CSI-API(v1.7) User Manual

2021 年 03 月 25 日

平头哥半导体有限公司 T-HEAD Semiconductor Co.,LTD

地址: 杭州市余杭区向往街 1122 号欧美金融城 (EFC) 英国中心西楼 T6

邮编: 311121

网址: www.t-head.cn

版本历史

版本	描述	日期
v1.7	第一次正式发布	2021.03.24

CSI-API(v1.7) User Manual

第一章　CSI 总体

1.1　简介

这份手册描述的是 CSI API (CSI 应用编程接口)

分类为:

- *CSI-Core*
- *CSI-Driver*
- *CSI-Kernel*

1.2　如何使用

1.2.1　头文件包含

1.2.1.1　CSI-Core

直接包含 <csi_core.h>

1.2.1.2　CSI-Driver

例如使用 Timer 驱动，包含 <drv_timer.h>
例如使用 UART 驱动，包含 <drv_usart.h>
依此类推……

1.2.1.3　CSI-Kernel

直接包含 <csi_kernel.h>

第二章　CSI-Core API

2.1　编译控制

2.1.1　宏列表

- *__ASM*
- *__INLINE*
- *__STATIC_INLINE*
- *__ALWAYS_STATIC_INLINE*

2.1.2　简要说明

提供基本的宏定义。

2.1.3　接口描述

2.1.3.1　__ASM

```
#define __ASM
```

功能描述:

内嵌汇编关键字。

2.1.3.2　__INLINE

```
#define __INLINE
```

功能描述:

内联函数关键字。

2.1.3.3 ___STATIC_INLINE

```
#define __STATIC_INLINE
```

功能描述:

静态内联函数关键字。

2.1.3.4 ___ALWAYS_STATIC_INLINE

```
#define __ALWAYS_STATIC_INLINE
```

功能描述:

强制静态内联函数关键字。

2.1.4 示例:

```
__STATIC_INLINE uint32_t csi_coret_config(uint32_t ticks, int32_t IRQn)
{
    if ((ticks - 1UL) > CORET_LOAD_RELOAD_Msk) {
        return (1UL);                                   /* Reload value impossible */
    }

    CORET->LOAD = (uint32_t)(ticks - 1UL);              /* Set reload register */
    CORET->VAL  = 0UL;                                  /* Load the CORET Counter Value */
    CORET->CTRL = CORET_CTRL_CLKSOURCE_Msk |
                  CORET_CTRL_TICKINT_Msk |
                  CORET_CTRL_ENABLE_Msk;                /* Enable CORET IRQ and CORET Timer */
    return (0UL);                                       /* Function successful */
}
```

```
__ALWAYS_STATIC_INLINE uint32_t __get_PSR(void)
{
    uint32_t result;

    __ASM volatile("mfcr %0, psr" : "=r"(result));
    return (result);
}
```

2.2 VIC

2.2.1 函数列表

- *csi_vic_enable_irq*
- *csi_vic_disable_irq*
- *csi_vic_enable_sirq*
- *csi_vic_disable_sirq*
- *csi_vic_get_enabled_irq*
- *csi_vic_get_pending_irq*
- *csi_vic_set_pending_irq*
- *csi_vic_clear_pending_irq*
- *csi_vic_set_wakeup_irq*
- *csi_vic_get_wakeup_irq*
- *csi_vic_clear_wakeup_irq*
- *csi_vic_get_active*
- *csi_vic_set_threshold*
- *csi_vic_set_prio*
- *csi_vic_get_prio*
- *csi_vic_set_vector*
- *csi_vic_get_vector*

2.2.2 简要说明

提供紧耦合矢量中断控制器（VIC）的基本操作，包括中断的使能与禁止，中断优先级设置、中断状态设置与获取等。

2.2.3 接口描述

2.2.3.1 csi_vic_enable_irq

```
__STATIC_INLINE void csi_vic_enable_irq(int32_t IRQn)
```

功能描述:

使能中断。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.2 csi_vic_disable_irq

```
__STATIC_INLINE void csi_vic_disable_irq(int32_t IRQn)
```

功能描述:

禁止中断。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.3 csi_vic_enable_sirq

```
__STATIC_INLINE void csi_vic_enable_sirq(int32_t IRQn)
```

功能描述:

使能安全中断。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.4 csi_vic_disable_sirq

```
__STATIC_INLINE void csi_vic_disable_sirq(int32_t IRQn)
```

功能描述:

禁止安全中断。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.5 csi_vic_get_enabled_irq

```
__STATIC_INLINE uint32_t csi_vic_get_enabled_irq(int32_t IRQn)
```

功能描述:

判断中断是否使能。

参数:

IRQn: 中断号（非异常号）。

返回值:

1: 使能。
0: 未使能。

2.2.3.6 csi_vic_get_pending_irq

```
__STATIC_INLINE uint32_t csi_vic_get_pending_irq(int32_t IRQn)
```

功能描述:

判断中断是否处于等待状态。

参数:

IRQn: 中断号（非异常号）。

返回值:

1: 等待状态。
0: 非等待状态。

2.2.3.7 csi_vic_set_pending_irq

```
__STATIC_INLINE void csi_vic_set_pending_irq(int32_t IRQn)
```

功能描述:

设置中断进入等待状态。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.8 csi_vic_clear_pending_irq

```
__STATIC_INLINE void csi_vic_clear_pending_irq(int32_t IRQn)
```

功能描述:

清除中断的等待状态。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.9 csi_vic_set_wakeup_irq

```
__STATIC_INLINE void csi_vic_set_wakeup_irq(int32_t IRQn)
```

功能描述:

设置中断为唤醒中断。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.10 csi_vic_get_wakeup_irq

```
__STATIC_INLINE uint32_t csi_vic_get_wakeup_irq(int32_t IRQn)
```

功能描述:

判断中断是否为唤醒中断。

参数:

IRQn: 中断号（非异常号）。

返回值:

1: 唤醒中断。
0: 非唤醒中断。

2.2.3.11 csi_vic_clear_wakeup_irq

```
__STATIC_INLINE void csi_vic_clear_wakeup_irq(int32_t IRQn)
```

功能描述:

清除中断的唤醒功能描述。

参数:

IRQn: 中断号（非异常号）。

返回值:

无。

2.2.3.12 csi_vic_get_active

```
__STATIC_INLINE uint32_t csi_vic_get_active(int32_t IRQn)
```

功能描述:

判断中断是否正在被响应。

参数:

IRQn: 中断号（非异常号）。

返回值:

1: 响应状态。

0: 非响应状态。

2.2.3.13 csi_vic_set_threshold

```
__STATIC_INLINE void csi_vic_set_threshold(uint32_t VectThreshold, uint32_t PrioThreshold)
```

功能描述:

设置 VIC 阈值。

参数:

`VectThreshold`: 指示优先级阈值对应的中断向量号（注：当 VIC 发现 CPU 从 VECTTHRESHOLD 对应的中断服务程序退出时，会硬件清除中断优先级阈值有效位)。

`PrioThreshold`: 指示中断抢占的优先级阈值。

返回值:

无。

2.2.3.14 csi_vic_set_prio

```
__STATIC_INLINE void csi_vic_set_prio(int32_t IRQn, uint32_t priority)
```

功能描述:

设置中断优先级。

参数:

`IRQn`: 中断号（非异常号)。

`priority`: 中断优先级 (0~3), 数值越小优先级越高。

返回值:

无。

2.2.3.15 csi_vic_get_prio

```
__STATIC_INLINE uint32_t csi_vic_get_prio(int32_t IRQn)
```

功能描述:

获取中断优先级。

参数:

IRQn: 中断号（非异常号）。

返回值:

获取中断优先级。

2.2.3.16 csi_vic_set_vector

```
__STATIC_INLINE void csi_vic_set_vector(int32_t IRQn, uint32_t handler)
```

功能描述:

设置中断处理函数。

参数:

IRQn: 中断号（非异常号）。
handler: 中断处理函数。

返回值:

无。

2.2.3.17 csi_vic_get_vector

```
__STATIC_INLINE uint32_t csi_vic_get_vector(int32_t IRQn)
```

功能描述:

获取中断处理函数。

参数:

IRQn: 中断号（非异常号）。

返回值:

中断处理函数。

2.2.4 示例:

```
/* 使能 0 号中断 */
csi_vic_enable_irq(0);

/* 设置 0 号中断优先级为 3 */
csi_vic_set_prio(0, 3);
```

2.3 CoreTIM

2.3.1 函数列表

- *csi_coret_config*
- *csi_coret_get_load*
- *csi_coret_get_value*

2.3.2 简要说明

提供紧耦合系统定时器的基本操作，包括定时器的配置，定时器计数值的获取等。

2.3.3 接口描述

2.3.3.1 csi_coret_config

```
__STATIC_INLINE uint32_t csi_coret_config(uint32_t ticks, int32_t IRQn)
```

功能描述:

配置 CoreTIM。

参数:

`ticks`: 单次计时的时钟数。
`IRQn`: 中断号（非异常号）。

返回值:

返回 0。

2.3.3.2 csi_coret_get_load

```
__STATIC_INLINE uint32_t csi_coret_get_load(void)
```

功能描述:

获取 CoreTIM 的加载值。

参数:

无。

返回值:

CoreTIM 加载值。

2.3.3.3 csi_coret_get_value

```
__STATIC_INLINE uint32_t csi_coret_get_value(void)
```

功能描述:

获取 CoreTIM 当前计数值。

参数:

无。

返回值:

CoreTIM 当前计数值。

2.3.4 示例:

```
uint32_t load, cur;

/* 使能 CoreTIM, 并设置 CoreTIM 的 load 值为最大 */
csi_coret_config(0xFFFFFFFF, CORET_IRQn);

load = csi_coret_get_load();
cur  = csi_coret_get_value();

/* 计算 CoreTIM 走过的时钟数 */
printf("spent clocks: %u\n", load - cur);
```

2.4 Cache

2.4.1 函数列表

- *csi_icache_enable*
- *csi_icache_disable*
- *csi_icache_invalid*
- *csi_dcache_enable*
- *csi_dcache_disable*
- *csi_dcache_invalid*
- *csi_dcache_clean*
- *csi_dcache_clean_invalid*
- *csi_dcache_invalid_range*
- *csi_dcache_clean_range*
- *csi_dcache_clean_invalid_range*
- *csi_cache_set_range*
- *csi_cache_enable_profile*
- *csi_cache_disable_profile*
- *csi_cache_reset_profile*
- *csi_cache_get_access_time*
- *csi_cache_get_miss_time*

2.4.2 简要说明

提供高速缓存器（Cache）的基本操作，包括 Cache 的刷新和 Cache 的 profiling 功能。

2.4.3 接口描述

2.4.3.1 csi_icache_enable

```
__STATIC_INLINE void csi_icache_enable (void)
```

功能描述:

使能指令高速缓存。

参数:

无。

返回值:

无。

2.4.3.2 csi_icache_disable

```
__STATIC_INLINE void csi_icache_disable (void)
```

功能描述:

禁止指令高速缓存。

参数:

无。

返回值:

无。

2.4.3.3 csi_icache_invalid

```
__STATIC_INLINE void csi_icache_invalid (void)
```

功能描述:

失效指令高速缓存。

参数:

无。

返回值:

无。

2.4.3.4 csi_dcache_enable

```
__STATIC_INLINE void csi_dcache_enable (void)
```

功能描述:

使能数据高速缓存。

参数:

无。

返回值:

无。

2.4.3.5 csi_dcache_disable

```
__STATIC_INLINE void csi_dcache_disable (void)
```

功能描述:

禁止数据高速缓存。

参数:

无。

返回值:

无。

2.4.3.6 csi_dcache_invalid

```
__STATIC_INLINE void csi_dcache_invalid (void)
```

功能描述:

失效整个数据高速缓存。

参数:

无。

返回值:

无。

2.4.3.7 csi_dcache_clean

```
__STATIC_INLINE void csi_dcache_clean (void)
```

功能描述:

清除整个数据高速缓存。

参数:

无。

返回值:

无。

2.4.3.8 csi_dcache_clean_invalid

```
__STATIC_INLINE void csi_dcache_clean_invalid (void)
```

功能描述:

清除和失效整个数据高速缓存。

参数:

无。

返回值:

无。

2.4.3.9 csi_dcache_invalid_range

```
__STATIC_INLINE void csi_dcache_invalid_range (uint32_t *addr, int32_t dsize)
```

功能描述:

按行失效数据高速缓存。

参数:

addr: 需要被失效的地址。
dsize: 地址长度。

返回值:

无。

2.4.3.10 csi_dcache_clean_range

```
__STATIC_INLINE void csi_dcache_clean_range (uint32_t *addr, int32_t dsize)
```

功能描述:

按行清除数据高速缓存。

参数:

addr: 需要被清除的地址。
dsize: 地址长度。

返回值:

无。

2.4.3.11 csi_dcache_clean_invalid_range

```
__STATIC_INLINE void csi_dcache_clean_invalid_range (uint32_t *addr, int32_t dsize)
```

功能描述:

按行清楚和失效数据高速缓存。

参数:

addr: 需要被清除和失效的地址。

dsize: 地址长度。

返回值:

无。

2.4.3.12 csi_cache_set_range

```
__STATIC_INLINE void csi_cache_set_range (uint32_t index, uint32_t baseAddr, uint32_t size, uint32_
↪t enable)
```

功能描述:

设置高速缓存的作用范围。

参数:

index: 高速缓存区号。

baseAddr: 作用地址。

size: 地址长度。

enable: 是否使能该区域。1: 使能；0: 禁止。

返回值:

无。

2.4.3.13 csi_cache_enable_profile

```
__STATIC_INLINE void csi_cache_enable_profile (void)
```

功能描述:

使能高速缓存的分析功能描述。

参数:

无。

返回值:

无。

2.4.3.14 csi_cache_disable_profile

```
__STATIC_INLINE void csi_cache_disable_profile (void)
```

功能描述:

禁止高速缓存的分析功能描述。

参数:

无。

返回值:

无。

2.4.3.15 csi_cache_reset_profile

```
__STATIC_INLINE void csi_cache_reset_profile (void)
```

功能描述:

复位高速缓存的分析功能。

参数:

无。

返回值:

无。

2.4.3.16 csi_cache_get_access_time

```
__STATIC_INLINE uint32_t csi_cache_get_access_time (void)
```

功能描述:

获取高速缓存的访问次数。

参数:

无。

返回值:

高速缓存访问的次数，每 256 次增加 1 个计数值。

2.4.3.17 csi_cache_get_miss_time

```
__STATIC_INLINE uint32_t csi_cache_get_miss_time (void)
```

功能描述:

获取高速缓存的未命中次数。

参数:

无。

返回值:

高速缓存未命中的次数，每 256 次增加 1 个计数值。

2.4.4 示例:

紧耦合 cache 的示例:

```
uint32_t access, miss;

/* 设置 cache 作用域 0 的范围为: 0x0-0x1000 */
csi_cache_set_range(0, 0x0, CACHE_CRCR_4K, 1);
/* 设置 cache 作用域 1 的范围为: 0x10000000-0x10080000 */
csi_cache_set_range(1, 0x10000000, CACHE_CRCR_512K, 1);

/* 使能指令 cache 和数据 cache */
```

(下页继续)

(续上页)

```
csi_icache_enable();
csi_dcache_enable();

/* 开启并重置 cache 的 profile 功能 */
csi_cache_enable_profile();
csi_cache_reset_profile();

/* 获取 cache access 和 miss 的次数 */
access = csi_cache_get_access_time();
miss   = csi_cache_get_miss_time();

printf("cache access times: %u\n", access * 256);
printf("cache miss times: %u\n", miss * 256);
```

2.5 MPU

2.5.1 函数列表

- *csi_mpu_enable*
- *csi_mpu_disable*
- *csi_mpu_config_region*
- *csi_mpu_enable_region*
- *csi_mpu_disable_region*

2.5.2 简要说明

提供 MPU 模块的基本操作，该部分接口可在有 MPU 的 CPU 上使用。

2.5.3 接口描述

2.5.3.1 csi_mpu_enable

```
__STATIC_INLINE void csi_mpu_enable(void)
```

功能描述:

使能 MPU。

参数:

无。

返回值:

无。

2.5.3.2 csi_mpu_disable

```
__STATIC_INLINE void csi_mpu_disable(void)
```

功能描述:

禁止 MPU。

参数:

无。

返回值:

无。

2.5.3.3 csi_mpu_config_region

```
__STATIC_INLINE void csi_mpu_config_region(uint32_t idx, uint32_t base_addr, region_size_e size,
                                           mpu_region_attr_t attr, uint32_t enable)
```

功能描述:

配置 MPU 的区域。

参数:

idx: MPU 区域号。

base_addr: 作用基地址。

size: 地址长度。定义参见表格*region_size_e* 。

attr: 属性。

enable: 是否使能。

返回值:

无。

region_size_e

名字	定义	备注
REGION_SIZE_128B	保护区大小为 128B	仅支持 801/802
REGION_SIZE_256B	保护区大小为 256B	仅支持 801/802
REGION_SIZE_512B	保护区大小为 512B	仅支持 801/802
REGION_SIZE_1KB	保护区大小为 1KB	仅支持 801/802
REGION_SIZE_2KB	保护区大小为 2KB	仅支持 801/802
REGION_SIZE_4KB	保护区大小为 4KB	
REGION_SIZE_8KB	保护区大小为 8KB	
REGION_SIZE_16KB	保护区大小为 16KB	
REGION_SIZE_32KB	保护区大小为 32KB	
REGION_SIZE_64KB	保护区大小为 64KB	
REGION_SIZE_128KB	保护区大小为 128KB	
REGION_SIZE_256KB	保护区大小为 256KB	
REGION_SIZE_512KB	保护区大小为 512KB	
REGION_SIZE_1MB	保护区大小为 1MB	
REGION_SIZE_2MB	保护区大小为 2MB	
REGION_SIZE_4MB	保护区大小为 4MB	
REGION_SIZE_8MB	保护区大小为 8MB	
REGION_SIZE_16MB	保护区大小为 16MB	
REGION_SIZE_32MB	保护区大小为 32MB	
REGION_SIZE_64MB	保护区大小为 64MB	
REGION_SIZE_128MB	保护区大小为 128MB	
REGION_SIZE_256MB	保护区大小为 256MB	
REGION_SIZE_512MB	保护区大小为 512MB	
REGION_SIZE_1GB	保护区大小为 1GB	
REGION_SIZE_2GB	保护区大小为 2GB	
REGION_SIZE_4GB	保护区大小为 4GB	

mpu_region_attr_t

名字	描述	定义
nx	执行权限	CK610 不支持 nx 属性 • 0: 可执行 • 1: 不可执行
ap	读写权限	access_permission_e: • AP_BOTH_INACCESSIBLE: 超级用户和普通用户都不可访问 • AP_SUPER_RW_USER_INACCESSIBLE: 超级用户可读写，普通用户不可访问 • AP_SUPER_RW_USER_RDONLY: 超级用户可读写，普通用户只读 • AP_BOTH_RW: 超级用户和普通用户都可读写
s	安全属性	• 0: 非安全 • 1: 安全

2.5.3.4 csi_mpu_enable_region

```
__STATIC_INLINE void csi_mpu_enable_region(uint32_t idx)
```

功能描述:

使能 MPU 某个区域。

参数:

`idx`: MPU 区域号。

返回值:

无。

2.5.3.5 csi_mpu_disable_region

```
__STATIC_INLINE void csi_mpu_disable_region(uint32_t idx)
```

功能描述:

禁止 MPU 某个区域。

参数:

idx: MPU 区域号。

返回值:

无。

2.5.4 示例:

```
mpu_region_attr_t attr;

attr.nx = 0;
attr.ap = AP_BOTH_RW;
attr.s = 0;

/* 配置 MPU 作用域 0 的范围为: 0x0-0x1000, 并使能该作用域 */
csi_mpu_config_region(0, 0x0, REGION_SIZE_4KB, attr, 1);

/* 禁止 MPU 作用域 0 */
csi_mpu_disable_region(0);
```

2.6 HAD

2.6.1 函数列表

- *csi_had_send_char*
- *csi_had_receive_char*
- *csi_had_check_char*

2.6.2 简要说明

提供 HAD（Hardware Assisted Debug）模块的基本操作，包括发送与接收数据等功能。

2.6.3 接口描述

2.6.3.1 csi_had_send_char

```
__STATIC_INLINE uint32_t csi_had_send_char(uint32_t ch)
```

功能描述:

通过 HAD 发送一个字符。

参数:

ch: 发送的字符。

返回值:

无。

2.6.3.2 csi_had_receive_char

```
__STATIC_INLINE int32_t csi_had_receive_char(void)
```

功能描述:

通过 HAD 接收一个字符。

参数:

无。

返回值:

接收到的字符。

2.6.3.3 csi_had_check_char

```
__STATIC_INLINE int32_t csi_had_check_char(void)
```

功能描述:

判断是否有数据可接收。

参数:

无。

返回值:

1：有数据；0：没有数据。

2.7 IRQ

2.7.1 函数列表

- *__enable_irq*
- *__disable_irq*
- *__enable_excp_irq*
- *__disable_excp_irq*
- *csi_irq_save*
- *csi_irq_restore*

2.7.2 简要说明

提供中断相关接口。

2.7.3 接口描述

2.7.3.1 __enable_irq

```
__ALWAYS_STATIC_INLINE void __enable_irq(void)
```

功能描述:

使能 CPU 中断。

参数:

无。

返回值:

无。

2.7.3.2 __disable_irq

```
__ALWAYS_STATIC_INLINE void __disable_irq(void)
```

功能描述:

禁止 CPU 中断。

参数:

无。

返回值:

无。

2.7.3.3 __enable_excp_irq

```
__ALWAYS_STATIC_INLINE void __enable_excp_irq(void)
```

功能描述:

使能 CPU 异常和中断。

参数:

无。

返回值:

无。

2.7.3.4 __disable_excp_irq

```
__ALWAYS_STATIC_INLINE void __disable_excp_irq(void)
```

功能描述:

禁止 CPU 异常和中断。

参数:

无。

返回值:

无。

2.7.3.5 csi_irq_save

```
__STATIC_INLINE uint32_t csi_irq_save(void)
```

功能描述:

保存处理器状态寄存器值，并禁止 CPU 中断。

参数:

无。

返回值:

PSR 状态值。

2.7.3.6 csi_irq_restore

```
__STATIC_INLINE void csi_irq_restore(uint32_t irq_state)
```

功能描述:

恢复处理器状态寄存器值。

参数:

`irq_state`: PSR 状态值。

返回值:

无。

2.7.4 示例:

```
uint32_t flags;

flags = csi_irq_save();

/* ... */

csi_irq_restore(flags);
```

2.8 MMU

2.8.1 函数列表

- *csi_mmu_enable*
- *csi_mmu_disable*
- *csi_mmu_set_tlb*
- *csi_mmu_set_pagesize*
- *csi_mmu_read_by_index*
- *csi_mmu_invalid_tlb_all*
- *csi_mmu_invalid_tlb_by_index*
- *csi_mmu_invalid_tlb_by_vaddr*

2.8.2 简要说明

提供 MMU（Memory Management Unit）模块的基本操作，该部分接口可在有 MMU 的 CPU 上使用。

2.8.3 接口描述

2.8.3.1 csi_mmu_enable

```
__STATIC_INLINE void csi_mmu_enable(void)
```

功能描述:

使能 MMU。

参数:

无。

返回值:

无。

2.8.3.2 csi_mmu_disable

```
__STATIC_INLINE void csi_mmu_disable(void)
```

功能描述:

禁止 MMU。

参数:

无。

返回值:

无。

2.8.3.3 csi_mmu_set_tlb

```
__STATIC_INLINE void csi_mmu_set_tlb(uint32_t vaddr, uint32_t paddr, uint32_t asid, page_attr_t␣
↪attr)
```

功能描述:

设置 tlb 表项映射。

参数:

vaddr: 虚拟地址页。
paddr: 物理地址页。
asid: ASID 号。
attr: 读写属性，见*page_attr_t* 定义。

返回值:

无。

page_attr_t:

属性	描述	定义
global	全局属性	• 0: 当前 ASID 有效 • 1: 全局有效
valid	合法属性	• 0: 表项无效 • 1: 表项有效
writeable	可写属性	• 0: 页面可写 • 1: 页面不可写
cacheable	可高缓属性	• 0: 页面不可高缓 • 1： 页面可高缓
is_secure	安全属性	• 0: 页面支持安全访问 • 1: 页面不支持安全访问
strong_order	读写访问顺序属性	• 0: 总线上内存的读写访问顺序和程序流的读写访问顺序不同 • 1: 该页总线上内存的读写访问顺序和程序流的读写访问顺序相同
bufferable	可 Buffer 属性	• 0: 页面可 Buffer • 1: 页面不可 Buffer

2.8.3.4 csi_mmu_set_pagesize

```
__STATIC_INLINE void csi_mmu_set_pagesize(page_size_e size)
```

功能描述:

设置映射页面的大小。

参数:

size: 映射页面的大小，见*page_size_e* 定义。

返回值:

无。

page_size_e:

名字	定义	备注
PAGE_SIZE_4KB	页面大小为 4KB	
PAGE_SIZE_16KB	页面大小为 16KB	
PAGE_SIZE_64KB	页面大小为 64KB	
PAGE_SIZE_256KB	页面大小为 256KB	
PAGE_SIZE_1MB	页面大小为 1MB	
PAGE_SIZE_4MB	页面大小为 4MB	
PAGE_SIZE_4MB	页面大小为 16MB	

2.8.3.5 csi_mmu_read_by_index

```
__STATIC_INLINE void csi_mmu_read_by_index(uint32_t index, uint32_t *meh, uint32_t *mel0, uint32_t␣
↪*mel1)
```

功能描述:

通过 index 读取 TLB 表项。

参数:

index: TLB 索引号。
meh: 存储返回的 MEH 寄存器值。
mel0: 存储返回的 MEL0 寄存器值。
mel1: 存储返回的 MEL1 寄存器值。

返回值:

无。

2.8.3.6 csi_mmu_invalid_tlb_all

```
__STATIC_INLINE void csi_mmu_invalid_tlb_all(void)
```

功能描述:

失效所有 TLB 表项。

参数:

无。

返回值:

无。

2.8.3.7 csi_mmu_invalid_tlb_by_index

```
__STATIC_INLINE void csi_mmu_invalid_tlb_by_index(uint32_t index)
```

功能描述:

按 index 失效 TLB 表项。

参数:

index: TLB index 号。

返回值:

无。

2.8.3.8 csi_mmu_invalid_tlb_by_vaddr

```
__STATIC_INLINE void csi_mmu_invalid_tlb_by_vaddr(uint32_t vaddr, uint32_t asid)
```

功能描述:

通过虚拟地址失效 TLB 表项。

参数:

vaddr: 需失效的虚拟地址。
asid: 虚拟地址所属 ASID 号。

返回值:

无。

2.8.4 示例:

```
/* 创建一个 4KB 大小的页面: 虚拟地址 0x0 映射到物理地址 0x0 */
page_attr_t attr;

attr.global = 1;
attr.valid = 1;
attr.writeable = 1;
attr.cacheable = 1;
attr.is_secure = 0;
attr.strong_order = 0;
```

(下页继续)

(续上页)

```
attr.bufferable = 0;

csi_mmu_set_pagesize(PAGE_SIZE_4KB);
csi_mmu_set_tlb(0x0, 0x0, 0, attr);
```

2.9 TCM

2.9.1 函数列表

- *csi_itcm_enable*
- *csi_dtcm_enable*
- *csi_itcm_disable*
- *csi_dtcm_disable*
- *csi_itcm_slave_access_enable*
- *csi_itcm_slave_access_disable*
- *csi_dtcm_slave_access_enable*
- *csi_dtcm_slave_access_disable*
- *csi_itcm_get_size*
- *csi_dtcm_get_size*
- *csi_itcm_get_delay*
- *csi_dtcm_get_delay*
- *csi_itcm_set_base_addr*
- *csi_dtcm_set_base_addr*

2.9.2 简要说明

提供设置 TCM 的接口。

2.9.3 接口描述

2.9.3.1 csi_itcm_enable

```
__STATIC_INLINE void csi_itcm_enable(void)
```

功能描述:

使能 ITCM。

参数:

无。

返回值:

无。

2.9.3.2 csi_dtcm_enable

```
__STATIC_INLINE void csi_dtcm_enable (void)
```

功能描述:

使能 DTCM。

参数:

无。

返回值:

无。

2.9.3.3 csi_itcm_disable

```
__STATIC_INLINE void csi_itcm_disable (void)
```

功能描述:

禁止 ITCM。

参数:

无。

返回值:

无。

2.9.3.4 csi_dtcm_disable

```
__STATIC_INLINE void csi_dtcm_disable (void)
```

功能描述:

禁止 DTCM。

参数:

无。

返回值:

无。

2.9.3.5 csi_itcm_slave_access_enable

```
__STATIC_INLINE void csi_itcm_slave_access_enable(void)
```

功能描述:

使能 ITCM 的 slave 访问。

参数:

无。

返回值:

无。

2.9.3.6 csi_itcm_slave_access_disable

```
__STATIC_INLINE void csi_itcm_slave_access_disable(void)
```

功能描述:

禁止 ITCM 的 slave 访问。

参数:

无。

返回值:

无。

2.9.3.7 csi_dtcm_slave_access_enable

```
__STATIC_INLINE void csi_dtcm_slave_access_enable(void)
```

功能描述:

使能 DTCM 的 slave 访问。

参数:

无。

返回值:

无。

2.9.3.8 csi_dtcm_slave_access_disable

```
__STATIC_INLINE void csi_dtcm_slave_access_disable(void)
```

功能描述:

禁止 DTCM 的 slave 访问。

参数:

无。

返回值:

无。

2.9.3.9 csi_itcm_get_size

```
__STATIC_INLINE uint32_t csi_itcm_get_size(void)
```

功能描述:

获取 ITCM 的作用大小。

参数:

无。

返回值:

无。

2.9.3.10 csi_dtcm_get_size

```
__STATIC_INLINE uint32_t csi_dtcm_get_size(void)
```

功能描述:

获取 DTCM 的作用大小。

参数:

无。

返回值:

无。

2.9.3.11 csi_itcm_get_delay

```
__STATIC_INLINE uint32_t csi_itcm_get_delay(void)
```

功能描述:

获取 ITCM 的访问延时。

参数:

无。

返回值:

访问延时。

2.9.3.12 csi_dtcm_get_delay

```
__STATIC_INLINE uint32_t csi_dtcm_get_delay(void)
```

功能描述:

获取 DTCM 的访问延时。

参数:

无。

返回值:

访问延时。

2.9.3.13 csi_itcm_set_base_addr

```
__STATIC_INLINE void csi_itcm_set_base_addr(uint32_t base_addr)
```

功能描述:

设置 ITCM 作用的基地址。

参数:

基地址。

返回值:

无。

2.9.3.14 csi_dtcm_set_base_addr

```
__STATIC_INLINE void csi_dtcm_set_base_addr(uint32_t base_addr)
```

功能描述:

设置 DTCM 作用的基地址。

参数:

基地址。

返回值:

无。

2.9.4 示例:

```
/* set dtcm */

csi_dtcm_set_base_addr(0xF0000000);
csi_dtcm_enable();

memcpy((void *)0xF0000000, (void *)0x40000, 256 * 1024);

csi_dtcm_set_base_addr(0x40000);
```

2.10 ECC

2.10.1 函数列表

- *csi_ecc_enable*
- *csi_ecc_disable*
- *csi_ecc_enable_error_fix*
- *csi_ecc_disable_error_fix*
- *csi_ecc_inject_error*
- *csi_ecc_get_error_info*

2.10.2 简要说明

提供设置 ECC 的接口。

2.10.3 接口描述

2.10.3.1 csi_ecc_enable

```
__STATIC_INLINE void csi_ecc_enable(void);
```

功能描述:

使能 ECC 功能。

参数:

无。

返回值:

无。

2.10.3.2 csi_ecc_disable

```
__STATIC_INLINE void csi_ecc_disable(void);
```

功能描述:

禁止 ECC 功能。

参数:

无。

返回值:

无。

2.10.3.3 csi_ecc_enable_error_fix

```
__STATIC_INLINE void csi_ecc_enable_error_fix(void)
```

功能描述:

使能 ECC 的错误修复功能。

参数:

无。

返回值:

无。

2.10.3.4 csi_ecc_disable_error_fix

```
__STATIC_INLINE void csi_ecc_disable_error_fix(void)
```

功能描述:

禁止 ECC 的错误修复功能。

参数:

无。

返回值:

无。

2.10.3.5 csi_ecc_inject_error

```
__STATIC_INLINE void csi_ecc_inject_error(ecc_error_type_e type, ecc_ramid_e ramid)
```

功能描述:

ECC 注入错误。

参数:

type: ECC 错误类型，见*ecc_error_type_e* 定义。
ramid: ECC RAM 索引，见*ecc_ramid_e* 定义。

返回值:

无。

ecc_error_type_e:

类型	描述	备注
ECC_ERROR_CORRECTABLE	可修复的错误	
ECC_ERROR_FATAL	不可修复的错误	

ecc_ramid_e:

类型	描述	备注
ECC_ICACHE_TAG_RAM	指令 cache 标签内存	
ECC_ICACHE_DATA_RAM	指令 cache 数据内存	
ECC_DCACHE_TAG_RAM	数据 cache 标签内存	
ECC_DCACHE_DATA_RAM	数据 cache 数据内存	
ECC_ITCM_RAM	ITCM 内存	
ECC_DTCM_RAM	DTCM 内存	

2.10.3.6 csi_ecc_get_error_info

```
__STATIC_INLINE void csi_ecc_get_error_info(ecc_error_info_t *info)
```

功能描述:

获取 ECC 错误信息。

参数:

info: ECC 错误信息，见*ecc_error_info_t* 定义。

返回值:

无。

ecc_error_info_t:

类型	描述	备注
uint32_t erraddr	错误地址	
uint32_t index	RAM index 索引位: ICACHE DATA：从 addr[3] 开始 DACHE DATA：从 addr[2] 开始 ICACHE TAG/DCACHE TAG：从 addr[5] 开始 TCM：从 addr[3] 开始	
uint8_t way	ICACHE/DCACHE WAY 索引位	
ecc_ramid_e ramid:8	ECC RAM 索引	
ecc_error_type_e ecc_type:8	ECC 错误类型	

2.11 Other

2.11.1 函数列表

- *csi_system_reset*
- *__get_REGISTER*
- *__set_REGISTER*

2.11.2 简要说明

提供 CPU 控制寄存器和状态寄存器的读写操作功能。

2.11.3 接口描述

2.11.3.1 csi_system_reset

```
__STATIC_INLINE void csi_system_reset(void)
```

功能描述:

复位 CPU。

参数:

无。

返回值:

无。

2.11.3.2 ___get__REGISTER

```
__ALWAYS_STATIC_INLINE __get_REGISTER(void)
```

功能描述:

获取寄存器的值。

参数:

无。

返回值:

寄存器值。

2.11.3.3 ___set__REGISTER

```
__ALWAYS_STATIC_INLINE void __set_REGISTER(uint32_t val)
```

功能描述:

设置寄存器的值。

参数:

`val`: 寄存器值。

返回值:

无。

___set__REGISTER 和 ___get__REGISTER 支持的寄存器

2.11.3.4 CSKY ARCH REGISTERS:

寄存器名	支持的 CPU	备注
PSR	All	
SP	All	
Int_SP	802-805	
VBR	All	
EPC	All	
EPSR	All	
CPUID	All	Read Only
CCR	All	
CCR2	807/810	
ERRLC	807	
ERRADDR	807	
ERRSTS	807	
ERRINJCR	807	
ERRINJCNT	807	
CINDEX	807	
CDATA0	807	
CDATA1	807	
CDATA2	807	
CINS	807	
DCSR	807/810	
CFR	807/810/610M	
CIR	807/810/610/610M	
CAPR	801-805/610/807	
CAPR1	807	
PACR	801-805/610/807	
PRSR	801-805/610/807	
ATTR0	807	
ATTR1	807	
UR14	801-805/807/810	
CHR	801-805	
HINT	807/810	
MIR	807/810/610M	
MEL0	807/810/610M	
MEL1	807/810/610M	
MEH	807/810/610M	
MPR	807/810/610M	
MCIR	807/810/610M	
MPGD	807/810/610M	
MAS0	807/810/610M	
MAS1	807/810/610M	
GSR	807/810/610/610M	
GCR	807/810/610/610M	

下页继续

表 2.1 – 续上页

寄存器名	支持的 CPU	备注
WSSR	TEE CPU	
WRCR	TEE CPU	
DCR	TEE CPU	
PCR	TEE CPU	
EBR	TEE CPU	

2.11.3.5 RISCV ARCH REGISTERS:

寄存器名	支持的 RISCV CPU	备注
MSTATUS	All	
MISA	All	
MIE	All	
MTVEC	All	
MTVT	All	
SP	All	
MSCRATCH	All	
MEPC	All	
MCAUSE	All	Read Only
MNXTI	All	
MINTSTATUS	All	Read Only
MTVAL	All	Read Only
MIP	All	
MCYCLE	All	Read Only
MCYCLEH	All	Read Only
MINSTRET	All	Read Only
MINSTRETH	All	Read Only
MVENDORID	All	Read Only
MARCHID	All	Read Only
MIMPID	All	Read Only
MHARTID	All	Read Only
PMPCFG0	All	
PMPCFG1	All	
PMPCFG2	All	
PMPCFG3	All	
PMPCFG0	All	
PMPCFG1	All	
PMPCFG2	All	
PMPCFG3	All	
PMPADDR0	All	

下页继续

表 2.2 – 续上页

寄存器名	支持的 RISCV CPU	备注
PMPADDR1	All	
PMPADDR2	All	
PMPADDR3	All	
PMPADDR4	All	
PMPADDR5	All	
PMPADDR6	All	
PMPADDR7	All	
PMPADDR8	All	
PMPADDR9	All	
PMPADDR10	All	
PMPADDR11	All	
PMPADDR12	All	
PMPADDR13	All	
PMPADDR14	All	
PMPADDR15	All	

/*

- Copyright (C) 2017-2019 Alibaba Group Holding Limited

*/

第三章　CSI-Driver API

3.1　Timer

3.1.1　函数列表

- *csi_timer_initialize*
- *csi_timer_uninitialize*
- *csi_timer_power_control*
- *csi_timer_config*
- *csi_timer_set_timeout*
- *csi_timer_start*
- *csi_timer_stop*
- *csi_timer_suspend*
- *csi_timer_resume*
- *csi_timer_get_current_value*
- *csi_timer_get_status*
- *csi_timer_get_load_value*

3.1.2　简要说明

Timer 硬件定时器是由时钟驱动的计数器，可用来实现计时或定时产生中断的功能。计数器可以是递增或递减，每个时钟让计数器增 1 或减 1。定时器支持自由运行和重载（reload）模式。其区别在于重载模式当计数器计数到指定值时会重新加载一个预设的值到计数器中，并可触发中断。

3.1.3　接口描述

3.1.3.1　csi_timer_initialize

```
timer_handle_t csi_timer_initialize(int32_t idx, timer_event_cb_t cb_event)
```

功能描述:

通过索引号初始化对应的 Timer 实例，返回 timer 实例的句柄。

参数:

idx: 控制器号。
cb_event: 中断回调函数。

返回值:

成功返回实例句柄，失败返回 NULL。

3.1.3.2 csi_timer_uninitialize

```
int32_t csi_timer_uninitialize(timer_handle_t handle)
```

功能描述:

timer 实例反初始化，该接口会停止 timer 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。实例句柄。

返回值:

错误码。

3.1.3.3 csi_timer_power_control

```
int32_t csi_timer_power_control(timer_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。
state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.1.3.4 csi_timer_config

```
int32_t csi_timer_config(timer_handle_t handle, timer_mode_e mode)
```

功能描述:

配置 Timer 实例的工作模式。

参数:

`handle`: 实例句柄。

`mode`: Timer 的工作模式，参考 timer_mode_e 的定义。

返回值:

错误码。

timer_mode_e:

名字	定义	备注
TIMER_MODE_FREE_RUNNING	timer 自由运行	
TIMER_MODE_RELOAD	timer 重复定时运行	

3.1.3.5 csi_timer_set_timeout

```
int32_t csi_timer_set_timeout(timer_handle_t handle, uint32_t timeout)
```

功能描述:

Timer 超时设置。

参数:

`handle`: 实例句柄。

`timeout`: Timer 超时时间，单位微秒（us）。

返回值:

错误码。

3.1.3.6 csi_timer_start

```
int32_t csi_timer_start(timer_handle_t handle)
```

功能描述:

Timer 计时启动。

参数:

handle: 实例句柄。

返回值:

错误码。

3.1.3.7 csi_timer_stop

```
int32_t csi_timer_stop(timer_handle_t handle)
```

功能描述:

Timer 计时停止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.1.3.8 csi_timer_suspend

```
int32_t csi_timer_suspend(timer_handle_t handle)
```

功能描述:

Timer 计时暂停。

参数:

handle: 实例句柄。

返回值:

错误码。

3.1.3.9 csi_timer_resume

```
int32_t csi_timer_resume(timer_handle_t handle)
```

功能描述:

Timer 计时恢复。

参数:

handle: 实例句柄。

返回值:

错误码。

3.1.3.10 csi_timer_get_current_value

```
int32_t csi_timer_get_current_value(timer_handle_t handle, uint32_t *value)
```

功能描述:

获取 Timer 当前计时值。

参数:

handle: 实例句柄。
value: 用来储存 Timer 当前计时值。

返回值:

错误码。

3.1.3.11 csi_timer_get_status

```
timer_status_t csi_timer_get_status(timer_handle_t handle)
```

功能描述:

获取 Timer 的状态。

参数:

handle: 实例句柄。

返回值:

Timer 状态。

3.1.3.12 csi_timer_get_load_value

```
int32_t csi_timer_get_load_value(timer_handle_t handle, uint32_t *value)
```

功能描述:

获取 Timer 的 Load 寄存器值。

参数:

`handle`: 实例句柄。

`value`: 获取出来的 load 寄存器值返回在 value 上。

返回值:

错误码。

3.2 USART

3.2.1 函数列表

- *csi_usart_initialize*
- *csi_usart_uninitialize*
- *csi_usart_get_capabilities*
- *csi_usart_send*
- *csi_usart_abort_send*
- *csi_usart_receive*
- *csi_usart_receive_query*
- *csi_usart_abort_receive*
- *csi_usart_transfer*
- *csi_usart_abort_transfer*
- *csi_usart_get_status*
- *csi_usart_flush*
- *csi_usart_set_interrupt*
- *csi_usart_config_baudrate*
- *csi_usart_config_mode*
- *csi_usart_config_parity*
- *csi_usart_config_stopbits*
- *csi_usart_config_databits*
- *csi_usart_getchar*

- *csi_usart_putchar*
- *csi_usart_get_tx_count*
- *csi_usart_get_rx_count*
- *csi_usart_power_control*
- *csi_usart_config_flowctrl*
- *csi_usart_config_clock*
- *csi_usart_control_tx*
- *csi_usart_control_rx*
- *csi_usart_control_break*

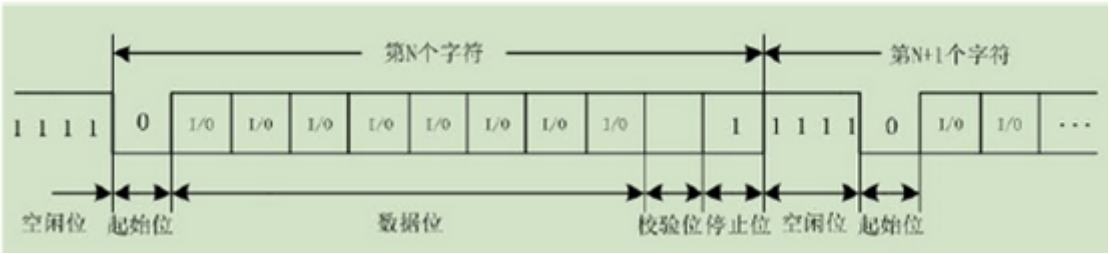
3.2.2 简要说明

USART(Universal Synchronous Asynchronous Receiver/Transmitter) 是一种同步或者异步的串行通信总线接口。当使用同步模式时，需要提供同步时钟，当使用异步模式 (UART) 的时候，不需要同步时钟，发送和接收方按照严格的格式（波特率和数据帧格式）发送和接收。

USART 的特点：

- 空闲时总线保持高电平状态
- 5~9 位数据位，低位在前
- 一个起始位
- 可选奇偶检验位
- 可选 0.5、1、1.5、2b 比特的停止位

USART 的传输时序描述如下：



3.2.3 接口描述

3.2.3.1 csi_usart_initialize

```
usart_handle_t csi_usart_initialize(int32_t idx, usart_event_cb_t cb_event)
```

功能描述：

通过设备号初始化对应的 usart 实例，返回 usart 实例句柄。

参数:

idx: 设备号

cb_event: usart 实例的事件回调函数 (一般在中断上下文执行)。回调函数原型定义见 usart_event_cb_t。

返回值:

NULL: 初始化失败。

其它: 初始化成功时的实例句柄。

usart_event_cb_t:

```
typedef void (*usart_event_cb_t)(int32_t idx, usart_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型。usart 回调事件枚举类型 usart_event_e 定义如下:

USART_EVENT_SEND_COMPLETE	数据发送完成事件
USART_EVENT_RECEIVE_COMPLETE	数据接收完成事件
USART_EVENT_TRANSFER_COMPLETE	数据传输完成事件
USART_EVENT_TX_COMPLETE	数据发送完成事件
USART_EVENT_TX_UNDERFLOW	数据发送溢出事件
USART_EVENT_RX_OVERFLOW	数据接收溢出事件
USART_EVENT_RX_TIMEOUT	数据接收超时事件
USART_EVENT_RX_BREAK	数据接收中断事件
USART_EVENT_RX_FRAMING_ERROR	帧数据接收错误事件
USART_EVENT_RX_PARITY_ERROR	数据接收奇偶校验错误事件
USART_EVENT_CTS	CTS 状态已改变
USART_EVENT_DSR	DSR 状态已改变
USART_EVENT_DCD	DCD 状态已改变
USART_EVENT_RI	RI 状态已改变
USART_EVENT_RECEIVED	数据接收并存在 USART FIFO， 可调用 receive 等函数去读取

3.2.3.2 csi_usart_uninitialize

```
int32_t csi_usart_uninitialize(usart_handle_t handle)
```

功能描述:

usart 实例反初始化。该接口会停止 usart 实例正在进行的传输（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.2.3.3 csi_usart_get_capabilities

```
usart_capabilities_t csi_usart_get_capabilities(int32_t idx)
```

功能描述:

获取 usart 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 usart 能力的结构体。

3.2.3.4 csi_usart_send

```
int32_t csi_usart_send(usart_handle_t handle, const void *data, uint32_t num)
```

功能描述:

usart 启动数据发送。

参数:

handle: 实例句柄。
data: 待发送数据的缓冲区地址。
num: 待发送数据的长度。

返回值:

错误码。

3.2.3.5 csi_usart_abort_send

```
int32_t csi_usart_abort_send(usart_handle_t handle)
```

功能描述:

usart 数据发送终止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.2.3.6 csi_usart_receive

```
int32_t csi_usart_receive(usart_handle_t handle, void *data, uint32_t num)
```

功能描述:

usart 启动数据接收。

参数:

handle: 实例句柄。
data: 待接收数据的缓冲区地址。
num: 待接收数据的长度。

返回值:

错误码。

3.2.3.7 csi_usart_receive_query

```
int32_t csi_usart_receive_query(usart_handle_t handle, void *data, uint32_t num)
```

功能描述:

查询方式从 USART 读取一定量数据。

参数:

handle: 实例句柄。

data: 待接收数据的缓冲区地址。

num: 预计接收数据的长度。

返回值:

错误码。

3.2.3.8 csi_usart_abort_receive

```
int32_t csi_usart_abort_receive(usart_handle_t handle)
```

功能描述:

usart 数据接收终止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.2.3.9 csi_usart_transfer

```
int32_t csi_usart_transfer(usart_handle_t handle, const void *data_out, void *data_in, uint32_t␣
↪num)
```

功能描述:

usart 启动数据传输，注意是同步传输。

参数:

handle: 实例句柄。

data_out: 待发送数据的缓冲区地址。

data_in: 待接收数据的缓冲区地址。

num: 数据的长度。

返回值:

错误码。

3.2.3.10 csi_usart_abort_transfer

```
int32_t csi_usart_abort_transfer(usart_handle_t handle)
```

功能描述:

usart 数据传输终止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.2.3.11 csi_usart_get_status

```
usart_status_t csi_usart_get_status(usart_handle_t handle)
```

功能描述:

获取当前时刻 usart 的状态。

参数:

handle: 实例句柄。

返回值:

usart 状态的结构体。详见*usart_status_t* 定义。

usart_status_t:

名字	定义	备注
tx_busy: 1	数据发送忙	
rx_busy: 1	数据接收忙	
tx_underflow: 1	数据发送溢出	
rx_overflow: 1	数据接收溢出	
rx_break: 1	数据接收中断	
rx_framing_error: 1	帧数据出错	
rx_parity_error: 1	数据接收奇偶检验错误	
tx_enable: 1	发送使能	
rx_enable: 1	接收使能	

3.2.3.12 csi_usart_flush

```
int32_t csi_usart_flush(usart_handle_t handle, usart_flush_type_e type)
```

功能描述:

清除 usart 数据缓存。

参数:

handle: 实例句柄。

type: usart 清除数据类型，参看*usart_flush_type_e* 定义。

返回值:

错误码。

usart_flush_type_e:

名字	定义	备注
USART_FLUSH_WRITE	清除写缓存空间	
USART_FLUSH_READ	清除读缓存空间	

3.2.3.13 csi_usart_set_interrupt

```
int32_t csi_usart_set_interrupt(usart_handle_t handle, usart_intr_type_e type, int32_t flag)
```

功能描述:

开关 USART 的中断。

参数:

handle: 实例句柄。

type: 中断类型。详见*usart_intr_type_e* 定义。

flag: 0：关；1：开。

返回值:

错误码。

usart_intr_type_e:

名字	定义	备注
USART_INTR_WRITE	usart 写中断	
USART_INTR_READ	usart 读中断	

3.2.3.14 csi_usart_config_baudrate

```
int32_t csi_usart_config_baudrate(usart_handle_t handle, uint32_t baud)
```

功能描述:

配置 usart 实例的波特率。

参数:

handle: 实例句柄。

baud: 波特率。

返回值:

错误码。

3.2.3.15 csi_usart_config_mode

```
int32_t csi_usart_config_mode(usart_handle_t handle, usart_mode_e mode)
```

功能描述:

配置 usart 实例的工作模式。

参数:

handle: 实例句柄。

mode: usart 的工作模式。详见*usart_mode_e* 定义。

返回值:

错误码。

usart_mode_e:

名字	定义	备注
USART_MODE_ASYNCHRONOUS	usart 异步模式	
USART_MODE_SYNCHRONOUS_MASTER	usart 同步全双工主模式	
USART_MODE_SYNCHRONOUS_SLAVE	usart 同步全双工从模式	
USART_MODE_SLAVE_WIRE	usart 半双工单线模式	
USART_MODE_SINGLE_IRDA	usart IRDA 模式	
USART_MODE_SINGLE_SMART_CARD	usart 智能卡模式	

3.2.3.16 csi_usart_config_parity

```
int32_t csi_usart_config_parity(usart_handle_t handle, usart_parity_e parity)
```

功能描述:

配置 usart 实例的奇偶校验模式。

参数:

handle: 实例句柄。

parity: usart 的奇偶校验，参看*usart_parity_e* 的定义。

返回值:

错误码。

usart_parity_e:

名字	定义	备注
USART_PARITY_NONE	无奇偶检验	
USART_PARITY_EVEN	偶校验	
USART_PARITY_ODD	奇校验	
USART_PARITY_1	校验位设置为 1	
USART_PARITY_0	检验位设置为 0	

3.2.3.17 csi_usart_config_stopbits

```
int32_t csi_usart_config_stopbits(usart_handle_t handle, usart_stop_bits_e stopbit)
```

功能描述:

配置 usart 实例的停止位模式。

参数:

handle: 实例句柄。

stopbit: usart 的停止位，参看*usart_stop_bits_e* 的定义。

返回值:

错误码。

usart_stop_bits_e:

名字	定义	备注
USART_STOP_BITS_1	1 停止位	
USART_STOP_BITS_2	2 停止位	
USART_STOP_BITS_1_5	1.5 停止位	
USART_STOP_BITS_0_5	0.5 停止位	

3.2.3.18 csi_usart_config_databits

```
int32_t csi_usart_config_databits(usart_handle_t handle, usart_data_bits_e databits)
```

功能描述:

参数:

`handle`: 实例句柄。

`databits`: usart 的数据位宽，参看*usart_data_bits_e* 的定义。

返回值:

错误码。

usart_data_bits_e:

名字	定义	备注
USART_DATA_BITS_5	5 位数据位宽	
USART_DATA_BITS_6	6 位数据位宽	
USART_DATA_BITS_7	7 位数据位宽	
USART_DATA_BITS_8	8 位数据位宽	
USART_DATA_BITS_9	9 位数据位宽	

3.2.3.19 csi_usart_getchar

```
int32_t csi_usart_getchar(usart_handle_t handle, uint8_t *ch)
```

功能描述:

从 usart 读取一个字节。

参数:

handle: 实例句柄。.

ch: 返回读取到的字节。

返回值:

错误码。

3.2.3.20 csi_usart_putchar

```
int32_t csi_usart_putchar(usart_handle_t handle, uint8_t ch)
```

功能描述:

从 usart 发送一个字节。

参数:

handle: 实例句柄。

ch: 需发送的字节内容。

返回值:

错误码。

3.2.3.21 csi_usart_get_tx_count

```
uint32_t csi_usart_get_tx_count(usart_handle_t handle)
```

功能描述:

获取设备实例上次已发送的数据个数。

参数:

handle: 实例句柄。

返回值:

已发送的数据个数。

3.2.3.22 csi_usart_get_rx_count

```
uint32_t csi_usart_get_rx_count(usart_handle_t handle)
```

功能描述:

获取设备实例上一次已接收的数据个数。

参数:

handle: 实例句柄。

返回值:

已接收的数据个数。

3.2.3.23 csi_usart_power_control

```
int32_t csi_usart_power_control(usart_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。
state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.2.3.24 csi_usart_config_flowctrl

```
int32_t csi_usart_config_flowctrl(usart_handle_t handle,
                                  usart_flowctrl_type_e flowctrl_type)
```

功能描述:

配置 usart 实例的流控。

参数:

handle: 实例句柄。
flowctrl_type: 流控类型，参看*usart_flowctrl_type_e* 的定义。

返回值:

错误码。

usart_flowctrl_type_e:

名字	定义	备注
USART_FLOWCTRL_NONE	不带流控	
USART_FLOWCTRL_CTS	支持 CTS 流控	
USART_FLOWCTRL_RTS	支持 RTS 流控	
USART_FLOWCTRL_CTS_RTS	同时支持 CTS 和 RTS 流控	

3.2.3.25 csi_usart_config_clock

```
int32_t csi_usart_config_clock(usart_handle_t handle, usart_cpol_e cpol, usart_cpha_e cpha)
```

功能描述:

配置 usart 实例的通信极性与相位。

参数:

`handle`: 实例句柄。

`cpol`: usart 的极性，参看*usart_cpol_e* 的定义。

`cpha`: usart 的相位，参看*usart_cpha_e* 的定义。

返回值:

错误码。

usart_cpol_e:

名字	定义	备注
USART_CPOL0	数据在上升沿捕获	
USART_CPOL1	数据在下降沿捕获	

usart_cpha_e:

名字	定义	备注
USART_CPHA0	数据在第 0 个边沿采样	
USART_CPHA1	数据在第 1 个边沿采样	

3.2.3.26 csi_usart_control_tx

```
int32_t csi_usart_control_tx(usart_handle_t handle, uint32_t enable)
```

功能描述:

控制 usart 实例的发送使能。

参数:

handle: 实例句柄。
enable: 1 - 允许发送数据，0 - 不允许发送数据。

返回值:

错误码。

3.2.3.27 csi_usart_control_rx

```
int32_t csi_usart_control_rx(usart_handle_t handle, uint32_t enable)
```

功能描述:

控制 usart 实例的接收使能。

参数:

handle: 实例句柄。
enable: 1 - 允许接收数据，0 - 不允许接收数据。

返回值:

错误码。

3.2.3.28 csi_usart_control_break

```
int32_t csi_usart_control_break(usart_handle_t handle, uint32_t enable)
```

功能描述:

控制 usart 实例的 break 帧发送。

参数:

handle: 实例句柄。
enable: 1 - 开始发送 break 帧，0 - 停止发送 break 帧。

返回值:

错误码。

3.3 GPIO

3.3.1 函数列表

- *csi_gpio_pin_initialize*
- *csi_gpio_pin_uninitialize*
- *csi_gpio_power_control*
- *csi_gpio_pin_config_mode*
- *csi_gpio_pin_config_direction*
- *csi_gpio_pin_write*
- *csi_gpio_pin_read*
- *csi_gpio_pin_set_irq*

3.3.2 简要说明

GPIO(General Purpose Input Output) 通用输入/输出接口提供了对每个信号管脚进行单独的高低电平状态控制功能，或者读入管脚高低电平状态的功能。

GPIO 可配置为输入模式或输出模式。 当配置为输出模式时，由软件来控制管脚的的高低电平状态。输出模式可支持开漏输出或推挽输出两种类型。

当配置为输入模式时，管脚的高低电平状态由外部电路控制，软件可以读取管脚的高低电平状态。输入模式可支持带上拉电阻、下拉电阻的属性。

输入模式也支持将 GPIO 配置为中断触发源。中断触发模式一般包括高电平触发、低电平触发、双边沿触发、下降沿触发、上升沿触发五种模式。

3.3.3 接口描述

3.3.3.1 csi_gpio_pin_initialize

```
gpio_pin_handle_t csi_gpio_pin_initialize(int32_t gpio_pin, gpio_event_cb_t cb_event)
```

功能描述:

通过传入的 pin 号初始化对应的 gpio pin 实例，返回 gpio pin 实例的句柄。

参数:

`gpio_pin`: pin 号。

`cb_event`: 对应 pin 中断回调函数。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

3.3.3.2 csi_gpio_pin_uninitialize

```
int32_t csi_gpio_pin_uninitialize(gpio_pin_handle_t handle)
```

功能描述:

gpio pin 实例反初始化。该接口会停止 gpio 实例正在进行的传输（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.3.3.3 csi_gpio_power_control

```
int32_t csi_gpio_power_control(gpio_pin_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。
state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.3.3.4 csi_gpio_pin_config_mode

```
int32_t csi_gpio_pin_config_mode(gpio_pin_handle_t handle, gpio_mode_e mode)
```

功能描述:

配置 gpio pin 实例的工作模式。

参数:

handle: 实例句柄。
mode: gpio 的工作方式，参看*gpio_mode_e* 定义。

返回值:

错误码。

gpio_mode_e:

名字	定义	备注
GPIO_MODE_PULLNONE	不操作	
GPIO_MODE_PULLUP	上拉操作	
GPIO_MODE_PULLDOWN	下拉操作	
GPIO_MODE_OPEN_DRAIN	开漏操作	
GPIO_MODE_PUSH_PULL	推挽操作	

3.3.3.5 csi_gpio_pin_config_direction

```
int32_t csi_gpio_pin_config_direction(gpio_pin_handle_t handle, gpio_direction_e dir);
```

功能描述:

参数:

`handle`: 实例句柄。

`dir`: gpio port 口的方向，参看*gpio_direction_e* 的定义。

返回值:

错误码。

gpio_direction_e:

名字	定义	备注
GPIO_DIRECTION_INPUT	输入模式	
GPIO_DIRECTION_OUTPUT	输出模式	

3.3.3.6 csi_gpio_pin_write

```
int32_t csi_gpio_pin_write(gpio_pin_handle_t handle, bool value)
```

功能描述:

设置 gpio pin 的电平状态。

参数:

`handle`: 实例句柄。

`value`: 对应 pin 口中的值。

返回值:

错误码。

3.3.3.7 csi_gpio_pin_read

```
int32_t csi_gpio_pin_read(gpio_pin_handle_t handle, bool *value)
```

功能描述:

获取 pin 脚的电平状态。

参数:

handle: 实例句柄。

value: 存放 pin 脚的电平状态的缓冲区地址。

返回值:

错误码。

3.3.3.8 csi_gpio_pin_set_irq

```
int32_t csi_gpio_pin_set_irq(gpio_pin_handle_t handle, gpio_irq_mode_e mode, bool enable)
```

功能描述:

设置 pin 脚的中断模式。

参数:

handle: 实例句柄。

mode: 中断模式，参见:ref: *gpio_irq_mode_e <gpio_irq_mode_e>* 定义。

enable: 设置中断是否开启，0-禁止，1-使能。

返回值:

错误码。

gpio_irq_mode_e:

名字	定义	备注
GPIO_IRQ_MODE_RISING_EDGE	上升沿中断模式	
GPIO_IRQ_MODE_FALLING_EDGE	下降沿中断模式	
GPIO_IRQ_MODE_DOUBLE_EDGE	双边沿中断模式	
GPIO_IRQ_MODE_LOW_LEVEL	低电平中断模式	
GPIO_IRQ_MODE_HIGH_LEVEL	高电平中断模式	

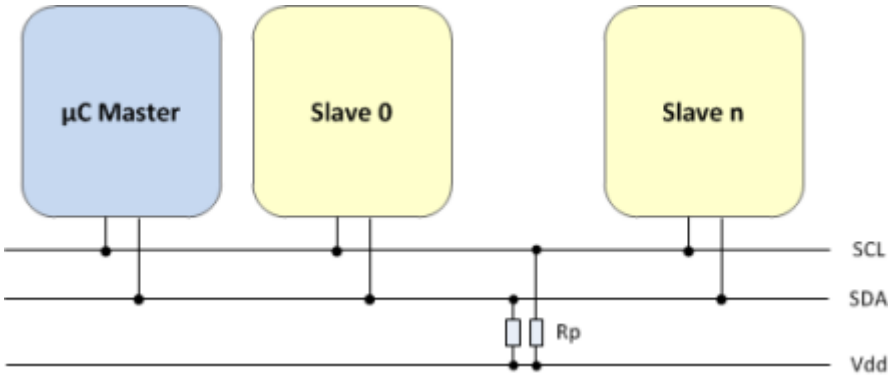
3.4 IIC

3.4.1 函数列表

- *csi_iic_initialize*
- *csi_iic_uninitialize*
- *csi_iic_get_capabilities*
- *csi_iic_master_send*
- *csi_iic_master_receive*
- *csi_iic_slave_send*
- *csi_iic_slave_receive*
- *csi_iic_abort_transfer*
- *csi_iic_get_status*
- *csi_iic_power_control*
- *csi_iic_config_mode*
- *csi_iic_config_speed*
- *csi_iic_config_addr_mode*
- *csi_iic_config_slave_addr*
- *csi_iic_get_data_count*
- *csi_iic_send_start*
- *csi_iic_send_stop*
- *csi_iic_reset*

3.4.2 简要说明

I2C (Inter-Integrated Circuit, 或 IIC) 是一种串行的同步通信总线。I2C 串行总线有两根信号线，一根是双向的数据线 SDA，另一根是时钟线 SCL。所有接到 I2C 总线设备上的串行数据 SDA 都接到总线的 SDA 上，各设备的时钟线 SCL 接到总线的 SCL 上。IIC 典型接线方式如下:



总线的通信由主机控制，即主机是数据的传送（发出启动信号）、发出时钟信号以及传送结束时发出停止信号的设备。被主机寻访的设备称为从机。每个接到 I2C 总线的设备都有一个唯一的地址，以便于主机寻访。主机和从机的数据传送，可以由主机发送数据到从机，也可以由从机发到主机。I2C 支持 7 位或者 10 位从设备地址模式。I2C 总线在开始条件后的首字节决定哪个被控器将被主控器选择，当主机输出一地址时，系统中的每一从设备都将开始条件后的地址和自己的地址进行比较，如果相同，该从机即认为自己被主机寻址。

3.4.3 接口描述

3.4.3.1 csi_iic_initialize

```
iic_handle_t csi_iic_initialize(int32_t idx, iic_event_cb_t cb_event)
```

功能描述:

通过设备号初始化对应的 iic 实例，返回 iic 实例的句柄。

参数:

`idx`: 设备号。

`cb_event`: iic 实例的事件回调函数（一般在中断上下文执行）。回调函数原型定义见 iic_event_cb_t。

回调函数类型 iic_event_cb_t 定义如下：

```
typedef void (*iic_event_cb_t)(int32_t idx, iic_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

IIC 回调事件枚举类型 iic_event_e 定义如下：

名字	定义	备注
IIC_EVENT_TRANSFER_DONE	传输完成事件	
IIC_EVENT_TRANSFER_INCOMPLETE	传输未完成事件	
IIC_EVENT_SLAVE_TRANSMIT	从设备发送操作请求事件	
IIC_EVENT_SLAVE_RECEIVE	从设备接收操作请求事件	
IIC_EVENT_ADDRESS_NACK	地址未应答事件	
IIC_EVENT_GENERAL_CALL	指示收到 general call（地址为 0）事件	
IIC_EVENT_ARBITRATION_LOST	主机仲裁丢失事件	
IIC_EVENT_BUS_ERROR	总线错误事件	
IIC_EVENT_BUS_CLEAR	总线清除完成事件	

3.4.3.2 csi_iic_uninitialize

```
int32_t csi_iic_uninitialize(iic_handle_t handle)
```

功能描述:

usart 实例反初始化。该接口会停止 usart 实例正在进行的传输（如果有），并且释放相关的软硬件资源。

参数:

`handle`: 实例句柄。

返回值:

错误码。

3.4.3.3 csi_iic_get_capabilities

```
iic_capabilities_t csi_iic_get_capabilities(int32_t idx)
```

功能描述:

获取 iic 实例支持的能力。

参数:

`idx`: 设备号。

返回值:

描述 iic 能力的结构体。

3.4.3.4 csi_iic_master_send

```
int32_t csi_iic_master_send(iic_handle_t handle, uint32_t devaddr, const void *data, uint32_t num,␣
↪bool xfer_pending)
```

功能描述:

iic 作为主机时启动数据发送。

参数:

handle: 实例句柄。

devaddr: Slave 设备地址。

data: 待发送数据的缓冲区地址。

num: 待发送数据的长度。

xfer_pending: 发送完成后是否发送停止位。1-不发送停止位，0-发送停止位。

返回值:

错误码。

3.4.3.5 csi_iic_master_receive

```
int32_t csi_iic_master_receive(iic_handle_t handle, uint32_t devaddr, void *data, uint32_t num,␣
↪bool xfer_pending)
```

功能描述:

参数:

handle: 实例句柄。

devaddr: Slave 设备地址。

data: 待接收数据的缓冲区地址。

num: 待接收数据的长度。

xfer_pending: 接收完成后是否发送停止位。1-不发送停止位，0-发送停止位。

返回值:

错误码。

3.4.3.6 csi_iic_slave_send

```
int32_t csi_iic_slave_send(iic_handle_t handle, const void *data, uint32_t num)
```

功能描述:

iic 作为从机时启动数据发送。

参数:

handle: 实例句柄。

data: 待发送数据的缓冲区地址。

num: 待发送数据的长度。

返回值:

错误码。

3.4.3.7 csi_iic_slave_receive

```
int32_t csi_iic_slave_receive(iic_handle_t handle, void *data, uint32_t num)
```

功能描述:

iic 作为从机时启动数据接收。

参数:

handle: 实例句柄。

data: 待接收数据的缓冲区地址。

num: 待接收数据的长度。

返回值:

错误码。

3.4.3.8 csi_iic_abort_transfer

```
int32_t csi_iic_abort_transfer(iic_handle_t handle)
```

功能描述:

停止正在执行的传输，包括发送和接收。若 iic 设备处于空闲状态时，不做任何操作。

参数:

handle: 实例句柄。

返回值:

错误码。

3.4.3.9 csi_iic_get_status

```
iic_status_t csi_iic_get_status(iic_handle_t handle)
```

功能描述:

获取当前时刻 IIC 的状态。

参数:

handle: 实例句柄。

返回值:

iic 状态的结构体。iic 的状态定义见*iic_status_t* 定义。

iic_status_t:

名字	定义	备注
busy：1	传输或发送忙状态位	
mode：1	模式位。1-主，0-从	
direction：1	传输方向：1-接收，0-发送	
general_call：1	general call 指示	
arbitration_lost：1	主机失去仲裁	
bus_error：1	总线错误	

3.4.3.10 csi_iic_power_control

```
int32_t csi_iic_power_control(iic_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置 iic 实例的功耗模式。

参数:

handle: 实例句柄。

state: iic 的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.4.3.11 csi_iic_config_mode

```
int32_t csi_iic_config_mode(iic_handle_t handle, iic_mode_e mode)
```

功能描述:

配置 iic 实例的主从工作模式。

参数:

handle: 实例句柄。

mode: iic 的主、从工作模式，参看*iic_mode_e* 的定义。

返回值:

错误码。

iic_mode_e:

名字	定义	备注
IIC_MODE_MASTER	IIC 主机模式	
IIC_MODE_SLAVE	IIC 从机模式	

3.4.3.12 csi_iic_config_speed

```
int32_t csi_iic_config_speed(iic_handle_t handle, iic_speed_e speed)
```

功能描述:

配置 iic 实例的工作速度。

参数:

handle: 实例句柄。

speed: iic 的速度，参看*iic_speed_e* 的定义。

返回值:

错误码。

iic_speed_e:

名字	定义	备注
I2C_BUS_SPEED_STANDARD	iic 标准速度 (100KHz)	
I2C_BUS_SPEED_FAST	iic 快速速度 (400KHz)	
I2C_BUS_SPEED_FAST_PLUS	iic 标准 + 速度 (1MHz)	
I2C_BUS_SPEED_HIGH	iic 高速速度 (3.4MHz)	

3.4.3.13 csi_iic_config_addr_mode

```
int32_t csi_iic_config_addr_mode(iic_handle_t handle, iic_address_mode_e addr_mode)
```

功能描述:

配置 iic 实例的地址模式。

参数:

`handle`: 实例句柄。

`addr_mode`: iic 的地址模式，参看*iic_address_mode_e* 的定义。

返回值:

错误码。

iic_address_mode_e:

名字	定义	备注
I2C_ADDRESS_7BIT	7bit 地址模式	
I2C_ADDRESS_10BIT	10bit 地址模式	

3.4.3.14 csi_iic_config_slave_addr

```
int32_t csi_iic_config_slave_addr(iic_handle_t handle, int32_t slave_addr)
```

功能描述:

配置 iic 实例的从设备地址。

参数:

`handle`: 实例句柄。

`slave_addr`: iic 从设备通信地址。

返回值:

错误码。

3.4.3.15 csi_iic_get_data_count

```
uint32_t csi_iic_get_data_count(iic_handle_t handle)
```

功能描述:

获取 iic 实例的已传输的数据个数。

参数:

handle: 实例句柄。

返回值:

已传输的数据个数。

3.4.3.16 csi_iic_send_start

```
int32_t csi_iic_send_start(iic_handle_t handle)
```

功能描述:

发送 START 命令。

参数:

handle: 实例句柄。

返回值:

错误码。

3.4.3.17 csi_iic_send_stop

```
int32_t csi_iic_send_stop(iic_handle_t handle)
```

功能描述:

发送 STOP 命令。

参数:

handle: 实例句柄。

返回值:

错误码。

3.4.3.18 csi_iic_reset

```
int32_t csi_iic_reset(iic_handle_t handle)
```

功能描述:

复位 IIC。

参数:

handle: 实例句柄。

返回值:

错误码。

3.4.4 示例

3.4.4.1 IIC 示例 1

```
static iic_handle_t iic_handle;
static uint8_t cb_transfer_flag = 0xff;

static void iic_event_cb_fun(int32_t idx, iic_event_e event)
{
    cb_transfer_flag = event;
}
```

(下页继续)

(续上页)

```
void example_main(void)
{
    iic_capabilities_t cap;
    int32_t ret;

    //receive buffer
    char rcv_buf[10] ;
    //data to send
    char send_buf[10] = {0,1,2,3,4,5,6,7,8,9};

    //get iic capabilities
    cap = csi_iic_get_capabilities(0);
    printf("iic %s 10bit address\n",cap.address_10_bit==1 ? "support":"not support");

    iic_handle = csi_iic_initialize(0, iic_event_cb_fun);
    if (iic_handle == NULL) {
        //fail
        return;
    }

    ret = csi_iic_power_control(iic_handle, DRV_POWER_FULL);
    if (ret < 0) {
       // power control failed
        return;
    }

   //config iic as master-standard speed-7bit address-slave addr=0x57
    csi_iic_config_mode(iic_handle, IIC_MODE_MASTER);
    csi_iic_config_speed(iic_handle, I2C_BUS_SPEED_STANDARD);
    csi_iic_config_addr_mode(iic_handle, I2C_ADDRESS_7BIT);
    csi_iic_config_slave_addr(iic_handle, 0x57);

    ret = csi_iic_master_send(iic_handle, 0x57, send_buf, sizeof(send_buf), 0);
    if (ret < 0) {
        //failed
    }

    while (cb_transfer_flag == 0xff);

    //check transfer result
    if (cb_transfer_flag == I2C_EVENT_TRANSFER_DONE) {
        //transmit done
    } else {
```

(下页继续)

(续上页)

```
        //failed
    }
    cb_transfer_flag = 0xff;

    ret = csi_iic_master_receive(iic_handle, 0x57, rcv_buf, sizeof(rcv_buf), 0);
    if (ret < 0) {
        //failed
    }

    while (cb_transfer_flag == 0xff);

    //check transfer result
    if (cb_transfer_flag == I2C_EVENT_TRANSFER_DONE) {
        //transmit done
    } else {
        //failed
    }

    ret = csi_iic_power_control(iic_handle, DRV_POWER_OFF);
    if (ret < 0) {
       // power control failed
        return;
    }

    //uninitialize iic
    ret = csi_iic_uninitialize(iic_handle);
    if (ret != 0) {
       //failed
    }

}
```

3.4.4.2 IIC 示例 2

```
static iic_handle_t iic_handle;

//event callback for iic
static void iic_event_cb_fun(int32_t idx, iic_event_e event)
{
    //...
}
```

(下页继续)

(续上页)

```
static void example_main(void)
{
    int32_t ret;
    iic_status_t st;

    //data to send
    char send_buf[10] = {0,1,2,3,4,5,6,7,8,9};
    //receive buffer
    char rcv_buf[10] ;

    iic_handle = csi_iic_initialize(0, iic_event_cb_fun);
    if (iic_handle == NULL) {
        return;
    }
    ret = csi_iic_power_control(iic_handle, DRV_POWER_FULL);
    if (ret < 0) {
       // power control failed
        return;
    }

    //config iic as slave-standard speed-7bit address-slave addr=0x57
    csi_iic_config_mode(iic_handle, IIC_MODE_SLAVE);
    csi_iic_config_speed(iic_handle, I2C_BUS_SPEED_STANDARD);
    csi_iic_config_addr_mode(iic_handle, I2C_ADDRESS_7BIT);
    csi_iic_config_slave_addr(iic_handle, 0x57);

    ret = csi_iic_slave_send(iic_handle, send_buf, sizeof(send_buf));
    if (ret < 0) {
        //failed
    }

    //wait send done, waiting for 100 ms max
    int32_t cnt = 100;
    while(cnt--) {
        mdelay(1);
        //check status
        st = csi_iic_get_status(iic_handle);
        //transmit done
        if (st.busy == 0){
            break;
        }
    }
```

(下页继续)

(续上页)

```
    //abort transmit when timeout
    if (cnt == 0) {
        csi_iic_abort_transfer(iic_handle);
    }

    st.busy = 0;
    ret = csi_iic_slave_receive(iic_handle, rcv_buf, sizeof(rcv_buf));
    if (ret < 0) {
        //failed
    }

    //wait receive done, waiting for 100 ms max
    cnt = 100;
    while(cnt--) {
        mdelay(1);
        //check status
        st = csi_iic_get_status(iic_handle);
        //transmit done
        if (st.busy == 0){
            break;
        }
    }

    //abort transmit when timeout
    if (cnt == 0) {
        csi_iic_abort_transfer(iic_handle);
    }
    ret = csi_iic_power_control(iic_handle, DRV_POWER_OFF);
    if (ret < 0) {
       // power control failed
        return;
    }

    //uninitialize iic
    ret = csi_iic_uninitialize(iic_handle);
    if (ret != 0) {
       //failed
    }
}
```

3.5 SPI

3.5.1 函数列表

- *csi_spi_initialize*
- *csi_spi_uninitialize*
- *csi_spi_get_capabilities*
- *csi_spi_send*
- *csi_spi_receive*
- *csi_spi_transfer*
- *csi_spi_abort_transfer*
- *csi_spi_get_status*
- *csi_spi_config_mode*
- *csi_spi_config_block_mode*
- *csi_spi_config_baudrate*
- *csi_spi_config_bit_order*
- *csi_spi_config_datawidth*
- *csi_spi_config_format*
- *csi_spi_config_ss_mode*
- *csi_spi_get_data_count*
- *csi_spi_power_control*
- *csi_spi_ss_control*

3.5.2 简要说明

SPI（Serial Peripheral Interface）是一种高速的、全双工、同步的通信总线。SPI 以主从方式工作，通常有一个主设备和一个或多个从设备。

SPI 控制器的信号线描述如下：

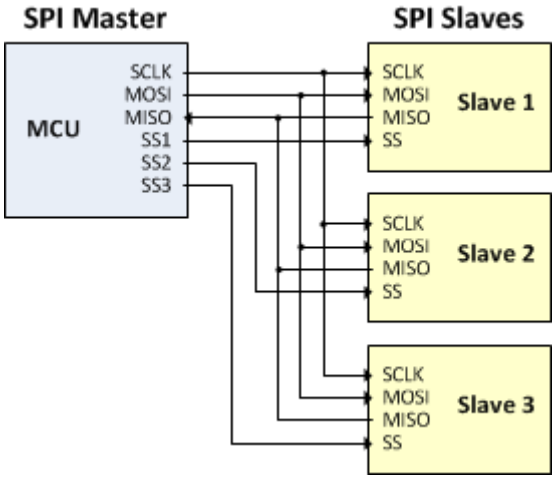
- MISO：主设备数据输入，从设备数据输出；
- MOSI：主设备数据输出，从设备数据输入；
- SCLK：时钟信号，由主设备产生；
- SS：从设备使能信号，由主设备控制。这个信号可以是 SPI 外设的一部分，也可用 GPIO 引脚实现。

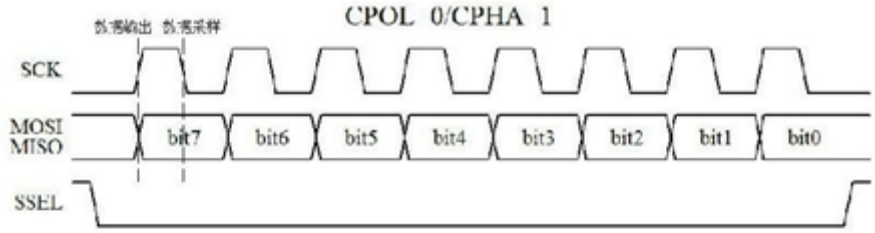
SPI 典型接线方式如下：

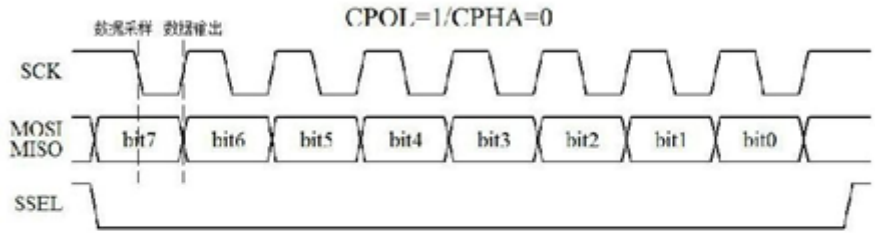
SPI 总线支持的四种工作方式，取决于串行同步时钟极性 (CPOL) 和串行同步时钟相位 CPHA 的组合。

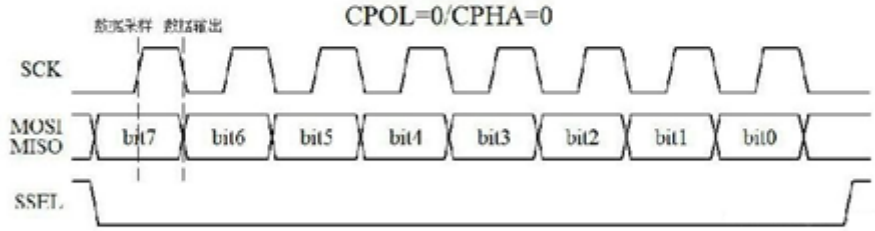
四种工作方式时序描述如下：

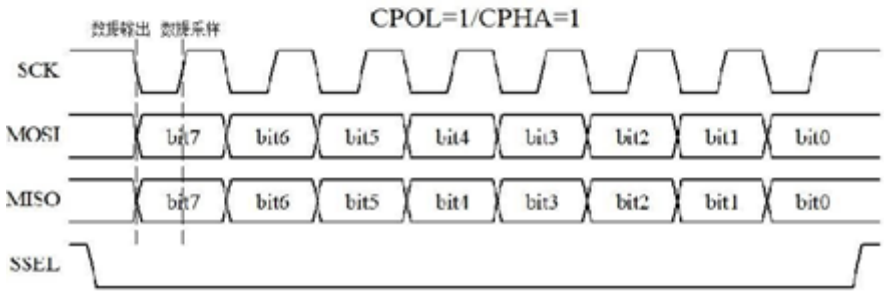
CPOL 是用来决定 SCLK 时钟信号空闲时的电平，CPOL=0，空闲电平为低电平，CPOL=1 时，空闲电平为高电平。CPHA 是用来决定采样时刻的，CPHA=0，在每个周期的第一个时钟沿采样，第二个时钟沿数据输出；CPHA=1，在每个周期的第二个时钟沿采样，第一个时钟沿数据输出。SPI 主模块和与之通信的外设时钟相位和极性应该一致。











3.5.3 接口描述

3.5.3.1 csi_spi_initialize

```
spi_handle_t csi_spi_initialize(int32_t idx, spi_event_cb_t cb_event)
```

功能描述:

通过设备号初始化对应的 spi 实例，返回 spi 实例的句柄。

参数:

idx: 设备号。

cb_event: spi 实例的事件回调函数（一般在中断上下文执行）。回调函数原型定义见 spi_event_cb_t。

回调函数类型 iic_event_cb_t 定义如下：

```
typedef void (*spi_event_cb_t)(int32_t idx, spi_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，spi 回调事件枚举类型*spi_event_e*

返回值:

NULL: 初始化失败。

其它: 实例句柄。

spi_event_e:

名字	定义	备注
SPI_EVENT_TRANSFER_COMPLETE	传输完成事件	
SPI_EVENT_TX_COMPLETE	发送完成事件	
SPI_EVENT_RX_COMPLETE	接收完成事件	
SPI_EVENT_DATA_LOST	数据丢失事件	
SPI_EVENT_MODE_FAULT	模式错误事件	

3.5.3.2 csi_spi_uninitialize

```
int32_t csi_spi_uninitialize(spi_handle_t handle)
```

功能描述:

spi 实例反初始化。该接口会停止 spi 实例正在进行的传输（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.5.3.3 csi_spi_get_capabilities

```
spi_capabilities_t csi_spi_get_capabilities(int32_t idx)
```

功能描述:

获取 spi 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 spi 能力的结构体，spi 的能力定义见*spi_capabilities_t* 定义。

spi_capabilities_t:

名字	定义	备注
simplex :1	支持单向传输模式，即半双工模式（主模式或从模式）	
ti_ssi :1	支持同步串行接口（SSI）工业通信接口	
microwire :1	支持 Microwire 串行接口	
event_mode_fault :1	信号模式故障事件	

3.5.3.4 csi_spi_send

```
int32_t csi_spi_send(spi_handle_t handle, const void *data, uint32_t num)
```

功能描述:

spi 启动数据发送。

参数:

handle: 实例句柄。
data: 待发送数据的缓冲区地址。
num: 待发送数据的长度。

返回值:

错误码。

3.5.3.5 csi_spi_receive

```
int32_t csi_spi_receive(spi_handle_t handle, void *data, uint32_t num)
```

功能描述:

spi 启动数据接收。

参数:

handle: 实例句柄。

data: 待接收数据的缓冲区地址。

num: 待接收数据的长度。

返回值:

错误码。

3.5.3.6 csi_spi_transfer

```
int32_t csi_spi_transfer(spi_handle_t handle, const void *data_out, void *data_in, uint32_t num_
↪out, uint32_t num_in)
```

功能描述:

spi 启动数据传输。

参数:

handle: 实例句柄。

data_out: 待发送数据的缓冲区地址。

data_in: 待接收数据的缓冲区地址。

num_out: 待发送数据的长度。

num_in: 待接收数据的长度。

返回值:

错误码。

3.5.3.7 csi_spi_abort_transfer

```
int32_t csi_spi_abort_transfer(spi_handle_t handle)
```

功能描述:

spi 数据传输终止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.5.3.8 csi_spi_get_status

```
spi_status_t csi_spi_get_status(spi_handle_t handle)
```

功能描述:

获取当前时刻 spi 的状态。

参数:

handle: 实例句柄。

返回值:

spi 状态的结构体，spi 的状态定义见*spi_status_t* 的定义。

spi_status_t:

名字	定义	备注
busy：1	传输或发送忙状态位	
data_lost：1	数据丢失	
mode_fault：1	模式错误	

3.5.3.9 csi_spi_config_mode

```
int32_t csi_spi_config_mode(spi_handle_t handle, spi_mode_e  mode)
```

功能描述:

配置 spi 实例的主从工作模式。

参数:

handle: 实例句柄。

mode: spi 的主从模式，参看*spi_mode_e* 的定义。

返回值:

错误码。

spi_mode_e:

名字	定义	备注
SPI_MODE_INACTIVE	spi 闲置	
SPI_MODE_MASTER	spi 全双工主模式	
SPI_MODE_SLAVE	spi 全双工从模式	
SPI_MODE_MASTER_SIMPLEX	spi 半双工主模式	
SPI_MODE_SLAVE_SIMPLEX	spi 半双工从模式	

3.5.3.10 csi_spi_config_block_mode

```
int32_t csi_spi_config_block_mode(spi_handle_t handle, int32_t flag)
```

功能描述:

配置 spi 实例的阻塞模式。

参数:

handle: 实例句柄。

flag: 1：开启 block 模式；0：关闭 block 模式

返回值:

错误码。

3.5.3.11 csi_spi_config_baudrate

```
int32_t csi_spi_config_baudrate(spi_handle_t handle, uint32_t baud)
```

功能描述:

配置 spi 实例的速率。

参数:

handle: 实例句柄。

baud: spi 的波特率。

返回值:

错误码。

3.5.3.12 csi_spi_config_bit_order

```
int32_t csi_spi_config_bit_order(spi_handle_t handle, spi_bit_order_e order)
```

功能描述:

配置 spi 实例的数据传输模式。

参数:

handle: 实例句柄。

order: spi 的数据传输模式，参看*spi_bit_order_e* 的定义。

返回值:

错误码。

spi_bit_order_e:

名字	定义	备注
SPI_ORDER_MSB2LSB	高位（MSB）在前，低位（LSB）在后	
SPI_ORDER_LSB2MSB	低位（LSB）在前，高位（MSB）在后	

3.5.3.13 csi_spi_config_datawidth

```
int32_t csi_spi_config_datawidth(spi_handle_t handle, uint32_t datawidth)
```

功能描述:

配置 spi 实例的工作模式。

参数:

handle: 实例句柄。

datawidth: spi 的数据位宽。

返回值:

错误码。

3.5.3.14 csi_spi_config_format

```
int32_t csi_spi_config_format(spi_handle_t handle, spi_format_e format)
```

功能描述:

配置 spi 实例的极性和相位模式。

参数:

handle: 实例句柄。

format: spi 的工作模式，参看*spi_format_e* 的定义。

返回值:

错误码。

spi_format_e:

名字	定义	备注
SPI_FORMAT_CPOL0_CPHA0	空闲电平为低电平，第 1 个时钟沿采样	
SPI_FORMAT_CPOL0_CPHA1	空闲电平为低电平，第 2 个时钟沿采样	
SPI_FORMAT_CPOL1_CPHA0	空闲电平为高电平，第 1 个时钟沿采样	
SPI_FORMAT_CPOL0_CPHA1	空闲电平为高电平，第 2 个时钟沿采样	

3.5.3.15 csi_spi_config_ss_mode

```
int32_t csi_spi_config_ss_mode(spi_handle_t handle, spi_ss_mode_e ss_mode)
```

功能描述:

配置 spi 实例的从设备选择模式。

参数:

handle: 实例句柄。

ss_mode: spi 的从设备使能模式，参看:ref: *spi_ss_mode_e <spi_ss_mode_e>* 的定义。

返回值:

错误码。

spi_ss_mode_e:

名字	定义	备注
SPI_SS_MASTER_UNUSED	从设备使能主模式未使能	
SPI_SS_MASTER_SW	从设备使能主模式软件模式	
SPI_SS_MASTER_HW_OUTPUT	从设备使能主模式硬件输出模式	
SPI_SS_MASTER_HW_INPUT	从设备使能主模式硬件输入模式	
SPI_SS_SLAVE_HW	从设备使能从模式硬件模式	
SPI_SS_SLAVE_SW	从设备使能从模式软件模式	

3.5.3.16 csi_spi_get_data_count

```
uint32_t csi_spi_get_data_count(spi_handle_t handle)
```

功能描述:

获取设备实例的上一次传输的数据个数。

参数:

handle: 实例句柄。

返回值:

上一次传输的数据个数。

3.5.3.17 csi_spi_power_control

```
int32_t csi_spi_power_control(spi_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。

state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.5.3.18 csi_spi_ss_control

```
int32_t csi_spi_ss_control(spi_handle_t handle, spi_ss_stat_e stat)
```

功能描述:

控制从设备实例的选择信号状态。

参数:

handle: 实例句柄。

stat: 设备选择信号状态，见*spi_ss_stat_e* 定义。

返回值:

错误码。

spi_ss_stat_e:

名字	定义	备注
SPI_SS_INACTIVE	从设备选择信号无效	
SPI_SS_ACTIVE	从设备选择信号有效	

3.5.4 示例

3.5.4.1 SPI 示例 1

```
static spi_handle_t spi_handle;

static void spi_event_cb_fun(int32_t idx, spi_event_e event)
{
    //do your job here according to event
}

void example_main(void)
{
    spi_capabilities_t cap;
    int32_t ret;
    uint8_t send_buf[8] ={0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8};
    uint8_t recv_buf[8];
    uint8_t i;

    spi_handle = csi_spi_initialize(1,spi_event_cb_fun);
    if (spi_handle == NULL) {
        //fail
        return;
    }

    ret = csi_spi_power_control(spi_handle, DRV_POWER_FULL);
    if (ret < 0) {
       // power control failed
        return;
    }

    //get spi capabilities
    cap = csi_spi_get_capabilities(1);
    printf("spi %s simplex mode\n",cap.simplex==1 ? "supports":"not supports");
    printf("spi %s TI Synchronous Serial Interface\n",cap.ti_ssi==1 ? "supports":
           "not supports");
    printf("spi %s Microwire Interface\n",cap.microwire==1 ? "supports":"not supports");
    printf("spi %s signal mode fault event\n",cap.event_mode_fault==1 ? "have":
           "have not");

    //config spi as: baud 115200,master mode ,mode= 0,MSB2LSB, 7bit
    ret = csi_spi_config_mode(spi_handle, SPI_MODE_MASTER);
    if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_config_block_mode(spi_handle, 1);
```

(下页继续)

(续上页)

```
    if (ret < 0) {
        //config fail
    return ;
}

ret = csi_spi_config_baudrate(spi_handle, 115200);
if (ret < 0) {
    //config fail
    return ;
}

ret = csi_spi_config_datawidth(spi_handle, 8);
if (ret < 0) {
    //config fail
    return ;
}

ret = csi_spi_config_bit_order(spi_handle, SPI_ORDER_MSB2LSB);
if (ret < 0) {
    //config fail
    return ;
}

ret = csi_spi_config_format(spi_handle, SPI_FORMAT_CPOL0_CPHA0);
if (ret < 0) {
    //config fail
    return ;
}

ret = csi_spi_send(spi_handle, send_buf, sizeof(send_buf));
if (ret < 0) {
    //send data failed
    return;
}
//check send result here
ret = csi_spi_receive(spi_handle, recv_buf, sizeof(recv_buf));
if (ret < 0) {
    //receive data failed
    return;
}
//check receive result here

for (i = 0; i < sizeof(send_buf); i++) {
```

(下页继续)

(续上页)

```
        if (send_buf[i] != recv_buf[i]) {
            // send and receive data failed
            return;
        }
    }

    ret = csi_spi_power_control(spi_handle, DRV_POWER_OFF);
    if (ret < 0) {
       // power control failed
        return;
    }

    //uninitialize spi
    ret = csi_spi_uninitialize(spi_handle);
    if (ret != 0) {
       //failed
    }
}
```

3.5.4.2 SPI 示例 2

```
static spi_handle_t spi_handle;

static void spi_event_cb_fun(int32_t idx, spi_event_e event)
{
    //do your job here according to event
}

void example_main(void)
{
    int32_t ret;
    uint8_t send_buf[8] ={0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8};
    uint8_t recv_buf[8];
    uint8_t i;

    spi_handle = csi_spi_initialize(1, spi_event_cb_fun);
    if (spi_handle == NULL) {
        //failed
        return;
    }

    ret = csi_spi_power_control(spi_handle, DRV_POWER_FULL);
```

(下页继续)

(续上页)

```
    if (ret < 0) {
       // power control failed
        return;
    }

    //config spi as: baud 115200,master mode ,mode= 0,MSB2LSB, 7bit
    ret = csi_spi_config_mode(spi_handle, SPI_MODE_MASTER);
        if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_config_block_mode(spi_handle, 1);
        if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_config_baudrate(spi_handle, 115200);
    if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_config_datawidth(spi_handle, 8);
    if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_config_bit_order(spi_handle, SPI_ORDER_MSB2LSB);
    if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_config_format(spi_handle, SPI_FORMAT_CPOL0_CPHA0);
    if (ret < 0) {
        //config fail
        return ;
    }

    ret = csi_spi_transfer(spi_handle, send_buf, recv_buf, sizeof(send_buf), sizeof(recv_buf));
```

(下页继续)

(续上页)

```
    if (ret < 0) {
        //spi transfer failed
        return;
    }

    for (i = 0; i < sizeof(send_buf); i++) {
        if (send_buf[i] == recv_buf[i]) {
            // send and receive data should not same
            break;
        }
    }

    ret = csi_spi_power_control(spi_handle, DRV_POWER_OFF);
    if (ret < 0) {
        // power control failed
        return;
    }

    //uninitialize spi
    ret = csi_spi_uninitialize(spi_handle);
        if (ret != 0) {
        //failed
    }
}
```

3.6 PWM

3.6.1 函数列表

- *csi_pwm_initialize*
- *csi_pwm_uninitialize*
- *csi_pwm_power_control*
- *csi_pwm_config*
- *csi_pwm_start*
- *csi_pwm_stop*

3.6.2 简要说明

脉冲宽度调制（PWM）基本原理：控制方式就是对逆变电路开关器件的通断进行控制，使输出端得到一系列幅值相等的脉冲，用这些脉冲来代替正弦波或所需要的波形。也就是在输出波形的半个周期中产生多个脉冲，使各脉冲的等值

电压为正弦波形，所获得的输出平滑且低次谐波少。按一定的规则对各脉冲的宽度进行调制，即可改变逆变电路输出电压的大小，也可改变输出频率。

3.6.3 接口描述

3.6.3.1 csi_pwm_initialize

```
pwm_handle_t csi_pwm_initialize(uint32_t idx)
```

功能描述:

通过传入的 idx 号初始化对应的 PWM 控制器实例，返回控制器实例的句柄。

参数:

idx: 设备号。

返回值:

NULL: 初始化失败。
其它: 实例句柄。

3.6.3.2 csi_pwm_uninitialize

```
void csi_pwm_uninitialize(pwm_handle_t handle)
```

功能描述:

PWM 实例反初始化。该接口会停止 PWM 实例正在进行的传输（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

无

3.6.3.3 csi_pwm_power_control

```
int32_t csi_pwm_power_control(pwm_handle_t handle, csi_power_stat_e state)
```

功能描述:

PWM 电源控制。

参数:

handle: 实例句柄。

state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.6.3.4 csi_pwm_config

```
int32_t csi_pwm_config(pwm_handle_t handle,
                       uint8_t channel,
                       uint32_t period_us,
                       uint32_t pulse_width_us)
```

功能描述:

配置 PWM channel 通道的占空比。

参数:

handle: 实例句柄。

channel: 通道号。

period_us: 一个周期时间（单位微秒）。

pulse_width_us: 一个高电平脉冲时间（单位微秒）。

返回值:

错误码。

3.6.3.5 csi_pwm_start

```
void csi_pwm_start(pwm_handle_t handle, uint8_t channel)
```

功能描述:

开始产生信号。

参数:

handle: 实例句柄。

channel: 通道号。

返回值:

无。

3.6.3.6 csi_pwm_stop

```
void csi_pwm_stop(pwm_handle_t handle, uint8_t channel)
```

功能描述:

停止产生信号。

参数:

handle: 实例句柄。

channel: 通道号。

返回值:

无。

3.6.4 示例

3.6.4.1 PWM 示例 1

```
int32_t  pwm_signal_test(uint32_t pwm_idx, uint8_t pwm_ch)
{
    int32_t ret;
    pwm_handle_t pwm_handle;

    pwm_handle = csi_pwm_initialize(pwm_idx);
```

(下页继续)

(续上页)

```
    if (pwm_handle == NULL) {
        printf("csi_pwm_initialize error\n");
        return -1;
    }

    ret = csi_pwm_config(pwm_handle, pwm_ch, 3000, 1500);

    if (ret < 0) {
        printf("csi_pwm_config error\n");
        return -1;
    }

    csi_pwm_start(pwm_handle, pwm_ch);
    mdelay(20);

    ret = csi_pwm_config(pwm_handle, pwm_ch, 200, 150);

    if (ret < 0) {
        printf("csi_pwm_config error\n");
        return -1;
    }

    mdelay(20);
    csi_pwm_stop(pwm_handle, pwm_ch);

    csi_pwm_uninitialize(pwm_handle);

    return 0;
}

int example_pwm(uint32_t pwm_idx, uint8_t pwm_pin)
{
    int32_t ret;
    ret = pwm_signal_test(pwm_idx, pwm_pin);

    if (ret < 0) {
        printf("pwm_signal_test error\n");
        return -1;
    }

    printf("pwm_signal_test OK\n");
```

(下页继续)

(续上页)

```
    return 0;
}

int main(void)
{
    return example_pwm(EXAMPLE_PWM_IDX, EXAMPLE_PWM_CH_IDX);
}
```

3.7 RTC

3.7.1 函数列表

- *csi_rtc_initialize*
- *csi_rtc_uninitialize*
- *csi_rtc_power_control*
- *csi_rtc_get_capabilities*
- *csi_rtc_set_time*
- *csi_rtc_get_time*
- *csi_rtc_start*
- *csi_rtc_stop*
- *csi_rtc_get_status*
- *csi_rtc_set_alarm*
- *csi_rtc_enable_alarm*

3.7.2 简要说明

RTC(Real_Time Clock) 实时时钟为系统提供可靠的时间基准, 一般有独立的晶振和电源，保证主电源掉电时还可以工作。RTC 一般可以提供日历格式的时间。

RTC 也可以提供定时中断功能，用于实现定时器功能或者定时唤醒系统的功能。

3.7.3 接口描述

3.7.3.1 csi_rtc_initialize

```
rtc_handle_t csi_rtc_initialize(int32_t idx, rtc_event_cb_t cb_event)
```

功能描述:

通过索引号初始化对应的 rtc 实例，返回 rtc 实例的句柄。

参数:

idx: rtc 实例的索引号。

cb_event: rtc 实例的事件回调函数（一般在中断上下文执行）。回调函数原型定义见 rtc_event_cb_t。

回调函数类型 rtc_event_cb_t 定义如下:

```
typedef void (*rtc_event_cb_t)(int32_t idx, rtc_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型。

rtc 回调时间枚举类型见*rtc_evnet_e* 定义

返回值:

NULL: 初始化失败。

其它: 实例句柄。

rtc_evnet_e:

名字	定义	备注
RTC_EVENT_TIMER_INTRERRUPT	产生中断事件	

3.7.3.2 csi_rtc_uninitialize

```
int32_t csi_rtc_uninitialize(rtc_handle_t handle)
```

功能描述:

rtc 实例反初始化。该接口会停止 rtc 实例的工作，并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.7.3.3 csi_rtc_power_control

```
int32_t csi_rtc_power_control(rtc_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置 rtc 实例的功耗模式。

参数:

handle: 实例句柄。

state: rtc 的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.7.3.4 csi_rtc_get_capabilities

```
rtc_capabilities_t csi_rtc_get_capabilities(int32_t idx)
```

功能描述:

获取 rtc 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 rtc 能力的结构体，rtc 的能力定义见*rtc_capabilities_t* 。

rtc_capabilities_t:

名字	定义	备注
interrupt_mode :1	支持中断模式	
wrap_mode :1	支持循环模式	

3.7.3.5 csi_rtc_set_time

```
int32_t csi_rtc_set_time(rtc_handle_t handle, const struct tm *rtctime)
```

功能描述:

设置 rtc 时间。

参数:

handle: 实例句柄。

rtctime: RTC 时间，参看*struct tm* 的定义。

返回值:

错误码。

struct tm:

名字	定义	备注
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	天 - 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，其值从 0 开始，代表 1900 年	

3.7.3.6 csi_rtc_get_time

```
int32_t csi_rtc_get_time(rtc_handle_t handle, struct tm *rtctime)
```

功能描述:

获取 rtc 时间。

参数:

handle: 实例句柄。

rtctime: RTC 时间，参看*struct tm* 的定义。

返回值:

错误码。

3.7.3.7 csi_rtc_start

```
int32_t csi_rtc_start(rtc_handle_t handle)
```

功能描述:

rtc 计时开始。

参数:

handle: 实例句柄。

返回值:

错误码。

3.7.3.8 csi_rtc_stop

```
int32_t csi_rtc_stop(rtc_handle_t handle)
```

功能描述:

rtc 计时停止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.7.3.9 csi_rtc_get_status

```
rtc_status_t csi_rtc_get_status(rtc_handle_t handle)
```

功能描述:

获取当前时刻 rtc 的状态。

参数:

handle: 实例句柄。

返回值:

rtc 状态的结构体，rtc 的状态定义见*rtc_status_t* 。

rtc_status_t:

名字	定义	备注
active :1	RTC 运行状态	

3.7.3.10 csi_rtc_set_alarm

```
int32_t csi_rtc_set_alarm(rtc_handle_t handle, const struct tm *rtctime)
```

功能描述:

配置 rtc 实例的闹钟日期。

参数:

handle: 实例句柄。

rtctime: rtc 日期，参看*struct tm* 的定义。

返回值:

错误码。

3.7.3.11 csi_rtc_enable_alarm

```
int32_t csi_rtc_enable_alarm(rtc_handle_t handle, uint8_t flag)
```

功能描述:

rtc 闹钟上报使能。

参数:

handle: 实例句柄。

flag: 1-使能，0-禁止。

返回值:

错误码。

3.7.4 示例

3.7.4.1 RTC 示例 1

```c
static rtc_handle_t rtc_handle;
static volatile uint8_t cb_rtc_flag;

extern void mdelay(uint32_t ms);
#define RTC_TIME_SECS 5

void rtc_event_cb_fun(int32_t idx, rtc_event_e event)
{
    if (event == RTC_EVENT_TIMER_INTRERRUPT) {
        cb_rtc_flag = 1;
    }
}
void example_main(void)
{
    int32_t ret;
    struct tm current_time, last_time;
    uint32_t secs = 0;

    rtc_capabilities_t cap;

    //get rtc capabilities
    cap = csi_rtc_get_capabilities(0);
    printf("rtc %s interrupt mode\n",cap.interrupt_mode==1 ? "support":"not support");
    printf("rtc %s wrap mode\n",cap.wrap_mode==1 ? "support":"not support");

    rtc_handle = csi_rtc_initialize(0, rtc_event_cb_fun);

    if (rtc_handle == NULL) {
        printf("csi_rtc_initialize error\n");
        return;
    }
    ret = csi_rtc_start(rtc_handle);

    if (ret < 0) {
        printf("csi_rtc_start error\n");
        return;
    }



    current_time.tm_sec     = 55;
    current_time.tm_min     = 59;
    current_time.tm_hour    = 23;
    current_time.tm_mday    = 28;
    current_time.tm_mon     = 1;
```

(下页继续)

(续上页)

```
    current_time.tm_year    = 100;

    ret = csi_rtc_set_time(rtc_handle, &current_time);

    if (ret < 0) {
        printf("csi_rtc_set_time error\n");
        return;
    }

    mdelay(RTC_TIME_SECS * 1000);
    ret = csi_rtc_get_time(rtc_handle, &last_time);

    if (ret < 0) {
        printf("csi_rtc_get_time error\n");
        return ;
    }

    if (current_time.tm_sec != last_time.tm_sec) {
        secs += (last_time.tm_sec - current_time.tm_sec);
    }

    if (current_time.tm_min != last_time.tm_min) {
        secs += (last_time.tm_min - current_time.tm_min) * 60;
    }

    if (current_time.tm_hour != last_time.tm_hour) {
        secs += (last_time.tm_hour - current_time.tm_hour) * 60 * 60;
    }

    if (current_time.tm_mday != last_time.tm_mday) {
        secs += (last_time.tm_mday - current_time.tm_mday) * 60 * 60 * 24;
    }

    if ((secs <= (RTC_TIME_SECS + 1)) && (secs >= (RTC_TIME_SECS - 1))) {
        last_time.tm_year = last_time.tm_year + 1900;
        last_time.tm_mon = last_time.tm_mon + 1;
        printf("The time is %d-%d-%d %d:%d:%d\n", last_time.tm_year, last_time.tm_mon, last_time.
↪tm_mday, last_time.tm_hour, last_time.tm_min, last_time.tm_sec);
    } else {
        printf("get rtc timer error\n");
        return ;
    }
    ret = csi_rtc_stop(rtc_handle);
```

(下页继续)

(续上页)

```
    if (ret < 0) {
        printf("csi_rtc_stop error\n");
        return;
    }
    ret = csi_rtc_uninitialize(rtc_handle);

    if (ret < 0) {
        printf("csi_rtc_uninitialize error\n");
        return;
    }
}
```

3.7.4.2 RTC 示例 2

```
static rtc_handle_t rtc_handle;
static volatile uint8_t cb_rtc_flag;

extern void mdelay(uint32_t ms);
#define RTC_TIME_SECS 5
#define RTC_TIMEOUT_SECS 15
#define RTC_TIMEOUT2_SECS 1

void rtc_event_cb_fun(int32_t idx, rtc_event_e event)
{
    if (event == RTC_EVENT_TIMER_INTRERRUPT) {
        cb_rtc_flag = 1;
    }
}
void example_main(void)
{
    int32_t ret;
    struct tm current_time,last_time,set_time;
    uint32_t secs = 0;

    rtc_handle = csi_rtc_initialize(0, rtc_event_cb_fun);

    if (rtc_handle == NULL) {
        printf("csi_rtc_initialize error\n");
        return;
    }
    ret = csi_rtc_start(rtc_handle);
```

(下页继续)

(续上页)

```
if (ret < 0) {
    printf("csi_rtc_start error\n");
    return;
}

current_time.tm_sec     = 55;
current_time.tm_min     = 59;
current_time.tm_hour    = 23;
current_time.tm_mday    = 28;
current_time.tm_mon     = 1;
current_time.tm_year    = 100;

ret = csi_rtc_set_time(rtc_handle, &current_time);

if (ret < 0) {
    printf("csi_rtc_set_time error\n");
    return;
}

mdelay(RTC_TIME_SECS * 1000);

set_time.tm_sec     = 10;
ret = csi_rtc_set_alarm(rtc_handle, &set_time);

if (ret < 0) {
    printf("csi_rtc_set_timeout error\n");
    return;
}

printf("test rtc timeout %ds\n", RTC_TIMEOUT_SECS);

ret = csi_rtc_enable_alarm(rtc_handle, 1);

if (ret < 0) {
    printf("csi_rtc_enable_alarm error\n");
    return;
}
while( csi_rtc_get_status(rtc_handle).active);

mdelay(RTC_TIMEOUT2_SECS * 1000  + 2000);

if (cb_rtc_flag == 1) {
```

(下页继续)

(续上页)

```
        ret = csi_rtc_get_time(rtc_handle, &last_time);

        if (ret < 0) {
            printf("csi_rtc_get_time error\n");
            return ;
        }

        if (set_time.tm_sec != last_time.tm_sec) {
            secs += (last_time.tm_sec - set_time.tm_sec);
        }

        if (set_time.tm_min != last_time.tm_min) {
            secs += (last_time.tm_min - set_time.tm_min) * 60;
        }

        if (set_time.tm_hour != last_time.tm_hour) {
            secs += (last_time.tm_hour - set_time.tm_hour) * 60 * 60;
        }

        if (set_time.tm_mday != last_time.tm_mday) {
            secs += (last_time.tm_mday - set_time.tm_mday) * 60 * 60 * 24;
        }

        last_time.tm_year = last_time.tm_year + 1900;
        last_time.tm_mon = last_time.tm_mon + 1;
        printf("The time is %d-%d-%d %d:%d:%d\n", last_time.tm_year, last_time.tm_mon, last_time.
↪tm_mday, last_time.tm_hour, last_time.tm_min, last_time.tm_sec);
        ret = csi_rtc_enable_alarm(rtc_handle, 0);

        if (ret < 0) {
            printf("csi_rtc_enable_alarm error\n");
            return ;
        }

        return;
    } else {
        return ;
    }
    ret = csi_rtc_stop(rtc_handle);

    if (ret < 0) {
        printf("csi_rtc_stop error\n");
```

(下页继续)

(续上页)

```
        return;
    }
    ret = csi_rtc_uninitialize(rtc_handle);

    if (ret < 0) {
        printf("csi_rtc_uninitialize error\n");
        return;
    }
}
```

3.8 WatchDog

3.8.1 函数列表

- *csi_wdt_initialize*
- *csi_wdt_uninitialize*
- *csi_wdt_power_control*
- *csi_wdt_set_timeout*
- *csi_wdt_start*
- *csi_wdt_stop*
- *csi_wdt_restart*
- *csi_wdt_read_current_value*

3.8.2 简要说明

WatchDog（看门狗）本质上时一个定时器，这个定时器可用来监控程序的运行。看门狗可以设置一个预定的时间，在指定时间内若未对定时器进行喂狗操作将会导致系统复位。程序设计时通过在程序的关键点预埋喂狗动作，当程序由于某种原因（软件或硬件故障）未按指定的逻辑运行时可复位系统，保证系统的可用性。

3.8.3 接口描述

3.8.3.1 csi_wdt_initialize

```
wdt_handle_t csi_wdt_initialize(int32_t idx, wdt_event_cb_t cb_event)
```

功能描述:

通过索引号初始化对应的 wdt 实例，返回 wdt 实例的句柄。

参数:

idx: 设备号。

cb_event: wdt 实例的事件回调函数（一般在中断上下文执行）。回调函数原型定义见 wdt_event_cb_t。

回调函数类型 wdt_event_cb_t 定义如下：

```
typedef void (*wdt_event_cb_t)(int32_t idx, wdt_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型。

wdt 回调事件枚举类型见*wdt_event_e* 定义。

返回值:

wdt_event_e:

名字	定义	备注
WDT_EVENT_TIMEOUT	中断触发事件	

3.8.3.2 csi_wdt_uninitialize

```
int32_t csi_wdt_uninitialize(wdt_handle_t handle)
```

功能描述:

wdt 实例反初始化。该接口会停止 wdt 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.8.3.3 csi_wdt_power_control

```
int32_t csi_wdt_power_control(wdt_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置 iic 实例的功耗模式。

参数:

handle: 实例句柄。

state: wdt 的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.8.3.4 csi_wdt_set_timeout

```
int32_t csi_wdt_set_timeout(wdt_handle_t handle, uint32_t value)
```

功能描述:

wdt 超时设置。

参数:

handle: 实例句柄。

value: wdt 超时时间，单位是毫秒。

返回值:

错误码。

3.8.3.5 csi_wdt_start

```
int32_t csi_wdt_start(wdt_handle_t handle)
```

功能描述:

wdt 计时启动。

参数:

handle: 实例句柄。

返回值:

错误码。

3.8.3.6 csi_wdt_stop

```
int32_t csi_wdt_stop(wdt_handle_t handle)
```

功能描述:

wdt 计时停止。

参数:

handle: 实例句柄。

返回值:

错误码。

3.8.3.7 csi_wdt_restart

```
int32_t csi_wdt_restart(wdt_handle_t handle)
```

功能描述:

wdt 计时重启。

参数:

handle: 实例句柄。

返回值:

错误码。

3.8.3.8 csi_wdt_read_current_value

```
int32_t csi_wdt_read_current_value(wdt_handle_t handle, uint32_t *value)
```

功能描述:

获得当前 wdt 计时值。

参数:

handle: 实例句柄。
value: 用于返回当前计时值。

返回值:

错误码。

3.8.4 示例

WDT 示例 1

```
#define WDT_TIMEOUT 10000
static wdt_handle_t wdt_handle;

static void wdt_event_cb_fun(int32_t idx, wdt_event_e event)
{
    //do your job here according to event
}

void example_main(void)
{
    int32_t ret;

    wdt_handle = csi_wdt_initialize(0, wdt_event_cb_fun);

    if (wdt_handle == NULL) {
        printf("csi_wdt_initialize error\n");
        return;
    }

    ret = csi_wdt_set_timeout(wdt_handle, WDT_TIMEOUT);

    if (ret < 0) {
        printf("csi_wdt_set_timeout error\n");
        return ;
    }
    ret = csi_wdt_start(wdt_handle);

    if (ret < 0) {
        return;
    }

    int i;
    // feedog use restart function
    for (i = 0; i < 10; i++) {
        mdelay(WDT_TIMEOUT - 10);
        ret = csi_wdt_restart(wdt_handle);

        if (ret < 0) {
            return ;
        }
    }
```

(下页继续)

(续上页)

```
    uint32_t value;
    csi_wdt_read_current_value(wdt_handle, &value);

    ret = csi_wdt_stop(wdt_handle);

    if (ret < 0) {
        printf("csi_wdt_stop error\n");
        return ;
    }

    ret = csi_wdt_uninitialize(wdt_handle);

    if (ret < 0) {
        printf("csi_wdt_uninitialize error\n");
        return ;
    }
}
```

3.9 eFlash

3.9.1 函数列表

- *csi_eflash_initialize*
- *csi_eflash_uninitialize*
- *csi_eflash_get_capabilities*
- *csi_eflash_power_control*
- *csi_eflash_read*
- *csi_eflash_program*
- *csi_eflash_erase_sector*
- *csi_eflash_erase_chip*
- *csi_eflash_get_info*
- *csi_eflash_get_status*

3.9.2 简要说明

eFlash 闪存是嵌入式系统中常用的非易失存储器，支持代码片上执行。eFlash 在写入之前必须先执行擦除动作，擦除以块为单位。eFlash 的写入也叫编程，需要通过特殊的命令来实现。

3.9.3 接口描述

3.9.3.1 csi_eflash_initialize

```
eflash_handle_t csi_eflash_initialize(int32_t idx, eflash_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 eflash 实例，返回 eflash 实例的句柄。

参数:

idx: 设备号。

cb_event: eflash 实例的事件回调函数。回调函数原型定义见 eflash_event_cb_t。

回调函数类型 eflash_event_cb_t 定义如下：

```
typedef void (*eflash_event_cb_t)(int32_t idx, eflash_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，eflash 回调事件枚举类型。

事件类型见*eflash_event_e* 定义。

返回值:

eflash_event_e:

名字	定义	备注
EFLASH_EVENT_READY	eflash ready 事件	
EFLASH_EVENT_ERROR	eflash 错误事件	

3.9.3.2 csi_eflash_uninitialize

```
int32_t csi_eflash_uninitialize(eflash_handle_t handle)
```

功能描述:

eflash 实例反初始化。该接口会停止 eflash 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.9.3.3 csi_eflash_get_capabilities

```
eflash_capabilities_t csi_eflash_get_capabilities(int32_t idx)
```

功能描述:

获取 eflash 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 eflash 能力的结构体，eflash 的能力定义见*eflash_capabilities_t* 。

eflash_capabilities_t:

名字	定义	备注
event_ready :1	支持 eventready	
data_width :1	支持的数据宽度	
erase_chip :1	支持整片擦除	

3.9.3.4 csi_eflash_power_control

```
int32_t csi_eflash_power_control(eflash_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置 eflash 实例的功耗模式。

参数:

handle: 实例句柄。
state: eflash 的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.9.3.5 csi_eflash_read

```
int32_t csi_eflash_read(eflash_handle_t handle, uint32_t addr, void *data, uint32_t cnt)
```

功能描述:

从 eflash 读数据。

参数:

handle: 实例句柄。

addr: 待读取的 eflash 地址。

data: 待接收数据的缓冲区地址。

cnt: 读取的数据的长度。

返回值:

错误码。

3.9.3.6 csi_eflash_program

```
int32_t csi_eflash_program(eflash_handle_t handle, uint32_t addr, const void *data, uint32_t cnt)
```

功能描述:

往 eflash 写数据。

参数:

handle: 实例句柄。

addr: 写入的 eflash 地址。

data: 待烧写数据的缓冲区地址。

cnt: 数据的长度。

返回值:

错误码。

3.9.3.7 csi_eflash_erase_sector

```
int32_t csi_eflash_erase_sector(eflash_handle_t handle, uint32_t addr)
```

功能描述:

以 sector 擦除 eflash 的数据。

参数:

handle: 实例句柄。

addr: 要擦除 eflash 地址。

返回值:

错误码。

3.9.3.8 csi_eflash_erase_chip

```
int32_t csi_eflash_erase_chip(eflash_handle_t handle)
```

功能描述:

擦除整片 eflash 的数据。

参数:

handle: 实例句柄。

返回值:

错误码。

3.9.3.9 csi_eflash_get_info

```
eflash_status_t csi_eflash_get_info(eflash_handle_t handle)
```

功能描述:

获取当前时刻 eflash 的信息。

参数:

handle: 实例句柄。

返回值:

eflash 信息的结构体，eflash 的状态定义见*eflash_info_t* 。

eflash_info_t:

名字	定义	备注
uint32_t start	开始地址	
uint32_t end	末尾地址	
uint32_t sector_count	sector 个数	
uint32_t sector_size	sector 的大小	
uint32_t page_size	page 的大小	
uint32_t program_unit	写入的最小单位	
uint32_t erased_value	擦除完成的值	

3.9.3.10 csi_eflash_get_status

```
eflash_status_t csi_eflash_get_status(eflash_handle_t handle)
```

功能描述:

获取当前时刻 eflash 的状态。

参数:

`handle`: 实例句柄。

返回值:

eflash 状态的结构体，eflash 的状态定义见*eflash_status_t* 。

eflash_status_t:

名字	定义	备注
busy：1	状态为忙	
error：1	写入/擦除错误	

3.9.4 示例

3.9.4.1 eFlash 示例 1

```
static eflash_handle_t eflash;

#define ELFASH_START_ADDR   0x10000000
#define EFLASH_READ_LEN   0x200
#define EFLASH_WRITE_LEN  0x200

void example_main(void)
{
    int32_t ret;
    uint32_t addr = ELFASH_START_ADDR;
    uint8_t read_data[EFLASH_READ_LEN]={0};
    uint32_t cnt = EFLASH_READ_LEN;

    eflash_capabilities_t cap;

    //get eflash capabilities
    cap = csi_eflash_get_capabilities(0);
    printf("eflash %s erase by chip \n",cap.erase_chip==1 ? "support":"not support");

    //initialize eflash by idx
    eflash = csi_eflash_initialize(0, NULL);
    if (eflash == NULL) {
        //fail
        return;
    }
    ret = csi_eflash_power_control(eflash, DRV_POWER_FULL);
    if (ret < 0) {
        // power control failed
        return;
    }
    //get the eflash information
    eflash_info_t *info=NULL;
    info = csi_eflash_get_info(eflash);
    printf("the eflash sector_count is %x \n sector size is %x \n program uint is %x \n erased␣
↪value is %x\r\n", info->sector_count, info->sector_size, info->program_unit, info->erased_value);

    //erase eflash by chip
    ret = csi_eflash_erase_chip(eflash);
    if (ret < 0) {
        //failed
    }

    //get the status
    eflash_status_t status;
```

(下页继续)

(续上页)

```
    while(1) {
        status = csi_eflash_get_status(eflash);
        if (status.busy==0) {
            break;
        }
    }

    uint8_t write_data[EFLASH_WRITE_LEN]={0};
    memset(write_data, 0x5a, EFLASH_WRITE_LEN);
    //write data to the eflash addr
        ret = csi_eflash_program(eflash, addr, write_data, cnt);
        if (ret < 0) {
            //failed
    }

    //read data from the eflash addr
    ret = csi_eflash_read(eflash, addr, read_data, cnt);
    if (ret < 0) {
        //failed
    }

    //erase eflash by sector
    ret = csi_eflash_erase_sector(eflash, addr);
    if (ret < 0) {
        //failed
    }

    ret = csi_eflash_power_control(eflash, DRV_POWER_OFF);
    if (ret < 0) {
        // power control failed
        return;
    }

    //uninitialize eflash
    ret = csi_eflash_uninitialize(eflash);
    if (ret != 0) {
        //failed
    }
}
```

3.10 SPIFlash

3.10.1 函数列表

- *csi_spiflash_initialize*
- *csi_spiflash_uninitialize*
- *csi_spiflash_get_capabilities*
- *csi_spiflash_config_data_line*
- *csi_spiflash_read*
- *csi_spiflash_program*
- *csi_spiflash_erase_sector*
- *csi_spiflash_erase_chip*
- *csi_spiflash_power_down*
- *csi_spiflash_power_on*
- *csi_spiflash_get_info*
- *csi_spiflash_get_status*

3.10.2 简要说明

SPIFlash 闪存是嵌入式系统中常用的非易失存储器，支持代码片上执行。SPIFlash 在写入之前必须先执行擦除动作，擦除以块为单位。SPIFlash 的写入也叫编程，需要通过特殊的命令来实现。

3.10.3 接口描述

3.10.3.1 csi_spiflash_initialize

```
spiflash_handle_t csi_spiflash_initialize(int32_t idx, spiflash_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 spiflash 实例，返回 spiflash 实例的句柄。

参数:

`idx`: 设备号。

`cb_event`: spiflash 实例的事件回调函数。回调函数原型定义见 spiflash_event_cb_t。

回调函数类型 spiflash_event_cb_t 定义如下:

```
typedef void (*spiflash_event_cb_t)(int32_t idx, spiflash_event_e␣
↪event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，spiflash 回调事件枚举类型。

事件类型见*spiflash_event_e* 定义。

返回值:

spiflash_event_e:

名字	定义	备注
SPIFLASH_EVENT_READY	spiflash ready 事件	
SPIFLASH_EVENT_ERROR	spiflash 错误事件	

3.10.3.2 csi_spiflash_uninitialize

```
int32_t csi_spiflash_uninitialize(spiflash_handle_t handle)
```

功能描述:

spiflash 实例反初始化。该接口会停止 spiflash 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

`handle`: 实例句柄。

返回值:

错误码。

3.10.3.3 csi_spiflash_get_capabilities

```
spiflash_capabilities_t csi_spiflash_get_capabilities(int32_t idx)
```

功能描述:

获取 spiflash 实例支持的能力。

参数:

`idx`: 设备号。

返回值:

描述 spiflash 能力的结构体，spiflash 的能力定义见*spiflash_capabilities_t* 。

spiflash_capabilities_t:

名字	定义	备注
event_ready :1	支持 eventready	
data_width :2	支持的数据宽度	
erase_chip :1	支持整片擦除	

3.10.3.4 csi_spiflash_config_data_line

```
int32_t csi_spiflash_config_data_line(spiflash_handle_t handle, spiflash_data_line_e line)
```

功能描述:

配置 spiflash 实例的运行模式。

参数:

`handle`: 实例句柄。

`line`: spiflash 的运行模式，参看*spiflash_data_line_e* 的定义。

返回值:

错误码。

spiflash_data_line_e:

名字	定义	备注
SPIFLASH_DATA_1_LINE	spiflash 单线模式	
SPIFLASH_DATA_2_LINES	spiflash 双线事件	
SPIFLASH_DATA_4_LINES	spiflash 四线事件	

3.10.3.5 csi_spiflash_read

```
int32_t csi_spiflash_read(spiflash_handle_t handle, uint32_t addr, void *data, uint32_t cnt)
```

功能描述:

从 spiflash 读数据。

参数:

`handle`: 实例句柄。

`addr`: 待读取的 spiflash 地址。

`data`: 待接收数据的缓冲区地址。

`cnt`: 读取的数据的长度。

返回值:

错误码。

3.10.3.6 csi_spiflash_program

```
int32_t csi_spiflash_program(spiflash_handle_t handle, uint32_t addr, const void *data, uint32_t␣
↪cnt)
```

功能描述:

往 spiflash 写数据。

参数:

handle: 实例句柄。

addr: 写入的 spiflash 地址。

data: 待烧写数据的缓冲区地址。

cnt: 数据的长度。

返回值:

错误码。

3.10.3.7 csi_spiflash_erase_sector

```
int32_t csi_spiflash_erase_sector(spiflash_handle_t handle, uint32_t addr)
```

功能描述:

以 sector 擦除 spiflash 的数据。

参数:

handle: 实例句柄。

addr: 要擦除 spiflash 地址。

返回值:

错误码。

3.10.3.8 csi_spiflash_erase_chip

```
int32_t csi_spiflash_erase_chip(spiflash_handle_t handle)
```

功能描述:

擦除整片 spiflash 的数据。

参数:

handle: 实例句柄。

返回值:

错误码。

3.10.3.9 csi_spiflash_power_down

```
int32_t csi_spiflash_power_down(spiflash_handle_t handle)
```

功能描述:

断点 spiflash。

参数:

handle: 实例句柄。

返回值:

错误码。

3.10.3.10 csi_spiflash_power_on

```
int32_t csi_spiflash_power_on(spiflash_handle_t handle)
```

功能描述:

上电 spiflash。

参数:

handle: 实例句柄。

返回值:

错误码。

3.10.3.11 csi_spiflash_get_info

```
spiflash_info_t csi_spiflash_get_info(spiflash_handle_t handle)
```

功能描述:

获取当前时刻 spiflash 的信息。

参数:

handle: 实例句柄。

返回值:

spiflash 信息的结构体，spiflash 的状态定义见*spiflash_info_t* 。

spiflash_info_t:

名字	定义	备注
uint32_t start	开始地址	
uint32_t end	末尾地址	
uint32_t sector_count	sector 个数	
uint32_t sector_size	sector 的大小	
uint32_t page_size	page 的大小	
uint32_t program_unit	写入的最小单位	
uint32_t erased_value	擦除完成的值	

3.10.3.12 csi_spiflash_get_status

```
spiflash_status_t csi_spiflash_get_status(spiflash_handle_t handle)
```

功能描述:

获取当前时刻 spiflash 的状态。

参数:

handle: 实例句柄。

返回值:

spiflash 状态的结构体，spiflash 的状态定义见*spiflash_status_t* 。

spiflash_status_t:

名字	定义	备注
busy：1	状态为忙	
error：1	写入/擦除错误	

3.10.4 示例

3.10.4.1 SPIFlash 示例 1

```
static spiflash_handle_t spiflash;

#define SPIFLASH_START_ADDR   0x10000000
#define SPIFLASH_READ_LEN   0x200
#define SPIFLASH_WRITE_LEN  0x200

void example_main(void)
{
    int32_t ret;
    uint32_t addr = SPIFLASH_START_ADDR;
    uint8_t read_data[SPIFLASH_READ_LEN]={0};
    uint32_t cnt = SPIFLASH_READ_LEN;

    spiflash_capabilities_t cap;

    //get spiflash capabilities
    cap = csi_spiflash_get_capabilities(0);
    printf("spiflash %s erase by chip \n",cap.erase_chip==1 ? "support":"not support");

    //initialize spiflash by idx
    spiflash = csi_spiflash_initialize(0, NULL);
    if (spiflash == NULL) {
        //fail
        return;
    }
    //get the spiflash information
    spiflash_info_t *info=NULL;
    info = csi_spiflash_get_info(spiflash);
    printf("the spiflash sector_count is %x \n sector size is %x \n program uint is %x \n erased␣
↪value is %x\r\n", info->sector_count, info->sector_size, info->program_unit, info->erased_value);

    //erase spiflash by chip
```

(下页继续)

(续上页)

```
    ret = csi_spiflash_erase_chip(spiflash);
    if (ret < 0) {
        //failed
    }

    //get the status
    spiflash_status_t status;
    while(1) {
        status = csi_spiflash_get_status(spiflash);
        if (status.busy==0) {
            break;
        }
    }

    uint8_t write_data[SPIFLASH_WRITE_LEN]={0};
    memset(write_data, 0x5a, SPIFLASH_WRITE_LEN);
    //write data to the spiflash addr
        ret = csi_spiflash_program(spiflash, addr, write_data, cnt);
        if (ret < 0) {
            //failed
    }

    //read data from the spiflash addr
    ret = csi_spiflash_read(spiflash, addr, read_data, cnt);
    if (ret < 0) {
        //failed
    }

    //erase spiflash by sector
    ret = csi_spiflash_erase_sector(spiflash, addr);
    if (ret < 0) {
        //failed
    }

    //uninitialize spiflash
    ret = csi_spiflash_uninitialize(spiflash);
    if (ret != 0) {
        //failed
    }
}
```

3.11 I2S

3.11.1 函数列表

- *csi_i2s_initialize*
- *csi_i2s_uninitialize*
- *csi_i2s_get_capabilities*
- *csi_i2s_enable*
- *csi_i2s_config*
- *csi_i2s_send*
- *csi_i2s_receive*
- *csi_i2s_send_ctrl*
- *csi_i2s_receive_ctrl*
- *csi_i2s_get_status*
- *csi_i2s_power_control*

3.11.2 简要说明

I2S(Inter—IC Sound) 总线，又称集成电路内置音频总线，是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专门用于音频设备之间的数据传输，广泛应用于各种多媒体系统。I2S 总线接口通常为三线:I2S_SCLK,I2S_WS, I2S_SDA，也有四线接口: I2S_SCLK，I2S_WS, I2S_SDA, I2S_MCLK(用作 codec 的时钟源），以及五线双工接口:I2S_SCLK，I2S_WS, I2S_SDA, I2S_SDI,I2S_MCLK。

3.11.3 接口描述

3.11.3.1 csi_i2s_initialize

```
i2s_handle_t csi_i2s_initialize(int32_t idx, i2s_event_cb_t cb_event, void *cb_arg);
```

功能描述:

通过设备号初始化对应的 i2s 实例，返回 i2s 实例的句柄。

参数:

`idx`: 设备号。

`cb_event`: **i2s** 实例的事件回调函数 (一般在中断上下文执行)。回调函数原型定义见 i2s_event_cb_t。回调函数类型 i2s_event_cb_t 定义如下:

```
typedef void (*i2s_event_cb_t)(int32_t idx, i2s_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型。

i2s 回调事件枚举类型见 i2s_event_e 定义。

`cb_arg`: 回调函数用户参数。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

3.11.4 i2s_event_e:

名称	定义	备注
I2S_EVENT_SEND_COMPLETE	周期数据发送完成事件	（见 i2s_config_t 定义 tx_period）
I2S_EVENT_RECEIVE_COMPLETE	周期数据接收完成事件	（见 i2s_config_t 定义 rx_period）
I2S_EVENT_TX_UNDERFLOW	发送下溢事件	
I2S_EVENT_TX_OVERFLOW	接收溢出事件	
I2S_EVENT_FRAME_ERROR	传输帧格式错误事件	

3.11.4.1 csi_i2s_uninitialize

```
int32_t csi_i2s_uninitialize(i2s_handle_t handle)
```

功能描述:

i2s 实例反初始化。该接口会停止 i2s 实例正在进行的传输（如果有），并且释放相关的软硬件资源。

参数:

`handle`: 实例句柄。

返回值:

0: 正常

其他: 错误码。。

3.11.4.2 csi_i2s_enable

```
void csi_i2s_enable(i2s_handle_t handle, int en);
```

功能描述:

控制 i2s 模块使能，开启后 i2s 开始输出时钟信号。

参数:

handle: 实例句柄。

en: 1 使能，0 关闭

返回值:

0: 正常

其他: 错误码。。

3.11.4.3 csi_i2s_get_capabilities

```
i2s_capabilities_t csi_i2s_get_capabilities(int32_t idx)
```

功能描述:

获取 i2s 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 i2s 能力的结构体，i2s 功能定义见 i2s_capabilities_t 。

3.11.4.4 i2s_capabilities_t:

名称	定义	备注
uint32_t protocol_user : 1	支持自定义协议	
uint32_t protocol_i2s : 1	支持 i2s 协议	
uint32_t protocol_justified : 1	支持左右对齐协议	
uint32_t mono_mode : 1	支持单通道	
uint32_t mclk_pin : 1	支持提供主机时钟脚	
uint32_t event_frame_error : 1	支持帧格式错误事件上报	

3.11.4.5 csi_i2s_config

```
int32_t csi_i2s_config(i2s_handle_t handle, i2s_config_t *config);
```

功能描述:

配置 i2s 参数。

参数:

handle: 实例句柄。

config: 配置参数，见 i2s_config_t 定义。

返回值:

0: 正常

其他: 错误码。。

名称	定义	备注
cfg	i2s 协议配置参数，见 i2s_config_type_t 定义	
rate	采样率	
tx_period	i2s 传输完毕 tx_period bytes 数据，触发用户回调函数，上报 I2S_EVENT_SEND_COMPLETE 事件。	
rx_period	i2s 接收完毕 tx_period bytes 数据，触发用户回调函数，上报 I2S_EVENT_RECEIVE_COMPLETE 事件。	
tx_buf	发送缓存区	
tx_buf_length	发送缓存区长度（bytes）	
rx_buf	接收缓存区	
rx_buf_length	接收缓存区长度（bytes）	

3.11.4.6 csi_i2s_send

```
uint32_t csi_i2s_send(i2s_handle_t handle, const uint8_t *data, uint32_t length);
```

功能描述:

i2s 发送数据。

参数:

handle: 实例句柄。

data: 发送数据

length: 发送数据长度

返回值:

实际写入缓存区的数据长度, 例如:length=1000，实际缓存区空余 100bytes，则只可写入 100bytes 数据，返回值为 100。

3.11.4.7 csi_i2s_receive

```
uint32_t csi_i2s_receive(i2s_handle_t handle, uint8_t *buf, uint32_t length);
```

功能描述: i2s 接收数据。

参数:

handle: 实例句柄。
buf: 存储接收数据
length: 接收数据长度

返回值:

实际获取的数据长度, 例如:length=1000，实际缓存区拥有 100bytes，则只可读出 100bytes 数据，返回值为 100。

3.11.4.8 csi_i2s_send_ctrl

```
int32_t csi_i2s_send_ctrl(i2s_handle_t handle, i2s_ctrl_e cmd);
```

功能描述:

i2s 发送控制。

参数:

handle: 实例句柄。
cmd：控制指令，见 i2s_ctrl_e 定义

返回值:

0: 成功
其他: 错误码

3.11.4.9 csi_i2s_receive_ctrl

```
int32_t csi_i2s_receive_ctrl(i2s_handle_t handle, i2s_ctrl_e cmd);
```

功能描述:

i2s 接收控制。

参数:

handle: 实例句柄。

cmd：控制指令，见 i2s_ctrl_e 定义

返回值:

0: 成功

其他: 错误码

3.11.5 i2s_ctrl_e:

名称	定义	备注
I2S_STREAM_PAUSE	I2S 传输暂停（缓存区数据保持）	
I2S_STREAM_RESUME	I2S 传输解除暂停（从缓存区断点继续）	
I2S_STREAM_STOP	I2S 停止传输（缓存区数据清空）	
I2S_STREAM_START	I2S 启动传输	

3.11.5.1 csi_i2s_get_status

```
i2s_status_t csi_i2s_get_status(i2s_handle_t handle)
```

功能描述:

获取当前时刻 i2s 的状态。

参数:

handle: 实例句柄。

返回值:

i2s 状态的结构体，i2s 的状态定义见 i2s_status_t <i2s_status_t *。

3.11.6 i2s_status_t:

名称	定义	备注
int32_t tx_runing: 1	发送忙	
uint32_t rx_runing: 1	接收忙	
uint32_t tx_fifo_empty: 1	发送缓存区空	
uint32_t rx_fifo_full: 1	接收缓存区满，接收溢出	
uint32_t frame_error: 1	帧数据错误	

3.11.6.1 csi_i2s_power_control

```
int32_t csi_i2s_power_control(i2s_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。

state: 设备实例的功耗模式，参看 csi_power_stat_e 的定义。

返回值:

正常:0

其他: 错误码

3.11.7 csi_power_stat_e:

名字	定义	备注
DRV_POWER_OFF	关电源状态	
DRV_POWER_LOW	低电平状态	
DRV_POWER_FULL	全电源状态	
DRV_POWER_SUSPEND	挂起电源状态	

3.12 AES

3.12.1 函数列表

- *csi_aes_initialize*
- *csi_aes_uninitialize*
- *csi_aes_power_control*
- *csi_aes_get_capabilities*
- *csi_aes_config*
- *csi_aes_set_key*
- *csi_aes_ecb_crypto*
- *csi_aes_cbc_crypto*
- *csi_aes_cfb1_crypto*
- *csi_aes_cfb8_crypto*

- *csi_aes_cfb128_crypto*
- *csi_aes_ofb_crypto*
- *csi_aes_ctr_crypto*
- *csi_aes_get_status*

3.12.2 简要说明

AES（Advanced Encryption Standard）高级加密标准是一种对称秘钥加密算法，即加密的密钥和解密的秘钥相同。AES 采用对称分组密码体制，密钥的长度支持 128、192、256，分组长度 128 位。

3.12.3 接口描述

3.12.3.1 csi_aes_initialize

```
aes_handle_t csi_aes_initialize(int32_t idx, aes_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 aes 实例，返回 aes 实例的句柄。

参数:

idx: 设备号。

cb_event: aes 实例的事件回调函数。回调函数原型定义见 aes_event_cb_t。

回调函数类型 aes_event_cb_t 定义如下：

```
typedef void (*aes_event_cb_t)(int32_t idx, aes_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，aes 回调事件枚举类型*aes_event_e* 。

返回值:

NULL: 初始化失败。

其它: 初始化成功时的实例句柄。

aes_event_e:

名字	定义	备注
AES_EVENT_CRYPTO_COMPLETE	AES 计算完成事件	

3.12.3.2 csi_aes_uninitialize

```
int32_t csi_aes_uninitialize(aes_handle_t handle)
```

功能描述:

aes 实例反初始化。该接口会停止 aes 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.12.3.3 csi_aes_power_control

```
int32_t csi_aes_power_control(aes_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。
state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.12.3.4 csi_aes_get_capabilities

```
aes_capabilities_t csi_aes_get_capabilities(int32_t idx)
```

功能描述:

获取 aes 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 aes 能力的结构体，aes 的能力定义见*aes_capabilities_t* 。

aes_capabilities_t:

名字	定义	备注
uint32_t ecb_mode :1	支持 ecb 模式	
uint32_t cbc_mode :1	支持 cbc 模式	
uint32_t cfb1_mode :1	支持 cfb1 模式	
uint32_t cfb8_mode :1	支持 cfb8 模式	
uint32_t cfb128_mode :1	支持 cfb128 模式	
uint32_t ofb_mode :1	支持 ofb 模式	
uint32_t ctr_mode :1	支持 ctr 模式	
uint32_t bits_128 :1	支持 128bits 模式	
uint32_t bits_192 :1	支持 192bits 模式	
uint32_t bits_256 :1	支持 256bits 模式	

3.12.3.5 csi_aes_config

```
int32_t csi_aes_config(aes_handle_t handle,
                       aes_mode_e mode,
                       aes_key_len_bits_e keylen_bits,
                       aes_endian_mode_e endian)
```

功能描述:

配置 aes 实例的工作模式、key 长度及大小端模式。

参数:

`handle`: 实例句柄。

`mode`: aes 模式，参看*aes_mode_e* 定义。

`keylen_bits`: 密钥的长度，参看*aes_key_len_bits_e* 。

`endian`: aes 的大小端模式，参看*aes_endian_mode_e* 的定义。

返回值:

错误码。

aes_mode_e:

名字	定义	备注
AES_MODE_ECB	ECB 模式	
AES_MODE_CBC	CBC 模式	
AES_MODE_CFB1	CFB1 模式	
AES_MODE_CFB8	CFB8 模式	
AES_MODE_CFB128	CFB128 模式	
AES_MODE_OFB	OFB 模式	
AES_MODE_CTR	CTR 模式	

aes_key_len_bits_e:

名字	定义	备注
AES_KEY_LEN_BITS_128	128bits 长度	
AES_KEY_LEN_BITS_192	192bits 长度	
AES_KEY_LEN_BITS_256	256bits 长度	

aes_endian_mode_e:

名字	定义	备注
AES_ENDIAN_LITTLE	AES 小端模式	
AES_ENDIAN_BIG	AES 大端模式	

3.12.3.6 csi_aes_set_key

```
int32_t csi_aes_set_key(aes_handle_t handle, void *context, void *key, aes_key_len_bits_e key_len,␣
↪aes_crypto_mode_e enc)
```

功能描述:

设置 aes 的秘钥。

参数:

`handle`: 实例句柄。

`context`: aes 的 context 的缓冲区。

`key`: 秘钥的缓冲区地址。

`key_len`: 待输入的秘钥长度。

`enc`: 加解密操作，参见*aes_crypto_mode_e* 的定义。

返回值:

错误码。

aes_crypto_mode_e:

名字	定义	备注
AES_CRYPTO_MODE_ENCRYPT	AES 加密模式	
AES_CRYPTO_MODE_DECRYPT	AES 解密模式	

3.12.3.7 csi_aes_ecb_crypto

```
int32_t csi_aes_ecb_crypto(aes_handle_t handle, void *context, void *in, void *out, uint32_t len)
```

功能描述:

用 ECB 模式加解密。

参数:

handle: 实例句柄。

context: aes 的 context 的缓冲区。

in: 操作前数据的缓冲区地址。

out: 操作后数据的缓冲区地址。

len: 待输入的数据长度。

返回值:

错误码。

3.12.3.8 csi_aes_cbc_crypto

```
int32_t csi_aes_cbc_crypto(aes_handle_t handle, void *context, void *in, void *out, uint32_t len,␣
↪uint8_t iv[16])
```

功能描述:

操作 aes 通过之前设置的操作模式，执行 aes cbc 模式加解密。

参数:

handle: 实例句柄。

context: aes 的 context 的缓冲区。

in: 操作前数据的缓冲区地址。

out: 操作后数据的缓冲区地址。

len: 待输入的数据长度。

iv: 初始向量。

返回值:

错误码。

3.12.3.9 csi_aes_cfb1_crypto

```
int32_t csi_aes_cfb1_crypto(aes_handle_t handle, void *context, void *in, void *out,  uint32_t len,
↪ uint8_t iv[16])
```

功能描述:

操作 aes 通过之前设置的操作模式，执行 aes cfb1 模式加解密。

参数:

handle: 实例句柄。

context: aes 的 context 的缓冲区。

in: 操作前数据的缓冲区地址。

out: 操作后数据的缓冲区地址。

len: 待输入的数据长度。

iv: 初始向量。

返回值:

错误码。

3.12.3.10 csi_aes_cfb8_crypto

```
int32_t csi_aes_cfb8_crypto(aes_handle_t handle, void *context, void *in, void *out, uint32_t len,␣
↪uint8_t iv[16])
```

功能描述:

操作 aes 通过之前设置的操作模式，执行 aes cfb8 模式加解密。

参数:

handle: 实例句柄。

context: aes 的 context 的缓冲区。

in: 操作前数据的缓冲区地址。

out: 操作后数据的缓冲区地址。

len: 待输入的数据长度。

iv: 初始向量。

返回值:

错误码。

3.12.3.11 csi_aes_cfb128_crypto

```
int32_t csi_aes_cfb128_crypto(aes_handle_t handle, void *context, void *in, void *out, uint32_t␣
↪len, uint8_t iv[16], uint32_t *num)
```

功能描述:

操作 aes 通过之前设置的操作模式，执行 aes cfb128 模式加解密。

参数:

handle: 实例句柄。
context: aes 的 context 的缓冲区。
in: 操作前数据的缓冲区地址。
out: 操作后数据的缓冲区地址。
len: 待输入的数据长度。
iv: 初始向量。
num: 已经计算完成的字节数在一个 block 中的偏移。

返回值:

错误码。

3.12.3.12 csi_aes_ofb_crypto

```
int32_t csi_aes_ofb_crypto(aes_handle_t handle, void *context, void *in, void *out, uint32_t len,␣
↪uint8_t iv[16], uint32_t *num)
```

功能描述:

操作 aes 通过之前设置的操作模式，执行 aes ofb 模式加解密。

参数:

handle: 实例句柄。
context: aes 的 context 的缓冲区。
in: 操作前数据的缓冲区地址。
out: 操作后数据的缓冲区地址。
len: 待输入的数据长度。
iv: 初始向量。
num: 已经计算完成的字节数在一个 block 中的偏移。

返回值:

错误码。

3.12.3.13 csi_aes_ctr_crypto

```
int32_t csi_aes_ctr_crypto(aes_handle_t handle, void *context, void *in, void *out,
                           uint32_t len, uint8_t nonce_counter[16], uint8_t stream_block[16],␣
↪uint32_t *num)
```

功能描述:

操作 aes 通过之前设置的操作模式，执行 aes ctr 模式加解密。

参数:

`handle`: 实例句柄。

`context`: aes 的 context 的缓冲区。

`in`: 操作前数据的缓冲区地址。

`out`: 操作后数据的缓冲区地址。

`len`: 待输入的数据长度。

`nonce_counter`: 随机计数器的缓冲区地址。

`stream_block`: 随机计数器加密后的缓冲区地址。

`num`: 已经计算完成的字节数在一个 block 中的偏移。

返回值:

错误码。

3.12.3.14 csi_aes_get_status

```
aes_status_t csi_aes_get_status(aes_handle_t handle)
```

功能描述:

获取当前时刻 aes 的状态。

参数:

`handle`: 实例句柄。

返回值:

aes 状态的结构体，aes 的状态定义见*aes_status_t* 。

aes_status_t:

名字	定义	备注
uint32_t busy :1	计算忙	

3.12.4 示例

3.12.4.1 AES 示例 1

```
static aes_handle_t aes= NULL;

void example_main(void)
{
    const uint8_t in[16] = "Hello, World!";
    uint8_t out[16];
    int32_t ret;

    aes_capabilities_t cap;
    //get aescapabilities
    cap = csi_aes_get_capabilities(0);
    printf("aes %s cbc mode \n",cap.cbc_mode==1 ? "support":"not support");

    const uint8_t key[32] = "Demo-Key";
    //initialize aes by idx
    aes= csi_aes_initialize(0, NULL);
    if (aes== NULL) {
        //fail
        return;
    }

    //config aes mode 、key bits len 、endian mode
    ret = csi_aes_config(aes, AES_MODE_ECB, AES_KEY_LEN_BITS_256, AES_ENDIAN_LITTLE);
    if (ret < 0) {
        //fail
        return;
    }
    //set aes key
    ret = csi_aes_set_key(aes, NULL, (void *)key, AES_KEY_LEN_BITS_256, AES_CRYPTO_MODE_ENCRYPT);
    if (ret < 0) {
        //fail
        return;
    }
```

(下页继续)

(续上页)

```
    //start the aes operate
    ret = csi_aes_ecb_crypto(aes, NULL, (void *)in, (void *)out, 16);
    if (ret < 0) {
        //fail
        return;
    }

    while (1) {
        aes_status_t status = csi_aes_get_status(aes);
        if (status.busy ==0) {
            break;
        }
    }

    //uninitialize aes
    ret = csi_aes_uninitialize(aes);
    if(ret != 0) {
       //failed
    }
}
```

3.13 CRC

3.13.1 函数列表

- *csi_crc_initialize*
- *csi_crc_uninitialize*
- *csi_crc_power_control*
- *csi_crc_get_capabilities*
- *csi_crc_config*
- *csi_crc_calculate*
- *csi_crc_get_status*

3.13.2 简要说明

CRC(Cyclic Redundancy Check）循环冗余校验是一种根据数据产生简短固定位数校验码的一种散列函数，主要用来检测或校验数据传输或者保存后可能出现的错误。CRC 算法可采用不同的生成多项式版本。

3.13.3 接口描述

3.13.3.1 csi_crc_initialize

```
crc_handle_t csi_crc_initialize(int32_t idx, crc_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 crc 实例，返回 crc 实例的句柄。

参数:

idx: 设备号。

cb_event: crc 实例的事件回调函数。回调函数原型定义见 crc_event_cb_t。

回调函数类型 crc_event_cb_t 定义如下:

```
typedef void (*crc_event_cb_t)(int32_t idx, crc_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，crc 回调事件枚举类型*crc_event_e* 。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

crc_event_e:

名字	定义	备注
CRC_EVENT_CALCULATE_COMPLETE	计算完成事件	

3.13.3.2 csi_crc_uninitialize

```
int32_t csi_crc_uninitialize(crc_handle_t handle)
```

功能描述:

crc 实例反初始化。该接口会停止 crc 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.13.3.3 csi_crc_power_control

```
int32_t csi_crc_power_control(crc_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。

state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.13.3.4 csi_crc_get_capabilities

```
crc_capabilities_t csi_crc_get_capabilities(int32_t idx)
```

功能描述:

获取 crc 实例支持的能力。

参数:

idx:

返回值:

描述 crc 能力的结构体，crc 的能力定义见*crc_capabilities_t* 。

crc_capabilities_t:

名字	定义	备注
uint32_t ROHC :1	支持 ROHC 模式	
uint32_t MAXIM :1	支持 MAXIM 模式	
uint32_t X25 :1	支持 X25 模式	
uint32_t CCITT :1	支持 CCITT 模式	
uint32_t USB :1	支持 USB 模式	
uint32_t IBM :1	支持 IBM 模式	
uint32_t MODBUS :1	支持 MODBUS 模式	

3.13.3.5 csi_crc_config

```
int32_t csi_crc_config(crc_handle_t handle, crc_mode_e mode, crc_standard_crc_e standard)
```

功能描述:

配置 crc 实例的工作模式。

参数:

handle: 实例句柄。

mode: crc 模式，参看*crc_mode_e* 定义。

standard: crc 的标准，参看*crc_standard_crc_e* 的定义。

返回值:

错误码。

crc_mode_e:

名字	定义	备注
CRC_MODE_CRC8	CRC8 模式	
CRC_MODE_CRC16	CRC16 模式	
CRC_MODE_CRC32	CRC32 模式	

crc_standard_crc_e:

名字	定义	备注
CRC_STANDARD_CRC_ROHC	CRC 标准 RHOC	
CRC_STANDARD_CRC_MAXIM	CRC 标准 MAXIM	
CRC_STANDARD_CRC_X25	CRC 标准 X25	
CRC_STANDARD_CRC_CCITT	CRC 标准 CCITT	
CRC_STANDARD_CRC_USB	CRC 标准 USB	
CRC_STANDARD_CRC_IBM	CRC 标准 IBM	
CRC_STANDARD_CRC_MODBUS	CRC 标准 MODBUS	

3.13.3.6 csi_crc_calculate

```
int32_t csi_crc_calculate(crc_handle_t handle, const void *in, void *out, uint32_t len)
```

功能描述:

计算 crc，如果传入的数据长度不是字对齐，接口内部会以 0 填充。

参数:

handle: 实例句柄。

in: 待传入数据的缓冲区地址。

out: 待接收数据的缓冲区地址。

len: 待传入的数据的长度。

返回值:

错误码。

3.13.3.7 csi_crc_get_status

```
crc_status_t csi_crc_get_status(crc_handle_t handle)
```

功能描述:

获取当前时刻 CRC 的状态。

参数:

handle: 实例句柄。

返回值:

crc 状态的结构体，crc 的状态定义见*crc_status_t* 。

crc_status_t:

名字	定义	备注
uint32_t busy :1	计算忙	

3.13.4 示例

3.13.4.1 CRC 示例 1

```
static crc_handle_t crc= NULL;

void example_main(void)
{
    int ret;
    crc_status_t status;
```

(下页继续)

(续上页)

```
    crc_capabilities_t cap;
    //get crccapabilities
    cap = csi_crc_get_capabilities(0);
    printf("crc %s ROHC standard \n",cap.ROHC==1 ? "support":"not support");

    //initialize crc by idx
    crc= csi_crc_initialize(0, NULL);
    if (crc== NULL) {
        //fail
        return;
    }
    uint32_t crc_input[] = {0x44332211, 0x44332211, 0x44332211, 0x44332211};
    uint32_t expect_out = 0x8efa;
    uint32_t out;

    //config crc mode and standard
    ret = csi_crc_config(crc, CRC_MODE_CRC16, CRC_STANDARD_CRC_MODBUS);
    if (ret < 0){
        //fail
        return;
    }
    ret = csi_crc_calculate(crc, &crc_input, &out, 4);

    do {
        status = csi_crc_get_status(crc);
    } while (status.busy == 1);

    ret = csi_crc_uninitialize(crc);
    if (out != expect_out) {
        printf("crc MODBUS mode calculate failed!!!\n");
    } else {
        printf("crc MODBUS mode calculate success!!!\n");
    }
}
```

3.14 RSA

3.14.1 函数列表

- *csi_rsa_initialize*
- *csi_rsa_uninitialize*

- *csi_rsa_power_control*
- *csi_rsa_get_capabilities*
- *csi_rsa_config*
- *csi_rsa_encrypt*
- *csi_rsa_decrypt*
- *csi_rsa_sign*
- *csi_rsa_verify*
- *csi_rsa_get_status*

3.14.2 简要说明

RSA 公开密钥密码体制。所谓的公开密钥密码体制就是使用不同的加密密钥与解密密钥，是一种“由已知加密密钥推导出解密密钥在计算上是不可行的”密码体制。

3.14.3 接口描述

3.14.3.1 csi_rsa_initialize

```
rsa_handle_t csi_rsa_initialize(int32_t idx, rsa_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 rsa 实例，返回 rsa 实例的句柄。

参数:

`idx`: 设备号。

`cb_event`: rsa 实例的事件回调函数。回调函数原型定义见 rsa_event_cb_t。

回调函数类型 rsa_event_cb_t 定义如下：

```
typedef void (*rsa_event_cb_t)(int32_t idx, rsa_event_e event);
```

其中 event 为传给回调函数的事件类型，rsa 回调事件枚举见类型*rsa_event_e* 定义。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

rsa_event_e:

名字	定义	备注
RSA_EVENT_ENCRYPT_COMPLETE	RSA 加密完成事件	
RSA_EVENT_DECRYPT_COMPLETE	RSA 解密完成事件	
RSA_EVENT_SIGN_COMPLETE	RSA 签名完成事件	
RSA_EVENT_VERIFY_COMPLETE	RSA 校验完成事件	

3.14.3.2 csi_rsa_uninitialize

```
int32_t csi_rsa_uninitialize(rsa_handle_t handle)
```

功能描述:

rsa 实例反初始化。该接口会停止 rsa 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

`handle`: 实例句柄。

返回值:

错误码。

3.14.3.3 csi_rsa_power_control

```
int32_t csi_rsa_power_control(rsa_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

`handle`: 实例句柄。

`state`: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.14.3.4 csi_rsa_get_capabilities

```
rsa_capabilities_t csi_rsa_get_capabilities(int32_t idx)
```

功能描述:

获取 rsa 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 rsa 能力的结构体，rsa 的能力定义见*rsa_capabilities_t* 。

rsa_capabilities_t:

名字	定义	备注
uint32_t bits_192 :1	支持 192bits 模式	
uint32_t bits_256 :1	支持 256bits 模式	
uint32_t bits_512 :1	支持 512bits 模式	
uint32_t bits_1024 :1	支持 1024bits 模式	
uint32_t bits_2048 :1	支持 2048bits 模式	
uint32_t bits_3072 :1	支持 3072bits 模式	

3.14.3.5 csi_rsa_config

```
int32_t csi_rsa_config(rsa_handle_t handle,
                       rsa_data_bits_e data_bits,
                       rsa_endian_mode_e endian,
                       void *arg)
```

功能描述:

配置 rsa 实例的 bit 长度及大小端模式。

参数:

handle: 实例句柄。

data_bits: rsa 数据长度，参看*rsa_data_bits_e* 定义。

endian: rsa 的大小端模式，参看*rsa_endian_mode_e* 的定义。

arg: 传模值的地址。

返回值:

错误码。

rsa_data_bits_e:

名字	定义	备注
RSA_DATA_BITS_192	192bits 长度	
RSA_DATA_BITS_256	256bits 长度	
RSA_DATA_BITS_512	512bits 长度	
RSA_DATA_BITS_1024	1024bits 长度	
RSA_DATA_BITS_2048	2048bits 长度	
RSA_DATA_BITS_3072	3072bits 长度	

rsa_endian_mode_e:

名字	定义	备注
RSA_ENDIAN_MODE_LITTLE	RSA 小端模式	
RSA_ENDIAN_MODE_BIG	RSA 大端模式	

3.14.3.6 csi_rsa_encrypt

```
int32_t csi_rsa_encrypt(rsa_handle_t handle, void *n, void *e, void *src, uint32_t src_size,
                        void *out, uint32_t *out_size, rsa_padding_t padding)
```

功能描述:

rsa 加密。

参数:

`handle`: 实例句柄。

`n`: rsa 的模值的缓冲区地址。

`e`: rsa 公钥的缓冲区地址。

`src`: 明文的缓冲区地址。

`src_size`: 明文的长度。

`out`: 加密结果的缓冲区地址。

`out_size`: 加密数据的长度缓冲区地址。

`padding`: padding 模式，参见*rsa_padding_t* 。

返回值:

错误码。

rsa_padding_t:

名字	定义	备注
rsa_padding_type_e padding_type	Padding 类型	
rsa_hash_type_e hash_type	Padding 的哈希类型	

rsa_padding_type_e:

名字	定义	备注
RSA_PADDING_MODE_PKCS1	PKCS1 PADDING 模式	
RSA_PADDING_MODE_NO	NO PADDING 模式	
RSA_PADDING_MODE_SSLV23	SSLV23 PADDING 模式	
RSA_PADDING_MODE_PKCS1_OAEP	PKCS1_OAEP PADDING 模式	
RSA_PADDING_MODE_X931	X931 PADDING 模式	
RSA_PADDING_MODE_PSS	PSS PADDING 模式	

rsa_hash_type_e:

名字	定义	备注
RSA_HASH_TYPE_MD5	MD5 类型	
RSA_HASH_TYPE_SHA1	SHA1 类型	
RSA_HASH_TYPE_SHA224	SHA224 类型	
RSA_HASH_TYPE_SHA256	SHA256 类型	
RSA_HASH_TYPE_SHA384	SHA384 类型	
RSA_HASH_TYPE_SHA512	SHA512 类型	

3.14.3.7 csi_rsa_decrypt

```
int32_t csi_rsa_decrypt(rsa_handle_t handle, void *n, void *d, void *src, uint32_t src_size,
                        void *out, uint32_t *out_size, rsa_padding_t padding)
```

功能描述:

rsa 解密。

参数:

`handle`: 实例句柄。

`n`: rsa 的模值的缓冲区地址。

`d`: rsa 私钥的缓冲区地址。

`src`: 密文的缓冲区地址。

src_size: 密文的长度。

out: 解密结果的缓冲区地址。

out_size: 解密数据的长度缓冲区地址。

padding: padding 模式，参见*rsa_padding_t* 。

返回值:

错误码。

3.14.3.8 csi_rsa_sign

```
int32_t csi_rsa_sign(rsa_handle_t handle, void *n, void *d, void *src, uint32_t src_size,
                     void *signature, uint32_t *sig_size, rsa_padding_t padding)
```

功能描述:

rsa 签名。

参数:

handle: 实例句柄。

n: rsa 的模值的缓冲区地址。

d: rsa 私钥的缓冲区地址。

src: 输入数据的缓冲区地址。

src_size: 输入数据的长度。

signature: 签名结果的缓冲区地址。

sig_size: 签名结果的长度缓冲区地址。

padding: padding 模式，参见*rsa_padding_t* 。

返回值:

错误码。

3.14.3.9 csi_rsa_verify

```
int32_t csi_rsa_verify(rsa_handle_t handle, void *n, void *e, void *src, uint32_t src_size,
                       void *signature, uint32_t sig_size, void *result, rsa_padding_t padding)
```

功能描述:

rsa 验签。

参数:

handle: 实例句柄。

n: rsa 的模值的缓冲区地址。

e: rsa 公钥的缓冲区地址。

src: 输入数据的缓冲区地址。

src_size: 输入数据的长度。

signature: 签名结果的缓冲区地址。

sig_size: 签名结果的长度。

result: 验签的结果。

padding: padding 模式，参见*rsa_padding_t* 。

返回值:

错误码。

3.14.3.10 csi_rsa_get_status

```
rsa_status_t csi_rsa_get_status(rsa_handle_t handle)
```

功能描述:

获取当前时刻 rsa 的状态。

参数:

handle: 实例句柄。

返回值:

rsa 状态的结构体，rsa 的状态定义见*rsa_status_t* 。

rsa_status_t:

名字	定义	备注
uint32_t busy :1	计算忙	

3.15 SHA

3.15.1 函数列表

- *csi_sha_initialize*
- *csi_sha_uninitialize*

- *csi_sha_power_control*
- *csi_sha_get_capabilities*
- *csi_sha_config*
- *csi_sha_start*
- *csi_sha_update*
- *csi_sha_finish*
- *csi_sha_get_status*

3.15.2 简要说明

SHA（Secure Hash Algorithm）安全哈希算法是一个种散列算法，接收一段明文，然后以一种不可逆的方式将它转换成一段（通常更小）密文，也可以简单的理解为取一串输入码，并把它们转化为长度较短、位数固定的输出序列即散列值（也称为信息摘要或信息认证代码）的过程。可用于实现数字签名。

3.15.3 接口描述

3.15.3.1 csi_sha_initialize

```
sha_handle_t csi_sha_initialize(int32_t idx, void *context, sha_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 sha 实例，返回 sha 实例的句柄。

参数:

`idx`: 设别号。

`context`: 保存 sha context。

`cb_event`: sha 实例的事件回调函数。回调函数原型定义见 sha_event_cb_t。

回调函数类型 sha_event_cb_t 定义如下：

```
typedef void (*sha_event_cb_t)(int32_t idx, sha_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，sha 回调事件枚举类型见*sha_event_e* 定义。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

sha_event_e:

名字	定义	备注
SHA_EVENT_COMPLETE	计算完成事件	

3.15.3.2 csi_sha_uninitialize

```
int32_t csi_sha_uninitialize(sha_handle_t handle)
```

功能描述:

sha 实例反初始化。该接口会停止 sha 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.15.3.3 csi_sha_power_control

```
int32_t csi_sha_power_control(sha_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。

state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.15.3.4 csi_sha_get_capabilities

```
sha_capabilities_t csi_sha_get_capabilities(int32_t idx)
```

功能描述:

获取 sha 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 sha 能力的结构体，sha 的能力定义见*sha_capabilities_t* 。

sha_capabilities_t:

名字	定义	备注
uint32_t sha1 :1	支持 sha1 模式	
uint32_t sha224 :1	支持 sha224 模式	
uint32_t sha256 :1	支持 sha256 模式	
uint32_t sha384 :1	支持 sha384 模式	
uint32_t sha512 :1	支持 sha512 模式	
uint32_t sha512_224 :1	支持 sha512_224 模式	
uint32_t sha512_256 :1	支持 sha512_256 模式	
uint32_t endianmode :1	大小端模式	
uint32_t interruptmode :1	中断模式	

3.15.3.5 csi_sha_config

```
int32_t csi_sha_config(sha_handle_t handle,
                       sha_mode_e mode,
                       sha_endian_mode_e endian)
```

功能描述:

配置 sha 实例的工作模式及大小端模式。

参数:

handle: 实例句柄。

mode: sha 模式，参看*sha_mode_e* 定义。

endian: sha 大小端模式，参看*sha_endian_mode_e* 的定义。

返回值:

错误码。

sha_mode_e:

名字	定义	备注
SHA_MODE_1	SHA1 模式	
SHA_MODE_224	SHA224 模式	
SHA_MODE_256	SHA256 模式	
SHA_MODE_512	SHA512 模式	
SHA_MODE_384	SHA384 模式	
SHA_MODE_512_224	SHA512_224 模式	
SHA_MODE_512_256	SHA512_256 模式	

sha_endian_mode_e:

名字	定义	备注
SHA_ENDIAN_MODE_BIG	SHA 大端模式	
SHA_ENDIAN_MODE_LITTLE	SHA 小端模式	

3.15.3.6 csi_sha_start

```
int32_t csi_sha_start(sha_handle_t handle, void *context)
```

功能描述:

开始 sha 计算。

参数:

`handle`: 实例句柄。

`context`: sha 的 context 的缓冲区地址。

返回值:

错误码。

3.15.3.7 csi_sha_update

```
int32_t csi_sha_update(sha_handle_t handle, void *context, const void *input, uint32_t len)
```

功能描述:

更新 sha 的计算。

参数:

handle: 实例句柄。

context: sha 的 context 的缓冲区。

input: 待输入数据的缓冲区地址。

len : 待输入的数据的长度。

返回值:

错误码。

3.15.3.8 csi_sha_finish

```
int32_t csi_sha_finish(sha_handle_t handle, void *context, void *output)
```

功能描述:

计算 sha 的最后一次 hash。

参数:

handle: 实例句柄。

context: sha 的 context 的缓冲区地址。

output: 待接收的 hash 结果的缓冲区地址。

返回值:

错误码。

3.15.3.9 csi_sha_get_status

```
sha_status_t csi_sha_get_status(sha_handle_t handle)
```

功能描述:

获取当前时刻 sha 的状态。

参数:

handle: 实例句柄。

返回值:

sha 状态的结构体，sha 的状态定义见*sha_status_t* 。

sha_status_t:

名字	定义	备注
uint32_t busy :1	计算忙	

3.16 TRNG

3.16.1 函数列表

- *csi_trng_initialize*
- *csi_trng_uninitialize*
- *csi_trng_power_control*
- *csi_trng_get_capabilities*
- *csi_trng_get_data*
- *csi_trng_get_status*

3.16.2 简要说明

TRNG（True Random Number Generator）真随机数生成器是一种通过物理过程而不是计算机程序来生成随机数字的设备。

3.16.3 接口描述

3.16.3.1 csi_trng_initialize

```
trng_handle_t csi_trng_initialize(int32_t idx, trng_event_cb_t cb_event)
```

功能描述:

通过传入设备数初始化对应的 trng 实例，返回 trng 实例的句柄。

参数:

`handle`: 实例句柄。

`cb_event`: trng 实例的事件回调函数。回调函数原型定义见 trng_event_cb_t。

回调函数类型 trng_event_cb_t 定义如下:

```
typedef void (*trng_event_cb_t)(int32_t idx, trng_event_e event);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，trng 回调事件枚举类型见*trng_event_e*定义。

返回值:

NULL: 初始化失败。
其它: 实例句柄。

trng_event_e:

名字	定义	备注
TRNG_EVENT_DATA_GENERATECOMPLETE	随机数生成完事件	

3.16.3.2 csi_trng_uninitialize

```
int32_t csi_trng_uninitialize(trng_handle_t handle)
```

功能描述:

trng 实例反初始化。该接口会停止 trng 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.16.3.3 csi_trng_power_control

```
int32_t csi_trng_power_control(trng_handle_t handle, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

handle: 实例句柄。
state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.16.3.4 csi_trng_get_capabilities

```
trng_capabilities_t csi_trng_get_capabilities(int32_t idx)
```

功能描述:

获取 trng 实例支持的能力。

参数:

idx: 设备号。

返回值:

描述 trng 能力的结构体，trng 的能力定义见*trng_capabilities_t* 。

trng_capabilities_t:

名字	定义	备注
uint32_t lowper_mode :1	支持低功耗模式	

3.16.3.5 csi_trng_get_data

```
int32_t csi_trng_get_data(trng_handle_t handle, void *data, uint32_t num)
```

功能描述:

获取 trng 随机数。

参数:

handle: 实例句柄。

data: 待生成数据的缓冲区地址。

num: 待生成的数据的长度。

返回值:

错误码。

3.16.3.6 csi_trng_get_status

```
trng_status_t csi_trng_get_status(trng_handle_t handle)
```

功能描述:

参数:

handle: 实例句柄。

返回值:

trng 状态的结构体，trng 的状态定义见*trng_status_t* 。

trng_status_t:

名字	定义	备注
uint32_t busy :1	计算忙	
uint32_t data_valid :1	数据有效	

3.16.4 示例

3.16.4.1 TRNG 示例 1

```
static trng_handle_t trng= NULL;
#define NUM     10
void example_main(void)
{
    trng_capabilities_t cap;
    //get trng capabilities
    cap = csi_trng_get_capabilities(0);
    printf("trng %s lowper mode \n",cap.lowper_mode==1 ? "support":"not support");

    uint8_t  data[NUM] = {0x0};
    trng_status_t status;
    //initialize trng by idx
    trng= csi_trng_initialize(0, NULL);
    if (trng== NULL) {
        //fail
        return;
    }
    //get the data
    int32_t ret = csi_trng_get_data(trng, data, NUM);
```

(下页继续)

(续上页)

```
    if (ret <0) {
        //fail
        return;
    }
    while (1) {
        status = csi_trng_get_status(trng);
        if (status.busy == 0 && status.data_valid == 1) {
            break;
        }
    }
    ret = csi_trng_uninitialize(trng);
    if (ret < 0) {
        //fail
        return;
    }
}
```

3.17 DMA

3.17.1 函数列表

- *csi_dma_alloc_channel*
- *csi_dma_power_control*
- *csi_dma_get_capabilities*
- *csi_dma_release_channel*
- *csi_dma_config_channel*
- *csi_dma_start*
- *csi_dma_stop*
- *csi_dma_get_status*

3.17.2 简要说明

DMA(Direct Memory Access，直接内存存取) 它允许不同速度的硬件装置之间沟通，而不需要依赖于 CPU 的大量中断负载。

3.17.3 接口描述

3.17.3.1 csi_dma_alloc_channel

```
int32_t csi_dma_alloc_channel(void)
```

功能描述:

申请 dma 通道号。

参数:

无

返回值:

>=0: 通道号。
其它: 错误码。

3.17.3.2 csi_dma_power_control

```
int32_t csi_dma_power_control(int32_t ch, csi_power_stat_e state)
```

功能描述:

配置设备实例的功耗模式。

参数:

ch: 通道号。
state: 设备实例的功耗模式，参看*csi_power_stat_e* 的定义。

返回值:

错误码。

3.17.3.3 csi_dma_get_capabilities

```
dma_capabilities_t csi_dma_get_capabilities(int32_t ch)
```

功能描述:

获取 dma 通道实例支持的能力。

参数:

ch: 通道号。

返回值:

描述 dma 能力的结构体，dma 的能力定义见*dma_capabilities_t* 。

dma_capabilities_t:

名字	定义	备注
uint32_t unalign_addr : 1	支持不对齐地址传输模式（传输源是 memory）	

3.17.3.4 csi_dma_release_channel

```
void csi_dma_release_channel(int32_t ch)
```

功能描述:

释放通道号 ch。

参数:

ch: 通道号。

返回值:

无。

3.17.3.5 csi_dma_config_channel

```
int32_t csi_dma_config_channel(int32_t ch, dma_config_t *config, dma_event_cb_t cb_event, void *cb_
↪arg)
```

功能描述:

配置 dma 通道 ch。

参数:

ch: 通道号。

config: 具体配置项结构体。配置结构体原型定义见*dma_config_t* 。

cb_event: dma 通道 ch 的事件回调函数(一般在中断上下文执行)。回调函数原型定义见 dma_event_cb_t。

回调函数类型 dma_event_cb_t 定义如下:

```
typedef void (*dma_event_cb_t)(int32_t ch,  dma_event_e event, void *arg);
```

其中 ch 为通道号，event 为传给回调函数的事件类型。

dma 中 ch 回调事件枚举类型见*dma_event_e* 定义。

返回值:

错误码。

dma_event_e:

名字	定义	备注
DMA_EVENT_TRANSFER_DONE	传输完成事件	
DMA_EVENT_TRANSFER_HALF_DONE	传输一半完成事件	
DMA_EVENT_TRANSFER_MODE_DONE	传输完成事件（触发模式）	
DMA_EVENT_CHANNEL_PEND	通道挂起事件	
DMA_EVENT_TRANSFER_ERROR	传输错误事件	

dma_config_t:

名字	定义	备注
src_inc	源地址变化方式，参看的定义	
dst_inc	目的地址变化方式，参看 dma_addr_inc_e 的定义	
src_endian	源地址大小端模式，参看 dma_addr_endian_e 的定义	
dst_endian	目的地址大小端模式，参看 dma_addr_endian_e 的定义	
src_tw	源传输数据宽度	
dst_tw	目的传输数据宽度	
hs_if	硬件握手（可选）	
preemption	某通道抢占配置	
type	传输类型配置，参看 dma_trans_type_e 的定义	
mode	触发模式配置，参看 dma_trig_trans_mode_e 的定义	
ch_mode	通道申请使用模式配置，参看 dma_channel_req_mode_e 定义	
single_dir	单次传输方向枚举类型 dma_single_dir_e 定义	
group_len	组传输长度设置，当触发模式配置为 GROUP_TRIGGER 模式	

dma_addr_inc_e:

名字	定义	备注
DMA_ADDR_INC	地址递增方式	
DMA_ADDR_DEC	地址递减方式	
DMA_ADDR_CONSTANT	地址不变方式	

dma_trans_type_e:

名字	定义	备注
DMA_MEM2MEM	内存至内存传输类型	
DMA_MEM2PERH	内存至外设传输类型	
DMA_PERH2MEM	外设至内存传输类型	
DMA_PERH2PERH	外设至外设传输类型	

dma_trig_trans_mode_e:

名字	定义	备注
DMA_SINGLE_TRIGGER	单次触发模式	
DMA_GROUP_TRIGGER	组触发模式	
DMA_BLOCK_TRIGGER	块触发模式	

dma_single_dir_e:

名字	定义	备注
DMA_DIR_DEST	目的方向	
DMA_DIR_SOURCE	源方向	

dma_addr_endian_e:

名字	定义	备注
DMA_ADDR_LITTLE	小端模式	
DMA_ADDR_BIG	大端模式	

dma_channel_req_mode_e:

名字	定义	备注
DMA_MODE_HARDWARE	硬件模式	
DMA_MODE_SOFTWARE	软件模式	

3.17.3.6 csi_dma_start

```
void csi_dma_start(int32_t ch, void *psrcaddr, void *pdstaddr, uint32_t length)
```

功能描述:

dma 传输开始。

参数:

`ch`: 通道号。

`psrcaddr`: 通道 ch 进行 dma 传输的源地址。

`pdstaddr`: 通道 ch 进行 dma 传输的目的地址。

`length`: 通道 ch 进行 dma 传输的数据长度。

返回值:

无。

3.17.3.7 csi_dma_stop

```
void csi_dma_stop(int32_t ch)
```

功能描述:

dma 传输结束。

参数:

ch: 通道号。

返回值:

无。

3.17.3.8 csi_dma_get_status

```
dma_status_e csi_dma_get_status(int32_t ch)
```

功能描述:

获取当前时刻 ch 通道号状态。

参数:

ch: 通道号。

返回值:

dma 状态的枚举，dma 的状态定义见*dma_status_e* 。

dma_status_e:

名字	定义	备注
DMA_EVENT_TRANSFER_DONE	传输完成	
DMA_EVENT_TRANSFER_HALF_DONE	传输完成一半	
DMA_EVENT_TRANSFER_MODE_DONE	传输完成（某触发模式）	
DMA_EVENT_CAHNNEL_PEND	被抢占挂起状态	
DMA_EVENT_TRANSFER_ERROR	传输异常	

3.17.4 示例

3.17.4.1 DMA 示例 1

```
static volatile uint8_t dma_cb_flag = 0;
#ifndef DMA_M2M_SIZE
#define DMA_M2M_SIZE    512
#endif
static uint8_t p_src[DMA_M2M_SIZE] = {0};
static uint8_t p_dst[DMA_M2M_SIZE] = {0};

static void sleep(uint32_t k)
{
    int i, j;

    for (i = 0; i < 1000; i++) {
        for (j = 0; j < k; j++);
    }
}
static void dma_event_cb_fun(int32_t ch, dma_event_e event, void*arg)
{
    dma_cb_flag = 1;
}

static int32_t dma_test_mem2mem(dmac_handle_t dma_handle, uint8_t ch)
{
    uint32_t i;
    dma_config_t config;
    uint32_t ret;

    for (i = 0; i < DMA_M2M_SIZE; i++) {
        p_src[i] = i;
    }

    memset(p_dst, 0, DMA_M2M_SIZE);

    ch = csi_dma_alloc_channel(dma_handle, ch);

    if (ch < 0) {
        printf("csi_dma_alloc_channel error\n");
        return -1;
    }
    config.src_inc  = DMA_ADDR_INC;
    config.dst_inc  = DMA_ADDR_INC;
```

(下页继续)

(续上页)

```
config.src_endian = DMA_ADDR_LITTLE;
config.dst_endian = DMA_ADDR_LITTLE;
config.src_tw   = 1;
config.dst_tw   = 1;
config.group_len = 8;
config.mode = DMA_BLOCK_TRIGGER;
config.type     = DMA_MEM2MEM;
config.ch_mode  = DMA_MODE_SOFTWARE;
ret = csi_dma_config_channel(dma_handle, ch, &config, dma_event_cb_fun, NULL);

if (ret < 0) {
    printf("csi_dma_config_channel error\n");
    return 0;
}

ret = csi_dma_start(dma_handle, ch, p_src, p_dst, DMA_M2M_SIZE);

if (ret < 0) {
    printf("csi_dma_start error\n");
    return 0;
}

printf("sleep or do other things while data in transfermation using DMA\n");
sleep(1);

//while (csi_dma_get_status(dma_handle, ch) != DMA_STATE_DONE) ;
while (!dma_cb_flag);

for (i = 0; i < DMA_M2M_SIZE; i++) {
    if (p_dst[i] != p_src[i]) {
        return -1;
    }
}
ret = csi_dma_stop(dma_handle, ch);

if (ret < 0) {
    printf("csi_dma_stop error\n");
}
ret = csi_dma_release_channel(dma_handle, ch);

if (ret < 0) {
    printf("csi_dma_release_channel error\n");
    return 0;
```

(下页继续)

(续上页)

```
    }

    ret = csi_dma_uninitialize(dma_handle);

    if (ret < 0) {
        printf("csi_dma_uninitialize error\n");
        return 0;
    }

    printf("dma_mem2mem_test_func OK\n");
    return 0;
}
void example_dmac(void)
{
    int32_t ret;
    dmac_handle_t dma_handle = NULL;

    dma_handle = csi_dma_initialize(0);

    if (dma_handle == NULL) {
        printf("csi_dma_initialize error\n");
        return ;
    }

    ret = dma_test_mem2mem(dma_handle, 0);

    if (ret < 0) {
        printf("test dma mem to mem error\n");
        return ;
    }
}
```

3.18 PMU

3.18.1 函数列表

- *csi_pmu_initialize*
- *csi_pmu_uninitialize*
- *csi_pmu_enter_sleep*
- *csi_pmu_power_control*
- *csi_pmu_config_wakeup_source*

3.18.2 简要说明

PMU(power management unit) 电源管理单元，是一种高度集成的、针对便携式应用的电源管理方案，即将传统分立的若干类电源管理器件整合在单个的封装之内，这样可实现更高的电源转换效率和更低功耗，及更少的组件数以适应缩小的板级空间。

3.18.3 接口描述

3.18.3.1 csi_pmu_initialize

```
pmu_handle_t csi_pmu_initialize(int32_t idx, pmu_event_cb_t cb_event)
```

功能描述:

通过设备号初始化对应的 pmu 实例，返回 pmu 实例的句柄。

参数:

idx: 设备号。

cb_event: pmu 实例的事件回调函数（一般在进入低功耗模式上下文执行）。应用可根据回调函数做相应的睡前睡后行为。回调函数原型定义见 pmu_event_cb_t。

回调函数类型 pmu_event_cb_t 定义如下:

```
typedef void (*pmu_event_cb_t)(int32_t idx, pmu_event_e event, pmu_mode_e␣
↪mode);
```

其中 idx 为设备号，event 为传给回调函数的事件类型，mode 为传给回调函数的低功耗模式。

pmu 回调事件枚举类型见*pmu_event_e* 定义。

返回值:

NULL: 初始化失败。

其它: 实例句柄。

pmu_event_e:

名字	定义	备注
PMU_EVENT_PREPARE_SLEEP	睡前准备事件	
PMU_EVENT_PREPARE_DONE	睡前唤醒事件	

3.18.3.2 csi_pmu_uninitialize

```
int32_t csi_pmu_uninitialize(pmu_handle_t handle)
```

功能描述:

pmu 实例反初始化。该接口会释放相关的软硬件资源。

参数:

handle: 实例句柄。

返回值:

错误码。

3.18.3.3 csi_pmu_enter_sleep

```
int32_t csi_pmu_enter_sleep(pmu_handle_t handle, pmu_mode_e mode)
```

功能描述:

pmu 实例反初始化。该接口会释放相关的软硬件资源。

参数:

handle: 实例句柄。

mode: pmu 模式定义见*pmu_mode_e* 。

返回值:

错误码。

pmu_mode_e:

名字	定义	备注
PMU_MODE_RUN	运行模式	
PMU_MODE_SLEEP	睡眠模式（关闭 cpu clock）	
PMU_MODE_DOZE	停止模式（关闭 cpu 以及外设 clock）	
PMU_MODE_DORMANT	休眠模式（关闭 cpu 以及大部分外设电源）	
PMU_MODE_STANDBY	待命模式（关闭 cpu、大部分外设以及 ram 电源）	
PMU_MODE_SHUTDOWN	关机模式	

3.18.3.4 csi_pmu_power_control

```
int32_t csi_pmu_power_control(pmu_handle_t handle, csi_power_stat_e state)
```

功能描述:

保存和恢复 pmu 的寄存器。

参数:

handle: 实例句柄。

state: 电源状态。原型定义见*csi_power_stat_e* 。

返回值:

错误码。

csi_power_stat_e:

名字	定义	备注
DRV_POWER_OFF	关电源状态	
DRV_POWER_LOW	低电平状态	
DRV_POWER_FULL	全电源状态	
DRV_POWER_SUSPEND	挂起电源状态	

3.18.3.5 csi_pmu_config_wakeup_source

```
int32_t csi_pmu_config_wakeup_source(pmu_handle_t handle, uint32_t wakeup_num, pmu_wakeup_type_e␣
↪type, pmu_wakeup_pol_e pol, uint8_t enable)
```

功能描述:

配置 pmu 的唤醒源。

参数:

handle: 实例句柄。

wakeup_num: 唤醒号。

type: 唤醒源的信号类型，定义见 pmu_wakeup_type_e。

pol: 唤醒源的信号极性，定义见 pmu_wakeup_pol_e。

enable: 是否使能唤醒源。

返回值:

错误码。

pmu_wakeup_type_e:

名字	定义	备注
PMU_WAKEUP_TYPE_PULSE	脉冲类型	
PMU_WAKEUP_TYPE_LEVEL	电平类型	

pmu_wakeup_pol_e:

名字	定义	备注
PMU_WAKEUP_POL_LOW	低电平/下降沿有效	
PMU_WAKEUP_POL_HIGH	高电平/上升沿有效	

3.18.4 示例

3.18.4.1 PMU 示例 1

```
pmu_handle_t pmu_handle;
/* do console save and restore operation through console_handle */
extern usart_handle_t console_handle;

void manager_device_power(int32_t idx, pmu_event_e event, pmu_mode_e mode)
{
    if (event == PMU_EVENT_PREPARE_SLEEP) {
        csi_usart_power_control(console_handle, DRV_POWER_SUSPEND);
        csi_pmu_power_control(pmu_handle, DRV_POWER_SUSPEND);
    } else if (event == PMU_EVENT_SLEEP_DONE) {
        csi_usart_power_control(console_handle, DRV_POWER_FULL);
        csi_pmu_power_control(pmu_handle, DRV_POWER_FULL);
    }
}
int32_t test_pmu_dormant_mode(void)
{
    int32_t ret;

    printf("test dormant mode\n");
    pmu_handle = csi_pmu_initialize(0, manager_device_power);
    csi_gpio_pin_initialize(WAKEUP_PIN, NULL);

    if (pmu_handle == NULL) {
        printf("csi_pmu_initialize failed\n");
        return -1;
```

(下页继续)

(续上页)

```
    }

    ret = csi_pmu_config_wakeup_source(pmu_handle, EXAMPLE_WAKEUP_NUM, PMU_WAKEUP_TYPE_LEVEL, PMU_
↪WAKEUP_POL_HIGH, 1);

    if (ret < 0) {
        printf("csi_pmu_config_wakeup_source failed\n");
        return -1;
    }

    printf("please change the wakeup pin %s from low to high\n", EXAMPLE_BOARD_WAKEUP_PIN_NAME);

    ret = csi_pmu_enter_sleep(pmu_handle, PMU_MODE_DORMANT);

    if (ret < 0) {
        printf("csi_pmu_enter_sleep failed\n");
        return -1;
    }

    ret = csi_pmu_config_wakeup_source(pmu_handle, EXAMPLE_WAKEUP_NUM, PMU_WAKEUP_TYPE_LEVEL, PMU_
↪WAKEUP_POL_HIGH, 0);

    ret = csi_pmu_uninitialize(pmu_handle);

    if (ret < 0) {
        printf("csi_pmu_uninitialize failed\n");
        return -1;
    }

    printf("test standby mode successfully\n");

    return 0;
}

int example_pmu(void)
{
    drv_pinmux_config(WAKEUP_PIN, WAKEUP_PIN_FUNC);
    test_pmu_dormant_mode();
    return 0;
}

int main(void)
{
```

(下页继续)

(续上页)

```
    return example_pmu();
}
```

3.19 Mailbox

3.19.1 函数列表

- *csi_mailbox_initialize*
- *csi_mailbox_uninitialize*
- *csi_mailbox_send*
- *csi_mailbox_receive*

3.19.2 简要说明

Mailbox 提供了核间通讯的一种方式。

3.19.3 接口描述

3.19.3.1 csi_mailbox_initialize

```
mailbox_handle_t csi_mailbox_initialize(mailbox_event_cb_t cb_event)
```

功能描述:

通过索引号初始化对应的 Mailbox 实例，返回 Mailbox 实例的句柄。

参数:

`cb_event`: 中断回调函数。

返回值:

成功返回实例句柄，失败返回 NULL。

mailbox_event_cb_t:

```
typedef void (*mailbox_event_cb_t)(mailbox_handle_t handle, int32_t mailbox_id, uint32_t received_
↪len, mailbox_event_e event);
```

其中 handle 为初始化对应的句柄，mailbox_id 为此次事件对应的 mailbox 号，received_len 为接收到的实际字节数，event 为传给回调函数的事件类型。mailbox 回调事件枚举类型 mailbox_event_e 定义如下：

MAILBOX_EVENT_SEND_COMPLETE	数据发送完成事件
MAILBOX_EVENT_RECEIVED	数据接收并存在 MAILBOX FIFO，可调用 receive 函数去读取

3.19.3.2 csi_mailbox_uninitialize

```
int32_t csi_mailbox_uninitialize(mailbox_handle_t handle)
```

功能描述:

mailbox 实例反初始化，该接口会停止 mailbox 实例正在进行的工作（如果有），并且释放相关的软硬件资源。

参数:

`handle`: 实例句柄。

返回值:

错误码。

3.19.3.3 csi_mailbox_send

```
int32_t csi_mailbox_send(mailbox_handle_t handle, int32_t mailbox_id, const void *data, uint32_t␣
↪num)
```

功能描述:

发送消息给目标 mailbox。

参数:

`handle`: 实例句柄。

`mailbox_id`: 目标 mailbox 号。

`data`: 存放发送数据的地址。

`num`: 发送字节数。

返回值:

发送成功的字节数或者错误码。

3.19.3.4 csi_mailbox_receive

```
int32_t csi_mailbox_receive(mailbox_handle_t handle, int32_t mailbox_id, void *data, uint32_t num)
```

功能描述:

从目标 mailbox 接收数据。

参数:

`handle`: 实例句柄。

`mailbox_id`: 目标 mailbox 号。

`data`: 存放接收数据的地址。

`num`: 接收的字节数，传入的值为 callback 返回的实际接收值。

返回值:

接收成功的字节数或者错误码。

3.20 Codec

3.20.1 函数列表

- *csi_codec_init*
- *csi_codec_uninit*
- *csi_codec_power_control*
- *csi_codec_input_open*
- *csi_codec_input_config*
- *csi_codec_input_close*
- *csi_codec_input_read*
- *csi_codec_input_buf_avail*
- *csi_codec_input_buf_reset*
- *csi_codec_input_start*
- *csi_codec_input_stop*
- *csi_codec_input_pause*
- *csi_codec_input_resume*
- *csi_codec_input_set_digital_gain*
- *csi_codec_input_set_analog_gain*
- *csi_codec_input_get_digital_gain*
- *csi_codec_input_get_analog_gain*
- *csi_codec_input_set_mixer_gain*

- *csi_codec_input_get_mixer_gain*
- *csi_codec_input_mute*
- *csi_codec_output_open*
- *csi_codec_output_close*
- *csi_codec_output_config*
- *csi_codec_output_write*
- *csi_codec_output_buf_avail*
- *csi_codec_output_buf_reset*
- *csi_codec_output_start*
- *csi_codec_output_stop*
- *csi_codec_output_pause*
- *csi_codec_output_resume*
- *csi_codec_output_set_digital_left_gain*
- *csi_codec_output_set_digital_right_gain*
- *csi_codec_output_set_analog_left_gain*
- *csi_codec_output_set_analog_right_gain*
- *csi_codec_output_get_digital_left_gain*
- *csi_codec_output_get_digital_right_gain*
- *csi_codec_output_get_analog_left_gain*
- *csi_codec_output_get_analog_right_gain*
- *csi_codec_output_set_mixer_left_gain*
- *csi_codec_output_set_mixer_right_gain*
- *csi_codec_output_get_mixer_left_gain*
- *csi_codec_output_get_mixer_right_gain*
- *csi_codec_output_mute*

3.20.2 简要说明

codec 是音频编解码器，其具有音频采集（将模拟量转换为数字量）和音频播放功能（数字音频转换为模拟量），同时可以进行数据滤波、增益等。

3.20.3 接口描述

3.20.3.1 csi_codec_init

```
int32_t csi_codec_init(uint32_t idx);
```

功能描述:

通过设备号初始化对应的 codec 实例，获取 codec 资源。

参数:

idx: 设备号。

返回值:

0: 初始化成功。
其它: 错误码。

3.20.3.2 csi_codec_uninit

```
void csi_codec_uninit(uint32_t idx);
```

功能描述:

反初始化 codec 实例，释放 codec 资源。

参数:

idx: 设备号。

返回值:

无。

3.20.3.3 csi_codec_power_control

```
int32_t csi_codec_power_control(int32_t idx, csi_power_stat_e state);
```

功能描述:

codec 功率控制

参数:

idx: 设备号。
state: 功率模式，见 csi_power_stat_e 定义。

返回值:

0: 成功。
其他: 错误码

3.20.4 csi_power_stat_e:

名字	定义	备注
DRV_POWER_OFF	关电源状态	
DRV_POWER_LOW	低电平状态	
DRV_POWER_FULL	全电源状态	
DRV_POWER_SUSPEND	挂起电源状态	

3.20.4.1 csi_codec_input_open

```
int32_t csi_codec_input_open(codec_input_t *handle);
```

功能描述:

开启 codec 输入通路（录音通路）。

参数:

handle: ocdec

input 句柄，用于配置与控制 input 通路，在通路开启时，为通路分配资源。见 codec_input_t 定义。

返回值:

0: 成功。

其他: 错误码

3.20.4.2 csi_codec_input_close

```
int32_t csi_codec_input_close(codec_input_t *handle);
```

功能描述:

关闭 codec 输入通路（录音通路）。

参数:

handle: ocdec

input 句柄，用于配置与控制 input 通路，见 codec_input_t 定义。

返回值:

0: 成功。

其他: 错误码

3.20.5 codec_input_t:

名称	定义	备注
codec_idx	codec 设备号	
ch_idx	input 通道号码	
cb	事件回调函数	
cb_arg	事件回调函数用户参数	
buf	input 数据缓存区	
buf_size	input 数据缓存区大小	
period	每采集 period bytes 数据量，触发一次回调函数	
priv	保留（用于接口桥接）	

3.20.6 codec_event_cb_t:

```
typedef void (*codec_event_cb_t)(codec_event_t event, void *arg);
```

功能描述:

codec 事件回调函数，用户注册。

参数:

`event`: codec 事件，见 codec_event_t 定义。

`arg`: 用户注册回调参数，当回调函数触发时，会传入用户定义的的参数。

返回值:

无。

3.20.7 codec_event_t:

名称	定义 | 备注
CODEC_EVENT_PERIOD_READ_COMPLETE	codec 输入通路，获取完毕 period bytes 数据 |
CODEC_EVENT_PERIOD_WRITE_COMPLETE	codec 输出通路，发送完毕 period bytes 数据 |
CODEC_EVENT_WRITE_BUFFER_EMPTY	codec 输出通路，数据缓存区已空（缓存区为空，codec 将输出 0 数据） |
CODEC_EVENT_READ_BUFFER_FULL	codec 输入通路，数据缓存区已满（缓存区满，录音数据将无法写入缓存，数据丢失）|
CODEC_EVENT_TRANSFER_ERROR	codec 运行异常。|

3.20.7.1 csi_codec_input_config

```
int32_t csi_codec_input_config(codec_input_t *handle, codec_input_config_t *config);
```

功能描述:

codec 输入通路初始化，需要在 codec input
通路启动前调用，运行过程中不可配置。

参数: `handle`: ocdec input 句柄，见 codec_input_t 定义。
`config`: codec 输入通路配置参数，见 codec_input_config_t 定义。

返回值:

0: 成功。
其他: 错误码

3.20.8 codec_input_config_t:

名称	定义	备注
sample_rate	采样率	见 codec_sample_rate_t 定义
channel_num	通路的通道个数	例如: 录制双声道（左右声道数据交叉存储）音频，channel_num 设置为 2
bit_width	采样精度	

3.20.9 codec_sample_rate_e:

名称	定义	备注
CODEC_SAMPLE_RATE_8000	采样率:8000	
CODEC_SAMPLE_RATE_11025	采样率:11025	
CODEC_SAMPLE_RATE_12000	采样率:12000	
CODEC_SAMPLE_RATE_16000	采样率:16000	
CODEC_SAMPLE_RATE_22050	采样率:22050	
CODEC_SAMPLE_RATE_24000	采样率:24000	
CODEC_SAMPLE_RATE_32000	采样率:32000	
CODEC_SAMPLE_RATE_44100	采样率:44100	
CODEC_SAMPLE_RATE_48000	采样率:48000	
CODEC_SAMPLE_RATE_96000	采样率:96000	
CODEC_SAMPLE_RATE_192000	采样率:192000	

3.20.9.1 csi_codec_input_read

```
uint32_t csi_codec_input_read(codec_input_t *handle, uint8_t *buf, uint32_t length);
```

功能描述:

读取 codec 输入通路数据

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

buf: 读出数据的存储区。

length: 读取数据长度，单位 bytes。

返回值:

实际读取数据长度（单位 bytes），例如:length-1000，缓存区内数据为 100，则实际读取数据为 100，返回值为 100。

3.20.9.2 csi_codec_input_buf_avail

```
uint32_t csi_codec_input_buf_avail(codec_input_t *handle);
```

功能描述:

获取 codec input 通路缓存区空闲区域大小。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

返回值:

codec input 通路缓存区空闲空间，单位 bytes。

3.20.9.3 csi_codec_input_buf_reset

```
int32_t csi_codec_input_buf_reset(codec_input_t *handle);
```

功能描述:

复位 codec input 通路缓存区，复位后，缓存区的数据全部清空。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

返回值:

0: 成功。
其他: 错误码。

3.20.9.4 csi_codec_input_start

```
int32_t csi_codec_input_start(codec_input_t *handle);
```

功能描述:

启动 codec input 通路，启动后，codec input 开始采集数据。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

返回值:

0: 成功。
其他: 错误码。

3.20.9.5 csi_codec_input_stop

```
int32_t csi_codec_input_stop(codec_input_t *handle);
```

功能描述:

停止 codec input 通路数据采集，调用 stop 后，当前缓存区内的数据全部被清空。若想停止采集并且数不被清空，可调用 csi_codec_input_pause 函数，见 csi_codec_input_pause 定义。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

返回值:

0: 成功。
其他: 错误码。

3.20.9.6 csi_codec_input_pause

```
int32_t csi_codec_input_pause(codec_input_t *handle);
```

功能描述:

暂停 codec input 通路，codec
input 通路暂停后，数据停止采集，缓存区内数据不丢失，调用 csi_codec_input_resume 恢复采集后，数据续接。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

返回值: 0: 成功。
其他: 错误码。

3.20.9.7 csi_codec_input_resume

```
int32_t csi_codec_input_resume(codec_input_t *handle);
```

功能描述:

从暂停状态恢复 codec input 通路的数据采集。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

返回值:

0: 成功。
其他: 错误码。

3.20.9.8 csi_codec_input_set_digital_gain

```
int32_t csi_codec_input_set_digital_gain(codec_input_t *handle, int32_t val);
```

功能描述:

设置 codec input 通路的数字增益。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。
val: 增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.9.9 csi_codec_input_set_analog_gain

```
int32_t csi_codec_input_set_analog_gain(codec_input_t *handle, int32_t val);
```

功能描述:

设置 codec input 通路模拟增益。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。
val: 增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.9.10 csi_codec_input_get_digital_gain

```
int32_t csi_codec_input_get_digital_gain(codec_input_t *handle, int32_t *val);
```

功能描述:

获取当前 codec input 通路数字增益值。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。
val: 存储增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.9.11 csi_codec_input_get_analog_gain

```
int32_t csi_codec_input_get_analog_gain(codec_input_t *handle, int32_t *val);
```

功能描述:

获取 codec input 通路当前模拟增益。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

val: 存储增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.9.12 csi_codec_input_set_mixer_gain

```
int32_t csi_codec_input_set_mixer_gain(codec_input_t *handle, int32_t val);
```

功能描述:

设置 codec input 通路混频器增益。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

val: 增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.9.13 csi_codec_input_get_mixer_gain

```
int32_t csi_codec_input_get_mixer_gain(codec_input_t *handle, int32_t *val);
```

功能描述:

获取 codec input 通路当前混频器增益。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

val: 存储增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.9.14 csi_codec_input_mute

```
int32_t csi_codec_input_mute(codec_input_t *handle, int en);
```

功能描述:

控制 codec input 通路静音，当使能静音时，input 通路采集的数据无声音。

参数:

handle: ocdec input 句柄，见 codec_input_t 定义。

en: 1: 使能，0 关闭。

返回值:

0: 成功。

其他: 错误码。

3.20.9.15 csi_codec_output_open

```
int32_t csi_codec_output_open(codec_output_t *handle);
```

功能描述:

开启 codec 输输出通路（播放通路）。

参数:

handle:

codec_output_t 句柄, 用于配置与控制 output 通路, 在通路开启时, 为通路分配资源。见 codec_output_t 定义。

返回值:

0: 成功。

其他: 错误码

3.20.9.16 csi_codec_output_close

```
int32_t csi_codec_output_close(codec_output_t *handle);
```

功能描述:

关闭 codec 输输出通路（播放通路）。

参数:

`handle`: codec_output_t 句柄。

返回值:

0: 成功。
其他: 错误码

3.20.10 codec_output_t:

名称	定义	备注
codec_idx	codec 设备号	
ch_idx	input 通道号码	
cb	事件回调函数	
cb_arg	事件回调函数用户参数	
buf	input 数据缓存区	
buf_size	input 数据缓存区大小	
period	每采集 period bytes 数据量，触发一次回调函数	
priv	保留（用于接口桥接）	

3.20.10.1 csi_codec_output_config

```
int32_t csi_codec_output_config(codec_output_t *handle, codec_output_config_t *config);
```

功能描述:

codec 输除通路初始化，需要在 codec
output 通路启动前调用，运行过程中不可配置。

参数:

`handle`: codec_output_t 句柄，见 codec_input_t 定义。
`config`: codec 输出通路配置参数，见 codec_output_config_t 定义。

返回值:

0: 成功。
其他: 错误码

3.20.11 codec_output_config_t:

名称	定义	备注
sample_rate	采样率	见 codec_sample_rate_t 定义
bit_width	采样精度	
mono_mode_en	使能单声道模式	mono_mode_en-1: 使能单声道模式，0 关闭单声道模式

3.20.12 codec_sample_rate_e:

名称	定义	备注
CODEC_SAMPLE_RATE_8000	采样率:8000	
CODEC_SAMPLE_RATE_11025	采样率:11025	
CODEC_SAMPLE_RATE_12000	采样率:12000	
CODEC_SAMPLE_RATE_16000	采样率:16000	
CODEC_SAMPLE_RATE_22050	采样率:22050	
CODEC_SAMPLE_RATE_24000	采样率:24000	
CODEC_SAMPLE_RATE_32000	采样率:32000	
CODEC_SAMPLE_RATE_44100	采样率:44100	
CODEC_SAMPLE_RATE_48000	采样率:48000	
CODEC_SAMPLE_RATE_96000	采样率:96000	
CODEC_SAMPLE_RATE_192000	采样率:192000	

3.20.12.1 csi_codec_output_write

```
uint32_t csi_codec_output_write(codec_output_t *handle, uint8_t *buf, uint32_t length);
```

功能描述:

向 codec 输出通路写入数据

参数:

`handle`: ocdec output 句柄，见 codec_input_t 定义。

`buf`: 写入数据。

`length`: 写入数据长度，单位 bytes。

返回值:

实际写入数据长度(单位 bytes),例如:length-1000bytes,缓存区空余 100bytes,则实际写入数据为 100bytes,返回值为 100。

3.20.12.2 csi_codec_output_buf_avail

```
uint32_t csi_codec_output_buf_avail(codec_output_t *handle);
```

功能描述:

获取 codec output 通路缓存区空闲区域大小。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

返回值:

codec output 通路缓存区空闲空间，单位 bytes。

3.20.12.3 csi_codec_output_buf_reset

```
int32_t csi_codec_output_buf_reset(codec_output_t *handle);
```

功能描述:

复位 codec input 通路缓存区，复位后，缓存区的数据全部清空。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

返回值:

0: 成功。
其他: 错误码。

3.20.12.4 csi_codec_output_start

```
int32_t csi_codec_output_start(codec_output_t *handle);
```

功能描述:

启动 codec output 通路，启动后，codec output 开始采集数据。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

返回值:

0: 成功。

其他: 错误码。

3.20.12.5 csi_codec_output_stop

```
int32_t csi_codec_output_stop(codec_output_t *handle);
```

功能描述:

停止 codec

output 通路数据输出，调用 stop 后，当前缓存区内的数据全部被清空。若想停止输出并且数不被清空，可调用 csi_codec_output_pause 函数，见 csi_codec_output_pause 定义。

参数:

`handle`: ocdec output 句柄，见 codec_output_t 定义。

返回值:

0: 成功。

其他: 错误码。

3.20.12.6 csi_codec_output_pause

```
int32_t csi_codec_output_pause(codec_output_t *handle);
```

功能描述:

暂停 codec output 通路，codec output 通路暂停后，停止播放数据，缓存区内数据不丢失，调用 csi_codec_output_resume 恢复播放，数据续接。

参数:

`handle`: ocdec output 句柄，见 codec_output_t 定义。

返回值:

0: 成功。

其他: 错误码。

3.20.12.7 csi_codec_output_resume

```
int32_t csi_codec_output_resume(codec_output_t *handle);
```

功能描述:

从暂停状态恢复 codec output 通路的数据输出。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

返回值:

0: 成功。
其他: 错误码。

3.20.12.8 csi_codec_output_set_digital_left_gain

```
int32_t csi_codec_output_set_digital_left_gain(codec_output_t *handle, int32_t val);
```

功能描述:

设置 codec output 通路的左声道数字增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。
val: 增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.12.9 csi_codec_output_set_digital_right_gain

```
int32_t csi_codec_output_set_digital_right_gain(codec_output_t *handle, int32_t val);
```

功能描述:

设置 codec output 通路的右声道数字增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。
val: 增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.12.10 csi_codec_output_set_analog_left_gain

```
int32_t csi_codec_output_set_analog_left_gain(codec_output_t *handle, int32_t val);
```

功能描述:

设置 codec output 通路左声道模拟增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。
val: 增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.12.11 csi_codec_output_set_analog_right_gain

```
int32_t csi_codec_output_set_analog_right_gain(codec_output_t *handle, int32_t val);
```

功能描述:

设置 codec output 通路右声道模拟增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。
val: 增益数值

返回值:

0: 成功。
其他: 错误码。

3.20.12.12 csi_codec_output_get_digital_left_gain

```
int32_t csi_codec_output_get_digital_left_gain(codec_output_t *handle, int32_t *val);
```

功能描述:

获取当前 codec output 通路左声道数字增益值。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 存储增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.12.13 csi_codec_output_get_digital_right_gain

```
int32_t csi_codec_output_get_digital_right_gain(codec_output_t *handle, int32_t *val);
```

功能描述:

获取当前 codec output 通路右声道数字增益值。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 存储增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.12.14 csi_codec_output_get_analog_left_gain

```
int32_t csi_codec_output_get_analog_left_gain(codec_output_t *handle, int32_t *val);
```

功能描述:

获取 codec output 通路当前左声道模拟增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 存储增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.12.15 csi_codec_output_get_analog_right_gain

```
int32_t csi_codec_output_get_analog_right_gain(codec_output_t *handle, int32_t *val);
```

功能描述:

获取 codec output 通路当前右声道模拟增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 存储增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.12.16 csi_codec_output_set_mixer_left_gain

```
int32_t csi_codec_output_set_mixer_left_gain(codec_output_t *handle, int32_t val);
```

功能描述:

设置 codec output 通路混频器左声道增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 增益数值

返回值:

0: 成功。

其他: 错误码。

3.20.12.17 csi_codec_output_set_mixer_right_gain

```
int32_t csi_codec_output_set_mixer_right_gain(codec_output_t *handle, int32_t val);
```

功能描述:

设置 codec output 通路混频器右声道增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 增益数值

返回值: 0: 成功。

其他: 错误码。

3.20.12.18 csi_codec_output_get_mixer_left_gain

```
int32_t csi_codec_output_get_mixer_left_gain(codec_output_t *handle, int32_t *val);
```

功能描述:

获取 codec output 通路当前混频器左声道增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 存储增益数值

返回值: 0: 成功。

其他: 错误码。

3.20.12.19 csi_codec_output_get_mixer_right_gain

```
int32_t csi_codec_output_get_mixer_right_gain(codec_output_t *handle, int32_t *val);
```

功能描述:

获取 codec output 通路当前混频器右声道增益。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

val: 存储增益数值

返回值: 0: 成功。

其他: 错误码。

3.20.12.20 csi_codec_output_mute

```
int32_t csi_codec_output_mute(codec_output_t *handle, int en);
```

功能描述:

控制 codec output 通路静音，当使能静音时，output 通路播放无声音输出。

参数:

handle: ocdec output 句柄，见 codec_output_t 定义。

en: 1: 使能，0 关闭。

返回值:

0: 成功。

其他: 错误码。

3.20.13 示例:

```
/*******************************************************************************
 * @file     main.c
 * @brief    CSI Source File for main
 * @version  V1.0
 * @date     12. June 2018
 *******************************************************************************/

#include <stdint.h>
#include <stdio.h>
#include <csi_kernel.h>
#include "pinmux.h"
#include "drv_gpio.h"
#include "pin.h"
#include "drv_codec.h"

static void speaker_init()
{
    drv_pinmux_config(EXAMPLE_CODEC_PA_CTRL_PIN, EXAMPLE_CODEC_PA_CTRL_PIN_FUNC);
    gpio_pin_handle_t pgpio_pin_handle = csi_gpio_pin_initialize(PB22, NULL);
    csi_gpio_pin_config_mode(pgpio_pin_handle, GPIO_MODE_PULLNONE);
```

(下页继续)

(续上页)

```
    csi_gpio_pin_config_direction(pgpio_pin_handle, GPIO_DIRECTION_OUTPUT);
    csi_gpio_pin_write(pgpio_pin_handle, true);
}

static k_sem_handle_t player_sem;
static void player_cb(codec_event_t event, void *arg)
{

    if (event == CODEC_EVENT_WRITE_BUFFER_EMPTY) {
        printf("empty\n");
    }

    csi_kernel_sem_post(player_sem);
}


#define PLAYER_BUF_SIZE (1024 * 10)
static uint8_t player_buf[PLAYER_BUF_SIZE];
static codec_output_t output_handle;

static int player_init()
{
    player_sem = csi_kernel_sem_new(1, 0);
    if (player_sem == NULL) {
        return -1;
    }

    output_handle.buf = player_buf;
    output_handle.buf_size = PLAYER_BUF_SIZE;
    output_handle.cb = player_cb;
    output_handle.cb_arg = NULL;
    output_handle.ch_idx = 0;
    output_handle.codec_idx = 0;
    output_handle.period = 4096;

    int ret = csi_codec_output_open(&output_handle);
    if (ret != 0) {
        printf("codec output open error\n");
        return -1;
    }

    codec_output_config_t config;
    config.bit_width = 16;
```

(下页继续)

(续上页)

```
    config.mono_mode_en = 0;
    config.sample_rate = 48000;

    ret = csi_codec_output_config(&output_handle, &config);
    if (ret != 0) {
        printf("output config error\n");
        return -1;
    }

    csi_codec_output_set_mixer_left_gain(&output_handle, -20);
    csi_codec_output_set_mixer_right_gain(&output_handle, -20);

    return 0;
}

extern unsigned char  voice_file[];
extern uint32_t voice_len;

static void play_sound()
{
    int ret = -1;

    csi_codec_output_start(&output_handle);
    printf("output start\n");

    while(1) {
        uint32_t tx_data_len = 150*1024;
        uint8_t *tx_data = voice_file;
        uint32_t ret_len = 0;
        while(tx_data_len) {
            ret_len = csi_codec_output_write(&output_handle, (uint8_t *)tx_data, tx_data_len);
            tx_data_len -= ret_len;
            tx_data += ret_len;
            csi_kernel_sem_wait(player_sem, -1);
        }
    }

    printf("player end\n");
    csi_codec_output_stop(&output_handle);

    ret = csi_codec_output_close(&output_handle);
    if (ret != 0) {
        printf("2 codec output close error\n");
```

(下页继续)

(续上页)

```
        return;
    }

}

void player_test()
{
    if (player_init() == -1) {
        printf("player init error\n");
        return;
    }

    printf("player init end\n");
    play_sound();
}


#define RECORD_BUF_SIZE (1024 * 100)
static uint8_t receive_buf[RECORD_BUF_SIZE];
static codec_input_t input_handle;

#define RX_BUF_SIZE (10 * 1024)
static uint8_t rx_buf[RX_BUF_SIZE];

static k_sem_handle_t record_sem;
static void record_cb(codec_event_t event, void *arg)
{

    if (event == CODEC_EVENT_WRITE_BUFFER_EMPTY) {
        printf("empty\n");
    }

    csi_kernel_sem_post(record_sem);
}

static int record_init()
{
    record_sem = csi_kernel_sem_new(1, 0);
    if (record_sem == NULL) {
        return -1;
    }

    input_handle.buf = rx_buf;
```

(下页继续)

(续上页)

```
    input_handle.buf_size = RX_BUF_SIZE;
    input_handle.cb = record_cb;
    input_handle.cb_arg = NULL;
    input_handle.ch_idx = 0;
    input_handle.codec_idx = 0;
    input_handle.period = 2560;

    int ret = csi_codec_input_open(&input_handle);
    if (ret != 0) {
        printf("codec input open error\n");
        return -1;
    }

    codec_input_config_t config;
    config.bit_width = 16;
    config.channel_num = 1;
    config.sample_rate = 16000;
    ret = csi_codec_input_config(&input_handle, &config);
    if (ret != 0) {
        printf("codec input config error\n");
        return -1;
    }

    csi_codec_input_set_analog_gain(&input_handle, 4);
    return 0;
}

void record_teset()
{
    int ret = record_init();
    if (ret != 0) {
        printf("record init error\n");
        return;
    }

    printf("record init end\n");

    uint32_t rx_data_len = RECORD_BUF_SIZE;
    uint8_t *rx_buf = (uint8_t *)receive_buf;
    uint32_t ret_len = 0;
    csi_codec_input_start(&input_handle);

    while(rx_data_len) {
```

(下页继续)

(续上页)

```
        csi_kernel_sem_wait(record_sem, -1);
        ret_len = csi_codec_input_read(&input_handle, rx_buf, rx_data_len);
        rx_data_len -= ret_len;
        rx_buf+= ret_len;
    }

    printf("record end\n");
}

static uint8_t receive_buf1[RECORD_BUF_SIZE];
static codec_input_t input_handle1;
static uint8_t rx_buf1[RX_BUF_SIZE];
static k_sem_handle_t record_ref_sem;

static void record_ref_cb(codec_event_t event, void *arg)
{

    if (event == CODEC_EVENT_WRITE_BUFFER_EMPTY) {
        printf("empty\n");
    }

    csi_kernel_sem_post(record_ref_sem);
}

static int record_ref_init()
{
    record_ref_sem = csi_kernel_sem_new(1, 0);
    if (record_ref_sem == NULL) {
        return -1;
    }

    input_handle1.buf = rx_buf1;
    input_handle1.buf_size = RX_BUF_SIZE;
    input_handle1.cb = record_ref_cb;
    input_handle1.cb_arg = NULL;
    input_handle1.ch_idx = 2;
    input_handle1.codec_idx = 0;
    input_handle1.period = 2560;

    int ret = csi_codec_input_open(&input_handle1);
    if (ret != 0) {
        printf("codec input open error\n");
        return -1;
```

(下页继续)

(续上页)

```
    }

    codec_input_config_t config;
    config.bit_width = 16;
    config.channel_num = 1;
    config.sample_rate = 16000;
    ret = csi_codec_input_config(&input_handle1, &config);
    if (ret != 0) {
        printf("codec input config error\n");
        return -1;
    }

    csi_codec_input_set_analog_gain(&input_handle1, 4);

    return 0;
}

void record_ref_teset()
{
    int ret = record_ref_init();
    if (ret != 0) {
        printf("record ref init error\n");
        return;
    }

    printf("record ref init end\n");

    uint32_t rx_data_len = RECORD_BUF_SIZE;
    uint8_t *rx_buf = (uint8_t *)receive_buf1;
    uint32_t ret_len = 0;
    csi_codec_input_start(&input_handle1);

    while(rx_data_len) {
        csi_kernel_sem_wait(record_ref_sem, -1);
        ret_len = csi_codec_input_read(&input_handle1, rx_buf, rx_data_len);
        rx_data_len -= ret_len;
        rx_buf+= ret_len;
    }

    printf("record ref end\n");
}

#define EXAMPLE_PRIO    5
```

(下页继续)

(续上页)

```
#define EXAMPLE_TASK_STK_SIZE 1024

k_task_handle_t player_task;
k_task_handle_t record_task;
k_task_handle_t ref_record_task;

void player(void)
{
    player_test();
}

void record()
{
    record_teset();
}

void ref_record()
{
    record_ref_teset();
}

int main(void)
{
    csi_kernel_init();

    speaker_init();
    int32_t ret = csi_codec_init(0);
    if (ret != 0) {
        printf("codec init error\n");
    }

    printf("codec init end\n");

    csi_kernel_task_new((k_task_entry_t)player, "player",
                        0, EXAMPLE_PRIO, 0, 0, EXAMPLE_TASK_STK_SIZE, &player_task);


    csi_kernel_task_new((k_task_entry_t)record, "record",
                        0, EXAMPLE_PRIO, 0, 0, EXAMPLE_TASK_STK_SIZE, &record_task);


    csi_kernel_task_new((k_task_entry_t)ref_record, "ref_record",
                        0, EXAMPLE_PRIO, 0, 0, EXAMPLE_TASK_STK_SIZE, &ref_record_task);
```

(下页继续)

(续上页)

```
    csi_kernel_start();

    return 0;
}
```

第四章　CSI-Kernel API

4.1　Kernel Management

4.1.1　函数列表

- *csi_kernel_init*
- *csi_kernel_start*
- *csi_kernel_get_stat*

4.1.2　简要说明

系统接口用于实时操作系统的初始化、启动、以及获取实时操作系统的运行状态。

在使用任何实时操作系统接口之前必须先使用 csi_kernel_init 初始化实时操作系统，调用 csi_kernel_start 之后系统将进入调度，不会返回。

4.1.3　接口描述

4.1.3.1　csi_kernel_init

```
k_status_t csi_kernel_init(void)
```

功能描述:

Kernel 初始化接口。

参数:

无。

返回值:

错误码。

---

csi_kernel_start

```
k_status_t csi_kernel_start(void)
```

功能描述:

Kernel 启动接口。

参数:

无。

返回值:

错误码。

---

csi_kernel_get_stat

```
k_sched_stat_t csi_kernel_get_stat(void)
```

功能描述:

Kernel 运行的状态

参数:

无。

返回值:

Kernel 运行状态码。

4.2 Scheduler Management

4.2.1 函数列表

- *csi_kernel_sched_lock*
- *csi_kernel_sched_unlock*
- *csi_kernel_sched_restore_lock*
- *csi_kernel_sched_suspend*
- *csi_kernel_sched_resume*

4.2.2 简要说明

调度控制接口对整个系统的调度器进行控制。具体功能包括：

1、锁定与解锁系统调度。可以用来实现临界区功能。这些接口会影响整个系统的系统实时性，需谨慎使用。

2、挂起与恢复系统调度。可用来实现 tick-less 功能，一般用于低功耗模式。

4.2.3 接口描述

4.2.3.1 csi_kernel_sched_lock

```
int32_t csi_kernel_sched_lock(void)
```

功能描述:

锁定 Kernel 的任务调度（即禁止任务调度）。返回锁定前的状态。

参数:

无。

返回值:

返回锁定前的状态。

---

csi_kernel_sched_unlock

```
int32_t csi_kernel_sched_unlock(void)
```

功能描述:

解锁任务调度。

参数:

无。

返回值:

返回调用该接口前的任务锁定状态。

---

csi_kernel_sched_restore_lock

```
int32_t csi_kernel_sched_restore_lock(int32_t lock)
```

功能描述:

参数:

`lock`: 0: 解锁状态；1: 锁定状态。

返回值:

0 执行该函数之后系统的调度状态为未锁定状态。
1: 执行该函数之后系统的调度状态为锁定状态。
其它: 失败时的错误码。

---

csi_kernel_sched_suspend

```
uint32_t csi_kernel_sched_suspend(void)
```

功能描述:

暂停系统的任务调度。用于使用 tick-less 的低功耗模式。

参数:

无。

返回值:

系统能够进入睡眠的时间。单位：tick。

---

csi_kernel_sched_resume

```
void csi_kernel_sched_resume(uint32_t sleep_ticks)
```

功能描述:

恢复系统的任务调度。

参数:

`sleep_ticks`: 系统已睡眠的时间。单位：tick。

返回值:

无。

4.3 Task

4.3.1 函数列表

- *csi_kernel_task_new*
- *csi_kernel_task_del*
- *csi_kernel_task_get_cur*
- *csi_kernel_task_get_stat*
- *csi_kernel_task_set_prio*
- *csi_kernel_task_get_prio*
- *csi_kernel_task_get_name*
- *csi_kernel_task_suspend*
- *csi_kernel_task_resume*
- *csi_kernel_task_terminate*
- *csi_kernel_task_exit*
- *csi_kernel_task_yield*
- *csi_kernel_task_get_count*
- *csi_kernel_task_get_stack_size*
- *csi_kernel_task_get_stack_space*
- *csi_kernel_task_list*

4.3.2 简要说明

任务是实时操作系统进行资源分配和调度的基本单位。实时操作系统的任务一般采用优先级抢占和时间片轮转的调度机制，即高优先级的任务可以抢占低优先级的任务，同等级别的任务采用时间片轮转的方式调度。

每个任务都有一个优先级，不同类型的实时操作系统对优先级可能有不同的定义，比如对 rhino 任务最高优先级为 0，对 freeRTOS 任务最低优先级为 0。CSI-Kernel 统一规范了优先级的定义，以保持接口对用户接口的一致性，参看 k_priority_t 的定义。

每个任务都拥有一个独立的栈空间，栈空间里保存的信息包含局部变量、寄存器、函数参数、函数返回地址等。任务在任务切换时会将切出任务的上下文信息保存在自身的任务栈空间里面，以便任务恢复时还原现场，从而在任务恢复后在切出点继续开始执行。

4.3.3 接口描述

4.3.3.1 csi_kernel_task_new

```
k_status_t csi_kernel_task_new(k_task_entry_t task, const char *name, void *arg,
                               k_priority_t prio, uint32_t time_quanta,
                               void *stack, uint32_t stack_size, k_task_handle_t *task_handle);
```

功能描述:

创建一个任务并将其加入到活动任务队列中。

参数:

`task`: 任务入口函数。

`name`: 任务名称。

`arg`: 任务入口函数的参数。

`prio`: 任务优先级。定义见*k_priority_t* 。

`time_quanta`: 在 round-robin 模式下的调度周期（以 tick 为单位）。当值为 0 时，使用 FIFO 的调度方式。

`stack`: 栈基址，栈空间用户可以自己定义；如果传入 NULL，那么栈空间由系统申请。

`stack_size`: 任务的 stack 大小。

`task_handle`: 函数执行成功使保存任务的句柄。

返回值:

错误码。

优先级类型	定义	备注
KPRIO_IDLE		
KPRIO_LOW0	priority: low	
KPRIO_LOW1	priority: low + 1	
KPRIO_LOW2	priority: low + 2	
KPRIO_LOW3	priority: low + 3	
KPRIO_LOW4	priority: low + 4	
KPRIO_LOW5	priority: low + 5	
KPRIO_LOW6	priority: low + 6	
KPRIO_LOW7	priority: low + 7	
KPRIO_NORMAL_BELOW0	priority: below normal	
KPRIO_NORMAL_BELOW1	priority: below normal + 1	
KPRIO_NORMAL_BELOW2	priority: below normal + 2	
KPRIO_NORMAL_BELOW3	priority: below normal + 3	
KPRIO_NORMAL_BELOW4	priority: below normal + 4	
KPRIO_NORMAL_BELOW5	priority: below normal + 5	
KPRIO_NORMAL_BELOW6	priority: below normal + 6	
KPRIO_NORMAL_BELOW7	priority: below normal + 7	
KPRIO_HIGH0	priority: high	
KPRIO_HIGH1	priority: high + 1	
KPRIO_HIGH2	priority: high + 2	
KPRIO_HIGH3	priority: high + 3	
KPRIO_HIGH4	priority: high + 4	
KPRIO_HIGH5	priority: high + 5	
KPRIO_HIGH6	priority: high + 6	

下页继续

表 4.1 – 续上页

优先级类型	定义	备注
KPRIO_HIGH7	priority: high + 7	
KPRIO_REALTIME0	priority: realtime	
KPRIO_REALTIME1	priority: realtime + 1	
KPRIO_REALTIME2	priority: realtime + 2	
KPRIO_REALTIME3	priority: realtime + 3	
KPRIO_REALTIME4	priority: realtime + 4	
KPRIO_REALTIME5	priority: realtime + 5	
KPRIO_REALTIME6	priority: realtime + 6	
KPRIO_REALTIME7	priority: realtime + 7	

---

csi_kernel_task_del

```
k_status_t csi_kernel_task_del(k_task_handle_t task_handle)
```

功能描述:

删除一个任务。

参数:

`task_handle`: 要删除任务的句柄。

返回值:

错误码。

---

csi_kernel_task_get_cur

```
k_task_handle_t csi_kernel_task_get_cur(void)
```

功能描述:

获取当前正在运行的任务句柄。

参数:

无。

返回值:

当前正在运行的任务句柄。

---

csi_kernel_task_get_stat

```
k_task_stat_t csi_kernel_task_get_stat(k_task_handle_t task_handle)
```

功能描述:

获取任务的运行状态。

参数:

`task_handle`: 需要操作的任务句柄。

返回值:

任务状态。

k_task_stat_t:

状态名	定义	备注
KTASK_ST_INACTIVE	任务状态为 Inactive	
KTASK_ST_READY	任务状态为 Ready	
KTASK_ST_RUNNING	任务状态为 Running	
KTASK_ST_BLOCKED	任务状态为 Blocked	
KTASK_ST_TERMINATED	任务状态为 Terminated	
KTASK_ST_ERROR	出错	

---

csi_kernel_task_set_prio

```
k_status_t csi_kernel_task_set_prio(k_task_handle_t task_handle, k_priority_t priority)
```

功能描述:

设置任务的优先级。

参数:

`task_handle`: 需要操作的任务句柄

`priority`: 任务需要设置的优先级。优先级定义见*k_priority_t* 。

返回值:

错误码。

---

csi_kernel_task_get_prio

```
k_priority_t csi_kernel_task_get_prio(k_task_handle_t task_handle)
```

功能描述:

获取任务的优先级。

参数:

task_handle: 需要操作的任务句柄。

返回值:

>=0: 任务优先级。优先级定义见*k_priority_t* 。

<0: 错误码。

---

csi_kernel_task_get_name

```
const char *csi_kernel_task_get_name(k_task_handle_t task_handle)
```

功能描述:

获取任务名。

参数:

task_handle: 需要操作的任务句柄。

返回值:

任务名。

---

csi_kernel_task_suspend

```
k_status_t csi_kernel_task_suspend(k_task_handle_t task_handle)
```

功能描述:

暂停某个任务的调度。

参数:

task_handle: 要暂停任务的句柄。

返回值:

错误号。

---

csi_kernel_task_resume

```
k_status_t csi_kernel_task_resume(k_task_handle_t task_handle)
```

功能描述:

恢复某个任务的调度。

参数:

task_handle: 要恢复调度任务的句柄。

返回值:

错误码。

---

csi_kernel_task_terminate

```
k_status_t csi_kernel_task_terminate(k_task_handle_t task_handle)
```

功能描述:

终止某个任务，被终止的任务不再会被调度。

参数:

task_handle: 要终止任务的句柄。

返回值:

错误码。

---

csi_kernel_task_exit

```
void csi_kernel_task_exit(void)
```

功能描述:

退出当前任务。

参数:

无。

返回值:

无。

---

csi_kernel_task_yield

```
k_status_t csi_kernel_task_yield(void)
```

功能描述:

当前任务让出本次调度。

参数:

无。

返回值:

错误码。

---

csi_kernel_task_get_count

```
uint32_t csi_kernel_task_get_count(void)
```

功能描述:

获取系统当前任务的个数。

参数:

无。

返回值:

系统当前任务的个数。

---

csi_kernel_task_get_stack_size

```
uint32_t csi_kernel_task_get_stack_size(k_task_handle_t task_handle)
```

功能描述:

获取任务栈大小。

参数:

`task_handle`: 任务的句柄。

返回值:

任务栈大小。

---

csi_kernel_task_get_stack_space

```
uint32_t csi_kernel_task_get_stack_space(k_task_handle_t task_handle)
```

功能描述:

获取任务的剩余栈大小。

参数:

`task_handle`: 任务的句柄。

返回值:

任务的剩余栈大小。

---

csi_kernel_task_list

```
uint32_t csi_kernel_task_list(k_task_handle_t *task_array, uint32_t array_items)
```

功能描述:

获取系统当前所有任务的句柄。

参数:

`task_array`: 保存任务句柄的数组地址。
`attay_items`: 保存任务句柄的数组元素个数。

返回值:

数组中实际获取到的任务个数。

备注:

1. 可先通过 csi_kernel_task_get_count 来获取到系统中任务的个数来确定数组的大小。
2. 当 array_items 大于返回值时，数组总的有效句柄个数根据返回值确定。

4.4 Semphore

- *csi_kernel_sem_new*
- *csi_kernel_sem_del*
- *csi_kernel_sem_wait*
- *csi_kernel_sem_post*
- *csi_kernel_sem_get_count*

信号量（Semaphore）是一种实现任务间通信的机制，实现任务之间同步或临界资源的互斥访问，常用于协助一组相互竞争的任务来访问临界资源。与互斥量不同，信号量可允许多个任务同时访问系统资源。信号量通过一个资源计数值来记录当前可用的资源数，当该计数值为 0 时，则不允许访问资源，直到有其他任务释放资源。

当信号量用于互斥功能时，信号量创建时将其资源计数值置为资源的最大值，任务需要使用临界资源时先获取信号量，当资源被使用完时，若还有任务申请使用临界资源时就会因为无法取到信号量而阻塞，从而保证了临界资源的安全。

当信号量用作同步功能时，信号量创建时将其资源计数值置为 0，任务 1 取信号量而阻塞，任务 2 在某种条件发生后，释放信号量，于是任务 1 得以执行，从而达到了两个任务间的同步。

```
k_sem_handle_t csi_kernel_sem_new(int32_t max_count, int32_t initial_count)
```

功能描述:

创建一个信号量。

参数:

`max_count`: 信号量的计数器最大个数。

`initial_count`: 信号量的计数器初始值。

返回值:

NULL: 创建失败。

其它: 信号量句柄。

```
k_status_t csi_kernel_sem_del(k_sem_handle_t sem_handle)
```

功能描述:

删除一个信号量。

参数:

`sem_handle`: 信号量的句柄。

返回值:

错误码。

```
k_status_t csi_kernel_sem_wait(k_sem_handle_t sem_handle, int32_t timeout)
```

功能描述:

等待信号量。

参数:

sem_handle: 信号量的句柄。

timeout: 超时时间 (单位：tick)。在该时间内若获取不到信号量，该函数返回。

0 - 不等待；

负值 - 一直等待；

其它正值 - 最大的等待时间。

返回值:

错误码。

```
k_status_t csi_kernel_sem_post(k_sem_handle_t sem_handle)
```

功能描述:

发送一个信号量。

参数:

sem_handle: 信号量的句柄。

返回值:

错误码。

```
int32_t csi_kernel_sem_get_count(k_sem_handle_t sem_handle)
```

功能描述:

获取信号量当前计数器的值。

参数:

sem_handle: 信号量的句柄。

返回值:

非负值：信号量当前计数器的值。

负值：错误码。

4.5 Mutex

- *csi_kernel_mutex_new*
- *csi_kernel_mutex_del*
- *csi_kernel_mutex_lock*
- *csi_kernel_mutex_unlock*
- *csi_kernel_mutex_get_owner*

互斥量（也称为互斥锁或互斥体）是一种特殊的二值性信号量，用于任务间的同步，保证在任何时刻仅有一个任务访问临界资源。

当一个任务访问临界资源时，通过将互斥量置于加锁状态，此时其他任务如果访问这个资源则会被阻塞，直到互斥量被该任务解锁后，才能访问。互斥量保证同一时刻只有一个任务访问临界资源，确保了资源的完整性。

```
k_mutex_handle_t csi_kernel_mutex_new(void)
```

功能描述:

创建一个互斥量。

参数:

无。

返回值:

NULL: 创建失败。
其它: 互斥量的句柄。

```
k_status_t csi_kernel_mutex_del(k_mutex_handle_t mutex_handle)
```

功能描述:

删除一个互斥量。

参数:

`mutex_handle`: 互斥量的句柄。

返回值:

错误码。

```
k_status_t csi_kernel_mutex_lock(k_mutex_handle_t mutex_handle, int32_t timeout)
```

功能描述:

获取并锁定一个互斥量。

参数:

mutex_handle: 互斥量的句柄

timeout: 超时时间 (单位：tick)。在该时间内若获取不到互斥量，该函数返回。

0 - 不等待。

负值 - 一直等待。

其它正值 - 最大的等待时间。

返回值:

错误码。

```
k_status_t csi_kernel_mutex_unlock(k_mutex_handle_t mutex_handle)
```

功能描述:

解锁一个互斥量。

参数:

mutex_handle: 互斥量的句柄。

返回值:

错误码。

```
k_task_handle_t csi_kernel_mutex_get_owner(k_mutex_handle_t mutex_handle)
```

功能描述:

获取互斥量的所有者。

参数:

mutex_handle: 互斥量的句柄。

返回值:

NULL: 获取失败。

其它: 互斥量所有者的任务句柄。

4.6 Message Queue

- *csi_kernel_msgq_new*
- *csi_kernel_msgq_del*
- *csi_kernel_msgq_put*
- *csi_kernel_msgq_get*
- *csi_kernel_msgq_get_count*
- *csi_kernel_msgq_get_capacity*
- *csi_kernel_msgq_get_msg_size*
- *csi_kernel_msgq_flush*

消息队列是一种常用的任务间异步通信机制，任务可往消息队列里面发送数据或者从队列里面读取消息。从消息队列中读消息时，若队列中的消息为空，挂起读取任务，若队列中有新消息时，挂起的读取任务被唤醒并处理新消息。

```
k_msgq_handle_t csi_kernel_msgq_new(int32_t msg_count, int32_t msg_size)
```

功能描述:

创建一个消息队列。

参数:

`msg_count`: 消息队列中消息的最大个数。

`msg_size`: 每个消息的最大长度。

返回值:

NULL: 创建失败。

其它: 消息队列句柄。

```
k_status_t csi_kernel_msgq_del(k_msgq_handle_t mq_handle)
```

功能描述:

删除一个消息队列。

参数:

`mq_handle`: 消息队列句柄。

返回值:

错误码。

```
k_status_t csi_kernel_msgq_put(k_msgq_handle_t mq_handle, const void *msg_ptr, uint8_t front_or_
↪back, int32_t timeout)
```

功能描述:

往消息队列插入一个消息。

参数:

mq_handle: 消息队列的句柄。

msg_ptr: 带插入消息的指针。

front_or_back: 插入到消息队列的队首还是队尾。1 - 队首；0 - 队尾。

timeout: 超时时间 (单位：tick)。在该时间内若消息插入失败，该函数返回。

0 - 不等待。

负值 - 一直等待。

其它正值 - 最大的等待时间。

返回值:

错误码。

```
k_status_t csi_kernel_msgq_get(k_msgq_handle_t mq_handle, void *msg_ptr, int32_t timeout)
```

功能描述:

从消息队列获取一个消息。

参数:

mq_handle: 消息队列的句柄。

msg_ptr: 保存获取到的消息的指针。

timeout: 超时时间 (单位：tick)。在该时间内若消息获取失败，该函数返回。

0 - 不等待

负值 - 一直等待

其它正值 - 最大的等待时间

返回值:

错误码。

```
int32_t csi_kernel_msgq_get_count(k_msgq_handle_t mq_handle)
```

功能描述:

获取消息队列中消息的数量。

参数:

mq_handle: 消息队列的句柄。

返回值:

>=0: 消息队列中消息的数量。

<0: 错误码。

```
uint32_t csi_kernel_msgq_get_capacity(k_msgq_handle_t mq_handle)
```

功能描述:

消息队列的最大消息个数。

参数:

mq_handle: 消息队列的句柄。

返回值:

<=0: 执行失败。

>0: 消息队列的最大消息个数。

```
uint32_t csi_kernel_msgq_get_msg_size(k_msgq_handle_t mq_handle)
```

功能描述:

消息队列中消息大小最大者。

参数:

mq_handle: 消息队列的句柄。

返回值:

<=0: 执行失败。

>0: 单条消息支持的最大大小，单位：字节。

```
k_status_t csi_kernel_msgq_flush(k_msgq_handle_t mq_handle)
```

功能描述:

清空消息队列中的消息。

参数:

mq_handle: 消息队列的句柄。

返回值:

错误码。

4.7 Timer

- *csi_kernel_timer_new*
- *csi_kernel_timer_del*
- *csi_kernel_timer_start*
- *csi_kernel_timer_stop*
- *csi_kernel_timer_get_stat*

软件定时器是基于系统时钟中断且由软件来实现的定时器，其精度取决于系统 tick 时钟的周期。软件定时器在当经过设定的 tick 时钟计数值后会触发用户定义的回调函数。

软件定时器支持单次触发模式和周期触发模式。单次触发定时器在启动后只会触发一次定时器回调函数，然后定时器停止运行。周期触发定时器会周期性地触发定时器回调函数，直到用户手动地停止定时器，否则将永远持续执行下去。

```
k_timer_handle_t csi_kernel_timer_new(k_timer_cb_t func, k_timer_type_t type, void *arg)
```

功能描述:

创建一个新的软件定时器。

参数:

func: 定时器超时回调函数。回调函数原型 k_timer_cb_t 定义如下：

```
typedef void (*k_timer_cb_t)(void *arg);
```

type: 指定创建的软件定时器的类型。见*k_timer_type_t* 定义

arg: 定时器回调函数 func 的参数。

返回值:

NULL: 创建失败。

其它值: 定时器的句柄。

类型	定义	备注
KTIMER_TYPE_ONCE	单次触发类型定时器	此类型的定时器仅会触发一次定时器事件，然后停止。
KTIMER_TYPE_PERIODIC	周期触发类型定时器	此类型的定时器会周期性地触发定时器事件,直到用户手动地停止定时器（通过 csi_kernel_timer_stop），否则会持续执行下去。

```
k_status_t csi_kernel_timer_del(k_timer_handle_t timer_handle)
```

功能描述:

删除一个软件定时器。

参数:

timer_handle: 软件定时器的句柄。

返回值:

错误码。

```
k_status_t csi_kernel_timer_start(k_timer_handle_t timer_handle, uint32_t ticks)
```

功能描述:

启动一个软件定时器。

参数:

timer_handle: 软件定时器的句柄。

ticks: 定时器周期。单位: 系统 tick。

返回值:

错误码。

```
k_status_t csi_kernel_timer_stop(k_timer_handle_t timer_handle)
```

功能描述:

停止一个软件定时器。

参数:

timer_handle: 软件定时器的句柄。

返回值:

错误码。

```
k_timer_stat_t csi_kernel_timer_get_stat(k_timer_handle_t timer_handle)
```

功能描述:

获取软件定时器的运行状态。

参数:

`timer_handle`: 软件定时器的句柄。

返回值:

负值: 错误码。
其它值: 定时器的状态。*k_timer_stat_t* 定义。

名字	定义	备注
PAGE_SIZE_4KB	页面大小为 4KB	
PAGE_SIZE_16KB	页面大小为 16KB	
PAGE_SIZE_64KB	页面大小为 64KB	
PAGE_SIZE_256KB	页面大小为 256KB	
PAGE_SIZE_1MB	页面大小为 1MB	
PAGE_SIZE_4MB	页面大小为 4MB	
PAGE_SIZE_4MB	页面大小为 16MB	

4.8 Generic Time

- *csi_kernel_delay*
- *csi_kernel_delay_until*
- *csi_kernel_tick2ms*
- *csi_kernel_ms2tick*
- *csi_kernel_delay_ms*
- *csi_kernel_get_ticks*
- *csi_kernel_get_tick_freq*
- *csi_kernel_get_systimer_freq*

系统定时器是实时操作系统正常运行的核心机制，操作系统通过系统定时器来进行系统时间的更新和任务调度的控制。系统定时器的时钟也称为时标或者 tick，其频率一般可配置，比如为 100HZ。操作系统对外提供的时间接口一般都是以 tick 为基本单位，因此其精度不是很高。

时间相关接口主要向用户提供如下功能：系统时间（tick）的获取、延时功能、时间与 tick 值的转换接口。

```
k_status_t csi_kernel_delay(uint32_t ticks)
```

功能描述:

从当前时间起延时指定的一段时间（以内核 tick 为单位）。延时过程中可释放 CPU。

参数:

`ticks`: 需要延时的 tick 数。

返回值:

错误码。

```
k_status_t csi_kernel_delay_until(uint64_t ticks)
```

功能描述:

延时到某个具体的时间（内核 tick 计数器指定）。延时过程中可释放 CPU。

参数:

`ticks`: 需要延时到的 tick 时间点。

返回值:

错误码。

```
uint64_t csi_kernel_tick2ms(uint32_t ticks)
```

功能描述:

系统 tick 到毫秒之间的转换。

参数:

`ticks`: 需要转换的 tick 个数。

返回值:

转换得到的毫秒数。

```
uint64_t csi_kernel_ms2tick(uint32_t ms)
```

功能描述:

毫秒到系统 tick 之间的转换。

参数:

ms: 需要转换的毫秒数。

返回值:

转换得到的系统 tick 数。

```
k_status_t csi_kernel_delay_ms(uint32_t ms)
```

功能描述:

从当前时间起延时指定的一段时间（以毫秒为单位）。延时过程中可释放 CPU。

参数:

ms: 需要延时的毫秒数。

返回值:

错误码。

```
uint64_t csi_kernel_get_ticks(void)
```

功能描述:

获取系统的内核 tick 计数值。

参数:

无。

返回值:

系统的内核 tick 计数值。

```
uint32_t csi_kernel_get_tick_freq(void)
```

功能描述:

获取 tick 的频度，即 1 秒钟执行多少次 tick 中断。

参数:

无。

返回值:

内核 tick 频度。

```
uint32_t csi_kernel_get_systimer_freq(void)
```

功能描述:

获取 systimer 的硬件频率值。

参数:

无。

返回值:

硬件系统时钟的频率。

4.9 Memory Pool

- *csi_kernel_mpool_new*
- *csi_kernel_mpool_del*
- *csi_kernel_mpool_alloc*
- *csi_kernel_mpool_free*
- *csi_kernel_mpool_get_count*
- *csi_kernel_mpool_get_capacity*
- *csi_kernel_mpool_get_block_size*

内存池通过固定长度的内存块来管理一片内存。用户申请和释放内存以固定长度为单位。通过内存池是可以保证正确且高效的申请释放内存。

```
k_mpool_handle_t csi_kernel_mpool_new(void *p_addr, int32_t block_count, int32_t block_size)
```

功能描述:

创建一个内存池。

参数:

`p_addr`: 内存池缓冲区的首地址。

`block_count`: 内存块的个数。

`block_size`: 每个内存块的大小。

返回值:

NULL: 创建失败。

其它: 创建成功时的句柄。


```
k_status_t csi_kernel_mpool_del(k_mpool_handle_t mp_handle)
```

功能描述:

删除一个内存池。

参数:

mp_handle: 内存池的句柄。

返回值:

错误码。

```
void *csi_kernel_mpool_alloc(k_mpool_handle_t mp_handle, int32_t timeout)
```

功能描述:

从内存池分配一个内存块。

参数:

mq_handle: 内存池的句柄。

timeout: 超时时间 (单位：tick)。在该时间内若无法分配内存块，该函数返回。

0 - 不等待。

负值 - 一直等待。

其它正值 - 最大的等待时间。

返回值:

NULL: 分配失败。

其它: 分配成功时的内存块首地址。

```
k_status_t csi_kernel_mpool_free(k_mpool_handle_t mp_handle, void *block)
```

功能描述:

释放一个内存块。

参数:

mq_handle: 内存池的句柄。

block: 待释放的内存块的首地址。

返回值:

错误码。

```
int32_t csi_kernel_mpool_get_count(k_mpool_handle_t mp_handle)
```

功能描述:

获取内存池中已经被使用的内存块个数。

参数:

mq_handle: 内存池的句柄。

返回值:

负值: 错误码。
其它: 内存池中已经被使用的内存块个数。

```
uint32_t csi_kernel_mpool_get_capacity(k_mpool_handle_t mp_handle)
```

功能描述:

获取内存池的总大小。

参数:

mq_handle: 内存池的句柄。

返回值:

0: 获取失败。
其它: 内存池的总大小。

```
uint32_t csi_kernel_mpool_get_block_size(k_mpool_handle_t mp_handle)
```

功能描述:

获取内存池中每个内存块的大小。

参数:

mq_handle: 内存池的句柄。

返回值:

0: 获取失败。
其它: 内存块的大小。

4.10 Event

- *csi_kernel_event_new*
- *csi_kernel_event_del*
- *csi_kernel_event_set*
- *csi_kernel_event_clear*
- *csi_kernel_event_get*
- *csi_kernel_event_wait*

事件/event 是一种实现任务间通信的机制，可用于实现任务间的同步。事件通过标志位来告知任务某个事件是否发生，不能在任务间传输数据，是一种轻量级的任务通信机制。接口中的每个事件 (event) 可包括 32 个标志位 (flag)，每一位可代表一个具体的事件。任务可以设置、清除和等待事件标志位。

通过事件的多个标志位，一个任务可以同时等待多个事件的发生。事件标志支持通过如下几种方式来唤醒等待事件的任务：

所有事件标志位置 1
任一事件标志位置 1
所有事件标志位清 0
任一事件标志位清 0

```
k_event_handle_t csi_kernel_event_new(void)
```

功能描述:

创建一个事件。

参数:

无。

返回值:

NULL: 创建失败。
其它: 创建成功时事件的句柄。

```
k_status_t csi_kernel_event_del(k_event_handle_t ev_handle)
```

功能描述:

删除一个事件。

参数:

`ev_handle`: 事件的句柄。

返回值:

错误码。

```
k_status_t csi_kernel_event_set(k_event_handle_t ev_handle, uint32_t flags, uint32_t *ret_flags)
```

功能描述:

设置一个事件的标志位。

参数:

ev_handle: 事件的句柄。

flags: 需要设置的事件标志位。

ret_flags: 返回设置后的事件标志位。

返回值:

错误码。

```
k_status_t csi_kernel_event_clear(k_event_handle_t ev_handle, uint32_t flags, uint32_t *ret_flags)
```

功能描述:

清除一个事件的标志状态。

参数:

ev_handle: 清除一个事件的标志状态。

flags: 需要被清除的事件标志位。

ret_flags: 在事件标志被清除之前的事件标志状态。

返回值:

错误码。

```
k_status_t csi_kernel_event_get(k_event_handle_t ev_handle, uint32_t *ret_flags)
```

功能描述:

获取一个事件的标志状态。

参数:

ev_handle: 要操作的事件句柄。

ret_flags: 成功执行时返回事件的标志位。

返回值:

错误码。

```
k_status_t csi_kernel_event_wait(k_event_handle_t ev_handle, uint32_t flags,
                                 k_event_opt_t options, uint8_t clr_on_exit,
                                 uint32_t *actl_flags, int32_t timeout)
```

功能描述:

参数:

ev_handle: 要操作的事件句柄。

flags: 等待的事件标志位。

options: 时间标志选项，参看*k_event_opt_t* 定义。

clr_on_exit: 是否在函数内部清除对应的事件标志位。1 - 清除；0 - 不清除。

actl_flags: 在事件标志被清除（参数 clr_on_exit 为 1）之前的事件标志状态或者等待超时后的事件标志状态。

timeout: 超时时间 (单位：tick)。在该时间内若要等待的事件标志未发生，该函数返回。

0 - 不等待。

负值 - 一直等待。

其他正值 - 最大的等待时间。

返回值:

0: 等待的事件标志发生。

其它: 未等待到指定事件标志时的错误码。

名字	定义	备注
KEVENT_OPT_SET_ANY	flags 中任何为 1 的比特位匹配	
KEVENT_OPT_SET_ALL	flags 中所有为 1 的比特位匹配	
KEVENT_OPT_CLR_ANY	flags 中任何为 0 的比特位匹配	
KEVENT_OPT_CLR_ALL	flags 中所有为 0 的比特位匹配	

4.11 Heap Management

- *csi_kernel_malloc*
- *csi_kernel_free*
- *csi_kernel_realloc*
- *csi_kernel_get_mminfo*
- *csi_kernel_mm_dump*

动态内存管理。

```
void *csi_kernel_malloc(int32_t size, void *caller)
```

功能描述:

申请一块内存。

参数:

`size`: 申请的内存大小。

`caller`: csi_kernel_malloc 的调用者，可以是 ___builtin_return_address(0) 或者 NULL。

返回值:

NULL: 申请失败。

其它: 内存首地址。

```
void csi_kernel_free(void *ptr, void *caller)
```

功能描述:

释放一块内存。

参数:

`ptr`: 需释放内存的首地址。

`caller`: csi_kernel_free 的调用者，可以是 ___builtin_return_address(0) 或者 NULL。

返回值:

无。

```
void *csi_kernel_realloc(void *ptr, int32_t size, void *caller)
```

功能描述:

改变已申请内存的大小。

参数:

`ptr`: 已申请的内存首地址。

`size`: 改变后的内存大小。

`caller`: csi_kernel_realloc 的调用者，可以是 ___builtin_return_address(0) 或者 NULL。

返回值:

NULL: 申请失败。

其它: 新内存首地址。

```
k_status_t csi_kernel_get_mminfo(int32_t *total, int32_t *used, int32_t *free, int32_t *peak)
```

功能描述:

获取动态内存的管理信息。

参数:

`total`: 返回堆总大小。

`used`: 返回已使用堆大小。

`free`: 返回剩余的堆大小。

`peak`: 返回堆使用的峰值。

返回值:

错误码。

```
k_status_t csi_kernel_mm_dump(void)
```

功能描述:

Dump 堆的调试信息。

参数:

无。

返回值:

无。

4.12 Interrupt

- *csi_kernel_intrpt_enter*
- *csi_kernel_intrpt_exit*

提供统一的进出中断接口。

```
k_status_t csi_kernel_intrpt_enter(void)
```

功能描述:

进入中断时调用。一般用于统计中断嵌套层数，每次执行中断嵌套层数加 1。

参数:

无。

返回值:

错误码。

```
k_status_t csi_kernel_intrpt_exit(void)
```

功能描述:

退出中断时调用。一般用于中断嵌套层数统计，每次执行中断嵌套层数减 1。

参数:

无。

返回值:

错误码。

4.13 CSI-Kernel ERRNO

错误码	说明
EPERM	Operation not permitted
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
E2BIG	Arg list too long
ENOEXEC	Exec format error
EBADF	Bad file number
ECHILD	No child processes
EAGAIN	Try again
ENOMEM	Out of memory
EACCES	Permission denied
EFAULT	Bad address
ENOTBLK	Block device required
EBUSY	Device or resource busy
EEXIST	File exists

下页继续

表 4.2 – 续上页

错误码	说明
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	File table overflow
EMFILE	Too many open files
ENOTTY	Not a typewriter
ETXTBSY	Text file busy
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument out of domain of func
ERANGE	Math result not representable
EDEADLK	Resource deadlock would occur
ENAMETOOLONG	File name too long
ENOLCK	No record locks available
ENOSYS	Function not implemented
ENOTEMPTY	Directory not empty
ELOOP	Too many symbolic links encountered
ENOMSG	No message of desired type
EIDRM	dentifier removed
ECHRNG	Channel number out of range
EL2NSYNC	Level 2 not synchronized
EL3HLT	Level 3 halted
EL3RST	Level 3 reset
ELNRNG	Link number out of range
EUNATCH	Protocol driver not attached
ENOCSI	No CSI structure available
EL2HLT	Level 2 halted
EBADE	Invalid exchange
EBADR	Invalid request descriptor
EXFULL	Exchange full
ENOANO	No anode
EBADRQC	Invalid request code
EBADSLT	Invalid slot
EBFONT	Bad font file format
ENOSTR	Device not a stream

下页继续

表 4.2 – 续上页

错误码	说明
ENODATA	No data available
ETIME	Timer expired
ENOSR	Out of streams resources
ENONET	Machine is not on the network
ENOPKG	Package not installed
EREMOTE	Object is remote
ENOLINK	Link has been severed
EADV	Advertise error
ESRMNT	Srmount error
ECOMM	Communication error on send
EPROTO	Protocol error
EMULTIHOP	Multihop attempted
EDOTDOT	RFS specific error
EBADMSG	Not a data message
EOVERFLOW	Value too large for defined data type
ENOTUNIQ	Name not unique on network
EBADFD	File descriptor in bad state
EREMCHG	Remote address changed
ELIBACC	Can not access a needed shared library
ELIBBAD	Accessing a corrupted shared library
ELIBSCN	.lib section in a.out corrupted
ELIBMAX	Attempting to link in too many shared libraries
ELIBEXEC	Cannot exec a shared library directly
EILSEQ	Illegal byte sequence
ERESTART	Interrupted system call should be restarted
ESTRPIPE	Streams pipe error
EUSERS	Too many users
ENOTSOCK	Socket operation on non-socket
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
EPROTOTYPE	Protocol wrong type for socket
ENOPROTOOPT	Protocol not available
EPROTONOSUPPORT	Protocol not supported
ESOCKTNOSUPPORT	Socket type not supported
EOPNOTSUPP	Operation not supported on transport endpoint
EPFNOSUPPORT	Protocol family not supported
EAFNOSUPPORT	Address family not supported by protocol
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Cannot assign requested address
ENETDOWN	Network is down
ENETUNREACH	Network is unreachable

下页继续

表 4.2 – 续上页

错误码	说明
ENETRESET	Network dropped connection because of reset
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by peer
ENOBUFS	No buffer space available
EISCONN	Transport endpoint is already connected
ENOTCONN	Transport endpoint is not connected
ESHUTDOWN	Cannot send after transport endpoint shutdown
ETOOMANYREFS	Too many references: cannot splice
ETIMEDOUT	Connection timed out
ECONNREFUSED	Connection refused
EHOSTDOWN	Host is down
EHOSTUNREACH	No route to host
EALREADY	Operation already in progress
EINPROGRESS	Operation now in progress
ESTALE	Stale NFS file handle
EUCLEAN	Structure needs cleaning
ENOTNAM	Not a XENIX named type file
ENAVAIL	No XENIX semaphores available
EISNAM	Is a named type file
EREMOTEIO	Remote I/O error
EDQUOT	Quota exceeded
ENOMEDIUM	No medium found
EMEDIUMTYPE	Wrong medium type
ECANCELED	Operation cancelled