CSI-API(v1.7) User Manual

March 25, 2021

T-HEAD Semiconductor Co.,LTD

Address: T6, West Building, British Center, European and American Financial City (EFC), No. 1122 Xiangxiang Street, Yuhang District, Hangzhou

Postcode: 311121

Website: www.t-head.cn

Version history

Version		date
v1.7	**Describe** the first official release	2021.03.24

CSI-API(v1.7) User Manual

Chapter 1 CSI Overall

1.1 Introduction

This manual describes the CSI API (CSI Application Programming Interface)

Classified as:

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

1.2 How to use

1.2.1 Header file contains

1.2.1.1 CSI-Core

Include <csi_core.h> directly

1.2.1.2 CSI-Driver

For example to use the Timer driver, include <drv_timer.h>

For example, to use the UART driver, include

<drv_usart.h> and so on...

1.2.1.3 CSI-Kernel

Include <csi_kernel.h> directly

Chapter 2 CSI-Core API

2.1 Compilation Control

2.1.1 Macro List

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

2.1.2 Brief description

Provides basic macro definitions.

2.1.3 Interface description

2.1.3.1 __ASM

```
#define __ASM
```

Function description:

Inline assembly keywords.

2.1.3.2 __INLINE

```
#define __INLINE
```

Function description:

Inline function keyword.

2.1.3.3 __STATIC_INLINE

```
#define __STATIC_INLINE
```

Function description:

Static inline function keyword.

2.1.3.4 __ALWAYS_STATIC_INLINE

```
#define __ALWAYS_STATIC_INLINE
```

Function description:

Force static inline function keyword.

2.1.4 Example:

```
__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 List of functions

2.2.2 Brief description

Provides the basic operations of tightly coupled vector interrupt controller (VIC), including interrupt enable and disable, interrupt priority setting, interrupt status setting and acquisition, etc.

2.2.3 Interface description

2.2.3.1 csi_vic_enable_irq

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

Function description:

Enable interrupts.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.2 csi_vic_disable_irq

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

Function description:

Interrupts are disabled.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.3 csi_vic_enable_sirq

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

Function description:

Enable safe interrupts.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.4 csi_vic_disable_sirq

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

Function description:

Safe interrupts are disabled.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.5 csi_vic_get_enabled_irq

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

Function description:

Determine if the interrupt is enabled.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

1: Enable. 0:

Disabled.

2.2.3.6 csi_vic_get_pending_irq

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

Function description:

Determine whether the interrupt is in the waiting state.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

1: Wait state. 0: No wait

state.

2.2.3.7 csi_vic_set_pending_irq

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

Function description:

Set interrupt into wait state.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.8 csi_vic_clear_pending_irq

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

Function description:

Clears the wait state of an interrupt.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.9 csi_vic_set_wakeup_irq

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

Function description:

Set interrupt as wake-up interrupt.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.10 csi_vic_get_wakeup_irq

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

Function description:

Determine whether the interrupt is a wake-up interrupt.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

1: Wake-up interrupt. 0:

Not wake-up interrupt.

2.2.3.11 csi_vic_clear_wakeup_irq

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

Function description:

Description of wake-up function for clear interrupt.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

none.

2.2.3.12 csi_vic_get_active

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

Function description:

Determine if the interrupt is being serviced.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

1: Response status.

0: Not responding status.

2.2.3.13 csi_vic_set_threshold

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

Function description:

Set the VIC threshold.

parameter:

VectThreshold: Indicates the interrupt vector number corresponding to the priority threshold (Note: When the VIC finds that the CPU exits from the interrupt service routine corresponding to VECTTHRESHOLD, the hardware will clear the interrupt priority threshold valid bit).

PrioThreshold: Indicates the priority threshold for interrupt preemption.

return value:

none.

2.2.3.14 csi_vic_set_prio

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

Function description:

Set interrupt priority.

parameter:

IRQn: Interrupt number (non-exception number).

priority: interrupt priority (0~3), the smaller the value, the higher the priority.

return value:

none.

2.2.3.15 csi_vic_get_prio

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

Function description:

Get interrupt priority.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

Get interrupt priority.

2.2.3.16 csi_vic_set_vector

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

Function description:

Set the interrupt handler function.

parameter:

IRQn: Interrupt number (non-exception number).

handler: Interrupt handler function.

return value:

none.

2.2.3.17 csi_vic_get_vector

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

Function description:

Get the interrupt handler function.

parameter:

IRQn: Interrupt number (non-exception number).

return value:

Interrupt handler.

2.2.4 Example:

```
/* Enable interrupt 0 */
csi_vic_enable_irq(0);

/* Set the priority of interrupt No. 0 to 3 */
csi_vic_set_prio(0, 3);
```

2.3 CoreTIM

2.3.1 Function List

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

2.3.2 Brief description

Provides basic operations of tightly coupled system timers, including timer configuration and timer count value acquisition.

2.3.3 Interface description

2.3.3.1 csi_coret_config

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

Function description:

Configure CoreTIM.

parameter:

ticks: The number of ticks for a single tick.

IRQn: Interrupt number (non-exception number).

return value:

Returns 0.

2.3.3.2 csi_coret_get_load

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

Function description:

Get the loaded value of CoreTIM.

parameter:

none.

return value:

CoreTIM load value.

2.3.3.3 csi_coret_get_value

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

Function description:

Get the current count value of CoreTIM.

parameter:

none.

return value:

CoreTIM current count value.

2.3.4 Example:

```
uint32_t load, cur;

/* Enable CoreTIM and set the load value of CoreTIM to the maximum */
csi_coret_config(0xFFFFFFFF, CORET_IRQn);

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

/* Count the number of clocks passed by CoreTIM
*/ printf("spent clocks: %u\n", load - cur);
```

2.4 Cache

2.4.1 Function List

2.4.2 Brief description

Provides the basic operations of the Cache, including Cache refresh and Cache profiling functions.

2.4.3 Interface description

2.4.3.1 csi_icache_enable

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

Function description:

Enable instruction cache.

parameter:

none.

return value:

none.

2.4.3.2 csi_icache_disable

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

Function description:

Instruction cache is disabled.

parameter:

none.

return value:

none.

2.4.3.3 csi_icache_invalid

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

Function description:

Invalidate the instruction cache.

parameter:

none.

return value:

none.

2.4.3.4 csi_dcache_enable

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

Function description:

Enable data cache.

parameter:

none.

return value:

none.

2.4.3.5 csi_dcache_disable

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

Function description:

Data caching is disabled.

parameter:

none.

return value:

none.

2.4.3.6 csi_dcache_invalid

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

Function description:

Invalidate the entire data cache.

parameter:

none.

return value:

none.

2.4.3.7 csi_dcache_clean

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

Function description:

Clear the entire data cache.

parameter:

none.

return value:

none.

2.4.3.8 csi_dcache_clean_invalid

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

Function description:

Flushes and invalidates the entire data cache.

parameter:

none.

return value:

none.

2.4.3.9 csi_dcache_invalid_range

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

Function description:

Invalidate the data cache row by row.

parameter:

addr: The address that needs to be invalidated.

dsize: address length.

return value:

none.

2.4.3.10 csi_dcache_clean_range

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

Function description:

Clear the data cache row by row.

parameter:

addr: The address that needs to be cleared.

dsize: address length.

return value:

none.

2.4.3.11 csi_dcache_clean_invalid_range

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

Function description:

Clear and invalidate data cache by line.

parameter:

addr: The address that needs to be cleared and invalidated.

dsize: address length.

return value:

none.

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_
,ÿt enable)
```

Function description:

Sets the scope of the cache.

parameter:

index: The cache area number.

baseAddr: Action address. size:

address length. enable: Whether to

enable this area. 1: Enable; 0: Disable.

return value:

none.

2.4.3.13 csi_cache_enable_profile

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

Function description:

Cache-enabled profiling function description.

parameter:

none.

return value:

none.

2.4.3.14 csi_cache_disable_profile

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

Function description:

Cache-disabled profiling function descriptions.

parameter:

none.

return value:

none.

2.4.3.15 csi_cache_reset_profile

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

Function description:

Reset cached analysis functions.

parameter:

none.

return value:

none.

2.4.3.16 csi_cache_get_access_time

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

Function description:

Get the number of accesses to the cache.

parameter:

none.

return value:

The number of cache accesses, incremented by 1 every 256 times.

2.4.3.17 csi_cache_get_miss_time

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

Function description:

Get the number of cache misses.

parameter:

none.

return value:

The number of cache misses, incremented by 1 every 256 times.

2.4.4 Example:

Example of tightly coupled cache:

```
uint32_t access, miss;

/* Set the range of cache scope 0 to: 0x0-0x1000 */ csi_cache_set_range(0, 0x0,
CACHE_CRCR_4K, 1); /* Set the range of cache scope 1 to: 0x10000000-0x10080000 */

csi_cache_set_range(1, 0x10000000, CACHE_CRCR_512K, 1);

/* Enable instruction cache and data cache */
```

(continued on next page)

(continued from previous page)

```
csi_icache_enable();
csi_dcache_enable();

/* Enable and reset the cache profile function */
csi_cache_enable_profile(); csi_cache_reset_profile();


/* Get the number of cache access and miss */
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 List of functions

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

2.5.2 Brief description

Provides the basic operation of the MPU module, this part of the interface can be used on the CPU with MPU.

2.5.3 Interface description

2.5.3.1 csi_mpu_enable

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

Function description:

Enable MPU.

parameter:

none.

return value:

none.

2.5.3.2 csi_mpu_disable

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

Function description:

Banned MPU.

parameter:

none.

return value:

none.

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)
```

Function description:

Configure the region of the MPU.

parameter:

idx: MPU region number.

base_addr: Action base address.

size: *address length. See table region_size_e for definition* . attr: attribute. enable:

Whether to enable.

return value:

none.

region_size_e

name	definition	Remark
REGION_SIZE_128B	The size of the protected area is 128B, only 801/802 is supported	
REGION_SIZE_256B	The size of the protected area is 256B, only 801/802 is supported	
REGION_SIZE_512B	The size of the protected area is 512B, only 801/802 is supported	
REGION_SIZE_1KB	The size of the protected area is 1KB. Only 801/802 is supported	
REGION_SIZE_2KB	The size of the protected area is 2KB, only 801/802 is supported	
REGION_SIZE_4KB	The size of the protected area is 4KB	
REGION_SIZE_8KB	The size of the protected area is 8KB	
REGION_SIZE_16KB	The size of the protected area is 16KB	
REGION_SIZE_32KB	The size of the protected area is 32KB	
REGION_SIZE_64KB	The size of the protected area is 64KB	
REGION_SIZE_128KB	The size of the protected area is 128KB	
REGION_SIZE_256KB	The size of the protected area is 256KB	
REGION_SIZE_512KB	The size of the protected area is 512KB	
REGION_SIZE_1MB	The size of the protected area is 1MB	
REGION_SIZE_2MB	The size of the protected area is 2MB	
REGION_SIZE_4MB	The size of the protected area is 4MB	
REGION_SIZE_8MB	The size of the protected area is 8MB	
REGION_SIZE_16MB	The size of the protected area is 16MB	
REGION_SIZE_32MB	The size of the protected area is 32MB	
REGION_SIZE_64MB	The size of the protected area is 64MB	
REGION_SIZE_128MB	The size of the protected area is 128MB	
REGION_SIZE_256MB	The size of the protected area is 256MB	
REGION_SIZE_512MB	The size of the protected area is 512MB	
REGION_SIZE_1GB	The size of the protected area is 1GB	
REGION_SIZE_2GB	The size of the protected area is 2GB	
REGION_SIZE_4GB	The size of the protected area is 4GB	

mpu_region_attr_t

name	describe	definition
nx	execute permission	CK610 does not support nx attribute • 0: executable • 1: Not executable
ap	Read and write permissions	access_permission_e: • AP_BOTH_INACCESSIBLE: superuser and normal user are not accessible • AP_SUPER_RW_USER_INACCESSIBLE: super use Users can read and write, but ordinary users cannot access • AP_SUPER_RW_USER_RDONLY: Super user readable write, read-only for normal users • AP_BOTH_RW: Both superusers and normal users can read and write
s	security properties	• 0: not secure • 1: Safe

2.5.3.4 csi_mpu_enable_region

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

Function description:

Enables a region of the MPU.

parameter:

idx: MPU region number.

return value:

none.

2.5.3.5 csi_mpu_disable_region

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

Function description:

Disable a region of the MPU.

parameter:

idx: MPU region number.

return value:

none.

2.5.4 Example:

```
mpu_region_attr_t attr;

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

/* Configure the range of MPU scope 0 : 0x0-0x1000, and enable this scope */
csi_mpu_config_region(0, 0x0, REGION_SIZE_4KB, attr, 1);

/* Disable MPU scope 0 */
csi_mpu_disable_region(0);
```

2.6 HAD

2.6.1 Function List

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

2.6.2 Brief description

Provides basic operations of the HAD (Hardware Assisted Debug) module, including functions such as sending and receiving data.

2.6.3 Interface description

2.6.3.1 csi_had_send_char

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

Function description:

Send a character through HAD.

parameter:

ch: The character to send.

return value:

none.

2.6.3.2 csi_had_receive_char

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

Function description:

Receive a character via HAD.

parameter:

none.

return value:

character received.

2.6.3.3 csi_had_check_char

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

Function description:

Determine if there is data to receive.

parameter:

none.

return value:

1: with data; 0: without data.

2.7 IRQ

2.7.1 List of functions

2.7.2 Brief description

Provides interrupt-related interfaces.

2.7.3 Interface description

2.7.3.1 __enable_irq

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

Function description:

Enable CPU interrupts.

parameter:

none.

return value:

none.

2.7.3.2 __disable_irq

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

Function description:

Disable CPU interrupts.

parameter:

none.

return value:

none.

2.7.3.3 __enable_excp_irq

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

Function description:

Enable CPU exceptions and interrupts.

parameter:

none.

return value:

none.

2.7.3.4 __disable_excp_irq

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

Function description:

CPU exceptions and interrupts are disabled.

parameter:

none.

return value:

none.

2.7.3.5 csi_irq_save

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

Function description:

The processor status register value is saved and CPU interrupts are disabled.

parameter:

none.

return value:

PSR status value.

2.7.3.6 csi_irq_restore

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

Function description:

Restores the processor status register value.

parameter:

irq_state: PSR state value.

return value:

none.

2.7.4 Example:

```
uint32_t flags;

flags = csi_irq_save();

/* ... */

csi_irq_restore(flags);
```

2.8 MMU

2.8.1 Function List

2.8.2 Brief description

Provides the basic operation of the MMU (Memory Management Unit) module, and this part of the interface can be used on CPUs with MMU.

2.8.3 Interface description

2.8.3.1 csi_mmu_enable

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

Function description:

Enable MMU.

parameter:

none.

return value:

none.

2.8.3.2 csi_mmu_disable

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

Function description:

Banned MMU.

parameter:

none.

return value:

none.

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_tÿ ,ÿattr)
```

Function description:

Set the tlb table entry mapping.

parameter:

vaddr: virtual address page. paddr:

Physical address page.

acid: ACIDÿ

attr: *read and write attributes, see page_attr_t* definition.

return value:

none.

page_attr_t:

Attributes	describe	definition
global	global properties	• 0: the current ASID is valid • 1: globally valid
valid	legal attributes	• 0: the entry is invalid • 1: Entry is valid
writeable	writable property	• 0: page writable • 1: Page not writable
cacheable	Cacheable properties	• 0: The page cannot be cached • 1: The page can be cached
is_secure	security properties	• 0: The page supports secure access • 1: The page does not support secure access
strong_order	Read and write access order properties	• 0: The sequence of read and write access to memory on the bus and the sequence of program flow Read and write access order is different • 1: Read and write access sequence and program of memory on this page bus The read and write access order of the stream is the same
bufferable	Buffer property	• 0: page can be buffered • 1: The page cannot be buffered

2.8.3.4 csi_mmu_set_pagesize

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

Function description:

Sets the size of the map page.

parameter:

size: *The size of the mapped page, see page_size_e* definition.

return value:

none.

page_size_e:

name	definition	Remark
PAGE_SIZE_4KB	page size is 4KB	
PAGE_SIZE_16KB	page size is 16KB	
PAGE_SIZE_64KB	page size is 64KB	
PAGE_SIZE_256KB	page size is 256KB	
PAGE_SIZE_1MB	page size is 1MB	
PAGE_SIZE_4MB	page size is 4MB	
PAGE_SIZE_4MB	page size is 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_tÿ
,ÿ*mel1)
```

Function description:

Read TLB entries by index.

parameter:

index: TLB index number.

meh: Store the returned MEH register value.

mel0: Stores the returned MEL0 register value.

mel1: Stores the returned MEL1 register value.

return value:

none.

2.8.3.6 csi_mmu_invalid_tlb_all

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

Function description:

Invalidate all TLB entries.

parameter:

none.

return value:

none.

2.8.3.7 csi_mmu_invalid_tlb_by_index

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

Function description:

Invalidate TLB entries by index.

parameter:

index: TLB index number.

return value:

none.

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)
```

Function description:

Invalidate TLB entries by virtual addresses.

parameter:

vaddr: The virtual address to be invalidated. asid:

The ASID number to which the virtual address belongs.

return value:

none.

2.8.4 Example:

```
/* Create a 4KB page: virtual address 0x0 is mapped to physical address 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;
```

(continued on next page)

(continued from previous page)

```
attr.bufferable = 0;

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

2.9 TCM

2.9.1 List of functions

- *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 Brief description

Provides an interface for setting the TCM.

2.9.3 Interface description

2.9.3.1 csi_itcm_enable

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

Function description:

Enable ITCM.

parameter:

none.

return value:

none.

2.9.3.2 csi_dtcm_enable

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

Function description:

Enable DTCM.

parameter:

none.

return value:

none.

2.9.3.3 csi_itcm_disable

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

Function description:

Ban ITCM.

parameter:

none.

return value:

none.

2.9.3.4 csi_dtcm_disable

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

Function description:

DTCM is prohibited.

parameter:

none.

return value:

none.

2.9.3.5 csi_itcm_slave_access_enable

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

Function description:

Enable slave access to ITCM.

parameter:

none.

return value:

none.

2.9.3.6 csi_itcm_slave_access_disable

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

Function description:

Slave access to ITCM is prohibited.

parameter:

none.

return value:

none.

2.9.3.7 csi_dtcm_slave_access_enable

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

Function description:

Enable DTCM slave access.

parameter:

none.

return value:

none.

2.9.3.8 csi_dtcm_slave_access_disable

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

Function description:

Slave access to DTCM is prohibited.

parameter:

none.

return value:

none.

2.9.3.9 csi_itcm_get_size

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

Function description:

Get the effect size of ITCM.

parameter:

none.

return value:

none.

2.9.3.10 csi_dtcm_get_size

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

Function description:

Gets the effect size of the DTCM.

parameter:

none.

return value:

none.

2.9.3.11 csi_itcm_get_delay

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

Function description:

Get access delay for ITCM.

parameter:

none.

return value:

Access delay.

2.9.3.12 csi_dtcm_get_delay

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

Function description:

Get the access delay of the DTCM.

parameter:

none.

return value:

Access delay.

2.9.3.13 csi_itcm_set_base_addr

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

Function description:

Sets the base address where ITCM acts.

parameter:

base address.

return value:

none.

2.9.3.14 csi_dtcm_set_base_addr

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

Function description:

Sets the base address where the DTCM acts.

parameter:

base address.

return value:

none.

2.9.4 Example:

```
/* 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 EXC

2.10.1 List of functions

2.10.2 Brief description

Provides an interface for setting ECC.

2.10.3 Interface description

2.10.3.1 csi_ecc_enable

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

Function description:

Enable ECC function.

parameter:

none.

return value:

none.

2.10.3.2 csi_ecc_disable

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

Function description:

The ECC function is disabled.

parameter:

none.

return value:

none.

2.10.3.3 csi_ecc_enable_error_fix

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

Function description:

Enable the error recovery function of ECC.

parameter:

none.

return value:

none.

2.10.3.4 csi_ecc_disable_error_fix

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

Function description:

Disable the error repair function of ECC.

parameter:

none.

return value:

none.

2.10.3.5 csi_ecc_inject_error

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

Function description:

ECC injection error.

parameter:

type: ECC *error type, see ecc_error_type_e* definition.

ramid: ECC RAM *index, see ecc_ramid_e* definition.

return value:

none.

ecc_error_type_e:

Type description		Remark
ECC_ERROR_CORRECTABLE repairable error		
ECC_ERROR_FATAL	unfixable error	

ecc_ramid_e:

type	describe	Remark
ECC_ICACHE_TAG_RAM instruction cache tag memory		
ECC_ICACHE_DATA_RAM instruction cache data memory		
ECC_DCACHE_TAG_RAM data cache tag memory		
ECC_DCACHE_DATA_RAM data cache data memory		
ECC_ITCM_RAM	ITCM memory	
ECC_DTCM_RAM	DTCM memory	

2.10.3.6 csi_ecc_get_error_info

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

Function description:

Get ECC error information.

parameter:

info: ECC *error information, see ecc_error_info_t* definition.

return value:

none.

ecc_error_info_t:

type	Describe	Remark
uint32_t erraddr	the wrong address	
uint32_t index	RAM index index bits: ICACHE DATA: start from addr[3] DACHE DATA: start from addr[2] ICACHE TAG/DCACHE TAG: start from addr[5] TCM: start from addr[3]	
uint8_t way	ICACHE/DCACHE WAY index bit ecc_ramid_e	
ramid:8 ECC RAM index		
ecc_error_type_e ecc_type:8	ECC error type	

2.11 Other

2.11.1 Function List

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

2.11.2 Brief description

Provides read and write functions for CPU control registers and status registers.

2.11.3 Interface description

2.11.3.1 csi_system_reset

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

Function description:

Reset the CPU.

parameter:

none.

return value:

none.

2.11.3.2 __get_REGISTER

```
__ALWAYS_STATIC_INLINE __get_REGISTER(void)
```

Function description:

Get the value of the register.

parameter:

none.

return value:

register value.

2.11.3.3 __set_REGISTER

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

Function description:

Set the value of the register.

parameter:

val: register value.

return value:

none.

Registers supported by __set_REGISTER and __get_REGISTER

2.11.3.4 CSKY ARCH REGISTERS:

CPUs supported by	register names	Remark
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	
GENDER	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	
ME	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	

Continue on next page

Table 2.1 - Continued from previous page

CPUs supported by register names		Remark
WSSR	TEE CPU	
WRCR	TEE CPU	
DCR	TEE CPU	
PCR	TEE CPU	
EBR	TEE CPU	

2.11.3.5 RISCV ARCH REGISTERS:

register name	Supported RISCV CPU Notes	
MSTATUS	All	
MASS	All	
MY	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		

Continue on next page

Table 2.2 - Continued from previous page

register name	Supported RISCV CPUs	Notes
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
*/
```

Chapter 3 CSI-Driver API

3.1 Timer

3.1.1 Function List

3.1.2 Brief description

Timer Hardware timers are clock-driven counters that can be used to implement timing or timing interrupt functions. The counter can be incremented or decremented, incrementing or decrementing the counter by 1 every clock. The timer supports free running and reload modes. The difference is that the reload mode will reload a preset value to the counter when the counter counts to a specified value, and can trigger an interrupt.

3.1.3 Interface description

3.1.3.1 csi_timer_initialize

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

Function description:

Initialize the corresponding Timer instance by index number, and return the handle of the timer instance.

parameter:

idx: Controller number.

cb_event: interrupt callback function.

return value:

Returns the instance handle on success, NULL on failure.

3.1.3.2 csi_timer_uninitialize

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

Function description:

The timer instance is deinitialized, this interface will stop the ongoing work of the timer instance (if any), and release the related software and hardware resources.

parameter:

handle: The instance handle. instance handle.

return value:

error code.

3.1.3.3 csi_timer_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.1.3.4 csi_timer_config


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

Function description:

Configure the working mode of the Timer instance.

parameter:

handle: The instance handle.

mode: The working mode of Timer, refer to the definition of timer_mode_e.

return value:

error code.

timer_mode_e:

name definition		Remark
TIMER_MODE_FREE_RUNNING timer Free running timer Repeated	timing operation	
TIMER_MODE_RELOAD		

3.1.3.5 csi_timer_set_timeout

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

Function description:

Timer timeout setting.

parameter:

handle: The instance handle.

timeout: Timer timeout time in microseconds (us).

return value:

error code.

3.1.3.6 csi_timer_start

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

Function description:

Timer starts.

parameter:

handle: The instance handle.

return value:

error code.

3.1.3.7 csi_timer_stop

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

Function description:

Timer stops.

parameter:

handle: The instance handle.

return value:

error code.

3.1.3.8 csi_timer_suspend

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

Function description:

Timer is paused.

parameter:

handle: The instance handle.

return value:

error code.

3.1.3.9 csi_timer_resume

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

Function description:

Timer timing resumes.

parameter:

handle: The instance handle.

return value:

error code.

3.1.3.10 csi_timer_get_current_value

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

Function description:

Get the current timer value of Timer.

parameter:

handle: The instance handle.

value: used to store the current timer value of the Timer.

return value:

error code.

3.1.3.11 csi_timer_get_status

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

Function description:

Get the status of the Timer.

parameter:

handle: The instance handle.

return value:

Timer status.

3.1.3.12 csi_timer_get_load_value

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

Function description:

Get the Timer's Load register value.

parameter:

handle: The instance

handle. value: The obtained load register value is returned on value.

return value:

error code.

3.2 USART

3.2.1 Function List

- *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*

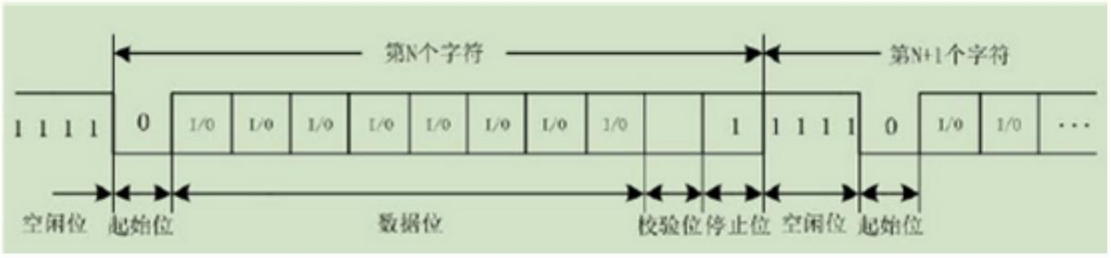
3.2.2 Brief description

USART (Universal Synchronous Asynchronous Receiver/Transmitter) is a synchronous or asynchronous serial communication bus interface. When using synchronous mode, a synchronous clock needs to be provided. When using asynchronous mode (UART), no synchronous clock is required. The sender and receiver send and receive according to a strict format (baud rate and data frame format).

Features of USART:

- The bus remains high when idle
- 5~9 data bits, low bit first • A start bit • Optional parity bit
- Optional 0.5, 1, 1.5, 2b stop bits

The transmission timing of USART is described as follows:



3.2.3 Interface description

3.2.3.1 csi_usart_initialize

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

Function description:

Initialize the corresponding usart instance through the device number, and return the handle of the usart instance.

parameter:

idx: device number

cb_event: event callback function of usart instance (usually executed in interrupt context). See usart_event_cb_t for the prototype definition of the callback function.

return value:

NULL: Initialization failed.

Miscellaneous: The instance handle when initialization is successful.

usart_event_cb_t:

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

Where idx is the device number, and event is the event type passed to the callback function. The usart callback event enumeration type usart_event_e is defined as follows:

USART_EVENT_SEND_COMPLETE data send completion event	
USART_EVENT_RECEIVE_COMPLETE data reception completion event	
USART_EVENT_TRANSFER_COMPLETE data transfer completion event	
USART_EVENT_TX_COMPLETE Data transmission completion event	
USART_EVENT_TX_UNDERFLOW	Data send overflow event
USART_EVENT_RX_OVERFLOW	Data receive overflow event
USART_EVENT_RX_TIMEOUT	Data receive timeout event
USART_EVENT_RX_BREAK	Data reception interrupt event
USART_EVENT_RX_FRAMING_ERROR Frame data receive error event	
USART_EVENT_RX_PARITY_ERROR Data receive parity error event	
USART_EVENT_CTS	CTS status changed
USART_EVENT_DSR	DSR state changed
USART_EVENT_DCD	DCD status changed
USART_EVENT_RI	RI status changed
USART_EVENT_RECEIVED	Data is received and stored in USART FIFO, Functions such as receive can be called to read

---

3.2.3.2 csi_usart_uninitialize

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

Function description:

The usart instance is deinitialized. This interface stops the ongoing transfer of the usart instance (if any) and releases the related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.2.3.3 csi_usart_get_capabilities

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

Function description:

Get the capabilities supported by the usart instance.

parameter:

idx: device number.

return value:

A structure describing the usart capability.

3.2.3.4 csi_usart_send

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

Function description:

usart starts data sending.

parameter:

handle: The instance handle.

data: The buffer address of the data to be sent.

num: The length of the data to be sent.

return value:

error code.

3.2.3.5 csi_usart_abort_send

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

Function description:

usart data transmission terminated.

parameter:

handle: The instance handle.

return value:

error code.

3.2.3.6 csi_usart_receive

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

Function description:

usart starts data reception.

parameter:

handle: The instance handle.

data: The buffer address of the data to be received.

num: The length of the data to be received.

return value:

error code.

3.2.3.7 csi_usart_receive_query

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

Function description:

The query mode reads a certain amount of data from the USART.

parameter:

handle: The instance handle.

data: The buffer address of the data to be received.

num: Expected length of received data.

return value:

error code.

3.2.3.8 csi_usart_abort_receive

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

Function description:

usart Data reception terminated.

parameter:

handle: The instance handle.

return value:

error code.

3.2.3.9 csi_usart_transfer

```
int32_t csi_usart_transfer(usart_handle_t handle, const void *data_out, void *data_in, uint32_tÿ
,ÿnum)
```

Function description:

usart starts data transfer, note that it is a synchronous transfer.

parameter:

handle: The instance handle.

data_out: The buffer address of the data to be sent. data_in:

The buffer address of the data to be received. num: The

length of the data.

return value:

error code.

3.2.3.10 csi_usart_abort_transfer

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

Function description:

usart Data transfer terminated.

parameter:

handle: The instance handle.

return value:

error code.

3.2.3.11 csi_usart_get_status

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

Function description:

Get the state of usart at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of the usart state. *See usart_status_t* definition for details.

usart_status_t:

name	Define	Remark
tx_busy: 1	data sending busy	
rx_busy: 1	data reception busy	
tx_underflow: 1 Data transmission overflow		
rx_overflow: 1	Data receive overflow	
rx_break: 1	Data reception interrupted	
rx_framing_error: 1 frame data error		
rx_parity_error: 1 Data receive parity error		
tx_enable: 1	send enable	
rx_enable: 1	receive enable	

3.2.3.12 csi_usart_flush

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

Function description:

Clear the usart data cache.

parameter:

handle: The instance

handle. type: usart *flush data type, see usart_flush_type_e* definition.

return value:

error code.

usart_flush_type_e:

name definition		Remark
USART_FLUSH_WRITE clear write cache space		
USART_FLUSH_READ clears read cache space		

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)
```

Function description:

Switch USART interrupt.

parameter:

handle: The instance

handle. type: *interrupt type. See usart_intr_type_e* definition for

details. flag: 0: off; 1: on.

return value:

error code.

usart_intr_type_e:

name definition		Remark
USART_INTR_WRITE usart write interrupt		
USART_INTR_READ usart read interrupt		

3.2.3.14 csi_usart_config_baudrate

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

Function description:

Configure the baud rate of the usart instance.

parameter:

handle: The instance handle.

baud: baud rate.

return value:

error code.

3.2.3.15 csi_usart_config_mode

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

Function description:

Configure the working mode of the usart instance.

parameter:

handle: The instance handle.

mode: *the working mode of* usart . *See usart_mode_e* definition for details.

return value:

error code.

usart_mode_e:

name	Define	Remark
USART_MODE_ASYNCHRONOUS	usart asynchronous mode	
USART_MODE_SYNCHRONOUS_MASTER usart synchronous full duplex master mode		
USART_MODE_SYNCHRONOUS_SLAVE usart synchronous full-duplex slave mode usart half-duplex single-wire mode		
USART_MODE_SLAVE_WIRE usart IRDA mode		
USART_MODE_SINGLE_IRDA		
USART_MODE_SINGLE_SMART_CARD usart smart card mode		

3.2.3.16 csi_usart_config_parity

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

Function description:

Configure the parity mode of the usart instance.

parameter:

handle: The instance handle.

parity: the parity of usart , *see the definition of usart_parity_e .*

return value:

error code.

usart_parity_e:

name definition		Remark
USART_PARITY_NONE no parity		
USART_PARITY_EVEN Even parity		
USART_PARITY_ODD odd parity check bit is set to 1		
USART_PARITY_1 parity bit set		
USART_PARITY_0 to 0		

3.2.3.17 csi_usart_config_stopbits

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

Function description:

Configure the stop bit pattern for the usart instance.

parameter:

handle: The instance handle.

stopbit: *the stop bit of usart, see* the definition of usart_stop_bits_e.

return value:

error code.

usart_stop_bits_e:

name	define	Remark
USART_STOP_BITS_1	1 stop bit	
USART_STOP_BITS_2 2 stop bits		
USART_STOP_BITS_1_5 1.5 stop bits		
USART_STOP_BITS_0_5 0.5 stop bits		

3.2.3.18 csi_usart_config_databits

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

Function description:

parameter:

handle: The instance handle.

databits: *The data bit width of usart, see the definition of usart_data_bits_e .*

return value:

error code.

usart_data_bits_e:

name definition		Remark
USART_DATA_BITS_5 5-bit data bit width		
USART_DATA_BITS_6 6-bit data bit width		
USART_DATA_BITS_7 7-bit data bit width		
USART_DATA_BITS_8 8-bit data bit width		
USART_DATA_BITS_9 9-bit data bit width		

3.2.3.19 csi_usart_getchar

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

Function description:

Read a byte from usart.

parameter:

handle: The instance handle. .

ch: Returns the bytes read.

return value:

error code.

3.2.3.20 csi_usart_putchar

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

Function description:

Send a byte from usart.

parameter:

handle: The instance handle.

ch: The byte content to be sent.

return value:

error code.

3.2.3.21 csi_usart_get_tx_count

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

Function description:

Get the number of data sent by the device instance last time.

parameter:

handle: The instance handle.

return value:

The number of data sent.

3.2.3.22 csi_usart_get_rx_count

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

Function description:

Get the number of data received by the device instance last time.

parameter:

handle: The instance handle.

return value:

The number of data received.

3.2.3.23 csi_usart_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.2.3.24 csi_usart_config_flowctrl

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

Function description:

Configure flow control for the usart instance.

parameter:

handle: The instance handle.

flowctrl_type: *flow control type, see* the definition of usart_flowctrl_type_e.

return value:

error code.

usart_flowctrl_type_e:

name	definition	Remark
USART_FLOWCTRL_NONE without flow control		
USART_FLOWCTRL_CTS	Support CTS flow control	
USART_FLOWCTRL_RTS supports RTS flow control		
USART_FLOWCTRL_CTS_RTS supports both CTS and RTS flow control		

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)
```

Function description:

Configure the communication polarity and phase of the usart instance.

parameter:

handle: The instance handle.

cpol: *the polarity of usart, see* the definition of usart_cpol_e.

cpha: *the phase of usart, see* the definition of usart_cpha_e.

return value:

error code.

usart_cpol_e:

name	definition	Remark
USART_CPOL0 data is captured on rising edge		
USART_CPOL1 data is captured on falling edge		

uset_cpha_e:

name definition		Remark
USART_CPHA0 data is sampled on edge 0		
USART_CPHA1 data is sampled on the 1st edge		

3.2.3.26 csi_usart_control_tx

int32_t csi_usart_control_tx(usart_handle_t handle, uint32_t enable)

Function description:

Controls the send enable of the usart instance.

parameter:

handle: The instance handle.

enable: 1 - data is allowed to be sent, 0 - data is not allowed to be sent.

return value:

error code.

3.2.3.27 csi_usart_control_rx

int32_t csi_usart_control_rx(usart_handle_t handle, uint32_t enable)

Function description:

Controls the receive enable of the usart instance.

parameter:

handle: The instance handle.

enable: 1 - allow to receive data, 0 - not allow to receive data.

return value:

error code.

3.2.3.28 csi_usart_control_break

int32_t csi_usart_control_break(usart_handle_t handle, uint32_t enable)

Function description:

Controls the sending of break frames for the usart instance.

parameter:

handle: The instance handle.

enable: 1 - start sending break frames, 0 - stop sending break frames.

return value:

error code.

3.3 GPIO

3.3.1 Function List

3.3.2 Brief description

GPIO (General Purpose Input Output) general-purpose input/output interface provides a separate high and low level state control function for each signal pin.
function, or the function of reading the high and low level status of the pin.

GPIO can be configured as input mode or output mode. When configured as output mode, the high and low level states of the pins are controlled by software. The output mode can support two types of open-drain output or push-pull output. When configured as input mode, the high and low level states of the pins are controlled by external circuits, and software can read the high and low level states of the pins. Input mode can support properties with pull-up and pull-down resistors.

Input mode also supports configuring GPIOs as interrupt trigger sources. Interrupt trigger mode generally includes high level trigger, low level trigger, double edge trigger, falling Five modes of edge trigger and rising edge trigger.

3.3.3 Interface description

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)
```

Function description:

Initialize the corresponding gpio pin instance through the input pin number, and return the handle of the gpio pin instance.

parameter:

gpio_pin: pin number.

cb_event: Corresponds to pin interrupt callback function.

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

3.3.3.2 csi_gpio_pin_uninitialize

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

Function description:

The gpio pin instance is deinitialized. This interface stops the ongoing transmission of the gpio instance (if any) and releases related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.3.3.3 csi_gpio_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

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)
```

Function description:

Configure the working mode of the gpio pin instance.

parameter:

handle: The instance handle.

mode: The working mode of gpio , *see the definition of gpio_mode_e .*

return value:

error code.

gpio_mode_e:

name	define remarks	
GPIO_MODE_PULLNONE does not operate pull-up operation		
GPIO_MODE_PULLUP		
GPIO_MODE_PULLDOWN pull-down operation		
GPIO_MODE_OPEN_DRAIN open-drain operation		
GPIO_MODE_PUSH_PULL push-pull operation		

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);
```

Function description:

parameter:

handle: The instance

handle. dir: *The direction of the gpio port, see the definition of gpio_direction_e .*

return value:

error code.

gpio_direction_e:

name definition remarks		
GPIO_DIRECTION_INPUT input mode		
GPIO_DIRECTION_OUTPUT output mode		

3.3.3.6 csi_gpio_pin_write

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

Function description:

Set the level state of the gpio pin.

parameter:

handle: The instance handle.

value: corresponds to the value in the pin.

return value:

error code.

3.3.3.7 csi_gpio_pin_read

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

Function description:

Get the level state of the pin.

parameter:

handle: The instance handle.

value: The buffer address that stores the level state of the pin.

return value:

error code.

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)
```

Function description:

Set the interrupt mode of the pin.

parameter:

handle: The instance handle.

mode: Interrupt mode, see :ref: *gpio_irq_mode_e <gpio_irq_mode_e>* definition. enable: Set whether the interrupt is enabled,

0-disable, 1-enable.

return value:

error code.

gpio_irq_mode_e:

name definition		Remark
GPIO_IRQ_MODE_RISING_EDGE rising edge interrupt mode		
GPIO_IRQ_MODE_FALLING_EDGE falling edge interrupt mode		
GPIO_IRQ_MODE_DOUBLE_EDGE Double edge interrupt mode		
GPIO_IRQ_MODE_LOW_LEVEL low level interrupt mode		
GPIO_IRQ_MODE_HIGH_LEVEL high level interrupt mode		

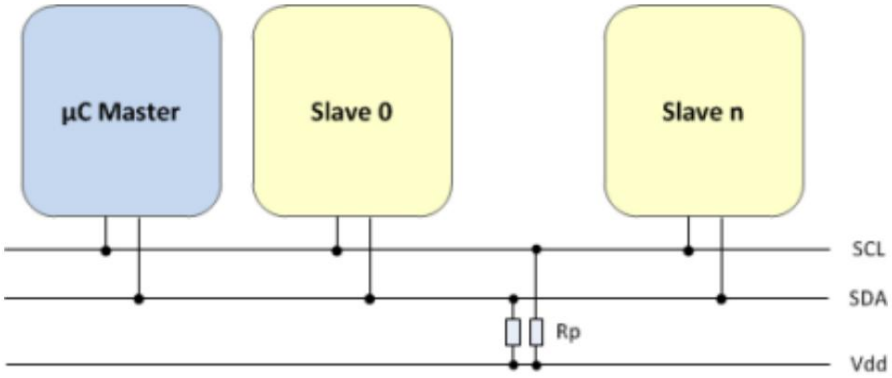
3.4 IIC

3.4.1 Function List

- *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 Brief description

I2C (Inter-Integrated Circuit, or IIC) is a serial synchronous communication bus. The I2C serial bus has two signal lines, one is the bidirectional data line SDA, and the other is the clock line SCL. All the serial data SDA connected to the I2C bus devices are connected to the SDA of the bus, and the clock line SCL of each device is connected to the SCL of the bus. The typical wiring of IIC is as follows:



The communication of the bus is controlled by the host, that is, the host is the device that transmits the data (issues a start signal), issues a clock signal, and issues a stop signal at the end of the transfer. A device sought by the master is called a slave. Each device connected to the I2C bus has a unique address for easy access by the host. The data transmission between the master and the slave can be sent by the master to the slave or from the slave to the master. I2C supports 7-bit or 10-bit slave address mode. The first byte after the start condition of the I2C bus determines which slave will be selected by the master. When the master outputs an address, each slave device in the system compares the address after the start condition with its own address. If they are the same, the slave considers itself to be addressed by the master.

3.4.3 Interface description

3.4.3.1 csi_iic_initialize

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

Function description:

Initialize the corresponding iic instance through the device number, and return the handle of the iic instance.

parameter:

idx: device number.

cb_event: The event callback function of the iic instance (usually executed in the interrupt context). See iic_event_cb_t for the definition of the callback function prototype.

The callback function type iic_event_cb_t is defined as follows:

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

Where idx is the device number, and event is the event type passed to the callback function.

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

The IIC callback event enumeration type iic_event_e is defined as follows:

name	Define	Remark
IIC_EVENT_TRANSFER_DONE	the transfer complete event	
IIC_EVENT_TRANSFER_INCOMPLETE transfer incomplete event		
IIC_EVENT_SLAVE_TRANSMIT	Send action request event from device	
IIC_EVENT_SLAVE_RECEIVE	Receive action request events from devices	
IIC_EVENT_ADDRESS_NACK	Address unanswered event	
IIC_EVENT_GENERAL_CALL	Indicates the receipt of a general call (address 0) event	
IIC_EVENT_ARBITRATION_LOST	Host Quorum Loss Event	
IIC_EVENT_BUS_ERROR	bus error event	
IIC_EVENT_BUS_CLEAR	bus clear complete event	

3.4.3.2 csi_iic_uninitialize

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

Function description:

The usart instance is deinitialized. This interface stops the ongoing transfer of the usart instance (if any) and releases the related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.4.3.3 csi_iic_get_capabilities

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

Function description:

Get the capabilities supported by the iic instance.

parameter:

idx: device number.

return value:

Structure describing iic capabilities.

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,ÿ
,ÿbool xfer_pending)
```

Function description:

Start data transmission when iic is the host.

parameter:

handle: The instance handle.

devaddr: Slave device address. data: The

buffer address of the data to be sent. num: The length

of the data to be sent. xfer_pending: Whether to send

the stop bit after the transmission is completed. 1 - stop bit is not sent, 0 - stop bit is sent.

return value:

error code.

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,ÿ
,ÿbool xfer_pending)
```

Function description:

parameter:

handle: The instance handle.

devaddr: Slave device address.

data: The buffer address of the data to be received.

num: The length of the data to be

received. xfer_pending: Whether to send stop bit after receiving is completed. 1 - stop bit is not sent, 0 - stop bit is sent.

return value:

error code.

3.4.3.6 csi_iic_slave_send

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

Function description:

iic starts data transmission when it acts as a slave.

parameter:

handle: The instance handle.

data: The buffer address of the data to be sent. num:

The length of the data to be sent.

return value:

error code.

3.4.3.7 csi_iic_slave_receive

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

Function description:

iic starts data reception when it acts as a slave.

parameter:

handle: The instance handle.

data: The buffer address of the data to be received.

num: The length of the data to be received.

return value:

error code.

3.4.3.8 csi_iic_abort_transfer

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

Function description:

Stop ongoing transfers, including sending and receiving. If the iic device is idle, do nothing.

parameter:

handle: The instance handle.

return value:

error code.

3.4.3.9 csi_iic_get_status

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

Function description:

Get the status of the IIC at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of the iic state. For the status definition of iic , *see the definition of iic_status_t .*

iic_status_t:

name	Define	Remark
busy : 1	transmit or transmit busy status bit	
mode : 1	mode bits. 1-Master, 0-Slave	
direction : 1	Transmission direction: 1-receive, 0-send	
general_call: 1 general call instructions		
arbitration_lost : 1 Host lost arbitration		
bus_error : 1	bus error	

3.4.3.10 csi_iic_power_control

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

Function description:

Configure the power mode of the iic instance.

parameter:

handle: The instance handle.

state: *The power consumption mode of iic, see the definition of csi_power_stat_e .*

return value:

error code.

3.4.3.11 csi_iic_config_mode

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

Function description:

Configure the master-slave working mode of the iic instance.

parameter:

handle: The instance handle.

mode: Master and slave working mode of iic, see the definition of iic_mode_e .

return value:

error code.

iic_mode_e:

name definition		Remark
IIC_MODE_MASTER IIC master mode		
IIC_MODE_SLAVE IIC slave mode		

3.4.3.12 csi_iic_config_speed

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

Function description:

Configure the working speed of the iic instance.

parameter:

handle: The instance handle.

speed: the speed of iic , *see the definition of iic_speed_e* .

return value:

error code.

iic_speed_e:

name	definition	Remark
I2C_BUS_SPEED_STANDARD iic standard speed (100KHz)		
I2C_BUS_SPEED_FAST	iic fast speed (400KHz)	
I2C_BUS_SPEED_FAST_PLUS iic Standard + Speed (1MHz)		
I2C_BUS_SPEED_HIGH	iic high speed (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)
```

Function description:

Configure the address mode of the iic instance.

parameter:

handle: The instance handle.

addr_mode: *The address mode of iic, see the definition of iic_address_mode_e .*

return value:

error code.

iic_address_mode_e:

name definition		Remark
I2C_ADDRESS_7BIT 7bit address mode		
I2C_ADDRESS_10BIT 10bit address mode		

3.4.3.14 csi_iic_config_slave_addr

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

Function description:

Configure the slave address of the iic instance.

parameter:

handle: The instance handle.

slave_addr: iic slave communication address.

return value:

error code.

3.4.3.15 csi_iic_get_data_count

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

Function description:

Get the number of transferred data for the iic instance.

parameter:

handle: The instance handle.

return value:

The number of data transferred.

3.4.3.16 csi_iic_send_start

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

Function description:

Send the START command.

parameter:

handle: The instance handle.

return value:

error code.

3.4.3.17 csi_iic_send_stop

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

Function description:

Send a STOP command.

parameter:

handle: The instance handle.

return value:

error code.

3.4.3.18 csi_iic_reset

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

Function description:

Reset the IIC.

parameter:

handle: The instance handle.

return value:

error code.

3.4.4 Example

3.4.4.1 IIC Example 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;
}
```

(continued on next page)

(continued from previous page)

```
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 {
```

(continued on next page)

(continued from previous page)

```
        // 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 Example 2

```
static iic_handle_t iic_handle;

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

(continued on next page)

(continued from previous page)

```
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;
        }
    }
```

(continued on next page)

(continued from previous page)

```
    // 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 Function List

- *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 Brief description

SPI (Serial Peripheral Interface) is a high-speed, full-duplex, synchronous communication bus. SPI works in a master-slave manner, usually with a master device and one or more slave devices.

The signal lines of the SPI controller are described as follows:

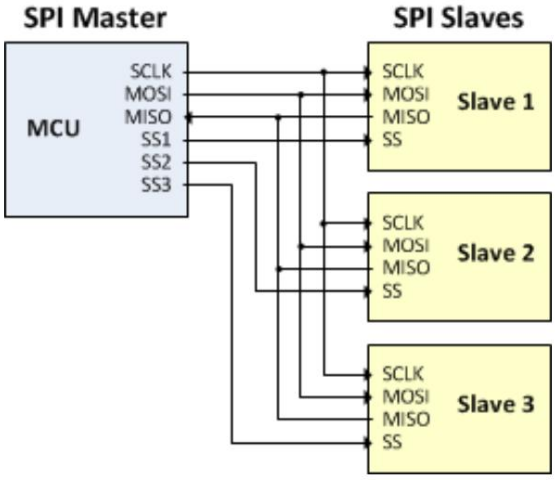
- MISO: Master device data input, slave device data output;
- MOSI: Master device data output, slave device data input;
- SCLK: clock signal, generated by the master device;
- SS: Slave device enable signal, controlled by the master device. This signal can be part of an SPI peripheral or implemented as a GPIO pin.

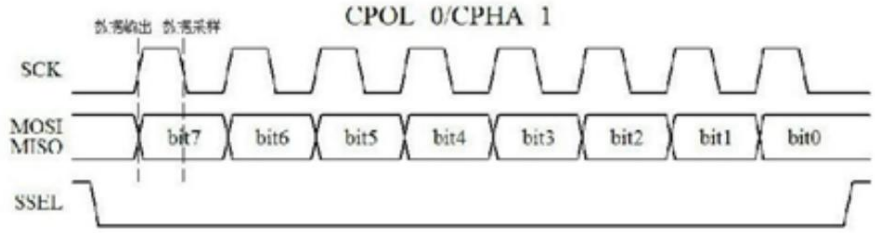
The typical wiring of SPI is as follows:

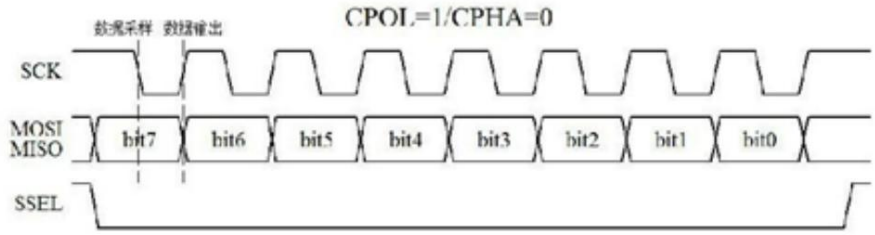
The four operating modes supported by the SPI bus depend on the combination of serial synchronous clock polarity (CPOL) and serial synchronous clock phase CPHA.

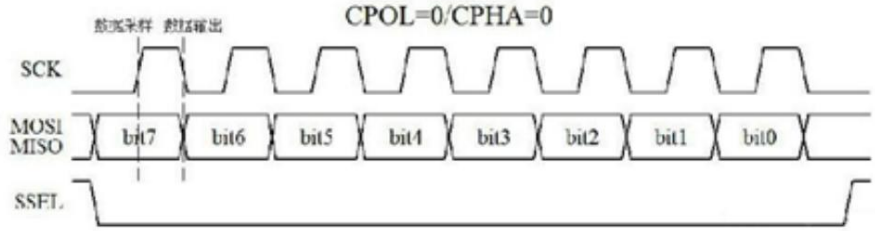
The four working modes are described as follows:

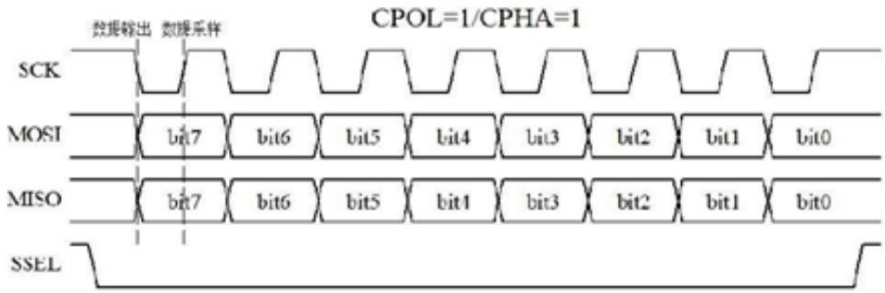
CPOL is used to determine the idle level of the SCLK clock signal. CPOL=0, the idle level is low, and when CPOL=1, the idle level is high. CPHA is used to determine the sampling time, CPHA=0, sampling on the first clock edge of each cycle, data output on the second clock edge; CPHA=1, sampling on the second clock edge of each cycle, the first clock edge One clock edge data output. The SPI master module and the peripherals communicating with it should have the same clock phase and polarity.











3.5.3 Interface description

3.5.3.1 csi_spi_initialize

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

Function description:

Initialize the corresponding spi instance through the device number, and return the handle of the spi instance.

parameter:

idx: device number.

cb_event: The event callback function of the spi instance (usually executed in the interrupt context). See spi_event_cb_t for the prototype definition of the callback function.

The callback function type iic_event_cb_t is defined as follows:

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

Where idx is the device number, event is the event type passed to the callback function, and the spi *callback event enumeration type spi_event_e*

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

spi_event_e:

name	definition	Remark
SPI_EVENT_TRANSFER_COMPLETE transfer complete event		
SPI_EVENT_TX_COMPLETE send completion event		
SPI_EVENT_RX_COMPLETE	receive completion event	
SPI_EVENT_DATA_LOST	data loss event	
SPI_EVENT_MODE_FAULT	pattern error event	

3.5.3.2 csi_spi_uninitialize

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

Function description:

The spi instance is deinitialized. This interface stops the ongoing transfer of the spi instance (if any), and releases related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.5.3.3 csi_spi_get_capabilities

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

Function description:

Get the capabilities supported by the spi instance.

parameter:

idx: device number.

return value:

A structure describing *the capabilities of spi. For the definition of spi capabilities, see the definition of spi_capabilities_t .*

spi_capabilities_t:

name		Remark
simplex :1	Definition Support unidirectional transmission mode, that is, half-duplex mode (master mode	
ti_ssi: 1	or slave mode) Support synchronous serial interface (SSI) industrial communication interface	
microwire :1	Support Microwire serial interface event_mode_fault :1 Signal mode fault event	

3.5.3.4 csi_spi_send

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

Function description:

spi starts data sending.

parameter:

handle: The instance handle.

data: The buffer address of the data to be sent. num:

The length of the data to be sent.

return value:

error code.

3.5.3.5 csi_spi_receive

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

Function description:

spi starts data reception.

parameter:

handle: The instance handle.

data: The buffer address of the data to be received.

num: The length of the data to be received.

return value:

error code.

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_ ,ÿout, uint32_t num_in)
```

Function description:

spi starts data transfer.

parameter:

handle: The instance handle.

data_out: The buffer address of the data to be sent. data_in: The

buffer address of the data to be received. num_out: The length of

the data to be sent.

num_in: The length of the data to be received.

return value:

error code.

3.5.3.7 csi_spi_abort_transfer

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

Function description:

spi data transfer terminated.

parameter:

handle: The instance handle.

return value:

error code.

3.5.3.8 csi_spi_get_status

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

Function description:

Get the state of the spi at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of spi *status, see the definition of spi_status_t* for the definition of spi status.

spi_status_t:

name	Define	Remark
busy : 1	transmit or transmit busy status bit	
data_lost : 1 data lost mode_fault : 1 mode error		

3.5.3.9 csi_spi_config_mode

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

Function description:

Configure the master-slave working mode of the spi instance.

parameter:

handle: The instance handle.

mode: *The master-slave mode of spi, see the definition of spi_mode_e .*

return value:

error code.

spi_mode_e:

name	define	Remark
SPI_MODE_INACTIVE	spi idle	
SPI_MODE_MASTER	spi full duplex master mode	
SPI_MODE_SLAVE	spi full duplex slave mode	
SPI_MODE_MASTER_SIMPLEX spi half-duplex master mode		
SPI_MODE_SLAVE_SIMPLEX spi half-duplex slave mode		

3.5.3.10 csi_spi_config_block_mode

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

Function description:

Configure the blocking mode of the spi instance.

parameter:

handle: The instance handle.

flag: 1: enable block mode; 0: disable block mode

return value:

error code.

3.5.3.11 csi_spi_config_baudrate

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

Function description:

Configure the rate of the spi instance.

parameter:

handle: The instance handle.

baud: baud rate of spi.

return value:

error code.

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)
```

Function description:

Configure the data transfer mode of the spi instance.

parameter:

handle: The instance handle.

order: *The data transmission mode of spi, see the definition of spi_bit_order_e .*

return value:

error code.

spi_bit_order_e:

name definition		Remark
SPI_ORDER_MSB2LSB High-order (MSB) first, low-order (LSB) last		
SPI_ORDER_LSB2MSB low-order (LSB) first, high-order (MSB) last		

3.5.3.13 csi_spi_config_datawidth

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

Function description:

Configure the working mode of the spi instance.

parameter:

handle: The instance handle.

datawidth: The data width of spi.

return value:

error code.

3.5.3.14 csi_spi_config_format

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

Function description:

Configure the polarity and phase mode of the spi instance.

parameter:

handle: The instance handle.

format: *The working mode of spi, see the definition of spi_format_e .*

return value:

error code.

spi_format_e:

name definition		Remark
SPI_FORMAT_CPOL0_CPHA0 idle level is low, the first clock edge sampling		
SPI_FORMAT_CPOL0_CPHA1 idle level is low, the second clock edge is sampled		
SPI_FORMAT_CPOL1_CPHA0 idle level is high, the first clock edge sampling		
SPI_FORMAT_CPOL0_CPHA1 idle level is high, the second clock edge is sampled		

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)
```

Function description:

Configure the slave selection mode for the spi instance.

parameter:

handle: The instance handle.

ss_mode: spi slave device enable mode, see the definition of :ref: *spi_ss_mode_e <spi_ss_mode_e>* .

return value:

error code.

spi_ss_mode_e:

name	Define	Remark
SPI_SS_MASTER_UNUSED	the slave device enable master mode is not enabled	
SPI_SS_MASTER_SW	Slave Device Enable Master Mode Software Mode	
SPI_SS_MASTER_HW_OUTPUT Slave enable master mode hardware output mode		
SPI_SS_MASTER_HW_INPUT Slave enable master mode hardware input mode		
SPI_SS_SLAVE_HW	Slave Device Enable Slave Mode Hardware Mode	
SPI_SS_SLAVE_SW	Slave Device Enable Slave Mode Software Mode	

3.5.3.16 csi_spi_get_data_count

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

Function description:

Get the number of data transferred last time for the device instance.

parameter:

handle: The instance handle.

return value:

The number of data transmitted last time.

3.5.3.17 csi_spi_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.5.3.18 csi_spi_ss_control

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

Function description:

Controls the select signal state of the slave instance.

parameter:

handle: The instance handle.

stat: *Device selection signal status, see spi_ss_stat_e* definition.

return value:

error code.

spi_ss_stat_e:

name	definition	Remark
SPI_SS_INACTIVE Slave select signal is invalid		
SPI_SS_ACTIVE Slave select signal is valid		

3.5.4 Example

3.5.4.1 SPI Example 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);
```

(continued on next page)

(continued from previous page)

```
    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++) {
```

(continued on next page)

(continued from previous page)

```
            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 Example 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);
```

(continued on next page)

(continued from previous page)

```
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));
```

(continued on next page)

(continued from previous page)

```
    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 Function List

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

3.6.2 Brief description

The basic principle of pulse width modulation (PWM): The control method is to control the on-off of the switching device of the inverter circuit, so that the output terminal can obtain a series of pulses of equal amplitude, and these pulses are used to replace the sine wave or the required waveform. That is, multiple pulses are generated in the half cycle of the output waveform, so that the value of each pulse is equal.

The voltage is sinusoidal and the output obtained is smooth with few low-order harmonics. By modulating the width of each pulse according to certain rules, the magnitude of the output voltage of the inverter circuit can be changed, and the output frequency can also be changed.

3.6.3 Interface description

3.6.3.1 csi_pwm_initialize

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

Function description:

Initialize the corresponding PWM controller instance by passing in the idx number, and return the handle of the controller instance.

parameter:

idx: device number.

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

3.6.3.2 csi_pwm_uninitialize

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

Function description:

PWM instance deinitialization. This interface stops the ongoing transfer of the PWM instance (if any) and releases the associated hardware and software resources.

parameter:

handle: The instance handle.

return value:

none

3.6.3.3 csi_pwm_power_control

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

Function description:

PWM power control.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

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)
```

Function description:

Configure the duty cycle of the PWM channel.

parameter:

handle: The instance handle.

channel: Channel number.

period_us: A period time in microseconds. pulse_width_us:

A high-level pulse time (in microseconds).

return value:

error code.

3.6.3.5 csi_pwm_start

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

Function description:

Start generating signals.

parameter:

handle: The instance handle.

channel: Channel number.

return value:

none.

3.6.3.6 csi_pwm_stop

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

Function description:

Stop generating the signal.

parameter:

handle: The instance handle.

channel: Channel number.

return value:

none.

3.6.4 Example

3.6.4.1 PWM Example 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);
```

(continued on next page)

(continued from previous page)

```
    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");
```

(continued on next page)

(continued from previous page)

```
    return 0;
}

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

3.7 RTC

3.7.1 Function List

- *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 Brief description

RTC (Real_Time Clock) The real-time clock provides a reliable time reference for the system. Generally, it has an independent crystal oscillator and power supply to ensure that it can still work when the main power is powered off. RTC can generally provide time in calendar format.

RTC can also provide timed interrupt function, which is used to realize timer function or function to wake up the system regularly.

3.7.3 Interface description

3.7.3.1 csi_rtc_initialize

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

Function description:

Initialize the corresponding rtc instance by the index number, and return the handle of the rtc instance.

parameter:

idx: The index number of the rtc instance.

cb_event: The event callback function of the rtc instance (usually executed in the interrupt context). See rtc_event_cb_t for the definition of the callback function prototype.

The callback function type rtc_event_cb_t is defined as follows:

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

Where idx is the device number, and event is the event type passed to the callback function.

See the definition *of rtc_evnet_e for the* rtc callback time enumeration type

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

rtc_evnet_e:

name	definition	Remark
RTC_EVENT_TIMER_INTRERRUPT generates an interrupt event		

3.7.3.2 csi_rtc_uninitialize

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

Function description:

The rtc instance is deinitialized. This interface will stop the work of the rtc instance and release related software and hardware resources.

parameter:

handle: The instance handle.

return value:

error code.

3.7.3.3 csi_rtc_power_control

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

Function description:

Configure the power mode of the rtc instance.

parameter:

handle: The instance handle.

state: *The power consumption mode of rtc, see the definition of csi_power_stat_e* .

return value:

error code.

3.7.3.4 csi_rtc_get_capabilities

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

Function description:

Get the capabilities backed by the rtc instance.

parameter:

idx: device number.

return value:

A structure describing rtc capabilities, *see rtc_capabilities_t for the definition* of rtc capabilities .

rtc_capabilities_t:

Name definition interrupt_mode: 1 supports		Remark
interrupt mode wrap_mode: 1 supports loop mode		

3.7.3.5 csi_rtc_set_time

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

Function description:

Set rtc time.

parameter:

handle: The instance handle.

rtctime: RTC *time, see the definition of struct tm .*

return value:

error code.

struct tm:

Name definition int tm_sec		Remark
seconds - the value range is [0,59] int tm_min minutes -		
the value range is [0,59] int tm_hour - the value range is		
[0,23] int tm_mday days - the value range is [1,31] int		
tm_mon month (starting from January, 0 represents		
January) - the value range is [0,11] int tm_year year, whose value starts from 0, representing the year 1900		

3.7.3.6 csi_rtc_get_time

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

Function description:

Get rtc time.

parameter:

handle: The instance handle.

rtctime: RTC *time, see the definition of struct tm .*

return value:

error code.

3.7.3.7 csi_rtc_start

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

Function description:

rtc timing starts.

parameter:

handle: The instance handle.

return value:

error code.

3.7.3.8 csi_rtc_stop

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

Function description:

rtc timing stopped.

parameter:

handle: The instance handle.

return value:

error code.

3.7.3.9 csi_rtc_get_status

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

Function description:

Get the state of rtc at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of rtc *status, see rtc_status_t for the definition* of rtc status .

rtc_status_t:

Name definition active : 1		Remark
RTC running state		

3.7.3.10 csi_rtc_set_alarm

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

Function description:

Configure the alarm date for the rtc instance.

parameter:

handle: The instance handle.

rtctime: rtc *date, see the definition of struct tm .*

return value:

error code.

3.7.3.11 csi_rtc_enable_alarm

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

Function description:

rtc Alarm reporting enable.

parameter:

handle: The instance handle.

flag: 1-enable, 0-disable.

return value:

error code.

3.7.4 Example

3.7.4.1 RTC Example 1

```
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;
```

(continued on next page)

(continued from previous page)

```
    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.
,ÿtm_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);
```

(continued on next page)

(continued from previous page)

```
    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 Example 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);
```

(continued on next page)

(continued from previous page)

```
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) {
```

(continued on next page)

(continued from previous page)

```
        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.
,ÿtm_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");
```

(continued on next page)

(continued from previous page)

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

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

3.8 WatchDog

3.8.1 Function List

- *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 Brief description

WatchDog (watchdog) is essentially a timer, which can be used to monitor the operation of the program. The watchdog can set a predetermined time. If the timer is not fed within the specified time, it will cause the system to reset. When the program is designed, the dog feeding action is embedded in the key point of the program. When the program does not run according to the specified logic for some reason (software or hardware failure), the system can be reset to ensure the availability of the system.

3.8.3 Interface description

3.8.3.1 csi_wdt_initialize

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

Function description:

Initialize the corresponding wdt instance by the index number, and return the handle of the wdt instance.

parameter:

idx: device number.

cb_event: event callback function of wdt instance (usually executed in interrupt context). See wdt_event_cb_t for the definition of the callback function prototype.

The callback function type wdt_event_cb_t is defined as follows:

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

Where idx is the device number, and event is the event type passed to the callback function. *See the wdt_event_e* definition for the wdt callback event enumeration type.

return value:

wdt_event_e:

name definition		Remark
WDT_EVENT_TIMEOUT interrupt trigger event		

3.8.3.2 csi_wdt_uninitialize

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

Function description:

The wdt instance is deinitialized. This interface stops the ongoing work of the wdt instance (if any), and releases related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.8.3.3 csi_wdt_power_control

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

Function description:

Configure the power mode of the iic instance.

parameter:

handle: The instance handle.

state: *The power mode of wdt, see the definition of csi_power_stat_e .*

return value:

error code.

3.8.3.4 csi_wdt_set_timeout

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

Function description:

wdt timeout setting.

parameter:

handle: The instance handle.

value: wdt timeout, in milliseconds.

return value:

error code.

3.8.3.5 csi_wdt_start

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

Function description:

wdt timer starts.

parameter:

handle: The instance handle.

return value:

error code.

3.8.3.6 csi_wdt_stop

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

Function description:

The wdt timer is stopped.

parameter:

handle: The instance handle.

return value:

error code.

3.8.3.7 csi_wdt_restart

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

Function description:

wdt timer restart.

parameter:

handle: The instance handle.

return value:

error code.

3.8.3.8 csi_wdt_read_current_value

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

Function description:

Get the current wdt timing value.

parameter:

handle: The instance handle.

value: used to return the current timing value.

return value:

error code.

3.8.4 Example

WDT Example 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 ;
        }
    }
```

(continued on next page)

(continued from previous page)

```
    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 List of functions

- *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 Brief description

eFlash is a non-volatile memory commonly used in embedded systems that supports on-chip execution of code. The eFlash must perform an erase action before writing.

Divide by blocks. The writing of eFlash is also called programming, which needs to be implemented through special commands.

3.9.3 Interface description

3.9.3.1 csi_eflash_initialize

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

Function description:

Initialize the corresponding eflash instance by passing in the number of devices, and return the handle of the eflash instance.

parameter:

idx: device number.

cb_event: The event callback function of the eflash instance. See eflash_event_cb_t for the prototype definition of the callback function.

The callback function type eflash_event_cb_t is defined as follows:

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

Where idx is the device number, event is the event type passed to the callback function, and the eflash callback event enumeration type.

The event type is defined in eflash_event_e .

return value:

eflash_event_e:

name definition		Remark
EFLASH_EVENT_READY eflash ready ÿÿ		
EFLASH_EVENT_ERROR eflash error event		

3.9.3.2 csi_eflash_uninitialize

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

Function description:

The eflash instance is deinitialized. This interface will stop the ongoing work of the eflash instance (if any), and release related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.9.3.3 csi_eflash_get_capabilities

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

Function description:

Get the capabilities backed by the eflash instance.

parameter:

idx: device number.

return value:

A structure describing *the capabilities of eflash, see eflash_capabilities_t for the definition of eflash capabilities .*

eflash_capabilities_t:

name	definition	Remark
event_ready : 1 support eventready data_width : 1 support		
data width erase_chip : 1 support chip erase		

3.9.3.4 csi_eflash_power_control

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

Function description:

Configure the power mode of the eflash instance.

parameter:

handle: The instance handle.

state: *The power consumption mode of eflash, see* the definition of csi_power_stat_e.

return value:

error code.

3.9.3.5 csi_eflash_read

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

Function description:

Read data from eflash.

parameter:

handle: The instance handle.

addr: The eflash address to be read.

data: The buffer address of the data to be received.

cnt: The length of the data read.

return value:

error code.

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)
```

Function description:

Write data to eflash.

parameter:

handle: The instance handle.

addr: The eflash address to be written.

data: The buffer address of the data to be programmed.

cnt: The length of the data.

return value:

error code.

3.9.3.7 csi_eflash_erase_sector

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

Function description:

Erase eflash data by sector.

parameter:

handle: The instance handle.

addr: To erase the eflash address.

return value:

error code.

3.9.3.8 csi_eflash_erase_chip

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

Function description:

Erase the data of the entire eflash.

parameter:

handle: The instance handle.

return value:

error code.

3.9.3.9 csi_eflash_get_info

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

Function description:

Get the information of the eflash at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of eflash information, *see eflash_info_t for the state definition of eflash .*

eflash_info_t:

name	define	Remark
uint32_t start	start address	
uint32_t end	end address	
uint32_t sector_count	number of sectors	
uint32_t sector_size	size of sector	
uint32_t page_size uint32_t	page size	
program_unit Minimum unit to write		
uint32_t erased_value erased value		

3.9.3.10 csi_eflash_get_status

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

Function description:

Get the state of eflash at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of eflash *status, see eflash_status_t for the definition of eflash status .*

eflash_status_t:

Name definition busy :		Remark
1 status is busy		
error : 1 write/erase error		

3.9.4 Example

3.9.4.1 eFlash Example 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 erasedÿ ,ÿvalue 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;
```

(continued on next page)

(continued from previous page)

```
    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 List of functions

3.10.2 Brief description

SPIFlash flash memory is a commonly used non-volatile memory in embedded systems that supports on-chip execution of code. SPIFlash must perform an erase operation before writing. operation, erasing is performed in blocks. The writing of SPIFlash is also called programming, which needs to be implemented through special commands.

3.10.3 Interface description

3.10.3.1 csi_spiflash_initialize

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

Function description:

Initialize the corresponding spiflash instance by passing in the number of devices, and return the handle of the spiflash instance.

parameter:

idx: device number.

cb_event: The event callback function of the spiflash instance. See spiflash_event_cb_t for the definition of the callback function prototype.

The callback function type spiflash_event_cb_t is defined as follows:

```
typedef void (*spiflash_event_cb_t)(int32_t idx, spiflash_event_eÿ
 ,ÿevent);
```

Where idx is the device number, event is the event type passed to the callback function, and the spiflash callback event enumeration type. *The event type is* defined in spiflash_event_e.

return value:

spiflash_event_e:

name	definition	Remark
SPIFLASH_EVENT_READY spiflash ready ÿÿ		
SPIFLASH_EVENT_ERROR Spiflash error event		

3.10.3.2 csi_spiflash_uninitialize

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

Function description:

The spiflash instance is deinitialized. This interface stops the ongoing work of the spiflash instance (if any), and releases related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.10.3.3 csi_spiflash_get_capabilities

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

Function description:

Get the capabilities backed by the spiflash instance.

parameter:

idx: device number.

return value:

A structure describing *the capabilities of spiflash, see spiflash_capabilities_t for the definition of spiflash capabilities .*

spiflash_capabilities_t:

name	definition	Remark
event_ready : 1 support event	ready data_width : 2 support	
data width erase_chip : 1 support chip erase		

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)
```

Function description:

Configure the running mode of the spiflash instance.

parameter:

handle: The instance handle.

line: *The running mode of spiflash, see* the definition of spiflash_data_line_e.

return value:

error code.

spiflash_data_line_e:

name definition		Remark
SPIFLASH_DATA_1_LINE spiflash single-line mode		
SPIFLASH_DATA_2_LINES spiflash two-line event		
SPIFLASH_DATA_4_LINES spiflash four-line event		

3.10.3.5 csi_spiflash_read

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

Function description:

Read data from spiflash.

parameter:

handle: The instance handle.

addr: The spiflash address to be read.

data: The buffer address of the data to be received.

cnt: The length of the data read.

return value:

error code.

3.10.3.6 csi_spiflash_program

```
int32_t csi_spiflash_program(spiflash_handle_t handle, uint32_t addr, const void *data, uint32_tÿ
,ÿcnt)
```

Function description:

Write data to spiflash.

parameter:

handle: The instance handle.

addr: The address of the written spiflash. data:

The buffer address of the data to be programmed.

cnt: The length of the data.

return value:

error code.

3.10.3.7 csi_spiflash_erase_sector

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

Function description:

Erase spiflash data by sector.

parameter:

handle: The instance handle.

addr: To erase the spiflash address.

return value:

error code.

3.10.3.8 csi_spiflash_erase_chip

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

Function description:

Erase the data of the entire spiflash.

parameter:

handle: The instance handle.

return value:

error code.

3.10.3.9 csi_spiflash_power_down

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

Function description:

Breakpoint spiflash.

parameter:

handle: The instance handle.

return value:

error code.

3.10.3.10 csi_spiflash_power_on

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

Function description:

Power on the spiflash.

parameter:

handle: The instance handle.

return value:

error code.

3.10.3.11 csi_spiflash_get_info

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

Function description:

Get the information of the spiflash at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of spiflash information, *see spiflash_info_t for the state definition of spiflash .*

spiflash_info_t:

name	define	Remark
uint32_t start	start address	
uint32_t end	end address	
uint32_t sector_count	number of sectors	
uint32_t sector_size	size of sector	
uint32_t page_size uint32_t	page size	
program_unit Minimum unit to write		
uint32_t erased_value erased value		

3.10.3.12 csi_spiflash_get_status

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

Function description:

Get the state of the spiflash at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of spiflash *status, see spiflash_status_t for the definition of spiflash status .*

spiflash_status_t:

Name Definition	Remark
busy : 1 status is busy	
error : 1 write/erase error	

3.10.4 Example

3.10.4.1 SPIFlash Example 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 erasedÿ ,ÿvalue is %x\r\n", info->sector_count, info-
>sector_size, info->program_unit, info->erased_value);

    //erase spiflash by chip
```

(continued on next page)

(continued from previous page)

```
    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 List of functions

3.11.2 Brief description

I2S (Inter-IC Sound) bus, also known as integrated circuit built-in audio bus, is a bus standard developed by Philips for audio data transmission between digital audio devices. This bus is specially used for data transmission between audio devices. , widely used in various multimedia systems. The I2S bus interface is usually three-wire: I2S_SCLK, I2S_WS, I2S_SDA, and there are also four-wire interfaces: I2S_SCLK, I2S_WS, I2S_SDA, I2S_MCLK (used as the clock source of the codec), and five-wire duplex interface: I2S_SCLK, I2S_WS, I2S_SDA, I2S_SDI, I2S_MCLK .

3.11.3 Interface description

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);
```

Function description:

Initialize the corresponding i2s instance by the device number, and return the handle of the i2s instance.

parameter:

idx: device number.

cb_event: The event callback function of the i2s instance (usually executed in the interrupt context). See i2s_event_cb_t for the definition of the callback function prototype. back

The call function type i2s_event_cb_t is defined as follows:

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

Where idx is the device number, and event is the event type passed to the callback function.

See the definition of i2s_event_e for the enumeration type of i2s callback event.

cb_arg: Callback function user parameter.

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

3.11.4 i2s_event_e:

name	definition	Remark
I2S_EVENT_SEND_COMPLETE Periodic data sending completion event (see i2s_config_t to define tx_period)		
I2S_EVENT_RECEIVE_COMPLETE Periodic data reception completion event (see i2s_config_t defines rx_period)		
I2S_EVENT_TX_UNDERFLOW send underflow event		
I2S_EVENT_TX_OVERFLOW receive overflow event transmit frame format error event		
I2S_EVENT_FRAME_ERROR		

3.11.4.1 csi_i2s_uninitialize

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

Function description:

The i2s instance is deinitialized. This interface will stop the ongoing transfer of the i2s instance (if any) and release related hardware and software resources.

parameter:

handle: The instance handle.

return value:

0: Normal

Other: Error code. .

3.11.4.2 csi_i2s_enable

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

Function description:

Control i2s module to enable, i2s starts to output clock signal after it is turned on.

parameter:

handle: The instance handle.

en: 1 enable, 0 disable

return value:

0: normal

Other: Error code. .

3.11.4.3 csi_i2s_get_capabilities

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

Function description:

Get the capabilities backed by an i2s instance.

parameter:

idx: device number.

return value:

A structure describing the capabilities of i2s, see i2s_capabilities_t for the definition of i2s capabilities.

3.11.4.4 i2s_capabilities_t:

Name Definition uint32_t protocol_user : 1 supports custom protocols		Remark
uint32_t protocol_i2s : 1 uint32_t protocol_justified : 1 supports left and right justified		
protocols	Support i2s protocol	
uint32_t mono_mode : 1	Supports single	
uint32_t mclk_pin : 1 uint32_t	channel support to provide master clock pin	
event_frame_error : 1 Support frame format error event reporting		

3.11.4.5 csi_i2s_config

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

Function description:

Configure i2s parameters.

parameter:

handle: The instance handle.

config: Configuration parameters, see i2s_config_t definition.

return value:

0: normal

Other: Error code. .

name	definition	prepare Note
cfg	i2s protocol configuration parameters, see i2s_config_type_t definition	
rate	Sampling Rate	
tx_period	i2s completes the transmission of tx_period bytes data, triggers the user callback function, and reports I2S_EVENT_SEND_COMPLETE incident.	
rx_period	i2s has finished receiving the tx_period bytes data, trigger user callback function, report I2S_EVENT_RECEIVE_COMPLETE event.	
tx_buf	send buffer	
tx_buf_length	send buffer length (bytes)	
rx_buf	receive buffer	
rx_buf_length	Receive buffer length (bytes)	

3.11.4.6 csi_i2s_send

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

Function description:

i2s sends data.

parameter:

handle: The instance handle.

data: send data

length: send data length

return value:

The length of the data actually written to the buffer, for example: length=1000, if the actual buffer is 100bytes free, only 100bytes of data can be written, and the return The return value is 100.

3.11.4.7 csi_i2s_receive

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

Function description: i2s receives data.

parameter:

handle: The instance handle.

buf: store received data

length: the length of the received data

return value:

The actual length of the data obtained, for example: length=1000, the actual buffer area has 100 bytes, then only 100 bytes of data can be read, and the return value is 100.

3.11.4.8 csi_i2s_send_ctrl

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

Function description:

i2s send control.

parameter:

handle: The instance handle.

cmd : control command, see i2s_ctrl_e definition

return value:

0: success

other: error code

3.11.4.9 csi_i2s_receive_ctrl

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

Function description:

i2s receives control.

parameter:

handle: The instance handle.

cmd : control command, see i2s_ctrl_e definition

return value:

0: success

Other: Error code

3.11.5 i2s_ctrl_e:

name definition		Remark
I2S_STREAM_PAUSE I2S transfer pause (buffer data hold)		
I2S_STREAM_RESUME I2S transfer unsuspended (continue from buffer breakpoint)		
I2S_STREAM_STOP	I2S stops transmission (buffer data is cleared)	
I2S_STREAM_START I2S start transmission		

3.11.5.1 csi_i2s_get_status

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

Function description:

Get the state of i2s at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of i2s status, see i2s_status_t <i2s_status_t * for the status definition of i2s.

3.11.6 i2s_status_t:

name	definition	Remark
int32_t tx_runing: 1	send busy	
uint32_t rx_runing: 1 receive busy		
uint32_t tx_fifo_empty: 1 send buffer empty		
uint32_t rx_fifo_full: 1 The receive buffer is full, receive overflow		
uint32_t frame_error: 1 frame data error		

3.11.6.1 csi_i2s_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: The power mode of the device instance, see the definition of csi_power_stat_e.

return value:

Normal:

0 Other: Error code

3.11.7 csi_power_stat_e:

name	definition	Remark
DRV_POWER_OFF power off state		
DRV_POWER_LOW low state		
DRV_POWER_FULL full power state		
DRV_POWER_SUSPEND Suspend power state		

3.12 AES

3.12.1 Function List

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

3.12.2 Brief description

AES (Advanced Encryption Standard) is a symmetric key encryption algorithm, that is, the encryption key and the decryption key are the same.

AES uses a symmetric block cipher system, the key length supports 128, 192, 256, and the block length is 128 bits.

3.12.3 Interface description

3.12.3.1 csi_aes_initialize

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

Function description:

Initialize the corresponding aes instance by passing in the number of devices, and return the handle of the aes instance.

parameter:

idx: device number.

cb_event: The event callback function of the aes instance. See aes_event_cb_t for the definition of the callback function prototype.

The callback function type aes_event_cb_t is defined as follows:

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

Where idx is the device number, event is the event type passed to the callback function, and the aes *callback event enumeration type is aes_event_e .*

return value:

NULL: Initialization failed.

Miscellaneous: The instance handle when initialization is successful.

aes_event_e:

name definition		Remark
AES_EVENT_CRYPTO_COMPLETE AES calculation complete event		

3.12.3.2 csi_aes_uninitialize

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

Function description:

aes instance deinitialized. This interface will stop the ongoing work of the aes instance (if any), and release related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.12.3.3 csi_aes_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.12.3.4 csi_aes_get_capabilities

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

Function description:

Get the capabilities backed by the aes instance.

parameter:

idx: device number.

return value:

A structure describing *the capabilities of aes, see aes_capabilities_t for the definition of aes capabilities .*

aes_capabilities_t:

Name Definition uint32_t ecb_mode :1 supports ecb		Remark
mode uint32_t cbc_mode :1 supports cbc mode uint32_t cfb1_mode		
:1 supports cfb1 mode uint32_t cfb8_mode :1 supports cfb8 mode		
uint32_t cfb128_mode :1 supports :cfb128 mode Mode uint32_t		
bits_128 :1 support 128bits mode uint32_t bits_192 :1 support		
192bits mode uint32_t bits_256 :1 support 256bits mode		

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)
```

Function description:

Configure the working mode, key length and endian mode of the aes instance.

parameter:

handle: The instance handle.

mode: aes *mode, see aes_mode_e* definition.

keylen_bits: *The length of the key, see aes_key_len_bits_e .* endian: *the big and small endian mode of aes, see the definition of aes_endian_mode_e .*

return value:

error code.

aes_mode_e:

name	definition	Remark
AES_MODE_ECB	ECB mode	
AES_MODE_CBC	CBC mode	
AES_MODE_CFB1 CFB1 mode		
AES_MODE_CFB8 CFB8 mode		
AES_MODE_CFB128 CFB128 mode		
AES_MODE_OFB	OFB mode	
AES_MODE_CTR CTR mode		

aes_key_len_bits_e:

name definition		Remark
AES_KEY_LEN_BITS_128 128bits length		
AES_KEY_LEN_BITS_192 192bits length		
AES_KEY_LEN_BITS_256 256bits length		

aes_endian_mode_e:

name definition		Remark
AES_ENDIAN_LITTLE AES little endian mode		
AES_ENDIAN_BIG	AES big endian mode	

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,ÿ
 ,ÿaes_crypto_mode_e enc)
```

Function description:

Set the key for aes.

parameter:

handle: The instance handle.

context: the buffer of the context of aes.

key: The buffer address of the key.

key_len: The length of the key to be entered.

enc: *Encryption and decryption operations, see the definition of aes_crypto_mode_e .*

return value:

error code.

aes_crypto_mode_e:

name definition		Remark
AES_CRYPTO_MODE_ENCRYPT AES encryption mode		
AES_CRYPTO_MODE_DECRYPT AES decryption mode		

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)
```

Function description:

Encrypt and decrypt in ECB mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes. in: The buffer

address of the data before the operation. out: The buffer

address of the data after the operation.

len: The length of the data to be input.

return value:

error code.

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,ÿ
,ÿuint8_t iv[16])
```

Function description:

Operation aes Through the previously set operation mode, perform encryption and decryption in aes cbc mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes. in: The buffer

address of the data before the operation. out: The buffer

address of the data after the operation. len: The length of

the data to be input. iv: Initial vector.

return value:

error code.

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, ,ÿ uint8_t iv[16])
```

Function description:

Operation aes Perform encryption and decryption in aes cfb1 mode through the previously set operation mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes.

in: The buffer address of the data before the operation.

out: The buffer address of the data after the operation.

len: The length of the data to be input. iv: Initial vector.

return value:

error code.

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,ÿ ,ÿuint8_t iv[16])
```

Function description:

Operation aes Perform the encryption and decryption of aes cfb8 mode through the previously set operation mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes.

in: The buffer address of the data before the operation.

out: The buffer address of the data after the operation.

len: The length of the data to be input. iv: Initial vector.

return value:

error code.

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_tÿ
,ÿlen, uint8_t iv[16], uint32_t *num)
```

Function description:

Operation aes Perform the encryption and decryption of aes cfb128 mode through the previously set operation mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes.

in: The buffer address of the data before the operation.

out: The buffer address of the data after the operation.

len: The length of the data to be input.

iv: Initial vector. num:

The offset in a block of the number of bytes that have been calculated.

return value:

error code.

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,ÿ ,ÿuint8_t iv[16], uint32_t *num)
```

Function description:

Operation aes Perform the encryption and decryption of aes ofb mode through the previously set operation mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes.

in: The buffer address of the data before the operation.

out: The buffer address of the data after the

operation. len: The length of the data to be input.

iv: Initial vector.

num: The offset in a block of the number of bytes that have been calculated.

return value:

error code.

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],ÿ
ÿ uint32_t *num)
```

Function description:

Operation aes Perform the encryption and decryption of aes ctr mode through the previously set operation mode.

parameter:

handle: The instance handle.

context: the buffer of the context of aes.

in: The buffer address of the data before the operation.

out: The buffer address of the data after the

operation. len: The length of the data to be input.

nonce_counter: The buffer address of the nonce counter.

stream_block: The encrypted buffer address of the random counter.

num: The offset in a block of the number of bytes that have been calculated.

return value:

error code.

3.12.3.14 csi_aes_get_status

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

Function description:

Get the state of aes at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of aes status, *see aes_status_t for the status definition of* aes .

aes_status_t:

Name Definition Remarks uint32_t busy :1		
Computing is busy		

3.12.4 Examples

3.12.4.1 AES Example 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 ÿkey bits len ÿendian 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;
    }
```

(continued on next page)

(continued from previous page)

```
    // 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 Function List

- *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 Brief description

CRC (Cyclic Redundancy Check) Cyclic Redundancy Check is a hash function that generates a short fixed-digit check code based on data.

To detect or verify errors that may occur after data transmission or saving. Different generator polynomial versions are available for the CRC algorithm.

3.13.3 Interface description

3.13.3.1 csi_crc_initialize

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

Function description:

Initialize the corresponding crc instance by passing in the number of devices, and return the handle of the crc instance.

parameter:

idx: device number.

cb_event: The event callback function of the crc instance. See crc_event_cb_t for the definition of the callback function prototype.

The callback function type crc_event_cb_t is defined as follows:

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

Where idx is the device number, event is the event type passed to the callback function, and the crc *callback event enumeration type is crc_event_e* .

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

crc_event_e:

name	definition	Remark
CRC_EVENT_CALCULATE_COMPLETE calculation complete event		

3.13.3.2 csi_crc_uninitialize

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

Function description:

The crc instance is deinitialized. This interface will stop the ongoing work of the crc instance (if any) and release related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.13.3.3 csi_crc_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.13.3.4 csi_crc_get_capabilities

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

Function description:

Get the capabilities backed by the crc instance.

parameter:

idx:

return value:

A structure describing crc capabilities, *see crc_capabilities_t for the definition* of crc capabilities .

crc_capabilities_t:

Name	definition	Remark
uint32_t ROHC :1 Support ROHC mode uint32_t MAXIM :1 Support MAXIM		
mode Support X25 mode uint32_t X25 :1 uint32_t CCITT :1 Support CCITT		
USB mode Support IBM mode	mode uint32_t USB :1 Support	
uint32_t IBM :1 uint32_t		
MODBUS :1 Support MODBUS mode		

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)
```

Function description:

Configure the working mode of the crc instance.

parameter:

handle: The instance handle.

mode: crc *mode, see crc_mode_e* definition.

standard: crc *standard, see the definition of crc_standard_crc_e .*

return value:

error code.

crc_mode_e:

name definition		Remark
CRC_MODE_CRC8 CRC8 mode		
CRC_MODE_CRC16 CRC16 mode		
CRC_MODE_CRC32 CRC32 mode		

crc_standard_crc_e:

name	definition	Remark
CRC_STANDARD_CRC_ROHC	CRC standard RHOC	
CRC_STANDARD_CRC_MAXIM CRC standard MAXIM		
CRC_STANDARD_CRC_X25	CRC standard X25	
CRC_STANDARD_CRC_CCITT CRC ÿÿ CCITT		
CRC_STANDARD_CRC_USB	CRC standard USB	
CRC_STANDARD_CRC_IBM	CRC standard IBM	
CRC_STANDARD_CRC_MODBUS CRC ÿÿ 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)
```

Function description:

Calculate crc, if the incoming data length is not word-aligned, the interface will be filled with 0 inside.

parameter:

handle: The instance handle.

in: The buffer address of the data to be passed in.

out: The buffer address of the data to be received.

len: The length of the data to be passed in.

return value:

error code.

3.13.3.7 csi_crc_get_status

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

Function description:

Get the status of the CRC at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of crc *status, see crc_status_t for the definition* of crc status .

crc_status_t:

Name Definition Remarks uint32_t busy :1 Computing		
is busy		

3.13.4 Example

3.13.4.1 CRC Example 1

```
static crc_handle_t crc= NULL;


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

(continued on next page)

(continued from previous page)

```
    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 Function List

- *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 Brief description

RSA public key cryptosystem. The so-called public key cryptosystem uses different encryption keys and decryption keys.
It is computationally infeasible to derive the decryption key" cryptosystem.

3.14.3 Interface description

3.14.3.1 csi_rsa_initialize

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

Function description:

Initialize the corresponding rsa instance by passing in the number of devices, and return the handle of the rsa instance.

parameter:

idx: device number.

cb_event: The event callback function of the rsa instance. See rsa_event_cb_t for the prototype definition of the callback function.

The callback function type rsa_event_cb_t is defined as follows:

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

The event is the event type passed to the callback function, and the rsa *callback event enumeration is defined in the type rsa_event_e* .

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

rsa_event_e:

name definition		Remark
RSA_EVENT_ENCRYPT_COMPLETE RSA encryption completion event		
RSA_EVENT_DECRYPT_COMPLETE RSA decryption complete event		
RSA_EVENT_SIGN_COMPLETE	RSA signature complete event	
RSA_EVENT_VERIFY_COMPLETE RSA verification completion event		

3.14.3.2 csi_rsa_uninitialize

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

Function description:

The rsa instance is deinitialized. This interface stops the ongoing work of the rsa instance (if any) and releases related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.14.3.3 csi_rsa_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.14.3.4 csi_rsa_get_capabilities

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

Function description:

Get the capabilities backed by the rsa instance.

parameter:

idx: device number.

return value:

A structure describing *the capabilities of rsa, see rsa_capabilities_t for the definition of rsa capabilities .*

rsa_capabilities_t:

name	definition	Remark
uint32_t bits_192 :1 support 192bits mode uint32_t bits_256 :1		
support 256bits mode uint32_t bits_512 :1 support 512bits mode		
uint32_t bits_1024 :1 support 1024bits mode		

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)
```

Function description:

Configure the bit length and endian mode of the rsa instance.

parameter:

handle: The instance handle.

data_bits: rsa *data length, see rsa_data_bits_e* definition. endian: *The big and small endian mode of rsa, see the definition of rsa_endian_mode_e .* arg: The address of the modulo value.

return value:

error code.

rsa_data_bits_e:

name definition		Remark
RSA_DATA_BITS_192 192bits length		
RSA_DATA_BITS_256 256bits length		
RSA_DATA_BITS_512 512bits length		
RSA_DATA_BITS_1024 1024bits length		
RSA_DATA_BITS_2048 2048bits length		
RSA_DATA_BITS_3072 3072bits length		

rsa_endian_mode_e:

name definition		Remark
RSA_ENDIAN_MODE_LITTLE RSA little endian mode		
RSA_ENDIAN_MODE_BIG	RSA big endian mode	

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)
```

Function description:

rsa encryption.

parameter:

handle: The instance handle.

n: The buffer address of the modulo value of

rsa. e: The buffer address of the rsa public key.

src: The buffer address of the plaintext.

src_size: The length of the plaintext.

out: The buffer address of the encrypted

result. out_size: Length buffer address of encrypted data.

padding: padding *mode, see rsa_padding_t .*

return value:

error code.

rsa_padding_t:

Name definition rsa_padding_type_e padding_type		Remark
Padding type		
rsa_hash_type_e hash_type	Padding hash type	

rsa_padding_type_e:

name	definition	Remark
RSA_PADDING_MODE_PKCS1	PKCS1 PADDING mode	
RSA_PADDING_MODE_NO	NO PADDING mode	
RSA_PADDING_MODE_SSLV23	SSLV23 PADDING mode	
RSA_PADDING_MODE_PKCS1_OAEP PKCS1_OAEP PADDING ÿÿ		
RSA_PADDING_MODE_X931	X931 PADDING mode	
RSA_PADDING_MODE_PSS	PSS PADDING mode	

rsa_hash_type_e:

name	definition	Remark
RSA_HASH_TYPE_MD5 MD5 type		
RSA_HASH_TYPE_SHA1 SHA1 type		
RSA_HASH_TYPE_SHA224 SHA224 type		
RSA_HASH_TYPE_SHA256 SHA256 type		
RSA_HASH_TYPE_SHA384 SHA384 type		
RSA_HASH_TYPE_SHA512 SHA512 type		

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)
```

Function description:

rsa decryption.

parameter:

handle: The instance handle.

n: The buffer address of the modulo value of rsa.

d: The buffer address of the rsa private key.

src: The buffer address of the ciphertext.

src_size: The length of the ciphertext.

out: The buffer address of the decryption result.

out_size: Length buffer address of decrypted data. padding:

padding *mode, see rsa_padding_t .*

return value:

error code.

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)
```

Function description:

rsa signature.

parameter:

handle: The instance handle. n:

The buffer address of the modulo value of rsa.

d: The buffer address of the rsa private key.

src: The buffer address of the input data.

src_size: The length of the input data. signature:

The buffer address of the signature result. sig_size: Length

buffer address of signed result.

padding: padding *mode, see rsa_padding_t .*

return value:

error code.

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)
```

Function description:

rsa signature verification.

parameter:

handle: The instance handle.

n: The buffer address of the modulo value of rsa.

e: The buffer address of the rsa public key.

src: The buffer address of the input data. src_size:

The length of the input data.

signature: The buffer address of the signature result. sig_size:

The length of the signed result. result: The result of the

signature verification. padding: padding *mode, see*

rsa_padding_t .

return value:

error code.

3.14.3.10 csi_rsa_get_status

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

Function description:

Get the state of rsa at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of rsa *status, see rsa_status_t for the definition* of rsa status .

rsa_status_t:

Name Definition Remarks uint32_t busy :1 Computing is		
busy		

3.15 SHA

3.15.1 List of functions

- *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 Brief description

SHA (Secure Hash Algorithm) secure hash algorithm is a hash algorithm that receives a piece of plaintext and then converts it into a piece of (usually smaller) ciphertext in an irreversible way. It can also be simply understood as taking a string of The process of taking input codes and converting them into a short-length, fixed-digit output sequence that is a hash value (also known as a message digest or message authentication code). Can be used to implement digital signatures.

3.15.3 Interface description

3.15.3.1 csi_sha_initialize

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

Function description:

Initialize the corresponding sha instance by passing in the number of devices, and return the handle of the sha instance.

parameter:

idx: set alias.

context: save the sha context. cb_event:

The event callback function of the sha instance. See sha_event_cb_t for the definition of the callback function prototype.

The callback function type sha_event_cb_t is defined as follows:

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

Where idx is the device number, event is the event type passed to the callback function, and the sha *callback event enumeration type is defined in sha_event_e .*

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

sha_event_e:

name definition		Remark
SHA_EVENT_COMPLETE Compute Completion Event		

3.15.3.2 csi_sha_uninitialize

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

Function description:

sha instance is deinitialized. This interface will stop the ongoing work of the sha instance (if any) and release related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.15.3.3 csi_sha_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.15.3.4 csi_sha_get_capabilities

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

Function description:

Get the capabilities backed by the sha instance.

parameter:

idx: device number.

return value:

A structure describing sha capabilities, *see sha_capabilities_t for the definition* of sha capabilities .

sha_capabilities_t:

name	Defines	Remark
uint32_t sha1 :1	support for sha1 mode	
uint32_t sha224 :1	Support sha224 mode	
uint32_t sha256 :1	Support sha256 mode	
uint32_t sha384 :1	Support sha384 mode	
uint32_t sha512 :1 uint32_t	Support sha512 mode	
sha512_224 :1 supports sha512_224 mode		
uint32_t sha512_256 :1 supports sha512_256 mode		
uint32_t endianmode :1 big and small endian mode		
uint32_t interruptmode :1 interrupt mode		

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)
```

Function description:

Configure the working mode and endian mode of the sha instance.

parameter:

handle: The instance handle.

mode: sha *mode, see sha_mode_e* definition.

endian: sha *big and small endian mode, see the definition of sha_endian_mode_e .*

return value:

error code.

sha_mode_e:

name	definition	Remark
SHA_MODE_1	SHA1 mode	
SHA_MODE_224	SHA224 mode	
SHA_MODE_256	SHA256 mode	
SHA_MODE_512	SHA512 mode	
SHA_MODE_384	SHA384 mode	
SHA_MODE_512_224 SHA512_224 mode		
SHA_MODE_512_256 SHA512_256 mode		

sha_endian_mode_e:

name	definition	Remark
SHA_ENDIAN_MODE_BIG	SHA big endian mode	
SHA_ENDIAN_MODE_LITTLE SHA little endian mode		

3.15.3.6 csi_sha_start

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

Function description:

Start sha calculation.

parameter:

handle: The instance handle.

context: The buffer address of the context of the sha.

return value:

error code.

3.15.3.7 csi_sha_update

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

Function description:

Update the calculation of sha.

parameter:

handle: The instance handle.

context: The buffer of the context of the sha.

input: The buffer address of the data to be input. len :

The length of the data to be entered.

return value:

error code.

3.15.3.8 csi_sha_finish

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

Function description:

Calculate the last hash of sha.

parameter:

handle: The instance handle.

context: The buffer address of the context of the sha. output: The

buffer address of the hash result to be received.

return value:

error code.

3.15.3.9 csi_sha_get_status

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

Function description:

Get the state of sha at the current moment.

parameter:

handle: The instance handle.

return value:

The structure of sha *status, see sha_status_t for the definition* of sha status .

sha_status_t:

Name Definition Remarks uint32_t busy :1		
Computing is busy		

3.16 WHITE

3.16.1 Function List

- *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 Brief description

TRNG (True Random Number Generator) A true random number generator is a device that generates random numbers through a physical process rather than a computer program.

3.16.3 Interface description

3.16.3.1 csi_trng_initialize

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

Function description:

Initialize the corresponding trng instance by passing in the number of devices, and return the handle of the trng instance.

parameter:

handle: The instance handle.

cb_event: The event callback function of the trng instance. See trng_event_cb_t for the prototype definition of the callback function.

The callback function type trng_event_cb_t is defined as follows:

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

Where idx is the device number, event is the event type passed to the callback function, and the trng *callback event enumeration type is* defined in trng_event_e.

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

trng_event_e:

name	definition	Remark
TRNG_EVENT_DATA_GENERATECOMPLETE random number generation complete event		

3.16.3.2 csi_trng_uninitialize

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

Function description:

The trng instance is deinitialized. This interface stops the ongoing work of the trng instance (if any), and releases related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.16.3.3 csi_trng_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

handle: The instance handle.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.16.3.4 csi_trng_get_capabilities

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

Function description:

Get the capabilities backed by the trng instance.

parameter:

idx: device number.

return value:

A structure describing *the capabilities of trng, see trng_capabilities_t for the definition of trng capabilities .*

trng_capabilities_t:

Name definition uint32_t lowper_mode : 1 supports low		Remark
power mode		

3.16.3.5 csi_trng_get_data

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

Function description:

Get the trng random number.

parameter:

handle: The instance handle.

data: The buffer address of the data to be generated.

num: The length of the data to be generated.

return value:

error code.

3.16.3.6 csi_trng_get_status

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

Function description:

parameter:

handle: The instance handle.

return value:

The structure of trng status, *see trng_status_t for the status definition of trng .*

trng_status_t:

name	Definition Remarks
uint32_t busy :1 uint32_t data_valid :1	0:busy 1:Data valid

3.16.4 Examples

3.16.4.1 TRNG Example 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);
```

(continued on next page)

(continued from previous page)

```
    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 Function List

- *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 Brief description

DMA (Direct Memory Access, Direct Memory Access) It allows communication between hardware devices of different speeds without relying on a large amount of interrupt load on the CPU.

3.17.3 Interface description

3.17.3.1 csi_dma_alloc_channel

```
int32_t csi_dma_alloc_channel(void)
```

Function description:

Request dma channel number.

parameter:

none

return value:

>=0: Channel number.

Other: Error code.

3.17.3.2 csi_dma_power_control

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

Function description:

Configure the power mode of the device instance.

parameter:

ch: channel number.

state: *The power mode of the device instance, see the definition of csi_power_stat_e .*

return value:

error code.

3.17.3.3 csi_dma_get_capabilities

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

Function description:

Get the capabilities supported by the dma channel instance.

parameter:

ch: channel number.

return value:

A structure describing dma capabilities, *see dma_capabilities_t for the definition* of dma capabilities .

dma_capabilities_t:

Name	definition	Remark
uint32_t unalign_addr : 1	Support unaligned address transfer mode (transfer source is memory)	

3.17.3.4 csi_dma_release_channel

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

Function description:

Release channel number ch.

parameter:

ch: channel number.

return value:

none.

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_ ,ÿarg)
```

Function description:

Configure the dma channel ch.

parameter:

ch: channel number.

config: The specific configuration item structure. Configuration structure prototype definition see dma_config_t . cb_event: Event

callback function of dma channel ch (usually executed in interrupt context). See dma_event_cb_t for the definition of the callback function prototype.

The callback function type dma_event_cb_t is defined as follows:

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

Where ch is the channel number, and event is the event type passed to the callback function.

See the definition of dma_event_e for the enumeration type of ch callback event in dma .

return value:

error code.

dma_event_e:

name	Define	Remark
DMA_EVENT_TRANSFER_DONE	the transfer complete event	
DMA_EVENT_TRANSFER_HALF_DONE transfer half complete event		
DMA_EVENT_TRANSFER_MODE_DONE transfer complete event (trigger mode)		
DMA_EVENT_CHANNEL_PEND	channel hang event	
DMA_EVENT_TRANSFER_ERROR	Transmission error event	

dma_config_t:

Name	definition	Remark
src_inc Source address change mode, see the definition		
dst_inc destination address change method, see the definition of dma_addr_inc_e		
src_endian source address endian mode, see the definition of dma_addr_endian_e		
dst_endian destination address endian mode, see the definition of dma_addr_endian_e		
src_tw source transfer data width		
dst_tw destination transfer data width		
hs_if	Hardware handshake (optional)	
preemption A channel preemption configuration		
type	Transmission type configuration, see the definition of dma_trans_type_e	
mode	Trigger mode configuration, see the definition of dma_trig_trans_mode_e	
ch_mode channel application use mode configuration, see dma_channel_req_mode_e definition		
single_dir single transfer direction enumeration type dma_single_dir_e definition		
group_len Group transmission length setting, when the trigger mode is configured as GROUP_TRIGGER mode		

dma_addr_inc_e:

name	Define	Remark
DMA_ADDR_INC	the address increment method	
DMA_ADDR_DEC address decrement method		
DMA_ADDR_CONSTANT address unchanged mode		

dma_trans_type_e:

name	definition	Remark
DMA_MEM2MEM memory-to-memory transfer type		
DMA_MEM2PERH memory to peripheral transfer type		
DMA_PERH2MEM peripheral to memory transfer type		
DMA_PERH2PERH Peripheral-to-peripheral transfer type		

dma_trig_trans_mode_e:

name definition		Remark
DMA_SINGLE_TRIGGER single trigger mode		
DMA_GROUP_TRIGGER group trigger mode		
DMA_BLOCK_TRIGGER block trigger mode		

dma_single_dir_e:

name definition remarks		
DMA_DIR_DEST destination direction		
DMA_DIR_SOURCE source direction		

dma_addr_endian_e:

name definition remarks		
DMA_ADDR_LITTLE little endian mode		
DMA_ADDR_BIG big endian mode		

dma_channel_req_mode_e:

name definition remarks		
DMA_MODE_HARDWARE hardware mode		
DMA_MODE_SOFTWARE software mode		

3.17.3.6 csi_dma_start

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

Function description:

dma transfer begins.

parameter:

ch: channel

number. psrcaddr: Source address of channel ch for dma

transfer. pdstaddr: The destination address of channel ch for dma

transmission. length: The data length of channel ch for dma transmission.

return value:

none.

3.17.3.7 csi_dma_stop

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

Function description:

dma transfer ended.

parameter:

ch: channel number.

return value:

none.

3.17.3.8 csi_dma_get_status

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

Function description:

Get the status of the ch channel number at the current moment.

parameter:

ch: channel number.

return value:

Enumeration of dma status, *see dma_status_e for dma status definition .*

dma_status_e:

name	Definition	Remark
DMA_EVENT_TRANSFER_DONE	transfer complete	
DMA_EVENT_TRANSFER_HALF_DONE transfer half done		
DMA_EVENT_TRANSFER_MODE_DONE transfer completed (a trigger mode)		
DMA_EVENT_CAHNNEL_PEND	preempted pending state	
DMA_EVENT_TRANSFER_ERROR	Transmission exception	

3.17.4 Examples

3.17.4.1 DMA Example 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;
```

(continued on next page)

(continued from previous page)

```
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;
```

(continued on next page)

(continued from previous page)

```
        }

        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 Function List

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

3.18.2 Brief description

PMU (power management unit) power management unit is a highly integrated power management solution for portable applications, which integrates several types of traditional discrete power management devices into a single package, which can achieve higher power conversion Efficiency and lower power consumption, and lower component count to accommodate shrinking board space.

3.18.3 Interface description

3.18.3.1 csi_pmu_initialize

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

Function description:

Initialize the corresponding pmu instance through the device number, and return the handle of the pmu instance.

parameter:

idx: device number.

cb_event: The event callback function of the pmu instance (usually executed in the context of entering a low power mode). The application can do the corresponding behaviors before and after going to bed according to the callback function. See pmu_event_cb_t for the definition of the callback function prototype.

The callback function type pmu_event_cb_t is defined as follows:

```
typedef void (*pmu_event_cb_t)(int32_t idx, pmu_event_e eventÿ pmu_mode_eÿ ,ÿmode);
```

Where idx is the device number, event is the event type passed to the callback function, and mode is the low-power mode passed to the callback function.

See pmu_event_e definition for pmu *callback event enumeration type* .

return value:

NULL: Initialization failed.

Miscellaneous: Instance handle.

pmu_event_e:

name definition		Remark
PMU_EVENT_PREPARE_SLEEP bedtime preparation event		
PMU_EVENT_PREPARE_DONE Bedtime wakeup event		

3.18.3.2 csi_pmu_uninitialize

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

Function description:

The pmu instance is deinitialized. This interface will release related software and hardware resources.

parameter:

handle: The instance handle.

return value:

error code.

3.18.3.3 csi_pmu_enter_sleep

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

Function description:

The pmu instance is deinitialized. This interface will release related software and hardware resources.

parameter:

handle: The instance handle.

mode: pmu *mode definition see pmu_mode_e .*

return value:

error code.

pmu_mode_e:

name	Define	Remark
PMU_MODE_RUN	the operating mode	
PMU_MODE_SLEEP	sleep mode (turn off cpu clock)	
PMU_MODE_DOZE	Stop mode (turn off cpu and peripheral clock)	
PMU_MODE_DORMANT sleep mode (turn off the cpu and most peripherals)		
PMU_MODE_STANDBY Standby mode (turn off cpu, most peripherals and ram power)		
PMU_MODE_SHUTDOWN shutdown mode		

3.18.3.4 csi_pmu_power_control

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

Function description:

Save and restore pmu registers.

parameter:

handle: The instance handle.

state: *Power state. See csi_power_stat_e for the prototype definition .*

return value:

error code.

csi_power_stat_e:

name definition		Remark
DRV_POWER_OFF power off state		
DRV_POWER_LOW low state		
DRV_POWER_FULL full power state		
DRV_POWER_SUSPEND Suspend power state		

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_eÿ ,ÿtype, pmu_wakeup_pol_e pol, uint8_t enable)
```

Function description:

Configure the wakeup source for pmu.

parameter:

handle: The instance handle.

wakeup_num: Wakeup number.

type: The signal type of the wakeup source, see pmu_wakeup_type_e for the definition. pol:

Signal polarity of wakeup source, see pmu_wakeup_pol_e for definition. enable: Whether to

enable the wake-up source.

return value:

error code.

pmu_wakeup_type_e:

name definition remarks		
PMU_WAKEUP_TYPE_PULSE pulse type		
PMU_WAKEUP_TYPE_LEVEL level type		

pmu_wakeup_pol_e:

name	definition	Remark
PMU_WAKEUP_POL_LOW low level/falling edge valid		
PMU_WAKEUP_POL_HIGH high level/rising edge valid		

3.18.4 Example

3.18.4.1 PMU Example 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;
```

(continued on next page)

(continued from previous page)

```
    }

    ret = csi_pmu_config_wakeup_source(pmu_handle, EXAMPLE_WAKEUP_NUM, PMU_WAKEUP_TYPE_LEVEL, PMU_
,ÿWAKEUP_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_
,ÿWAKEUP_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)
{
```

(continued on next page)

(continued from previous page)

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

3.19 Mailbox

3.19.1 List of functions

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

3.19.2 Brief description

Mailbox provides a way to communicate between cores.

3.19.3 Interface description

3.19.3.1 csi_mailbox_initialize

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

Function description:

Initialize the corresponding Mailbox instance by index number, and return the handle of the Mailbox instance.

parameter:

cb_event: interrupt callback function.

return value:

Returns the instance handle on success, NULL on failure.

mailbox_event_cb_t:

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

where handle is the handle corresponding to the initialization, mailbox_id is the mailbox number corresponding to this event, and received_len is the actual byte received Number, event is the event type passed to the callback function. The mailbox callback event enumeration type mailbox_event_e is defined as follows:

MAILBOX_EVENT_SEND_COMPLETE data sending completion event	
MAILBOX_EVENT_RECEIVED	The data is received and stored in the MAILBOX FIFO, and the receive function can be called to read

3.19.3.2 csi_mailbox_uninitialize

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

Function description:

The mailbox instance is deinitialized, this interface will stop the ongoing work of the mailbox instance (if any), and release related hardware and software resources.

parameter:

handle: The instance handle.

return value:

error code.

3.19.3.3 csi_mailbox_send

```
int32_t csi_mailbox_send(mailbox_handle_t handle, int32_t mailbox_id, const void *data, uint32_tÿ
,ÿnum)
```

Function description:

Send a message to the target mailbox.

parameter:

handle: The instance handle.

mailbox_id: The target mailbox number. data: The

address where the sent data is stored.

num: The number of bytes sent.

return value:

Number of bytes sent successfully or error code.

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)
```

Function description:

Receive data from the target mailbox.

parameter:

handle: The instance handle.

mailbox_id: The target mailbox number. data:

The address where the received data is stored.

num: The number of bytes received, the value passed in is the actual received value returned by the callback.

return value:

Number of bytes successfully received or error code.

3.20 Codec

3.20.1 Function List

- *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 Brief description

codec is an audio codec, which has audio capture (convert analog to digital) and audio playback (digital audio to analog), with the same Data filtering, gain, etc. can be performed.

3.20.3 Interface description

3.20.3.1 csi_codec_init

```
int32_t csi_codec_init(uint32_t idx);
```

Function description:

Initialize the corresponding codec instance through the device number to obtain codec resources.

parameter:

idx: device number.

return value:

0: Initialization succeeded.

Other: Error code.

3.20.3.2 csi_codec_uninit

```
void csi_codec_uninit(uint32_t idx);
```

Function description:

Deinitialize the codec instance and release the codec resources.

parameter:

idx: device number.

return value:

none.

3.20.3.3 csi_codec_power_control

```
int32_t csi_codec_power_control(int32_t idx, csi_power_stat_e state);
```

Function description:

codec power control

parameter:

idx: device number.

state: Power mode, see csi_power_stat_e definition.

return value:

0: Success.

Other: Error code

3.20.4 csi_power_stat_e:

name	Define	Remark
DRV_POWER_OFF	the power-off state	
DRV_POWER_LOW low state		
DRV_POWER_FULL full power state		
DRV_POWER_SUSPEND Suspend power state		

3.20.4.1 csi_codec_input_open

```
int32_t csi_codec_input_open(codec_input_t *handle);
```

Function description:

Turn on the codec input path (recording path).

parameter:

handle: ocdec

The input handle is used to configure and control the input channel. When the channel is opened, allocate resources for the channel. See codec_input_t definition.

return value:

0: Success.

Other: Error code

3.20.4.2 csi_codec_input_close

```
int32_t csi_codec_input_close(codec_input_t *handle);
```

Function description:

Turn off the codec input path (recording path).

parameter:

handle: ocdec

The input handle, used to configure and control the input channel, see the definition of codec_input_t.

return value:

0: Success.

Other: Error code

3.20.5 codec_input_t:

name definition codec_idx		Remark
codec device number		
ch_idx input channel number		
cb	event callback function	
cb_arg event callback function user parameter		
buf	input data buffer	
buf_size	input data buffer size	
period The callback function is triggered once every period bytes data is collected.		
priv	Reserved (for interface bridging)	

3.20.6 codec_event_cb_t:

```
typedef void (*codec_event_cb_t)(codec_event_t event, void *arg);
```

Function description:

codec event callback function, user registration.

parameter:

event: codec event, see codec_event_t definition.

arg: User registered callback parameters, when the callback function is triggered, the user-defined parameters will be passed in.

return value:

none.

3.20.7 codec_event_t:

Name Definition | Remarks	
CODEC_EVENT_PERIOD_READ_COMPLETE codec input channel, after obtaining period bytes data|	
CODEC_EVENT_PERIOD_WRITE_COMPLETE codec output channel, after sending period bytes data|	
CODEC_EVENT_WRITE_BUFFER_EMPTY codec output channel, the data buffer is empty (the buffer is empty, the codec will output 0 data)	
CODEC_EVENT_READ_BUFFER_FULLcodec input channel, the data buffer is full (the buffer is full, the recording data will not be able to be written into the buffer storage, data loss)|	
CODEC_EVENT_TRANSFER_ERROR The codec is running abnormally|	

3.20.7.1 csi_codec_input_config

```
int32_t csi_codec_input_config(codec_input_t *handle, codec_input_config_t *config);
```

Function description:

The codec input channel initialization needs to be called before the codec input channel is started, and cannot be configured during the running process.

Parameters: handle: ocdec input handle, see codec_input_t definition.

config: codec input channel configuration parameters, see codec_input_config_t definition.

return value:

0: Success.

Other: Error code

3.20.8 codec_input_config_t:

The name defines the		
sample_rate sampling rate		Remarks see codec_sample_rate_t definition
channel_num The number of channels in the channel. For example: to record two-channel (left and right channel data cross-storage) audio, channel_num is set		
to 2 bit_width sampling precision		

3.20.9 codec_sample_rate_e:

name	definition	Remark
CODEC_SAMPLE_RATE_8000 Sample rate: 8000		
CODEC_SAMPLE_RATE_11025 Sample rate: 11025		
CODEC_SAMPLE_RATE_12000 Sample rate: 12000		
CODEC_SAMPLE_RATE_16000 Sample rate: 16000		
CODEC_SAMPLE_RATE_22050 Sample rate: 22050		
CODEC_SAMPLE_RATE_24000 Sample rate: 24000		
CODEC_SAMPLE_RATE_32000 Sample rate: 32000		
CODEC_SAMPLE_RATE_44100 Sample rate: 44100		
CODEC_SAMPLE_RATE_48000 Sample rate: 48000		
CODEC_SAMPLE_RATE_96000 Sampling rate: 96000		
CODEC_SAMPLE_RATE_192000 Sample rate: 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);
```

Function description:

Read codec input path data

parameter:

handle: ocdec input handle, see codec_input_t definition.

buf: The memory area for reading data.

length: The length of the read data, in bytes.

return value:

The length of the actual read data (unit: bytes), for example: length-1000, if the data in the buffer is 100, the actual read data is 100, and the return value is 100.

3.20.9.2 csi_codec_input_buf_avail

```
uint32_t csi_codec_input_buf_avail(codec_input_t *handle);
```

Function description:

Get the size of the free area of the codec input path buffer.

parameter:

handle: ocdec input handle, see codec_input_t definition.

return value:

codec input Channel buffer free space, in bytes.

3.20.9.3 csi_codec_input_buf_reset

```
int32_t csi_codec_input_buf_reset(codec_input_t *handle);
```

Function description:

Reset the codec input channel buffer area, after reset, the data in the buffer area are all cleared.

parameter:

handle: ocdec input handle, see codec_input_t definition.

return value:

0: Success.

Other: Error code.

3.20.9.4 csi_codec_input_start

```
int32_t csi_codec_input_start(codec_input_t *handle);
```

Function description:

Start the codec input channel, after startup, the codec input starts to collect data.

parameter:

handle: ocdec input handle, see codec_input_t definition.

return value:

0: Success.

Other: Error code.

3.20.9.5 csi_codec_input_stop

```
int32_t csi_codec_input_stop(codec_input_t *handle);
```

Function description:

Stop the data collection of the codec input channel. After calling stop, all the data in the current buffer area will be cleared. If you want to stop the acquisition without clearing the data, you can call the csi_codec_input_pause function, see the definition of csi_codec_input_pause.

parameter:

handle: ocdec input handle, see codec_input_t definition.

return value:

0: Success.

Other: Error code.

3.20.9.6 csi_codec_input_pause

```
int32_t csi_codec_input_pause(codec_input_t *handle);
```

Function description:

Pause the codec input channel. After the codec

input channel is suspended, the data collection stops, and the data in the buffer is not lost. After calling csi_codec_input_resume to resume the collection, the data continues.

parameter:

handle: ocdec input handle, see codec_input_t definition.

Return value: 0: Success.

Other: Error code.

3.20.9.7 csi_codec_input_resume

```
int32_t csi_codec_input_resume(codec_input_t *handle);
```

Function description:

Resume the data collection of the codec input channel from the paused state.

parameter:

handle: ocdec input handle, see codec_input_t definition.

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Set the digital gain of the codec input channel.

parameter:

handle: ocdec input handle, see codec_input_t definition.

val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Set the analog gain of the codec input channel.

parameter:

handle: ocdec input handle, see codec_input_t definition. val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the digital gain value of the current codec input channel.

parameter:

handle: ocdec input handle, see codec_input_t definition. val: store the gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the current analog gain of the codec input channel.

parameter:

handle: ocdec input handle, see codec_input_t definition. val: store the gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Sets the codec input channel mixer gain.

parameter:

handle: ocdec input handle, see codec_input_t definition.

val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the current mixer gain of the codec input channel.

parameter:

handle: ocdec input handle, see codec_input_t definition. val: store the gain value

return value:

0: Success.

Other: Error code.

3.20.9.14 csi_codec_input_mute

```
int32_t csi_codec_input_mute(codec_input_t *handle, int en);
```

Function description:

Control the mute of the codec input channel. When mute is enabled, the data collected by the input channel has no sound.

parameter:

handle: ocdec input handle, see codec_input_t definition.

en: 1: enable, 0 disable.

return value:

0: Success.

Other: Error code.

3.20.9.15 csi_codec_output_open

```
int32_t csi_codec_output_open(codec_output_t *handle);
```

Function description:

Open the codec input and output channel (playback channel).

parameter:

handle:

The codec_output_t handle is used to configure and control the output channel. When the channel is opened, allocate resources for the channel. See codec_output_t definition.

return value:

0: Success.

Other: Error code

3.20.9.16 csi_codec_output_close

```
int32_t csi_codec_output_close(codec_output_t *handle);
```

Function description:

Close the codec input and output channel (playback channel).

parameter:

handle: codec_output_t handle.

return value:

0: Success.

Other: Error code

3.20.10 codec_output_t:

name definition codec_idx		Remark
codec device number		
ch_idx input channel number		
cb	event callback function	
cb_arg event callback function user parameter		
buf	input data buffer	
buf_size	input data buffer size	
period The callback function is triggered once every period bytes data is collected.		
priv	Reserved (for interface bridging)	

3.20.10.1 csi_codec_output_config

```
int32_t csi_codec_output_config(codec_output_t *handle, codec_output_config_t *config);
```

Function description:

codec output channel initialization, need to be in codec

The output channel is called before starting, and cannot be configured during running.

parameter:

handle: codec_output_t handle, see codec_input_t definition.

config: codec output channel configuration parameters, see codec_output_config_t definition.

return value:

0: Success.

Other: Error code

3.20.11 codec_output_config_t:

name definition sample_rate	Remark
sampling rate bit_width sampling	See codec_sample_rate_t definition
precision	
mono_mode_en enable mono mode mono_mode_en	-1: enable mono mode, 0 disable mono mode

3.20.12 codec_sample_rate_e:

name	definition	Remark
CODEC_SAMPLE_RATE_8000 Sample rate: 8000		
CODEC_SAMPLE_RATE_11025 Sample rate: 11025		
CODEC_SAMPLE_RATE_12000 Sample rate: 12000		
CODEC_SAMPLE_RATE_16000 Sample rate: 16000		
CODEC_SAMPLE_RATE_22050 Sample rate: 22050		
CODEC_SAMPLE_RATE_24000 Sample rate: 24000		
CODEC_SAMPLE_RATE_32000 Sample rate: 32000		
CODEC_SAMPLE_RATE_44100 Sample rate: 44100		
CODEC_SAMPLE_RATE_48000 Sample rate: 48000		
CODEC_SAMPLE_RATE_96000 Sampling rate: 96000		
CODEC_SAMPLE_RATE_192000 Sample rate: 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);
```

Function description:

Write data to the codec output path

parameter:

handle: ocdec output handle, see codec_input_t definition.

buf: write data.

length: The length of the written data, in bytes.

return value:

The actual written data length (in bytes), for example: length-1000bytes, if the buffer area is 100bytes free, then the actual written data is 100bytes,

The return value is 100.

3.20.12.2 csi_codec_output_buf_avail

```
uint32_t csi_codec_output_buf_avail(codec_output_t *handle);
```

Function description:

Get the size of the free area of the codec output path buffer.

parameter:

handle: ocdec output handle, see codec_output_t definition.

return value:

codec output Channel buffer free space, in bytes.

3.20.12.3 csi_codec_output_buf_reset

```
int32_t csi_codec_output_buf_reset(codec_output_t *handle);
```

Function description:

Reset the codec input channel buffer area, after reset, the data in the buffer area are all cleared.

parameter:

handle: ocdec output handle, see codec_output_t definition.

return value:

0: Success.

Other: Error code.

3.20.12.4 csi_codec_output_start

```
int32_t csi_codec_output_start(codec_output_t *handle);
```

Function description:

Start the codec output channel. After startup, the codec output starts to collect data.

parameter:

handle: ocdec output handle, see codec_output_t definition.

return value:

0: Success.

Other: Error code.

3.20.12.5 csi_codec_output_stop

```
int32_t csi_codec_output_stop(codec_output_t *handle);
```

Function description:

stop codec

output Channel data output, after calling stop, all the data in the current buffer will be cleared. To stop the output without clearing the data, call the csi_codec_output_pause function, see the definition of csi_codec_output_pause.

parameter:

handle: ocdec output handle, see codec_output_t definition.

return value:

0: Success.

Other: Error code.

3.20.12.6 csi_codec_output_pause

```
int32_t csi_codec_output_pause(codec_output_t *handle);
```

Function description:

Pause the codec output path. After the codec output path is paused, stop playing the data. The data in the buffer is not lost. Call csi_codec_output_resume to resume playback and continue the data.

parameter:

handle: ocdec output handle, see codec_output_t definition.

return value:

0: Success.

Other: Error code.

3.20.12.7 csi_codec_output_resume

```
int32_t csi_codec_output_resume(codec_output_t *handle);
```

Function description:

Resumes the data output of the codec output path from the paused state.

parameter:

handle: ocdec output handle, see codec_output_t definition.

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Set the left channel digital gain of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Sets the digital gain of the right channel of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Set the analog gain of the left channel of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Set the analog gain of the right channel of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the digital gain value of the left channel of the current codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: store the gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the digital gain value of the right channel of the current codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition.

val: store the gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the current left channel analog gain of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: store the gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Get the current analog gain of the right channel of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: store the gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Sets the left channel gain of the codec output channel mixer.

parameter:

handle: ocdec output handle, see codec_output_t definition.

val: gain value

return value:

0: Success.

Other: Error code.

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);
```

Function description:

Sets the right channel gain of the codec output channel mixer.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: gain value

Return value: 0: Success.

Other: Error code.

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);
```

Function description:

Get the current mixer left channel gain of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: store the gain value

Return value: 0: Success.

Other: Error code.

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);
```

Function description:

Get the current mixer right channel gain of the codec output channel.

parameter:

handle: ocdec output handle, see codec_output_t definition. val: store the gain value

Return value: 0: Success.

Other: Error code.

3.20.12.20 csi_codec_output_mute

```
int32_t csi_codec_output_mute(codec_output_t *handle, int en);
```

Function description:

Control the mute of the codec output channel. When mute is enabled, the output channel will play no sound output.

parameter:

handle: ocdec output handle, see codec_output_t definition. en: 1:

enable, 0 disable.

return value:

0: Success.

Other: Error code.

3.20.13 Example:

```
/*******************************************************************************
 * @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);
```

(continued on next page)

(continued from previous page)

```
    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;
```

(continued on next page)

(continued from previous page)

```
    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");
```

(continued on next page)

(continued from previous page)

```
            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;
```

(continued on next page)

(continued from previous page)

```
    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) {
```

(continued on next page)

(continued from previous page)

```
            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;
```

(continued on next page)

(continued from previous page)

```
    }

    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
```

(continued on next page)

(continued from previous page)

```
#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);
```

(continued on next page)

(continued from previous page)

```
    csi_kernel_start ();

    return 0;
}
```

Chapter 4 CSI-Kernel API

4.1 Kernel Management

4.1.1 Function List

- *csi_kernel_init*
- *csi_kernel_start*
- *csi_kernel_get_stat*

4.1.2 Brief description

The system interface is used for the initialization, startup of the RTOS, and obtaining the running status of the RTOS.

The RTOS must be initialized with csi_kernel_init before using any RTOS interface, after calling csi_kernel_start

The system will enter the dispatch and will not return.

4.1.3 Interface description

4.1.3.1 csi_kernel_init

```
k_status_t csi_kernel_init(void)
```

Function description:

Kernel initialization interface.

parameter:

none.

return value:

error code.

csi_kernel_start

```
k_status_t csi_kernel_start(void)
```

Function description:

Kernel startup interface.

parameter:

none.

return value:

error code.

---

csi_kernel_get_stat

```
k_sched_stat_t csi_kernel_get_stat(void)
```

Function description:

Kernel running status

parameter:

none.

return value:

Kernel running status code.

4.2 Scheduler Management

4.2.1 Function List

4.2.2 Brief description

The scheduling control interface controls the scheduler of the entire system. Specific features include:

ff 1. Lock and unlock system scheduling. Can be used to implement critical section functions. These interfaces will affect the system real-time performance of the entire system and should be used with caution.

ff 2. Suspend and resume system scheduling. Can be used to implement tick-less function, generally used in low power mode.

4.2.3 Interface description

4.2.3.1 csi_kernel_sched_lock

```
int32_t csi_kernel_sched_lock(void)
```

Function description:

Lock the task scheduling of the Kernel (ie, disable task scheduling). Return to the state before locking.

parameter:

none.

return value:

Return to the state before locking.

---

csi_kernel_sched_unlock

```
int32_t csi_kernel_sched_unlock(void)
```

Function description:

Unlock task scheduling.

parameter:

none.

return value:

Returns the task lock state before calling this interface.

---

csi_kernel_sched_restore_lock

```
int32_t csi_kernel_sched_restore_lock(int32_t lock)
```

Function description:

parameter:

lock: 0: unlocked state; 1: locked state.

return value:

0 After executing this function, the scheduling state of the system is unlocked.

1: After executing this function, the scheduling state of the system is locked.

Others: Error code on failure.

---

csi_kernel_sched_suspend

```
uint32_t csi_kernel_sched_suspend(void)
```

Function description:

Pause the task scheduling of the system. Low power mode for using tick-less.

parameter:

none.

return value:

The amount of time the system can go to sleep. Unit: tick.

---

csi_kernel_sched_resume

```
void csi_kernel_sched_resume(uint32_t sleep_ticks)
```

Function description:

Restore the task scheduling of the system.

parameter:

sleep_ticks: The time the system has been sleeping. Unit: tick.

return value:

none.

4.3 Task

4.3.1 Function List

- *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 Brief description

A task is the basic unit of a real-time operating system for resource allocation and scheduling. The tasks of real-time operating systems generally use priority preemption and time slice rotation. A degree mechanism, that is, high-priority tasks can preempt low-priority tasks, and tasks of the same level are scheduled by time slice rotation.

Each task has a priority. Different types of real-time operating systems may have different definitions for the priority. For example, the highest priority for rhino tasks is 0, and the lowest priority for freeRTOS tasks is 0. CSI-Kernel uniformly standardizes the definition of priority to maintain the consistency of the interface to the user interface, see the definition of k_priority_t.

Each task has an independent stack space, and the information saved in the stack space includes local variables, registers, function parameters, function return addresses, etc. When the task is switched, the context information of the cut-out task will be saved in its own task stack space, so that the scene can be restored when the task resumes, so that the execution will continue at the cut-out point after the task resumes.

4.3.3 Interface description

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);
```

Function description:

Create a task and add it to the active task queue.

parameter:

task: task entry function.

name: task name.

arg: The parameter of the task entry function.

prio: *task priority. See k_priority_t for definition .*

time_quanta: The scheduling period (in ticks) in round-robin mode. When the value is 0, the FIFO scheduling method is used.

stack: The base address of the stack, the stack space can be defined by the user; if NULL is passed in, the stack space is requested by the system.

stack_size: The stack size of the task.

task_handle: The handle of the saved task when the function executes successfully.

return value:

error code.

priority type	definition	Remark
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	

Continue on next page

Table 4.1 - Continued from previous page

priority type	Define	Remark
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)
```

Function description:

Delete a task.

parameter:

task_handle: The handle of the task to delete.

return value:

error code.

---

csi_kernel_task_get_cur

```
k_task_handle_t csi_kernel_task_get_cur(void)
```

Function description:

Get the currently running task handle.

parameter:

none.

return value:

The currently running task handle.

---

csi_kernel_task_get_stat

```
k_task_stat_t csi_kernel_task_get_stat(k_task_handle_t task_handle)
```

Function description:

Get the running status of the task.

parameter:

task_handle: The handle of the task that needs to be operated on.

return value:

task status.

k_task_stat_t:

state name	definition	Remark
KTASK_ST_INACTIVE task status is Inactive		
KTASK_ST_READY	Task status is Ready	
KTASK_ST_RUNNING task status is Running		
KTASK_ST_BLOCKED task status is Blocked		
KTASK_ST_TERMINATED task status is Terminated		
KTASK_ST_ERROR	error	

---

csi_kernel_task_set_prio

```
k_status_t csi_kernel_task_set_prio(k_task_handle_t task_handle, k_priority_t priority)
```

Function description:

Set the priority of the task.

parameter:

task_handle: the handle of the task that needs to be operated

priority: The priority that the task needs to set. See k_priority_t for priority definition .

return value:

error code.

---

csi_kernel_task_get_prio

```
k_priority_t csi_kernel_task_get_prio(k_task_handle_t task_handle)
```

Function description:

Get the priority of the task.

parameter:

task_handle: The handle of the task that needs to be operated on.

return value:

>=0: *task priority. See k_priority_t for priority definition .* <0: Error code.

---

csi_kernel_task_get_name

```
const char *csi_kernel_task_get_name(k_task_handle_t task_handle)
```

Function description:

Get the task name.

parameter:

task_handle: The handle of the task that needs to be operated on.

return value:

task name.

---

csi_kernel_task_suspend

```
k_status_t csi_kernel_task_suspend(k_task_handle_t task_handle)
```

Function description:

Pause the scheduling of a task.

parameter:

task_handle: The handle of the task to suspend.

return value:

Error number.

---

csi_kernel_task_resume

```
k_status_t csi_kernel_task_resume(k_task_handle_t task_handle)
```

Function description:

Resume the scheduling of a task.

parameter:

task_handle: The handle of the scheduled task to resume.

return value:

error code.

---

csi_kernel_task_terminate

```
k_status_t csi_kernel_task_terminate(k_task_handle_t task_handle)
```

Function description:

Terminate a task, the terminated task will no longer be scheduled.

parameter:

task_handle: The handle of the task to terminate.

return value:

error code.

---

csi_kernel_task_exit

```
void csi_kernel_task_exit(void)
```

Function description:

Exit the current task.

parameter:

none.

return value:

none.

---

csi_kernel_task_yield

```
k_status_t csi_kernel_task_yield(void)
```

Function description:

The current task yields this schedule.

parameter:

none.

return value:

error code.

---

csi_kernel_task_get_count

```
uint32_t csi_kernel_task_get_count(void)
```

Function description:

Get the number of current tasks in the system.

parameter:

none.

return value:

The number of current tasks in the system.

---

csi_kernel_task_get_stack_size

```
uint32_t csi_kernel_task_get_stack_size(k_task_handle_t task_handle)
```

Function description:

Get the task stack size.

parameter:

task_handle: The handle of the task.

return value:

Task stack size.

---

csi_kernel_task_get_stack_space

```
uint32_t csi_kernel_task_get_stack_space(k_task_handle_t task_handle)
```

Function description:

Get the remaining stack size of the task.

parameter:

task_handle: The handle of the task.

return value:

The remaining stack size of the task.

---

csi_kernel_task_list

```
uint32_t csi_kernel_task_list(k_task_handle_t *task_array, uint32_t array_items)
```

Function description:

Get the handles of all the current tasks of the system.

parameter:

task_array: The address of the array that holds the task handle.

attay_items: The number of array elements to save the task handle.

return value:

The number of tasks actually obtained in the array.

Remark:

1. You can first get the number of tasks in the system through csi_kernel_task_get_count to determine the size of the array.
2. When array_items is greater than the return value, the total number of valid handles of the array is determined according to the return value.

4.4 Semphore

- *csi_kernel_sem_new*
- *csi_kernel_sem_del*
- *csi_kernel_sem_wait*
- *csi_kernel_sem_post*
- *csi_kernel_sem_get_count*

Semaphore (Semaphore) is a mechanism for realizing communication between tasks, realizing synchronization between tasks or mutually exclusive access to critical resources, and is often used to assist a group of competing tasks to access critical resources. Unlike mutexes, semaphores allow multiple tasks to access system resources simultaneously. The semaphore records the number of resources currently available through a resource count value. When the count value is 0, the resource is not allowed to be accessed until another task releases the resource.

When the semaphore is used for the mutual exclusion function, the resource count value of the semaphore is set to the maximum value of the resource when the semaphore is created. When the task needs to use the critical resource, the semaphore is obtained first. When the resource is used up, if there are still tasks to apply for use Critical resources will be blocked because the semaphore cannot be obtained, thus ensuring the safety of critical resources.

When the semaphore is used as a synchronization function, the resource count value of the semaphore is set to 0 when the semaphore is created, task 1 takes the semaphore and blocks, and task 2 occurs under certain conditions After that, the semaphore is released, so task 1 is executed, thus achieving synchronization between the two tasks.

```
k_sem_handle_t csi_kernel_sem_new(int32_t max_count, int32_t initial_count)
```

Function description:

Create a semaphore.

parameter:

max_count: The maximum number of semaphore counters. initial_count:

The initial value of the counter of the semaphore.

return value:

NULL: Creation failed.

Miscellaneous: Semaphore handle.

```
k_status_t csi_kernel_sem_del (k_sem_handle_t sem_handle)
```

Function description:

Delete a semaphore.

parameter:

sem_handle: The handle of the semaphore.

return value:

error code.

```
k_status_t csi_kernel_sem_wait(k_sem_handle_t sem_handle, int32_t timeout)
```

Function description:

Waiting for a semaphore.

parameter:

sem_handle: The handle of the semaphore.

timeout: Timeout time (unit: tick). If the semaphore cannot be obtained within this time, the function returns.

0 - do not wait;

negative value - always

wait; other positive values - maximum wait time.

return value:

error code.

---

k_status_t csi_kernel_sem_post (k_sem_handle_t sem_handle)

Function description:

Send a semaphore.

parameter:

sem_handle: The handle of the semaphore.

return value:

error code.

---

int32_t csi_kernel_sem_get_count(k_sem_handle_t sem_handle)

Function description:

Get the value of the current counter of the semaphore.

parameter:

sem_handle: The handle of the semaphore.

return value:

Non-negative value: The value of the current counter of the

semaphore. Negative value: error code.

4.5 Mutex

- *csi_kernel_mutex_new*
- *csi_kernel_mutex_del*
- *csi_kernel_mutex_lock*
- *csi_kernel_mutex_unlock*
- *csi_kernel_mutex_get_owner*

A mutex (also known as a mutex or mutex) is a special binary semaphore used for synchronization between tasks, ensuring that only one task has access to a critical resource at any time.

When a task accesses a critical resource, by placing the mutex in a locked state, other tasks will be blocked if they access the resource until the mutex is locked.

The amount can only be accessed after the amount is unlocked by the task. Mutexes ensure that only one task accesses critical resources at the same time, ensuring resource integrity.

```
k_mutex_handle_t csi_kernel_mutex_new(void)
```

Function description:

Create a mutex.

parameter:

none.

return value:

NULL: Creation failed.

Miscellaneous: A handle to the mutex.

```
k_status_t csi_kernel_mutex_del(k_mutex_handle_t mutex_handle)
```

Function description:

Delete a mutex.

parameter:

mutex_handle: The handle of the mutex.

return value:

error code.

```
k_status_t csi_kernel_mutex_lock(k_mutex_handle_t mutex_handle, int32_t timeout)
```

Function description:

Acquire and lock a mutex.

parameter:

mutex_handle: handle of mutex timeout: timeout

(unit: tick). If the mutex cannot be obtained within this time, the function returns.

0 - do not wait.

Negative value - keep waiting.

Other positive values - maximum wait time.

return value:

error code.

---

k_status_t csi_kernel_mutex_unlock(k_mutex_handle_t mutex_handle)

Function description:

Unlock a mutex.

parameter:

mutex_handle: The handle of the mutex.

return value:

error code.

---

k_task_handle_t csi_kernel_mutex_get_owner(k_mutex_handle_t mutex_handle)

Function description:

Get the owner of the mutex.

parameter:

mutex_handle: The handle of the mutex.

return value:

NULL: Fetch failed.

Miscellaneous: The task handle of the mutex owner.

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*

Message queue is a common asynchronous communication mechanism between tasks. Tasks can send data to the message queue or read messages from the queue. from the message team

When reading messages in the queue, if the message in the queue is empty, the read task is suspended. If there is a new message in the queue, the suspended read task is awakened and the new message is processed.

```
k_msgq_handle_t csi_kernel_msgq_new(int32_t msg_count, int32_t msg_size)
```

Function description:

Create a message queue.

parameter:

msg_count: The maximum number of messages in the message

queue. msg_size: The maximum length of each message.

return value:

NULL: Creation failed.

Miscellaneous: The message queue handle.

```
k_status_t csi_kernel_msgq_del(k_msgq_handle_t mq_handle)
```

Function description:

Delete a message queue.

parameter:

mq_handle: message queue handle.

return value:

error code.

```
k_status_t csi_kernel_msgq_put(k_msgq_handle_t mq_handle, const void *msg_ptr, uint8_t front_or_
,ÿback, int32_t timeout)
```

Function description:

Insert a message into the message queue.

parameter:

mq_handle: The handle of the message queue.

msg_ptr: Pointer with inserted message.

front_or_back: The front or back of the queue inserted into the message queue. 1 - head of team; 0 - tail of team. timeout:

Timeout time (unit: tick). If the message insertion fails within this time, the function returns.

0 - do not wait.

Negative value - keep

waiting. Other positive values - maximum wait time.

return value:

error code.

```
k_status_t csi_kernel_msgq_get(k_msgq_handle_t mq_handle, void *msg_ptr, int32_t timeout)
```

Function description:

Get a message from the message queue.

parameter:

mq_handle: The handle of the message queue.

msg_ptr: Pointer to save the acquired message. timeout:

Timeout time (unit: tick). If the message acquisition fails within this time, the function returns.

0 - do not wait

Negative value - keep

waiting Other positive value - maximum waiting time

return value:

error code.

```
int32_t csi_kernel_msgq_get_count(k_msgq_handle_t mq_handle)
```

Function description:

Get the number of messages in the message queue.

parameter:

mq_handle: The handle of the message queue.

return value:

>=0: The number of messages in the message

queue. <0: Error code.

---

uint32_t csi_kernel_msgq_get_capacity(k_msgq_handle_t mq_handle)

Function description:

The maximum number of messages in the message queue.

parameter:

mq_handle: The handle of the message queue.

return value:

<=0: Execution failed.

>0: The maximum number of messages in the message queue.

---

uint32_t csi_kernel_msgq_get_msg_size(k_msgq_handle_t mq_handle)

Function description:

The one with the largest message size in the message queue.

parameter:

mq_handle: The handle of the message queue.

return value:

<=0: Execution failed.

>0: The maximum size supported by a single message, unit: bytes.

---

k_status_t csi_kernel_msgq_flush(k_msgq_handle_t mq_handle)

Function description:

Empty messages in the message queue.

parameter:

mq_handle: The handle of the message queue.

return value:

error code.

4.7 Timer

- *csi_kernel_timer_new*
- *csi_kernel_timer_del*
- *csi_kernel_timer_start*
- *csi_kernel_timer_stop*
- *csi_kernel_timer_get_stat*

The software timer is a timer based on the system clock interrupt and implemented by software, and its precision depends on the period of the system tick clock. The software timer is The user-defined callback function will be triggered after the set tick clock count value.

The software timer supports one-shot mode and periodic trigger mode. A single-trigger timer will only trigger the timer callback function once after it is started, and then the timer will stop running. The periodic trigger timer will periodically trigger the timer callback function until the user manually stops the timer, otherwise it will continue to execute forever.

```
k_timer_handle_t csi_kernel_timer_new(k_timer_cb_t func, k_timer_type_t type, void *arg)
```

Function description:

Create a new software timer.

parameter:

func: Timer timeout callback function. The callback function prototype k_timer_cb_t is defined as follows:

```
typedef void (*k_timer_cb_t)(void *arg);
```

type: Specifies the type of software timer to create. See k_timer_type_t definition arg: the

parameter of the timer callback function func.

return value:

NULL: Creation failed.

Other values: The handle of the timer.

type definition		Note
KTIMER_TYPE_ONCE single trigger type timer		This type of timer fires a timer event only once and then stops.
KTIMER_TYPE_PERIODIC Period trigger type timer		Timers of this type will periodically trigger timer events until the user manually stops the timer (via csi_kernel_timer_stop), otherwise it will continue to execute.

k_status_t csi_kernel_timer_del (k_timer_handle_t timer_handle)

Function description:

Remove a software timer.

parameter:

timer_handle: The handle of the software timer.

return value:

error code.

k_status_t csi_kernel_timer_start(k_timer_handle_t timer_handle, uint32_t ticks)

Function description:

Start a software timer.

parameter:

timer_handle: The handle of the software timer.

ticks: timer period. Unit: System tick.

return value:

error code.

k_status_t csi_kernel_timer_stop(k_timer_handle_t timer_handle)

Function description:

Stop a software timer.

parameter:

timer_handle: The handle of the software timer.

return value:

error code.

k_timer_stat_t csi_kernel_timer_get_stat(k_timer_handle_t timer_handle)

Function description:

Get the running status of the software timer.

parameter:

timer_handle: The handle of the software timer.

return value:

Negative value: error code.

Other values: *The state of the timer. k_timer_stat_t* definition.

name	definition	Remark
PAGE_SIZE_4KB page size is 4KB		
PAGE_SIZE_16KB page size is 16KB		
PAGE_SIZE_64KB page size is 64KB		
PAGE_SIZE_256KB page size is 256KB		
PAGE_SIZE_1MB page size is 1MB		
PAGE_SIZE_4MB page size is 4MB		
PAGE_SIZE_4MB page size is 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*

The system timer is the core mechanism for the normal operation of the real-time operating system. The operating system uses the system timer to update the system time and control task scheduling. The clock of the system timer is also called time stamp or tick, and its frequency is generally configurable, such as 100HZ. The time interface provided by the operating system generally takes tick as the basic unit, so its precision is not very high.

The time-related interface mainly provides users with the following functions: acquisition of system time (tick), delay function, and conversion interface between time and tick value.

k_status_t csi_kernel_delay(uint32_t ticks)

Function description:

Delay the specified amount of time (in kernel ticks) from the current time. The CPU can be released during the delay.

parameter:

ticks: The number of ticks to delay.

return value:

error code.

---

k_status_t csi_kernel_delay_until(uint64_t ticks)

Function description:

Delay to a specific time (specified by the kernel tick counter). The CPU can be released during the delay.

parameter:

ticks: The tick time point that needs to be delayed.

return value:

error code.

---

uint64_t csi_kernel_tick2ms(uint32_t ticks)

Function description:

Conversion between system ticks and milliseconds.

parameter:

ticks: The number of ticks to convert.

return value:

Converted milliseconds.

---

uint64_t csi_kernel_ms2tick(uint32_t ms)

Function description:

Conversion between milliseconds to system ticks.

parameter:

ms: The number of milliseconds to convert.

return value:

Converted number of system ticks.

---

k_status_t csi_kernel_delay_ms(uint32_t ms)

Function description:

Delay the specified amount of time (in milliseconds) from the current time. The CPU can be released during the delay.

parameter:

ms: The number of milliseconds to delay.

return value:

error code.

---

uint64_t csi_kernel_get_ticks(void)

Function description:

Get the system's kernel tick count value.

parameter:

none.

return value:

The system's kernel tick count value.

---

uint32_t csi_kernel_get_tick_freq(void)

Function description:

Get the frequency of ticks, that is, how many tick interrupts are executed in 1 second.

parameter:

none.

return value:

Kernel tick frequency.

```
uint32_t csi_kernel_get_systimer_freq(void)
```

Function description:

Get the hardware frequency value of the systimer.

parameter:

none.

return value:

The frequency of the hardware system clock.

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*

A memory pool manages a piece of memory through fixed-length memory blocks. The user allocates and releases memory in units of fixed length. The memory pool is guaranteed to be positive
Accurate and efficient application for freeing memory.

```
k_mpool_handle_t csi_kernel_mpool_new(void *p_addr, int32_t block_count, int32_t block_size)
```

Function description:

Create a memory pool.

parameter:

p_addr: The first address of the memory pool

buffer. block_count: The number of memory

blocks. block_size: The size of each memory block.

return value:

NULL: Creation failed. Others:

The handle when the creation is successful.

k_status_t csi_kernel_mpool_del(k_mpool_handle_t mp_handle)

Function description:

Delete a memory pool.

parameter:

mp_handle: The handle of the memory pool.

return value:

error code.

---

void *csi_kernel_mpool_alloc(k_mpool_handle_t mp_handle, int32_t timeout)

Function description:

Allocate a block of memory from the memory pool.

parameter:

mq_handle: The handle of the memory pool.

timeout: Timeout time (unit: tick). If the memory block cannot be allocated within this time, the function returns.

0 - do not wait.

Negative value - keep

waiting. Other positive values - maximum wait time.

return value:

NULL: Allocation failed.

Others: The first address of the memory block when the allocation is successful.

---

k_status_t csi_kernel_mpool_free(k_mpool_handle_t mp_handle, void *block)

Function description:

Free a block of memory.

parameter:

mq_handle: The handle of the memory pool.

block: The first address of the memory block to be freed.

return value:

error code.

```
int32_t csi_kernel_mpool_get_count(k_mpool_handle_t mp_handle)
```

Function description:

Get the number of used memory blocks in the memory pool.

parameter:

mq_handle: The handle of the memory pool.

return value:

Negative value: error

code. Others: The number of used memory blocks in the memory pool.

```
uint32_t csi_kernel_mpool_get_capacity(k_mpool_handle_t mp_handle)
```

Function description:

Get the total size of the memory pool.

parameter:

mq_handle: The handle of the memory pool.

return value:

0: Get failed.

Miscellaneous: The total size of the memory pool.

```
uint32_t csi_kernel_mpool_get_block_size(k_mpool_handle_t mp_handle)
```

Function description:

Get the size of each memory block in the memory pool.

parameter:

mq_handle: The handle of the memory pool.

return value:

0: Get failed.

Miscellaneous: The size of the memory block.

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*

Events/events are a mechanism for implementing inter-task communication and can be used to achieve synchronization between tasks. The event informs the task whether an event occurs through the flag bit, and cannot transmit data between tasks. It is a lightweight task communication mechanism. Each event (event) in the interface can include 32 flag bits (flag), and each bit can represent a specific event. Tasks can set, clear, and wait for event flags.

Through the multiple flag bits of the event, a task can wait for the occurrence of multiple events at the same time. The event flag supports waking up tasks waiting for events in the following ways:

All event flag bits are set. Any
event flag bit is set. All event
flag bits are cleared to 0. Any
event flag bit is cleared to 0.

```
k_event_handle_t csi_kernel_event_new(void)
```

Function description:

Create an event.

parameter:

none.

return value:

NULL: Creation failed.

Miscellaneous: A handle to the event when the creation is successful.

```
k_status_t csi_kernel_event_del(k_event_handle_t ev_handle)
```

Function description:

Delete an event.

parameter:

ev_handle: The handle of the event.

return value:

error code.

---

k_status_t csi_kernel_event_set(k_event_handle_t ev_handle, uint32_t flags, uint32_t *ret_flags)

Function description:

Set the flag bit of an event.

parameter:

ev_handle: The handle of the event.

flags: Event flags that need to be set. ret_flags:

Returns the set event flags.

return value:

error code.

---

k_status_t csi_kernel_event_clear(k_event_handle_t ev_handle, uint32_t flags, uint32_t *ret_flags)

Function description:

Clears the flag state of an event.

parameter:

ev_handle: Clear the flag state of an event. flags: Event flags

that need to be cleared. ret_flags: The state of the event flags

before the event flags were cleared.

return value:

error code.

---

k_status_t csi_kernel_event_get(k_event_handle_t ev_handle, uint32_t *ret_flags)

Function description:

Get the flag state of an event.

parameter:

ev_handle: The event handle to operate on. ret_flags:

The flags of the event returned upon successful execution.

return value:

error code.

---

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)

Function description:

parameter:

ev_handle: The event handle to operate on.

flags: pending event flags. options: *time flag options, see the definition of k_event_opt_t .*

clr_on_exit: Whether to clear the corresponding event flag within the function. 1 - clear; 0 - not clear.

actl_flags: The event flag state before the event flag is cleared (parameter clr_on_exit is 1) or the event flag state after waiting for a timeout.

timeout: Timeout time (unit: tick). If the event flag to wait for has not occurred within this time, the function returns.

0 - do not wait.

Negative value - keep waiting.

Other positive values - maximum wait time.

return value:

0: The pending event flag occurs.

Others: Error code when the specified event flag is not waited.

name	definition	Remark
Any bit of 1 in KEVENT_OPT_SET_ANY flags matches		
All bits set to 1 in KEVENT_OPT_SET_ALL flags match		
Any bits in the KEVENT_OPT_CLR_ANY flags that are 0 match		
All 0 bits in KEVENT_OPT_CLR_ALL flags match		

4.11 Heap Management

- *csi_kernel_malloc*
- *csi_kernel_free*
- *csi_kernel_realloc*
- *csi_kernel_get_mminfo*
- *csi_kernel_mm_dump*

Dynamic memory management.

```
void *csi_kernel_malloc(int32_t size, void *caller)
```

Function description:

Allocate a block of memory.

parameter:

size: The requested memory size.

caller: The caller of csi_kernel_malloc, which can be __builtin_return_address(0) or NULL.

return value:

NULL: Application failed. Others:

The first address of memory.

```
void csi_kernel_free(void *ptr, void *caller)
```

Function description:

Free up a block of memory.

parameter:

ptr: The first address of the memory to be freed.

caller: The caller of csi_kernel_free, which can be __builtin_return_address(0) or NULL.

return value:

none.

```
void *csi_kernel_realloc(void *ptr, int32_t size, void *caller)
```

Function description:

Change the size of the allocated memory.

parameter:

ptr: The first address of the memory that has

been requested. size: The changed memory size.

caller: The caller of csi_kernel_realloc, which can be __builtin_return_address(0) or NULL.

return value:

NULL: Application failed. Others:

The first address of the new memory.

```
k_status_t csi_kernel_get_mminfo(int32_t *total, int32_t *used, int32_t *free, int32_t *peak)
```

Function description:

Get dynamic memory management information.

parameter:

total: Returns the total heap size. used:

Returns the used heap size.

free: Returns the remaining heap size.

peak: Returns the peak value of heap usage.

return value:

error code.

```
k_status_t csi_kernel_mm_dump(void)
```

Function description:

Dump heap debugging information.

parameter:

none.

return value:

none.

4.12 Interrupt

- *csi_kernel_intrpt_enter*
- *csi_kernel_intrpt_exit*

Provides a unified interface for incoming and outgoing interrupts.

```
k_status_t csi_kernel_intrpt_enter(void)
```

Function description:

Called when an interrupt is entered. It is generally used to count the number of interrupt nesting levels, and the number of interrupt nesting levels is incremented by 1 each time it is executed.

parameter:

none.

return value:

error code.

---

k_status_t csi_kernel_intrpt_exit(void)

Function description:

Called when the interrupt is exited. It is generally used for the statistics of interrupt nesting levels, and the number of interrupt nesting levels is reduced by 1 each time it is executed.

parameter:

none.

return value:

error code.

4.13 CSI-Kernel ERRNO

error code	illustrate
Upper	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
ENAMEM	Out of memory
EACCES	Permission denied
EFAULT	Bad address
ENOTBLK	Block device required
EBUSY	Device or resource busy
EEXIST	File exists

Continue on next page

Table 4.2 - Continued from previous page

error code	illustrate
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
PUT ON	File table overflow
HE IS DEAD	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

Continue on next page

Table 4.2 - Continued from previous page

error code	illustrate
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

Continue on next page

Table 4.2 - Continued from previous page

error code	illustrate
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
STAY	Stale NFS file handle
EUCLEAN	Structure needs cleaning
UNITNAM	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