

**Ingenic SDK使用说明**

文档历史：

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

目录

[1 GPIO 1](#_Toc495393210)

[1.1 源码文件 1](#_Toc495393211)

[1.2 注意事项 1](#_Toc495393212)

[1.3 get\_gpio\_manager 1](#_Toc495393213)

[1.4 gpio\_init 1](#_Toc495393214)

[1.5 gpio\_deinit 1](#_Toc495393215)

[1.6 gpio\_open 2](#_Toc495393216)

[1.7 gpio\_close 2](#_Toc495393217)

[1.8 gpio\_get\_direction 2](#_Toc495393218)

[1.9 gpio\_set\_direction 3](#_Toc495393219)

[1.10 gpio\_get\_value 3](#_Toc495393220)

[1.11 gpio\_set\_value 4](#_Toc495393221)

[1.12 gpio\_set\_irq\_func 4](#_Toc495393222)

[1.13 gpio\_enable\_irq 4](#_Toc495393223)

[1.14 gpio\_disable\_irq 5](#_Toc495393224)

[2 timer 6](#_Toc495393225)

[2.1 源码文件 6](#_Toc495393226)

[2.2 配置参数 6](#_Toc495393227)

[2.3 注意事项 6](#_Toc495393228)

[2.4 get\_timer\_manager 6](#_Toc495393229)

[2.5 timer\_init 6](#_Toc495393230)

[2.6 timer\_deinit 7](#_Toc495393231)

[2.7 timer\_start 7](#_Toc495393232)

[2.8 timer\_stop 8](#_Toc495393233)

[2.9 timer\_get\_counter 8](#_Toc495393234)

[3 watchdog 9](#_Toc495393235)

[3.1 源码文件 9](#_Toc495393236)

[3.2 get\_watchdog\_manager 9](#_Toc495393237)

[3.3 watchdog\_init 9](#_Toc495393238)

[3.4 watchdog\_deinit 9](#_Toc495393239)

[3.5 watchdog\_reset 10](#_Toc495393240)

[3.6 watchdog\_enable 10](#_Toc495393241)

[3.7 watchdog\_disable 10](#_Toc495393242)

[4 power 11](#_Toc495393243)

[4.1 源码文件 11](#_Toc495393244)

[4.2 get\_power\_manager 11](#_Toc495393245)

[4.3 pm\_power\_off 11](#_Toc495393246)

[4.4 pm\_reboot 11](#_Toc495393247)

[4.5 pm\_sleep 12](#_Toc495393248)

[5 pwm 13](#_Toc495393249)

[5.1 源码文件 13](#_Toc495393250)

[5.2 get\_pwm\_manager 13](#_Toc495393251)

[5.3 pwm\_init 13](#_Toc495393252)

[5.4 pwm\_deinit 13](#_Toc495393253)

[5.5 pwm\_setup\_freq 14](#_Toc495393254)

[5.6 pwm\_setup\_duty 14](#_Toc495393255)

[5.7 pwm\_setup\_state 14](#_Toc495393256)

[6 uart 16](#_Toc495393257)

[6.1 源码文件 16](#_Toc495393258)

[6.2 配置参数 16](#_Toc495393259)

[6.3 get\_uart\_manager 16](#_Toc495393260)

[6.4 uart\_init 16](#_Toc495393261)

[6.5 uart\_deinit 17](#_Toc495393262)

[6.6 uart\_flow\_control 17](#_Toc495393263)

[6.7 uart\_read 18](#_Toc495393264)

[6.8 uart\_write 18](#_Toc495393265)

[7 i2c 20](#_Toc495393266)

[7.1 源码文件 20](#_Toc495393267)

[7.2 配置参数 20](#_Toc495393268)

[7.3 get\_i2c\_manager 20](#_Toc495393269)

[7.4 i2c\_init 20](#_Toc495393270)

[7.5 i2c\_deinit 21](#_Toc495393271)

[7.6 i2c\_read 21](#_Toc495393272)

[7.7 i2c\_write 21](#_Toc495393273)

[8 camera 23](#_Toc495393274)

[8.1 源码文件 23](#_Toc495393275)

[8.2 配置参数 23](#_Toc495393276)

[8.3 get\_camera\_manager 23](#_Toc495393277)

[8.4 camera\_init 23](#_Toc495393278)

[8.5 camera\_deinit 24](#_Toc495393279)

[8.6 camera\_read 24](#_Toc495393280)

[8.7 set\_img\_param 24](#_Toc495393281)

[8.8 set\_timing\_param 24](#_Toc495393282)

[8.9 sensor\_setup\_addr 25](#_Toc495393283)

[8.10 sensor\_setup\_regs 25](#_Toc495393284)

[8.11 sensor\_write\_reg 25](#_Toc495393285)

[8.12 sensor\_read\_reg 26](#_Toc495393286)

[9 flash 27](#_Toc495393287)

[9.1 源码文件 27](#_Toc495393288)

[9.2 get\_flash\_manager 27](#_Toc495393289)

[9.3 flash\_init 27](#_Toc495393290)

[9.4 flash\_deinit 27](#_Toc495393291)

[9.5 flash\_get\_erase\_unit 27](#_Toc495393292)

[9.6 flash\_erase 28](#_Toc495393293)

[9.7 flash\_read 28](#_Toc495393294)

[9.8 flash\_write 28](#_Toc495393295)

[10 efuse 30](#_Toc495393296)

[10.1 源码文件 30](#_Toc495393297)

[10.2 get\_efuse\_manager 30](#_Toc495393298)

[10.3 efuse\_read 30](#_Toc495393299)

[10.4 efuse\_write 30](#_Toc495393300)

[11 rtc 32](#_Toc495393301)

[11.1 源码文件 32](#_Toc495393302)

[11.2 get\_rtc\_manager 32](#_Toc495393303)

[11.3 rtc\_read 32](#_Toc495393304)

[11.4 rtc\_write 33](#_Toc495393305)

[12 spi 33](#_Toc495393306)

[12.1 源码文件 33](#_Toc495393307)

[12.2 get\_spi\_manager 34](#_Toc495393308)

[12.3 spi\_init 34](#_Toc495393309)

[12.4 spi\_deinit 34](#_Toc495393310)

[12.5 spi\_read 34](#_Toc495393311)

[12.6 spi\_write 35](#_Toc495393312)

[12.7 spi\_transfer 35](#_Toc495393313)

[13 usb 36](#_Toc495393314)

[13.1 源码文件 36](#_Toc495393315)

[13.2 配置参数 36](#_Toc495393316)

[13.3 get\_usb\_device\_manager 36](#_Toc495393317)

[13.4 usb\_device\_init 36](#_Toc495393318)

[13.5 usb\_device\_deinit 36](#_Toc495393319)

[13.6 usb\_device\_switch\_func 38](#_Toc495393320)

[13.7 usb\_device\_get\_max\_transfer\_unit 38](#_Toc495393321)

[13.8 usb\_device\_write 38](#_Toc495393322)

[13.9 usb\_device\_read 39](#_Toc495393323)

[14 Security 41](#_Toc495393324)

[14.1 源码文件 41](#_Toc495393325)

[14.2 get\_security\_manager 41](#_Toc495393326)

[14.3 security\_init 41](#_Toc495393327)

[14.4 security\_deinit 41](#_Toc495393328)

[14.5 simple\_aes\_load\_key 41](#_Toc495393329)

[14.6 simple\_aes\_crypt 42](#_Toc495393330)

[15 zigbee 43](#_Toc495393331)

[15.1 源码文件 43](#_Toc495393332)

[15.2 get\_zigbee\_manager 43](#_Toc495393333)

[15.3 init 43](#_Toc495393334)

[15.4 deinit 43](#_Toc495393335)

[15.5 reset 44](#_Toc495393336)

[15.6 ctrl 44](#_Toc495393337)

[15.7 get\_info 44](#_Toc495393338)

[15.8 factory 45](#_Toc495393339)

[15.9 reboot 45](#_Toc495393340)

[15.10 set\_role 45](#_Toc495393341)

[15.11 set\_panid 45](#_Toc495393342)

[15.12 set\_channel 46](#_Toc495393343)

[15.13 set\_key 46](#_Toc495393344)

[15.14 set\_join\_aging 46](#_Toc495393345)

[15.15 set\_cast\_type 47](#_Toc495393346)

[15.16 set\_group\_id 47](#_Toc495393347)

[15.17 set\_poll\_rate 47](#_Toc495393348)

[15.18 set\_tx\_power 48](#_Toc495393349)

[16 74hc595 49](#_Toc495393350)

[16.1 源码文件 49](#_Toc495393351)

[16.2 配置参数 49](#_Toc495393352)

[16.3 get\_sn74hc595\_manager 49](#_Toc495393353)

[16.4 sn74hc595\_init 49](#_Toc495393354)

[16.5 sn74hc595\_deinit 50](#_Toc495393355)

[16.6 sn74hc595\_get\_outbits 50](#_Toc495393356)

[16.7 sn74hc595\_write 50](#_Toc495393357)

[16.8 sn74hc595\_read 51](#_Toc495393358)

[16.9 sn74hc595\_clear 51](#_Toc495393359)

[17 cypress 52](#_Toc495393360)

[17.1 源码文件 52](#_Toc495393361)

[17.2 get\_cypress\_manager 52](#_Toc495393362)

[17.3 cypress\_init 52](#_Toc495393363)

[17.4 cypress\_deinit 52](#_Toc495393364)

[17.5 cypress\_ mcu\_reset 53](#_Toc495393365)

[18 fpc fingerprint 54](#_Toc495393366)

[18.1 源码文件 54](#_Toc495393367)

[18.2 fpc\_fingerprint\_init 54](#_Toc495393368)

[18.3 fpc\_fingerprint\_destroy 55](#_Toc495393369)

[18.4 fpc\_fingerprint\_reset 55](#_Toc495393370)

[18.5 fpc\_fingerprint\_enroll 55](#_Toc495393371)

[18.6 fpc\_fingerprint\_authenticate 55](#_Toc495393372)

[18.7 fpc\_fingerprint\_delete 55](#_Toc495393373)

[18.8 fpc\_fingerprint\_cancel 56](#_Toc495393374)

[18.9 fpc\_fingerprint\_get\_template\_info 56](#_Toc495393375)

[19 lock cylinder 57](#_Toc495393376)

[19.1 源码文件 57](#_Toc495393377)

[19.2 get\_lock\_cylinder\_manager 57](#_Toc495393378)

[19.3 lock\_cylinder\_init 57](#_Toc495393379)

[19.4 lock\_cylinder\_deinit 57](#_Toc495393380)

[19.5 lock\_cylinder\_get\_keystatue 58](#_Toc495393381)

[19.6 lock\_cylinder\_get\_romid 58](#_Toc495393382)

[19.7 lock\_cylinder\_register\_key 58](#_Toc495393383)

[19.8 lock\_cylinder\_authenticate\_key 59](#_Toc495393384)

[19.9 lock\_cylinder\_power\_ctrl 59](#_Toc495393385)

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