, 在进行下次读写操作时的延时时间,单位:us, 值不能太小, 否则导致读写出错

## 7.3 get_i2c_manager

函数原型: i2c_manager *get_i2c_manager(void);

函数功能: 获取 i2c_manager 句柄

函数参数: 无

返 回 值: 返回 i2c_manager 结构体指针

其　他: 通过该结构体指针访问 i2c_manager 内部提供的方法

## 7.4 i2c_init

函数原型: int32_t i2c_init(struct i2c_unit *i2c);

函数功能: I2C 初始化

函数参数:

i2c:每个 I2C 设备对应 struct i2c_unit 结构体指针, 必须先初始化结构体的成员

其中: id 的值应大于 0, 小于 I2C_BUS_MAX; chip_addr: 为设备的 7 位地址

返 回 值: 0: 成功; 非 0: 失败

其　他: 必须优先调用 init 函数, 可以被多次调用, 用于初始化不同的 I2C 设备

## 7.5 i2c_deinit

函数原型: void_t i2c_deinit(struct i2c_unit *i2c);

函数功能: I2C 设备释放

函数参数:

i2c:每个 I2C 设备对应 struct i2c_unit 结构体指针, 必须先初始化结构体的成员

其中: id 的值应大于 0, 小于 I2C_BUS_MAX; chip_addr: 为设备的 7 位地址

返 回 值: 无

其　他: 无

## 7.6 i2c_read

函数原型: int32_t i2c_read(struct i2c_unit *i2c, uint8_t *buf, int addr, int count);

函数功能: I2C 初始化

函数参数:

i2c:每个 I2C 设备对应 struct i2c_unit 结构体指针, 必须先初始化结构体的成员

其中: id 的值应大于 0, 小于 I2C_BUS_MAX; chip_addr: 为设备的 7 位地址

buf: 指向存储读取数据的缓存区指针, 不能是空指针

addr: 指定从 I2C 设备的哪个地址开始读取数据

count: 读取的字节数

返 回 值: 0: 成功; -1: 失败

其　他: 无

## 7.7 i2c_write

函数原型:int32_t i2c_write(struct i2c_unit *i2c, uint8_t *buf, int addr, int count);

函数功能: I2C 初始化

函数参数:

i2c:每个 I2C 设备对应 struct i2c_unit 结构体指针, 必须先初始化结构体的成员

其中: id 的值应大于 0, 小于 I2C_BUS_MAX; chip_addr: 为设备的 7 位地址

buf: 指向存储待写入数据的缓存区指针, 不能是空指针

addr: 指定从 I2C 设备的哪个地址开始写入数据

count: 写入的字节数

返 回 值: 0: 成功; -1: 失败

其　他: 无

# 8 camera

该套接口是基于 JZ CIM & sensor drivers 实现的，详细使用方法看 API 接口的说明以及 camera 的测试代码。

## 8.1 源码文件

头 文 件: sdk/include/camera/camera_manager.h

源 文 件: sdk/camera/camera_manager.c

测试程序: sdk/examples/camera/

## 8.2 配置参数

SENSOR_SET_REG_DELAY_US: sensor 每设置一个寄存器之后的延时时间, 单位是 us, 可以根据实际要求修改此宏值

SENSOR_ADDR_LENGTH: sensor 寄存器地址的长度, 以 BIT 为单位, 有 8BIT 或 16BIT, 应该根据实际使用的 sensor 修改此宏值, 默认是 8BIT

## 8.3 get_camera_manager

函数原型: camera_manager *get_camera_manager(void);

函数功能: 获取 camera_manager 句柄

函数参数: 无

返 回 值: 返回 camera_manager 结构体指针

其　他: 通过该结构体指针访问 camera_manager 内部提供的方法

## 8.4 camera_init

函数原型: void camera_init(void);

函数功能: 摄像头初始化

函数参数: 无

返 回 值: 0:成功; -1:失败

其　他: 必须优先调用 camera_init

## 8.5 camera_deinit

函数原型: void camera_deinit(void);

函数功能: 摄像头释放

函数参数: 无

返 回 值:  0: 成功;  -1: 失败

其　他: 对应 camera_init, 不再使用 camera 时调用

## 8.6 camera_read

函数原型: int32_t camera_read(uint8_t *yuvbuf, uint32_t size);

函数功能: 读取摄像头采集数据, 保存在 yuvbuf 指向的缓存区中

函数参数:

yuvbuf: 图像缓存区指针, 缓存区必须大于或等于读取的大小

size: 读取数据大小,字节为单位, 一般设为 image_size

返 回 值:  -1: 失败; 成功: 返回实际读取到的字节数

其　他: 在此函数中会断言 yuvbuf 是否等于 NULL, 如果为 NULL, 将推出程序

## 8.7 set_img_param

函数原型: int32_t set_img_param(struct camera_img_param *img);

函数功能: 设置控制器捕捉图像的分辨率和像素深度

函数参数:

img: struct img_param_t 结构体指针, 指定图像的分辨率和像素深度

返 回 值:  0: 成功;  -1: 失败

其　他: 在此函数中会断言 img 是否等于 NULL, 如果为 NULL, 将推出程序

## 8.8 set_timing_param

函数原型: int32_t set_timing_param(struct camera_timing_param *timing);

函数功能: 设置控制器时序, 包括 mclk 频率、pclk 有效电平、hsync 有效电平、vsync 有效电平

函数参数:

timing: struct timing_param_t 结构体指针, 指定 mclk 频率、pclk 有效电平、hsync 有效电平、vsync 有效电平。 在 camera_init 函数中分别初始化为:24000000、0、1、1, 为 1 是高电平有效, 为 0 则是低电平有效

返 回 值: 0: 成功; -1: 失败

其　他: 在此函数中会断言 timing 是否等于 NULL, 如果为 NULL, 将推出程序

## 8.9 sensor_setup_addr

函数原型: int32_t sensor_setup_addr(int32_t chip_addr);

函数功能: 设置摄像头 sensor 的 i2c 地址, 为保证 probe sensor ID 成功，应该调用此函数

chip_addr: 摄像头 sensor 的 I2C 地址, 不包括读写控制位

size: 读取数据大小, 字节为单位, 一般设为 image_size

返 回 值: 0: 成功; -1: 失败

其　他: 在此函数中会断言 chip_addr 是否大于 0, 如果断言失败, 将推出程序

## 8.10 sensor_setup_regs

函数原型: int32_t sensor_setup_regs(const struct camera_regval_list *vals);

函数功能: 设置摄像头 sensor 的多个寄存器, 用于初始化 sensor

函数参数:

vals: struct regval_list 结构体指针, 通常传入 struct regval_list 结构数组

返 回 值: 0: 成功; -1: 失败

其　他: 无

## 8.11 sensor_write_reg

函数原型: int32_t sensor_write_reg(uint32_t regaddr, uint8_t regval);

函数功能: 设置摄像头 sensor 的单个寄存器

函数参数:

regaddr: 摄像头 sensor 的寄存器地址

regval: 摄像头 sensor 寄存器的值

返 回 值: 0: 成功; -1: 失败

其　他: 无

## 8.12 sensor_read_reg

函数原型: uint8_t sensor_read_reg(uint32_t regaddr);

函数功能: 读取摄像头 sensor 某个寄存器的值

函数参数:

regaddr: 摄像头 sensor 的寄存器地址

返 回 值: -1: 失败; 其他: 寄存器的值

其　他: 无

# 9 flash

## 9.1 源码文件

头 文 件: sdk/include/flash/flash_manager.h

源 文 件: sdk/flash/flash_manager.c

测试程序: sdk/examples/flash/

## 9.2 get_flash_manager

函数原型: struct flash_manager* get_flash_manager(void);

函数功能: 获取 flash_manager 操作指针, 以操作 flash_manager 内部方法

返回值: 返回 flash_manager 结构体指针

其 他: 通过该结构体指针访问 flash_manager 内部提供的方法

## 9.3 flash_init

函数原型: int32_t (*init)(void);

函数功能: flash 初始化

返回值: 0:成功; -1: 失败

其 他: 在 flash 的读/写/擦除操作之前，首先执行初始化

## 9.4 flash_deinit

函数原型: int32_t (*deinit)(void);

函数功能: flash 释放

返回值: 0:成功; -1: 失败

其 他: 与 flash_init 相对应

## 9.5 flash_get_erase_unit

函数原型: int32_t (*get_erase_unit)(void);

函数功能: 获取 flash 擦除单元， 单位: bytes

返回值: 大于 0: 成功返回擦除单元大小 等于 0: 失败

其　他: 在 erase 调用之前使用

## 9.6 flash_erase

函数原型: int64_t (*erase)(int64_t offset, int64_t length);

函数功能: flash 擦除

函数参数:

offset: flash 片内偏移物理地址

length: 擦除大小，单位: byte, 该大小必须是擦除单元大小的整数倍

返回值: 0:成功 -1:失败

## 9.7 flash_read

函数原型: int64_t (*read)(int64_t offset, void* buf, int64_t length);

函数功能: flash 读取

函数参数:

offset: flash 片内偏移物理地址

buf: 读取缓冲区

length: 擦除大小，单位: byte

返回值: 大于等于 0: 返回成功读取的字节数 -1:失败

## 9.8 flash_write

函数原型: int64_t (*write)(int64_t offset, void* buf, int64_t length);

函数功能: flash 写入

函数参数:

offset: flash 片内偏移物理地址

buf: 写入缓冲区

length: 写入大小，单位: byte

返回值: 大于等于 0: 返回成功写入的字节数 -1:失败

# 10 efuse

该套接口提供了 efuse 的读写方法，详细使用方法看 API 接口的说明以及 camera 的测试代码。

## 10.1 源码文件

头 文 件: sdk/include/efuse/efuse_manager.h

源 文 件: sdk/efuse/efuse_manager.c

测试程序: sdk/examples/efuse/

## 10.2 get_efuse_manager

函数原型: efuse_manager *get_efuse_manager(void);

函数功能: 获取 efuse_manager 句柄

函数参数: 无

返 回 值: 返回 efuse_manager 结构体指针

其　他: 通过该结构体指针访问 efuse_manager 内部提供的方法

## 10.3 efuse_read

函数原型: efuse_read(enum efuse_segment seg_id, uint32_t *buf, uint32_t length);

函数功能: 读 efuse 指定的段

函数参数:

seg_id: 读取 EFUSE 段的 id

buf: 存储读取数据的缓存区指针

length: 读取的长度，以字节为单位

返 回 值: 0: 成功; -1: 失败

## 10.4 efuse_write

函数原型: efuse_write(enum efuse_segment seg_id, uint32_t *buf, uint32_t length);

函数功能: 写数据到指定的 efuse 段

函数参数:

seg_id: 写 EFUSE 目标段的 id

buf: 存储待写入数据的缓存区指针

length: 写入的长度，以字节为单位

返 回 值: 0: 成功; -1: 失败

# 11 rtc

该套接口基于 kernel 的 rtc 应用设计标准实现的，提供了 rtc 设备的读写方法，详细使用方法看 API 接口的说明以及 rtc 的测试代码。

## 11.1 源码文件

头 文 件: sdk/include/rtc/rtc_manager.h

源 文 件: sdk/rtc/rtc_manager.c

测试程序: sdk/examples/rtc/

## 11.2 get_rtc_manager

函数原型: rtc_manager *get_rtc_manager(void);

函数功能: 获取 rtc_manager 句柄

函数参数: 无

返 回 值: 返回 rtc_manager 结构体指针

其　他: 通过该结构体指针访问 rtc_manager 内部提供的方法

## 11.3 rtc_read

函数原型: int32_t rtc_read(**struct** rtc_time *time);

函数功能: rtc 读时间

函数参数:

time: 获取时间参数

```
struct rtc_time {
    int tm_sec;         秒 – 取值区间为[0,59]
    int tm_min;         分 - 取值区间为[0,59]
    int tm_hour;        时 - 取值区间为[0,23]
    int tm_mday;        一个月中的日期 - 取值区间为[1,31]
```

int tm_mon;　　月份（从一月开始，0 代表一月） - 取值区间为[0,11]

int tm_year;　　年份，其值等于实际年份减去 1900

int tm_wday;　　星期 – 取值区间为[0,6]其中 0 代表星期天，1 代表星期一，以此类推

int tm_yday;　　从每年的 1 月 1 日开始的天数 – 取值区间为[0,365]，

其中 0 代表 1 月 1 日，1 代表 1 月 2 日，以此类推

int tm_isdst;　　夏令时标识符，实行夏令时的时候，tm_isdst 为正。

不实行夏令时的进候，tm_isdst 为 0；不了解情况时，tm_isdst()为负。

};

返 回 值: 0: 成功; -1: 失败

## 11.4 rtc_write

函数原型: int32_t rtc_write(**const struct** rtc_time *time);

函数功能: rtc 读时间

函数参数:

time: 设置时间参数（参考 rtc_read 的参数说明）

返 回 值: 0: 成功; -1: 失败

# 12 spi

该套接口基于 kernel 的 spi 应用设计标准实现的，提供了 spi 设备的读写方法，详细使用方法看 API 接口的说明以及 spi 的测试代码。

## 12.1 源码文件

头 文 件: sdk/include/spi/spi_manager.h

源 文 件: sdk/spi/spi_manager.c

测试程序: sdk/examples/spi/

## 12.2 get_spi_manager

函数原型: spi_manager *get_spi_manager(void);

函数功能: 获取 spi_manager 句柄

函数参数: 无

返 回 值: 返回 spi_manager 结构体指针

其　他: 通过该结构体指针访问 spi_manager 内部提供的方法

## 12.3 spi_init

函数原型: int32_t spi_init(enum spi id, uint8_t mode, uint8_t bits, uint32_t speed);

函数功能: spi 设备初始化

函数参数:

id: spi 设备 id, 其值必须小于 SPI_DEVICE_MAX

mode: spi 设备工作模式

bits: spi 读写一个 word 的位数, 其值有: 8/16/32, 通常为 8

speed: spi 读写最大速率

返 回 值: 0:成功; -1: 失败

其　他: 在使用每个 SPI 设备之前，必须优先调用 init 函数进行初始化

## 12.4 spi_deinit

函数原型: void spi_deinit(enum spi id);

函数功能: spi 设备初始化

函数参数:

id: spi 设备 id, 其值必须小于 SPI_DEVICE_MAX

返 回 值: 无

其　他: 对应于 init 函数, 不再使用某个 SPI 设备时, 应该调用此函数释放

## 12.5 spi_read

函数原型: int32_t spi_read(enum spi id, uint8_t *rxbuf, uint32_t length);

函数功能: spi 读设备

函数参数:

id: spi 设备 id, 其值必须小于 SPI_DEVICE_MAX

rxbuf: 存储读取数据的缓存区指针, 不能是空指针

length: 读取的字节数

返 回 值: 大于等于 0: 成功返回实际读取的字节数 -1: 失败

## 12.6 spi_write

函数原型: int32_t spi_write(enum spi id, uint8_t *txbuf, uint32_t length);

函数功能: spi 读设备

函数参数:

id: spi 设备 id, 其值必须小于 SPI_DEVICE_MAX

txbuf: 存储待写入数据的缓存区指针, 不能是空指针

length: 写入的字节数

返 回 值: 大于等于 0: 成功返回实际写入到的字节数 -1: 失败

## 12.7 spi_transfer

函数原型: int32_t spi_transfer(enum spi id, uint8_t *txbuf, uint8_t *rxbuf, uint32_t length);

函数功能: spi 读设备

函数参数:

id: spi 设备 id, 其值必须小于 SPI_DEVICE_MAX

txbuf: 存储待写入数据的缓存区指针, 不能是空指针

rxbuf: 存储读取数据的缓存区指针, 不能是空指针

length: 读写的字节数

返 回 值: 0: 成功, -1: 失败

# 13 usb

该套接口目前支持的 usb 设备包括 hid 和 cdc acm。

## 13.1 源码文件

头 文 件: sdk/include/usb/usb_manager.h

源 文 件: sdk/usb/usb_device_manager.c

测试程序: sdk/examples/usb/

## 13.2 配置参数

USB_DEVICE_MAX_COUNT 表示系统支持的最大 USB 设备数，默认值为 1

## 13.3 get_usb_device_manager

函数原型: struct usb_device_manager*  get_usb_device_manager(void);

函数功能: 获取 usb_device_manager 操作指针, 以操作 usb_device_manager 内部方法

返回值: 返回 usb_device_manager 结构体指针

其　他: 通过该结构体指针访问 usb_device_manager 内部提供的方法

## 13.4 usb_device_init

函数原型: int32_t (*init)(char* devname);

函数功能: usb 设备初始化

函数参数:

devname: usb 设备名称  例如: 共支持 2 类 usb 设备, 分别是 hid 设备和 cdc acm 设备

hid 设备名称是/dev/hidg, cdc acm 设备名称是/dev/ttyGS0

返 回 值:  0: 成功;  -1: 失败

其　他: 每个设备在使用前必须优先调用 usb_device_init

## 13.5 usb_device_deinit

函数原型: int32_t (*deinit)(char* devname);

函数功能: usb 设备释放

函数参数:

devname: usb 设备名称, 例如: 共支持 2 类 usb 设备, 分别是 hid 设备和 cdc acm 设备

hid 设备名称是/dev/hidg0, cdc acm 设备名称是/dev/ttyGS0

返 回 值: 0: 成功; -1: 失败

其　他: 每个设备在使用前必须优先调用 usb_device_init, 与 usb_device_deinit 函数相对应

## 13.6 usb_device_switch_func

函数原型: int32_t (*switch_func)(char* switch_to, char* switch_from);

函数功能: usb 功能设备切换

函数参数:

switch_to: 目标切换功能设备名称

switch_from: 当前功能设备名称

举例: 从 hid 切换到 cdc acm, switch_from 应设置为/dev/ttyhidg0, switch_to 应设置为 /dev/ttyGS0

返 回 值: 0: 成功; -1: 失败

其 他: switch_from 指定的设备必须首先 init 后，才能调用该函数。

功能设备切换的另一种方法是先 deinit 释放当前设备，再 init 初始化新设备, 详细信息请参考测试程序为 test_usb_switch。

## 13.7 usb_device_get_max_transfer_unit

函数原型: uint32_t (*get_max_transfer_unit)(char* devname);

函数功能: 获取 usb 最大传输单元

函数参数:

devname: usb 设备名称

例如: 共支持 2 类 usb 设备, 分别是 hid 设备和 cdc acm 设备

hid 设备名称是/dev/hidg0

cdc acm 设备名称是/dev/ttyGS0

返 回 值: 大于 0: 成功获取最大传输单元大小; 0: 失败

其 他: 每个设备在使用前必须优先调用 usb_device_init

## 13.8 usb_device_write

函数原型: int32_t (*write)(char* devname, void* buf, uint32_t count,uint32_t timeout_ms);

函数功能: 写数据

函数参数:

devname: usb 设备名称

例如: 共支持 2 类 usb 设备, 分别是 hid 设备和 cdc acm 设备

hid 设备名称是/dev/hidg0

cdc acm 设备名称是/dev/ttyGS0

buf: 存储写入数据的缓存区指针

count: 要写入的字节数

timeout_ms: 写入超时时间，单位 ms

返 回 值: 大于等于 0: 成功读取到的字节数; -1: 失败

其　他: 该函数在指定超时时间内写入 count 个字节数据，返回实际写入大小, 在使用之前要调用 usb_device_init

## 13.9 usb_device_read

函数原型: int32_t (*read)(char* devname, void* buf, uint32_t count,uint32_t timeout_ms);

函数功能: 读数据

函数参数:

devname: usb 设备名称

例如: 共支持 2 类 usb 设备, 分别是 hid 设备和 cdc acm 设备

hid 设备名称是/dev/hidg0

cdc acm 设备名称是/dev/ttyGS0

buf: 存储读取数据的缓存区指针

count: 读取的字节数

timeout_ms: 读取超时时间，单位 ms

返 回 值: 大于等于 0: 成功读取字节数; -1: 失败

其　他: 该函数在指定超时时间内读取 count 个字节数据，返回实际读取大小, 在使用之前要先调用 usb_device_init

# 14 Security

该套接口目前支持 AES128/AES192/AES256 加解密

## 14.1 源码文件

头 文 件: sdk/include/security/security_manager.h

源 文 件: sdk/security/security_manager.c

测试程序: sdk/examples/security/

## 14.2 get_security_manager

函数原型: struct security_managerr*  get_security_manager(void);

函数功能: 获取 security_manager 操作指针, 以操作 security_manager 内部方法

返回值: 返回 security_manager 结构体指针

其　他: 通过该结构体指针访问 security_manager 内部提供的方法

## 14.3 security_init

函数原型: int32_t security_init(void);

函数功能: security 模块初始化

函数参数: 无

返 回 值:  0:成功; -1: 失败

## 14.4 security_deinit

函数原型: void security_deinit(void);

函数功能: security 模块释放

函数参数: 无

返 回 值:  0:成功; -1: 失败

## 14.5 simple_aes_load_key

函数原型: int32_t (*simple_aes_load_key)(struct aes_key* aes_key);

函数功能: 加载 AES key

返 回 值: 0: 成功; -1: 失败

## 14.6 simple_aes_crypt

函数原型: int32_t (*simple_aes_crypt)(struct aes_data* aes_data);

函数功能: AES 加/解密输入数据

返 回 值: 0: 成功; -1: 失败

# 15 zigbee

该套接口基于串口与从模块 TI CC2530 进行交互，结合建立在 Z-Stack 协议栈之上的应用工程(CC2530 工程源码见附件)

以下只提供 API 说明，关于 zigbee 功能开发详细见《iLock_Zigbee_Develop_Manual__CN.pdf》

## 15.1 源码文件

头 文 件: sdk/include/zigbee/zigbee_manager.h

源 文 件: sdk/zigbee/zigbee/zigbee_manager.c

测试程序: sdk/examples/zigbee/

## 15.2 get_zigbee_manager

函数原型: uart_zigbee_manager* get_zigbee_manager(void);

函数功能: 获取 uart_zigbee_manager 句柄

函数参数: 无

返 回 值: 返回 uart_zigbee_manager 结构体指针

其　他: 通过该结构体指针访问 uart_zigbee_manager 内部提供的方法

## 15.3 init

函数原型: int (*init)(uart_zigbee_recv_cb recv_cb);

函数功能: 初始化 zigbee 功能模块及相关组件

函数参数:

recv_cb: 处理解析到完整数据包后的回调函数，由用户编写并传入

返 回 值: 0: 成功; <0: 失败

其　他: 在使用 zigbee 模块之前，必须优先调用 init 函数进行初始化

## 15.4 deinit

函数原型: void (*deinit)(void);

函数功能:释放 zigbee 模块资源及相关组件

函数参数: 无

返 回 值: 无

其　他: 对应于 init 函数, 不再使用时,应该调用此函数释放

## 15.5 reset

函数原型: int (*reset)(void);

函数功能: 硬件复位

函数参数: 无

返 回 值:  0: 成功;  -1: 失败

其　他: 此函数操作硬件 IO 复位 CC2530，需等待 CC2530 重启

## 15.6 ctrl

函数原型: int (*ctrl)(uint8_t* pl, uint16_t len);

函数功能: 控制数据透明传输

函数参数:

pl : 控制数据的载荷部分

len : 载荷数据长度

返 回 值:  0: 成功;  -1:入参非法; -2 发送失败

其　他: 阻塞发送，只需要填入应用数据及长度，并关心返回值

## 15.7 get_info

函数原型: int (*get_info)(void);

函数功能: 获取设备当前配置信息

函数参数: 无

返 回 值:  0: 成功;  <0: 失败

其　他: 阻塞发送，此函数成功后，CC2530 随后上报设备参数，由接收回调函数接收

## 15.8 factory

函数原型: int (*factory)(void);

函数功能: 令 CC2530 的 zigbee 当前参数失效，恢复默认配置

函数参数: 无

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，需等待 CC2530 重启。

## 15.9 reboot

函数原型: int (*reboot)(void);

函数功能: 软件重启 CC2530

函数参数: 无

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值, 需等待 CC2530 重启。

## 15.10 set_role

函数原型: int (*set_role)(uint8_t role);

函数功能: 设置 zigbee 设备的角色

函数参数:

role : 00 协调器 01 路由器 02 终端节点

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值, 需等待 CC2530 重启。

## 15.11 set_panid

函数原型: int (*set_panid)(uint16_t panid);

函数功能: 设置 zigbee 设备的 pan id 指定个域网 ID 进行网络创建(协调器)或加入(节点)

函数参数:

panid : 0x0001~0xFFFF (0xFFFF: 随机)

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值, 需等待 CC2530 重启

## 15.12 set_channel

函数原型: int (*set_channel)(uint8_t channel);

函数功能: 设置 zigbee 设备工作的信道

函数参数:

channel: 0x0B~0x1A

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值, 需等待 CC2530 重启

## 15.13 set_key

函数原型: int (*set_key)(uint8_t* key, uint8_t keylen);

函数功能: 设置 aes 加密的密钥，16 bytes

函数参数:

key: 密钥

ketlen：密钥长度，固定 16

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，需等待 CC2530 重启

## 15.14 set_join_aging

函数原型: int (*set_join_aging)(uint8_t aging);

函数功能: 设置协调器和路由器角色下，允许设备加入网络的时限，终端无作用

函数参数:

aging : 0x00~0xFF， 0 为不可加入，0xFF 为永久可加入

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，不需要重启

## 15.15 set_cast_type

函数原型: int (*set_cast_type)(uint8_t type, uint16_t addr);

函数功能: 数据发送方式

函数参数:

type: 00 广播、01 点播、02 组播

addr : 指定 16bit 的发送目的地址，广播为 0xFFFF，组播为组 ID

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，不需要重启

## 15.16 set_group_id

函数原型: int (*set_group_id)(uint16_t id);

函数功能: 设置设备加入本地的组，用于接收相对应的组播数据，同时只加入一个组

函数参数:

id: 指定加入的组 ID

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，不需要重启

## 15.17 set_poll_rate

函数原型: int (*set_poll_rate)(uint16_t rate);

函数功能: 设置睡眠唤醒请求数据周期，配置功耗的关键参数，一般为睡眠唤醒周期

函数参数:

rate: 周期请求数据的时间，单位 ms 范围 0~7s

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，需等待 CC2530 重启，只对终端节点有效，协调器和路由器不睡眠

## 15.18 set_tx_power

函数原型: int (*set_tx_power)(int8_t power);

函数功能: 设置 zigbee 模块发射功率

函数参数:

power: 3 ~ -22 dbm

返 回 值: 0: 成功; <0: 失败

其　他: 阻塞发送，关心返回值，不需要重启

# 16 74hc595

该套接口用于控制 CMOS 移位寄存器 74hc595 输出的用户指定数据，详细使用方法看 API 接口的说明以及测试代码。

## 16.1 源码文件

头 文 件: sdk/include/74hc595/74hc595_manager.h

源 文 件: sdk/74hc595/74hc595_manager.c

测试程序: sdk/examples/74hc595

## 16.2 配置参数

SN74HC595_DEVICE_NUM: 该宏在 74hc595_manager.h 头文件中定义，表示 74hc595 设备的个数（注意多个 74hc595 串联算一个，目前驱动支持最多四个串联）。

## 16.3 get_sn74hc595_manager

函数原型: sn74hc595_manager *get_sn74hc595_manager(void);

函数功能: 获取 sn74hc595_manager 句柄

函数参数: 无

返 回 值: 返回 sn74hc595_manager 结构体指针

其　他: 通过该结构体指针访问 sn74hc595_manager 内部提供的方法

## 16.4 sn74hc595_init

函数原型: int32_t sn74hc595_init(enum sn74hc595 id);

函数功能: 74hc595 设备初始化

函数参数:

id: 每个 74hc696 设备对应的 id 号, id 的值应大于 0, 小于 SN74HC595_DEVICE_NUM

返 回 值: 0: 成功; <0: 失败

其　他: 在调用其它 API 之前，必须先调用此函数进行初始化

## 16.5 sn74hc595_deinit

函数原型: void sn74hc595_deinit(enum sn74hc595 id);

函数功能: 74hc595 设备释放

函数参数:

id: 每个 74hc696 设备对应的 id 号, id 的值应大于 0, 小于 SN74HC595_DEVICE_NUM

返 回 值: 无

其　他: 对应 sn74hc595_init 函数，不再使用设备必须释放

## 16.6 sn74hc595_get_outbits

函数原型: uint32_t sn74hc595_get_outbits(enum sn74hc595 id, uint32_t *out_bits);

函数功能: 从内核驱动获取 74hc595 设备的输出位大小

函数参数:

id: 74hc595 设备的 id, 两个 74hc595 级联看作是一个设备

out_bits: 保存 74hc595 输出数据的长度变量指针

返 回 值: 非 0: 成功; <0: 失败

其　他: 在不清楚内核设置 74hc595 设备的输出位大小时，可通过此函数获取

## 16.7 sn74hc595_write

函数原型: int32_t sn74hc595_write(enum sn74hc595 id, void *data, uint32_t out_bits);

函数功能: 74hc595 写数据

函数参数:

id: 74hc595 设备的 id，多个 74hc595 级联看作是一个设备

data: 写数据的指针

out_bits: 74hc595 输出数据的长度，单位：bits，例如，

一个 8-bit 74hc595, out_bits 为 8，两个 8-bit 74hc595 级联，out_bits 为 16，

通过 sn74hc595_get_outbits 函数可以从内核驱动中获取设定的值

返 回 值: 等于 out_bits: 成功; <0: 失败

## 16.8 sn74hc595_read

函数原型: int32_t sn74hc595_read(enum sn74hc595 id, void *data, uint32_t out_bits);

函数功能: 读取 74hc595 正在输出的数据

函数参数:

id: 74hc595 设备的 id，多个 74hc595 级联看作是一个设备

data: 存放读取数据的指针

out_bits: 74hc595 输出数据的长度，单位：bits，例如，

一个 8-bit 74hc595, out_bits 为 8，两个 8-bit 74hc595 级联，out_bits 为 16，

通过 sn74hc595_get_outbits 函数可以从内核驱动中获取设定的值

返 回 值: 等于 out_bits: 成功; <0: 失败

## 16.9 sn74hc595_clear

函数原型: int32_t sn74hc595_clear(enum sn74hc595 id);

函数功能: 清除 74hc595 移位寄存器，相当于写 0

函数参数:

id: 74hc595 设备的 id，多个 74hc595 级联看作是一个设备

返 回 值: 等于 out_bits: 成功; <0:

# 17 cypress

该套接口用于控制和处理 cypress MCU 的上报事件，详细使用方法看 API 接口的说明以及测试代码。

## 17.1 源码文件

头 文 件: sdk/include/cypress/cypress_manager.h

源 文 件: sdk/cypress/cypress_manager.c

测试程序: sdk/examples/cypress

## 17.2 get_cypress_manager

函数原型: cypress _manager *get_ cypress_manager(void);

函数功能: 获取 cypress_manager 句柄

函数参数: 无

返 回 值: 返回 cypress_manager 结构体指针

其　他: 通过该结构体指针访问 cypress _manager 内部提供的方法

## 17.3 cypress_init

函数原型: int32_t cypress_init(deal_card_report_handler card_handler);

函数功能: cypress 设备初始化

函数参数:

card_handler: 处理读卡上报事件的回调函数，有卡上报事件时自动被调用，

函数原型: void (*deal_card_report_handler)(int dev_fd)

返 回 值: 0: 成功; <0: 失败

其　他: 在调用其它 API 之前，必须先调用此函数进行初始化

## 17.4 cypress_deinit

函数原型: void cypress_deinit(void);

函数功能: cypress 设备

函数参数: 无

返 回 值: 无

其　他: 对应 cypress_init 函数，不再使用设备必须释放

## 17.5 cypress_ mcu_reset

函数原型: void cypress_mcu_reset(void);

函数功能: 复位 cypress MCU

函数参数: 无

返 回 值: 无

其　他: 主控判断 cypress MCU 工作异常时，可以调用此函数复位 cypress MCU

# 18 fpc fingerprint

该套接口用于操作 fpc 指纹传感器和获取到的指纹识别算法，详细使用方法看 API 接口的说明以及测试代码。

## 18.1 源码文件

头 文 件: sdk/include/fingerprint/fpc/fpc_fingerprint.h

源 文 件: sdk/fingerprint/fpc/fpc_fingerprint.c

测试程序: sdk/examples/fingerprint/fpc/test_fpc.c

## 18.2 fpc_fingerprint_init

函数原型: int fpc_fingerprint_init(notify_callback notify, void *param_config);

函数功能: 初始化

函数参数: notify : 事件通知的回调函数，参数：消息类型、注册百分比、模版 ID

由消息类型决定第二和第三个参数的有效性

typedef void (*notify_callback)(int msg, int percent, int finger_id);

param_config：用户配置参数

```
typedef struct customer_config
{
  int max_enroll_finger_num;            /* 指纹个数 ， 0～200*/
  int min_enroll_count_for_one_finger; /* 模版注册需要指纹个数，固定 3 */
  int enroll_timeout;                   /* 注册指纹超时时间 */
  int authenticate_timeout;             /* 验证指纹超时时间 */
  char*uart_devname;                    /* 算法芯片通讯串口 /dev/tty*  */
  char file_path[128];                  /* 模版及 ID 文件保存路径 */
 } customer_config_t;
```

返 回 值: 0 成功， －1 失败

## 18.3 fpc_fingerprint_destroy

函数原型: int fpc_fingerprint_destroy(void);

函数功能: 关闭指纹模块及算法

函数参数: 无

返 回 值: 0 成功， －1 失败

其它： 不再使用时调用此接口释放资源

## 18.4 fpc_fingerprint_reset

函数原型: int fpc_fingerprint_reset(void);

函数功能: 指纹传感器硬件复位

函数参数: 无

返 回 值: 0 成功， －1 失败

## 18.5 fpc_fingerprint_enroll

函数原型: int fpc_fingerprint_enroll(void);

函数功能: 注册指纹模版，立即返回，结果从回调函数返回

函数参数: 无

返 回 值: 0 成功， －1 失败

## 18.6 fpc_fingerprint_authenticate

函数原型: int fpc_fingerprint_authenticate(void);

函数功能: 验证指纹，立即返回 ，结果从回调函数返回

函数参数: 无

返 回 值: 0 成功， －1 失败

## 18.7 fpc_fingerprint_delete

函数原型: fpc_fingerprint_delete(int fingerprint_id, int type);

函数功能: 删除指纹模版，立即返回 ，结果从回调函数返回

函数参数: type：0 删除指定 ID， 1 删除所有指纹

fingerprint_id： 删除的 ID

返 回 值: 0 成功， －1 失败

## 18.8 fpc_fingerprint_cancel

函数原型: int fpc_fingerprint_cancel(void);

函数功能: 取消当前操作

函数参数：无

返 回 值: 0 成功， －1 失败

其它: 如注册过程中取消注册

## 18.9 fpc_fingerprint_get_template_info

函数原型: int fpc_fingerprint_get_template_info(uint32_t template_info[]);

函数功能: 获取已注册所有的指纹的 ID

函数参数： template_info： 返回所有已注册指纹 ID

返 回 值: 0 成功， －1 失败

# 19 lock cylinder

该套接口用于锁芯的钥匙加密注册、安全认证以及控制锁芯能否被钥匙转动，详细使用方法看 API 接口的说明以及测试代码。

## 19.1 源码文件

头 文 件: sdk/include/lock_cylinder/lock_cylinder.h

源 文 件: sdk/lock_cylinder/lock_cylinder.c

测试程序: sdk/examples/lock_cylinde /main.c

## 19.2 get_lock_cylinder_manager

函数原型: lock_cylinder_manager *get_lock_cylinder_manager(void);

函数功能: 获取 lock_cylinder_manager 句柄

函数参数: 无

返 回 值: 返回 lock_cylinder_manager 结构体指针

其　他: 通过该结构体指针访问 lock_cylinder_manager 内部提供的方法

## 19.3 lock_cylinder_init

函数原型: int32_t lock_cylinder_init(deal_key_detected_handler handler, void *handler_arg);

函数功能: cypress 设备初始化

函数参数:

handler: 检测到钥匙插入或拔出的回调函数，

函数原型: void (*deal_key_detected_handler)(void *arg)

handler_arg: 回调函数的参数

返 回 值: >=0: 成功; <0: 失败

其　他: 在调用其它 API 之前，必须先调用此函数进行初始化

## 19.4 lock_cylinder_deinit

函数原型: void lock_cylinder_deinit(void);

函数功能: lock cylinder 设备释放

函数参数: 无

返 回 值: 无

其　他: 不再使用设备应该释放

## 19.5 lock_cylinder_get_keystatue

函数原型: int8_t lock_cylinder_get_keystatue(void);

函数功能: 获取钥匙的拔插的状态

函数参数: 无

返 回 值: <0: 获取状态失败; true: 钥匙插入状态; false: 钥匙拔出状态

其　他: 此函数应该在回调函数调用，来实时获取钥匙的拔插状态，注意检测此函数的返回值

## 19.6 lock_cylinder_get_romid

函数原型: int32_t lock_cylinder_get_romid(uint8_t *romid);

函数功能: 获取钥匙的唯一 id

函数参数:

romid: 保存读取到 ROM ID 的缓存指针

返 回 值: =0: 成功; <0: 失败

其　他: 此函数应该在钥匙插入状态下调用

## 19.7 lock_cylinder_register_key

函数原型: int32_t lock_cylinder_register_key(void);

函数功能: 注册钥匙，把密钥写入钥匙的加密芯片

函数参数: 无

返 回 值: =0: 成功; <0: 失败

其　他: 此函数应该在钥匙插入状态下调用。注册成功后，保存 ROM ID 可以用来判断钥匙是否已经注册

## 19.8 lock_cylinder_authenticate_key

函数原型: int32_t lock_cylinder_authenticate_key(void);

函数功能: 验证已经注册过的钥匙

函数参数: 无

返 回 值: =0: 成功; <0: 失败

其　他: 此函数应该在钥匙插入状态下并判断钥匙已经注册后调用

## 19.9 lock_cylinder_power_ctrl

函数原型: int32_t lock_cylinder_power_ctrl(bool pwr_en);

函数功能: 控制锁芯磁管的电源

函数参数:

pwr_en: 锁心磁管上电或掉电，上电: 钥匙不能转动锁心; 掉电: 钥匙能够转动锁心

返 回 值: =0: 成功; <0: 失败

其　他: 此函数应该在钥匙插入状态下并验证成功后调用，如果钥匙验证不成功调用此函数，是无法控制锁芯掉电的