
目录

Introduction	1.1
COMMON	1.2
GPIO	1.3
UART	1.4
IRQ	1.5
TIMER	1.6
RTC	1.7
WDT	1.8
IIC	1.9
SPI	1.10
SPIFLASH	1.11
ADC	1.12
DMA	1.13
PWM	1.14
I2S	1.15
CODEC	1.16
ETB	1.17
QSPI	1.18
PIN	1.19
PM	1.20

Copyright © 2020 T-HEAD Semiconductor Co.,Ltd. All rights reserved.

This document is the property of T-HEAD Semiconductor Co.,Ltd. This document may only be distributed to: (i) a T-HEAD party having a legitimate business need for the information contained herein, or (ii) a non-T-HEAD party having a legitimate business need for the information contained herein. No license, expressed or implied, under any patent, copyright or trade secret right is granted or implied by the conveyance of this document. No part of this document may be reproduced, transmitted, transcribed, stored in a retrieval system, translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise without the prior written permission of T-HEAD Semiconductor Co.,Ltd.

Trademarks and Permissions

The T-HEAD Logo and all other trademarks indicated as such herein are trademarks of T-HEAD Semiconductor Co.,Ltd. All other products or service names are the property of their respective owners.

Notice

The purchased products, services and features are stipulated by the contract made between T-HEAD and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied. The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Copyright © 2020 平头哥半导体有限公司，保留所有权利。

本文档的所有权及知识产权归属于平头哥半导体有限公司及其关联公司(下称“平头哥”)。本文档仅能分派给：(i)拥有合法雇佣关系，并需要本文档的信息的平头哥员工，或(ii)非平头哥组织但拥有合法合作关系，并且其需要本文档的信息的合作方。对于本文档，未经平头哥半导体有限公司明示同意，则不能使用该文档。在未经平头哥半导体有限公司的书面许可的情形下，不得复制本文档的任何部分，传播、转录、储存在检索系统中或翻译成任何语言或计算机语言。

SDK API 手册

软件遵循CSI（Chip System Interface）接口规范，使用CSI接口进行测试与评估，同时用户可以参考SDK中集成的各种常用组件以及示例程序进行应用开发，快速形成产品方案。

COMMON

设备说明

CSI的通用接口、结构体定义。

宏与枚举列表

函数	说明
CSI_ASSERT	参数判空，如果为空，则程序停止
CSI_PARAM_CHK	参数判空，如果为空，返回用户传入的第二个参数
CSI_PARAM_CHK_NORETVAL	参数判空，如果为空，直接返回
HANDLE_REG_BASE	返回句柄中的设备基址
HANDLE_IRQ_NUM	返回句柄中的设备中断号
HANDLE_DEV_IDX	返回句柄中的设备ID
HANDLE_IRQ_HANDLER	返回句柄中的设备中断服务函数
csi_error_t	CSI错误码
csi_pm_dev_state_t	设备的电源控制状态
csi_pm_mode_t	电源控制的模式
csi_dev_tag_t	设备标签，区分设备类型和厂商

结构体列表

结构体	说明
csi_dev_t	CSI设备描述
csi_pm_dev_t	CSI电源控制描述
csi_state_t	CSI设备的读写状态
csi_perip_info_t	设备信息存储结构，用于芯片对接

接口列表

函数	说明
target_get	获取设备基本信息描述（中断号，基址等）
target_get_optimal_dma_channel	根据父设备，获取DMA控制器中的最优通道

mdelay	毫秒分辨率的延时函数
udelay	微秒分辨率的延时函数

宏与枚举详细说明

CSI_ASSERT

```
#ifdef CONFIG_DEBUG_MODE
#define CSI_ASSERT(expr) \
    do { \
        if ((int32_t)expr == (int32_t)NULL) { \
            printf("PROGRAM ASSERT\n"); \
            while(1); \
        } \
    } while(0);
#else
#define CSI_ASSERT(expr) ((void)0U)
#endif
};
```

- 功能描述:
 - 用于判空，如果为空，则程序停止，此功能由宏CONFIG_DEBUG_MODE控制。
- 使用场景：
 - 一般用于函数中的参数判断。
- 使用举例

```
static void func_a(uint8_t *buffer)
{
    /* 如果此时打开了宏CONFIG_DEBUG_MODE，当判断参数buffer为空时，则程序会暂停以报警 */
    CSI_ASSERT(buffer);
}
```

CSI_PARAM_CHK

```
#ifdef CONFIG_PARAM_NOT_CHECK
#define CSI_PARAM_CHK(para, err) \
    do { \
        if ((int32_t)para == (int32_t)NULL) { \
            return (err); \
        } \
    } while (0)
#else
#define CSI_PARAM_CHK(para, err)
```

```
#endif
```

- 功能描述:
 - 用于判空，如果传入的第一个参数 `para` 为空，则返回用户传入的第二个参数 `err`。
- 使用场景：
 - 一般用于函数中的参数判断，服务于带返回值的接口。
- 使用示例

```
csi_error_t func_a(uint8_t *buffer)
{
    /* 如果buffer为空，会返回第二个参数（这里是CSI_ERROR），以告知用户 */
    CSI_PARAM_CHK(buffer, CSI_ERROR);
}
```

CSI_PARAM_CHK_NORETVAL

```
#ifdef CONFIG_PARAM_NOT_CHECK
#define CSI_PARAM_CHK_NORETVAL(para) \
    do { \
        if ((int32_t)para == (int32_t)NULL) { \
            return; \
        } \
    } while (0)
#else
#define CSI_PARAM_CHK_NORETVAL(para)
#endif
```

- 功能描述:
 - 用于判空，如果传入的 `para` 为空，则直接退出。
- 使用场景：
 - 一般用于函数中的参数判断，服务于无返回值的接口。
- 使用示例

```
void func_a(uint8_t *buffer)
{
    /* 如果buffer为空，函数会直接退出 */
    CSI_PARAM_CHK_NORETVAL(buffer);
}
```

HANDLE_REG_BASE

```
#define HANDLE_REG_BASE(handle)    (handle->dev.reg_base)
```

- 功能描述:
 - 返回传入句柄的设备基址。

HANDLE_IRQ_NUM

```
#define HANDLE_IRQ_NUM(handle)      (handle->dev.irq_num)
```

- 功能描述:
 - 返回传入句柄的设备中断号。

HANDLE_DEV_IDX

```
#define HANDLE_DEV_IDX(handle)      (handle->dev.idx)
```

- 功能描述:
 - 返回传入句柄的设备ID。

HANDLE_IRQ_HANDLER

```
#define HANDLE_IRQ_HANDLER(handle)  (handle->dev.irq_handler)
```

- 功能描述:
 - 返回传入句柄的设备中断服务函数。

csi_error_t

类型	说明
CSI_OK	成功
CSI_ERROR	错误
CSI_BUSY	忙碌
CSI_TIMEOUT	超时
CSI_UNSUPPORTED	不支持

用于返回CSI接口的调用情况

csi_pm_dev_action_t

类型	说明
PM_DEV_SUSPEND	设备挂起

PM_DEV_SUSPEND	设备挂起
PM_DEV_RESUME	设备恢复

在低功耗应用场景中，用于设置设备的行为模式

csi_pm_mode_t

类型	说明
PM_MODE_RUN	设备正常运行模式
PM_MODE_SLEEP_1	设备1级睡眠
PM_MODE_SLEEP_2	设备2级睡眠
PM_MODE_DEEP_SLEEP_1	设备1级深度睡眠
PM_MODE_DEEP_SLEEP_2	设备2级深度睡眠

在低功耗应用场景中，对设备进行模式设置

csi_dev_tag_t

类型	说明
DEV_DW_UART_TAG	DW厂商的UART设备标签
DEV_DW_TIMER_TAG	DW厂商的TIMER设备标签
.....	持续扩充中

用于区分不同的外设类型

结构体详细说明

csi_dev_t

```
typedef struct csi_dev csi_dev_t;
struct csi_dev {
    uint32_t    reg_base;
    uint8_t     irq_num;
    uint8_t     idx;
    uint16_t    dev_tag;
    void        (*irq_handler)(void *);
#ifdef CONFIG_PM
    csi_pm_dev_t pm_dev;
#endif
};
```

成员	类型	说明
----	----	----

reg_base	uint32_t	设备基址
irq_num	uint8_t	设备中断号
idx	uint8_t	设备ID
dev_tag	uint16_t	设备标签，区分设备类型和厂商
irq_handler	void (irq_handler)(void)	设备中断服务函数
pm_dev	csi_pm_dev_t	CSI的电源控制描述

本结构体为CSI设备的基础描述，包含中断号、基址等信息，其中的 `pm_dev` 电源控制描述是可选项。

csi_pm_dev_t

```
typedef struct {
    slist_t      next;
    csi_error_t  (*pm_action)(csi_dev_t *dev, csi_pm_dev_action_t action);
    uint32_t     *reten_mem;
    uint32_t     size;
} csi_pm_dev_t;
```

成员	类型	说明
next	slist_t	设备描述链表指针
pm_action	csi_error_t (pm_action)(csi_dev_t dev, csi_pm_dev_action_t action)	电源控制事件回调函数
reten_mem	uint32_t	待保存空间的地址
size	uint32_t	待保存空间的容量

csi_state_t

```
typedef struct {
    uint8_t      readable;
    uint8_t      writeable;
    uint8_t      error;
} csi_state_t;
```

成员	类型	说明
readable	uint8_t	可读状态，为1时代表可读，为0时代表不可读
writeable	uint8_t	可写状态，为1时代表可写，为0时代表不可写
error	uint8_t	错误状态，为1表示错误，为0表示没有错误

此结构体作用于CSI接口的异步读写操作，用户在进行异步读写时，可以调用csi_xx_get_state接口获取设备的当前状态来判断是否进行读写操作。

csi_perip_info_t

```
typedef struct {
    uint32_t reg_base;
    uint8_t  irq_num;
    uint8_t  idx;
    uint16_t dev_tag;
} csi_perip_info_t;
```

成员	类型	说明
reg_base	uint32_t	设备寄存器基址
irq_num	uint8_t	设备中断号
idx	uint8_t	设备ID
dev_tag	uint16_t	设备标签

此结构体用于芯片对接时，存储外设的基本信息。在工程中，一款芯片的devices.c中，存在外设信息列表const csi_perip_info_t g_soc_info[]，这个是芯片对接者需要填充的。

接口详细说明

target_get

```
csi_error_t target_get(csi_dev_tag_t dev_tag, uint32_t idx, csi_dev_t *dev)
```

- 功能描述:
 - 通过用户传入的设备标签，获取此设备的基本信息。
- 参数：
 - dev_tag :设备标签，由用户传入
 - idx : 设备ID，同种设备可能同时存在多个，因此需要ID作区分，如UART0的idx=0，UART1的idx=1
 - dev : 设备信息，将设备基址，中断号等信息返回给用户
- 返回值:
 - 错误码csi_error_t

target_get_optimal_dma_channel

```
csi_error_t target_get_optimal_dma_channel(void *dma_list, uint32_t ctrl_num, csi_dev_t *parent_dev, void *ch_info)
```

- 功能描述:
 - 通过用户传入的设备信息，获取当前最优的DMA通道。
 - 传参：
 - `dma_list` : dma控制器句柄数组指针
 - `ctrl_num` : dma控制器数量
 - `parent_dev` : dma通道将要服务的对象设备信息
 - `ch_info` : 用于返回的通道信息
 - 返回值:
 - 错误码csi_error_t
-

mdelay

```
void mdelay(uint32_t ms)
```

- 功能描述:
 - 延时一段时间，单位是毫秒。
 - 参数：
 - `ms` : 需要延时的时间量，单位毫秒
-

udelay

```
void udelay(uint32_t us)
```

- 功能描述:
 - 延时一段时间，单位是微秒。
 - 参数：
 - `ms` : 需要延时的时间量，单位微秒
-

GPIO设备

设备说明

GPIO(General-purpose input/output) 通用型之输入输出的简称。在嵌入式系统中，经常需要控制许多结构简单的外部设备或者电路，这些设备有的需要通过CPU控制，有的需要CPU提供输入信号。并且，许多设备或电路只要求有开/关两种状态就够了，比如LED的亮与灭。对这些设备的控制，使用传统的串口或者并口就显得比较复杂，所以，在嵌入式微处理器上通常提供了一种“通用可编程I/O端口”，也就是GPIO。

接口列表

GPIO的CSI接口说明如下所示：

函数	说明
csi_gpio_init	GPIO设备初始化
csi_gpio_uninit	GPIO设备反初始化
csi_gpio_dir	GPIO配置输入/输出配置
csi_gpio_mode	GPIO配置引脚模式
csi_gpio_irq_mode	配置GPIO中断模式
csi_gpio_irq_enable	使能引脚中断
csi_gpio_debounce	设置debounce模式
csi_gpio_write	设置引脚的电平状态
csi_gpio_read	读取引脚的电平状态
csi_gpio_toggle	翻转引脚电平状态
csi_gpio_attach_callback	注册回调函数
csi_gpio_detach_callback	注销回调函数

接口详细说明

csi_gpio_init

```
csi_error_t csi_gpio_init(csi_gpio_t *gpio, uint32_t port_idx)
```

- 功能描述:

- 通过设备ID初始化对应的GPIO实例。
- 参数:
 - `gpio` : 设备句柄 (需要用户申请句柄空间) 。
 - `port_idx` : 设备ID。
- 返回值:
- 错误码`csi_error_t`

csi_gpio_t

成员	类型	说明
dev	csi_dev_t	设备统一句柄
callback	void (callback)(csi_gpio_t gpio, uint32_t pins, void *arg)	用户回调函数
arg	void*	用户回调函数对应的传参
priv	void*	设备私有变量

csi_gpio_uninit

```
csi_error_t csi_gpio_uninit(csi_gpio_t *gpio)
```

- 功能描述:
 - GPIO实例反初始化。
 - 该接口会释放所有IO口,恢复为默认状态停止GPIO,并且释放相关的软硬件资源。
- 参数:
 - `gpio` : 实例句柄。
- 返回值:
- 错误码`csi_error_t`

csi_gpio_dir

```
csi_error_t csi_gpio_dir(csi_gpio_t *gpio, uint32_t pin_mask, csi_gpio_dir_t dir)
```

- 功能描述:
 - GPIO输入/输出模式配置
- 参数
 - `gpio` : 实例句柄。
 - `pin_mask` : bit位掩码,指定需要设置的bit位,如:0x00ff,代表设置pin0~pin7。
 - `dir` : 输入/输出模式。

- 返回值:
- 错误码csi_error_t

csi_gpio_dir_t

类型	说明
GPIO_DIRECTION_INPUT	输入模式
GPIO_DIRECTION_OUTPUT	输出模式

csi_gpio_mode

```
csi_error_t csi_gpio_mode(csi_gpio_t *gpio, uint32_t pin_mask, csi_gpio_mode_t mode
)
```

- 功能描述:
 - 配置GPIO的引脚模式。
- 参数:
 - gpio : 实例句柄。
 - pin_mask : bit位掩码，指定需要设置的bit位，如:0x00ff，代表设置pin0~pin7。
 - mode : 引脚模式。
- 返回值:
 - 错误码csi_error_t
- 返回值策略说明：
 - 1.传入的引脚中如果有一个引脚配置失败，函数直接退出并返回CSI_ERROR，剩下引脚不配置。
 - 2.传入的引脚中有一个或者多个UNSUPPORT，函数返回CSI_UNSUPPORT，但其它引脚依然会配置。

csi_gpio_mode_t

类型	说明
GPIO_MODE_PULLNONE	悬空输入
GPIO_MODE_PULLUP	上拉输入
GPIO_MODE_PULLDOWN	下拉输入
GPIO_MODE_OPEN_DRAIN	开漏输出
GPIO_MODE_PUSH_PULL	推挽输出

csi_gpio_irq_mode

```
csi_error_t csi_gpio_irq_mode(csi_gpio_t *gpio, uint32_t pin_mask, csi_gpio_irq_mode_t mode)
```

- 功能描述:
 - 配置GPIO的中断模式。
- 参数:
 - `gpio` : 实例句柄。
 - `pin_mask` : bit位掩码, 指定需要设置的bit位, 如:0x00ff, 代表设置pin0~pin7。
 - `mode` : 中断模式。
- 返回值:
- 错误码csi_error_t

csi_gpio_irq_mode_t

类型	说明
GPIO_IRQ_MODE_RISING_EDGE	上升沿中断模式
GPIO_IRQ_MODE_FALLING_EDGE	下降沿中断模式
GPIO_IRQ_MODE_BOTH_EDGE	双边沿模式
GPIO_IRQ_MODE_LOW_LEVEL	低电平模式
GPIO_IRQ_MODE_HIGH_LEVEL	高电平模式

csi_gpio_irq_enable

```
csi_error_t csi_gpio_irq_enable(csi_gpio_t *gpio, uint32_t pin_mask, bool enable)
```

- 功能描述:
 - 使能/禁止引脚中断。
- 参数:
 - `gpio` : 实例句柄。
 - `pin_mask` : bit位掩码, 指定需要设置的bit位, 如:0x00ff代表, 设置pin0~pin7。
 - `enable` : 使能标志位。true: 使能, false: 禁止。
- 返回值:
- 错误码csi_error_t

csi_gpio_debounce

```
csi_error_t csi_gpio_debounce(csi_gpio_t *gpio, uint32_t pin_mask, bool enable)
```

- 功能描述:
 - 设置debounce模式。
- 参数:
 - gpio : 实例句柄。
 - pin_mask : bit位掩码，指定需要设置的bit位，如:0x00ff，代表设置pin0~pin7。
 - enable : 使能标志位。true: 开启去抖功能，false：关闭去抖功能。
- 返回值:
- 错误码csi_error_t

csi_gpio_write

```
void csi_gpio_write(csi_gpio_t *gpio, uint32_t pin_mask, csi_gpio_pin_state_t value)
```

- 功能描述:
 - 设置引脚的电平状态。
- 参数:
 - gpio : 实例句柄。
 - pin_mask : bit位掩码，指定需要设置的bit位，如:0x00ff，代表设置pin0~pin7。
 - value : 引脚状态，定义见csi_gpio_pin_state_t。
- 返回值: 无

csi_gpio_pin_state_t

类型	说明
GPIO_PIN_LOW	低电平
GPIO_PIN_HIGH	高电平

csi_gpio_toggle

```
void csi_gpio_toggle(csi_gpio_t *gpio, uint32_t pin_mask)
```

- 功能描述:
 - 翻转引脚的电平状态。
- 参数:

- `gpio` : 实例句柄。
- `pin_mask` : bit位掩码, 指定需要设置的bit位, 如:0x00ff, 代表设置pin0~pin7。
- 返回值: 无

csi_gpio_read

```
uint32_t csi_gpio_read(csi_gpio_t *gpio, uint32_t pin_mask)
```

- 功能描述:
 - 读取指定引脚掩码的电平状态。
- 参数:
 - `gpio` : 实例句柄。
 - `pin_mask` : bit位掩码, 指定需要设置的bit位, 如:0x00ff, 代表设置pin0~pin7。
- 返回值:
 - 根据位掩码, 得到对应的引脚状态。

csi_gpio_attach_callback

```
csi_error_t csi_gpio_attach_callback(csi_gpio_t *gpio, void *callback, void *arg)
```

- 功能描述:
 - 注册回调函数。
- 参数:
 - `gpio` : 实例句柄。
 - `callback` : GPIO实例的事件回调函数 (一般在中断上下文执行)。
 - `arg` : 回调函数的参数。
- 参数:
- 错误码csi_error_t

csi_gpio_detach_callback

```
void csi_gpio_detach_callback(csi_gpio_t *gpio)
```

- 功能描述:
 - 注销回调函数。
- 参数:
 - `gpio` : 实例句柄。

电平翻转使用示例

```
#include <stdio.h>

#include <soc.h>
#include <drv/gpio.h>
#include <drv/tick.h>
#include <board_config.h>
#include <board_init.h>

// Select pin29 and pin30
#define EXAMPLE_GPIO_PIN_MASK    ( ( 1 << 29 ) | ( 1 << 30 ) )

#define GPIO_CHECK_RETURN(ret)   \
do {                             \
    if (ret != CSI_OK) {         \
        return -1;              \
    }                             \
} while(0);

static csi_gpio_t gpio;

int main(void)
{
    uint32_t tmp = 0;
    int ret;

    board_init();

    /* Initialize GPIO peripheral */
    ret = csi_gpio_init(&gpio, EXAMPLE_TOGGLE_GPIO_IDX);
    GPIO_CHECK_RETURN(ret);

    /* Set input mode */
    ret = csi_gpio_dir(&gpio, EXAMPLE_GPIO_PIN_MASK, GPIO_DIRECTION_OUTPUT);
    GPIO_CHECK_RETURN(ret);

    ret = csi_gpio_write(&gpio, EXAMPLE_GPIO_PIN_MASK, GPIO_PIN_LOW);
    GPIO_CHECK_RETURN(ret);
    while (1) {
        csi_gpio_toggle(&gpio, EXAMPLE_GPIO_PIN_MASK);
        tmp ^= 1;
        printf("gpio set to: %s \r\n", (tmp == 1) ? "high level" : "low level");
        mdelay(1000);
    }
}
```

中断模式使用示例

```
#include <stdio.h>

#include <soc.h>
#include <drv/gpio.h>
#include <drv/tick.h>
#include <board_config.h>
#include <board_init.h>

// Select pin5 and pin6
#define EXAMPLE_GPIO_PIN_MASK      ( ( 1 << 5 ) | ( 1 << 6 ) )

#define GPIO_CHECK_RETURN(ret)      \
do {                                \
    if (ret != CSI_OK) {            \
        return -1;                  \
    }                                \
} while(0);

volatile static bool intr_flag = false;
static csi_gpio_t gpio;

static void gpio_interrupt_handler(csi_gpio_t *gpio, uint32_t pin_mask, void *arg)
{
    intr_flag = true;
}

int main(void)
{
    int ret;

    board_init();

    /* Initialize GPIO peripheral */
    ret = csi_gpio_init(&gpio, EXAMPLE_INTR_GPIO_IDX);
    GPIO_CHECK_RETURN(ret);

    /* Attach callback */
    ret = csi_gpio_attach_callback(&gpio, gpio_interrupt_handler, NULL);
    GPIO_CHECK_RETURN(ret);

    /* Set pull-up mode */
    ret = csi_gpio_mode(&gpio, EXAMPLE_GPIO_PIN_MASK, GPIO_MODE_PULLUP);
    GPIO_CHECK_RETURN(ret);

    /* Set input mode */
    ret = csi_gpio_dir(&gpio, EXAMPLE_GPIO_PIN_MASK, GPIO_DIRECTION_INPUT);
    GPIO_CHECK_RETURN(ret);

    /* Set falling-edge trigger mode */
    ret = csi_gpio_irq_mode(&gpio, EXAMPLE_GPIO_PIN_MASK, GPIO_IRQ_MODE_FALLING_EDG
```

```
E);
GPIO_CHECK_RETURN(ret);

/* Enable irq */
ret = csi_gpio_irq_enable(&gpio, EXAMPLE_GPIO_PIN_MASK, true);
GPIO_CHECK_RETURN(ret);

printf("please change the gpio pin from high to low\n");

while (1) {
    if (intr_flag) {
        printf("gpio pin passed!!!\n");
        intr_flag = false;
    }
}
```

UART设备

说明

UART(Universal Asynchronous Receiver/Transmitter)是一种同步或者异步的串行通信总线接口，发送和接收方按照严格的格式（波特率和数据帧格式）发送和接收。

UART的特点：

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

接口列表

UART的CSI接口如下所示：

函数	说明
csi_uart_init	UART设备初始化
csi_uart_uninit	UART设备反初始化
csi_uart_baud	设置波特率
csi_uart_format	设置数据位，停止位，校验位
csi_uart_flowctrl	设置流控
csi_uart_attach_callback	注册回调函数
csi_uart_detach_callback	注销回调函数
csi_uart_link_dma	设置与DMA设备的连接
csi_uart_send	同步发送数据
csi_uart_send_async	异步发送数据
csi_uart_receive	同步接收数据
csi_uart_receive_async	异步接收数据
csi_uart_putc	发送字节（同步）
csi_uart_getc	接收字节（同步）
csi_uart_get_state	获取UART设备的当前的读写状态

接口详细说明

csi_uart_init

```
csi_error_t csi_uart_init(csi_uart_t *uart, uint32_t idx);
```

- 功能描述:
 - 通过设备号初始化对应的uart实例。
- 参数:
 - `uart` : 设备句柄（需要用户申请句柄空间）
 - `idx` : 设备号
- 返回值:
 - 错误码csi_error_t

csi_uart_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
callback	void (callback)(csi_uart_t uart, csi_uart_event_t event, void *arg)	用户回调函数
arg	void*	回调函数参数（用户自定义）
tx_data	uint8_t*	发送缓存区地址
tx_size	uint32_t	发送数据字节数
rx_data	uint8_t*	接收缓存区地址
rx_size	uint32_t	接收数据字节数
tx_dma	csi_dma_ch_t	用于发送的DMA通道句柄
rx_dma	csi_dma_ch_t	用于接收的DMA通道句柄
send	csi_error_t (send)(csi_uart_t uart, const void *data, uint32_t size)	异步发送函数指针
receive	csi_error_t (receive)(csi_uart_t uart, void *data, uint32_t size)	异步接收函数指针
state	csi_state_t	UART设备读写状态
priv	void*	设备私有变量

csi_uart_uninit

```
void csi_uart_uninit(csi_uart_t *uart);
```

- 功能描述:
 - uart实例反初始化。
 - 该接口会停止uart实例正在进行的传输，并且释放相关的软硬件资源。
- 参数:
 - `uart` : 实例句柄。

csi_uart_baud

```
csi_error_t csi_uart_baud(csi_uart_t *uart, uint32_t baud);
```

- 功能描述:
 - uart设备配置波特率
- 参数
 - `uart` : 实例句柄
 - `baud` : 波特率
- 返回值:
 - 错误码csi_error_t
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_uart_t g_uart;
csi_error_t ret;
ret = csi_uart_baud(&g_uart, 115200);
if (ret != CSI_OK) {
    return -1;
}
```

csi_uart_format

```
csi_error_t csi_uart_format(csi_uart_t *uart, csi_uart_data_bits_t data_bits, csi_uart_parity_t parity, csi_uart_stop_bits_t stop_bits);
```

- 功能描述:
 - uart设备配置数据位，奇偶校验位和停止位
- 参数:
 - `uart` : 实例句柄
 - `data_bits` : 数据位

- `parity` : 校验位
- `stop_bits` : 停止位
- 返回值:
 - 错误码 `csi_error_t`
- 使用示例:

```
/* 句柄使用前请先初始化 */
static csi_uart_t g_uart;
csi_error_t ret;
ret = csi_uart_format(&g_uart, UART_DATA_BITS_8, UART_PARITY_NONE, UART_STOP_BITS_1);
if (ret != CSI_OK) {
    return -1;
}
```

`csi_uart_data_bits_t`

类型	说明
<code>UART_DATA_BITS_5</code>	5位数据位宽
<code>UART_DATA_BITS_6</code>	6位数据位宽
<code>UART_DATA_BITS_7</code>	7位数据位宽
<code>UART_DATA_BITS_8</code>	8位数据位宽
<code>UART_DATA_BITS_9</code>	9位数据位宽

`csi_uart_parity_t`

类型	说明
<code>UART_PARITY_NONE</code>	无校验
<code>UART_PARITY_EVEN</code>	偶校验
<code>UART_PARITY_ODD</code>	奇校验

`csi_uart_stop_bits_t`

类型	说明
<code>UART_STOP_BITS_1</code>	1停止位
<code>UART_STOP_BITS_2</code>	2停止位
<code>UART_STOP_BITS_1_5</code>	1.5停止位

csi_uart_flowctrl

```
csi_error_t csi_uart_flowctrl(csi_uart_t *uart, csi_uart_flowctrl_t flowctrl);
```

- 功能描述:
 - 设置uart设备的流控功能。
- 参数:
 - `uart` : 实例句柄。
 - `flowctrl` : 流控模式
- 返回值:
 - 错误码csi_error_t

csi_uart_flowctrl_t

类型	说明
UART_FLOWCTRL_NONE	无流控
UART_FLOWCTRL_RTS	发送请求（Require ToSend）
UART_FLOWCTRL_CTS	发送允许（Clear ToSend）
UART_FLOWCTRL_RTS_CTS	发送请求与发送允许功能同时打开

注意：由于并不是每个UART设备都支持流控功能，因此设备默认为UART_FLOWCTRL_NONE，不需要特地调用csi_uart_flowctrl将uart设置为UART_FLOWCTRL_NONE。

csi_uart_attach_callback

```
csi_error_t csi_uart_attach_callback(csi_uart_t *uart, void *callback, void *arg);
```

- 功能描述:
 - 设置回调函数，并打开中断的异步模式读写功能
- 参数:
 - `uart` : 实例句柄。
 - `callback` : uart实例的事件回调函数（一般在中断上下文执行）。
 - `arg` : 回调函数参数（可选，由用户定义）。
- 返回值:
 - 错误码csi_error_t。

callback

```
void (*callback)(csi_uart_t *uart, csi_uart_event_t event, void *arg);
```

其中uart为设备句柄，idx为设备号，event 为传给回调函数的事件类型，arg为用户自定义的回调函数对应的参数。uart 回调事件枚举类型csi_uart_event_t定义如下：

事件类型	事件说明
UART_EVENT_SEND_COMPLETE	数据发送完成事件
UART_EVENT_RECEIVE_COMPLETE	数据接收完成事件
UART_EVENT_RECEIVE_FIFO_READABLE	FIFO残留数据等待接收事件
UART_EVENT_BREAK_INTR	数据接收中断事件
UART_EVENT_ERROR_OVERFLOW	数据接收溢出事件
UART_EVENT_ERROR_PARITY	数据校验位错误事件
UART_EVENT_ERROR_FRAMING	数据无有效停止位事件

注意：在使用异步工作模式前，必须调用本函数来注册回调函数，否则将无法使用异步接口。

csi_uart_detach_callback

```
void csi_uart_detach_callback(csi_uart_t *uart);
```

- 功能描述:
 - 注销UART设备的回调函数，并关闭异步读写功能。
- 参数:
 - uart : 实例句柄。

csi_uart_link_dma

```
csi_error_t csi_uart_link_dma(csi_uart_t *uart, csi_dma_ch_t *tx_dma, csi_dma_ch_t *rx_dma);
```

- 功能描述:
 - 将DMA通道句柄连入UART句柄中，并打开UART的DMA发送与接收功能
- 参数:
 - uart : 实例句柄。
 - tx_dma : 用于发送的DMA通道句柄，传NULL时会关闭DMA异步发送功能。
 - rx_dma : 用于接收的DMA通道句柄，传NULL时会关闭DMA异步接收功能。
- 返回值:

- CSI_ERROR: 调用失败。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_uart_t g_uart;
csi_error_t ret;
csi_dma_ch_t g_dma_ch_tx;
csi_dma_ch_t g_dma_ch_rx;
/* 为发送和接收设置DMA通道 */
ret = csi_uart_link_dma(&g_uart, &g_dma_ch_tx, &g_dma_ch_rx);
if (ret != CSI_OK) {
    return -1;
}
```

注意：在调用此函数前，需要先调用csi_uart_attach_callback来注册回调函数，否则将调用失败。

csi_uart_send

```
int32_t csi_uart_send(csi_uart_t *uart, const void *data, uint32_t size, uint32_t timeout);
```

- 功能描述:
 - uart以同步模式启动数据发送。
- 参数:
 - `uart` : 实例句柄。
 - `data` : 待发送数据的缓冲区地址。
 - `size` : 待发送数据的长度。
 - `timeout` : 超时时间，单位是毫秒ms，此超时时间指的是字节与字节的间隔时间，即发送上一个字节与下一个字节的间隔时间超过timeout，则会从函数退出。
- 返回值:
 - 成功发送的字节数或者CSI_ERROR。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_uart_t g_uart;
int32_t ret_num;
const uint8_t tx_test_msg[] = "I am uart";
/* 超时时间请根据实际应用场景进行设置 */
ret_num = csi_uart_send(&g_uart, tx_test_msg, sizeof(tx_test_msg), 50);
```

csi_uart_send_async

```
csi_error_t csi_uart_send_async(csi_uart_t *uart, const void *data, uint32_t size);
```

- 功能描述:
 - uart以异步模式启动数据发送。
- 参数:
 - `uart` : 实例句柄。
 - `data` : 待发送数据的缓冲区地址。
 - `size` : 待发送数据的长度
- 返回值:
 - 错误码`csi_error_t`

注意：在调用此函数前，需要先调用`csi_uart_attach_callback`来注册回调函数。本函数根据用户设置的不同，会进入中断工作模式或DMA工作模式。具体使用方式会在文末用例中综合说明。

csi_uart_receive

```
int32_t csi_uart_receive(csi_uart_t *uart, void *data, uint32_t size, uint32_t timeout);
```

- 功能描述:
 - UART以同步模式启动数据接收。
- 参数:
 - `uart` : 实例句柄。
 - `data` : 待接收数据的缓冲区地址。
 - `size` : 待接收数据的长度。
 - `timeout` : 超时时间，单位是毫秒ms，此超时时间指的是字节与字节的间隔时间，即接收上一个字节与下一个字节的间隔时间超过`timeout`，则会从函数退出。
- 返回值:
 - 成功接收的字节数或者`CSI_ERROR`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_uart_t g_uart;
int32_t ret_num;
uint8_t rx_test_msg[16];
/* 超时时间请根据实际应用场景进行设置 */
ret_num = csi_uart_receive(&g_uart, rx_test_msg, sizeof(rx_test_msg), 0xffffffff);
```

csi_uart_receive_async

```
csi_error_t csi_uart_receive_async(csi_uart_t *uart, void *data, uint32_t size);
```

- 功能描述:
 - UART以异步模式读取数据。
- 参数:
 - `uart` : 实例句柄。
 - `data` : 待接收数据的缓冲区地址。
 - `size` : 预计接收数据的长度。
- 返回值:
 - 错误码`csi_error_t`

注意：在调用此函数前，需要先调用`csi_uart_attach_callback`来注册回调函数。本函数根据用户设置的不同，会进入中断工作模式或DMA工作模式。具体使用方式会在文末用例中综合说明。

`csi_uart_getc`

```
uint8_t csi_uart_getc(csi_uart_t *uart);
```

- 功能描述:
 - 从`uart` 读取一个字节。
- 参数:
 - `uart` : 实例句柄。
- 返回值:
 - 返回读取到的字节内容。

`csi_uart_putc`

```
void csi_uart_putc(csi_uart_t *uart, uint8_t ch);
```

- 功能描述:
 - 从`uart` 发送一个字节。
- 参数:
 - `uart` : 实例句柄。
 - `ch` : 需发送的字节内容。

`csi_uart_get_state`

```
csi_error_t csi_uart_get_state(csi_uart_t *uart, csi_state_t *state);
```

- 功能描述:

- 获取UART的状态。通过此函数来判断UART设备在获取状态的时刻是否可以进行send和receive操作。
- 参数:
 - `uart` : 实例句柄。
 - `state` : 返回读写状态信息
- 返回值:
 - 错误码`csi_error_t`

csi_state_t

类型	说明
readable	设备可读
writable	设备可写
error	错误状态

初始化示例

```
/* 句柄空间一般使用静态空间 */
static csi_uart_t g_uart;

int main() {
    csi_error_t ret;
    /* init函数的idx参数, 请根据soc的实际情况进行选择 */
    ret = csi_uart_init(&g_uart, 0);
    if (ret != CSI_OK) {
        return -1;
    }
}
```

同步模式使用示例

```
static csi_uart_t g_uart;
static uint8_t recv_buf[16];
static char send_buf[128];
#define EXAMPLE_UART_IDX 0
#define EXAMPLE_UART_BAUDRATE 115200
#define UART_CHECK_RETURN(ret) \
do { \
    if (ret != CSI_OK) { \
        return -1; \
    } \
} while(0);
```

```

int example_uart(void)
{
    csi_error_t ret;
    int32_t ret_num;

    /* init uart */
    ret = csi_uart_init(&g_uart, EXAMPLE_UART_IDX);
    UART_CHECK_RETURN(ret);

    /* set uart baudrate */
    ret = csi_uart_baud(&g_uart, EXAMPLE_UART_BAUDRATE);
    UART_CHECK_RETURN(ret);

    /* set uart format */
    ret = csi_uart_format(&g_uart, UART_DATA_BITS_8, UART_PARITY_NONE, UART_STOP_BITS_1);
    UART_CHECK_RETURN(ret);

    strcpy(send_buf, "hello world\n\r");

    ret_num = csi_uart_send(&g_uart, send_buf, strlen(send_buf), 50);
    if (ret_num != strlen(send_buf)) {
        return -1;
    }

    ret_num = csi_uart_receive(&g_uart, recv_buf, sizeof(recv_buf), 0xffffffff);
    if (ret_num != sizeof(recv_buf)) {
        return -1;
    }

    /* Uninit the uart device */
    csi_uart_uninit(&g_uart);
    return 0;
}

```

异步模式使用示例

中断模式和DMA模式的使用流程基本一致，差别仅在是否调用了csi_uart_link_dma接口。

DMA模式的使用示例

```

static csi_uart_t g_uart;
static csi_dma_ch_t g_dma_ch_tx;
static csi_dma_ch_t g_dma_ch_rx;
static volatile uint8_t rx_async_flag = 0;
static volatile uint8_t tx_async_flag = 0;
static uint8_t recv_buf[16];

```

```
static uint8_t send_buf[128];

#define EXAMPLE_UART_BAUDRATE 115200
#define UART_CHECK_RETURN(ret) \
    do { \
        if (ret != CSI_OK) { \
            return -1; \
        } \
    } while(0);

static void uart_event_cb(csi_uart_t *uart, csi_uart_event_t event, void *arg)
{
    switch (event) {
        case UART_EVENT_SEND_COMPLETE:
            tx_async_flag = 1;
            break;

        case UART_EVENT_RECEIVE_COMPLETE:
            rx_async_flag = 1;
            break;

        default:
            break;
    }
}

/* use console uart to show how to use dma mode */
int example_uart_dma()
{
    csi_error_t ret;

    /* init uart, EXAMPLE_UART_IDX == 0 */
    ret = csi_uart_init(&g_uart, EXAMPLE_UART_IDX);
    UART_CHECK_RETURN(ret);

    /* set uart baudrate */
    ret = csi_uart_baud(&g_uart, EXAMPLE_UART_BAUDRATE);
    UART_CHECK_RETURN(ret);

    /* set uart format */
    ret = csi_uart_format(&g_uart, UART_DATA_BITS_8, UART_PARITY_NONE, UART_STOP_BITS_1);
    UART_CHECK_RETURN(ret);

    /* attach callback to uart device */
    ret = csi_uart_attach_callback(&g_uart, uart_event_cb, NULL);
    UART_CHECK_RETURN(ret);

    /* Link DMA */
    ret = csi_uart_link_dma(&g_uart, &g_dma_ch_tx, &g_dma_ch_rx);
    UART_CHECK_RETURN(ret);
}
```

```

strcpy(send_buf, "hello world\n\r");

ret = csi_uart_send_async(&g_uart, send_buf, strlen(send_buf));
UART_CHECK_RETURN(ret);

while(1) {
    if (tx_async_flag) {
        tx_async_flag = 0;
        break;
    }
}

ret = csi_uart_receive_async(&g_uart, recv_buf, sizeof(recv_buf));
UART_CHECK_RETURN(ret);

while(1) {
    if (rx_async_flag) {
        rx_async_flag = 0;
        break;
    }
}

/* Unlink DMA */
ret = csi_uart_link_dma(&g_uart, NULL, NULL);
UART_CHECK_RETURN(ret);

/* Detach the uart callback */
csi_uart_detach_callback(&g_uart);

/* Uninit the uart device */
csi_uart_uninit(&g_uart);
return 0;
}

```

中断模式的使用示例

```

static csi_uart_t g_uart;
static volatile uint8_t rx_async_flag = 0;
static volatile uint8_t tx_async_flag = 0;
static uint8_t recv_buf[16];
static uint8_t send_buf[128];

#define EXAMPLE_UART_BAUDRATE    115200
#define UART_CHECK_RETURN(ret) \
do { \
    if (ret != CSI_OK) { \
        return -1; \
    } \
} \

```



```
    } while(0);

static void uart_event_cb(csi_uart_t *uart, csi_uart_event_t event, void *arg)
{
    switch (event) {
        case UART_EVENT_SEND_COMPLETE:
            tx_async_flag = 1;
            break;

        case UART_EVENT_RECEIVE_COMPLETE:
            rx_async_flag = 1;
            break;

        default:
            break;
    }
}

/* use console uart to show how to use intr mode */
int example_uart_intr()
{
    csi_error_t ret;

    /* init uart, EXAMPLE_UART_IDX == 0 */
    ret = csi_uart_init(&g_uart, EXAMPLE_UART_IDX);
    UART_CHECK_RETURN(ret);

    /* set uart baudrate */
    ret = csi_uart_baud(&g_uart, EXAMPLE_UART_BAUDRATE);
    UART_CHECK_RETURN(ret);

    /* set uart format */
    ret = csi_uart_format(&g_uart, UART_DATA_BITS_8, UART_PARITY_NONE, UART_STOP_BITS_1);
    UART_CHECK_RETURN(ret);

    /* attach callback to uart device */
    ret = csi_uart_attach_callback(&g_uart, uart_event_cb, NULL);
    UART_CHECK_RETURN(ret);

    strcpy(send_buf, "hello world\n\r");

    ret = csi_uart_send_async(&g_uart, send_buf, strlen(send_buf));
    UART_CHECK_RETURN(ret);

    while(1) {
        if (tx_async_flag) {
            tx_async_flag = 0;
            break;
        }
    }
}
```

```
ret = csi_uart_receive_async(&g_uart, recv_buf, sizeof(recv_buf));
UART_CHECK_RETURN(ret);

while(1) {
    if (rx_async_flag) {
        rx_async_flag = 0;
        break;
    }
}

/* Detach the uart callback */
csi_uart_detach_callback(&g_uart);

/* Uninit the uart device */
csi_uart_uninit(&g_uart);
return 0;
}
```

IRQ

说明

IRQ用于管理系统中的中断相关的功能，包括低功耗的中断唤醒，中断服务函数的注册等。

接口列表

函数	说明
csi_irq_enable	使能中断号对应的中断
csi_irq_disable	禁止中断号对应的中断
csi_irq_attach	注册中断号的服务函数
csi_irq_detach	注销中断号对应的服务函数
csi_irq_priority	设置中断优先级
csi_irq_is_enabled	查询中断是否使能
csi_irq_enable_wakeup	使能中断号唤醒功能
csi_irq_disable_wakeup	禁止中断号唤醒功能
csi_irq_context	判断是否在中断处理中
do_irq	统一中断入口

接口详细说明

csi_irq_enable

```
__ALWAYS_STATIC_INLINE void csi_irq_enable(uint32_t irq_num)
```

- 功能描述:
 - 使能中断号对应的中断
- 参数 :
 - `irq_num` : 中断号

csi_irq_disable

```
__ALWAYS_STATIC_INLINE void csi_irq_disable(uint32_t irq_num)
```

- 功能描述:
 - 关闭中断号对应的中断
- 参数 :
 - `irq_num` : 中断号

csi_irq_attach

```
void csi_irq_attach(uint32_t irq_num, void *irq_handler, csi_dev_t *dev)
```

- 功能描述:
 - 注册中断号与设备对应的中断服务函数
 - 参数 :
 - `irq_num` : 中断号
 - `irq_handler` : 中断服务函数
 - `dev` : CSI设备
-

csi_irq_detach

```
void csi_irq_detach(uint32_t irq_num)
```

- 功能描述:
 - 注销中断号对应的中断服务函数
 - 参数 :
 - `irq_num` : 中断号
-

csi_irq_priority

```
__ALWAYS_STATIC_INLINE void csi_irq_priority(uint32_t irq_num, uint32_t priority)
```

- 功能描述:
 - 设置中断的优先级。
 - 参数 :
 - `irq` : 中断号
 - `priority` : 优先级
-

csi_irq_is_enabled

```
static inline bool csi_irq_is_enabled(uint32_t irq_num)
```

- 功能描述:
 - 查看中断是否使能
 - 参数 :
 - `irq_num` : 中断号
 - 返回值:
 - `true`表示已经使能, `false`表示未使能
-

csi_irq_enable_wakeup

```
__ALWAYS_STATIC_INLINE void csi_irq_enable_wakeup(uint32_t irq_num)
```

- 功能描述:
 - 使能对应中断的唤醒功能, 用于低功耗模式唤醒
 - 参数 :
 - `irq_num` : 中断号
-

csi_irq_disable_wakeup

```
__ALWAYS_STATIC_INLINE void csi_irq_disable_wakeup(uint32_t irq_num)
```

- 功能描述:
 - 关闭对应中断的唤醒功能, 用于低功耗模式唤醒
 - 参数 :
 - `irq_num` : 中断号
-

csi_irq_context

```
bool csi_irq_context(void)
```

- 功能描述:
-

- 查询当前是否处于中断中
 - 返回值：
 - true为处于中断，false为不处于中断中
-

do_irq

```
void do_irq(uint32_t irq_num)
```

- 功能描述:
 - 统一中断入口函数
 - 参数：
 - `irq_num` : 中断号
-

Timer

说明

硬件定时器为自动加载计数器，通过API设定超时时间，定时产生中断并进入用户回调函数，以实现定时响应、定时控制。

特点

- 独立可编程定时器
- 可编程超时范围周期
- 达到预定值触发中断事件

接口列表

TIMER的CSI接口如下所示：

函数	说明
csi_timer_init	初始化
csi_timer_uninit	反初始化
csi_timer_start	开始定时
csi_timer_stop	停止定时
csi_timer_get_remaining_value	获取定时中断剩余计数值
csi_timer_get_load_value	获取定时器加载值
csi_timer_is_running	检测定时器工作状态
csi_timer_attach_callback	注册回调函数
csi_timer_detach_callback	注销回调函数

接口详细说明

csi_timer_init

```
csi_error_t csi_timer_init(csi_timer_t *timer, uint32_t idx)
```

- 功能描述:
 - 通过设备ID初始化对应的TIMER实例。

- 参数:
 - `timer` :设备句柄（需要用户申请句柄空间）。
 - `idx` :设备ID。
- 返回值:
 - 错误码`csi_error_t`。

csi_timer_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
callback	void (callback)(csi_timer_t timer, void *arg)	用户回调函数
arg	void *	回调函数对应传参
priv	void *	设备私有变量

csi_timer_uninit

```
void csi_timer_uninit(csi_timer_t *timer)
```

- 功能描述:
 - TIMER实例反初始化，该接口会停止TIMER实例正在进行的工作，并且释放相关的软硬件资源。
- 参数:
 - `timer` :设备句柄。
- 返回值:
 - 无。

csi_timer_start

```
csi_error_t csi_timer_start(csi_timer_t *timer, uint32_t timeout_us)
```

- 功能描述:
 - TIMER计时启动。
- 参数:

- `timer` : 设备句柄。
 - `timeout_us` : 设置超时 (单位us)。
 - 返回值:
 - 错误码`csi_error_t`。
-

csi_timer_stop

```
void csi_timer_stop(csi_timer_t *timer)
```

- 功能描述:
 - TIMER计时停止。
 - 参数:
 - `timer` : 设备句柄。
 - 返回值:
 - 无。
-

csi_timer_get_remaining_value

```
uint32_t csi_timer_get_remaining_value(csi_timer_t *timer)
```

- 功能描述:
 - 获取TIMER距离超时中断的剩余计时值。
 - 参数:
 - `timer` : 设备句柄。
 - 返回值:
 - TIMER剩余计时值。
-

csi_timer_get_load_value

```
uint32_t csi_timer_get_load_value(csi_timer_t *timer)
```

- 功能描述:
 - 获取TIMER加载值。
 - 参数:
-

- `timer` : 设备句柄。
 - 返回值:
 - TIMER加载值。
-

csi_timer_is_running

```
bool csi_timer_is_running(csi_timer_t *timer)
```

- 功能描述:
 - 检测TIMER是否正在工作。
 - 参数:
 - `timer` : 设备句柄。
 - 返回值:
 - `true`代表TIMER正在工作。
 - `false`代表TIMER工作停止。
-

csi_timer_attach_callback

```
csi_error_t csi_timer_attach_callback(csi_timer_t *timer, void *callback, void *arg)
```

- 功能描述:
 - 注册回调函数。
- 参数:
 - `timer` : 设备句柄。
 - `callback` : 中断回调函数。
 - `arg` : 回调函数对应的传参，由用户自定义。
- 返回值:
 - 错误码`csi_error_t`。

callback

```
void (*callback)(csi_timer_t *timer, void *arg);
```

其中TIMER为设备句柄，arg为用户自定义的回调函数对应的参数。

csi_timer_detach_callback

```
void csi_timer_detach_callback(csi_timer_t *timer)
```

- 功能描述:
 - 注销回调函数。
- 参数:
 - timer : 设备句柄。
- 返回值 :
 - 无。

初始化示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/timer.h>

static csi_timer_t g_timer;

int main(void)
{
    csi_error_t ret = 0;

    /* init timer 0 */
    ret = csi_timer_init(&g_timer, 0);

    return ret;
}
```

定时中断模式

```
#include <stdio.h>
#include <soc.h>
#include <drv/timer.h>

#define CHECK_RETURN(ret) \
    do { \
        if (ret != 0) { \
            return -1; \
        } \
    } while(0);
```

```
static csi_timer_t g_timer;
static uint8_t cb_timer_flag = 0;

static void timer_event_cb_reload_fun(csi_timer_t *timer_handle, void *arg)
{
    (void)arg;
    cb_timer_flag = 1;
}

int main(void)
{
    csi_error_t ret = 0;

    /* STEP 1: init timer 0 */
    ret = csi_timer_init(&g_timer, 0);
    CHECK_RETURN(ret);

    /* STEP 2: register callback func */
    ret = csi_timer_attach_callback(&g_timer, timer_event_cb_reload_fun, NULL);
    CHECK_RETURN(ret);

    /* STEP 3: start timer 0 */
    ret = csi_timer_start(&g_timer, 10000000);
    CHECK_RETURN(ret);

    /* STEP 4: clear flag and delay */
    cb_timer_flag = 0;
    mdelay(11000);

    /* STEP 5: judge whether the system enters the interrupt function */
    if (0 == cb_timer_flag) {
        ret = -1;
    }

    /* STEP 6: stop timer 0 */
    csi_timer_stop(&g_timer);

    /* STEP 7: cancel timer */
    csi_timer_detach_callback(&g_timer);

    /* STEP 8: uninit timer */
    csi_timer_uninit(&g_timer);

    return ret;
}
```

RTC

说明

RTC（Real-Time Clock）实时时钟可以提供精确的实时时间，它可以用于产生年、月、日、时、分、秒等信息。目前实时时钟芯片大多采用精度较高的晶体振荡器作为时钟源。有些时钟芯片为了在主电源掉电时还可以工作，会外加电池供电，使时间信息一直保持有效。

接口列表

RTC的CSI接口如下所示：

函数	说明
csi_rtc_init	初始化
csi_rtc_uninit	反初始化
csi_rtc_set_time	设置时间（需要同步时间）
csi_rtc_set_time_no_wait	设置时间（不需要同步时间）
csi_rtc_get_time	获取当前时间
csi_rtc_get_alarm_remaining_time	获取距离闹钟剩余时间
csi_rtc_set_alarm	设置闹钟
csi_rtc_cancel_alarm	取消闹钟
csi_rtc_is_running	获取工作状态

接口详细说明

csi_rtc_init

```
csi_error_t csi_rtc_init(csi_rtc_t *rtc, uint32_t idx)
```

- 功能描述:
 - 通过设备ID初始化对应的RTC实例。
- 参数:
 - `rtc`：设备句柄（需要用户申请句柄空间）。
 - `idx`：设备ID。
- 返回值:

- 错误码csi_error_t。

csi_rtc_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
callback	void (callback)(csi_rtc_t rtc, void *arg)	用户回调函数
arg	void *	回调函数对应传参
priv	void *	设备私有变量

csi_rtc_uninit

```
void csi_rtc_uninit(csi_rtc_t *rtc)
```

- 功能描述:
 - RTC实例反初始化，该接口会停止RTC实例正在进行的工作，并且释放相关的软硬件资源。
- 参数:
 - rtc : 设备句柄。
- 返回值：
 - 无。

csi_rtc_set_time

```
csi_error_t csi_rtc_set_time(csi_rtc_t *rtc, const csi_rtc_time_t *rtctime);
```

- 功能描述:
 - 通过rtctime设置时间（等待同步时间接口）。
- 参数:
 - rtc : 设备句柄。
 - rtctime : rtc设置起始时间年、月、日、时、分、秒。
- 返回值:
 - 错误码csi_error_t。

csi_rtc_time_t

成员	类型	说明
tm_sec	int	秒 取值范围[0-59]
tm_min	int	分 取值范围[0-59]
tm_hour	int	小时 取值范围[0-23]
tm_mday	int	天 取值范围[1-31]
tm_mon	int	月 取值范围[0-11]
tm_year	int	年 取值范围[70-199]
tm_wday	int	周 取值范围[0-6]
tm_yday	int	一年中的第几天 取值范围[0-365]

例：设置2020年1月13日23点59分59秒则如下配置：

```
rtctime.tm_year = 120 , rtctime.tm_mon = 0 , rtctime.tm_mday = 13 rtctime.tm_hour = 23 , rtctime.tm_min = 59 , rtctime.tm_sec = 59
```

csi_rtc_set_time_no_wait

```
csi_error_t csi_rtc_set_time_no_wait(csi_rtc_t *rtc, const csi_rtc_time_t *rtctime)
```

- 功能描述:
 - 通过rtctime设置时间（无需等待同步时间接口）。
- 参数:
 - `rtc`：设备句柄。
 - `rtctime`：rtc设置起始时间年、月、日、时、分、秒。
- 返回值:
 - 错误码csi_error_t。

注意：因硬件不同设计的两种设置时间接口，有些硬件设置时间写入寄存器比较快，直接调用csi_rtc_set_time_no_wait即可快速完成设置时间；有些硬件设置时间写入寄存器响应比较慢，就需要调用csi_rtc_set_time进行设置时间；

csi_rtc_get_time

```
csi_error_t csi_rtc_get_time(csi_rtc_t *rtc, csi_rtc_time_t *rtctime)
```

- 功能描述:

- 获取RTC当前时间。
- 参数:
 - `rtc` : 设备句柄。
 - `rtctime` : rtc返回当前时间年、月、日、时、分、秒。
- 返回值:
 - 错误码`csi_error_t`。

`csi_rtc_get_alarm_remaining_time`

```
uint32_t csi_rtc_get_alarm_remaining_time(csi_rtc_t *rtc)
```

- 功能描述:
 - 获取距离下次闹钟间隔时间。
- 参数:
 - `rtc` : 设备句柄。
- 返回值:
 - 闹钟倒计时(单位 : s)。

`csi_rtc_set_alarm`

```
csi_error_t csi_rtc_set_alarm(csi_rtc_t *rtc, const csi_rtc_time_t *rtctime, void *callback, void *arg)
```

- 功能描述:
 - 设置RTC闹钟时间。
- 参数:
 - `rtc` : 设备句柄。
 - `rtctime` : 设置rtc闹钟时间年、月、日、时、分、秒。
 - `callback` : rtc实例的事件回调函数（一般在中断上下文执行）。
 - `arg` : 回调函数参数（可选，由用户定义）。
- 返回值:
 - 错误码`csi_error_t`。

`callback`


```
void (*callback)(csi_rtc_t *rtc, void *arg);
```

其中rtc为设备句柄，arg为用户自定义的回调函数对应的参数。

csi_rtc_cancel_alarm

```
csi_error_t csi_rtc_cancel_alarm(csi_rtc_t *rtc)
```

- 功能描述:
 - 取消闹钟控制。
 - 参数:
 - rtc : 设备句柄。
 - 返回值:
 - 错误码csi_error_t。
-

csi_rtc_is_running

```
bool csi_rtc_is_running(csi_rtc_t *rtc)
```

- 功能描述:
 - 判断RTC是否还在工作。
 - 参数:
 - rtc : 设备句柄。
 - 返回值:
 - true代表RTC正在工作。
 - false代表RTC工作停止。
-

初始化示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/rtc.h>

static csi_rtc_t g_rtc;
```

```
int main(void)
{
    csi_error_t ret = 0;

    ret = csi_rtc_init(&g_rtc, 0);

    return ret;
}
```

设置时间/读取时间示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/rtc.h>

static csi_rtc_t g_rtc;

#define CHECK_RETURN(ret) \
    do { \
        if (ret != 0) { \
            return -1; \
        } \
    } while(0);

int main(void)
{
    csi_error_t ret;
    csi_rtc_time_t last_time, base_time;

    /* STEP 1: init rtc */
    ret = csi_rtc_init(&g_rtc, 0);
    CHECK_RETURN(ret);

    base_time.tm_year   = 120;
    base_time.tm_mon    = 0;
    base_time.tm_mday   = 5;
    base_time.tm_hour   = 23;
    base_time.tm_min    = 59;
    base_time.tm_sec    = 55;

    /* STEP 2: set base time */
    ret = csi_rtc_set_time(&g_rtc, &base_time);
    CHECK_RETURN(ret);

    /* STEP 3: delay 10s */
    mdelay(10000);

    /* STEP 4: read time */
    ret = csi_rtc_get_time(&g_rtc, &last_time);
```

```

CHECK_RETURN(ret);

/* STEP 5: check time(add 10 sec)  There is a certain error */
CHECK_RETURN((last_time.tm_year - 120));
CHECK_RETURN((last_time.tm_mon - 0));
CHECK_RETURN((last_time.tm_mday - 6));
CHECK_RETURN((last_time.tm_hour - 0));
CHECK_RETURN((last_time.tm_min - 0));
CHECK_RETURN((last_time.tm_sec - 5));

/* STEP 6: uinit rtc */
csi_rtc_uninit(&g_rtc);

return ret;
}

```

设置闹钟示例

```

#include <stdio.h>
#include <soc.h>
#include <drv/rtc.h>

static csi_rtc_t g_rtc;
static uint8_t cb_rtc_flag;

#define CHECK_RETURN(ret) \
    do { \
        if (ret != 0) { \
            return -1; \
        } \
    } while(0);

static void rtc_event_cb_fun(csi_rtc_t *rtc, void *arg)
{
    cb_rtc_flag = 1;
}

int main(void)
{
    csi_error_t ret;
    csi_rtc_time_t last_time, base_time, alarm_time;

    /* STEP 1: init rtc */
    ret = csi_rtc_init(&g_rtc, 0);
    CHECK_RETURN(ret);

    base_time.tm_year    = 120;
    base_time.tm_mon     = 0;
    base_time.tm_mday    = 5;

```

```
base_time.tm_hour    = 23;
base_time.tm_min     = 59;
base_time.tm_sec     = 55;

/* STEP 2: set base time */
ret = csi_rtc_set_time(&g_rtc, &base_time);
CHECK_RETURN(ret);

alarm_time.tm_year    = 120;
alarm_time.tm_mon     = 0;
alarm_time.tm_mday    = 6;
alarm_time.tm_hour    = 0;
alarm_time.tm_min     = 0;
alarm_time.tm_sec     = 5;

cb_rtc_flag = 0;

/* STEP 3: set alarm time */
ret = csi_rtc_set_alarm(&g_rtc, &alarm_time, rtc_event_cb_fun, NULL);
CHECK_RETURN(ret);

/* STEP 4: delay 15s, Trigger alarm interrupt in delay time */
mdelay(15000);

if (cb_rtc_flag == 1) {

    /* STEP 5: get the current time */
    ret = csi_rtc_get_time(&g_rtc, &last_time);
    CHECK_RETURN(ret);

    /* STEP 6: check time(add 15 sec)  There is a certain error */
    CHECK_RETURN((last_time.tm_year - 120));
    CHECK_RETURN((last_time.tm_mon - 0));
    CHECK_RETURN((last_time.tm_mday - 6));
    CHECK_RETURN((last_time.tm_hour - 0));
    CHECK_RETURN((last_time.tm_min - 0));
    CHECK_RETURN((last_time.tm_sec - 10));

    ret = csi_rtc_cancel_alarm(&g_rtc);
    CHECK_RETURN(ret);

} else {
    ret = -1;
}

/* STEP 7: uinit rtc */
csi_rtc_uninit(&g_rtc);

return ret;
}
```

WDT

说明

WatchDog（看门狗）本质上是一个定时器，这个定时器可用来监控程序的运行。看门狗可以设置一个预定的时间，在指定时间内若未对定时器进行喂狗操作将会导致系统复位。程序设计时通过在程序的关键点预埋喂狗动作，当程序由于某种原因（软件或硬件故障）未按指定的逻辑运行时可复位系统，保证系统的可用性。

接口列表

WDT的CSI接口如下所示：

函数	说明
csi_wdt_init	初始化
csi_wdt_uninit	反初始化
csi_wdt_set_timeout	设置超时时间
csi_wdt_start	看门狗开始工作
csi_wdt_stop	看门狗停止工作
csi_wdt_feed	看门狗喂狗
csi_wdt_get_remaining_time	获取看门狗复位剩余时间
csi_wdt_is_running	检测看门狗工作状态
csi_wdt_attach_callback	注册回调函数
csi_wdt_detach_callback	注销回调函数

接口详细说明

csi_wdt_init

```
csi_error_t csi_wdt_init(csi_wdt_t *wdt, uint32_t idx)
```

- 功能描述:
 - 通过设备ID初始化对应的WDT实例。
- 参数:
 - `wdt`：设备句柄（需要用户申请句柄空间）。

- `idx` : 设备ID。
- 返回值:
 - 错误码`csi_error_t`。

`csi_wdt_t`

成员	类型	说明
<code>dev</code>	<code>csi_dev_t</code>	csi设备统一句柄
<code>callback</code>	<code>void (callback)(csi_timer_t timer, void *arg)</code>	用户回调函数
<code>arg</code>	<code>void *</code>	回调函数对应传参
<code>priv</code>	<code>void *</code>	设备私有变量

`csi_wdt_uninit`

```
void csi_wdt_uninit(csi_wdt_t *wdt)
```

- 功能描述:
 - WDT实例反初始化，该接口会停止WDT实例正在进行的工作，并且释放相关的软硬件资源。
- 参数:
 - `wdt` : 实例句柄。
- 返回值:
 - 无。

`csi_wdt_set_timeout`

```
csi_error_t csi_wdt_set_timeout(csi_wdt_t *wdt, uint32_t ms)
```

- 功能描述:
 - 设置看门狗超时时间。
- 参数:
 - `wdt` :设备句柄。
 - `ms` :设置超时时间（单位ms）。
- 返回值:
 - 错误码`csi_error_t`。

csi_wdt_start

```
csi_error_t csi_wdt_start(csi_wdt_t *wdt)
```

- 功能描述:
 - WDT计时启动。
- 参数:
 - `wdt` : 设备句柄。
- 返回值:
 - 错误码csi_error_t。

csi_wdt_stop

```
void csi_wdt_stop(csi_wdt_t *wdt)
```

- 功能描述:
 - WDT计时停止。
- 参数:
 - `wdt` : 设备句柄。
- 返回值:
 - 无。

csi_wdt_feed

```
csi_error_t csi_wdt_feed(csi_wdt_t *wdt)
```

- 功能描述:
 - 看门狗喂狗函数（喂狗会清除中断标志）。
- 参数:
 - `wdt` : 设备句柄。
- 返回值:
 - 错误码csi_error_t。

csi_wdt_get_remaining_time

```
uint32_t csi_wdt_get_remaining_time(csi_wdt_t *wdt)
```

- 功能描述:
 - 获取WDT距离超时中断的剩余计时值。
 - 参数:
 - `wdt` : 设备句柄。
 - 返回值:
 - WDT剩余计时值。
-

csi_wdt_is_running

```
bool csi_wdt_is_running(csi_wdt_t *wdt)
```

- 功能描述:
 - 检测WDT是否正在工作。
 - 参数:
 - `wdt` : 设备句柄。
 - 返回值:
 - `true`代表WDT正在工作。
 - `false`代表WDT工作停止。
-

csi_wdt_attach_callback

```
csi_error_t csi_wdt_attach_callback(csi_wdt_t *wdt, void *callback, void *arg)
```

- 功能描述:
 - 注册回调函数。
 - 参数:
 - `wdt` : 设备句柄。
 - `callback` : 中断回调函数。
 - `arg` : 回调函数对应的传参，由用户自定义。
 - 返回值:
-

- 错误码csi_error_t。

callback

```
void (*callback)(csi_wdt_t *wdt, void *arg);
```

其中wdt为设备句柄，arg为用户自定义的回调函数对应的参数。

csi_wdt_detach_callback

```
void csi_wdt_detach_callback(csi_wdt_t *wdt)
```

- 功能描述:
 - 注销回调函数。
 - 参数:
 - wdt : 设备句柄。
 - 返回值:
 - 无。
-

初始化示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/wdt.h>

static csi_wdt_t g_wdt;

int main(void)
{
    csi_error_t ret = 0;

    /* init wdt */
    ret = csi_wdt_init(&g_wdt, 0);

    return ret;
}
```

正常喂狗示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/wdt.h>

/**
 * The exact time-out can be calculated according to the formula
 * freq = 67500000 Hz
 * user changeable para is 10 (range is 0 ~ 15)
 * 994ms = ((0x10000 <= 10)/ (67500000 / 1000))
 */
#define WDT_TIMEOUT_MS          (994U)          ///< timeout: 0.94s

#define CHECK_RETURN(ret)      \
    do {                        \
        if (ret != 0) {        \
            return -1;         \
        }                     \
    } while(0);

static csi_wdt_t g_wdt;
static uint8_t cb_wdt_flag = 0;

static void wdt_event_cb_fun(csi_wdt_t *wdt, void *arg)
{
    (void)arg;
    cb_wdt_flag = 1;
}

int main(void)
{
    csi_error_t ret = 0;

    /* STEP 1: init wdt */
    ret = csi_wdt_init(&g_wdt, 0);
    CHECK_RETURN(ret);

    /* STEP 2: register callback func */
    ret = csi_wdt_attach_callback(&g_wdt, wdt_event_cb_fun, NULL);
    CHECK_RETURN(ret);

    /* STEP 3: set timeout time(994ms) */
    ret = csi_wdt_set_timeout(&g_wdt, WDT_TIMEOUT_MS);
    CHECK_RETURN(ret);

    cb_wdt_flag = 0;

    /* STEP 4: start work */
    ret = csi_wdt_start(&g_wdt);
    CHECK_RETURN(ret);
}
```

```
/* STEP 5: delay 993ms */
mdelay(WDT_TIMEOUT_MS - 1);

/* STEP 6: system should not enters the interrupt function */
if (cb_wdt_flag) {
    ret = -1;
}

/* STEP 7: feed wdt */
ret = csi_wdt_feed(&g_wdt);
CHECK_RETURN(ret);

return ret;
}
```

系统复位示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/wdt.h>

/**
 * The exact time-out can be calculated according to the formula
 * freq = 67500000 Hz
 * user changeable para is 10 (range is 0 ~ 15)
 * 994ms = ((0x10000 << 10) / (67500000 / 1000))
 */
#define WDT_TIMEOUT_MS          (994U)          ///< timeout: 0.94s

#define CHECK_RETURN(ret)      \
    do {                       \
        if (ret != 0) {        \
            return -1;         \
        }                     \
    } while(0);

static csi_wdt_t g_wdt;
static uint8_t cb_wdt_flag = 0;

static void wdt_event_cb_fun(csi_wdt_t *wdt, void *arg)
{
    (void)arg;
    cb_wdt_flag = 1;
    csi_wdt_feed(wdt);
}

int main(void)
{
    csi_error_t ret = 0;
```

```
/* STEP 1: init wdt */
ret = csi_wdt_init(&g_wdt, 0);
CHECK_RETURN(ret);

/* STEP 2: register callback func */
ret = csi_wdt_attach_callback(&g_wdt, wdt_event_cb_fun, NULL);
CHECK_RETURN(ret);

/* STEP 3: set timeout time(994ms) */
ret = csi_wdt_set_timeout(&g_wdt, WDT_TIMEOUT_MS);
CHECK_RETURN(ret);

cb_wdt_flag = 0;

/* STEP 4: start work */
ret = csi_wdt_start(&g_wdt);
CHECK_RETURN(ret);

/* STEP 5: delay 1194ms */
mdelay(WDT_TIMEOUT_MS + 200);

/* STEP 6: system should enters the interrupt function */
if (0 == cb_wdt_flag) {
    ret = -1;
}

/* STEP 7: start work */
csi_wdt_detach_callback(&g_wdt);

/* STEP 8: delay 995ms */
mdelay(WDT_TIMEOUT_MS + 1);

/* !!!system should be restarted, and not be here! */
ret = -1;

return ret;
}
```

IIC

说明

IIC（Inter Integrated Circuit）总线是 PHILIPS 公司开发的一种半双工、双向二线制同步串行总线。IIC 总线包含两根信号线，双向数据线 SDA，时钟线 SCL）。

接口列表

IIC的CSI接口如下所示：

函数	说明
csi_iic_init	IIC设备初始化
csi_iic_uninit	IIC设备反初始化
csi_iic_mode	IIC设备设置主从模式
csi_iic_addr_mode	IIC设备设置地址模式（7位地址模式/10位地址模式
csi_iic_speed	IIC设备设置传输速度
csi_iic_own_addr	IIC设备设置自身的地址（仅在从机模式下有效）
csi_iic_attach_callback	注册回调函数
csi_iic_detach_callback	注销回调函数
csi_iic_link_dma	设置IIC设备与DMA设备的连接
csi_iic_master_send	IIC设备主机模式下同步发送数据
csi_iic_master_send_async	IIC设备主机模式下异步发送数据
csi_iic_master_receive	IIC设备主机模式下同步接收数据
csi_iic_master_receive_async	IIC设备主机模式下异步接收数据
csi_iic_slave_send	IIC设备从机模式下同步发送数据
csi_iic_slave_send_async	IIC设备从机模式下异步发送数据
csi_iic_slave_receive	IIC设备从机模式下同步接收数据
csi_iic_slave_receive_async	IIC设备从机模式下异步接收数据
csi_iic_mem_send	EEPROM同步发送数据
csi_iic_mem_receive	EEPROM同步接收数据
csi_iic_xfer_pending	IIC设备是否使能连续的发送
csi_iic_get_state	获取IIC设备的当前读写状态

接口详细说明

csi_iic_init

```
csi_error_t csi_iic_init(csi_iic_t *iic, uint32_t idx)
```

- 功能描述:
 - 通过设备ID初始化对应的IIC实例。
- 参数:
 - `iic` : 设备句柄（需要用户申请句柄空间）。
 - `idx` : 设备ID。
- 返回值:
 - 错误码csi_error_t。

csi_iic_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
callback	void(callback)(csi_iic_t iic, csi_iic_event_t event, void *arg)	用户回调函数
arg	void *	回调函数对应传参
data	uint8_t *	用户传入的数据地址
size	uint32_t	用户传入的数据长度
mode	csi_iic_mode_t	IIC设备工作模式
tx_dma	csi_dma_ch_t *	用于发送的DMA通道句柄
rx_dma	csi_dma_ch_t *	用于接收的DMA通道句柄
send	void *	用于异步发送的函数指针
receive	void *	用于异步接收的函数指针
state	csi_state_t	IIC设备的当前状态
priv	void *	设备私有变量

csi_iic_uninit

```
void csi_iic_uninit(csi_iic_t *iic)
```

- 功能描述:
 - iic实例反初始化。
 - 该接口会清理并释放相关的软硬件资源。
- 参数:
 - `iic` : 实例句柄。
- 返回值 :
 - 无。

csi_iic_mode

```
csi_error_t csi_iic_mode(csi_iic_t *iic, csi_iic_mode_t mode)
```

- 功能描述:
 - 设置iic主/从机模式。
- 参数:
 - `iic` : 实例句柄。
 - `mode` : 主从模式。
- 返回值 :
 - 错误码csi_error_t。
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_csi_t g_iic;
csi_error_t ret;
ret = csi_iic_mode(&g_iic, IIC_MODE_MASTER);
if (ret != CSI_OK) {
    return -1;
}
```

csi_iic_mode_t

类型	说明
IIC_MODE_MASTER	IIC主机模式
IIC_MODE_SLAVE	IIC从机模式

csi_iic_addr_mode

```
csi_error_t csi_iic_addr_mode(csi_iic_t *iic, csi_iic_addr_mode_t addr_mode)
```

- 功能描述:

- iic 设置地址模式。
- 参数：
 - `iic` : 实例句柄。
 - `addr_mode` : 地址模式设置。
- 返回值：
 - 错误码 `csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_csi_t g_iic;
csi_error_t ret;
ret = csi_iic_addr_mode(&g_iic, IIC_ADDRESS_7BIT);
if (ret != CSI_OK) {
    return -1;
}
```

`csi_iic_addr_mode_t`

类型	说明
<code>IIC_ADDRESS_7BIT</code>	7位地址模式
<code>IIC_ADDRESS_10BIT</code>	10位地址模式

`csi_iic_speed`

```
csi_error_t csi_iic_speed(csi_iic_t *iic, csi_iic_speed_t speed)
```

- 功能描述：
 - 设置iic传输速度。
- 参数：
 - `iic` : 实例句柄。
 - `speed` : 传输速度设置。
- 返回值：
 - 错误码 `csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_csi_t g_iic;
csi_error_t ret;
ret = csi_iic_speed(&g_iic, IIC_BUS_SPEED_STANDARD);
if (ret != CSI_OK) {
    return -1;
}
```



```
}
```

csi_iic_speed_t

类型	说明
IIC_BUS_SPEED_STANDARD	standard Speed (<=100kHz)
IIC_BUS_SPEED_FAST	fast Speed (<=400kHz)
IIC_BUS_SPEED_FAST_PLUS	fast+ Speed (<= 1MHz)
IIC_BUS_SPEED_HIGH	high Speed (<=3.4MHz)

csi_iic_own_addr

```
csi_error_t csi_iic_own_addr(csi_iic_t *iic, uint32_t own_addr)
```

- 功能描述:
 - 从机模式下，IIC 设置自身地址。
- 参数:
 - `iic` : 实例句柄。
 - `own_addr` : 从机模式下，设置设备响应地址。
- 返回值:
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_csi_t g_iic;
csi_error_t ret;
ret = csi_iic_own_addr(&g_iic, 0x50);
if (ret != CSI_OK) {
    return -1;
}
```

csi_iic_attach_callback

```
csi_error_t csi_iic_attach_callback(csi_iic_t *iic, void *callback, void *arg)
```

- 功能描述：
 - 注册回调函数。
- 参数：
 - `iic` :实例句柄。

- `callback` : iic实例的事件回调函数（一般在中断上下文执行）。
- `arg` : 回调函数参数（可选，由用户定义）。
- 返回值：
 - 错误码`csi_error_t`。

callback

```
void (*callback)(csi_iic_t *iic, csi_iic_event_t event, void *arg)
```

其中 `iic` 为设备句柄，`event` 为传给回调函数的事件类型，`arg` 为用户自定义的回调函数对应的参数。

iic 回调事件枚举类型`csi_iic_event_t`定义如下：

事件类型	事件说明
IIC_EVENT_SEND_COMPLETE	数据发送完成事件
IIC_EVENT_RECEIVE_COMPLETE	数据接收完成事件
IIC_EVENT_ERROR_OVERFLOW	IIC总线产生FIFO溢出事件
IIC_EVENT_ERROR_UNDERFLOW	IIC总线产生FIFO下溢事件
IIC_EVENT_ERROR	总线产生错误

注意：在使用异步工作模式前，必须调用本函数来注册回调函数，否则将无法使用异步接口。

csi_iic_detach_callback

```
void csi_iic_detach_callback(csi_iic_t *iic)
```

- 功能描述：
 - 注销IIC设备的回调函数，并关闭异步读写功能。
- 参数：
 - `iic` : 实例句柄。
- 返回值：
 - 无。

csi_iic_link_dma

```
csi_error_t csi_iic_link_dma(csi_iic_t *iic, csi_dma_ch_t *tx_dma, csi_dma_ch_t *rx_dma)
```

- 功能描述：
 - 将DMA通道句柄连入iic句柄中，并打开IIC的DMA发送与接收功能。
- 参数：

- `iic` : 实例句柄。
- `tx_dma` : 用于发送的DMA通道句柄, 传NULL时会关闭DMA异步发送功能。
- `rx_dma` : 用于接收的DMA通道句柄, 传NULL时会关闭DMA异步发送功能。
- 返回值:
 - 错误码`csi_error_t`。
- 使用示例:

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
csi_dma_ch_t g_dma_ch_tx;
csi_dma_ch_t g_dma_ch_rx;
/* 为发送和接收设置DMA通道 */
ret = csi_iic_link_dma(&g_iic, &g_dma_ch_tx, &g_dma_ch_rx);
if (ret != CSI_OK) {
    return -1;
}
```

csi_iic_master_send

```
int32_t csi_iic_master_send(csi_iic_t *iic, uint32_t devaddr, const void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
 - 主机模式下, IIC 以同步模式数据发送。
- 参数:
 - `iic` : 实例句柄。
 - `devaddr` : 从机设备地址。
 - `data` : 待发送数据缓冲区地址。
 - `size` : 待发送数据的长度。
 - `timeout` : 设置主机发送超时时间(在超时时间内, 无从机响应, 或传输过程中超时), 单位是毫秒。
- 返回值:
 - 成功发送的字节数或错误码`csi_error_t`。
- 使用示例:

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
uint32_t devaddr = 0x50;
int32_t num = 0;
const uint8_t tx_test_msg[3];
tx_test_msg[0] = 0x01;
```

```

tx_test_msg[1] = 0x02;
tx_test_msg[2] = 0x03;
/* 超时时间请根据实际应用场景进行设置 */
num = csi_iic_master_send(&iic, devaddr, tx_test_msg, sizeof(tx_test_msg), 50);
if (num != sizeof(tx_test_msg)) {
    return -1;
}

```

csi_iic_master_send_async

```

csi_error_t csi_iic_master_send_async(csi_iic_t *iic, uint32_t devaddr, void *data,
uint32_t size)

```

- 功能描述:
 - 主机模式下，IIC 以异步模式启动数据发送
- 参数:
 - `iic` : 实例句柄
 - `devaddr` : 从机设备地址
 - `data` : 待发送数据缓冲区地址
 - `size` : 待发送数据的长度
- 返回值:
 - 错误码 `csi_error_t`。

注意：在调用此函数前，需要先调用 `csi_iic_attach_callback` 来注册回调函数。本函数会根据用户设置的不同，进入中断工作模式或DMA工作模式。具体使用方式会在文末用例中综合说明。

csi_iic_master_receive

```

int32_t csi_iic_master_receive(csi_iic_t *iic, uint32_t devaddr, void *data, uint32_t size, uint32_t timeout)

```

- 功能描述:
 - 主机模式下，IIC 以同步模式进行数据接收。
- 参数:
 - `iic` : 实例句柄。
 - `devaddr` : 从机设备地址。
 - `data` : 待接收数据缓冲区地址。
 - `size` : 待接收数据的长度。
 - `timeout` : 设置主机接收超时时间(在超时时间内，无从机响应,或传输过程中超时)，单位是毫秒。

- 返回值:
 - 成功接收的字节数或错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
uint32_t devaddr = 0x50;
int32_t num = 0;
const uint8_t rx_test_msg[3];
/* 超时时间请根据实际应用场景进行设置 */
num = csi_iic_master_receive(&g_iic, devaddr, rx_test_msg, sizeof(rx_test_msg),
50);
if (num != sizeof(rx_test_msg)) {
    return -1;
}
```

csi_iic_master_receive_async

```
csi_error_t csi_iic_master_receive_async(csi_iic_t *iic, uint32_t devaddr, void *data,
uint32_t size)
```

- 功能描述:
 - 主机模式下，IIC 以异步模式进行数据读取。
- 参数:
 - iic : 实例句柄。
 - devaddr : 从机设备地址。
 - data : 待接收数据缓冲区地址。
 - size : 数据的长度。
- 返回值:
 - 错误码csi_error_t。

注意：在调用此函数前，需要先调用csi_iic_attach_callback来注册回调函数。本函数会根据用户设置的不同，进入中断工作模式或DMA工作模式。具体使用方式会在文末用例中综合说明

csi_iic_mem_send

```
int32_t csi_iic_mem_send(csi_iic_t *iic, uint32_t devaddr, uint16_t memaddr, csi_iic_mem_addr_size_t memaddr_size, const void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
 - IIC 以同步模式对eeprom写数据。

- 参数:

- `iic` : 实例句柄。
- `devaddr` : 从机设备地址。
- `memaddr` : 待写入的器件存储地址。
- `memaddr_size` : 器件存储地址大小。
- `data` : 待写入数据缓冲区地址。
- `size` : 待写入数据的长度。
- `timeout` : 超时时间, 单位是毫秒。

- 返回值:

- 成功写入的字节数或错误码 `csi_error_t`。

- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
uint32_t devaddr = 0x50;
uint32_t memaddr = 0x0010;
int32_t num = 0;
const uint8_t tx_test_msg[3];
tx_test_msg[0] = 0x01;
tx_test_msg[1] = 0x02;
tx_test_msg[2] = 0x03;
/* 超时时间请根据实际应用场景进行设置 */
num = csi_iic_mem_send(&g_iic, devaddr, memaddr, IIC_MEM_ADDR_SIZE_16BIT, tx_test_msg, sizeof(tx_test_msg), 50);
if (num != sizeof(tx_test_msg)) {
    return -1;
}
```

csi_iic_mem_addr_size_t

| 类型 | 说明 | | :----- | :----- | | IIC_MEM_ADDR_SIZE_8BIT | EEPROM的地址模式是8位 | | IIC_MEM_ADDR_SIZE_16BIT | EEPROM的地址模式是16位 |

csi_iic_mem_receive

```
int32_t csi_iic_mem_receive(csi_iic_t *iic, uint32_t devaddr, uint16_t memaddr, csi_iic_mem_addr_size_t memaddr_size, void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:

- IIC 以同步模式对eeprom读数据。

- 参数:

- `iic` : 实例句柄。

- `devaddr` : 从机设备地址。
- `memaddr` : 待读取的器件存储地址。
- `memaddr_size` : 器件存储地址模式。
- `data` : 待读取数据缓冲区地址。
- `size` : 待读取数据的长度。
- `timeout` : 超时时间，单位是毫秒。
- 返回值:
 - 成功读出的字节数或错误码 `csi_error_t`。
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
uint32_t devaddr = 0x50;
uint32_t memaddr = 0x0010;
int32_t num = 0;
const uint8_t rx_test_msg[3];
/* 超时时间请根据实际应用场景进行设置 */
num = csi_iic_mem_receive(&g_iic, devaddr, memaddr, IIC_MEM_ADDR_SIZE_16BIT, rx_test_msg, sizeof(rx_test_msg), 50);
if (num != sizeof(rx_test_msg)) {
    return -1;
}
```

csi_iic_slave_send

```
int32_t csi_iic_slave_send(csi_iic_t *iic, const void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
 - 从机模式下，IIC 以同步模式数据发送。
- 参数:
 - `iic` : 实例句柄。
 - `data` : 待发送数据缓冲区地址。
 - `size` : 待发送数据的长度。
 - `timeout` : 设置从机发送数据超时时间(在超时时间内，无主机对从机读取数据操作，或传输过程中超时)，单位是毫秒。
- 返回值:
 - 成功发送的字节数或错误码 `csi_error_t`。
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
```

```

csi_error_t ret;
int32_t num = 0;
const uint8_t tx_test_msg[3];
tx_test_msg[0] = 0x01;
tx_test_msg[1] = 0x02;
tx_test_msg[2] = 0x03;
/* 超时时间请根据实际应用场景进行设置 */
num = csi_iic_slave_send(&g_iic, tx_test_msg, sizeof(tx_test_msg), 50);
if (num != sizeof(tx_test_msg)) {
    return -1;
}

```

csi_iic_slave_send_async

```
csi_error_t csi_iic_slave_send_async(csi_iic_t *iic, void *data, uint32_t size)
```

- 功能描述:
 - 从机模式下，IIC 以异步模式启动数据发送。
- 参数:
 - `iic` : 实例句柄。
 - `data` : 待发送数据缓冲区地址。
 - `size` : 待发送数据的长度。
- 返回值:
 - 错误码 `csi_error_t`。

注意：在调用此函数前，需要先调用 `csi_iic_attach_callback` 来注册回调函数。本函数会根据用户设置的不同，进入中断工作模式或DMA工作模式。具体使用方式会在文末用例中综合说明。

csi_iic_slave_receive

```
int32_t csi_iic_slave_receive(csi_iic_t *iic, void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
 - 从机模式下，IIC 以同步模式进行数据接收。
- 参数:
 - `iic` : 实例句柄。
 - `data` : 待接收数据缓冲区地址。
 - `size` : 待接收数据的长度。
 - `timeout` : 设置从机接收超时时间(在超时时间内，无主机对从机进行写数据操作，或传输过程中超时)，单位是毫秒。
- 返回值:

- 成功接收的字节数或错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
uint32_t devaddr = 0x50;
int32_t num = 0;
const uint8_t rx_test_msg[3];
/* 超时时间请根据实际应用场景进行设置 */
num = csi_iic_slave_receive(&g_iic, rx_test_msg, sizeof(rx_test_msg), 50);
if (num != sizeof(rx_test_msg)) {
    return -1;
}
```

csi_iic_slave_receive_async

```
csi_error_t csi_iic_slave_receive_async(csi_iic_t *iic, void *data, uint32_t size)
```

- 功能描述:
 - 从机模式下，IIC 以异步模式进行数据读取。
- 参数:
 - iic : 实例句柄。
 - data : 待接收数据缓冲区地址。
 - size : 数据的长度。
- 返回值:
 - 错误码csi_error_t。

注意：在调用此函数前，需要先调用csi_iic_attach_callback来注册回调函数。本函数会根据用户设置的不同，进入中断工作模式或DMA工作模式。具体使用方式会在文末用例中综合说明

csi_iic_xfer_pending

```
csi_error_t csi_iic_xfer_pending(csi_iic_t *iic, bool enable)
```

- 功能描述：
 - IIC是否使能连续的发送。
- 参数：
 - iic : 实例句柄。
 - enable : 是否使能连续的发送。
- 返回值：

- 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_iic_t g_iic;
csi_error_t ret;
ret = csi_iic_xfer_pending(&g_iic, true);
if (ret != CSI_OK) {
    return -1;
}
```

csi_iic_get_state

```
csi_error_t csi_iic_get_state(csi_iic_t *iic, csi_state_t *state)
```

- 功能描述:
 - 获取 iic 的状态。通过此函数来判断IIC设备在获取状态的时刻是否可以进行send和receive操作。
- 参数：
 - iic：实例句柄。
 - state: 用于返回状态信息的参数地址。
- 返回值：
 - 错误码csi_error_t。

初始化示例

```
/* 句柄空间一般使用静态空间 */
static csi_iic_t g_iic;

int main(void) {
    csi_error_t ret;
    /* init函数的idx参数，请根据soc的实际情况进行选择 */
    ret = csi_iic_init(&g_iic, 0);
    if (ret != CSI_OK) {
        return -1;
    }
}
```

同步模式使用示例

主机模式发送接收使用示例

```
#include "stdio.h"
#include "drv/iic.h"
```

```
#define IIC_SLAVE_ADDR    0x50
#define IIC_IDX           0
static csi_iic_t master_iic;

int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    int32_t num;
    ret = csi_iic_init(&master_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
    /* config iic master mode */
    ret = csi_iic_mode(&master_iic, IIC_MODE_MASTER);
    if (ret != CSI_OK) {
        printf("csi_iic_set_mode error\n");
        return -1;
    }
    /* config iic 7bit address mode */
    ret = csi_iic_addr_mode(&master_iic, IIC_ADDRESS_7BIT);
    if (ret != CSI_OK) {
        printf("csi_iic_set_addr_mode error\n");
        return -1;
    }
    /* config iic standard speed*/
    ret = csi_iic_speed(&master_iic, IIC_BUS_SPEED_STANDARD);
    if (ret != CSI_OK) {
        printf("csi_iic_set_speed error\n");
        return -1;
    }

    for (i = 0; i < sizeof(write_data); i++) {
        write_data[i] = i;          ///< init write_data value
    }

    num = csi_iic_master_send(&master_iic, IIC_SLAVE_ADDR, write_data, sizeof(write_data), 100000);
    if (num != sizeof(write_data)) {
        printf("csi_iic_master_send error\n");
        return -1;
    }

    num = csi_iic_master_receive(&master_iic, IIC_SLAVE_ADDR, read_data, sizeof(read_data), 100000);
    if (num != sizeof(read_data)) {
        printf("csi_iic_master_receive error\n");
    }
}
```

```

        return -1;
    }
    csi_iic_uninit(&master_iic);
    return 0;
}

```

注意：本示例包含了IIC的主机初始化，配置，主机同步收发接口的调用示例。实际使用还需要GPIO的初始化，以及和从设备的上拉连接，主机收发需要从机配合读取。

从机模式发送接收使用示例

```

#include "stdio.h"
#include "drv/iic.h"

#define IIC_OWN_ADDR    0x50
#define IIC_IDX         0
static csi_iic_t slave_iic;

int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    int32_t num;
    ret = csi_iic_init(&slave_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
    /* config iic slave mode */
    ret = csi_iic_mode(&slave_iic, IIC_MODE_SLAVE);
    if (ret != CSI_OK) {
        printf("csi_iic_set_mode error\n");
        return -1;
    }
    /* config iic 7bit address mode */
    ret = csi_iic_addr_mode(&slave_iic, IIC_ADDRESS_7BIT);
    if (ret != CSI_OK) {
        printf("csi_iic_set_addr_mode error\n");
        return -1;
    }
    /* config iic standard speed*/
    ret = csi_iic_speed(&slave_iic, IIC_BUS_SPEED_STANDARD);
    if (ret != CSI_OK) {
        printf("csi_iic_set_speed error\n");
        return -1;
    }
    /* config iic slave own address*/
}

```

```

ret = csi_iic_own_addr(&slave_iic, IIC_OWN_ADDR);
if (ret != CSI_OK) {
    printf("csi_iic_own_addr error\n");
    return -1;
}

for (i = 0; i < sizeof(write_data); i++) {
    write_data[i] = i;          ///< init write_data value
}

num = csi_iic_salve_send(&slave_iic, write_data, sizeof(write_data), 100000);
if (num != sizeof(write_data)) {
    printf("csi_iic_salve_send error\n");
    return -1;
}

num = csi_iic_salve_receive(&slave_iic, read_data, sizeof(read_data), 100000);
if (num != sizeof(read_data)) {
    printf("csi_iic_salve_receive error\n");
    return -1;
}
csi_iic_uninit(&salve_iic);
return 0;
}

```

注意：本示例包含了IIC的从机初始化，配置，从机同步收发接口的调用示例。实际使用还需要GPIO的初始化，以及和主设备的上拉连接，从机发送需要主机配合读取。

EEPROM读写使用示例

```

#include "stdio.h"
#include "drv/iic.h"

#define IIC_SLAVE_ADDR    0x50
#define IIC_MEM_ADDR      0x0010
#define IIC_IDX           0
static csi_iic_t master_iic;

int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    int32_t num;
    ret = csi_iic_init(&master_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
}

```

```

}
/* config iic master mode */
ret = csi_iic_mode(&master_iic, IIC_MODE_MASTER);
if (ret != CSI_OK) {
    printf("csi_iic_set_mode error\n");
    return -1;
}
/* config iic 7bit address mode */
ret = csi_iic_addr_mode(&master_iic, IIC_ADDRESS_7BIT);
if (ret != CSI_OK) {
    printf("csi_iic_set_addr_mode error\n");
    return -1;
}
/* config iic standard speed*/
ret = csi_iic_speed(&master_iic, IIC_BUS_SPEED_STANDARD);
if (ret != CSI_OK) {
    printf("csi_iic_set_speed error\n");
    return -1;
}

for (i = 0; i < sizeof(write_data); i++) {
    write_data[i] = i;          ///< init write_data value
}

num = csi_iic_mem_send(&master_iic, IIC_SLAVE_ADDR, IIC_MEM_ADDR, IIC_MEM_ADDR_SIZE_16BIT, write_data, sizeof(write_data), 100000);
if (num != sizeof(write_data)) {
    printf("csi_iic_mem_send error\n");
    return -1;
}

num = csi_iic_mem_receive(&master_iic, IIC_SLAVE_ADDR, IIC_MEM_ADDR, IIC_MEM_ADDR_SIZE_16BIT, read_data, sizeof(read_data), 100000);
if (num != sizeof(read_data)) {
    printf("csi_iic_mem_receive error\n");
    return -1;
}
}
csi_iic_uninit(&master_iic);
return 0;
}

```

注意：本示例包含了IIC的主机初始化，配置，EEPROM同步收发接口的调用示例。实际使用还需要GPIO的初始化，以及和EEPROM的上拉连接，本示例选用的是16bit地址宽度的EEPROM。

异步模式使用示例

中断模式和DMA模式的使用流程基本一致，差别仅在是否调用了csi_iic_link_dma接口。

DMA模式的主机收发使用示例

```
#include "stdio.h"
#include "drv/iic.h"

#define IIC_SLAVE_ADDR    0x50
#define IIC_IDX           0
static csi_iic_t master_iic;
static csi_dma_ch_t dma_ch_tx_handle;
static volatile uint8_t cb_transfer_flag = 0;
static void iic_event_cb_fun(csi_iic_t *iic, csi_iic_event_t event, void *arg)
{
    if (event == IIC_EVENT_RECEIVE_COMPLETE) {
        cb_transfer_flag = 1;
    }

    if (event == IIC_EVENT_SEND_COMPLETE) {
        cb_transfer_flag = 1;
    }
}

int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    ret = csi_iic_init(&master_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
    /* iic attach callback */
    ret = csi_iic_attach_callback(&master_iic, iic_event_cb_fun, NULL);
    if (ret != CSI_OK) {
        printf("csi_iic_attach_callback error\n");
        return -1;
    }

    /* config iic master mode */
    ret = csi_iic_mode(&master_iic, IIC_MODE_MASTER);
    if (ret != CSI_OK) {
        printf("csi_iic_set_mode error\n");
        return -1;
    }
    /* config iic 7bit address mode */
    ret = csi_iic_addr_mode(&master_iic, IIC_ADDRESS_7BIT);
    if (ret != CSI_OK) {
        printf("csi_iic_set_addr_mode error\n");
        return -1;
    }
    /* config iic standard speed*/
}
```

```

ret = csi_iic_speed(&master_iic, IIC_BUS_SPEED_STANDARD);
if (ret != CSI_OK) {
    printf("csi_iic_set_speed error\n");
    return -1;
}

/* use this interface connect iic tx with dma */
ret = csi_iic_link_dma(&master_iic, &dma_ch_tx_handle, NULL);
if (ret != CSI_OK) {
    printf("csi_iic_link_dma fail \n");
    return -1;
}

for (i = 0; i < sizeof(write_data); i++) {
    write_data[i] = i;          ///< init write_data value
}

ret = csi_iic_master_send_async(&master_iic, IIC_SLAVE_ADDR, write_data, sizeof(w
rite_data), 100000);
if (ret != CSI_OK) {
    printf("csi_iic_master_send error\n");
    return -1;
}

ret = csi_iic_master_receive_async(&master_iic, IIC_SLAVE_ADDR, read_data, sizeof
(read_data), 100000);
if (ret != CSI_OK) {
    printf("csi_iic_master_receive error\n");
    return -1;
}

/* use this interface disconnect iic with dma */
csi_iic_link_dma(&master_iic, NULL, NULL);
/* use this interface disconnect iic callback */
csi_iic_detach_callback(&master_iic);
csi_iic_uninit(&master_iic);
return 0;
}

```

DMA模式的从机收发使用示例

```

#include "stdio.h"
#include "drv/iic.h"

#define IIC_OWN_ADDR    0x50
#define IIC_IDX         0
static csi_iic_t slave_iic;
static csi_dma_ch_t dma_ch_rx_handle;
static volatile uint8_t slave_cb_transfer_done_flag = 0;
static void slave_iic_event_cb_fun(csi_iic_t *iic, csi_iic_event_t event, void *arg
)

```



```
{
    if (event == IIC_EVENT_RECEIVE_COMPLETE) {
        slave_cb_transfer_done_flag = 1;
    }

    if (event == IIC_EVENT_SEND_COMPLETE) {
        slave_cb_transfer_done_flag = 1;
    }
}

int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    ret = csi_iic_init(&slave_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
    /* iic attach callback */
    ret = csi_iic_attach_callback(&slave_iic, slave_iic_event_cb_fun, NULL);
    if (ret != CSI_OK) {
        printf("csi_iic_attach_callback error\n");
        return -1;
    }
    /* config iic slave mode */
    ret = csi_iic_mode(&slave_iic, IIC_MODE_SLAVE);
    if (ret != CSI_OK) {
        printf("csi_iic_set_mode error\n");
        return -1;
    }
    /* config iic 7bit address mode */
    ret = csi_iic_addr_mode(&slave_iic, IIC_ADDRESS_7BIT);
    if (ret != CSI_OK) {
        printf("csi_iic_set_addr_mode error\n");
        return -1;
    }
    /* config iic standard speed*/
    ret = csi_iic_speed(&slave_iic, IIC_BUS_SPEED_STANDARD);
    if (ret != CSI_OK) {
        printf("csi_iic_set_speed error\n");
        return -1;
    }
    /* config iic slave own address*/
    ret = csi_iic_own_addr(&slave_iic, IIC_OWN_ADDR);
    if (ret != CSI_OK) {
        printf("csi_iic_own_addr error\n");
        return -1;
    }
}
```

```

/* use this interface connect iic rx with dma */
ret = csi_iic_link_dma(&slave_iic, NULL, &dma_ch_rx_handle);
if (ret != CSI_OK) {
    printf("csi_iic_link_dma fail \n");
    return -1;
}
for (i = 0; i < sizeof(write_data); i++) {
    write_data[i] = i;          ///< init write_data value
}

ret = csi_iic_salve_send(&slave_iic, write_data, sizeof(write_data), 100000);
if (ret != CSI_OK) {
    printf("csi_iic_salve_send error\n");
    return -1;
}

ret = csi_iic_salve_receive(&slave_iic, read_data, sizeof(read_data), 100000);
if (ret != CSI_OK) {
    printf("csi_iic_salve_receive error\n");
    return -1;
}
/* use this interface disconnect iic with dma */
csi_iic_link_dma(&slave_iic, NULL, NULL);
/* use this interface disconnect iic callback */
csi_iic_detach_callback(&slave_iic);
csi_iic_uninit(&salve_iic);
return 0;
}

```

中断模式的主机收发使用示例

```

#include "stdio.h"
#include "drv/iic.h"

#define IIC_SLAVE_ADDR    0x50
#define IIC_IDX           0
static csi_iic_t master_iic;
static volatile uint8_t cb_transfer_flag = 0;
static void iic_event_cb_fun(csi_iic_t *iic, csi_iic_event_t event, void *arg)
{
    if (event == IIC_EVENT_RECEIVE_COMPLETE) {
        cb_transfer_flag = 1;
    }

    if (event == IIC_EVENT_SEND_COMPLETE) {
        cb_transfer_flag = 1;
    }
}

```

```
int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    ret = csi_iic_init(&master_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
    /* iic attach callback */
    ret = csi_iic_attach_callback(&master_iic, iic_event_cb_fun, NULL);
    if (ret != CSI_OK) {
        printf("csi_iic_attach_callback error\n");
        return -1;
    }

    /* config iic master mode */
    ret = csi_iic_mode(&master_iic, IIC_MODE_MASTER);
    if (ret != CSI_OK) {
        printf("csi_iic_set_mode error\n");
        return -1;
    }
    /* config iic 7bit address mode */
    ret = csi_iic_addr_mode(&master_iic, IIC_ADDRESS_7BIT);
    if (ret != CSI_OK) {
        printf("csi_iic_set_addr_mode error\n");
        return -1;
    }
    /* config iic standard speed*/
    ret = csi_iic_speed(&master_iic, IIC_BUS_SPEED_STANDARD);
    if (ret != CSI_OK) {
        printf("csi_iic_set_speed error\n");
        return -1;
    }

    for (i = 0; i < sizeof(write_data); i++) {
        write_data[i] = i;          ///< init write_data value
    }

    ret = csi_iic_master_send_async(&master_iic, IIC_SLAVE_ADDR, write_data, sizeof(w
rite_data), 100000);
    if (ret != CSI_OK) {
        printf("csi_iic_master_send error\n");
        return -1;
    }

    ret = csi_iic_master_receive_async(&master_iic, IIC_SLAVE_ADDR, read_data, sizeof
(read_data), 100000);
    if (ret != CSI_OK) {
```

```

        printf("csi_iic_master_receive error\n");
        return -1;
    }
    /* use this interface disconnect iic callback */
    csi_iic_detach_callback(&master_iic);
    csi_iic_uninit(&master_iic);
    return 0;
}

```

中断模式的从机收发使用示例

```

#include "stdio.h"
#include "drv/iic.h"

#define IIC_OWN_ADDR    0x50
#define IIC_IDX         0
static csi_iic_t slave_iic;
static volatile uint8_t slave_cb_transfer_done_flag = 0;
static void slave_iic_event_cb_fun(csi_iic_t *iic, csi_iic_event_t event, void *arg)
{
    if (event == IIC_EVENT_RECEIVE_COMPLETE) {
        slave_cb_transfer_done_flag = 1;
    }

    if (event == IIC_EVENT_SEND_COMPLETE) {
        slave_cb_transfer_done_flag = 1;
    }
}

int example_iic(void)
{
    uint8_t write_data[10];
    uint8_t read_data[10];
    csi_error_t ret;
    uint32_t i;
    ret = csi_iic_init(&slave_iic, IIC_IDX);
    if (ret != CSI_OK) {
        printf("csi_iic_initialize error\n");
        return -1;
    }
    /* iic attach callback */
    ret = csi_iic_attach_callback(&slave_iic, slave_iic_event_cb_fun, NULL);
    if (ret != CSI_OK) {
        printf("csi_iic_attach_callback error\n");
        return -1;
    }
    /* config iic slave mode */
    ret = csi_iic_mode(&slave_iic, IIC_MODE_SLAVE);
}

```

```
if (ret != CSI_OK) {
    printf("csi_iic_set_mode error\n");
    return -1;
}
/* config iic 7bit address mode */
ret = csi_iic_addr_mode(&slave_iic, IIC_ADDRESS_7BIT);
if (ret != CSI_OK) {
    printf("csi_iic_set_addr_mode error\n");
    return -1;
}
/* config iic standard speed*/
ret = csi_iic_speed(&slave_iic, IIC_BUS_SPEED_STANDARD);
if (ret != CSI_OK) {
    printf("csi_iic_set_speed error\n");
    return -1;
}
/* config iic slave own address*/
ret = csi_iic_own_addr(&slave_iic, IIC_OWN_ADDR);
if (ret != CSI_OK) {
    printf("csi_iic_own_addr error\n");
    return -1;
}
for (i = 0; i < sizeof(write_data); i++) {
    write_data[i] = i;          ///< init write_data value
}

ret = csi_iic_salve_send(&slave_iic, write_data, sizeof(write_data), 100000);
if (ret != CSI_OK) {
    printf("csi_iic_salve_send error\n");
    return -1;
}

ret = csi_iic_salve_receive(&slave_iic, read_data, sizeof(read_data), 100000);
if (ret != CSI_OK) {
    printf("csi_iic_salve_receive error\n");
    return -1;
}
/* use this interface disconnect iic callback */
csi_iic_detach_callback(&slave_iic);
csi_iic_uninit(&salve_iic);
return 0;
}
```

SPI设备

设备说明

SPI是串行外设接口(Serial Peripheral Interface)一种同步串行接口技术，是一种高速的，全双工，同步的通信总线。

接口列表

SPI的CSI接口说明如下所示：

函数	说明
csi_spi_init	SPI设备初始化
csi_spi_uninit	SPI设备反初始化
csi_spi_attach_callback	设置回调函数
csi_spi_detach_callback	注销回调函数
csi_spi_mode	设置SPI主/从模式
csi_spi_cp_format	设置SPI相位模式
csi_spi_frame_len	设置SPI数据宽度
csi_spi_baud	设置SPI波特率
csi_spi_send	数据发送(同步模式)
csi_spi_send_async	数据发送(异步模式)
csi_spi_receive	数据接收(同步模式)
csi_spi_receive_async	数据接收(异步模式)
csi_spi_send_receive	数据发送/接收(同步模式)
csi_spi_send_receive_async	数据发送/接收(异步模式)
csi_spi_select_slave	选择指定从机
csi_spi_link_dma	绑定/解除DMA通道
csi_spi_get_state	获取SPI状态

接口详细说明

csi_spi_init

```
csi_error_t csi_spi_init(csi_spi_t *spi, uint32_t idx)
```

- 功能描述:
- 通过设备ID初始化对应的SPI实例，返回结果值。
- 参数:
 - spi : 设备句柄（需要用户申请句柄空间）
 - idx : 设备ID
- 返回值:
 - 错误码csi_error_t

csi_spi_t

成员	类型	说明
dev	csi_dev_t	设备统一句柄
callback	void (callback)(csi_spi_t spi, csi_spi_event_t event, void *arg)	用户回调函数
arg	*void	用户回调函数对应的传参
tx_data	*void	指向发送缓存的地址
tx_size	uint32_t	发送数据的大小
rx_data	*void	指向接收缓存的地址
rx_size	uint32_t	接收数据的大小
send	csi_error_t (csi_spi_send_async_t)(csi_spi_t spi, const void *data, uint32_t size)	指向发送函数(异步)
receive	csi_error_t (csi_spi_receive_async_t)(csi_spi_t spi, void *data, uint32_t size)	指向接收函数(异步)
send_receive	csi_error_t (csi_spi_send_receive_async_t)(csi_spi_t spi, const void *data_out, void *data_in, uint32_t size)	指向发送/接收函数(异步)
state	csi_spi_state_t	运行状态
tx_dma	*csi_dma_ch_t	指向发送DMA通道
rx_dma	*csi_dma_ch_t	指向接收DMA通道
priv	*void	设备私有变量

csi_spi_uninit

```
void csi_spi_uninit(csi_spi_t *spi)
```

- 功能描述:
- SPI实例反初始化, 并且释放相关的软硬件资源。
- 参数:
 - spi : 实例句柄。
- 返回值
 - 无

csi_spi_attach_callback

```
csi_error_t csi_spi_attach_callback(csi_spi_t *spi, void *callback, void *arg)
```

- 功能描述:
- 注册SPI事件回调函数。
- 参数:
 - spi : 实例句柄。
 - callback : 指向事件回调函数。
 - arg : 指向事件回调函数的参数。
- 返回值:
 - 错误码csi_error_t

callback

```
void (*callback)(csi_spi_t *spi, csi_spi_event_t event, void *arg);
```

其中SPI为设备句柄, idx为设备ID, event 为传给回调函数的事件类型, arg为用户自定义的回调函数对应的参数。SPI回调事件枚举类型csi_spi_event_t定义如下:

csi_spi_event_t

类型	说明
SPI_EVENT_TRANSMIT_COMPLETE	数据发送完成事件
SPI_EVENT_RECEIVE_COMPLETE	数据接收完成事件
SPI_EVENT_TRANSMIT_RECEIVE_COMPLETE	数据发送/接收完成事件
SPI_EVENT_ERROR_OVERFLOW	数据溢出事件

SPI_EVENT_ERROR_UNDERFLOW	数据下溢事件
SPI_EVENT_ERROR	总线错误事件

csi_spi_detach_callback

```
void csi_spi_detach_callback(csi_spi_t *spi)
```

- 功能描述:
- 注销SPI事件回调函数。
- 参数:
 - spi : 实例句柄。
- 返回值:
 - 无。

csi_spi_mode

```
csi_error_t csi_spi_mode(csi_spi_t *spi, csi_spi_mode_t mode)
```

- 功能描述:
- 设置SPI主/从模式。
- 参数:
 - spi : 实例句柄。
 - mode : 主/从模式。
- 返回值:
 - 错误码csi_error_t

csi_spi_mode_t

类型	说明
SPI_MASTER	主机模式
SPI_SLAVE	从机模式

csi_spi_cp_format

```
csi_error_t csi_spi_cp_format(csi_spi_t *spi, csi_spi_cp_format_t format)
```

- 功能描述:
- 设置SPI相位模式。
- 参数:
 - spi :实例句柄。
 - format :相位格式。
- 返回值:
 - 错误码csi_error_t

csi_spi_cp_format_t

类型	说明
SPI_FORMAT_CPOL0_CPHA0	CPOL=0 CPHA=0
SPI_FORMAT_CPOL0_CPHA1	CPOL=0 CPHA=1
SPI_FORMAT_CPOL1_CPHA0	CPOL=1 CPHA=0
SPI_FORMAT_CPOL1_CPHA1	CPOL=1 CPHA=1

csi_spi_frame_len

```
csi_error_t csi_spi_frame_len(csi_spi_t *spi, csi_spi_frame_len_t length)
```

- 功能描述:
- 设置SPI数据宽度。
- 参数:
 - spi :实例句柄。
 - length :数据宽度。
- 返回值:
 - 错误码csi_error_t

csi_spi_frame_len_t

类型	说明
SPI_FRAME_LEN_4	数据宽度：4bit
SPI_FRAME_LEN_5	数据宽度：5bit
SPI_FRAME_LEN_6	数据宽度：6bit
SPI_FRAME_LEN_7	数据宽度：7bit

SPI_FRAME_LEN_8	数据宽度：8bit
SPI_FRAME_LEN_9	数据宽度：9bit
SPI_FRAME_LEN_10	数据宽度：10bit
SPI_FRAME_LEN_11	数据宽度：11bit
SPI_FRAME_LEN_12	数据宽度：12bit
SPI_FRAME_LEN_13	数据宽度：13bit
SPI_FRAME_LEN_14	数据宽度：14bit
SPI_FRAME_LEN_15	数据宽度：15bit
SPI_FRAME_LEN_16	数据宽度：16bit

csi_spi_baud

```
uint32_t csi_spi_baud(csi_spi_t *spi, uint32_t baud)
```

- 功能描述:
- 设置SPI波特率。
- 参数:
 - spi :实例句柄。
 - baud :波特率。
- 返回值:
 - 实际频率值，单位HZ。

csi_spi_send

```
int32_t csi_spi_send(csi_spi_t *spi, const void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
- 以阻塞模式进行数据发送，接收数据将被忽略。当发送成功或者错误时，函数返回。
- 参数:
 - spi :实例句柄。
 - data :指向发送数据的缓存。
 - size :发送数据的长度。
 - timeout :发送超时时间，单位ms。
- 返回值

- 发送成功，返回实际发送长度。发送失败，返回错误码。
- 错误码

csi_spi_send_async

```
csi_error_t csi_spi_send_async(csi_spi_t *spi, const void *data, uint32_t size)
```

- 功能描述:
- 以中断或者DMA模式进行数据发送，接收数据将被忽略。当发送成功时，会收到SPI_EVENT_SEND_COMPLETE事件。
- 参数:
 - spi : 实例句柄。
 - data : 指向发送数据的缓存。
 - size : 发送数据的长度。
- 返回值
 - 错误码csi_error_t

csi_spi_receive

```
uint32_t csi_spi_receive(csi_spi_t *spi, void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
- 以阻塞模式进行数据接收。当接收成功或者错误时，函数返回。
- 参数:
 - spi : 实例句柄。
 - data : 指向接收数据的缓存。
 - size : 接收数据的长度。
 - timeout : 发送超时时间，单位ms。
- 返回值
 - 接收成功，返回实际接收长度。接收失败，返回错误码。
- 错误码

csi_spi_receive_async

```
csi_error_t csi_spi_receive_async(csi_spi_t *spi, void *data, uint32_t size)
```

- 功能描述:

- 以中断或者DMA模式进行数据接收。当接收成功时，会收到SPI_EVENT_RECEIVE_COMPLETE事件。
- 参数:
 - spi : 实例句柄。
 - data : 指向接收数据的缓存。
 - size : 需要接收的长度。
- 返回值
 - 错误码csi_error_t

csi_spi_send_receive

```
uint32_t csi_spi_send_receive(csi_spi_t *spi, const void *data_out, void *data_in,
uint32_t size, uint32_t timeout)
```

- 功能描述:
- 以阻塞模式进行数据发送/接收。当传输成功或者错误时，函数返回。
- 参数:
 - spi : 实例句柄。
 - data_out : 指向发送数据的缓存。
 - data_in : 指向接收数据的缓存。
 - size : 需要传输的数据长度。
 - timeout : 传输超时，单位ms。
- 返回值
 - 发送/接收成功，返回实际发送/接收长度。发送/接收失败，返回错误码。
- 错误码
 - 错误码csi_error_t

csi_spi_send_receive_async

```
csi_error_t csi_spi_send_receive_async(csi_spi_t *spi, const void *data_out, void *
data_in, uint32_t size)
```

- 功能描述:
- 以中断或者DMA模式进行数据发送/接收。当传输成功时，会收到SPI_EVENT_SEND_RECEIVE_COMPLETE事件。
- 参数:
 - spi : 实例句柄。

- `data_out` : 指向发送数据的缓存。
 - `data_in` : 指向接收数据的缓存。
 - `size` : 需要传输的数据长度。
- 返回值
 - 错误码 `csi_error_t`

`csi_spi_select_slave`

```
void csi_spi_select_slave(csi_spi_t *spi, int slave_num)
```

- 功能描述:
- 选择指定从机，仅在master模式下有效。
- 参数:
 - `spi` : 实例句柄。
 - `slave_num` : 指定从机号。
- 返回值
 - 无

`csi_spi_link_dma`

```
csi_error_t csi_spi_link_dma(csi_spi_t *spi, csi_dma_ch_t *tx_dma, csi_dma_ch_t *rx_dma)
```

- 功能描述:
- 绑定/解除DMA通道。
- 参数:
 - `spi` : 实例句柄。
 - `tx_dma` : 指向发送dma通道。
 - `rx_dma` : 指向接收dma通道。
- 返回值:
 - 错误码 `csi_error_t`

```
csi_error_t csi_spi_get_state(csi_spi_t *spi, csi_state_t *state)
```

- 功能描述:
- 获取SPI状态。
- 参数:

- spi : 实例句柄。
- state : 指向接收的状态值。
- 返回值:
 - 错误码csi_error_t

同步模式使用示例

示例展示了如何使用同步模式的数据发送，需要使用两个SPI。一个SPI作为MASTER同步发送数据，另一个作为SLAVE异步接收数据。使用前需要检查IO口是否连接。

```
#include <stdio.h>
#include <string.h>

#include <soc.h>
#include <drv/spi.h>
#include <drv/tick.h>
#include <csi_config.h>
#include <board_config.h>
#include <board_init.h>

#define DEFAULT_TRANSFER_TIMEOUT 5000
#define SPI_CHECK_RETURN(ret) \
do { \
    if (ret != CSI_OK) { \
        return -1; \
    } \
} while(0);

csi_spi_event_t spi_master_event, spi_slave_event;
csi_spi_t spi_master, spi_slave;

static void slave_spi_event_callback(csi_spi_t *spi, csi_spi_event_t event, void *arg)
{
    spi_slave_event = event;
}

void example_pin_spi_init(void)
{
    csi_pin_set_mux(EXAMPLE_PIN_SPI_MISO, EXAMPLE_PIN_SPI_MISO_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_MOSI, EXAMPLE_PIN_SPI_MOSI_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_CS, EXAMPLE_PIN_SPI_CS_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SCK, EXAMPLE_PIN_SPI_SCK_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_MISO, EXAMPLE_PIN_SPI_SLAVE_MISO_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_MOSI, EXAMPLE_PIN_SPI_SLAVE_MOSI_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_CS, EXAMPLE_PIN_SPI_SLAVE_CS_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_SCK, EXAMPLE_PIN_SPI_SLAVE_SCK_FUNC);
}
```

```
}

int main(void)
{
    int      ret;
    uint8_t  rx_data[11];
    uint32_t timestart;

    example_pin_spi_init();
    board_init();

    /* Initialize master SPI */
    ret = csi_spi_init(&spi_master, EXAMPLE_SPI_IDX);
    SPI_CHECK_RETURN(ret);

    /* Set master SPI as master mode */
    ret = csi_spi_mode(&spi_master, SPI_MASTER);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_cp_format(&spi_master, SPI_FORMAT_CPOLO_CPHA0);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_frame_len(&spi_master, SPI_FRAME_LEN_8);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_baud(&spi_master, 1000000);
    if(ret == 0){
        return -1;
    }

    /* Select slave 0 of master SPI */
    csi_spi_select_slave(&spi_master, 0);

    /* Initialize slave spi */
    ret = csi_spi_init(&spi_slave, EXAMPLE_SPI_SLAVE_IDX);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_attach_callback(&spi_slave, slave_spi_event_callback, NULL);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_mode(&spi_slave, SPI_SLAVE);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_cp_format(&spi_slave, SPI_FORMAT_CPOL1_CPHA0);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_frame_len(&spi_slave, SPI_FRAME_LEN_8);
    SPI_CHECK_RETURN(ret);

    ret = csi_spi_baud(&spi_slave, 1000000);
    if(ret == 0){
```



```

        return -1;
    }

    /* Slave SPI start receive */
    spi_slave_event = -1;
    ret = csi_spi_receive_async(&spi_slave, rx_data, 11);
    SPI_CHECK_RETURN(ret);

    /* Master SPI send data */
    ret = csi_spi_send(&spi_master, "hello word", 11, 1000);
    printf("send: %s\n", "hello word");
    if (ret != 11) {
        return -1;
    }

    /* Wait transfer complete */
    timestart = csi_tick_get();
    while (spi_slave_event != SPI_EVENT_RECEIVE_COMPLETE) {
        if ((csi_tick_get() - timestart) > 5000) {
            return -1;
        }
    }
    printf("receive: %s\n", rx_data);

    if (0 != strcmp((char *)rx_data, "hello word")) {
        return -1;
    }

    /* Uninit SPI */
    csi_spi_detach_callback(&spi_master);

    csi_spi_detach_callback(&spi_slave);

    csi_spi_uninit(&spi_master);
    csi_spi_uninit(&spi_slave);
    return 0;
}

```

中断模式使用示例

示例展示了如何使用中断模式的数据发送，需要使用两个SPI。一个SPI作为MASTER异步发送数据，另一个作为SLAVE异步接收数据。使用前需要检查IO口是否连接。

使用中断模式需要调用csi_spi_attach_callback(),并使用异步传输接口csi_spi_xxxx_async()

```

#include <stdio.h>
#include <string.h>

#include <soc.h>

```

```
#include <drv/spi.h>
#include <drv/tick.h>
#include <csi_config.h>
#include <board_config.h>
#include <board_init.h>

#define DEFAULT_TRANSFER_TIMEOUT 5000
#define SPI_CHECK_RETURN(ret)      \
do {                               \
    if (ret != CSI_OK) {           \
        return -1;                \
    }                               \
} while(0);

csi_spi_event_t spi_master_event, spi_slave_event;
csi_spi_t spi_master, spi_slave;

static void master_spi_event_callback(csi_spi_t *spi, csi_spi_event_t event, void *
arg)
{
    spi_master_event = event;
}

static void slave_spi_event_callback(csi_spi_t *spi, csi_spi_event_t event, void *a
rg)
{
    spi_slave_event = event;
}

void example_pin_spi_init(void)
{
    csi_pin_set_mux(EXAMPLE_PIN_SPI_MISO,      EXAMPLE_PIN_SPI_MISO_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_MOSI,      EXAMPLE_PIN_SPI_MOSI_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_CS,        EXAMPLE_PIN_SPI_CS_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SCK,       EXAMPLE_PIN_SPI_SCK_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_MISO, EXAMPLE_PIN_SPI_SLAVE_MISO_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_MOSI, EXAMPLE_PIN_SPI_SLAVE_MOSI_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_CS,  EXAMPLE_PIN_SPI_SLAVE_CS_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_SCK, EXAMPLE_PIN_SPI_SLAVE_SCK_FUNC);
}

int main(void)
{
    int      ret;
    uint8_t  rx_data[41];
    uint32_t timestart;

    example_pin_spi_init();
    board_init();
```

```
/* Initialize master SPI */
ret = csi_spi_init(&spi_master, EXAMPLE_SPI_IDX);
SPI_CHECK_RETURN(ret);

/* Master SPI attach callback */
ret = csi_spi_attach_callback(&spi_master, master_spi_event_callback, NULL);
SPI_CHECK_RETURN(ret);

/* Set master SPI as master mode */
ret = csi_spi_mode(&spi_master, SPI_MASTER);
SPI_CHECK_RETURN(ret);

ret = csi_spi_cp_format(&spi_master, SPI_FORMAT_CPOL0_CPHA0);
SPI_CHECK_RETURN(ret);

ret = csi_spi_frame_len(&spi_master, SPI_FRAME_LEN_8);
SPI_CHECK_RETURN(ret);

ret = csi_spi_baud(&spi_master, 100000000);
if(ret == 0){
    return -1;
}

/* Select slave 0 of master SPI */
csi_spi_select_slave(&spi_master, 0);

/* Initialize slave SPI */
ret = csi_spi_init(&spi_slave, EXAMPLE_SPI_SLAVE_IDX);
SPI_CHECK_RETURN(ret);

ret = csi_spi_attach_callback(&spi_slave, slave_spi_event_callback, NULL);
SPI_CHECK_RETURN(ret);

ret = csi_spi_mode(&spi_slave, SPI_SLAVE);
SPI_CHECK_RETURN(ret);

ret = csi_spi_cp_format(&spi_slave, SPI_FORMAT_CPOL1_CPHA0);
SPI_CHECK_RETURN(ret);

ret = csi_spi_frame_len(&spi_slave, SPI_FRAME_LEN_8);
SPI_CHECK_RETURN(ret);

ret = csi_spi_baud(&spi_slave, 100000000);
if(ret == 0){
    return -1;
}

/* Slave SPI start receive */
spi_slave_event = -1;
ret = csi_spi_receive_async(&spi_slave, rx_data, 11);
SPI_CHECK_RETURN(ret);
```

```

/* Master SPI send data */
ret = csi_spi_send_async(&spi_master, "hello word", 11);
printf("send: %s\n", "hello word");
SPI_CHECK_RETURN(ret);

/* Wait transfer complete */
timestart = csi_tick_get();
while (( spi_slave_event != SPI_EVENT_RECEIVE_COMPLETE ) || ( spi_master_event
!= SPI_EVENT_SEND_COMPLETE )) {
    if ((csi_tick_get() - timestart) > 5000) {
        return -1;
    }
}
printf("receive: %s\n", rx_data);

if (0 != strcmp((char *)rx_data, "hello word")) {
    return -1;
}

/* Uninit SPI */
csi_spi_detach_callback(&spi_master);

csi_spi_detach_callback(&spi_slave);

csi_spi_uninit(&spi_master);
csi_spi_uninit(&spi_slave);
return 0;
}

```

DMA模式使用示例

示例展示了如何使用DMA模式的数据发送，需要使用两个SPI。一个SPI作为MASTER DMA发送数据，另一个作为SLAVE中断接收数据。使用前需要检查IO口是否连接。

使用DMA模式需要调用csi_spi_link_dma()

```

#include <stdio.h>
#include <string.h>

#include <soc.h>
#include <drv/spi.h>
#include <drv/tick.h>
#include <csi_config.h>
#include <board_config.h>
#include <board_init.h>

#define DEFAULT_TRANSFER_TIMEOUT 5000

```

```
#define SPI_CHECK_RETURN(ret)          \
do {                                  \
    if (ret != CSI_OK) {              \
        return -1;                    \
    }                                  \
} while(0);

csi_spi_event_t spi_master_event, spi_slave_event;
csi_spi_t spi_master, spi_slave;
csi_dma_ch_t dma_tx_ch;
csi_dma_ch_t dma_rx_ch;

static void master_spi_event_callback(csi_spi_t *spi, csi_spi_event_t event, void *
arg)
{
    spi_master_event = event;
}

static void slave_spi_event_callback(csi_spi_t *spi, csi_spi_event_t event, void *a
rg)
{
    spi_slave_event = event;
}

void example_pin_spi_init(void)
{
    csi_pin_set_mux(EXAMPLE_PIN_SPI_MISO,      EXAMPLE_PIN_SPI_MISO_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_MOSI,      EXAMPLE_PIN_SPI_MOSI_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_CS,        EXAMPLE_PIN_SPI_CS_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SCK,       EXAMPLE_PIN_SPI_SCK_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_MISO, EXAMPLE_PIN_SPI_SLAVE_MISO_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_MOSI, EXAMPLE_PIN_SPI_SLAVE_MOSI_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_CS,  EXAMPLE_PIN_SPI_SLAVE_CS_FUNC);
    csi_pin_set_mux(EXAMPLE_PIN_SPI_SLAVE_SCK, EXAMPLE_PIN_SPI_SLAVE_SCK_FUNC);
}

int main(void)
{
    int      ret;
    uint8_t  rx_data[41];
    uint32_t timestart;

    example_pin_spi_init();
    board_init();

    /* Initialize master SPI */
    ret = csi_spi_init(&spi_master, EXAMPLE_SPI_IDX);
    SPI_CHECK_RETURN(ret);

    /* Master spi attach callback */
```

```
ret = csi_spi_attach_callback(&spi_master, master_spi_event_callback, NULL);
SPI_CHECK_RETURN(ret);

ret = csi_spi_link_dma(&spi_master, &dma_tx_ch, &dma_rx_ch);
SPI_CHECK_RETURN(ret);

/* Set master spi as master mode */
ret = csi_spi_mode(&spi_master, SPI_MASTER);
SPI_CHECK_RETURN(ret);

ret = csi_spi_cp_format(&spi_master, SPI_FORMAT_CPOL0_CPHA0);
SPI_CHECK_RETURN(ret);

ret = csi_spi_frame_len(&spi_master, SPI_FRAME_LEN_8);
SPI_CHECK_RETURN(ret);

ret = csi_spi_baud(&spi_master, 1000000);
if(ret == 0){
    return -1;
}

/* Select salve 0 of master SPI */
csi_spi_select_slave(&spi_master, 0);

/* Initialize slave SPI */
ret = csi_spi_init(&spi_slave, EXAMPLE_SPI_SLAVE_IDX);
SPI_CHECK_RETURN(ret);

ret = csi_spi_attach_callback(&spi_slave, slave_spi_event_callback, NULL);
SPI_CHECK_RETURN(ret);

ret = csi_spi_mode(&spi_slave, SPI_SLAVE);
SPI_CHECK_RETURN(ret);

ret = csi_spi_cp_format(&spi_slave, SPI_FORMAT_CPOL1_CPHA0);
SPI_CHECK_RETURN(ret);

ret = csi_spi_frame_len(&spi_slave, SPI_FRAME_LEN_8);
SPI_CHECK_RETURN(ret);

ret = csi_spi_baud(&spi_slave, 1000000);
if(ret == 0){
    return -1;
}

/* Slave SPI start receive */
spi_slave_event = -1;
ret = csi_spi_receive_async(&spi_slave, rx_data, 11);
SPI_CHECK_RETURN(ret);

/* Master SPI send data */
```

```
ret = csi_spi_send_async(&spi_master, "hello word", 11);
printf("send: %s\n", "hello word");
SPI_CHECK_RETURN(ret);

/* Wait transfer complete */
timestart = csi_tick_get();
while (( spi_slave_event != SPI_EVENT_RECEIVE_COMPLETE ) || ( spi_master_event
!= SPI_EVENT_SEND_COMPLETE )) {
    if ((csi_tick_get() - timestart) > 5000) {
        return -1;
    }
}
printf("receive: %s\n", rx_data);

if (0 != strcmp((char *)rx_data, "hello word")) {
    return -1;
}

/* Uninit SPI */
csi_spi_detach_callback(&spi_master);

csi_spi_detach_callback(&spi_slave);

ret = csi_spi_link_dma(&spi_master, NULL, NULL);
SPI_CHECK_RETURN(ret);

csi_spi_uninit(&spi_master);
csi_spi_uninit(&spi_slave);
return 0;
}
```

SPIFLASH

设备说明

SPI-FLASH 的物理介质是NOR-FLASH，是一种非易失性的内存，适合存放程序指令，程序只读属性数据据，文件系统，用户参数配置数据等。

接口列表

spiflash的CSI接口说明如下所示：

函数	说明
csi_spiflash_spi_init	SPI-FLASH 初始化基于SPI控制器的设备初始化
csi_spiflash_qspi_init	SPI-FLASH 反初始化基于SPI控制器的实例
csi_spiflash_spi_uninit	SPI-FLASH 反初始化设置回调函数
csi_spiflash_qspi_uninit	SPI-FLASH 反初始化基于QSPI控制器的实例
csi_spiflash_get_flash_info	SPI-FLASH 获取设备能力信息
csi_spiflash_read	SPI-FLASH 读操作
csi_spiflash_program	SPI-FLASH 写操作
csi_spiflash_erase	SPI-FLASH 擦除flash
csi_spiflash_read_reg	SPI-FLASH 读状态寄存器
csi_spiflash_write_reg	SPI-FLASH 写状态寄存器
csi_spiflash_lock	SPI-FLASH 安全使能
csi_spiflash_unlock	SPI-FLASH 安全禁止
csi_spiflash_is_locked	SPI-FLASH 安全查询
csi_spiflash_config_data_line	SPI-FLASH 配置总线线数

接口详细说明

csi_spiflash_spi_init

```
csi_error_t csi_spiflash_spi_init(csi_spiflash_t *spiflash, uint32_t spi_idx, void *spi_cs_callback)
```

- 功能描述:

使用SPI控制器初始化对应的SPI-FLASH实例。

- 参数:
 - `spiflash` : 设备句柄 (需要用户申请句柄空间)。
 - `idx` : SPI设备ID。
 - `spi_cs_callback` : 软件片选控制函数。
- 返回值:
 - 错误码`csi_error_t`

csi_spiflash_t

成员	类型	说明
spi_qspi	csi_spi_qspi_t	设备统一句柄
spi_cs_callback	void (*spi_cs_callback)(csi_gpio_pin_state_t value)	软件片选控制函数
flash_prv_info	void	flash设备信息
spi_send	csi_error_t (spi_send)(void spi, uint8_t cmd, uint32_t addr, uint32_t addr_size, const void *data, uint32_t size)	指向发送缓存的地址
spi_receive	csi_error_t (spi_receive)(void spi, uint8_t cmd, uint32_t addr, uint32_t addr_size, void *data, uint32_t size)	发送数据的大小
priv	void*	设备私有变量

csi_spi_callback_t

```
typedef void (*csi_spi_cs_callback_t)(csi_gpio_pin_state_t value);
```

成员	类型	说明
value	csi_gpio_pin_state_t	引脚状态

csi_spiflash_addr_mode_t

类型	说明
FLASH_ADDR_24	24位地址
FLASH_ADDR_32	32位地址

csi_spiflash_lock_region_t

类型	说明
LOCK_TP_NONE	无保护
LOCK_TP_TOP	Upper region
LOCK_TP_BOTTOM	Lower region
LOCK_TP_DUAL	32位地址

csi_spiflash_info_t

成员	类型	说明
flash_name	char	spiflash名称
flash_id	uint32_t	JEDEC ID
flash_size	uint32_t	spiflash大小
xip_addr	uint32_t	XIP地址
sector_size	uint32_t	扇区大小
page_size	uint32_t	页大小
protect_blk_size	uint32_t	保护区域单位
protect_type	csi_spiflash_lock_region_t	保护区域类型

csi_spiflash_qspi_init

```
csi_error_t csi_spiflash_qspi_init(csi_spiflash_t *spiflash, uint32_t qspi_idx)
```

- 功能描述:
- 使用qspi控制器初始化对应的SPI-FLASH实例。
- 参数:
 - spiflash :实例句柄。
 - qspi_idx :设备ID。
- 返回值
 - 错误码csi_error_t

csi_spiflash_spi_uninit

```
void csi_spiflash_spi_uninit(csi_spiflash_t *spiflash)
```

- 功能描述:
- 反初始化基于spi控制器的SPI-FLASH实例, 并且释放相关的软硬件资源。
- 参数:
 - `spiflash` : 实例句柄。
- 返回值:
 - 无

csi_spiflash_qspi_uninit

```
void csi_spiflash_qspi_uninit(csi_spiflash_t *spiflash)
```

- 功能描述:
- 反初始化基于qspi控制器的SPI-FLASH实例, 并且释放相关的软硬件资源。
- 参数:
 - `spiflash` : 实例句柄。
- 返回值:
 - 无

csi_spiflash_get_flash_info

```
csi_error_t csi_spiflash_get_flash_info(csi_spiflash_t *spiflash, csi_spiflash_info_t *flash_info);
```

- 功能描述:
- SPI-FLASH 获取设备能力信息。
- 参数:
 - `spiflash` : 实例句柄。
 - `flash_info` : 保存flash信息的结构体指针。
- 返回值:
 - 错误码`csi_error_t`

csi_spiflash_read

```
int32_t csi_spiflash_read(csi_spiflash_t *spiflash, uint32_t offset, void *data, uint32_t size);
```

- 功能描述:
- 读取SPI-FLASH数据。
- 参数:
 - `spiflash` : 实例句柄。
 - `offset` : flash的偏移地址。
 - `data` : 指向接收数据缓存。
 - `size` : 接收数据的大小。
- 返回值
 - 错误码`csi_error_t`

csi_spiflash_program

```
int32_t csi_spiflash_program(csi_spiflash_t *spiflash, uint32_t offset, const void *data, uint32_t size);
```

- 功能描述:
- SPI-FLASH写数据。
- 参数:
 - `spiflash` : 实例句柄。
 - `offset` : flash的偏移地址。
 - `data` : 指向发送数据的缓存。
 - `size` : 发送数据的大小。
- 返回值
 - 错误码`csi_error_t`

csi_spiflash_erase

```
csi_error_t csi_spiflash_erase(csi_spiflash_t *spiflash, uint32_t offset, uint32_t size);
```

- 功能描述:
- 擦除SPI-FLASH数据。
- 参数:
 - `spiflash` : 实例句柄。
 - `offset` : flash的偏移地址，地址必须为擦除单位大小对齐。
 - `mode` : 擦除长度必须为擦除单位的整数倍。

- 返回值:
 - 错误码csi_error_t

csi_spiflash_read_reg

```
csi_error_t csi_spiflash_read_reg(csi_spiflash_t *spiflash, uint8_t cmd_code, uint8_t *data, uint32_t size);
```

- 功能描述:
- SPI-FLASH 读状态寄存器。
- 参数:
 - spiflash : 实例句柄
 - cmd_code : 读取spi 寄存器的命令字
 - data : 指向接收的数据缓存
 - size : 数据大小
- 返回值:
 - 错误码csi_error_t

csi_spiflash_write_reg

```
csi_error_t csi_spiflash_write_reg(csi_spiflash_t *spiflash, uint8_t cmd_code, uint8_t *data, uint32_t size);
```

- 功能描述:
- SPI-FLASH 写状态寄存器。
- 参数:
 - spiflash : 实例句柄
 - cmd_code : 写spi 寄存器的命令字
 - data : 指向发送的数据缓存
 - size : 数据大小
- 返回值:
 - 错误码csi_error_t

csi_spiflash_lock

```
csi_error_t csi_spiflash_lock(csi_spiflash_t *spiflash, uint32_t offset, uint32_t size);
```

- 功能描述:
- SPI-FLASH 安全使能。
- 参数:
 - `spiflash` : 实例句柄
 - `offset` : flash 保护的开始地址，地址必须protect block size 对齐，请参考csi_spiflash_info_t 查看flash的保护单位size
 - `size` : flash 保护的区域大小，大小为 protect blk size的整数倍
- 返回值:
 - 错误码csi_error_t

csi_spiflash_unlock

```
csi_error_t csi_spiflash_unlock(csi_spiflash_t *spiflash, uint32_t offset, uint32_t size);
```

- 功能描述:
- SPI-FLASH 安全禁止。
- 参数:
 - `spiflash` : 实例句柄
 - `offset` : flash 保护的开始地址，地址必须protect block size 对齐，请参考csi_spiflash_info_t 查看flash的保护单位size。
 - `size` : flash 保护的区域大小，大小为 protect blk size的整数倍
- 返回值:
 - 错误码csi_error_t

csi_spiflash_is_locked

```
int csi_spiflash_is_locked(csi_spiflash_t *spiflash, uint32_t offset, uint32_t size);
```

- 功能描述:
- SPI-FLASH 安全查询。
- 参数:
 - `spiflash` : 实例句柄
 - `offset` : flash 偏移地址。
 - `size` : flash 查询的大小，大小必须为protect_blk_size 的整数倍
- 返回值:

- 错误码csi_error_t

csi_spiflash_config_data_line

```
csi_error_t csi_spiflash_config_data_line(csi_spiflash_t *spiflash, csi_spiflash_data_line_t line);
```

- 功能描述:
- SPI-FLASH 配置总线线数。
- 参数:
 - spiflash :实例句柄
 - line :总线线数
- 返回值:
 - 错误码csi_error_t

csi_spiflash_data_line_t

类型	说明
SPIFLASH_DATA_1_LINE	单线模式
SPIFLASH_DATA_2_LINES	双线模式
SPIFLASH_DATA_4_LINES	四线模式

基于QSPI控制器的SPIFLASH示例

```
使用前需要确保已经增加SPIFLASH信息到vendor列表 ``c

include

include
```

include

include

include

include

include

include

/ example parameters /

define EXAMPLE_FLASH_START_ADDRESS 0x50

define EXAMPLE_FLASH_WRITE_SIZE 0x250

**define CHECK_RETURN(ret) **

```
do {
    if (ret != CSI_OK) {
        return -1;
    }
} while(0);
```

static csi_spiflash_t spiflash;

int main(void) { int ret = 0; csi_spiflash_info_t info; uint8_t tx_data[EXAMPLE_FLASH_WRITE_SIZE];
uint8_t rx_data[EXAMPLE_FLASH_WRITE_SIZE]; uint32_t address, size;

```
board_init();

/* Initialize SPIFLASH based on spi controler */
ret = csi_spiflash_qspi_init(&spiflash, 0);
CHECK_RETURN(ret);

/* Get SPIFLASH information */
ret = csi_spiflash_get_flash_info(&spiflash, &info);
CHECK_RETURN(ret);

/* Chip erase */
ret = csi_spiflash_erase(&spiflash, 0, info.flash_size);
CHECK_RETURN(ret);

/* Prefill tx_data buffer */
memset(tx_data, 0xaa, sizeof(tx_data));

/* Write data to flash */
address = EXAMPLE_FLASH_START_ADDRESS;
```



```

size    = EXAMPLE_FLASH_WRITE_SIZE;
ret = csi_spiflash_program(&spiflash, address, tx_data, size);

if(ret != size){
    return -1;
}

/* Read data from SPIFLASH */
address = EXAMPLE_FLASH_START_ADDRESS;
size    = EXAMPLE_FLASH_WRITE_SIZE;
ret = csi_spiflash_read(&spiflash, address, rx_data, size);

if(ret != size){
    return -1;
}

/* Data compare */
ret = memcmp(tx_data, rx_data, size);
CHECK_RETURN(ret);

csi_spiflash_qspi_uninit(&spiflash);
return 0;
}

```

基于SPI控制器的SPIFLASH示例

使用前需要确保：

- >1. 确保已经增加SPIFLASH信息到vendor列表
- >2. 检查IO连接

```c

```

#include <stdio.h>
#include <string.h>

#include <soc.h>
#include <drv/tick.h>
#include <csi_config.h>
#include <drv/spiflash.h>
#include <board_init.h>
#include <board_config.h>

/* example parameters */
#define EXAMPLE_FLASH_START_ADDRESS 0x50
#define EXAMPLE_FLASH_WRITE_SIZE 0x250

```

```
#define CHECK_RETURN(ret) \
do { \
 if (ret != CSI_OK) { \
 return -1; \
 } \
} while(0);

static csi_gpio_t gpio;
static csi_spiflash_t spiflash;

static void example_pin_init(void)
{
 csi_pin_set_mux(EXAMPLE_PIN_SPI_MISO, EXAMPLE_PIN_SPI_MISO_FUNC);
 csi_pin_set_mux(EXAMPLE_PIN_SPI_MOSI, EXAMPLE_PIN_SPI_MOSI_FUNC);
 csi_pin_set_mux(EXAMPLE_PIN_SPI_CS, EXAMPLE_PIN_SPI_CS_FUNC);
 csi_pin_set_mux(EXAMPLE_PIN_SPI_SCK, EXAMPLE_PIN_SPI_SCK_FUNC);
}

static void soft_cs_pin_init(void)
{
 csi_pin_set_mux(EXAMPLE_PIN_SPI_CS, PIN_FUNC_GPIO);
 csi_gpio_init(&gpio, 0);
 csi_gpio_dir(&gpio, EXAMPLE_PIN_SPI_CS_MSK, GPIO_DIRECTION_OUTPUT);
}

static void ioctl(csi_gpio_pin_state_t value)
{
 csi_gpio_write(&gpio, EXAMPLE_PIN_SPI_CS_MSK, value);
}

int main(void)
{
 int ret = 0;
 csi_spiflash_info_t info;
 uint8_t tx_data[EXAMPLE_FLASH_WRITE_SIZE];
 uint8_t rx_data[EXAMPLE_FLASH_WRITE_SIZE];
 uint32_t address, size;

 board_init();
 example_pin_init();
 soft_cs_pin_init();

 /* Initialize SPIFLASH based on spi controler */
 ret = csi_spiflash_spi_init(&spiflash, 0, ioctl);
 CHECK_RETURN(ret);

 /* Get SPIFLASH information */
 ret = csi_spiflash_get_flash_info(&spiflash, &info);
 CHECK_RETURN(ret);

 /* Chip erase */
```

```
ret = csi_spiflash_erase(&spiflash, 0, info.flash_size);
CHECK_RETURN(ret);

/* Prefill tx_data buffer */
memset(tx_data, 0xaa, sizeof(tx_data));

/* Write data to flash */
address = EXAMPLE_FLASH_START_ADDRESS;
size = EXAMPLE_FLASH_WRITE_SIZE;
ret = csi_spiflash_program(&spiflash, address, tx_data, size);

if(ret != size){
 return -1;
}

/* Read data from SPIFLASH */
address = EXAMPLE_FLASH_START_ADDRESS;
size = EXAMPLE_FLASH_WRITE_SIZE;
ret = csi_spiflash_read(&spiflash, address, rx_data, size);

if(ret != size){
 return -1;
}

/* Data compare */
ret = memcmp(tx_data, rx_data, size);
CHECK_RETURN(ret);

csi_spiflash_spi_uninit(&spiflash);
return 0;
}
```

# ADC设备

## 设备说明

ADC是Analog-to-Digital Converter的缩写。指模/数转换器或者模拟/数字转换器。是指将连续变量的模拟信号转换为离散的数字信号的器件。

ADC的特点

- 支持单次转换、连续转换

## 接口列表

ADC的CSI接口说明如下所示：

| 函数                            | 说明            |
|-------------------------------|---------------|
| csi_adc_init                  | ADC设备初始化      |
| csi_adc_uninit                | ADC设备反初始化     |
| csi_adc_set_buffer            | 设置ADC数据接收缓存   |
| csi_adc_start                 | 启动ADC数据转换(同步) |
| csi_adc_start_async           | 启动ADC数据转换(异步) |
| csi_adc_stop                  | 停止ADC数据转换(同步) |
| csi_adc_stop_async            | 停止ADC数据转换(异步) |
| csi_adc_channel_enable        | 开启或者关闭一个ADC通道 |
| csi_adc_channel_sampling_time | 设置通道采样时间      |
| csi_adc_sampling_time         | 设置ADC采样时间     |
| csi_adc_continue_mode         | 开启/关闭连续采样模式   |
| csi_adc_freq_div              | 设置ADC分频系数     |
| csi_adc_read                  | 读取ADC转换结果     |
| csi_adc_get_state             | 获取ADC状态       |
| csi_adc_attach_callback       | 注册回调函数        |
| csi_adc_detach_callback       | 注销回调函数        |
| csi_adc_link_dma              | 绑定/解除DMA通道    |

## 接口详细说明

## csi\_adc\_init

```
csi_error_t csi_adc_init(csi_adc_t *adc, uint32_t idx)
```

- 功能描述:
- 通过设备ID初始化对应的ADC实例，返回结果值。
- 参数:
  - `adc` : 设备句柄（需要用户申请句柄空间）
  - `idx` : 设备ID
- 返回值:
  - 错误码 `csi_error_t`

## csi\_adc\_t

| 成员       | 类型                                                               | 说明              |
|----------|------------------------------------------------------------------|-----------------|
| dev      | csi_dev_t                                                        | 设备统一句柄          |
| callback | void (callback)(csi_adc_t adc, csi_adc_event_t event, void *arg) | 用户回调函数          |
| arg      | *void                                                            | 用户回调函数对应的传参     |
| data     | *uint32_t                                                        | 指向接收数据缓存的地址     |
| size     | uint32_t                                                         | 接收数据的大小         |
| start    | csi_error_t (start)(csi_adc_t adc)                               | 指向ADC开始转换函数(异步) |
| stop     | csi_error_t (stop)(csi_adc_t adc)                                | 指向ADC停止转换函数(异步) |
| state    | csi_state_t                                                      | 运行状态            |
| dma      | *csi_dma_ch_t                                                    | 指向接收DMA通道       |
| priv     | *void                                                            | 设备私有变量          |

## csi\_adc\_uninit

```
void csi_adc_uninit(csi_adc_t *adc)
```

- 功能描述:
- ADC实例反初始化, 并且释放相关的软硬件资源。
- 参数:

- `adc` : 实例句柄。
- 返回值
  - 无

## **csi\_adc\_set\_buffer**

```
csi_error_t csi_adc_set_buffer(csi_adc_t *adc, uint32_t *data, uint32_t num)
```

- 功能描述:
- 设置ADC数据接收缓存。
- 参数:
  - `adc` : 实例句柄。
  - `data` : 指向数据接收缓存。
  - `num` : 接收缓存的大小。
- 返回值:
  - 错误码`csi_error_t`

## **csi\_adc\_start**

```
csi_error_t csi_adc_start(csi_adc_t *adc)
```

- 功能描述:
- 启动ADC数据转换(同步)。
- 参数:
  - `adc` : 实例句柄。
- 返回值:
  - 错误码`csi_error_t`

## **csi\_adc\_start\_async**

```
csi_error_t csi_adc_start_async(csi_adc_t *adc)
```

- 功能描述:
- 启动ADC数据转换(异步)。
- 参数:
  - `adc` : 实例句柄。

- 返回值:
  - 错误码csi\_error\_t

## csi\_adc\_stop

```
csi_error_t csi_adc_stop(csi_adc_t *adc)
```

- 功能描述:
- 停止ADC数据转换(同步)。
- 参数:
  - `adc` : 实例句柄。
- 返回值:
  - 错误码csi\_error\_t

## csi\_adc\_stop\_async

```
csi_error_t csi_adc_stop_async(csi_adc_t *adc)
```

- 功能描述:
- 停止ADC数据转换(异步)。
- 参数:
  - `adc` : 实例句柄。
- 返回值:
  - 错误码csi\_error\_t

## csi\_adc\_channel\_enable

```
csi_error_t csi_adc_channel_enable(csi_adc_t *adc, uint8_t ch_id, bool is_enable)
```

- 功能描述:
- 开启或者关闭一个ADC通道。
- 参数:
  - `adc` : 实例句柄。
  - `ch_id` : ADC通道ID。
  - `is_enable` : true->使能, false->禁止。
- 返回值:

- 错误码csi\_error\_t

## csi\_adc\_channel\_sampling\_time

```
csi_error_t csi_adc_channel_sampling_time(csi_adc_t *adc, uint8_t ch_id, uint16_t clock_num)
```

- 功能描述:
- 设置ADC指定通道采样时间，单位：时钟周期个数。
- 参数:
  - adc :实例句柄。
  - ch\_id : ADC通道ID。
  - clock\_num : 时钟周期个数。
- 返回值:
  - 错误码csi\_error\_t

## csi\_adc\_sampling\_time

```
csi_error_t csi_adc_sampling_time(csi_adc_t *adc, uint16_t clock_num)
```

- 功能描述:
- 设置ADC控制器采样时间，单位：时钟周期个数。
- 参数:
  - adc :实例句柄。
  - clock\_num : 时钟周期个数。
- 返回值:
  - 错误码csi\_error\_t

## csi\_adc\_continue\_mode

```
csi_error_t csi_adc_continue_mode(csi_adc_t *adc, bool is_enable)
```

- 功能描述:
- 开启/关闭ADC连续采样模式。
- 参数:
  - adc :实例句柄。
  - is\_enable : true->使能，false->禁止。



- 返回值:
  - 错误码csi\_error\_t

## csi\_adc\_freq\_div

```
uint32_t csi_adc_freq_div(csi_adc_t *adc, uint32_t div)
```

- 功能描述:
- 设置ADC分频系数。
- 参数:
  - `adc` : 实例句柄。
  - `div` : 分频系数。
- 返回值:
  - 实际频率值, 单位HZ。

## csi\_adc\_read

```
int32_t csi_adc_read(csi_adc_t *adc)
```

- 功能描述:
- 读取ADC转换结果。
- 参数:
  - `adc` : 实例句柄。
- 返回值
  - 错误码csi\_error\_t

## csi\_adc\_get\_state

```
csi_error_t csi_adc_get_state(csi_adc_t *adc, csi_state_t *state)
```

- 功能描述:
- 获取ADC状态。
- 参数:
  - `adc` : 实例句柄。
  - `state` : 指向接收的状态值。
- 返回值:

- 错误码csi\_error\_t

csi\_state\_t

| 类型       | 说明   |
|----------|------|
| readable | 设备可读 |
| writable | 设备可写 |
| error    | 错误状态 |

csi\_adc\_attach\_callback

```
csi_error_t csi_adc_attach_callback(csi_adc_t *adc, void *callback, void *arg)
```

- 功能描述:
- 注册ADC事件回调函数。
- 参数:
  - adc :实例句柄。
  - callback :指向事件回调函数。
  - arg :指向事件回调函数的参数。
- 返回值:
  - 错误码csi\_error\_t

callback

```
typedef void (*csi_adc_cb_t)(csi_adc_t *adc, csi_adc_event_t event, void *arg)
```

其中ADC为设备句柄，idx为设备ID，event 为传给回调函数的事件类型，arg为用户自定义的回调函数对应的参数。 ADC回调事件枚举类型csi\_adc\_event\_t定义如下：

csi\_adc\_event\_t

| 类型                          | 说明                   |
|-----------------------------|----------------------|
| ADC_EVENT_CONVERT_COMPLETE  | 转换完成事件，当指定的数据转换完成时触发 |
| ADC_EVENT_CONVERT_HALF_DONE | 转换完成一半，当指定的数据转换完成时触发 |
| ADC_EVENT_ERROR             | 数据丢失事件，未及时读取数据时触发    |

csi\_adc\_detach\_callback

```
void csi_adc_detach_callback(csi_adc_t *adc)
```

- 功能描述:
- 解除ADC事件回调函数。
- 参数:
  - `adc` : 实例句柄。
- 返回值
  - 无

## `csi_adc_link_dma`

```
csi_error_t csi_adc_link_dma(csi_adc_t *adc, csi_dma_ch_t *dma);
```

- 功能描述:
- 绑定/解除DMA通道。
- 参数:
  - `adc` : 实例句柄。
  - `dma` : 指向dma通道。
- 返回值:
  - 错误码`csi_error_t`

## 同步模式使用示例

ADC可采用单次采样模式或者连续采样模式。默认使用单次转换模式，通过调用`csi_adc_continue_mode(true)`后，可将ADC切换到连续采样模式 单次采样模式示例如下: ``c

```
include
```

```
include
```

```
include
```

```
include
```

```
include
```

```
include
```

## define ADC\_CHECK\_RETURN(ret) \

```
do { \ if (ret != CSI_OK) { \ return -1; \ } \ } while(0);
```

```
static csi_adc_t adc;
```

```
int main() { int ret; uint32_t data;
```

```
board_init();

/* GPIO init */
csi_pin_set_mux(EXAMPLE_ADC_CHANNEL0_PIN, EXAMPLE_ADC_CHANNEL0_PIN_FUNC);

/* Initialize ADC peripheral */
ret = csi_adc_init(&adc, 0);
ADC_CHECK_RETURN(ret);

/* Configure frequency division, this value can be one of(4 8 16 32 64 128) */
ret = csi_adc_freq_div(&adc, 128);
if(ret == 0){
 return -1;
}

/* Configure sampling time */
ret = csi_adc_sampling_time(&adc, 2);
ADC_CHECK_RETURN(ret);

/* Enable channel */
ret = csi_adc_channel_enable(&adc, 0, true);
ADC_CHECK_RETURN(ret);

/* Trigger new conversion */
ret = csi_adc_start(&adc);
ADC_CHECK_RETURN(ret);

/* Read result */
data = csi_adc_read(&adc);
printf("get adc result: %d\n", data);

/* Uninit adc */
csi_adc_uninit(&adc);
return 0;
```

```
}
```

```
> 连续采样模式
```

```
```\n#include <stdio.h>
```

```
#include <string.h>

#include <drv/adc.h>
#include <drv/tick.h>
#include <board_config.h>
#include <board_init.h>

#define ADC_CHECK_RETURN(ret)      \
do {                               \
    if (ret != CSI_OK) {           \
        return -1;                \
    }                               \
} while(0);

static csi_adc_t adc;

int main()
{
    int      ret;
    uint32_t data;
    uint8_t  i = 0;

    board_init();

    /* GPIO init */
    csi_pin_set_mux(EXAMPLE_ADC_CHANNEL0_PIN, EXAMPLE_ADC_CHANNEL0_PIN_FUNC);

    /* Initialize ADC peripheral */
    ret = csi_adc_init(&adc, 0);
    ADC_CHECK_RETURN(ret);

    /* Configure frequency division, this value can be one of(4 8 16 32 64 128) */
    ret = csi_adc_freq_div(&adc, 128);
    if(ret == 0){
        return -1;
    }

    /* Configure sampling time */
    ret = csi_adc_sampling_time(&adc, 2);
    ADC_CHECK_RETURN(ret);

    /* Enable continue mode */
    ret = csi_adc_continue_mode(&adc, true);
    ADC_CHECK_RETURN(ret);

    /* Enable channel */
    ret = csi_adc_channel_enable(&adc, 0, true);
    ADC_CHECK_RETURN(ret);

    /* Trigger new conversion */
    ret = csi_adc_start(&adc);
```

```

ADC_CHECK_RETURN(ret);

/* Read result */
for(i = 0; i < 10; i++){
    data = csi_adc_read(&adc);
    printf("get adc result: %d\n", data);
}

/* Uninit ADC */
csi_adc_uninit(&adc);
return 0;
}

```

异步模式使用示例

DMA模式

使用异步模式时，推荐使用DMA模式。中断模式不建议使用，频繁中断会造成大量的CPU消耗，同时有可能导致中断响应跟不上ADC采样频率，导致采样结果丢失。

1. 需要调用csi_adc_attach_callback()绑定回调，否则无法使用异步模式。
2. 需要调用csi_adc_set_buffer()设置数据缓存，否则无法使用异步模式。
3. 需要调用csi_adc_continue()开启连续采样模式，否则可能无法达到预期的size。

```

#include <stdio.h>
#include <string.h>

#include <drv/adc.h>
#include <drv/tick.h>
#include <board_config.h>
#include <board_init.h>

#define ADC_CHECK_RETURN(ret) \
do { \
    if (ret != CSI_OK) { \
        return -1; \
    } \
} while(0);

static csi_adc_t adc;
static csi_dma_ch_t dma_ch;
static int adc_event = -1;

static void adc_event_callback(csi_adc_t *adc, csi_adc_event_t event, void *arg)
{
    adc_event = event;
}

```

```
static void dump_adc_data(char *prefix, uint32_t *data, uint16_t size)
{
    printf("===== %s =====\n", prefix);

    while (size--) {
        printf("%d, ", *data++);
    }

    printf("\n===== \n");
}

int main()
{
    int      ret;
    uint32_t rx_data[10];
    uint32_t timestart;

    board_init();

    /* GPIO init */
    csi_pin_set_mux(EXAMPLE_ADC_CHANNEL0_PIN, EXAMPLE_ADC_CHANNEL0_PIN_FUNC);

    /* Initialize ADC peripheral */
    ret = csi_adc_init(&adc, 0);
    ADC_CHECK_RETURN(ret);

    /* Attach user callback */
    ret = csi_adc_attach_callback(&adc, adc_event_callback, NULL);
    ADC_CHECK_RETURN(ret);

    /* Enable dma mode */
    ret = csi_adc_link_dma(&adc, &dma_ch);
    ADC_CHECK_RETURN(ret);

    /* Set continue mode */
    csi_adc_continue_mode(&adc, true);

    /* Set data buffer */
    ret = csi_adc_set_buffer(&adc, rx_data, 10);
    ADC_CHECK_RETURN(ret);

    /* Configure frequency division, this value can be one of(4 8 16 32 64 128) */
    ret = csi_adc_freq_div(&adc, 128);
    if(ret == 0){
        return -1;
    }

    /* Configure sampling time */
    ret = csi_adc_sampling_time(&adc, 2);
    ADC_CHECK_RETURN(ret);
}
```

```
/* Enable channel */
ret = csi_adc_channel_enable(&adc, 0, true);
ADC_CHECK_RETURN(ret);

ret = csi_adc_channel_enable(&adc, 1, true);
ADC_CHECK_RETURN(ret);

/* Trigger new conversion */
adc_event = -1;
ret = csi_adc_start_async(&adc);
ADC_CHECK_RETURN(ret);

/* Read result */
timestart = csi_tick_get_ms();
while (adc_event != ADC_EVENT_CONVERSION_COMPLETE) {
    if ((csi_tick_get_ms() - timestart) > 5000) {
        return -1;
    }
}

dump_adc_data("rx_data", rx_data, 10);

/* Stop conversion */
ret = csi_adc_stop_async(&adc);
ADC_CHECK_RETURN(ret);

ret = csi_adc_link_dma(&adc, NULL);
ADC_CHECK_RETURN(ret);

csi_adc_detach_callback(&adc);

/* Uninit ADC */
csi_adc_uninit(&adc);
return 0;
}
```


DMA

函数列表

简要说明

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

接口描述

csi_dma_ch_t

成员	类型	说明
parent	void *	存放使用DMA的外设句柄，如uart，iic等
ctrl_id	int8_t	DMA控制器id
ch_id	int8_t	通道句柄
etb_ch_id	int16_t	ETB通道id
cb	void *	通道回调函数
arg	void *	回调函数的参数指针
next	struct csi_dma_ch *	链表的下一个节点

csi_dma_ctrl_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
ch_list	csi_dma_ch_t *	通道链表首地址
alloc_status	uint32_t	通道占用状态
ch_num	uint32_t	控制器拥有的通道数量
priv	void	对接接口句柄地址

csi_dma_cb_t

```
typedef void (*csi_dma_cb_t)(csi_dma_ch_t *dma, csi_dma_event_t event, void *arg);
```

其中dma为通道句柄，event 为传给回调函数的事件类型，arg为用户自定义的回调函数对应的参数。
dma 回调事件枚举类型csi_dma_event_t定义如下：

事件类型	事件说明
DMA_EVENT_TRANSFER_DONE	数据传输完成事件
DMA_EVENT_TRANSFER_HALF_DONE	数据传输完成一半事件
DMA_EVENT_TRANSFER_ERROR	传输异常事件

csi_dma_ctrl_init

```
csi_error_t csi_dma_ctrl_init(csi_dma_ctrl_t *dma, int8_t ctrl_id)
```

- 功能描述:
 - 初始化DMA控制器
- 参数:
 - dma : DMA控制器句柄
 - ctrl_id : 控制器id
- 返回值:
 - 错误码。

csi_dma_ctrl_uninit

```
void csi_dma_ctrl_uninit(csi_dma_ctrl_t *dma)
```

- 功能描述:
 - DMA控制器反初始化
- 参数:
 - dma : DMA控制器句柄
- 返回值:
 - 无。

csi_dma_ch_alloc

```
csi_error_t csi_dma_ch_alloc(csi_dma_ch_t *dma, int8_t ch_id, int8_t ctrl_id)
```

- 功能描述:

- 申请dma通道号。
- 参数:
 - dma :DMA通道句柄
 - ch_id :DMA通道ID，当传入-1时，将由驱动自行选择
 - ctrl_id :DMA控制器ID，当传入-1时，将由驱动自行选择
- 返回值:
 - 错误码。

csi_dma_ch_free

```
void csi_dma_ch_free(csi_dma_ch_t *dma)
```

- 功能描述:
 - 释放DMA特定通道占用。
- 参数:
 - dma :DMA通道句柄。

csi_dma_addr_inc_t

类型	说明
DMA_ADDR_INC	地址递增
DMA_ADDR_DEC	地址递减
DMA_ADDR_CONSTANT	地址固定

csi_dma_data_width_t

类型	说明
DMA_DATA_WIDTH_8_BITS	8位数据宽度
DMA_DATA_WIDTH_16_BITS	16位数据宽度
DMA_DATA_WIDTH_32_BITS	32位数据宽度

csi_dma_endian_t (仅WJ使用)

类型	说明
DMA_ENDIAN_LITTLE	小端数据传输

DMA_ENDIAN_BIG	大端数据传输
----------------	--------

csi_dma_trig_t (仅WJ使用)

类型	说明
DMA_SINGLE_TRIGGER	单次触发
DMA_GROUP_TRIGGER	组触发
DMA_BLOCK_TRIGGER	块触发

csi_dma_req_mode_t (仅WJ使用)

类型	说明
DMA_MODE_HARDWARE	硬件模式
DMA_MODE_SOFTWARE	软件模式

csi_dma_single_dir_t (仅WJ使用)

类型	说明
DMA_SINGLE_DIR_DST	单次模式中方向为目的
DMA_SINGLE_DIR_SRC	软件模式中方向为源

wj_dma_ch_config (仅WJ使用)

成员	类型	说明
group_len	uint8_t	传输组长度
half_int_en	uint8_t	半完成中断使能
secure_en	uint8_t	安全模式使能
src_endian	csi_dma_endian_t	源大小端配置
dst_endian	csi_dma_endian_t	目的大小端配置
trig_mode	csi_dma_trig_t	触发模式配置
req_mode	csi_dma_req_mode_t	请求模式配置
single_dir	csi_dma_single_dir_t	单次方向配置
etb	csi etb config t	ETB配置 (定义请参阅ETB接口文档)

csi_dma_trans_dir_t (仅DW使用)

类型	说明
DMA_MEM2MEM	内存到内存
DMA_MEM2PERH	内存到外设
DMA_PERH2MEM	外设到内存
DMA_PERH2PERH	外设到外设

dw_dma_ch_config (仅DW使用)

成员	类型	说明
hs_if	uint8_t	硬件握手号
src_reload_en	uint8_t	源地址自动加载使能
dest_reload_en	uint8_t	目的地址自动加载使能
trans_dir	csi_dma_trans_dir_t	传输数据方向配置

csi_dma_ch_config_t

成员	类型	说明
src_inc	csi_dma_addr_inc_t	DMA源地址增长方式
dst_inc	csi_dma_addr_inc_t	DMA目的地址增长方式
src_tw	csi_dma_data_width_t	DMA源数据宽度
dst_tw	csi_dma_data_width_t	DMA目的数据宽度
wj	wj_dma_ch_config	wj DMA配置结构体
dw	dw_dma_ch_config	dw DMA配置结构体

csi_dma_ch_config

```
csi_error_t csi_dma_ch_config(csi_dma_ch_t *dma, csi_dma_ch_config_t *config, csi_dma_cb_t cb, void *arg);
```

- 功能描述:

- 配置DMA通道工作方式，并配置回调函数。
 - 参数:
 - `dma` : DMA通道句柄。
 - `config` : DMA通道配置结构体
 - `cb` : DMA用户回调函数
 - `arg` : DMA回调函数参数指针
 - 返回值:
 - 错误码。
-

csi_dma_ch_start

```
void csi_dma_ch_start(csi_dma_ch_t *dma, void *srcaddr, void *dstaddr, uint32_t length)
```

- 功能描述:
 - 使能DMA通道，开始工作。
 - 参数:
 - `dma` : DMA通道句柄。
 - `srcaddr` : 传输的源地址
 - `dstaddr` : 传输的目的地址
 - `length` : 数据的长度（字节为单位）
 - 返回值:
 - 无。
-

csi_dma_ch_stop

```
void csi_dma_ch_stop(csi_dma_ch_t *dma)
```

- 功能描述:
 - 关闭DMA通道，结束工作。
- 参数:
 - `dma` : DMA通道句柄

PWM

说明

脉冲宽度调制（Pulse width modulation，简称PWM）是一种强大的模拟信号数字编码技术，它利用高分辨率计数器产生方波，并通过调制方波占空比对模拟信号进行编码。典型应用包括开关电源和电机控制。

接口列表

PWM的CSI接口如下所示：

函数	说明
csi_pwm_init	初始化
csi_pwm_uninit	反初始化
csi_pwm_out_config	配置输出模式
csi_pwm_out_start	开始输出
csi_pwm_out_stop	停止输出
csi_pwm_capture_config	配置捕获模式
csi_pwm_capture_start	开始捕获
csi_pwm_capture_stop	停止捕获
csi_pwm_attach_callback	注册回调
csi_pwm_detach_callback	注销回调

接口详细说明

csi_pwm_init

```
csi_error_t csi_pwm_init(csi_pwm_t *pwm, uint32_t idx)
```

- 功能说明：
 - 通过设备号初始化对于的PWM实例。
- 参数：
 - `pwm`：设备句柄（需要用户申请句柄空间）。
 - `idx`：设备号。

- 返回值：
 - 错误码csi_error_t。

csi_pwm_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
callback	void (callback)(csi_pwm_t pwm, csi_pwm_event_t event, uint32_t ch, uint32_t time_us, void *arg)	用户回调函数
arg	void *	回调函数参数（用户自定义）
priv	void *	设备私有变量

csi_pwm_uninit

```
void csi_pwm_uninit(csi_pwm_t *pwm)
```

- 功能说明：
 - PWM实例反初始化，该接口会停止PWM实例正在进行的工作，并且释放相关的软硬件资源。
- 参数：
 - pwm：设备句柄。
- 返回值：
 - 无。

csi_pwm_out_config

```
csi_error_t csi_pwm_out_config(csi_pwm_t *pwm,  
                               uint32_t channel,  
                               uint32_t period_us,  
                               uint32_t pulse_width_us,  
                               csi_pwm_polarity_t polarity)
```

- 功能说明：
 - 配置PWM信号输出模式。
- 参数：
 - pwm：设备句柄。

- `channel` : 通道号 (范围0~5)。
- `period_us` : 信号周期时间 (单位us)。
- `pulse_width_us` : 有效电平时间 (单位us)。
- `polarity` : 有效电平极性。
- 返回值 :
 - 错误码csi_error_t。

csi_pwm_polarity_t

类型	说明
PWM_POLARITY_HIGH	高电平
PWM_POLARITY_LOW	低电平

csi_pwm_out_start

```
csi_error_t csi_pwm_out_start(csi_pwm_t *pwm, uint32_t channel)
```

- 功能说明 :
 - PWM信号开始输出。
- 参数 :
 - `pwm` : 设备句柄。
 - `channel` : 通道号 (范围0~5)。
- 返回值 :
 - 错误码csi_error_t。

csi_pwm_out_stop

```
void csi_pwm_out_stop(csi_pwm_t *pwm, uint32_t channel)
```

- 功能说明 :
 - PWM信号停止输出。
- 参数 :
 - `pwm` : 设备句柄。

- `channel` : 通道号 (范围0~5)。
- 返回值 :
 - 错误码`csi_error_t`。

csi_pwm_capture_config

```
csi_error_t csi_pwm_capture_config(csi_pwm_t *pwm,
                                   uint32_t channel,
                                   csi_pwm_capture_polarity_t polarity,
                                   uint32_t count)
```

- 功能说明 :
 - 配置PWM输入捕获模式。
- 参数 :
 - `pwm` : 设备句柄。
 - `channel` : 通道号 (范围0~5)。
 - `polarity` : 配置捕获方式 (上升沿/下降沿/边沿)。
 - `count` : 配置捕获次数 (捕获达到count次以后产生中断)。
- 返回值 :
 - 错误码`csi_error_t`。

csi_pwm_capture_polarity_t

类型	说明
PWM_CAPTURE_POLARITY_POSEDGE	上升沿捕获类型
PWM_CAPTURE_POLARITY_NEGEDGE	下降沿捕获类型
PWM_CAPTURE_POLARITY_BOTHEDGE	边沿捕获类型

csi_pwm_capture_start

```
csi_error_t csi_pwm_capture_start(csi_pwm_t *pwm, uint32_t channel)
```

- 功能说明 :
 - PWM开始捕获。

- 参数：
 - `pwm` : 设备句柄。
 - `channel` : 通道号 (范围0~5)。
 - 返回值：
 - 错误码`csi_error_t`。
-

csi_pwm_capture_stop

```
void csi_pwm_capture_stop(csi_pwm_t *pwm, uint32_t channel)
```

- 功能说明：
 - PWM停止捕获。
 - 参数：
 - `pwm` : 设备句柄。
 - `channel` : 通道号 (范围0~5)。
 - 返回值：
 - 无。
-

csi_pwm_attach_callback

```
csi_error_t csi_pwm_attach_callback(csi_pwm_t *pwm, void *callback, void *arg)
```

- 功能说明：
 - 注册中断回调函数。
- 参数：
 - `pwm` : 设备句柄。
 - `callback` : 中断回调函数。
 - `arg` : 回调函数参数 (可选 , 由用户定义)。
- 返回值：
 - 错误码`csi_error_t`。

callback

```
void (*callback)(csi_pwm_t *pwm, csi_pwm_event_t event, uint32_t ch, uint32_t time_us, void *arg);
```

其中pwm为设备句柄，event 为传给回调函数的事件类型，ch为pwm中断响应通道号，time_us为事件触发间隔事件，arg为用户自定义的回调函数对应的参数。pwm回调事件枚举类型csi_pwm_event_t定义如下：

事件类型	事件说明
PWM_EVENT_CAPTURE_POSEDGE	上升沿事件类型
PWM_EVENT_CAPTURE_NEGEDGE	下降沿事件类型
PWM_EVENT_CAPTURE_BOTHEDGE	边沿事件类型
PWM_EVENT_ERROR	错误事件类型

csi_pwm_detach_callback

```
void csi_pwm_detach_callback(csi_pwm_t *pwm)
```

- 功能说明：
 - 注销中断回调函数。
- 参数：
 - pwm：设备句柄。
- 返回值：
 - 无。

初始化示例

```
#include <stdio.h>
#include <soc.h>
#include <drv/pwm.h>

static csi_pwm_t g_pwm;

int main(void)
{
    csi_error_t ret = 0;

    ret = csi_pwm_init(&g_pwm, 0);
```

```

    return ret;
}

```

信号输出模式

```

#include <stdio.h>
#include <soc.h>
#include <drv/pwm.h>
#include <drv/pin.h>

#define EXAMPLE_PWM_IDX                0U
/**
 * PWM has 6 channels in total
 */
#define EXAMPLE_PWM_CH_IDX            0U

#define EXAMPLE_PWM_CH                PA0
#define EXAMPLE_PWM_CH_FUNC          PA0_PWM_CH0

#define CHECK_RETURN(ret)              \
do {                                  \
    if (ret != 0) {                    \
        return -1;                     \
    }                                  \
} while(0);

static csi_pwm_t g_pwm;

int main(void)
{
    csi_error_t ret = 0;

    /* STEP 1: pwm port */
    csi_pin_set_mux(EXAMPLE_PWM_CH, EXAMPLE_PWM_CH_FUNC);

    /* STEP 2: pwm init */
    ret = csi_pwm_init(&g_pwm, EXAMPLE_PWM_IDX);
    CHECK_RETURN(ret);

    /**
     * STEP 3: pwm channel 0 configure operating mode and signal period
     * period time: 30us
     * low level time: 10us
     */
    ret = csi_pwm_out_config(&g_pwm, EXAMPLE_PWM_CH_IDX, 30, 10, PWM_POLARITY_LOW);
    CHECK_RETURN(ret);

    /* STEP 4: pwm channel 0 start output */

```

```

    ret = csi_pwm_out_start(&g_pwm, EXAMPLE_PWM_CH_IDX);
    CHECK_RETURN(ret);

    mdelay(10000);

    /* STEP 5: pwm channel 0 stop output */
    csi_pwm_out_stop(&g_pwm, EXAMPLE_PWM_CH_IDX);

    /* STEP 6: pwm uninit */
    csi_pwm_uninit(&g_pwm);

    return ret;
}

```

输入捕获模式

```

#include <stdio.h>
#include <soc.h>
#include <drv/pwm.h>
#include <drv/pin.h>

#define EXAMPLE_PWM_IDX                0U

#define EXAMPLE_PWM_CH_IDX            1U
#define EXAMPLE_PWM_CAPTURE_CH_IDX    0U

#define EXAMPLE_PWM_CH                PA2
#define EXAMPLE_PWM_CH_FUNC           PA2_PWM_CH2
#define EXAMPLE_PWM_CAPTURE_CH        PA0
#define EXAMPLE_PWM_CAPTURE_CH_FUNC   PA0_PWM_CH0

#define CHECK_RETURN(ret)              \
do {                                   \
    if (ret != 0) {                     \
        return -1;                      \
    }                                    \
} while(0);

static csi_pwm_t g_pwm;
static uint32_t cb_pwm_time = 0;
static csi_pwm_event_t cb_pwm_flag = PWM_EVENT_ERROR;

static void pwm_event_cb_fun(csi_pwm_t *pwm_handle, csi_pwm_event_t event, uint8_t
ch, uint32_t time_us, void *arg)
{
    switch (event) {
        case PWM_EVENT_CAPTURE_POSEDGE:

```

```

        cb_pwm_flag = PWM_EVENT_CAPTURE_POSEDGE;
        cb_pwm_time = time_us;
        break;
    case PWM_EVENT_CAPTURE_NEGEDGE:
        cb_pwm_flag = PWM_EVENT_CAPTURE_NEGEDGE;
        cb_pwm_time = time_us;
        break;
    case PWM_EVENT_CAPTURE_BOTHEDGE:
        cb_pwm_flag = PWM_EVENT_CAPTURE_BOTHEDGE;
        cb_pwm_time = time_us;
        break;
    default:
        break;
}
}

int main(void)
{
    csi_error_t ret = 0;

    /**
     * STEP 1: pwm port mux
     * enable channel 0/1
     */
    csi_pin_set_mux(EXAMPLE_PWM_CH, EXAMPLE_PWM_CH_FUNC);
    csi_pin_set_mux(EXAMPLE_PWM_CAPTURE_CH, EXAMPLE_PWM_CAPTURE_CH_FUNC);

    /* STEP 2: pwm init */
    ret = csi_pwm_init(&g_pwm, EXAMPLE_PWM_IDX);
    CHECK_RETURN(ret);

    /**
     * STEP 3: pwm channel 1 to output
     * period time: 5ms
     * low time:    2.5ms
     */
    csi_pwm_out_config(&g_pwm, EXAMPLE_PWM_CH_IDX, 5000, 2500, PWM_POLARITY_LOW);
    csi_pwm_out_start(&g_pwm, EXAMPLE_PWM_CH_IDX);

    /* STEP 4: register callback func */
    ret = csi_pwm_attach_callback(&g_pwm, pwm_event_cb_fun, NULL);
    CHECK_RETURN(ret);

    /**
     * STEP 5: config channel 0 to input capture
     * posedge trigger 10 times into callback func
     */
    ret = csi_pwm_capture_config(&g_pwm, EXAMPLE_PWM_CAPTURE_CH_IDX, PWM_CAPTURE_PO
LARITY_POSEDGE, 10);
    CHECK_RETURN(ret);
}

```

```
/* STEP 6: start capture channel 0 */
ret = csi_pwm_capture_start(&g_pwm, EXAMPLE_PWM_CAPTURE_CH_IDX);
CHECK_RETURN(ret);

cb_pwm_flag = PWM_EVENT_ERROR;
/* Verify data after waiting for 10 posedges */
while(PWM_EVENT_CAPTURE_POSEDGE != cb_pwm_flag);

/* STEP 7: stop capture */
csi_pwm_capture_stop(&g_pwm, EXAMPLE_PWM_CAPTURE_CH_IDX);

/* STEP 8: cancel callback */
csi_pwm_detach_callback(&g_pwm);

csi_pwm_uninit(&g_pwm);

return ret;
}
```


I2S

说明

I2S(Inter—IC Sound)总线, 又称集成电路内置音频总线，是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专门用于音频设备之间的数据传输。

接口列表

I2S的CSI接口如下所示：

函数	说明
csi_i2s_init	I2S设备初始化
csi_i2s_uninit	I2S设备反初始化
csi_i2s_enbale	I2S设备使能/禁止接口
csi_i2s_format	I2S设备参数配置接口
csi_i2s_tx_select_sound_channel	I2S发送通道配置单/双声道
csi_i2s_rx_select_sound_channel	I2S接收通道配置单/双声道
csi_i2s_tx_link_dma	I2S发送通道配置DMA
csi_i2s_rx_link_dma	I2S接收通道配置DMA
csi_i2s_tx_set_buffer	I2S发送通道配置缓冲区地址
csi_i2s_rx_set_buffer	I2S接收通道配置缓冲区地址
csi_i2s_tx_set_period	设置完成多少数据发送上报周期
csi_i2s_rx_set_period	设置完成多少数据接收上报周期
csi_i2s_tx_buffer_avail	返回I2S发送缓冲区内数据量
csi_i2s_tx_buffer_reset	重置I2S发送缓冲区
csi_i2s_rx_buffer_avail	返回I2S接收缓冲区内数据量
csi_i2s_rx_buffer_reset	重置I2S接收缓冲区
csi_i2s_send	I2S同步模式发送数据
csi_i2s_receive	I2S同步模式接收数据
csi_i2s_send_async	I2S异步模式发送数据
csi_i2s_receive_async	I2S异步模式接收数据
csi_i2s_send_start	I2S开始发送音频流
csi_i2s_send_end	I2S新信道音频流

csi_i2s_send_pause	I2S暂停发送音频流
csi_i2s_send_resume	I2S恢复发送音频流
csi_i2s_send_stop	I2S停止发送音频流
csi_i2s_receive_start	I2S开始接收音频流
csi_i2s_receive_stop	I2S停止接收音频流
csi_i2s_attach_callback	I2S注册回调函数
csi_i2s_detach_callback	I2S注销回调函数
csi_i2s_get_state	获取I2S设备的当前读写状态

接口详细说明

csi_i2s_init

```
csi_error_t csi_i2s_init(csi_i2s_t *i2s, uint32_t idx)
```

- 功能描述:
 - 通过设备ID初始化对应的I2S实例。
- 参数:
 - `i2s` : 设备句柄（需要用户申请句柄空间）。
 - `idx` : 设备ID。
- 返回值:
 - 错误码csi_error_t。

csi_i2s_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄
callback	void (callback)(csi_i2s_t i2s, csi_i2s_event_t event, void *arg)	用户回调函数
arg	void *	回调函数对应传参
tx_buf	ringbuffer_t *	发送音频流用的ringbuffer缓冲区
rx_buf	ringbuffer_t *	接收音频流用的ringbuffer缓冲区
tx_dma	csi_dma_ch_t *	用于发送的DMA通道句柄
rx_dma	csi_dma_ch_t *	用于接收的DMA通道句柄
tx_period	uint32_t	设置完成多少数据发送上报 周期

		周期
rx_period	uint32_t	设置完成多少数据接收上报周期
state	csi_state_t	I2S设备的当前读写状态
priv	void *	对接接口句柄地址

csi_i2s_uninit

```
void csi_i2s_uninit(csi_i2s_t *i2s)
```

- 功能描述:
 - i2s实例反初始化。
 - 该接口会清理并释放相关的软硬件资源。
- 参数:
 - i2s : 实例句柄。
- 返回值 :
 - 无。

csi_i2s_enable

```
void csi_i2s_enable(csi_i2s_t *i2s, bool en)
```

- 功能描述:
 - 设置I2S设备的使能或禁止。
- 参数:
 - i2s : 实例句柄。
 - en : True代表I2S设备打开，Flase代表I2S设备关闭。
- 返回值 :
 - 无。
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_enable(&g_i2s, true);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_format

```
csi_error_t csi_i2s_format(csi_i2s_t *i2s, csi_i2s_format_t *format)
```

- 功能描述：
 - 配置I2S的参数。
- 参数：
 - `i2s` :实例句柄。
 - `format` :配置I2S的结构体指针。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
csi_i2s_format_t i2s_format;
i2s_format.mode = I2S_MODE_MASTER;           ///< 设置I2S 为主机
i2s_format.protocol = I2S_PROTOCOL_I2S;       ///< 设置I2S协议为I2S
i2s_format.width = I2S_SAMPLE_WIDTH_16BIT;    ///< 设置采样宽度为16bit
i2s_format.rate = I2S_SAMPLE_RATE_48000;     ///< 设置采样率为48K
i2s_format.polarity = I2S_LEFT_POLARITY_LOW;  ///< WSCLK的高低极性电平对应的声道
i2s_format.sclk_nfs = I2S_SCLK_32FS;          ///< 设置SCLK为FS的32倍
i2s_format.mclk_nfs = I2S_MCLK_256FS;        ///< 设置MCLK为FS的256倍
ret = csi_i2s_format(&g_i2s, &format);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_format_t

类型	成员	说明
csi_i2s_mode_t	mode	I2S主从模式设置
csi_i2s_protocol_t	protocol	I2S传输协议设置
csi_i2s_ws_left_polarity_t	polarity	WSCLK电平极性对应声道设置
csi_i2s_sample_rate_t	rate	I2S采样比率设置
csi_i2s_sample_width_t	width	I2S采样宽度设置
csi_i2s_sclk_freq_t	sclk_nfs	SCLK的频率设置
csi_i2s_mclk_freq_t	mclk_nfs	MCLK的频率设置

csi_i2s_mode_t

类型	说明
I2S_MODE_MASTER	I2S设置为主机
I2S_MODE_SLAVE	I2S设置为从机

csi_i2s_protocol_t

类型	说明
I2S_PROTOCOL_I2S	I2S传输协议为I2S
I2S_PROTOCOL_MSB_JUSTIFIED	I2S传输协议为MSB_JUSTIFIED
I2S_PROTOCOL_LSB_JUSTIFIED	I2S传输协议为LSB_JUSTIFIED
I2S_PROTOCOL_PCM	I2S传输协议为PCM

csi_i2s_ws_left_polarity_t

类型	说明
I2S_LEFT_POLARITY_LOW	低电平对应左声道
I2S_LEFT_POLARITY_HIGH	高电平对应左声道

csi_i2s_sample_rate_t

类型	说明
I2S_SAMPLE_RATE_8000	采样比率8K
I2S_SAMPLE_RATE_11025	采样比率11.025K
I2S_SAMPLE_RATE_12000	采样比率12K
I2S_SAMPLE_RATE_16000	采样比率16K
I2S_SAMPLE_RATE_22050	采样比率22.05K
I2S_SAMPLE_RATE_24000	采样比率24K
I2S_SAMPLE_RATE_32000	采样比率32K
I2S_SAMPLE_RATE_44100	采样比率44.1K
I2S_SAMPLE_RATE_48000	采样比率48K
I2S_SAMPLE_RATE_96000	采样比率96K
I2S_SAMPLE_RATE_192000	采样比率192K
I2S_SAMPLE_RATE_256000	采样比率256K

csi_i2s_sample_width_t

类型	说明
I2S_SAMPLE_WIDTH_16BIT	采样宽度16bits

I2S_SAMPLE_WIDTH_24BIT	采样宽度24bits
I2S_SAMPLE_WIDTH_32BIT	采样宽度32bits

csi_i2s_sclk_freq_t

类型	说明
I2S_SCLK_16FS	SCLK频率为FS的16倍
I2S_SCLK_32FS	SCLK频率为FS的32倍
I2S_SCLK_48FS	SCLK频率为FS的48倍
I2S_SCLK_64FS	SCLK频率为FS的64倍

csi_i2s_mclk_freq_t

类型	说明
I2S_MCLK_256FS	MCLK频率为256倍
I2S_MCLK_384FS	MCLK频率为384倍

csi_i2s_tx_select_sound_channel

```
csi_error_t csi_i2s_tx_select_sound_channel(csi_i2s_t *i2s, csi_i2s_sound_channel_t ch)
```

- 功能描述：
 - 设置I2S发送通道配置单/双声道。
- 参数：
 - `i2s` : 实例句柄。
 - `ch` : 设置单双声道。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_tx_select_sound_channel(&g_i2s, I2S_LEFT_CHANNEL);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_sound_channel_t

类型	说明
I2S_LEFT_CHANNEL	单声道选择左声道
I2S_RIGHT_CHANNEL	单声道选择右声道
I2S_LEFT_RIGHT_CHANNEL	选择左右声道

csi_i2s_rx_select_sound_channel

```
csi_error_t csi_i2s_rx_select_sound_channel(csi_i2s_t *i2s, csi_i2s_sound_channel_t ch)
```

- 功能描述：
 - 设置I2S接收通道配置单/双声道。
- 参数：
 - `i2s` : 实例句柄。
 - `ch` : 选择单声道。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_i2s_t g_i2s;  
csi_error_t ret;  
ret = csi_i2s_rx_select_sound_channel(&g_i2s, I2S_LEFT_CHANNEL);  
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_i2s_tx_link_dma

```
csi_error_t csi_i2s_tx_link_dma(csi_i2s_t *i2s, csi_dma_ch_t *tx_dma)
```

- 功能描述：
 - 设置发送通道连接DMA。
- 参数：
 - `i2s` : 实例句柄
 - `tx_dma` : 用于发送的DMA通道句柄，传NULL时会关闭DMA。
- 返回值：

- 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
csi_dma_ch_t g_dma_ch_tx;
/* 为发送设置DMA通道 */
ret = csi_i2s_tx_link_dma(&g_i2s, &g_dma_ch_tx);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_rx_link_dma

```
csi_error_t csi_i2s_rx_link_dma(csi_i2s_t *i2s, csi_dma_ch_t *rx_dma)
```

- 功能描述：
 - 设置接收通道连接DMA。
- 参数：
 - i2s :实例句柄
 - rx_dma :用于接收的DMA通道句柄，传NULL时会关闭DMA。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
csi_dma_ch_t g_dma_ch_rx;
/* 为接收设置DMA通道 */
ret = csi_i2s_rx_link_dma(&g_i2s, &g_dma_ch_rx);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_tx_set_buffer

```
void csi_i2s_tx_set_buffer(csi_i2s_t *i2s, ringbuffer_t *buffer)
```

- 功能描述：
 - 对i2s发送通道实例注册ringbuffer缓冲区。

- 参数：
 - `i2s` :实例句柄。
 - `buffer` :ringbuffer地址。
- 返回值：
 - 无。
- 使用示例

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ringbuffer_t tx_ring_buffer;
/* 将 ring_buffer传递到i2s实例 */
ret = csi_i2s_tx_set_buffer(&g_i2s, &tx_ring_buffer);
if (ret != CSI_OK) {
    return -1;
}
```

ringbuffer_t

成员	类型	描述
buffer	uint8_t *	环形缓冲区地址
size	uint32_t	环形缓冲区大小
write	uint32_t	环形缓冲区当前写指针位置
read	uint32_t	环形缓冲区当前读指针位置
data_len	uint32_t	环形缓冲区当前可读数据长度

csi_i2s_rx_set_buffer

```
void csi_i2s_rx_set_buffer(csi_i2s_t *i2s, ringbuffer_t *buffer)
```

- 功能描述：
 - 对i2s接收通道实例注册ringbuffer缓冲区
- 参数：
 - `i2s` :实例句柄
 - `buffer` :ringbuffer地址
- 返回值：
 - 无。
- 使用示例

```
/* 句柄使用前请先初始化 */
```

```
static csi_i2s_t g_i2s;
csi_error_t ret;
ringbuffer_t rx_ring_buffer;
/* 将 ring_buffer传递到i2s实例 */
ret = csi_i2s_rx_set_buffer(&g_i2s, &rx_ring_buffer);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_tx_set_period

```
csi_error_t csi_i2s_tx_set_period(csi_i2s_t *i2s, uint32_t period)
```

- 功能描述：
 - 设置完成多少数据发送上报周期，注意：，这里要设置period要小于缓冲区大小，且可以整除缓冲区大小。
- 参数：
 - `i2s` :实例句柄。
 - `period` : 产生回调数据的大小。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
#define I2S_MASTER_TX_BUF_SIZE 2048 //< 缓冲区长度
uint8_t i2s_master_tx_buf[I2S_MASTER_TX_BUF_SIZE]; //< 缓冲区
/* 将 ring_buffer传递到i2s实例 */
ret = csi_i2s_tx_set_period(&g_i2s, (I2S_MASTER_TX_BUF_SIZE >> 1));
if (ret != CSI_OK) {
    return -1;
}
```

注意：

设置period，必须设置为环形缓冲区的一半长度，否则驱动不会正常工作

csi_i2s_rx_set_period

```
csi_error_t csi_i2s_rx_set_period(csi_i2s_t *i2s, uint32_t period)
```

- 功能描述：

- 设置完成多少数据接收上报周期，注意：，这里要设置period要小于缓冲区大小，且可以整除缓冲区大小。
- 参数：
 - `i2s` :实例句柄。
 - `period` : 产生回调数据的大小。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
#define I2S_MASTER_RX_BUF_SIZE 2048          ///< 缓冲区长度
uint8_t i2s_master_rx_buf[I2S_MASTER_RX_BUF_SIZE]; ///< 缓冲区
/* 将 ring_buffer传递到i2s实例 */
ret = csi_i2s_rx_set_period(&g_i2s, (I2S_MASTER_RX_BUF_SIZE >> 1));
if (ret != CSI_OK) {
    return -1;
}
```

注意：

设置period，必须设置为环形缓冲区的一半长度，否则驱动不会正常工作

csi_i2s_tx_buffer_avail

```
uint32_t csi_i2s_tx_buffer_avail(csi_i2s_t *i2s)
```

- 功能描述：
 - 读取缓冲区剩余数据。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - uint32_t 当前缓冲区剩余数据数。
- 使用实例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
uint32_t num;
num = csi_i2s_tx_buffer_avail(&g_i2s);
```

csi_i2s_rx_buffer_avail

```
uint32_t csi_i2s_rx_buffer_avail(csi_i2s_t *i2s)
```

- 功能描述：
 - 读取缓冲区剩余数据。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - `uint32_t` 当前缓冲区剩余数据数。
- 使用实例：

```
/* 句柄使用前请先初始化 */  
static csi_i2s_t g_i2s;  
uint32_t num;  
num = csi_i2s_rx_buffer_avail(&g_i2s);
```

csi_i2s_tx_buffer_reset

```
csi_error_t csi_i2s_tx_buffer_reset(csi_i2s_t *i2s)
```

- 功能描述：
 - 对I2S发送缓冲区（ringbuffer）进行重置。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_i2s_t g_i2s;  
csi_error_t ret;  
ret = csi_i2s_tx_buffer_reset(&g_i2s);  
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_i2s_rx_buffer_reset

```
csi_error_t csi_i2s_rx_buffer_reset(csi_i2s_t *i2s)
```

- 功能描述：
 - 对I2S接收的缓冲区进行重置。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_rx_buffer_reset(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_send

```
int32_t csi_i2s_send(csi_i2s_t *i2s, const void *data, uint32_t size)
```

- 功能描述：
 - `i2s` 同步模式发送数据。
- 参数：
 - `i2s` :实例句柄。
 - `data` : 用户缓冲区数据指针。
 - `size` : 要发送的数据长度。
- 返回值：
 - `int32_t` 返回发送的数据长度或错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
uint8_t send_data[1024];
int32_t num;
num = csi_i2s_send(&g_i2s, send_data, sizeof(send_data));
if (num != sizeof(send_data)) {
    return -1;
}
```

csi_i2s_send_async

```
uint32_t csi_i2s_send_async(csi_i2s_t *i2s, const void *data, uint32_t size)
```

- 功能描述：
 - I2S 异步模式发送数据。
- 参数：
 - `i2s` :实例句柄。
 - `data` :用户缓冲区数据指针。
 - `size` :要发送的数据长度。
- 返回值：
 - `uint32_t` 返回成功写入缓冲区的数据长度。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
uint8_t send_data[1024];
uint32_t num;
num = csi_i2s_send_async(&g_i2s, send_data, sizeof(send_data));
```

注意：

由于I2S的主要功能是播放音频，提供的接口都是写数据到缓冲区的。用户发送一段音频需要根据I2S的回调函数，不断的写数据到缓冲区，数据进入缓冲区I2S会自动发送数据。

csi_i2s_receive

```
int32_t csi_i2s_receive(csi_i2s_t *i2s, const void *data, uint32_t size)
```

- 功能描述：
 - `i2s` 同步模式接收数据。
- 参数：
 - `i2s` :实例句柄。
 - `data` : 用户缓冲区数据指针。
 - `size` : 要接收的数据长度。
- 返回值：
 - `uint32_t` 返回接收的数据长度。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
uint8_t read_data[1024];
int32_t num;
```

```

num = csi_i2s_receive(&g_i2s, read_data, sizeof(read_data));
if (num != sizeof(read_data)) {
    return -1;
}

```

csi_i2s_receive_async

```
uint32_t csi_i2s_receive_async(csi_i2s_t *i2s, const void *data, uint32_t size)
```

- 功能描述：
 - I2S 异步模式接收数据。
- 参数：
 - `i2s` :实例句柄。
 - `data` :用户缓冲区数据指针。
 - `size` :要读取的数据长度。
- 返回值：
 - `uint32_t` 返回成功读取缓冲区的数据长度。
- 使用示例：

```

/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
uint8_t read_data[1024];
uint32_t num;
num = csi_i2s_receive_async(&g_i2s, read_data, sizeof(read_data));

```

注意：

由于I2S的主要功能是播放音频，提供的接口都是写数据到缓冲区的。用户发送一段音频需要根据I2S的回调函数，不断的写数据到缓冲区，数据进入缓冲区I2S会自动发送数据。

csi_i2s_send_start

```
csi_error_t csi_i2s_send_start(csi_i2s_t *i2s)
```

- 功能描述：
 - I2S 发送数据流开始运行，调用该接口后，DMA会开始数据搬运。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_send_start(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_send_resume

```
csi_error_t csi_i2s_send_resume(csi_i2s_t *i2s)
```

- 功能描述：
 - I2S发送数据暂停后调用该接口，可以在暂停的缓冲区位置重新启动发送数据。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_send_resume(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_send_pause

```
csi_error_t csi_i2s_send_pause(csi_i2s_t *i2s)
```

- 功能描述：
 - 调用该接口，DMA停止搬运，数据流停止，但是会保留缓冲区，保留发送的指针，以便恢复发送。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：


```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_send_pause(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_send_stop

```
csi_error_t csi_i2s_send_stop(csi_i2s_t *i2s)
```

- 功能描述：
 - I2S 发送数据流停止，调用该接口后，缓冲区会重置，DMA会停止。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_send_stop(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_receive_start

```
csi_error_t csi_i2s_receive_start(csi_i2s_t *i2s)
```

- 功能描述：
 - I2S 接收数据流开始运行，调用该接口后，DMA会不断的从I2S拿取数据到缓冲区。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_receive_start(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_receive_stop

```
csi_error_t csi_i2s_receive_stop(csi_i2s_t *i2s)
```

- 功能描述：
 - I2S 接收数据流停止，调用该接口后，缓冲区会重置，DMA会停止。
- 参数：
 - `i2s` :实例句柄。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_i2s_t g_i2s;
csi_error_t ret;
ret = csi_i2s_receive_stop(&g_i2s);
if (ret != CSI_OK) {
    return -1;
}
```

csi_i2s_attach_callback

```
csi_error_t csi_i2s_attach_callback(csi_i2s_t *i2s, void *callback, void *arg)
```

- 功能描述：
 - 注册回调函数。
- 参数：
 - `i2s` :实例句柄。
 - `callback` : i2s实例的事件回调函数（一般在中断上下文执行）。
 - `arg` :回调函数参数（可选，由用户定义）。
- 返回值：
 - 错误码csi_error_t。

callback

```
void (*callback)(csi_i2s_t *i2s, csi_i2s_event_t event, void *arg)
```

其中 i2s 为设备句柄，event 为传给回调函数的事件类型，arg 为用户自定义的回调函数对应的参数。

i2s 回调事件枚举类型csi_i2s_event_t定义如下：

事件类型	事件说明
I2S_EVENT_TRANSMIT_COMPLETE	数据发送完成事件
I2S_EVENT_RECEIVE_COMPLETE	数据接收完成错误事件
I2S_EVENT_TX_BUFFER_EMPTY	I2S发送FIFO数据为空
I2S_EVENT_RX_BUFFER_FULL	I2S接收FIFO数据满
I2S_EVENT_ERROR_OVERFLOW	I2S总线产生FIFO溢出错误事件
I2S_EVENT_ERROR_UNDERFLOW	I2S总线产生FIFO下溢错误事件
I2S_EVENT_ERROR	I2S总线数据错误

注意：在使用异步工作模式前，必须调用本函数来注册回调函数，否则将无法使用异步接口。

csi_i2s_detach_callback

```
void csi_i2s_detach_callback(csi_i2s_t *i2s)
```

- 功能描述：
 - 注销I2S设备的回调函数。
- 参数：
 - i2s：实例句柄。
- 返回值：
 - 无。

csi_i2s_get_state

```
csi_error_t csi_i2s_get_state(csi_i2s_t *i2s, csi_state_t *state)
```

- 功能描述：
 - 获取 i2s 的状态。通过此函数来判断I2S设备在获取状态的时刻是否可以进行send和receive操作。
- 参数：
 - i2s：实例句柄。
 - state: 用于返回状态信息的参数地址。
- 返回值：

- 错误码csi_error_t。

初始化及配置示例

```

/* 句柄空间一般使用静态空间 */
static csi_i2s_t g_i2s;
csi_dma_ch_t dma_ch_tx_handle;
#define I2S_TX_BUF_SIZE 2048
uint8_t i2s_tx_buf[I2S_TX_BUF_SIZE]; ///< 定义接收的缓冲区
static ringbuffer_t tx_ring_buffer;          ///< 定义环形缓冲区

volatile uint8_t cb_i2s_transfer_flag = 0;
static void i2s_event_cb_fun(csi_i2s_t *i2s, csi_i2s_event_t event, void *arg)
{
    if (event == I2S_EVENT_TRANSMIT_COMPLETE) {
        cb_i2s_transfer_flag = 1;
    }
}

/* 本示例 示例了I2S的master发送流启动 */
int main(void) {
    csi_error_t ret;
    csi_i2s_format_t i2s_format;
    /* init函数的idx参数, 请根据soc的实际情况进行选择 */
    ret = csi_i2s_init(&g_i2s, 0);
    if (ret != CSI_OK) {
        return -1;
    }

    csi_i2s_attach_callback(&g_i2s, i2s_event_cb_fun, NULL); ///< 注册回调函数

    i2s_format.mode = I2S_MODE_MASTER;          ///< 设置I2S为主机模式
    i2s_format.protocol = I2S_PROTOCOL_I2S;      ///< 设置为I2S协议
    i2s_format.width = I2S_SAMPLE_WIDTH_16BIT;   ///< 设置采样宽度为16bits
    i2s_format.rate = I2S_SAMPLE_RATE_48000;    ///< 设置采样比率为48K
    i2s_format.polarity = I2S_LEFT_POLARITY_LOW; ///< 设置高电平代表左声道
    csi_i2s_format(&g_i2s, &i2s_format);        ///< 调用该接口进行设置I2S

    csi_i2s_tx_link_dma(&g_i2s, &dma_ch_tx_handle); ///< 设置I2S的dma句柄, 连接DMA

    csi_i2s_tx_set_period(&g_i2s, I2S_TX_BUF_SIZE / 2); ///< 设置发送period值
    tx_ring_buffer.buffer = i2s_tx_buf;          ///< 将缓冲区地址传递给ring
    tx_ring_buffer.size = I2S_TX_BUF_SIZE;      ///< 将缓冲区长度传递给ring
    csi_i2s_tx_set_buffer(&g_i2s, &tx_ring_buffer);
    csi_i2s_tx_buffer_reset(&g_i2s);          ///< 将缓冲区重置

    csi_i2s_enable(&g_i2s, true);             ///< 开启I2S外设

```

```
csi_i2s_send_start(&g_i2s);          ///< 开启I2S发送流
}
```

注意：

开启I2S的enable

开启发送流以后，假如缓冲区是刚重置的状态 将对外发数据都是0

同步模式传输

注意：

无论是同步模式，异步模式都要求注册回调函数，DMA进行连接，启动发送流

同步模式发送数据

```
/* 句柄空间一般使用静态空间 */
static csi_i2s_t g_i2s;
csi_dma_ch_t dma_ch_tx_handle;
#define I2S_TX_BUF_SIZE 2048
uint8_t i2s_tx_buf[I2S_TX_BUF_SIZE];          ///< 定义缓冲区
static ringbuffer_t tx_ring_buffer;           ///< 定义环形缓冲区
uint8_t write_data[2048];                     ///< 用户发送数据
volatile uint8_t cb_i2s_transfer_flag = 0;
static void i2s_event_cb_fun(csi_i2s_t *i2s, csi_i2s_event_t event, void *arg)
{
    if (event == I2S_EVENT_TRANSMIT_COMPLETE) {
        cb_i2s_transfer_flag = 1;
    }
}

/* 本示例 示例了I2S的master发送流启动 */
int main(void) {
    csi_error_t ret;
    csi_i2s_format_t i2s_format;
    /* init函数的idx参数，请根据soc的实际情况进行选择 */
    ret = csi_i2s_init(&g_i2s, 0);
    if (ret != CSI_OK) {
        return -1;
    }

    csi_i2s_attach_callback(&g_i2s, i2s_event_cb_fun, NULL); ///< 注册回调函数

    i2s_format.mode = I2S_MODE_MASTER;          ///< 设置I2S为主机模式
    i2s_format.protocol = I2S_PROTOCOL_I2S;      ///< 设置为I2S协议
    i2s_format.width = I2S_SAMPLE_WIDTH_16BIT;   ///< 设置采样宽度为16bits
    i2s_format.rate = I2S_SAMPLE_RATE_48000;     ///< 设置采样比率为48K
```

```

i2s_format.polarity = I2S_LEFT_POLARITY_LOW;          ///< 设置高电平代表左声道
csi_i2s_format(&g_i2s, &i2s_format);                  ///< 调用该接口进行设置I2S

csi_i2s_tx_link_dma(&g_i2s, &dma_ch_tx_handle);        ///<设置I2S的dma句柄，连接DMA

csi_i2s_tx_set_period(&g_i2s, I2S_TX_BUF_SIZE / 2);    ///< 设置发送period值
tx_ring_buffer.buffer = i2s_tx_buf;                    ///< 将缓冲区地址传递给ringbuffer
tx_ring_buffer.size = I2S_TX_BUF_SIZE;                  ///< 将缓冲区长度传递给ringbuffer
csi_i2s_tx_set_buffer(&g_i2s, &tx_ring_buffer);
csi_i2s_tx_buffer_reset(&g_i2s);                        ///< 将缓冲区重置

csi_i2s_enable(&g_i2s, true);                          ///< 开启I2S外设
csi_i2s_send_start(&g_i2s);                            ///< 开启I2S发送流

for (uint32_t i = 0; i < sizeof(write_data); i++) {
    write_data[i] = 0x5a;
}

int32_t write_size = 0;
write_size = csi_i2s_send(&g_i2s, write_data, sizeof(write_data)); ///<等待数据发送

if (write_size == sizeof(write_data)) {
    printf("test_i2s successfully\n");
} else {
    printf("test_i2s failed\n");
}
csi_i2s_enable(&g_i2s, 0);
csi_i2s_tx_link_dma(&g_i2s, NULL);
csi_i2s_detach_callback(&g_i2s);
csi_i2s_uninit(&g_i2s);
}

```

同步模式接收数据

```

/* 句柄空间一般使用静态空间 */
static csi_i2s_t g_i2s;
csi_dma_ch_t dma_ch_rx_handle;
#define I2S_RX_BUF_SIZE 2048
uint8_t i2s_rx_buf[I2S_RX_BUF_SIZE];          ///< 定义缓冲区
static ringbuffer_t rx_ring_buffer;            ///< 定义环形缓冲区
uint8_t read_data[2048];                      ///< 用户接收数据
volatile uint8_t cb_i2s_transfer_flag = 0;
static void i2s_event_cb_fun(csi_i2s_t *i2s, csi_i2s_event_t event, void *arg)
{
    if (event == I2S_EVENT_TRANSMIT_COMPLETE) {
        cb_i2s_transfer_flag = 1;
    }
}

```

```

}

/* 本示例 示例了I2S的master接收流启动 */
int main(void) {
    csi_error_t ret;
    csi_i2s_format_t i2s_format;
    /* init函数的idx参数, 请根据soc的实际情况进行选择 */
    ret = csi_i2s_init(&g_i2s, 0);
    if (ret != CSI_OK) {
        return -1;
    }

    csi_i2s_attach_callback(&g_i2s, i2s_event_cb_fun, NULL); ///< 注册回调函数

    i2s_format.mode = I2S_MODE_MASTER;                ///< 设置I2S为主机模式
    i2s_format.protocol = I2S_PROTOCOL_I2S;            ///< 设置为I2S协议
    i2s_format.width = I2S_SAMPLE_WIDTH_16BIT;        ///< 设置采样宽度为16bits
    i2s_format.rate = I2S_SAMPLE_RATE_48000;          ///< 设置采样比率为48K
    i2s_format.polarity = I2S_LEFT_POLARITY_LOW;      ///< 设置高电平代表左声道
    csi_i2s_format(&g_i2s, &i2s_format);              ///< 调用该接口进行设置I2S

    csi_i2s_rx_link_dma(&g_i2s, &dma_ch_rx_handle);  ///< 设置I2S的dma句柄, 连接DMA

    csi_i2s_rx_set_period(&g_i2s, I2S_RX_BUF_SIZE / 2); ///< 设置发送period值
    tx_ring_buffer.buffer = i2s_rx_buf;                ///< 将缓冲区地址传递给ringbuffer
    tx_ring_buffer.size = I2S_RX_BUF_SIZE;             ///< 将缓冲区长度传递给ringbuffer
    csi_i2s_rx_set_buffer(&g_i2s, &rx_ring_buffer);
    csi_i2s_rx_buffer_reset(&g_i2s);                  ///< 将缓冲区重置

    csi_i2s_enable(&g_i2s, true);                    ///< 开启I2S外设
    csi_i2s_receive_start(&g_i2s);                   ///< 开启I2S接收流

    int32_t read_size = 0;
    read_size = csi_i2s_receive(&g_i2s, read_data, sizeof(read_data)); ///< 等待数据发送

    if (read_size == sizeof(read_data)) {
        printf("test_i2s successfully\n");
    } else {
        printf("test_i2s failed\n");
    }
    csi_i2s_enable(&g_i2s, 0);
    csi_i2s_rx_link_dma(&g_i2s, NULL);
    csi_i2s_detach_callback(&g_i2s);
    csi_i2s_uninit(&g_i2s);
}

```

异步模式传输

注意：

无论是同步模式，异步模式都要求注册回调函数，DMA进行连接，启动发送流

异步模式发送数据

```
/* 句柄空间一般使用静态空间 */
static csi_i2s_t g_i2s;
csi_dma_ch_t dma_ch_tx_handle;
#define I2S_TX_BUF_SIZE 2048
uint8_t i2s_tx_buf[I2S_TX_BUF_SIZE];          ///< 定义缓冲区
static ringbuffer_t tx_ring_buffer;           ///< 定义环形缓冲区
uint8_t write_data[2048];                     ///< 用户发送数据
volatile uint8_t cb_i2s_transfer_flag = 0;
static void i2s_event_cb_fun(csi_i2s_t *i2s, csi_i2s_event_t event, void *arg)
{
    if (event == I2S_EVENT_TRANSMIT_COMPLETE) {
        cb_i2s_transfer_flag--;
    }
}

/* 本示例 示例了I2S的master发送流启动 */
int main(void) {
    csi_error_t ret;
    csi_i2s_format_t i2s_format;
    /* init函数的idx参数，请根据soc的实际情况进行选择 */
    ret = csi_i2s_init(&g_i2s, 0);
    if (ret != CSI_OK) {
        return -1;
    }

    csi_i2s_attach_callback(&g_i2s, i2s_event_cb_fun, NULL); ///< 注册回调函数

    i2s_format.mode = I2S_MODE_MASTER;                ///< 设置I2S为主机模式
    i2s_format.protocol = I2S_PROTOCOL_I2S;            ///< 设置为I2S协议
    i2s_format.width = I2S_SAMPLE_WIDTH_16BIT;         ///< 设置采样宽度为16bits
    i2s_format.rate = I2S_SAMPLE_RATE_48000;          ///< 设置采样比率为48K
    i2s_format.polarity = I2S_LEFT_POLARITY_LOW;       ///< 设置高电平代表左声道
    csi_i2s_format(&g_i2s, &i2s_format);              ///< 调用该接口进行设置I2S

    csi_i2s_tx_link_dma(&g_i2s, &dma_ch_tx_handle);    ///< 设置I2S的dma句柄，连接DMA

    csi_i2s_tx_set_period(&g_i2s, I2S_TX_BUF_SIZE / 2); ///< 设置发送period值
    tx_ring_buffer.buffer = i2s_tx_buf;                ///< 将缓冲区地址传递给ringbuffer
    tx_ring_buffer.size = I2S_TX_BUF_SIZE;             ///< 将缓冲区长度传递给ringbuffer
}
```



```

er
csi_i2s_tx_set_buffer(&g_i2s, &tx_ring_buffer);
csi_i2s_tx_buffer_reset(&g_i2s);          ///< 将缓冲区重置

csi_i2s_enable(&g_i2s, true);             ///< 开启I2S外设
csi_i2s_send_start(&g_i2s);              ///< 开启I2S发送流

for (uint32_t i = 0; i < sizeof(write_data); i++) {
    write_data[i] = 0x5a;
}

uint32_t write_size = 0;
cb_i2s_transfer_flag = sizeof(write_data) / g_i2s->tx_period; ///< 请确保发送数据是peri
od的整数倍
write_size = csi_i2s_send_async(&g_i2s, write_data, sizeof(write_data)); ///< 等待数
据发送
while(cb_i2s_transfer_flag != 0); ///< 等待同步
if (write_size == sizeof(write_data)) {
    printf("test_i2s successfully\n");
} else {
    printf("test_i2s failed\n");
}
csi_i2s_enable(&g_i2s, 0);
csi_i2s_tx_link_dma(&g_i2s, NULL);
csi_i2s_detach_callback(&g_i2s);
csi_i2s_uninit(&g_i2s);
}

```

异步模式接收数据

```

/* 句柄空间一般使用静态空间 */
static csi_i2s_t g_i2s;
csi_dma_ch_t dma_ch_rx_handle;
#define I2S_RX_BUF_SIZE 2048
uint8_t i2s_rx_buf[I2S_RX_BUF_SIZE];          ///< 定义缓冲区
static ringbuffer_t rx_ring_buffer;           ///< 定义环形缓冲区
uint8_t read_data[2048];                      ///< 用户接收数据
volatile uint8_t cb_i2s_transfer_flag = 0;
static void i2s_event_cb_fun(csi_i2s_t *i2s, csi_i2s_event_t event, void *arg)
{
    if (event == I2S_EVENT_TRANSMIT_COMPLETE) {
        cb_i2s_transfer_flag = 1;
    }
}

/* 本示例 示例了I2S的master接收流启动 */
int main(void) {
    csi_error_t ret;
    csi_i2s_format_t i2s_format;

```

```

/* init函数的idx参数, 请根据soc的实际情况进行选择 */
ret = csi_i2s_init(&g_i2s, 0);
if (ret != CSI_OK) {
    return -1;
}

csi_i2s_attach_callback(&g_i2s, i2s_event_cb_fun, NULL); ///< 注册回调函数

i2s_format.mode = I2S_MODE_MASTER;                ///< 设置I2S为主机模式
i2s_format.protocol = I2S_PROTOCOL_I2S;            ///< 设置为I2S协议
i2s_format.width = I2S_SAMPLE_WIDTH_16BIT;         ///< 设置采样宽度为16bits
i2s_format.rate = I2S_SAMPLE_RATE_48000;          ///< 设置采样比率为48K
i2s_format.polarity = I2S_LEFT_POLARITY_LOW;       ///< 设置高电平代表左声道
csi_i2s_format(&g_i2s, &i2s_format);              ///< 调用该接口进行设置I2S

csi_i2s_rx_link_dma(&g_i2s, &dma_ch_rx_handle);    ///< 设置I2S的dma句柄, 连接DMA

csi_i2s_rx_set_period(&g_i2s, I2S_RX_BUF_SIZE / 2); ///< 设置发送period值
tx_ring_buffer.buffer = i2s_rx_buf;                ///< 将缓冲区地址传递给ringbuffer
tx_ring_buffer.size = I2S_RX_BUF_SIZE;             ///< 将缓冲区长度传递给ringbuffer
csi_i2s_rx_set_buffer(&g_i2s, &rx_ring_buffer);
csi_i2s_rx_buffer_reset(&g_i2s);                  ///< 将缓冲区重置

csi_i2s_enable(&g_i2s, true);                      ///< 开启I2S外设
csi_i2s_receive_start(&g_i2s);                    ///< 开启I2S接收流

printf("start i2s receive\n");
uint32_t read_size = 0;
uint32_t period_num = sizeof(read_data) / i2s_slave.rx_period; ///< 请确保数据是period的整数倍
cb_slave_transfer_flag = 0;
while (1) {
    if (cb_slave_transfer_flag == 1) {
        cb_slave_transfer_flag = 0;
        read_size += csi_i2s_receive_async(&g_i2s, (read_data + read_size), (sizeof(read_data) - read_size)); ///< 接收完注意: 偏移接收指针
        period_num--;

        if (period_num == 0) {
            break;
        }
    }
}

if (read_size == sizeof(read_data)) {
    printf("test_i2s successfully\n");
} else {
    printf("test_i2s failed\n");
}

```

```
}  
csi_i2s_enable(&g_i2s, 0);  
csi_i2s_rx_link_dma(&g_i2s, NULL);  
csi_i2s_detach_callback(&g_i2s);  
csi_i2s_uninit(&g_i2s);  
}
```

CODEC

说明

CODEC在这里指的是同时具有D/A（数字讯号转换成模拟讯号）和A/D（模拟讯号转换成数字讯号）转换功能的编解码器，播放音乐的时候用到的是D/A转换功能。在录音的时候用到的是A/D转换功能。在接口中，D/A指的是输出通道，A/D指的是输入通道。

接口列表

CODEC的CSI接口如下所示：

函数	说明
csi_codec_init	CODEC设备初始化
csi_codec_uninit	CODEC设备去初始化
csi_codec_output_open	CODEC输出通道打开
csi_codec_output_config	CODEC输出通道配置
csi_codec_output_attach_callback	CODEC输出通道注册回调函数
csi_codec_output_detach_callback	CODEC输出通道注销回调函数
csi_codec_output_close	CODEC输出通道关闭
csi_codec_output_link_dma	CODEC输出通道配置DMA
csi_codec_output_write	CODEC输出通道同步模式发送数据
csi_codec_output_write_async	CODEC输出通道异步模式发送数据
csi_codec_output_start	CODEC开始输出音频流
csi_codec_output_stop	CODEC结束输出音频流
csi_codec_output_pause	CODEC暂停输出音频流
csi_codec_output_resume	CODEC恢复输出音频流
csi_codec_output_buffer_avail	返回CODEC输出缓冲区内数据量
csi_codec_output_buffer_reset	重置CODEC输出缓冲区
csi_codec_output_mute	CODEC输出通道静音
csi_codec_output_digital_gain	CODEC输出通道设置数字增益
csi_codec_output_analog_gain	CODEC输出通道设置模拟增益
csi_codec_output_mix_gain	CODEC输出通道设置混频增益

csi_codec_output_get_state	CODEC输出通道获取当前读写状态
csi_codec_input_open	CODEC输入通道打开
csi_codec_input_config	CODEC输入通道配置
csi_codec_input_attach_callback	CODEC输入通道注册回调函数
csi_codec_input_detach_callback	CODEC输入通道注销回调函数
csi_codec_input_close	CODEC输入通道关闭
csi_codec_input_link_dma	CODEC输入通道配置DMA
csi_codec_input_read	CODEC输入通道同步模式读取数据
csi_codec_input_read_async	CODEC输入通道异步模式读取数据
csi_codec_input_start	CODEC接收输入音频流
csi_codec_input_stop	CODEC结束接收输入音频流
csi_codec_input_mute	CODEC输入通道静音
csi_codec_input_digital_gain	CODEC输入通道设置数字增益
csi_codec_input_analog_gain	CODEC输入通道设置模拟增益
csi_codec_input_mix_gain	CODEC输入通道设置混频增益
csi_codec_input_get_state	CODEC输入通道获取当前读写状态

接口描述

csi_codec_init

```
csi_error_t csi_codec_init(csi_codec_t *codec, uint32_t idx)
```

- 功能描述:
 - 通过设备ID初始化对应的CODEC实例。
- 参数:
 - `codec` : 设备句柄（需要用户申请句柄空间）。
 - `idx` : 设备ID。
- 返回值:
 - CSI_OK: 初始化成功。
 - CSI_ERROR: 初始化失败。

csi_codec_t

成员	类型	说明
dev	csi_dev_t	csi设备统一句柄

output_chs	csi_codec_output_t	输出通道句柄
input_chs	csi_codec_input_t	输入通道句柄
*priv	void	设备私有变量

ringbuffer_t

成员	类型	描述
buffer	uint8_t *	环形缓冲区地址
size	uint32_t	环形缓冲区大小
write	uint32_t	环形缓冲区当前写指针位置
read	uint32_t	环形缓冲区当前读指针位置
data_len	uint32_t	环形缓冲区当前可读数据长度

csi_codec_output_t

成员	类型	描述
codec	csi_codec_t *	CODEC设备句柄
ch_idx	uint32_t	当前通道的序号
callback	void (callback)(csi_codec_output_t output, csi_codec_event_t event, void *arg)	当前通道的回调
arg	void *	当前通道的用户参数
ring_buf	ringbuffer_t *	当前通道的缓冲器句柄
period	uint32_t	设置完成多少数据发送上报周期
sound_channel_num	uint32_t	声道数
state	csi_state_t	当前通道的状态
dma	csi_dma_ch_t *	当前通道的DMA句柄
next	struct csi_codec_output *	下一个输出通道的地址指针
priv	void *	设备私有变量

csi_codec_input_t

成员	类型	描述
codec	csi_codec_t *	CODEC设备句柄
ch_idx	uint32_t	当前通道的序号

callback	void (callback)(csi_codec_input_t input, csi_codec_event_t event, void *arg)	当前通道的回调
arg	void *	当前通道的用户参数
ring_buf	ringbuffer_t *	当前通道的缓冲器句柄
period	uint32_t	设置完成多少数据接收上报周期
sound_channel_num	uint32_t	声道数
state	csi_state_t	当前通道的状态
dma	csi_dma_ch_t *	当前通道的DMA句柄
next	struct csi_codec_input *	下一个输入通道的地址指针
priv	void *	设备私有变量

csi_codec_uninit

```
void csi_codec_uninit(csi_codec_t *codec)
```

- 功能描述:
 - codec实例反初始化。
 - 该接口会清理并释放相关的软硬件资源。
- 参数:
 - `codec` : 实例句柄。
- 返回值 :
 - 无。

csi_codec_output_open

```
csi_error_t csi_codec_output_open(csi_codec_t *codec, csi_codec_output_t *ch, uint32_t ch_idx)
```

- 功能描述 :
 - 将输出通道的ch句柄注册到codec句柄中。
 - 初始化输出通道有关的硬件资源。
- 参数 :
 - `codec` : codec实例句柄。
 - `ch` : 输出通道的实例句柄。
 - `ch_idx` : 通道的ID。

- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_t g_codec;
csi_error_t ret;
csi_codec_output_t output_ch;
/* ch_idx 需要根据实际硬件是否支持填入*/
ret = csi_codec_output_open(&g_codec, &output_ch, 0);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_output_config

```
csi_error_t csi_codec_output_config(csi_codec_output_t *ch, csi_codec_output_config_t *config)
```

- 功能描述：
 - 根据传入的配置配置输出通道。
 - 配置输出通道采样宽度、采样比率、设置缓冲区地址、设置输出通道的输出模式（差分输出还是单端输出）。
- 参数
 - ch :通道实例句柄。
 - config : 配置参数结构体指针。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_t g_codec;
csi_error_t ret;
csi_codec_output_t output_ch;
csi_codec_output_config_t output_config;    ///< 输出通道的配置参数
output_config.bit_width = 16;                ///< 设置采样宽度为16bit
output_config.sample_rate = 48000;           ///< 设置采样比率为48K
output_config.buffer = output_buf;           ///< 设置缓冲区地址
output_config.buffer_size = OUTPUT_BUF_SIZE; ///< 设置缓冲区大小
output_config.period = OUTPUT_BUF_SIZE/2;   ///< 设置发送多少数据上报值
output_config.mode = CODEC_OUTPUT_DIFFERENCE; ///< 设置差分输出
ret = csi_codec_output_config(&output_ch,&output_config);
if (ret != CSI_OK) {
    return -1;
}
```



```
}

```

csi_codec_output_config_t

成员	类型	说明
sample_rate	uint32_t	输出通道的采样比率
bit_width	uint32_t	输出通道的宽度
mode	csi_codec_output_mode_t	输出通道的模式
buffer	uint8_t *	输出通道缓冲区地址
buffer_size	uint32_t	缓冲区地址长度
period	uint32_t	发送多少数据上报值
sound_channel_num	uint32_t	声道数

csi_codec_output_mode_t

类型	说明
CODEC_OUTPUT_SINGLE_ENDED	差分输出
CODEC_OUTPUT_DIFFERENCE	单端输出

csi_codec_output_attach_callback

```
csi_error_t csi_codec_output_attach_callback(csi_codec_output_t *ch, void *callback, void *arg)
```

- 功能描述：
 - 设置输出通道回调函数
- 参数：
 - csi_codec_output_t : 输出通道实例句柄
 - callback : codec 输出通道的事件回调函数（一般在上下文执行）函数传参见下文
 - arg :用户参数
- 返回值：
 - 错误码csi_error_t。

callback

```
void (*callback)(csi_codec_output_t *output, csi_codec_event_t event, void *arg)
```

其中 output为输出通道句柄，event 为传给回调函数的事件类型，arg 为用户自定义的回调函数对应的参数。

codec 回调事件枚举类型csi_codec_event_t定义如下：

类型	说明
CODEC_EVENT_PERIOD_READ_COMPLETE	CODEC接收period数据数完成
CODEC_EVENT_PERIOD_WRITE_COMPLETE	CODEC发送period数据数完成
CODEC_EVENT_WRITE_BUFFER_EMPTY	CODEC发送缓冲区已经空
CODEC_EVENT_READ_BUFFER_FULL	CODEC接收缓冲区已经满
CODEC_EVENT_ERROR_OVERFLOW	CODEC的FIFO产生溢出错误
CODEC_EVENT_ERROR_UNDERFLOW	CODEC的FIFO产生下溢错误
CODEC_EVENT_ERROR	CODEC传输错误事件

csi_codec_output_detach_callback

```
void csi_codec_output_detach_callback(csi_codec_output_t *ch)
```

- 功能描述：
 - 注销CODEC 输出通道的回调函数。
- 参数：
 - ch：通道实例句柄。
- 返回值：
 - 无。

csi_codec_output_close

```
void csi_codec_output_close(csi_codec_output_t *ch)
```

- 功能描述：
 - 关闭输出通道。
 - 调用该接口会马上停止输出数据。
- 参数：
 - ch：通道实例句柄。
- 返回值：
 - 无。

csi_codec_output_link_dma

```
csi_error_t csi_codec_output_link_dma(csi_codec_output_t *ch, csi_dma_ch_t *dma)
```

- 功能描述：

- 输出通道连接DMA。
- 参数：
 - `ch`：输出通道的实例句柄。
 - `dma`：dma实例句柄。
- 返回：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
csi_error_t ret;
csi_dma_ch_t g_dma_ch;
/* 为输出通道设置DMA通道 */
ret = csi_codec_output_link_dma(&codec_output, &g_dma_ch);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_output_send

```
uint32_t csi_codec_output_send(csi_codec_output_t *ch, const void *data, uint32_t size)
```

- 功能描述：
 - 输出通道同步模式发送数据。
- 参数：
 - `ch`：输出通道的实例句柄。
 - `data`：发送数据指针。
 - `size`：发送数据长度。
- 返回：
 - uint32_t 发送成功数据的长度。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
uint8_t write_data[1024];
uint32_t num;
num = csi_codec_output_send(&codec_output, write_data, sizeof(write_data));
if (num != sizeof(write_data)) {
    return -1;
}
```

csi_codec_output_send_async

```
uint32_t csi_codec_output_send_async(csi_codec_output_t *ch, const void *data, uint32_t size)
```

- 功能描述：
 - 输出通道异步模式发送数据。
- 参数：
 - `ch` : 输出通道的实例句柄。
 - `data` : 发送数据指针。
 - `size` : 发送数据长度。
- 返回：
 - `uint32_t` 发送成功数据的长度。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
uint8_t write_data[1024];
uint32_t num;
/*调用该接口后，数据会写到缓冲区，用户需要根据设定的period值来判断数据是否发送完毕*/
num = csi_codec_output_send_async(&codec_output, write_data, sizeof(write_data));
);
if (num != sizeof(write_data)) {
    return -1;
}
```

csi_codec_output_start

```
csi_error_t csi_codec_output_start(csi_codec_output_t *ch)
```

- 功能描述：
 - 输出通道开始数据流。
- 参数：
 - `ch` : 输出通道的实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
csi_error_t ret;
```

```
num = csi_codec_output_start(&codec_output);  
if (num != CSI_OK) {  
    return -1;  
}
```

csi_codec_output_stop

```
csi_error_t csi_codec_output_stop(csi_codec_output_t *ch)
```

- 功能描述：
 - 输出通道结束数据流。
- 参数：
 - `ch` : 输出通道的实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_codec_output_t codec_output;  
csi_error_t ret;  
ret = csi_codec_output_stop(&codec_output);  
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_codec_output_pause

```
csi_error_t csi_codec_output_pause(csi_codec_output_t *ch)
```

- 功能描述：
 - 输出通道暂停数据流。
- 参数：
 - `ch` : 输出通道的实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_codec_output_t codec_output;  
csi_error_t ret;  
ret = csi_codec_output_pause(&codec_output);
```

```
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_codec_output_resume

```
csi_error_t csi_codec_output_resume(csi_codec_output_t *ch)
```

- 功能描述：
 - 输出通道恢复发送数据流。
- 参数：
 - `ch` : 输出通道的实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_codec_output_t codec_output;  
csi_error_t ret;  
ret = csi_codec_output_resume(&codec_output);  
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_codec_output_buffer_avail

```
uint32_t csi_codec_output_buffer_avail(csi_codec_output_t *ch)
```

- 功能描述：
 - 返回当前缓冲区剩余数据数。
- 参数：
 - `ch` : 输出通道的实例句柄。
- 返回值：
 - `uint32_t` 剩余数据数。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_codec_output_t codec_output;  
uint32_t num;  
num = csi_codec_output_buffer_avail(&codec_output);
```

csi_codec_output_buffer_reset

```
csi_error_t csi_codec_output_buffer_reset(csi_codec_output_t *ch)
```

- 功能描述：
 - 对缓冲区数据全部置0，重置ringbuffer。
- 参数：
 - `ch`：输出通道的实例句柄。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_codec_output_t codec_output;  
csi_error_t ret;  
ret = csi_codec_output_buffer_reset(&codec_output);  
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_codec_output_mute

```
csi_error_t csi_codec_output_mute(csi_codec_output_t *ch, bool en)
```

- 功能描述：
 - 设置输出通道静音。
- 参数：
 - `ch`：输出通道的实例句柄。
 - `en`：true 代表输出静音, Flase 代表输出不静音。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */  
static csi_codec_output_t codec_output;  
csi_error_t ret;  
ret = csi_codec_output_mute(&codec_output, true);  
if (ret != CSI_OK) {  
    return -1;  
}
```

csi_codec_output_digital_gain

```
csi_error_t csi_codec_output_digital_gain(csi_codec_output_t *ch, uint32_t val)
```

- 功能描述：
 - 设置输出通道数码增益。
- 参数：
 - `ch` : 输出通道的实例句柄。
 - `val` : 增益的DB值。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
csi_error_t ret;
/* 设置的数码增益 需要根据实际硬件支持的值进行设置 */
ret = csi_codec_output_digital_gain(&codec_output, 0x20);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_output_analog_gain

```
csi_error_t csi_codec_output_analog_gain(csi_codec_output_t *ch, uint32_t val)
```

- 功能描述：
 - 设置输出通道模拟增益。
- 参数：
 - `ch` : 输出通道的实例句柄。
 - `val` : 增益的DB值。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
csi_error_t ret;
/* 设置的数码增益 需要根据实际硬件支持的值进行设置 */
ret = csi_codec_output_analog_gain(&codec_output, 0x20);
if (ret != CSI_OK) {
```



```
    return -1;
}
```

csi_codec_output_mix_gain

```
csi_error_t csi_codec_output_mix_gain(csi_codec_output_t *ch, uint32_t val)
```

- 功能描述：
 - 设置输出通道混频增益。
- 参数：
 - `ch`：输出通道的实例句柄。
 - `val`：增益的DB值。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_output_t codec_output;
csi_error_t ret;
/* 设置的混频增益 需要根据实际硬件支持的值进行设置 */
ret = csi_codec_output_mix_gain(&codec_output, 0x20);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_output_get_state

```
csi_error_t csi_codec_output_get_state(csi_codec_output_t *ch, csi_state_t *state)
```

- 功能描述：
 - 获取输出通道的状态。通过此函数来判断code通道在获取状态的时刻是否可以进行读写操作。
- 参数
 - `ch`：实例句柄。
 - `state`：用于返回状态信息的参数地址。
- 返回值：
 - 错误码csi_error_t。

csi_codec_input_open

```
csi_error_t csi_codec_input_open(csi_codec_t *codec, csi_codec_input_t *ch, uint32_t
```

```
t ch_idx)
```

- 功能描述：
 - 将输入通道的ch句柄注册到codec句柄中。
 - 初始化输入通道有关的硬件资源。
- 参数：
 - `codec` : codec实例句柄。
 - `ch` : 输入通道的实例句柄。
 - `ch_idx` : 通道的ID。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_t g_codec;
csi_error_t ret;
csi_codec_input_t input_ch;
/* ch_idx 需要根据实际硬件是否支持填入*/
ret = csi_codec_input_open(&g_codec, &input_ch, 0);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_config

```
csi_error_t csi_codec_input_config(csi_codec_input_t *ch, csi_codec_input_config_t
*config)
```

- 功能描述：
 - 根据传入的配置配置输入通道。
 - 配置输入通道采样宽度、采样比率、设置缓冲区地址、设置输入通道的输出模式（差分输入还是单端输入）。
- 参数
 - `ch` : 通道实例句柄。
 - `config` : 配置参数。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_t g_codec;
```

```

#define CODEC_BUF_SIZE 2048
uint8_t input_buf[CODEC_BUF_SIZE]; ///< 定义接收的缓冲区
csi_error_t ret;
csi_codec_input_t input_ch;
csi_codec_input_config_t input_config;          ///< 输入通道的配置参数
input_config.bit_width = 16;                    ///< 设置采样宽度为16bit
input_config.sample_rate = 48000;               ///< 设置采样比率为48K
input_config.buffer = input_buf;                ///< 设置缓冲区地址
input_config.buffer_size = CODEC_BUF_SIZE;      ///< 设置缓冲区大小
input_config.period = 1024;                     ///< 设置接收多少数据上报值
input_config.mode = CODEC_INPUT_DIFFERENCE;    ///< 设置差分输入
ret = csi_codec_input_config(&input_ch,&input_config);
if (ret != CSI_OK) {
    return -1;
}

```

csi_codec_input_config_t

成员	类型	说明
sample_rate	uint32_t	输入通道的采样比率
bit_width	uint32_t	输入通道的宽度
mode	csi_codec_input_mode_t	输入通道的模式
buffer	uint8_t *	输入通道缓冲区地址
buffer_size	uint32_t	缓冲区地址长度
period	uint32_t	接收多少数据上报值
sound_channel_num	uint32_t	声道数

csi_codec_input_mode_t

类型	说明
CODEC_INPUT_SINGLE_ENDED	差分输入
CODEC_INPUT_DIFFERENCE	单端输入

csi_codec_input_attach_callback

```

csi_error_t csi_codec_input_attach_callback(csi_codec_input_t *ch, void *callback,
void *arg)

```

- 功能描述：
 - 设置输入通道回调函数。
- 参数：
 - `csi_codec_input_t`：输入通道实例句柄。

- `callback`: codec 输入通道的事件回调函数（一般在上下文执行）。
- `arg`: 回调函数参数（可选，由用户定义）。
- 返回值：
 - 错误码 `csi_error_t`。

callback

```
void (*callback)(csi_codec_input_t *input, csi_codec_event_t event, void *arg)
```

其中 `input` 为输入通道句柄，`event` 为传给回调函数的事件类型，`arg` 为用户自定义的回调函数对应的参数。

codec 回调事件枚举类型 `csi_codec_event_t` 定义如下：

类型	说明
<code>CODEC_EVENT_PERIOD_READ_COMPLETE</code>	接收period完成
<code>CODEC_EVENT_PERIOD_WRITE_COMPLETE</code>	发送period完成
<code>CODEC_EVENT_WRITE_BUFFER_EMPTY</code>	发送缓冲区已经空
<code>CODEC_EVENT_READ_BUFFER_FULL</code>	接收缓冲区已经满
<code>CODEC_EVENT_TRANSFER_ERROR</code>	传输错误

csi_codec_input_detach_callback

```
void csi_codec_input_detach_callback(csi_codec_input_t *ch)
```

- 功能描述：
 - 注销CODEC 输入通道的回调函数。
- 参数：
 - `ch`：通道实例句柄。
- 返回值：
 - 无。

csi_codec_input_close

```
void csi_codec_input_close(csi_codec_input_t *ch)
```

- 功能描述：
 - 关闭输入通道。
 - 调用该接口会马上停止接收数据。
- 参数：
 - `ch`：通道实例句柄。

csi_codec_input_link_dma

```
csi_error_t csi_codec_input_link_dma(csi_codec_input_t *ch, csi_dma_ch_t *dma)
```

- 功能描述：
 - 输入通道连接DMA。
- 参数：
 - `ch` : 输入通道的实例句柄。
 - `dma` : dma实例句柄。
- 返回：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
csi_dma_ch_t g_dma_ch;
/* 为输入通道设置DMA通道 */
ret = csi_codec_input_link_dma(&codec_input, &g_dma_ch);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_send

```
uint32_t csi_codec_input_send(csi_codec_input_t *ch, const void *data, uint32_t size)
```

- 功能描述：
 - 输入通道同步模式发送数据。
- 参数：
 - `ch` : 输入通道的实例句柄。
 - `data` : 发送数据指针。
 - `size` : 发送数据长度。
- 返回：
 - uint32_t 发送成功数据的长度。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
uint8_t write_data[1024];
```

```
uint32_t num;
num = csi_codec_input_send(&codec_input, write_data, sizeof(write_data));
if (num != sizeof(write_data)) {
    return -1;
}
```

csi_codec_input_send_async

```
uint32_t csi_codec_input_send_async(csi_codec_input_t *ch, const void *data, uint32_t size)
```

- 功能描述：
 - 输入通道异步模式发送数据。
- 参数：
 - `ch` : 输入通道的实例句柄。
 - `data` : 发送数据指针。
 - `size` : 发送数据长度。
- 返回：
 - `uint32_t` 发送成功数据的长度。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
uint8_t write_data[1024];
uint32_t num;
/*调用该接口后，数据会写到缓冲区，用户需要根据设定的period值来判断数据是否发送完毕*/
num = csi_codec_input_send_async(&codec_input, write_data, sizeof(write_data));
if (num != sizeof(write_data)) {
    return -1;
}
```

csi_codec_input_start

```
csi_error_t csi_codec_input_start(csi_codec_input_t *ch)
```

- 功能描述：
 - 输入通道开始数据流。
- 参数：
 - `ch` : 输入通道的实例句柄。
- 返回值：
 - 错误码`csi_error_t`。

- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
num = csi_codec_input_start(&codec_input);
if (num != CSI_OK) {
    return -1;
}
```

csi_codec_input_stop

```
csi_error_t csi_codec_input_stop(csi_codec_input_t *ch)
```

- 功能描述：
 - 输入通道结束数据流。
- 参数：
 - `ch`：输入通道的实例句柄。
- 返回值：
 - 错误码`csi_error_t`。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
ret = csi_codec_input_stop(&codec_input);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_mute

```
csi_error_t csi_codec_input_mute(csi_codec_input_t *ch, bool en)
```

- 功能描述：
 - 设置输入通道静音。
- 参数：
 - `ch`：输入通道的实例句柄。
 - `en`：`true` 代表输入静音, `Flase` 代表输入不静音。
- 返回值：

- 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
ret = csi_codec_input_mute(&codec_input, true);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_digital_gain

```
csi_error_t csi_codec_input_digital_gain(csi_codec_input_t *ch, uint32_t val)
```

- 功能描述：
 - 设置输入通道数码增益。
- 参数：
 - ch：输入通道的实例句柄。
 - val：增益的DB值。
- 返回值：
 - 错误码csi_error_t。
- 使用示例：

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
/* 设置的数码增益 需要根据实际硬件支持的值进行设置 */
ret = csi_codec_input_digital_gain(&codec_input, 0x20);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_analog_gain

```
csi_error_t csi_codec_input_analog_gain(csi_codec_input_t *ch, uint32_t val)
```

- 功能描述：
 - 设置输入通道模拟增益。
- 参数：
 - ch：输入通道的实例句柄。

- `val` : 增益的DB值。
- 返回值 :
 - 错误码`csi_error_t`。
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
/* 设置的数码增益 需要根据实际硬件支持的值进行设置*/
ret = csi_codec_input_analog_gain(&codec_input, 0x20);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_mix_gain

```
csi_error_t csi_codec_input_mix_gain(csi_codec_input_t *ch, uint32_t val)
```

- 功能描述 :
 - 设置输入通道混频增益。
- 参数 :
 - `ch` : 输入通道的实例句柄。
 - `val` : 增益的DB值。
- 返回值 :
 - 错误码`csi_error_t`。
- 使用示例 :

```
/* 句柄使用前请先初始化 */
static csi_codec_input_t codec_input;
csi_error_t ret;
/* 设置的混频增益 需要根据实际硬件支持的值进行设置*/
ret = csi_codec_input_mix_gain(&codec_input, 0x20);
if (ret != CSI_OK) {
    return -1;
}
```

csi_codec_input_get_state

```
csi_error_t csi_codec_input_get_state(csi_codec_input_t *ch, csi_state_t *state)
```

- 功能描述 :

- 获取输出通道的状态。通过此函数来判断code通道在获取状态的时刻是否可以进行读写操作。
- 参数
 - `ch` : 实例句柄。
 - `state` : 用于返回状态信息的参数地址。
- 返回值 :
 - 错误码`csi_error_t`。

初始化及配置示例

```

/* 句柄空间一般使用静态空间 */
static csi_codec_t codec;

static csi_codec_output_t codec_output_ch;
static csi_codec_input_t codec_input_ch;

static csi_dma_ch_t dma_ch_output_handle;
static csi_dma_ch_t dma_ch_input_handle;

#define OUTPUT_BUF_SIZE 2048
uint8_t output_buf[OUTPUT_BUF_SIZE];

#define INPUT_BUF_SIZE 2048
uint8_t input_buf[INPUT_BUF_SIZE];

static ringbuffer_t output_ring_buffer;
static ringbuffer_t input_ring_buffer;

volatile uint8_t cb_output_transfer_flag = 0;
volatile uint8_t cb_input_transfer_flag = 0;

static void codec_output_event_cb_fun(csi_codec_output_t *output, csi_codec_event_t
event, void *arg)
{
    if (event == CODEC_EVENT_PERIOD_WRITE_COMPLETE) {
        cb_output_transfer_flag = 1;
    }
}

static void codec_input_event_cb_fun(csi_codec_input_t *i2s, csi_codec_event_t even
t, void *arg)
{
    if (event == CODEC_EVENT_PERIOD_READ_COMPLETE) {
        cb_input_transfer_flag = 1;
    }
}

/* 本示例 示例了codec 的输出输入流启动 */

```

```
int main(void) {
    csi_error_t ret;
    csi_codec_output_config_t output_config;
    csi_codec_input_config_t input_config;
    /* init函数的idx参数, 请根据soc的实际情况进行选择 */
    ret = csi_codec_init(&codec, 0);
    if (ret != CSI_OK) {
        return -1;
    }

    /* output ch config */
    csi_codec_output_attach_callback(&codec_output_ch, codec_output_event_cb_fun, NULL);

    codec_output_ch.period = OUTPUT_BUF_SIZE/2;
    codec_output_ch.ring_buf = &output_ring_buffer;
    csi_codec_output_open(&codec, &codec_output_ch, 0);

    output_config.bit_width = 16;
    output_config.sample_rate = 48000;
    output_config.buffer = output_buf;
    output_config.buffer_size = OUTPUT_BUF_SIZE;
    output_config.period = OUTPUT_BUF_SIZE/2;
    output_config.mode = CODEC_OUTPUT_DIFFERENCE;
    csi_codec_output_config(&codec_output_ch, &output_config);
    csi_codec_output_buffer_reset(&codec_output_ch);

    csi_codec_output_link_dma(&codec_output_ch, &dma_ch_output_handle);

    /* input ch config */
    csi_codec_input_attach_callback(&codec_input_ch, codec_input_event_cb_fun, NULL);

    codec_input_ch.period = INPUT_BUF_SIZE/2;
    codec_input_ch.ring_buf = &input_ring_buffer;
    csi_codec_input_open(&codec, &codec_input_ch, 0);

    input_config.bit_width = 16;
    input_config.sample_rate = 48000;
    input_config.buffer = input_buf;
    input_config.buffer_size = INPUT_BUF_SIZE;
    input_config.period = INPUT_BUF_SIZE/2;
    input_config.mode = CODEC_INPUT_DIFFERENCE;
    csi_codec_input_config(&codec_input_ch, &input_config);

    csi_codec_input_link_dma(&codec_input_ch, &dma_ch_input_handle);

    printf("start code input and output\n");
    csi_codec_input_start(&codec_input_ch);
    csi_codec_output_start(&codec_output_ch);
}
```

```
}
```

同步模式传输

注意：

无论是同步模式，异步模式都要求注册回调函数，DMA进行连接，启动发送流

```
/* 句柄空间一般使用静态空间 */
static csi_codec_t codec;

static csi_codec_output_t codec_output_ch;
static csi_codec_input_t codec_input_ch;

static csi_dma_ch_t dma_ch_output_handle;
static csi_dma_ch_t dma_ch_input_handle;

#define OUTPUT_BUF_SIZE 2048
uint8_t output_buf[OUTPUT_BUF_SIZE];

#define INPUT_BUF_SIZE 2048
uint8_t input_buf[INPUT_BUF_SIZE];

static ringbuffer_t output_ring_buffer;
static ringbuffer_t input_ring_buffer;

volatile uint8_t cb_output_transfer_flag = 0;
volatile uint8_t cb_input_transfer_flag = 0;

static void codec_output_event_cb_fun(csi_codec_output_t *output, csi_codec_event_t
event, void *arg)
{
    if (event == CODEC_EVENT_PERIOD_WRITE_COMPLETE) {
        cb_output_transfer_flag = 1;
    }
}

static void codec_input_event_cb_fun(csi_codec_input_t *i2s, csi_codec_event_t even
t, void *arg)
{
    if (event == CODEC_EVENT_PERIOD_READ_COMPLETE) {
        cb_input_transfer_flag = 1;
    }
}

/* 本示例 示例了codec 的同步发送和接收启动 */
int main(void) {
    uint8_t read_data[2048];
    uint8_t write_data[2048];
```

```

csi_error_t ret;
csi_codec_output_config_t output_config;
csi_codec_input_config_t input_config;
/* init函数的idx参数, 请根据soc的实际情况进行选择 */
ret = csi_codec_init(&codec, 0);
if (ret != CSI_OK) {
    return -1;
}
/* output ch config */
csi_codec_output_attach_callback(&codec_output_ch, codec_output_event_cb_fun, NULL);

codec_output_ch.period = OUTPUT_BUF_SIZE/2;
codec_output_ch.ring_buf = &output_ring_buffer;
csi_codec_output_open(&codec, &codec_output_ch, 0);

output_config.bit_width = 16;
output_config.sample_rate = 48000;
output_config.buffer = output_buf;
output_config.buffer_size = OUTPUT_BUF_SIZE;
output_config.period = OUTPUT_BUF_SIZE/2;
output_config.mode = CODEC_OUTPUT_DIFFERENCE;
csi_codec_output_config(&codec_output_ch, &output_config);
csi_codec_output_buffer_reset(&codec_output_ch);

csi_codec_output_link_dma(&codec_output_ch, &dma_ch_output_handle);

/* input ch config */
csi_codec_input_attach_callback(&codec_input_ch, codec_input_event_cb_fun, NULL);

codec_input_ch.period = INPUT_BUF_SIZE/2;
codec_input_ch.ring_buf = &input_ring_buffer;
csi_codec_input_open(&codec, &codec_input_ch, 0);

input_config.bit_width = 16;
input_config.sample_rate = 48000;
input_config.buffer = input_buf;
input_config.buffer_size = INPUT_BUF_SIZE;
input_config.period = INPUT_BUF_SIZE/2;
input_config.mode = CODEC_INPUT_DIFFERENCE;
csi_codec_input_config(&codec_input_ch, &input_config);

csi_codec_input_link_dma(&codec_input_ch, &dma_ch_input_handle);

printf("start code input and output\n");
csi_codec_input_start(&codec_input_ch);
csi_codec_output_start(&codec_output_ch);
uint32_t read_size = 0;
read_size = csi_codec_input_receive(&codec_input_ch, read_data, sizeof(read_data));

```

```

    if (read_size == sizeof(read_data)) {
        printf("test_code_receive_sync successfully\n");
    } else {
        printf("test_code_receive_sync failed\n");
    }
    uint32_t write_size = 0;
    write_size = csi_codec_output_send(&codec_output_ch, write_data, sizeof(write_data));

    if (write_size == sizeof(write_data)) {
        printf("test_code_send_sync successfully\n");
    } else {
        printf("test_code_send_sync failed\n");
    }
    csi_codec_input_stop(&codec_input_ch);
    csi_codec_output_stop(&codec_output_ch);

    csi_codec_input_link_dma(&codec_input_ch, NULL);
    csi_codec_output_link_dma(&codec_output_ch, NULL);

    csi_codec_output_detach_callback(&codec_output_ch);
    csi_codec_input_detach_callback(&codec_input_ch);

    csi_codec_uninit(&codec);
}

```

异步模式传输

注意：

无论是同步模式，异步模式都要求注册回调函数，DMA进行连接，启动发送流

```

/* 句柄空间一般使用静态空间 */
static csi_codec_t codec;

static csi_codec_output_t codec_output_ch;
static csi_codec_input_t codec_input_ch;

static csi_dma_ch_t dma_ch_output_handle;
static csi_dma_ch_t dma_ch_input_handle;

#define OUTPUT_BUF_SIZE 2048
uint8_t output_buf[OUTPUT_BUF_SIZE];

#define INPUT_BUF_SIZE 2048
uint8_t input_buf[INPUT_BUF_SIZE];

static ringbuffer_t output_ring_buffer;

```

```

static ringbuffer_t input_ring_buffer;

volatile uint8_t cb_output_transfer_flag = 0;
volatile uint8_t cb_input_transfer_flag = 0;

static void codec_output_event_cb_fun(csi_codec_output_t *output, csi_codec_event_t
event, void *arg)
{
    if (event == CODEC_EVENT_PERIOD_WRITE_COMPLETE) {
        cb_output_transfer_flag --;
    }
}

static void codec_input_event_cb_fun(csi_codec_input_t *i2s, csi_codec_event_t even
t, void *arg)
{
    if (event == CODEC_EVENT_PERIOD_READ_COMPLETE) {
        cb_input_transfer_flag = 1;
    }
}

/* 本示例 示例了codec 的异步发送和接收启动 */
int main(void) {
    csi_error_t ret;
    csi_codec_output_config_t output_config;
    csi_codec_input_config_t input_config;
    /* init函数的idx参数, 请根据soc的实际情况进行选择 */
    ret = csi_codec_init(&codec, 0);
    if (ret != CSI_OK) {
        return -1;
    }

    /* output ch config */
    csi_codec_output_attach_callback(&codec_output_ch, codec_output_event_cb_fun, N
ULL);

    codec_output_ch.period = OUTPUT_BUF_SIZE/2;
    codec_output_ch.ring_buf = &output_ring_buffer;
    csi_codec_output_open(&codec, &codec_output_ch, 0);

    output_config.bit_width = 16;
    output_config.sample_rate = 48000;
    output_config.buffer = output_buf;
    output_config.buffer_size = OUTPUT_BUF_SIZE;
    output_config.period = OUTPUT_BUF_SIZE/2;
    output_config.mode = CODEC_OUTPUT_DIFFERENCE;
    csi_codec_output_config(&codec_output_ch, &output_config);
    csi_codec_output_buffer_reset(&codec_output_ch);

    csi_codec_output_link_dma(&codec_output_ch, &dma_ch_output_handle);

```

```

/* input ch config */
csi_codec_input_attach_callback(&codec_input_ch, codec_input_event_cb_fun, NULL
);

codec_input_ch.period = INPUT_BUF_SIZE/2;
codec_input_ch.ring_buf = &input_ring_buffer;
csi_codec_input_open(&codec, &codec_input_ch, 0);

input_config.bit_width = 16;
input_config.sample_rate = 48000;
input_config.buffer = input_buf;
input_config.buffer_size = INPUT_BUF_SIZE;
input_config.period = INPUT_BUF_SIZE/2;
input_config.mode = CODEC_INPUT_DIFFERENCE;
csi_codec_input_config(&codec_input_ch,&input_config);

csi_codec_input_link_dma(&codec_input_ch,&dma_ch_input_handle);

printf("start code input and output\n");
csi_codec_input_start(&codec_input_ch);
csi_codec_output_start(&codec_output_ch);

for (uint32_t i = 0; i < sizeof(write_data); i++) {
    write_data[i] = 0x5a;
}

uint32_t write_size = 0;
cb_output_transfer_flag = sizeof(write_data) / g_i2s->tx_period;///< 请确保发送数
据是period的整数倍
write_size = csi_codec_output_write_async(&codec_output_ch, write_data, sizeof(
write_data)); ///<等待数据发送
while(cb_output_transfer_flag != 0); ///<等待同步
if (write_size == sizeof(write_data)) {
    printf("test_codec_send_async successfully\n");
} else {
    printf("test_codec_send_async failed\n");
}

uint32_t read_size = 0;
uint32_t period_num = sizeof(read_data) / i2s_slave.rx_period;///< 请确保数据是pe
riod的整数倍
cb_slave_transfer_flag = 0;
while (1) {
    if (cb_slave_transfer_flag == 1) {
        cb_slave_transfer_flag = 0;
        read_size += csi_codec_input_receive_async(&codec_input_ch, (read_data
+ read_size), (sizeof(read_data) - read_size));///<接收完注意：偏移接收指针
        period_num --;

        if (period_num == 0) {
            break;

```



```
        }  
    }  
}  
  
if (read_size == sizeof(read_data)) {  
    printf("test_i2s successfully\n");  
} else {  
    printf("test_i2s failed\n");  
}  
  
csi_codec_input_stop(&codec_input_ch);  
csi_codec_output_stop(&codec_output_ch);  
  
csi_codec_input_link_dma(&codec_input_ch, NULL);  
csi_codec_output_link_dma(&codec_output_ch, NULL);  
  
csi_codec_output_detach_callback(&codec_output_ch);  
csi_codec_input_detach_callback(&codec_input_ch);  
  
csi_codec_uninit(&codec);  
}
```

ETB

简要说明

ETB的功能是通过发送信息的方式，是一个IP触发另一个IP，并且释放CPU。ETB会管理所有触发事件。

接口描述

csi_etb_init

```
csi_error_t csi_etb_init(void)
```

- 功能描述:
 - 初始化ETB设备，并使能ETB。
 - 返回值:
 - 错误码。
-

csi_etb_uninit

```
void csi_etb_uninit(void)
```

- 功能描述:
 - 去初始化ETB设备，并关闭ETB。
-

csi_etb_ch_type_t

类型	定义
ETB_CH_ONE_TRIGGER_ONE	单IP触发单IP的通道类型
ETB_CH_ONE_TRIGGER_MORE	单IP触发多IP的通道类型
ETB_CH_MORE_TRIGGER_ONE	多IP触发单IP的通道类型

csi_etb_ch_alloc

```
int32_t csi_etb_ch_alloc(csi_etb_ch_type_t ch_type)
```

- 功能描述:
 - 根据传入的通道类型，申请一个空闲通道，并返回通道号。
- 参数:
 - `ch_type` : 通道工作类型。
- 返回值:
 - 成功返回通道号，失败-1。

csi_etb_ch_free

```
void csi_etb_ch_free(int32_t ch_id)
```

- 功能描述:
 - 通过传入的通道号，释放对应的通道。
- 参数:
 - `ch_id` : 通道号。

csi_etb_trig_mode_t

类型	说明
ETB_HARDWARE_TRIG	硬件触发
ETB_SOFTWARE_TRIG	软件触发

csi_etb_config_t

成员	类型	说明
src_ip	uint8_t	源IP设备号
dst_ip	uint8_t	目的IP设备号
trig_mode	csi_etb_trig_mode_t	触发模式
ch_type	csi_etb_ch_type_t	通道类型

csi_etb_ch_config

```
int32_t csi_etb_ch_config(int32_t ch_id, csi_etb_config_t *config)
```

- 功能描述:
 - 配置通道工作模式。
 - 参数:
 - `ch_id` : 通道号。
 - `config` : 通道配置。
 - 返回值:
 - 错误码。
-

csi_etb_ch_start

```
void csi_etb_ch_start(int32_t ch_id)
```

- 功能描述:
 - 使能通道功能。
 - 参数:
 - `ch_id` : 通道号。
-

csi_etb_ch_stop

```
void csi_etb_ch_stop(int32_t ch_id)
```

- 功能描述:
 - 关闭通道使能。
 - 参数:
 - `ch_id` : 通道号。
-

QSPI

简要说明

QSPI 是 Quad SPI 的简写，是 Motorola 公司推出的 SPI 接口的扩展，比 SPI 应用更加广泛。在 SPI 协议的基础上，Motorola 公司对其功能进行了增强，增加了队列传输机制，推出了队列串行外围接口协议（即 QSPI 协议）。使用该接口，用户可以一次性传输包含多达 16 个 8 位或 16 位数据的传输队列。一旦传输启动，直到传输结束，都不需要 CPU 干预，极大的提高了传输效率。与 SPI 相比，QSPI 的最大结构特点是以 80 字节的 RAM 代替了 SPI 的发送和接收数据寄存器。

接口列表

QSPI的CSI接口说明如下所示：

函数	说明
csi_qspi_init	QSPI初始化
csi_qspi_uninit	QSPI反初始化
csi_qspi_attach_callback	注册回调函数
csi_qspi_detach_callback	注销回调函数
csi_qspi_frequence	配置QSPI频率
csi_qspi_mode	配置QSPI时钟模式
csi_qspi_send	发送数据（同步模式）
csi_qspi_receive	接收数据（同步模式）
csi_qspi_send_receive	发送接收数据（同步模式）
csi_qspi_send_async	发送数据（异步模式）
csi_qspi_receive_async	接收数据（异步模式）
csi_qspi_send_receive_async	发送接收数据（异步模式）
csi_qspi_link_dma	绑定/注销DMA通道
csi_qspi_get_state	获取QSPI状态
csi_qspi_memory_mapped	配置内存映射

接口详细说明

csi_qspi_init

```
csi_error_t csi_qspi_init(csi_qspi_t *qspi, uint32_t idx)
```

- 功能描述:

通过设备ID初始化对应的QSPI实例，返回结果值。

- 参数:

- `qspi` : 设备句柄（需要用户申请句柄空间）。
- `idx` : 设备ID。

- 返回值:

- 错误码 `csi_error_t`

csi_qspi_t

成员	类型	说明
dev	csi_dev_t	设备统一句柄
cb	void (callback)(csi_qspi_t qspi, csi_qspi_event_t event, void *arg)	用户回调函数
arg	void*	用户回调函数对应的传参
tx_data	void*	指向发送缓存的地址
tx_size	uint32_t	发送数据的大小
rx_data	void*	指向接收缓存的地址
rx_size	uint32_t	接收缓存的大小
send	void*	指向发送函数(异步)
receive	void*	指向接收函数(异步)
send_receive	void*	指向发送接收函数(异步)
state	csi_state_t	运行状态
tx_dma	csi_dma_ch_t*	指向发送DMA句柄
rx_dma	csi_dma_ch_t*	指向接收DMA句柄
priv	void*	设备私有变量

csi_qspi_uninit

```
void csi_qspi_uninit(csi_qspi_t *qspi)
```

- 功能描述:
- QSPI实例反初始化, 并且释放相关的软硬件资源。
- 参数:
 - `qspi` : 实例句柄。
- 返回值
 - 无

csi_qspi_attach_callback

```
csi_error_t csi_qspi_attach_callback(csi_qspi_t *qspi, void *callback, void *arg)
```

- 功能描述:
 - 注册回调函数到指定QSPI控制器。
- 参数:
 - `qspi` : 实例句柄。
 - `callback` : 回调函数。
 - `arg` : 回调函数的参数。
- 参数:
 - 错误码 `csi_error_t`

csi_qspi_detach_callback

```
void csi_qspi_detach_callback(csi_qspi_t *qspi)
```

- 功能描述:
 - 注销回调函数。
- 参数:
 - `qspi` : 实例句柄。

csi_qspi_frequence

```
uint32_t csi_qspi_frequence(csi_qspi_t *qspi, uint32_t hz)
```

- 功能描述:
- 设置QSPI频率。
- 参数:

- `qspi` : 实例句柄。
 - `hz` : 工作频率。
- 返回值
 - 实际设置频率

csi_qspi_mode

```
csi_error_t csi_qspi_mode(csi_qspi_t *qspi, csi_qspi_mode_t mode)
```

- 功能描述:
- 设置QSPI时钟模式。
- 参数:
 - `qspi` : 实例句柄。
 - `mode` : 时钟模式。
- 返回值
 - 错误码`csi_error_t`

csi_qspi_clock_mode_t

类型	说明
QSPI_CLOCK_MODE_0	模式0: CPOL = 0, CPHA = 0
QSPI_CLOCK_MODE_3	模式3: CPOL = 1, CPHA = 1

csi_qspi_send

```
int32_t csi_qspi_send(csi_qspi_t *qspi, csi_qspi_command_t *cmd, const void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
- 以轮询模式进行数据发送。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。
 - `data` : 指向发送数据缓存。
 - `size` : 指定需要发送的数据长度。
 - `timeout` : 发送超时时间，单位ms。
- 返回值

- 发送成功，返回实际发送长度。 发送失败，返回错误码。
- 错误码

csi_qspi_command_t

成员	类型	说明
instruction.bus_width	csi_qspi_bus_width_t	指令阶段总线线数
instruction.value	uint8_t	指令值
instruction.disabled	bool	指令使能/禁能
address.bus_width	csi_qspi_bus_width_t	地址阶段总线线数
address.size	csi_qspi_address_size_t	地址字节数
address.value	uint32_t	地址值
address.disabled	bool	地址使能/禁能
alt.bus_width	csi_qspi_bus_width_t	交替字节总线数
alt.size	csi_qspi_alt_size_t	交替字节字节数
alt.value	uint32_t	交替字节数值
alt.disabled	bool	交替字节使能/禁能
dummy_count	uint8_t	dummy数量
data.bus_width	csi_qspi_bus_width_t	data总线数

csi_qspi_alt_size_t

类型	说明
QSPI_ALTERNATE_BYTES_8_BITS	1字节
QSPI_ALTERNATE_BYTES_16_BITS	2字节
QSPI_ALTERNATE_BYTES_24_BITS	3字节
QSPI_ALTERNATE_BYTES_32_BITS	4字节

csi_qspi_bus_width_t

类型	说明
QSPI_CFG_BUS_SINGLE	单线
QSPI_CFG_BUS_DUAL	双线
QSPI_CFG_BUS_QUAD	四线

QSPI_CFG_BUS_QUAD	四线
-------------------	----

csi_qspi_address_size_t

类型	说明
QSPI_ADDRESS_8_BITS	8比特
QSPI_ADDRESS_16_BITS	16比特
QSPI_ADDRESS_24_BITS	24比特
QSPI_ADDRESS_32_BITS	32比特

csi_qspi_receive

```
uint32_t csi_qspi_receive(csi_qspi_t *qspi, csi_qspi_command_t *cmd, void *data, uint32_t size, uint32_t timeout)
```

- 功能描述:
- 以轮询模式进行数据接收。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。
 - `data` : 指向接收数据缓存。
 - `size` : 指定需要接收的数据长度。
 - `timeout` : 数据接收超时时间, 单位ms。
- 返回值
 - 错误码csi_error_t

csi_qspi_send_receive

```
uint32_t csi_qspi_send_receive(csi_qspi_t *qspi, csi_qspi_command_t *cmd, const void *tx_data, void *rx_data, uint32_t size, uint32_t timeout)
```

- 功能描述:
- 以轮询模式进行数据发送/接收。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。

- `tx_data` : 指向发送数据缓存。
- `rx_data` : 指向接收数据缓存。
- `size` : 指定发送/接收的数据长度。
- `timeout` : 数据接收超时时间, 单位ms。
- 返回值
 - 错误码`csi_error_t`

`csi_qspi_send_async`

```
csi_error_t csi_qspi_send_async(csi_qspi_t *qspi, csi_qspi_command_t *cmd, const void *data, uint32_t size)
```

- 功能描述:
- 以异步模式进行数据发送。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。
 - `data` : 指向发送数据缓存。
 - `size` : 指定需要发送的数据长度。
- 返回值
 - 错误码`csi_error_t`

`csi_qspi_receive_async`

```
csi_error_t csi_qspi_receive_async(csi_qspi_t *qspi, csi_qspi_command_t *cmd, void *data, uint32_t size)
```

- 功能描述:
- 以异步模式进行数据接收。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。
 - `data` : 指向数据数据缓存。
 - `size` : 指定需要接收的数据长度。
- 返回值
 - 错误码`csi_error_t`

`csi_qspi_send_receive_async`

```
csi_error_t csi_qspi_send_receive_async(csi_qspi_t *qspi, csi_qspi_command_t *cmd,
const void *tx_data, void *rx_data, uint32_t size)
```

- 功能描述:
- 以异步模式进行数据发送/接收。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。
 - `tx_data` : 指向发送数据缓存。
 - `rx_data` : 指向接收数据缓存。
 - `size` : 指定发送接收的数据长度。
- 返回值
 - 错误码`csi_error_t`

csi_qspi_link_dma

```
csi_error_t csi_qspi_link_dma(csi_qspi_t *qspi, csi_dma_ch_t *tx_dma, csi_dma_ch_t
*rx_dma)
```

- 功能描述:
- 绑定/注销DMA通道。当传入参数为NULL时注销通道，当参数不为NULL时绑定通道。
- 参数:
 - `qspi` : 实例句柄。
 - `tx_dma` : 指向发送dma通道。
 - `rx_dma` : 指向接收dma通道。
- 返回值:
 - 错误码`csi_error_t`

csi_qspi_get_state

```
csi_error_t csi_qspi_get_state(csi_qspi_t *qspi, csi_state_t *state)
```

- 功能描述:
- 获取QSPI状态。
- 参数:
 - `qspi` : 实例句柄。

- `state` : 指向接收的状态值。
- 返回值:
 - 错误码`csi_error_t`

csi_state_t

类型	说明
readable	设备可读
writable	设备可写
error	错误状态

csi_qspi_memory_mapped

```
csi_error_t csi_qspi_memory_mapped(csi_qspi_t *qspi, csi_qspi_command_t *cmd)
```

- 功能描述:
- 设置QSPI内存映射模式。
- 参数:
 - `qspi` : 实例句柄。
 - `cmd` : 指向command配置信息。
- 返回值:
 - 错误码`csi_error_t`

使用示例

```
示例展示了如何使用QSPI读取SPIFLASH的JEDEC ID

#include <stdio.h>
#include <string.h>

#include <soc.h>
#include <drv/qspi.h>
#include <drv/tick.h>
#include <csi_config.h>
#include <board_config.h>
#include <board_init.h>

#define W25Q64FV_READ_JEDEC_ID          0x9F

static csi_qspi_t    qspi_handle;
```

```
static csi_qspi_command_t  command;
int main(void)
{
    int ret;
    uint8_t device_id[3];

    board_init();

    ret = csi_qspi_init(&qspi_handle, 0);

    if (ret != CSI_OK) {
        return -1;
    }

    /* Read device id operations */
    command.instruction.value      = W25Q64FV_READ_DEVICE_ID;
    command.instruction.bus_width = QSPI_CFG_BUS_SINGLE;
    command.instruction.disabled  = false;
    command.alt.disabled          = true;
    command.address.disabled      = true;
    command.address.value         = 0;
    command.address.size          = 0;
    command.address.bus_width     = QSPI_CFG_BUS_SINGLE;
    command.data.bus_width        = QSPI_CFG_BUS_SINGLE;
    ret = csi_qspi_receive(&qspi_handle, &command, device_id, 3, 1000);
    if (ret != 3) {
        return -1;
    }

    printf("device id: %x%x%x", device_id[0], device_id[1], device_id[2]);
    return 0;
}
```

PIN

说明

在 SOC 设计中，IOCTL（IO controller）作为 APB 的外设，提供了 IO cell 和外设交换数据的通道，除此之外，还可以配置输入输出的方向，IO cell 复用功能（ALT）选择，驱动能力大小，上拉/下拉和其他 IO cell 的属性 功能 通过配置管脚复用控制寄存器，来选择管脚作为其复用的一种功能，还可以调整其他属性如上拉（或 下拉），驱动能力增强等，IOCTL 模块具有以下特点 连接 IO cell 和外设支持功能模式或者测试模式 支持 IO CELL 的 GPIO 功能或者其他复用（ALT）功能 可配置 IO cell 其他属性，如驱动能力，上下拉使能 支持 test mode 的特殊使用

接口列表

PIN的CSI接口说明如下所示：

函数	说明
csi_pin_set_mux	设置PIN的复用功能
csi_pin_get_mux	获取PIN的复用功能
csi_pin_mode	设置PIN的模式
csi_pin_speed	设置PIN的速度
csi_pin_drive	设置PIN的驱动能力
csi_pin_get_gpio_devidx	通过pin name找GPIO端口号
csi_pin_get_uart_devidx	通过pin name查找UART设备号
csi_pin_get_iic_devidx	通过pin name查找IIC设备号
csi_pin_get_spi_devidx	通过pin name查找SPI设备号
csi_pin_get_i2s_devidx	通过pin name查找I2S设备号
csi_pin_get_gpio_channel	通过pin name获取通道号
csi_pin_get_pwm_channel	通过pin name获取PWM通道
csi_pin_get_adc_channel	通过pin name获取ADC通道
csi_pin_get_pinname_by_gpio	通过PIN端口号与通道号获取pin name

接口详细说明

csi_pin_set_mux

```
csi_error_t csi_pin_set_mux(pin_name_t pin_name, pin_func_t pin_func)
```

- 功能描述:
 - 设置PIN为复用功能。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。枚举定义详见pin_name。
 - pin_func: 复用功能编号,每一款芯片都有自己的复用编号。枚举定义详见pin_func。
- 返回值:
 - 错误码csi_error_t。
- 使用示例：

```
/* 将PA2引脚设置为UART0发送复用功能 */
csi_error_t ret;
ret = csi_pin_set_mux(PA2, PA2_UART0_TX);
if (ret != CSI_OK) {
    return -1;
}
```

pin_name (例)

定义	值
PA0	0
PA1	1
PA2	2
PA3	3
PA4	4
PA5	5
PB0	6
PB1	7
PB2	8
PB3	9
PA6	10
PA7	11
PA8	12
PA9	13

PA10	14
PA11	15
PA12	16
PA13	17
PA14	18
PA15	19
PA16	20
PA17	21
PA18	22
PA19	23
PA20	24
PA21	25
PA22	26
PA23	27
PA24	28
PA25	29
PA26	30
PA27	31
PC0	32
PC1	33

pin_func (例)

定义	值
PA0_ETB_TRIG0	0
PA0_JTAG_TCK	2
PA1_ETB_TRIG1	0
PA1_JTAG_TMS	2
PA2_SPI0_MISO	2
PA2_UART0_SIROUT	3
.....
PC1_PWM_CH11	2
PC1_ADC_A15	3
PIN_FUNC_GPIO	4

--	--

csi_pin_get_mux

```
pin_func_t csi_pin_get_mux(pin_name_t pin_name)
```

- 功能描述:
 - 获取PIN的复用功能。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - 复用功能编号。
- 使用示例：

```
/* 获取PA2引脚复用功能*/
pin_func_t pin_func;
pin_func = csi_pin_get_mux(PA2);
return pin_func;
```

csi_pin_mode

```
csi_error_t csi_pin_mode(pin_name_t pin_name, csi_pin_mode_t mode)
```

- 功能描述:
 - 设置PIN的模式。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
 - mode：工作模式，枚举定义详见csi_pin_mode_t。
- 返回值:
 - 错误码csi_error_t。

csi_pin_mode_t

类型	说明
GPIO_MODE_PULLNONE	悬空输入
GPIO_MODE_PULLUP	上拉输入
GPIO_MODE_PULLDOWN	下拉输入
GPIO_MODE_OPEN_DRAIN	开漏输出
GPIO_MODE_PUSH_PULL	推挽输出

csi_pin_speed

```
csi_error_t csi_pin_speed(pin_name_t pin_name, csi_pin_speed_t speed)
```

- 功能描述:
 - 设置PIN的速度。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
 - speed：响应速度，枚举定义详见csi_pin_speed_t。
- 返回值:
 - 错误码csi_error_t。

csi_pin_speed_t（0级为最低速，依次类推）

类型	说明
PIN_SPEED_LV0	0级速度
PIN_SPEED_LV1	1级速度
PIN_SPEED_LV2	2级速度
PIN_SPEED_LV3	3级速度

csi_pin_drive

```
csi_error_t csi_pin_drive(pin_name_t pin_name, csi_pin_drive_t drive)
```

- 功能描述:
 - 设置PIN的驱动能力。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
 - drive：驱动能力，枚举定义详见csi_pin_drive_t。
- 返回值:
 - 错误码csi_error_t。

csi_pin_drive_t（0级驱动能力最弱，依次类推）

类型	说明
PIN_DRIVE_LV0	0级驱动能力
PIN_DRIVE_LV1	1级驱动能力
PIN_DRIVE_LV2	2级驱动能力

PIN_DRIVE_LV3	3级驱动能力
---------------	--------

csi_pin_get_gpio_devidx

```
uint32_t csi_pin_get_gpio_devidx(pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name查找GPIO端口号。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - GPIO端口号。（PORTA为0，PORTB为1，依次类推，失败返回0xFFFFFFFFU）。

csi_pin_get_uart_devidx

```
uint32_t csi_pin_get_uart_devidx(pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name查找UART设备号。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - UART设备号。（UART0为0，UART1为1，依次类推，失败返回0xFFFFFFFFU）。

csi_pin_get_iic_devidx

```
uint32_t csi_pin_get_iic_devidx(pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name查找IIC设备号。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - IIC设备号。（IIC0为0，IIC1为1，依次类推，失败返回0xFFFFFFFFU）。

csi_pin_get_spi_devidx

```
uint32_t csi_pin_get_spi_devidx(pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name查找SPI设备号。

- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - SPI设备号。(SPI0为0，SPI1为1，依次类推，失败返回0xFFFFFFFFFU)。

csi_pin_get_i2s_devidx

```
uint32_t csi_pin_get_i2s_devidx(pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name查找I2S设备号。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - I2S设备号。(I2S0为0，I2S1为1，依次类推，失败返回0xFFFFFFFFFU)。

csi_pin_get_gpio_channel

```
uint32_t csi_pin_get_gpio_channel (pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name获取GPIO通道号。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - GPIO通道号。(PA0为0，PA1为1，依次类推，失败返回0xFFFFFFFFFU)。

csi_pin_get_pwm_channel

```
uint32_t csi_pin_get_pwm_channel (pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name获取PWM通道。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - PWM通道。(PWM_CH0为0，PWM_CH1为1，依次类推，失败返回0xFFFFFFFFFU)。

csi_pin_get_adc_channel

```
uint32_t csi_pin_get_adc_channel (pin_name_t pin_name)
```

- 功能描述:
 - 通过pin name获取ADC通道。
- 参数:
 - pin_name: pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。
- 返回值:
 - ADC通道。(ADC_A0为0，ADC_A1为1，依次类推，失败返回0xFFFFFFFFU)。

csi_pin_get_pinname_by_gpio

```
pin_name_t csi_pin_get_pinname_by_gpio(uint8_t gpio_idx, uint8_t channel)
```

- 功能描述:
 - 通过PIN端口号与通道号获取pin name。
- 参数:
 - gpio_idx:GPIO端口号。
 - channel:GPIO通道号。
- 返回值:
 - pin name，每一款芯片都有自己的pin name，与芯片手册pin name一一对应。

csi_pin_uart_t枚举说明

- 功能描述:
 - 用于查找UART设备具体的某一个引脚。

类型	说明
PIN_UART_TX	UART发送引脚
PIN_UART_RX	UART接收引脚
PIN_UART_CTS	UART CTS引脚
PIN_UART_RTS	UART RTS引脚

csi_pin_iic_t枚举说明

- 功能描述:
 - 用于查找iic设备具体的某一个引脚。

类型	说明
PIN_IIC_SCL	IIC时钟引脚
PIN_IIC_SDA	IIC数据引脚

csi_pin_spi_t枚举说明

- 功能描述:
 - 用于查找spi设备具体的某一个引脚。

类型	说明
PIN_SPI_MISO	SPI MISO引脚
PIN_SPI_MOSI	SPI MOSI引脚
PIN_SPI_SCK	SPI时钟引脚
PIN_SPI_CS	SPI片选引脚

csi_pin_i2s_t枚举说明

- 功能描述:
 - 用于查找I2S设备具体的某一个引脚。

类型	说明
PIN_I2S_MCLK	I2S MCLK引脚
PIN_I2S_SCLK	I2S SCLK引脚
PIN_I2S_WSCLK	I2S WSCLK引脚
PIN_I2S_SDA	I2S数据引脚

示例

将PA1引脚设置为GPIO功能、上拉输入、低速响应、低驱动能力。

```
int main(void){
    csi_error_t ret;
    ret = csi_pin_set_mux(PA1, PIN_FUNC_GPIO);
    if (ret != CSI_OK) {
        return -1;
    }
    ret = csi_pin_set_mode(PA1, GPIO_MODE_PULLUP);
    if (ret != CSI_OK) {
        return -1;
    }
    ret = csi_pin_speed(PA1, PIN_SPEED_LV0);
    if (ret != CSI_OK) {
        return -1;
    }
    ret = csi_pin_drive(PA1, PIN_DRIVE_LV0);
    if (ret != CSI_OK) {
        return -1;
    }
}
```

```
    return 1;  
}
```


PM

设备说明

PM（power manager）负责统一管理整个SOC芯片的低功耗行为。控制各个驱动设备在进入/退出低功耗模式时执行对应的操作。

接口列表

PIN的CSI接口说明如下所示：

函数	说明
csi_pm_init	power manager初始化
csi_pm_uninit	power manager去初始化
csi_pm_set_reten_mem	设置retention内存空间
csi_pm_config_wakeup_source	配置低功耗模式的唤醒源
csi_pm_enter_sleep	进入低功耗模式
csi_pm_dev_register	注册设备到低功耗管理模块(驱动设备调用)
csi_pm_dev_unregister	从低功耗管理模块取消初始化(驱动设备调用)
csi_pm_dev_save_regs	保存寄存器到内存中(驱动设备调用)
csi_pm_dev_restore_regs	从内存中恢复到寄存器中(驱动设备调用)
csi_pm_dev_notify	执行驱动设备回调函数
soc_pm_enter_sleep	进入低功耗模式(实现在soc的pmu驱动中)
soc_pm_config_wakeup_source	配置低功耗模式的唤醒源(实现在soc的pmu驱动中)

接口详细说明

csi_pm_init

csi_error_t csi_pm_init(void)

- 功能描述:
 - 功耗管理模块初始化
- 参数:
 - 无
- 返回值:

- CSI_OK: 设置成功。
- CSI_ERROR: 设置失败。
- CSI_UNSUPPORTED: 参数错误。

csi_pm_uninit

void csi_pm_uninit(void)

- 功能描述:
 - 功耗管理模块去初始化
- 参数:
 - 无
- 返回值:
 - 无

csi_pm_set_reten_mem

csi_error_t csi_pm_set_reten_mem(uint32_t *mem, uint32_t num)

- 功能描述:
 - 设置保留内存的区域，用于保存在掉电模式下相关IP寄存器的内容。
- 参数:
 - mem：以字为单位
 - num：指定内存大小，以字为单位
- 返回值:
 - CSI_OK: 设置成功。
 - CSI_ERROR: 设置失败。
 - CSI_UNSUPPORTED: 参数错误。

csi_pm_config_wakeup_source

csi_error_t csi_pm_config_wakeup_source(uint32_t wakeup_num, bool enable)

- 功能描述:
 - 配置唤醒源
- 参数:
 - wakeup_num: 唤醒号
 - enable: 是否使能唤醒功能
- 返回值:
 - CSI_OK: 配置成功。
 - CSI_ERROR: 配置失败。
 - CSI_UNSUPPORTED: 模式不支持。

csi_pm_enter_sleep

csi_error_t csi_pm_enter_sleep(csi_pm_mode_t mode)

- 功能描述:
 - 进入低功耗模式
- 参数:
 - mode: 低功耗模式
- 返回值:
 - CSI_OK: 配置成功。
 - CSI_ERROR: 配置失败。
 - CSI_UNSUPPORTED: 模式不支持。

csi_pm_mode_t

类型	说明
PM_MODE_RUN	运行模式
PM_MODE_SLEEP_1	一级浅睡眠模式
PM_MODE_SLEEP_2	二级浅睡眠模式
PM_MODE_DEEP_SLEEP_1	一级深度睡眠模式
PM_MODE_DEEP_SLEEP_2	二级深度睡眠模式

csi_pm_dev_register

csi_error_t csi_pm_dev_register(csi_dev_t dev, void pm_action, uint32_t mem_size, uint8_t priority)

- 功能描述:
 - 驱动设备通过调用该接口将驱动设备在掉电模式下所需空间、执行的回调函数及其优先级注册到低功耗管理模块中
- 参数:
 - dev: 目标设备的句柄
 - pm_action: 进入低功耗模式前/退出低功耗模式后，执行的设备回调函数
 - mem_size: 目标设备在掉电模式下所需的保存空间
 - priority: 执行设备回调函数的优先级（值越小，进入低功耗模式前越晚调用/退出低功耗模式后越先执行）
- 返回值:
 - CSI_OK: 配置成功。
 - CSI_ERROR: 配置失败。
 - CSI_UNSUPPORTED: 模式不支持。

csi_dev_t

成员	类型	说明
reg_base	uint32_t	寄存器基址
irq_num	uint8_t	中断号
idx	uint8_t	设备号
dev_tag	uint16_t	设备的tag
irq_handler	void ()(void)	中断回调函数
pm_dev	csi_pm_dev_t	低功耗管理模块的设备

csi_pm_dev_t

成员	类型	说明
next	slist_t	寄存器基址
pm_action	csi_error_t ()(csi_dev_t dev, csi_pm_dev_action_t action)	低功耗设备的回调函数
reten_mem	uint32_t *	驱动设备保存的区域地址
size	uint32_t	驱动设备需要保存的大小

csi_pm_dev_unregister

uint32_t csi_pm_dev_unregister(csi_dev_t *dev)

- 功能描述:
 - 从低功耗管理模块中取消注册目标设备
- 参数:
 - dev: 目标设备的句柄
- 返回值:
 - 无

csi_pm_dev_save_regs

void csi_pm_dev_save_regs(uint32_t mem, uint32_t addr, uint32_t num)

- 功能描述:
 - 保存指定个数的寄存器内容到内存
- 参数:
 - mem: 内存地址
 - addr: 寄存器地址
 - num: 寄存器个数
- 返回值:
 - 无

csi_pm_dev_restore_regs

void csi_pm_dev_restore_regs(uint32_t mem, uint32_t addr, uint32_t num)

- 功能描述:
 - 从内存恢复指定个数的寄存器
- 参数:
 - mem: 内存地址
 - addr: 寄存器地址
 - num: 寄存器个数
- 返回值:
 - 无

csi_pm_dev_notify

csi_error_t csi_pm_dev_notify(csi_pm_dev_action_t action)

- 功能描述:
 - 通知注册到低功耗管理模块的设备，执行进入/退出低功耗模式所对应的回调函数
- 参数:
 - action: 通知回调函数的命令
- 返回值:
 - CSI_OK: 配置成功。
 - CSI_ERROR: 配置失败。
 - CSI_UNSUPPORTED: 模式不支持。

soc_pm_enter_sleep

csi_error_t soc_pm_enter_sleep(csi_pm_mode_t mode)

- 功能描述:
 - 进入低功耗模式，对接csi_pm_enter_sleep接口，实现在各个soc的pmu中
- 参数:
 - mode: 低功耗模式
- 返回值:
 - CSI_OK: 配置成功。
 - CSI_ERROR: 配置失败。
 - CSI_UNSUPPORTED: 模式不支持。

soc_pm_config_wakeup_source

csi_error_t csi_pm_config_wakeup_source(uint32_t wakeup_num, bool enable)

- 功能描述:
 - 配置唤醒源
- 参数:
 - wakeup_num: 唤醒号

- enable: 是否使能唤醒功能
- 返回值:
 - CSI_OK: 配置成功。
 - CSI_ERROR: 配置失败。
 - CSI_UNSUPPORTED: 模式不支持。