S9070x_ SDK_API_Reference

Document Revision: V1.1

Document Release: 2019/04/20

SmartChip Integration Inc.

9B, Science Plaza, International
Science Park,1355 Jinjihu Avenue,
Suzhou Industrial Park, Suzhou,
Jiangsu, China.
ZIP: 215021
Telephone: +86-512-62620006
Fax: +86-512-62620002
E-mail: sales@sci-inc.com.cn
Website: http://www.sci-inc.com.cn

目录

1 概述

1.1 简介

实时操作系统（RTOS）是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统做出快速响应，调度一切可利用的资源完成实时任务，并控制所有实时任务协调一致运行的操作系统。提供及时响应和高可靠性是其主要特点。

1.2 FreeRTOS 说明

FreeRTOS 是一个迷你的实时操作系统内核。作为一个轻量级的操作系统，FreeRTOS 提供的功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能等，可基本满足较小系统的需要。FreeRTOS 内核支持优先级调度算法，每个任务可根据重要程度的不同被赋予一定的优先级，CPU 总是让处于就绪态的、优先级最高的任务先运行。FreeRTOS 内核同时支持轮换调度算法，系统允许不同的任务使用相同的优先级，在没有更高优先级任务就绪的情况下，同一优先级的任务共享 CPU 的使用时间。

FreeRTOS 的内核可根据用户需要设置为可剥夺型内核或不可剥夺型内核。当 FreeRTOS 被设置为可剥夺型内核时，处于就绪态的高优先级任务能剥夺低优先级任务的 CPU 使用权，这样可保证系统满足实时性的要求；当 FreeRTOS 被设置为不可剥夺型内核时，处于就绪态的高优先级任务只有等当前运行任务主动释放 CPU 的使用权后才能获得运行，这样可提高 CPU 的运行效率。

2 API 说明

2.1 RTOS

Index	Type	API	FreeRTOS	Rhino
1	Memory	void *wl_memset(void *dec, int val, size_t num);	**Yes**	**Yes**
2	Memory	void *wl_memcpy(void *dec, const void *src, size_t num);	**Yes**	**Yes**
3	Memory	int wl_memcmp(const void *a, const void *b, size_t len);	**Yes**	**Yes**
4	Semaphore	void wl_init_sema(sema_t *sema, u32 val, sema_e type);	**Yes**	**Yes**
5	Semaphore	void wl_free_sema(sema_t *sema);	**Yes**	**Yes**
6	Semaphore	void wl_send_sema(sema_t *sema);	**Yes**	**Yes**
7	Semaphore	void wl_send_sema_fromisr(sema_t *sema);	**Yes**	**Yes**
8	Semaphore	u32 wl_wait_sema(sema_t *sema, u32 timeout);	**Yes**	**Yes**
9	Mutex	void wl_init_mutex(mutex_t *pmutex);	**Yes**	**Yes**
10	Mutex	void wl_free_mutex(mutex_t *pmutex);	**Yes**	**Yes**
11	Mutex	void wl_lock_mutex(mutex_t *plock);	**Yes**	**Yes**
12	Mutex	int wl_lock_mutex_to(mutex_t *plock, u32 timeout_ms);	**Yes**	**Yes**
13	Mutex	void wl_unlock_mutex(mutex_t *plock);	**Yes**	**Yes**
14	Critical	void wl_enter_critical(void);	**Yes**	**Yes**
15	Critical	void wl_exit_critical(void);	**Yes**	**Yes**
16	Queue	int wl_init_queue(queue_t* queue, u32 msg_size, u32 depth);	**Yes**	**Yes**
17	Queue	int wl_send_queue(queue_t* queue, void* message, u32 timeout_ms);	**Yes**	**Yes**
18	Queue	int wl_wait_queue(queue_t* queue, void* message, u32 timeout_ms);	**Yes**	**Yes**
19	Queue	int wl_free_queue(queue_t* queue);	**Yes**	**Yes**
20	SystemTime	u32 wl_get_systemtick(void);	**Yes**	**Yes**
21	SystemTime	u32 wl_systemtick_to_ms(u32 tick);	**Yes**	**Yes**
22	SystemTime	u32 wl_ms_to_systemtick(u32 ms);	**Yes**	**Yes**
23	SystemTime	void wl_os_mdelay(int ms);	**Yes**	**Yes**
24	SystemTime	void wl_hal_udelay(int us);	**Yes**	**Yes**
25	SystemTime	void wl_os_yield(void);	**Yes**	**Yes**
26	Atomic	void wl_atomic_set(atomic_t *v, int i);	**Yes**	**Yes**
27	Atomic	int wl_atomic_read(atomic_t *v);	**Yes**	**Yes**
28	Atomic	void wl_atomic_add(atomic_t *v, int i);	**Yes**	**Yes**
29	Atomic	void wl_atomic_sub(atomic_t *v, int i);	**Yes**	**Yes**

30	Atomic	int wl_atomic_add_return(atomic_t *v, int i);	**Yes**	**Yes**
31	Atomic	int wl_atomic_sub_return(atomic_t *v, int i);	**Yes**	**Yes**
32	Thread/Task	thread_hdl_t wl_create_thread(const char *name, u32 stack_size, u32 priority, thread_func_t func, void *thctx);	**Yes**	**Yes**
33	Thread/Task	void* wl_get_threadid(const char *task_name);	**Yes**	**NO**
34	Thread/Task	thread_hdl_t wl_currrnet_threadid(void);	**Yes**	**NO**
35	Thread/Task	void wl_destory_thread(thread_hdl_t hdl);	**Yes**	**Yes**
36	Thread/Task	void wl_destory_threadself(void);	**Yes**	**Yes**
37	Thread/Task	u32 wl_get_prio(thread_hdl_t task);	**Yes**	**NO**
38	Thread/Task	void wl_set_prio(thread_hdl_t task, int prio);	**Yes**	**NO**
39	Timer	void wl_init_timer(timer_t *ptimer, timer_callback pfunc, void* context);	**TBD**	**Yes**
40	Timer	void wl_start_timer(timer_t *ptimer, u32 ms);	**TBD**	**Yes**
41	Timer	void wl_stop_timer(timer_t *ptimer);	**TBD**	**Yes**
42	Timer	void wl_destory_timer(timer_t *ptimer);	**TBD**	**Yes**
43	Memory	u32 wl_get_freeheapsize(void);	**Yes**	**Yes**
44	Memory	void* wl_malloc(u32 sz);	**Yes**	**Yes**
45	Memory	void* wl_zmalloc(u32 sz);	**Yes**	**Yes**
46	Memory	void* wl_realloc(void *ptr, u32 sz);	**Yes**	**Yes**
47	Memory	void wl_free(void *pbuf);	**Yes**	**Yes**
48	System	u32 wl_get_random32(void);	**Yes**	**Yes**
49	System	void wl_os_init(void);	**Yes**	**Yes**
50	System	void wl_os_start(void);	**Yes**	**Yes**

2.1.1 void *wl_memset(void *dest, int val, size_t num)

```
/**
   * @brief     memory set
   * @detail    Sets the first num bytes of the block of memory pointed by ptr to the specified
value (interpreted as an unsigned char)
   * @param    dest: Pointer to buffer
   * @param    val : value will be set to buffer
   * @param    num : how many buffer length will be set
   * @retval
   */
```

说明

设置 dest 指向的内存区的初值为 val。

对于使用 *wl_malloc* 或者其他途径获取的内存区域，在使用之前需要使用该 API 对该内存区进行初始化，统一设置为 val 值。通常 val 为 0，对于特殊的使用场景（比如 Flash 空间）可能会需要配置为特殊值，比如 0xFF。

参数

dest：需要初始化的起始内存地址
val：初始化的值
num：dest 地址开始的内存区中初始化的内存字节数

返回

无

示例

```
#define TEST_STRING_SIZE       32
void funcA()
{
    int ret = 0;
    char stringA[TEST_STRING_SIZE] = "TestFunctionA";

    printf("Orig stringA [%s]\n", stringA);
    ret = wl_memset(stringA, 'M', TEST_STRING_SIZE - 1);
    printf("Latest StringA [%s]\n", stringA);
}
```

2.1.2 void *wl_memcpy(void *dest, const void *src, size_t num)

```
/**
 * @brief     memory copy
 * @detail    Copies the values of num bytes from the location pointed to by source directly to
the memory block pointed to by destination.
 * @param     dest: Pointer to destination buffer
 * @param     src : Pointer to source buffer
 * @param     num : how many source buffer length will be copy
 * @retval
 */
```

说明

从源内存的起始位置 src 开始拷贝 num 个字节到目标内存地址 dest 中。

参数

dest：目标内存地址

src：源内存地址
num：需要拷贝的个数

返回
无

示例

```
#define TEST_STRING_SIZE       32
void funcA()
{
      int ret = 0;
      char stringA[TEST_STRING_SIZE] = "TestFunctionA";
      char stringB[TEST_STRING_SIZE] = "SrcTestFunctionB";

      printf("Orig stringA [%s]\n", stringA);
      ret = wl_memcpy(stringA, stringB, TEST_STRING_SIZE);
      printf("Latest StringA [%s]\n", stringA);
}
```

2.1.3 int wl_memcmp(const void *a, const void *b, size_t len)

```
/**
   * @brief      wl_memcmp copy
   * @detail     Compares the first len bytes of the block of memory pointed by a to the first len
   bytes pointed by b, returning zero if they all match or a value different from zero
representing which is greater if they do not.
   * @param     a : Pointer to    buffer a
   * @param     b : Pointer to    buffer b
   * @param     len : how many source buffer length will be compare
   * @retval    if len of buffer a equals len of buffer b reuturn 1 else return 0
   */
```

说明
比较内存区域 a 和 b 的前 len 个数据。

参数
a：需要对比的内存地址 a
b：需要对比的内存地址 b
num：从内存地址 a、b 开始，需要比较的字节个数

返回
0：从内存地址 a、b 开始的前 num 字节数据相同

非 0：从内存地址 a、b 开始的前 num 字节数据不同

示例

```
#define TEST_STRING_SIZE       32
void funcA()
{
    int ret = 0;
    char stringA[TEST_STRING_SIZE] = "TestFunctionA";
    char stringB[TEST_STRING_SIZE] = "TestFunctionB";

    ret = wl_memcmp(stringA, stringB, TEST_STRING_SIZE);
    printf("StringA equal to StringB: [%s]\n", (0 ==    ret) ? "Yes" : "No");
}
```

2.1.4 void wl_init_sema(sema_t *sema, u32 val, sema_e type)

```
/**
  * @brief     create and init    semaphores
  * @detail    Semaphores are created with wl_init_sema(), where sema is a pointer to
                space for a struct sema_t, val is the initial value of the semaphore,
and                             type describes the type of semaphore.
  * @param     sema:pointer of semaphores
  * @param     val:init value
  * @param     type:type of    semaphores
  * @retval    None
  */
```

说明

初始化信号量。

根据 *type* 的配置，初始化不同类型的信号量（二值信号量或者计数信号量，AliOS 环境下，该参数无效）。该 API 会申请并占用部分的 RAM 空间，初始化后的信号量，后续需要调用 *wl_free_sema* 来删除并释放占用的内存空间。

参数

sema：需要初始化的信号量的地址。

val：初始化的信号量初值。二值信号量初值可选项为 0 和 1。

type：信号量类型。*sema_binary* 对应二值信号量，*sema_counter* 对应计数信号量

返回

无

示例

```
static sema_t dema_sema = NULL;
void taskA(void *param)
{
    wl_init_sema(&dema_sema, 0, sema_binary);
    if (NULL == dema_sema) {
        // Print error message, init sema fail.
    }
    else {
        // Init sema succ
        // Now block to wait the sema to do related things.
        wl_wait_sema(&dema_sema, portMAX_DELAY);
        // do somethings……
        // ……

        // Free sema and return
        wl_free_sema(&dema_sema);
    }
}
```

2.1.5 void wl_free_sema(sema_t *sema)

```
/**
  * @brief    free memory of semaphores
  * @detail   Sema is destroyed by this function.
  * @param    sema:   pointer of semaphores to be freed
  * @retval   None
  */
```

说明

删除之前由 *wl_init_sema* 初始化的信号量，并释放占用的内存资源。

参数

sema 是需要删除的信号量的地址。

返回

无

示例

参考 *wl_init_sema* 的示例。

2.1.6 void wl_send_sema(sema_t *sema)

```
/**
   * @brief      send a semaphore
   * @detail     Sema is send (incremented) by this function.
   * @param      sema:    pointer of semaphores
   * @retval     None
   */
```

说明

发送或者释放信号量。

释放之前 *wl_init_sema* 初始化并成功占有的信号量。这个 API 不能在中断处理程序中使用，对应可以在中断处理程序中使用的 API 都会带有*_fromisr* 的后缀，因此如果想在中断处理程序中释放某个信号量，应该使用 *wl_send_sema_fromisr*。

参数

sema：需要释放的信号量的地址。

返回

无

示例

```
static sema_t dema_sema = NULL;
void taskA(void *param)
{
    wl_init_sema(&dema_sema, 0, sema_binary);
    if (NULL == dema_sema) {
        // Print error message, init sema fail.
    }
    else {
        // Init sema succ
        // Now block to wait the sema to do related things.
        wl_wait_sema(&dema_sema, portMAX_DELAY);
        // do somethings……
        // ……

        // Free sema and return
        wl_free_sema(&dema_sema);
```

```
    }
}
void taskB(void *param)
{
    // prepare for send sema
    if (NULL != dema_sema) {
        wl_send_sema(&dema_sema);
    }
}
```

2.1.7 void wl_send_sema_fromisr(sema_t *sema)

```
/**
  * @brief     send a semaphore in interrupt service routine (ISR)
  * @detail    Sema is send (incremented) in interrupt service routine by this function.
  * @param     sema:    pointer of semaphores
  * @retval    None
  */
```

说明

从中断中释放信号量。

类似 *wl_send_sema*，这个是可以使用在 ISR、实现类似功能的接口。

如果在 ISR 中调用该 API，导致某个任务（比如例子中的 taskA）从之前等待该信号量的堵塞状态变为可执行状态，而且该任务（taskA）的优先级大于当前被中断的任务（比如 taskB）的优先级，则在当前 ISR 执行后，变为可执行状态、优先级更高的任务（taskA）会被调用执行。

参数

sema 是需要释放的信号量的地址。

返回

无

示例

```
static sema_t dema_sema = NULL;
void taskA(void *param)
{
    wl_init_sema(&dema_sema, 0, sema_binary);
    if (NULL == dema_sema) {
        // Print error message, init sema fail.
```

```
        }
        else {
            // Init sema succ
            // Now block to wait the sema to do related things.
            wl_wait_sema(&dema_sema, portMAX_DELAY);
            // do somethings……
            // ……

            // Free sema and return
            wl_free_sema(&dema_sema);
        }
}
void isrB(void *param)
{
        // prepare for send sema
        if (NULL != dema_sema) {
            wl_send_sema_fromisr(&dema_sema);
        }
}
```

2.1.8 u32 wl_wait_sema(sema_t *sema, u32 timeout)

```
/**
  * @brief     wait a semaphore
  * @detail    Sema is waited on (decremented) with wl_wait_sema. The timeout
                argument specifies the minimum time in ticks to wait before returning
                with failure .
  * @param     sema:    pointer of    semaphores
  * @param     timeout:    delay time in ms
  * @retval    if success return TRUE, or return FALSE
  */
```

说明

等待或者获取信号量 sema。获取一个之前被 *wl_init_sema* 成功初始化的信号量，该 API 不能用于中断处理程序中。

参数

sema：需要等待的信号量的地址。

timeout：定时超时时间，单位为 ms。

如果该值为 0，当信号量获取失败后（当前没有可用的信号量），该函数会立

刻返回；

如果该值为 portMAX_DELAY（0xFFFF）啊啊，当信号量获取失败（当前没有可用的信号量），该函数会一直等待（堵塞当前线程）直到该信号量成功获取（其他线程释放对应的信号量）；

如果该值为其他超时时间，当信号量获取失败（当前没有可用的信号量），该函数会在该指定的时间内等待（堵塞当前线程）知道该信号量成功获取（其他线程释放对应的信号量）；

返回

等待信号量结果值，成功或失败类型

TRUE：成功获取信号量。如果 timeout 时间有指定（非 0），则表示在等待的时间间隔内，信号量被其他线程（或者 ISR）释放，当前线程成功获取信号量并从堵塞状态转为可执行状态。

FALSE：获取信号量失败。如果 timeout 时间有指定（非 0），则表示在等待的时间间隔内，当前线程没有成功获取信号量，但是超时时间已经到了（超时后调度器唤醒当前线程）。

示例

参考 *wl_send_sema* 示例。

2.1.9 void wl_init_mutex(mutex_t *pmutex)

```
/**
 * @brief     create and init mutex
 * @detail    This function creates a mutual exclusion semaphore, where pmutex is a
 *            pointer to space for a struct mutex_t.
 * @param     pmutex:    pointer of mutex
 * @retval    None
 */
```

说明

初始化互斥型信号量 pmutex。

该 API 会申请并占用部分的 RAM 空间，初始化后的互斥信号量，在使用完毕后，需要调用 *wl_free_mutex* 来释放占用的内存。

参数

pmutex：需要初始化的互斥型信号量的指针。初始化成功后，该指针指向对应的互斥信号量。

返回

无

备注

互斥信号量和二值信号量非常相似，都可以对特定资源的保护而实现互斥访问，用于进程间的同步，但两者也有不同的地方

二值信号量：用于进程间的同步，成功获取信号量的进程不需要进行释放。进程间的同步通常是一个进程/中断处理程序释放一个信号量，另外一个进程等待获取该信号量。

互斥信号量：会发生优先级的反转，持有互斥信号量的线程 A，在持有期间，其他更高优先级的线程 B 尝试获取该信号量，会导致线程 A 的优先级临时提高并继承线程 B 的优先级，在线程 A 释放互斥信号量后，优先级会重新恢复到原值。成功获取信号量的线程，后续需要主动释放获取的互斥信号量。

示例

```
static mutex_t dema_semaphore = NULL;

void taskA(void *param)
{
    // Do somthings..
    wl_init_mutex(&dema_semaphore);
    if (NULL != dema_semaphore) {
        // Try to task the mutex semaphore, this will block current task forever
        wl_lock_mutex(&dema_semaphore);
        // Do for related things
        wl_unlock_mutex(&dema_semaphore);
        …
        …
        wl_free_mutex(&dema_semaphore);
    }
    else {
        // Init mutex semaphore fail, …
    }
}
```

2.1.10 void wl_free_mutex(mutex_t *pmutex)

```
/**
  * @brief      free memory of a mutex
  * @detail     This function deletes a mutual exclusion semaphore and readies all tasks
                pending on pmutex.
  * @param      pmutex:    pointer of mutex to be freed
```

```
 * @retval None
 */
```

说明

删除互斥型信号量 pmutex。成功初始化后的互斥信号量，在使用完毕之前需要调用 *wl_free_mutex* 释放初始化过程中占用的内存空间。

参数

pmutex：指向需要删除的互斥型信号量的地址。

返回

无

示例

参考 *wl_init_mutex* 的示例

2.1.11 void wl_lock_mutex(mutex_t *plock)

```
/**
 * @brief     lock a mutex
 * @detail    This function waits for a mutual exclusion semaphore until get it.
 * @param     pmutex:pointer of mutex to be unlocked
 * @retval    None
 */
```

说明

获取并锁定互斥型信号量 plock。

获取成功后，其他尝试获取该信号量的线程会堵塞等待。因为会发生优先级的反转，所以在获取并锁定后，临界区的执行时间应该尽可能短，并在处理完临界区资源后尽快释放锁定的信号量。

参数

plock：需要锁定的互斥型信号量。

返回

无

备注

该 API，默认等待时间为 45 秒，如果在该等待间隔内没有成功获取并锁定对应的互斥信号量，则会打印当前的线程名称并退出。对应该 API，*wl_lock_mute_to* 可以指

定具体的等待时间间隔。类似的，如果等待超时，也会打印对应的线程名称。

两者的不同在于，*wl_lock_mutex_to* 有返回值，可以根据返回值确认获取信号量失败还是成功。建议使用 *wl_lock_mutex_to*。

示例

功能参考 *wl_init_mutex* 示例。

2.1.12 int wl_lock_mutex_to(mutex_t *plock, u32 timeout_ms)

```
/**
 * @brief      lock a mutex in timeout
 * @detail     This function waits for a mutual exclusion semaphore in timeout_ms.
 * @param      pmutex:pointer of mutex to be unlocked
 * @param      timeout_ms: time in ms of blocktime
 * @retval     if ok ,return TURE, or return FALSE
 */
```

说明

规定时间内获取并锁定互斥量 plock。

参数

plock：指向需要获取并锁定的互斥量地址

timeout_ms：等待的时间间隔，单位 ms

返回

TRUE：在 timeout_ms 超时间隔内，成功获取并锁定信号量

FALSE：在 timeout_ms 超时间隔内，获取信号量失败

备注

如果在该等待间隔内没有成功获取并锁定对应的互斥信号量，则会打印当前的线程名称并返回 FALSE 标识。

示例

功能参考 *wl_lock_mutex* 示例。

2.1.13 void wl_unlock_mutex(mutex_t *plock)

```
/**
 * @brief      unlock a mutex
```

```
* @detail    This function signals a mutual exclusion semaphore.
* @param     pmutex:pointer of mutex to be locked
* @retval    None
*/
```

说明

释放互斥型信号量 plock。在线程成功获取并锁定互斥信号量，完成临界区资源访问和处理之后，需要及时释放锁定的互斥信号量。

参数

plock：指向要释放的互斥量的地址。

返回

无

示例

参考 *wl_init_mutex* 示例。

2.1.14 void wl_enter_critical(void)

```
/**
* @brief     task enter critical state
* @detail    Disable interrupts by preserving the state of interrupts
* @param     None
* @retval    None
*/
```

说明

进入“临界区”。关中断函数，保护即将执行的操作不被打断。

该 *wl_enter_critical* 和 *wl_exit_critical* 通过关闭/打开中断的方式，提供基本的临界区的实现。该 API 执行后，会关闭部分/全部的中断，具体需要查看系统的配置。

对于 FreeRTOS 系统而言，如果有配置使用 configMAX_SYSCALL_INTERRUPT_PRIORITY 宏，则所有优先级低于该配置的中断都被关闭，所有高于/等于该优先级的中断则保持打开状态；如果没有配置使用该宏，则所有的中断都会被关闭。

wl_enter_critical 和 *wl_exit_critical* 实现为可以嵌套调用，因此只有在最后一个（每个 *wl_enter_critical* 调用后，都需要成对调用一次 *wl_exit_critical*、*wl_exit_critical* 被调用后，才实际退出临界区。

在使用上，临界区的执行时间应该是尽可能短，因为在此期间系统的中断是处于被关闭状态。

参数
无

返回
无

备注
该 API 不能在中断处理程序中调用。

示例
```
void dema_procedure(void)
{
    // Enter critical section. For this example, here enter the critical section will result in a nesting depth of 2.
    wl_enter_critical();
    // Execute the code that requires the critical section here.
    // Exit the critical section. In this example, this function is itself called from a critical section, so this call to wl_exit_critical () will decrement the nesting count by one, but not result in interrupts becoming enabled.
    wl_exit_critical();

}

void taskA(void *param)
{
    // Create a critical section.
    wl_enter_critical();
    // Execute the code that requires the critical section here.
    // Calls to wl_enter_critical () can be nested so it is safe to call a function that includes its own calls to wl_enter_critical () and wl_exit_critical ().
    dema_procedure();
    // The operation that required the critical section is complete so exit the critical section. After this call to wl_exit_critical (), the nesting depth will be zero, so interrupts will have been re-enabled.
    wl_exit_critical();
}
```

2.1.15 void wl_exit_critical(void)

```
/**
  * @brief      task exit critical state
```

```
* @detail     Enable interrupts and restore the interrupt disable state.
* @param      None
* @retval     None
*/
```

说明

退出“临界区”。

该 *wl_enter_critical* 和 *wl_exit_critical* 通过关闭/打开中断的方式，提供基本的临界区的实现。该 API 执行后，会关闭部分/全部的中断，具体需要查看系统的配置。

对于 FreeRTOS 系统而言，如果有配置使用 configMAX_SYSCALL_INTERRUPT_PRIORITY 宏，则所有优先级低于该配置的中断都被关闭，所有高于/等于该优先级的中断则保持打开状态；如果没有配置使用该宏，则所有的中断都会被关闭。

wl_enter_critical 和 *wl_exit_critical* 实现为可以嵌套调用，因此只有在最后一个（每个 *wl_enter_critical* 调用后，都需要成对调用一次 *wl_exit_critical*、*wl_exit_critical* 被调用后，才实际退出临界区。

在使用上，临界区的执行时间应该是尽可能短，因为在此期间系统的中断是处于被关闭状态

参数

无

返回

无

备注

该 API 不能在中断处理程序中使用。

示例

参考 *wl_enter_critical* 示例。

2.1.16 int wl_init_queue(queue_t* queue, u32 msg_size, u32 depth)

```
/**
* @brief      create and init message queue
* @detail     This function creates a message queue and initialize it.
* @param      queue: pointer of create queue
* @param      msg_size: size of item
* @param      depth: number of elements in queue
* @retval     if ok ,return 0,or return -1
*/
```

说明

初始化消息队列，成功后返回指向新创建的队列的指针。

成功初始化消息队列后，会新建并占用部分系统的 RAM 空间，在使用完毕后需要调用 *wl_free_queue* 来释放对应的 RAM 空间。

参数

queue：指向消息队列的地址，成功初始化后，该指针指向对应的消息队列；

msg_size：消息类型大小（指针或结构体），能够保存在队列中的消息数据大小，单位为字节，通常是消息对应结构体的大小；

depth：消息队列深度(最大 5)，在任意时间点上，能够保存在队列中的消息个数。

返回

NULL：消息队列初始化失败，通常是因为没有足够的内存空间导致相关的资源申请失败；

其他值：消息队列初始化成功，返回值是一个指向该消息队列的句柄。

备注

消息队列可以在线程之间、线程与中断之间传递数据；消息队列在调度器启动之前或者之后都可以进行创建。

示例

```
// Define the data type that will be queued.
typedef struct messageA
{
    char messageID;
    char data[ 20 ];
} messageA_t;

// Define the queue parameters.
#define DEMO_QUEUE_LENGTH 5
#define DEMO_QUEUE_ITEM_SIZE sizeof( messageA_t)

int main( void )
{
    int ret = 0;
    queue_t dema_queue;
    // Create the queue, storing the returned handle in the dema_queue variable.
    ret = wl_init_queue(&dema_queue, DEMO_QUEUE_LENGTH,
DEMO_QUEUE_ITEM_SIZE );
    if( -1 == ret) {
        // The queue could not be created.
```

```
        return;
    }

    // Rest of code goes here.
    ...
    // Free the queue
    wl_free_queue(&dema_queue);

}
```

2.1.17 int wl_send_queue(queue_t* queue, void* message, u32 timeout_ms)

```
/**
  * @brief     send message to queue
  * @detail    This function sends a message to a queue.
  * @param     queue:pointer of send queue
  * @param     message: message information
  * @param     timeout_ms : timeout value or 0 in case of no-timeout
  * @retval    if ok ,return 0,or return -1
  */
```

说明

发送消息到消息队列的尾部。

当前发送的消息，会被放置到消息队列的尾部，如果队列已满（当前在队列中的消息个数等于消息队列的深度）则可能返回-1 表示入队失败。

参数

queue：指向消息队列的地址；

message：发送到消息队列的消息；

timeout_ms：入队等待时间，单位 ms，

如果当前消息队列已满，则会堵塞当前线程并等待 timeout_ms 毫秒；

如果传入的 timeout_ms 为 portMAX_DELAY，则会一直堵塞当前进程并等待（不超时），直到对应的消息队列非满并成功入队为止

返回

-1：入队失败。如果传入的 timeout_ms 不为 portMAX_DELAY，则表示在等待时间间隔内，消息队列一直都是满的状态，提示入队失败

0：入队成功。在超时时间间隔内，消息成功写入到消息队列末尾。

示例

```
// Define the data type that will be queued.
typedef struct messageA
{
    char messageID;
    char data[ 20 ];
} messageA_t;

// Define the queue parameters.
#define DEMO_QUEUE_LENGTH        5
#define DEMO_QUEUE_ITEM_SIZE     sizeof( messageA_t)
#define SEND_EXAMPLE             10
int main( void )
{
    int ret = 0;
    queue_t dema_queue;
    // Create the queue, storing the returned handle in the dema_queue variable.
    ret = wl_init_queue(&dema_queue, DEMO_QUEUE_LENGTH,
DEMO_QUEUE_ITEM_SIZE );
    if( -1 == ret) {
    // The queue could not be created.
    return;
    }

    // Rest of code goes here.
    wl_create_thread("task A", MAIN_TASK_STACK_SZ, MAIN_TASK_PRIO,
(thread_func_t)taskA, NULL);

    for(;;);
    // Free the queue
    wl_free_queue(&dema_queue);

}

void taskA(void *param)
{
    queue_t queue;
    messageA_t message;
    // The queue handle is passed into this task as the task parameter. Cast the
parameter back to a queue handle.
    queue = (queue_t) param;
    for(;;) {
        // Create a message to send on the queue.
```

```
        message.messageID = SEND_EXAMPLE;
        // Send the message to the queue, waiting for 10 ticks for space to become
available if the queue is already full.
        if( wl_send_queue(queue, &message, 10) != 0) {
            // Data could not be sent to the queue even after waiting 10 ms.
        }
    }
}
```

2.1.18 int wl_wait_queue(queue_t* queue, void* message, u32 timeout_ms)

```
/**
  * @brief     get or wait for message from a queue
  * @detail    This function waits for a message to be sent to a queue.
  * @param     queue:pointer of a queue
  * @param     message: message information
  * @param     timeout_ms: timeout value or 0 in case of no-timeout
  * @retval    if ok ,return 0,or return -1
  */
```

说明

等待消息队列。

在 queue 指定的消息队列上等待并获取消息。

参数

queue：指向消息队列的地址；

message：消息内容，该 API 返回后，成功获取到的消息将放置到该数据结构中；

timeout_ms：等待时间,单位 ms，

如果当前消息队列为空，则会堵塞当前线程并等待 timeout_ms 毫秒；

如果传入的 timeout_ms 为 portMAX_DELAY，则会一直堵塞当前进程并等待（不超时），直到对应的消息队列非空并获取到消息为止

返回

-1：获取消息失败。如果传入的 timeout_ms 不为 portMAX_DELAY，则表示在等待时间间隔内，消息队列一直都是空的状态，提示等待并获取消息失败。

0：获取消息成功。在超时时间间隔内，消息成功获取，并存放到 message 结构体中。

示例

```
// Define the data type that will be queued.
typedef struct messageA
{
    char messageID;
    char data[ 20 ];
} messageA_t;

// Define the queue parameters.
#define DEMO_QUEUE_LENGTH        5
#define DEMO_QUEUE_ITEM_SIZE     sizeof( messageA_t)

int main( void )
{
    int ret = 0;
    queue_t dema_queue;
    // Create the queue, storing the returned handle in the dema_queue variable.
    ret = wl_init_queue(&dema_queue, DEMO_QUEUE_LENGTH,
DEMO_QUEUE_ITEM_SIZE );
    if( -1 == ret) {
    // The queue could not be created.
    return;
    }

    // Rest of code goes here.
    wl_create_thread("task Wait", MAIN_TASK_STACK_SZ, MAIN_TASK_PRIO,
(thread_func_t)taskWait, NULL);

    for(;;);
    // Free the queue
    wl_free_queue(&dema_queue);

}

void taskWait(void *param)
{
    queue_t queue;
    messageA_t message;
    // The queue handle is passed into this task as the task parameter. Cast the
parameter back to a queue handle.
    queue = (queue_t) param;
    for(;;) {
        // Wait for the maximum period for data to become available on the queue.
```

```
            if( wl_wait_queue(queue, &message, portMAX_DELAY) != 0 ) {
                // Nothing was received from the queue even after blocking to wait for
data to arrive.
            }
            else {
                // message now contains the received data.
            }
        }
}
```

2.1.19 int wl_free_queue(queue_t* queue)

```
/**
  * @brief wl_free_queue:    free    memory of    a queue
  * @detail     This function deletes a message queue and readies all tasks pending on
                the queue.
  * @param    queue:    pointer of aqueue
  * @retval : if ok ,return 0,or return -1
  */
```

说明

删除消息队列。

消息队列使用完毕（通常是在系统启动时候或者相关模块启动后，初始化需要使用的消息队列，系统结束或者模块结束运行后，删除对应的消息队列），删除对应的消息队列（初始化后消息队列占用的 RAM 空间会被释放）。

参数

queue：指向消息队列的地址

返回

-1：提示删除的消息队列上，还有未被读取的消息；但是对应的消息队列是被成功删除，消息队列占用的空间也成功释放

0：消息队列被成功删除，消息队列占用的空间也成功释放。

备注

在删除消息队列之前，会先判断当前消息队列上是否还有未被读取的消息，如果是，设置返回值为-1，并会继续删除对应的消息队列。该 API 的返回值只是标识当前正在删除的消息队列上是否有未被读取的消息而已。

示例

参考 wl_init_queue 示例。

2.1.20 u32 wl_get_systemtick(void);

```
/**
  * @brief     get system counter
  * @detail    This function is used by your application to obtain the current value of the
                32-bit counter which keeps track of the number of clock ticks.
  * @param     None
  * @retval    system counter as 32-bit value
  */
```

说明：获取系统 tick 值
参数：无
返回：系统启动时间（tick 个数）

说明

获取当前系统 tick 值。
该值为调度器启动后，系统时钟的滴答值。

参数

无

返回

系统启动时间（tick 个数）。

备注

该值为系统当前的 tick 值，实际的时间取决于系统配置的 configTICK_RATE_HZ。可以使用 *wl_systemtick_to_ms* 来将系统时钟的 tick 值转换成对应单位为 ms 的时间值，或者使用 *wl_ms_to_systemtick* 来将对应单位为 ms 的时间值转换为系统时钟的 tick 值。。

tick 会出现溢出并回复为 0，但是该值不会影响系统内核的 tick 计数。对于上层的应用程序，如果对 tick 值敏感的情况下，需要考虑溢出的场景而做额外的逻辑判断。

tick 值溢出的频率，取决于系统的配置（configUSE_16_BIT_TICKS 参数的配置），该参数为 1 的情况下，tick 由 16 位数据空间来进行存储；该参数为 0 的情况下，tick 值是一个 32 位的数据。

当前系统配置下，tick 值为 32 位数据值。

示例

```
void taskA()
{
    u32 tick;
    u32 time_ms;

    // Get the current tick.
    tick = wl_get_systemtick();
    // Translate the system tick to time in ms
    time_ms = wl_systemtick_to_ms(tick);
}
```

2.1.21 u32 wl_systemtick_to_ms(u32 tick);

```
/**
  * @brief     get system counter to ms
  * @detail    Convert tick to millisecond.
  * @param     system tick
  * @retval    value in ms
  */
```

说明

将系统 tick 转换成具体时间的毫秒值。

实际的时间取决于系统配置的 configTICK_RATE_HZ。可以使用 *wl_systemtick_to_ms* 来将系统时钟的 tick 值转换成对应单位为 ms 的时间值，或者使用 *wl_ms_to_systemtick* 来将对应单位为 ms 的时间值转换为系统时钟的 tick 值。

参数

tick：系统启动 tick 值

返回

系统启动毫秒数

示例

参考 *wl_get_sytemtick()*示例

2.1.22 u32 wl_ms_to_systemtick(u32 time_ms)

```
/**
```

```
* @brief      get value ms to    system counter
* @detail     Convert millisecond to tick.
* @param      time in ms
* @retval     value in system tick
*/
```

说明

时间的毫秒值转换成系统时钟的 tick 值。

实际的时间取决于系统配置的 configTICK_RATE_HZ。可以使用 *wl_systemtick_to_ms* 来将系统时钟的 tick 值转换成对应单位为 ms 的时间值，或者使用 *wl_ms_to_systemtick* 来将对应单位为 ms 的时间值转换为系统时钟的 tick 值。

参数

time_ms：系统时间的毫秒值

返回

单位为 ms 的系统时间 time_ms 对应的系统时钟的 tick 值

2.1.23 void wl_os_mdelay(int time_ms)

```
/**
* @brief      os time delay
* @detail     This function is called to delay execution of the currently running task
              until the specified millisecond expires. This, of course, directly equates to
              delaying the current task for some time to expire. No delay will result If
              the specified delay is 0. If the specified delay is greater than 0 then, a
              context switch will result.
* @param      delay time in ms
* @retval     None
*/
```

说明：延时 ms 毫秒

参数：ms 是毫秒值

返回：无

说明

延时当前线程 ms 毫秒后再执行。

调用该 API 的线程会被设置为堵塞状态并持续 time_ms 毫秒；如果配置的 time_ms 为 0，则调用的线程会被设置为就绪状态而非堵塞状态，但是会发生一次任务切换，导致调度器选择其他相同优先级并处于 ReadyState 的任务执行。

参数

time_ms：当前进程被堵塞的时间，单位 ms。

返回

无

示例

```
void taskA(void* param)
{
        while(1){
            // Sleep 2seconds
            wl_os_mdelay(2000);
             //Excecute ..
        }
}
```

2.1.24 void wl_hal_udelay(int time_us)

```
/**
   * @brief     hal time delay
   * @detail    This function is called to delay execution of the currently running task
                until the specified microsecond expires. This, of course, directly equates
                to delaying the current task for some time to expire. No delay will result If
                the specified delay is 0. If the specified delay is greater than 0 then, a
                context switch will result.
   * @param     time in us
   * @retval    None
   */
```

说明：延时 us 微秒

参数：us 是微秒值

返回：无

说明

延时当前线程 time_us 微秒后在执行。

该 API 类似 *wl_os_mdelay*，不同在于该 API 是微秒为单位，而 *wl_os_mdelay* 是毫秒为单位。

参数

time_us：当前线程被堵塞的时间，单位 us。

返回

无

示例

```
void taskA(void* param)
{
        while(1){
            // Sleep 2ms
            wl_hal_udelay(2000);
             //Excecute ..
        }
}
```

2.1.25 void wl_os_yield(void)

```
/**
   * @brief      os yield task
   * @detail     This function assigns system resources to other tasks
   * @param      None
   * @retval     None
   */
```

说明：让出系统资源给其他任务

参数：无

返回：无

说明

让出系统资源给其他相同优先级的任务。

该行为发生在没有发生内核抢占，并且当前线程的时间片到期之前，主动放弃系统 cpu 资源。

参数

无

返回

无

备注

wl_os_yield 必须在正在运行的线程中调用，因此当调度器启动之前，不能调用该 API。

调用该 API 后，调度器会选择一个相同优先级、处于 ReadySate 的任务运行，如果当前没有其他相同优先级的任务处于 ReadyState，则当前线程继续执行；调度器不会选择其他更高优先级并处于 ReadyState 的任务，因为如此不能保证后续在第一时间重新调度当前线程进行运行。

示例

```
void taskA(void* param)
{
        while(1){
            // Volunteers to leave the Running State
            wl_os_yield();
             // Return to Running State and excecute ..
        }
}
```

2.1.26 void wl_atomic_set(atomic_t *v, int i)

```
/**
   * @brief     set atomic variable
   * @detail    The atomic_t type should be defined as a signed integer.
   * @param     v:pointer of type atomic_t
   * @param     i : required value
   * @retval    None
   */
```

说明

设置原子量的值。

参数

v：指向需要设置的原子量的指针

i：原子量需要设置的值

返回

无

示例

2.1.27 int wl_atomic_read(atomic_t *v)

```
/**
  * @brief      read atomic variable
  * @detail     This routine reads value from the given atomic_t value.
  * @param      pointer of type atomic_t
  * @retval     value of @v
  */
```

说明

读取原子当前值

参数

v：指向需要读取的原子量的指针

返回

读取的原子量的值。

示例

2.1.28 void wl_atomic_add(atomic_t *v, int i)

```
/**
  * @brief      add atomic variable
  * @detail     This routine adds integral values to the given atomic_t value.
  * @param      v:pointer of type atomic_t
  * @param      i : required add value
  * @retval     None
  */
```

说明

原子量增加 i 值

参数

v：指向需要设置的原子量的指针

i：原子量需要增加的值

返回

无

示例

2.1.29 void wl_atomic_sub(atomic_t *v, int i)

```
/**
 * @brief      sub    atomic variable
 * @detail     This routine adds subtract values from the given atomic_t value.
 * @param      v : pointer of type atomic_t
 * @param      i: required sub value
 * @retval     None
 */
```

说明

原子量减小 i 值

参数

v：指向需要设置的原子量的指针

i：原子量需要减少的值

返回

无

示例

2.1.30 int wl_atomic_add_return(atomic_t *v, int i)

```
/**
 * @brief      add atomic variable
 * @detail     This routine adds i, from the given atomic_t and return the new counter
               value after the operation is performed.
 * @param      v : pointer of type atomic_t
 * @param      i: required add value
```

```
 * @retval     value of @v add @i
 */
```

说明

原子量增加 i 值并返回操作后原子量的值。
该 API 通常应用在操作后需要检查原子量值的场景下。

参数

v：指向需要设置的原子量的指针
i：原子量需要增加的值

返回

返回增加后原子量 v 的值

示例

2.1.31 int wl_atomic_sub_return(atomic_t *v, int i)

```
/**
 * @brief      sub atomic variable
 * @detail     This routine subtracts i, from the given atomic_t and return the new
               counter value after the operation is performed.
 * @param      v : pointer of type atomic_t
 * @param      i: required sub value
 * @retval     value of @v sub @i
 */
```

说明

原子量减小 i 值并返回操作后原子量的值。
该 API 通常应用在操作后需要检查原子量值的场景下。

参数

v：指向需要设置的原子量的指针
i：原子量需要减少的值

返回

返回减小后原子量 v 的值

示例

2.1.32 thread_hdl_t wl_create_thread(const char *name, u32 stack_size, u32 priority, thread_func_t func, void *thctx)

```
/**
 * @brief      create thread
 * @detail     This function is used to have OS manage the execution of a task.  Tasks
               can either be created prior to the start of multitasking or by a running task.
               A task cannot be created by an ISR.
 * @param      name : just a name for task to aid debugging
 * @param      stack_size : stack size
 * @param      priority :priority assgined to task
 * @param      func : function that implements tsak
 * @param      thctx : task handler
 * @retval     if create thread sucess ,return thread_hdl_t ,or return NULL
 */
```

说明

创建一个线程。

每次创建都会占用部分的内存空间来存放任务相关的信息，默认是从系统的堆上分配。新创建的任务被标记为就绪状态，如果没有其他更高优先级的就绪任务，就会被调度器选择并成为运行状态。任务在调度器开启前后都可以进行创建。

参数

name：线程名称，后续可以通过该字符串格式的线程名称获取该线程的句柄；

stack_size：申请栈大小，线程在创建时系统会保留对应 stack_size*4 Bytes 的内存作为线程的占空间，对于不同的类型的任务，根据需要进行合理配置；

priority：任务优先级，该优先级从最低的优先级 0 开始，一直到 configMAX_PRIORITIES – 1；

func:线程调用的函数,线程被调度后实际执行的代码,具体实现各自不同的功能；

thctx：func 函数的参数，线程被调度后，传给 func 的参数 void *param

返回

NULL：线程创建失败，由于系统没有足够的内存空间；

Not NULL：其他值，表示线程创建成功，返回 thread_hdl_t 类型的指针，作为线程的句柄。

示例

```
#define IDLE_PRIORITY                (0U)
```

```
#define MAIN_TASK_PRIO                (IDLE_PRIORITY + 1)
#define MAIN_TASK_STACK_SZ            (512)
void taskA(void* context)
{
      thread_hdl_t thread_hdl = NULL;
        while(1){
             wl_os_mdelay(20000);
              // List current heap size
              printf("Current heap size[%x].", wl_get_freeheapsize());
        }
         // Destroy itself
         thread_hdl = wl_current_threadid;
         wl_destroy_thread(thread_hdl);

}
int main( void )
{

      // Create taskA to print heapsize.
      wl_create_thread("TaskAShowHeapSize", MAIN_TASK_STACK_SZ, MAIN_TASK_PRIO,
(thread_func_t)taskA, NULL);

      // Do related things
      for(;;);

}
```

2.1.33 void *wl_get_threadid (const char *task_name)

```
/**
   * @brief wl_currrnet_threadid:    get thread id
   * @param    task_name : task name whose thread id to get
   * @retval : thread id
   */
```

说明

获取特定任务的句柄。

根据 task_name，获取该任务的句柄。task_name 对应 wl_create_thread 时候传入的 name 参数。

参数

task_name：任务名称，获取该名称对应任务的句柄。

返回

任务 id

示例

```
#define IDLE_PRIORITY                (0U)
#define MAIN_TASK_PRIO               (IDLE_PRIORITY + 1)
#define MAIN_TASK_STACK_SZ           (512)
#define TASK_NAME_TASKA              "TaskAShowHeapSize"
#define TASK_NAME_TASKB              "TaskB_GetThreadID"
void taskA(void* context)
{
      thread_hdl_t thread_hdl = NULL;
      while(1){
            wl_os_mdelay(20000);
            // List current heap size
            printf("Current heap size[%x].", wl_get_freeheapsize());
     }
     // Destroy itself
     thread_hdl = wl_current_threadid;
     wl_destroy_thread(thread_hdl);

}

void taskB(void *context)
{
      thread_hdl_t taskA_hdl = NULL;

      taskA_hdl = wl_get_threadid(TASK_NAME_TASKA);
      if (NULL == taskA_hdl) {
            printf("Get threadid for taskA fail.\n");
            wl_destroy_threadself();
            return;
      }
      while(1) {
            wl_os_mdelay(20000);
            // List taskA's priority
            printf("Priority for taskA [%x].", wl_get_prio(taskA_hdl));
      }
      wl_destroy_threadself();
}
```

```
int main( void )
{

    // Create taskA to print heapsize.
    wl_create_thread(TASK_NAME_TASKA,  MAIN_TASK_STACK_SZ,  MAIN_TASK_PRIO,
(thread_func_t)taskA, NULL);

    wl_create_thread(TASK_NAME_TASKB,  MAIN_TASK_STACK_SZ,  MAIN_TASK_PRIO,
(thread_func_t)taskB, NULL);

    // Do related things
    for(;;);

}
```

2.1.34 void *wl_current_threadid(void)

```
/**
  * @brief     get current thread id
  * @detail    This routine reads id from current thread.
  * @param     Noner
  * @retval    thread id
  */
```

说明

获取当前任务的句柄。

参数

tick：值

返回

任务 id

示例

参考 *wl_create_thread* 示例。

2.1.35 void wl_destory_thread(thread_hdl_t hdl)

```
/**
  * @brief     delete thread
  * @detail    This function allows you to delete a task.
  * @param     hdl :thread to be deleted
  * @retval    None
  */
```

说明

销毁当前任务。

任务销毁后，创建任务时候占用的内存空间将会被系统自动释放；对于任务内部自行新建、占用的内存空间，需要在调用该 API 销毁任务前，自行进行释放。

该 API 和后续 *wl_destroy_threadself* 实现类似的功能，不同在于该 API 通过传入的任务句柄可以删除指定的任务，*wl_destroy_threadself* 则只能在任务内部进行调用，并删除任务本身。

参数

hdl：指向任务的句柄。

返回

无

示例

参考 *wl_create_thread* 示例。

2.1.36 void wl_destory_threadself(void)

```
/**
  * @brief     delete calling task
  * @detail    The calling task can delete itself.
  * @param     None
  * @retval    None
  */
```

说明

任务自我销毁。

任务销毁后，创建任务时候占用的内存空间将会被系统自动释放；对于任务内部自行新建、占用的内存空间，需要在调用该 API 销毁任务前，自行进行释放。

该 API 只能在任务内部调用，并删除任务自身；如果想要删除指定的其他任务，可以参考 *wl_destroy_thread*，传入指定任务的句柄即可。

参数
无

返回
无

示例
参考 *wl_get_threadid* 示例。

2.1.37 u32 wl_get_prio(thread_hdl_t task_hdl)

```
/**
 * @brief      get the priority of the task
 * @detail     This routine reads priority from the task.
 * @param      the task to be getted the priority
 * @retval     the prioroty
 */
```

说明
获取指定任务的优先级。

通过传入的 task_hdl，获取指定任务的优先级。任务句柄 task_hdl，可以通过 *wl_create_thread* 创建任务之后保存的返回值来获取，或者使用 *wl_get_threadid*（传入对应任务的名称）来获取。

如果传入的 task_hdl 为 NULL，则表示获取当前任务的优先级。

参数
task_hdl：指向需要获取优先级的任务的指针。

返回
无

示例
```
#define IDLE_PRIORITY                (0U)
#define MAIN_TASK_PRIO               (IDLE_PRIORITY + 1)
#define MAIN_TASK_STACK_SZ           (512)
#define TEST_TASK_A_PRIO             (IDLE_PRIORITY + 4)
thread_hdl_t taskA_hdl = NULL;
```

```
void taskA(void* context)
{
    int task_prio = 0;

    task_prio = wl_get_prio(taskA_hdl);
    printf("Change priority from %d to %d.\n", task_prio, TEST_TASK_A_PRIO);

    wl_set_prio(taskA_hdl, TEST_TASK_A_PRIO);

}
int main( void )
{

    // Create taskA to print heapsize.
    taskA_hdl = wl_create_thread("TaskAShowHeapSize", MAIN_TASK_STACK_SZ,
MAIN_TASK_PRIO, (thread_func_t)taskA, NULL);

    for(;;);
    //

}
```

2.1.38 void wl_set_prio(thread_hdl_t task_hdl, int prio)

```
/**
  * @brief    set the priority of the task
  * @detail   This routine sets priority to the task.
  * @param    task: the task will be settted new priority
  * @param    prio : the new priority
  * @retval   None
  */
```

说明

设置任务的优先级。

通过传入的 task_hdl，确定需要修改优先级的任务。任务句柄 task_hdl，可以通过 *wl_create_thread* 创建任务之后保存的返回值来获取，或者使用 *wl_get_threadid*（传入对应任务的名称）来获取。

如果传入的 task_hdl 为 NULL，则表示修改当前任务的优先级。

任务可以配置的优先级，从最低优先级 0 开始，一直到最高优先级 10（由

configMAX_PRIORITIES 宏变量限制，可配最大优先级为 configMAX_PRIORITIES-1，当前该宏变量默认配置为 11）

参数

task_hdl：任务句柄，指向需要设置优先级的任务
prio：优先级，可配参数从 0-10

返回

无

示例

参考 *wl_get_prio* 示例。

2.1.39 void wl_init_timer(timer_t *ptimer, timer_callback pfunc, void* context)

```
/**
 * @brief      init timer handler ,and set timer handler function
 * @detail     This routine initializes a timer and sets the callback function.
 * @param      ptimer : pointer of timer to be inited
 * @param      pfunc : function    of timer handler
 * @param      context : data of    ptimer
 * @retval     None
 */
```

说明

初始化定时器。定时器初始化后，在使用之前需要调用 *wl_start_timer* 来启用；使用 *wl_stop_timer* 来关闭指定的定时器；如果定时器不再需要，应该调用 *wl_destroy_timer* 来销毁。

定时器初始化后，会设置对应定时器句柄，ptimer 变量，其他相关的定时器 API 都需要使用该定时器句柄才能操作对应的定时器。

参数

ptimer：定时器，其他相关的接口，*wl_start_timer,wl_stop_timer,wl_destroy_timer* 等等都需要使用该定时器的句柄，才能操作对应的定时器。
pfunc：定时处理函数，
context：pfunc 函数的参数

返回

无

示例

```
#define IDLE_PRIORITY                (0U)
#define MAIN_TASK_PRIO               (IDLE_PRIORITY + 1)
#define MAIN_TASK_STACK_SZ           (512)

// Define the queue parameters.
#define DEMO_QUEUE_LENGTH            5
#define DEMO_QUEUE_ITEM_SIZE         sizeof( messageA_t)
#define SEND_EXAMPLE                 10

#define DEMO_TIMER_DURATION          100        // 100ms
queue_t dema_queue;

void timer_cb(unsigned long *param)
{
    os_timer_t *ptimer = *param;
    printf("Timeout, stop related timer.");
    wl_send_queue(dema_queue, NULL, portMAX_DELAY);
}

void taskWait(void *param)
{
    queue_t queue;
    uint32 message;
    // The queue handle is passed into this task as the task parameter. Cast the parameter back to a queue handle.
    queue = (queue_t)*param;
    os_timer_t *ptimer = NULL;

    wl_init_timer(&ptimer, timer_cb, &ptimer);
    for(;;) {
        wl_start_timer(ptimer, DEMO_TIMER_DURATION);
        // Wait for the maximum period for data to become available on the queue.
        if( wl_wait_queue(queue, &message, portMAX_DELAY) != 0 ) {
            if (message) {
                wl_stop_timer(ptimer);
            }
            // Nothing was received from the queue even after blocking to wait for data to arrive.
        }
        else {
            // message now contains the received data.
```

```
            }
        }
        wl_destory_timer(ptimer);
}

int main( void )
{
        int ret = 0;
        // Create the queue, storing the returned handle in the dema_queue variable.
        ret = wl_init_queue(&dema_queue, DEMO_QUEUE_LENGTH,
DEMO_QUEUE_ITEM_SIZE );
        if( -1 == ret) {
        // The queue could not be created.
        return;
        }

        // Rest of code goes here.
        wl_create_thread("task Wait", MAIN_TASK_STACK_SZ, MAIN_TASK_PRIO,
(thread_func_t)taskWait, &dema_queue);

        for(;;);
        // Free the queue
        wl_free_queue(&dema_queue);

}
```

2.1.40 void wl_start_timer(timer_t *ptimer, u32 ms)

```
/**
  * @brief    start timer handler and set delay time
  * @detail   This routine starts the timer and execute the callback function in every
             interval time.
  * @param    ptimer : pointer of timer to be started
  * @param    ms : delay time
  * @retval   None
  */
```

说明

启动定时器。

使用 *wl_init_timer* 初始化定时器后，在使用前调用 *wl_start_timer* 来启动定时器，

同时配置定时器的超时时间 ms，该定时器在 ms 毫秒后超时，并调用 *wl_init_timer* 中的定时器处理函数 pfunc。

参数

ptimer：定时器句柄，*wl_init_timer* 初始化的定时器，其他相关的接口，*wl_start_timer, wl_stop_timer, wl_destroy_timer* 等等都需要使用该定时器的句柄，才能操作对应的定时器，

ms：定时时间，即每隔 ms 时间执行一次初始化中的回调函数

返回

无

示例

参考 *wl_init_timer* 示例。

2.1.41 void wl_stop_timer(timer_t *ptimer)

```
/**
 * @brief      stop timer handler
 * @detail     This routine stops the timer.
 * @param      ptimer : pointer of timer to be stopped
 * @retval     None
 */
```

说明

定时器停止。

在定时器超时之前，如果需要关闭，可以调用 wl_stop_timer 来停止定时器。

参数

ptimer：定时器句柄，*wl_init_timer* 初始化的定时器，其他相关的接口，*wl_start_timer, wl_stop_timer, wl_destroy_timer* 等等都需要使用该定时器的句柄，才能操作对应的定时器

返回

无

示例

参考 *wl_init_timer* 示例。

2.1.42 void wl_destory_timer(timer_t *ptimer)

```
/**
  * @brief      delete timer handler
  * @detail     This routine delete the timer.
  * @param      ptimer : pointer of timer to be deleted
  * @retval     None
  */
```

说明

销毁定时器。

在定时器在在需要时候，需要进行销毁，释放初始化时候占用的系统资源和内存空间。

参数

ptimer：定时器句柄，*wl_init_timer* 初始化的定时器，其他相关的接口，*wl_start_timer, wl_stop_timer, wl_destroy_timer* 等等都需要使用该定时器的句柄，才能操作对应的定时器

返回

无

示例

参考 *wl_init_timer* 示例。

2.1.43 u32 wl_get_freeheapsize(void)

```
/**
  * @brief      system dynamic memory heap left
  * @detail     This routine gets the unused memory heap.
  * @param      None
  * @retval     None
  */
```

说明

获取可用堆大小。

参数

无

返回

可用堆大小，字节为单位。

示例

```
void taskA(void* context)
{
      thread_hdl_t thread_hdl = NULL;
        while(1){
              wl_os_mdelay(20000);
               // List current heap size
               printf("Current heap size[%x].", wl_get_freeheapsize());
        }
         // Destroy itself
         thread_hdl = wl_current_threadid;
         wl_destroy_thread(thread_hdl);

}
```

2.1.44 void* wl_malloc(u32 sz)

```
/**
   * @brief      allocate memory
   * @detail     This routine allocates memory.
   * @param      sz : size of memeroy will be allocated
   * @retval     if success ,return    pointer to allocated memory , or return NULL
   */
```

说明

申请内存空间。

根据 sz 参数，申请并获取特定大小的内存空间。具体的内存分配方式，根据系统配置的不同，会有差异。

使用该类 API（对应的还有 *wl_zmalloc*，*wl_realloc*）申请的内存，在不需要的时候，需要使用 *wl_free* 进行释放，否则该内存将会被一直占用。

类似于 *wl_zmalloc*，两者都是申请 sz 字节的内存，不同在于，*wl_malloc* 申请的内存，其默认值是不确定的，该内存空间的内容为上次使用时候遗留的无效值；*wl_zmalloc* 申请的内存，系统会负责将对应内存区初始化为 0。

参数

sz：本次需要申请的内存大小，单位为字节。

返回

NULL：如果内存申请失败，返回 NULL
其他值：如果内存申请成功，返回该片内存的首地址

示例

```
#define TEST_MEM_SIZE          32
void funcA()
{
    char *pdata = NULL;

    // Malloc memory
    pdata = wl_malloc(TEST_MEM_SIZE);
    if (NULL == pdata) {
        printf("Malloc memory fail.\n");
        return;
    }
    else {
        memset(pdata, 0, TEST_MEM_SIZE);
        // Do somethings using pdata.
    }

    // Delete memory
    wl_free(pdata);
}
```

2.1.45 void* wl_zmalloc(u32 sz)

```
/**
 * @brief     allocate and zero memory
 * @detail    This routine allocates memory and zeros it.
 * @param     sz :how many source buffer length will be allocated and zero
 * @retval    if success ,return    pointer to allocated memory , or return NULL
 */
```

说明

申请特定大小的内存空间，并将该片内存初始化为 0。

根据 sz 参数，申请并获取特定大小的内存空间。具体的内存分配方式，根据系统配置的不同，会有差异。

使用该类 API（对应的还有 *wl_malloc*，*wl_realloc*）申请的内存，在不需要的时候，需要使用 *wl_free* 进行释放，否则该内存将会被一直占用。

类似于 *wl_zmalloc*，两者都是申请 sz 字节的内存，不同在于，*wl_malloc* 申请的内存，其默认值是不确定的，该内存空间的内容为上次使用时候遗留的无效值；*wl_zmalloc* 申请的内存，系统会负责将对应内存区初始化为 0。

参数

sz：本次需要申请的内存大小，单位为字节。

返回

NULL：如果内存申请失败，返回 NULL

其他值：如果内存申请成功，返回该片内存的首地址

示例

```
#define TEST_MEM_SIZE          32
void funcA()
{
    char *pdata = NULL;

    // Malloc memory
    pdata = wl_zmalloc(TEST_MEM_SIZE);
    if (NULL == pdata) {
        printf("Malloc memory fail.\n");
        return;
    }
    else {
        // Do somethings using pdata.
    }

    // Delete memory
    wl_free(pdata);
}
```

2.1.46 void* wl_realloc(void *ptr, u32 sz)

```
/**
  * @brief wl_zmalloc:    reallocate and zero memory
  * @param    sz :how many source buffer length will be allocated and zero
  * @retval if success ,return    pointer to reallocated memory , or return NULL
```

*/

说明

重新申请特定大小的内存空间，并复制 ptr 中的数据到新申请的内存空间。

根据 sz 参数，申请并获取特定大小的内存空间。具体的内存分配方式，根据系统配置的不同，会有差异。

使用该类 API（对应的还有 *wl_malloc, wl_zmalloc*）申请的内存，在不需要的时候，需要使用 *wl_free* 进行释放，否则该内存将会被一直占用。

该 API 不同于之前两个 APIs（*wl_malloc*，*wl_zmalloc*），如果内存空间申请成功，则会使用原 ptr 内存中的数据来填充新申请的内存空间。

如果传入的 sz 大小为 0，则该 API 的功能类似于 *wl_free*，将释放 ptr 指向的内存空间；

如果传入的 sz 大小不为 0，而且 sz 小于 ptr 指向的内存空间大小，则会将 ptr 指向的内存中前 sz 字节的数据复制到新申请的内存；

如果传入的 sz 大小部位 0，而且 sz 大于/等于 ptr 指向的内存空间的大小 ptr_sz（比如 ptr=*wl_malloc*(ptr_sz)），则会将 ptr 指向的内存空间中 ptr_sz 字节的数据复制到新申请的内存空间（新申请的内存中最后面的 sz-ptr_sz 字节的数据是未知的）

参数

ptr：指向原内存空间的首地址。

sz：新申请内存的大小

返回

无

备注

如果 sz 大于 ptr 指向的原内存的大小 ptr_sz，则新申请内存的最后 sz-ptr_sz 字节的数据是未知的，是随机值

该 API 调用后，ptr 指向的原内存会被释放，以后不能再继续使用

示例

```
#define TEST_MEM_SIZE          32
void funcA()
{
    char *pdata = NULL;
    char *pdata_new = NULL;

    // Malloc memory
    pdata = wl_zmalloc(TEST_MEM_SIZE);
    if (NULL == pdata) {
        printf("Malloc memory fail.\n");
        return;
```

```
    }

    pdata_new = wl_realloc(pdata, TEST_MEM_SIZE*2);
    if (NULL == pdata_new) {
        printf("Realloc memory fail.\n");
        return;
    }
    // Do somethings using pdata_new.


    // Delete memory
    wl_free(pdata_new);
}
```

2.1.47 void wl_free(void *pbuf)

```
/**
  * @brief     free allocated memory
  * @detail    This routine frees allocated memory.
  * @param     pbuf :array will be free
  * @retval    None
  */
```

说明

释放内存。

使用 *wl_malloc* 或者 *wl_zmalloc* 获取的内存，在不再使用的时候，需要调用 *wl_free* 释放对应的内存。

参数

pbuf：指向需要释放的内存空间的首地址。

返回

无

示例

参考 *wl_malloc, wl_zmalloc, wl_realloc* 示例。

2.1.48 int wl_get_random32(void)

```
/**
  * @brief      get random number
  * @detail     This routine gets random number.
  * @param      None
  * @retval     the generated random number
  */
```

说明：获取随机数
参数：无
返回：生成的随机数

说明

获取随机数。
该 API 用于生成并获取一个 32 位的随机数。

参数

无

返回

生成的 32 位随机数。

示例

```
void funcA(void)
{
    uint32 delay_duration = 0;

    delay_duration = wl_get_random32();
    wl_os_mdelay(delay_duration);

    // Do somethings else...
}
```

2.1.49 void wl_os_init(void *svc_func, void *pendsv_func, void* system_tick_func)

```
/**
  * @brief      init handler for OS
  * @detail     This routine gets random number.
```

```
 * @param    None
 * @retval   None
 */
```

说明

操作系统初始化函数。

在系统启动之前，需要调用该 API，进行 OS 的相关初始化配置。

之前配置相关其他模块、相关线程，调用 *wl_os_start* 来启动 OS（包括系统的调度器）

参数

svc_func：传入的 system service 处理任务的句柄

pendsv_func：传入的 pend service 处理任务的句柄

system_tick_func：传入的 system tick 处理任务的句柄

返回

无

示例

```
static void main_entry(void)
{
    // Init bss

    // Init system

    // Init OS
    wl_os_init(vPortSVCHandler, xPortPendSVHandler, xPortSysTickHandler);

    // Start wlan modules

    // Start main task

    // Start OS
    wl_os_start();

    //nerver come here
    while(1);
}
```

2.1.50 void wl_os_start(void)

```
/**
  * @brief     enable the schedule , start the RTOS kernel
  * @detail    This function is used to initialize the internals of RTOS and MUST be called
               prior to creating any RTOS object and, prior to calling wl_os_start().
  * @param     None
  * @retval    None
  */
```

说明

操作系统任务启动函数。

在 OS 启动之前，系统由默认的 main 函数入口执行相关的配置（根据不同的平台和系统，入口函数会有所不同）；在 OS 启动之后，OS 的调度器启动，后续的执行由调度器来完成，只有中断和线程会处于运行状态。

在 OS 启动时候，会选择之前创建的、最高优先级的、处于就绪状态的任务开始执行。

参数

无

返回

无

示例

参考 *wl_os_init* 示例。

2.2 HAL

2.2.1 UART

- hal_status_e hal_uart_init(uart_hdl_t *uart);
- hal_status_e hal_uart_deinit(uart_hdl_t *uart);
- int hal_uart_tx(uart_hdl_t *uart, u8 *pbuf, uint16_t size, uint32_t timeout);
- int hal_uart_rx(uart_hdl_t *uart, u8 *pbuf, uint16_t size, uint32_t timeout);
- hal_status_e hal_uart_tx_it(uart_hdl_t *uart, u8 *pbuf, uint16_t size);

- hal_status_e hal_uart_rx_it(uart_hdl_t *uart, u8 *pbuf, uint16_t size);
- hal_status_e hal_uart_rx_it_to(uart_hdl_t *uart, u8*pbuf);
- hal_status_e hal_uart_tx_dma(uart_hdl_t *uart, u8 *pbuf, uint16_t size);
- hal_status_e hal_uart_rx_dma(uart_hdl_t *uart, u8 *pbuf, uint16_t size);
- u32 hal_uart_rx_dma_adddress(uart_hdl_t *uart);
- hal_status_e hal_uart_dma_txstop(uart_hdl_t *uart);
- hal_status_e hal_uart_dma_rxstop(uart_hdl_t *uart);

2.2.1.1 hal_status_e hal_uart_init(uart_hdl_t *uart);

参数:

[输入] uart 串口实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 1

注解:

功能：用户使用此函数初始化并启动一个 S907A 串口设备，使用之前创建一个 **uart_hdl_t** 类型的串口实例，并根据用户自己需求对这个串口实例进行相应设置（S907x 串口支持轮询，中断，DMA），再调用此函数，将上述的串口实例指针作为参数传入即可；返回值为 HAL_OK 代表操作成功，如为其它值，则代表初始化并启动该设备失败，根据相应值自检

***本文当中凡是像 uart_hdl_t 以 _t 结尾的自定义类型请查看第三章节结构体**

2.2.1.2 hal_status_e hal_uart_deinit(uart_hdl_t *uart);

参数:

[输入] uart 串口实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 2

注解:

功能：用户调用此函数将串口恢复为缺省值

2.2.1.3 int hal_uart_tx(uart_hdl_t *uart, u8 *pbuf, uint16_t size, uint32_t timeout);

参数:

[输入]	uart	串口实例指针
[输入]	pbuf	要写入设备的数据缓冲区指针
[输入]	size	要写入的字节个数
[输入]	timeout	写入延时

返回:

实际发送出去的字节数

示例:

example 1

注解:

功能：用户调用此函数可以通过轮询的方式去发送固定字节的数据，并且支持发送延时

2.2.1.4 int hal_uart_rx(uart_hdl_t *uart, u8 *pbuf, uint16_t size, uint32_t timeout);

参数:

[输入]	uart	串口实例指针

[输出] pbuf 存放读取数据的缓冲区指针

[输入] size 最多要读取的字节个数

[输入] timeout 读取延时

返回:

实际读取到的字节数

示例:

example 1

注解:

功能：用户调用此函数可以通过轮询的方式去接收固定字节的数据，并且支持最长等待时间

2.2.1.5 hal_status_e hal_uart_tx_it(uart_hdl_t *uart, u8 *pbuf, uint16_t size);

参数:

[输入] uart 串口实例指针

[输入] pbuf 要写入设备的数据缓冲区指针

[输入] size 要写入的字节个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 2

注解:

功能：用户调用此函数可以通过中断的方式向串口设备写入固定的字节数，并且支持写入完成回调，用户可以在 static void txdone_cb(void * context){}中自定义回调函数

2.2.1.6 hal_status_e hal_uart_rx_it(uart_hdl_t *uart, u8 *pbuf,

uint16_t size);

参数:

[输入] uart 串口实例指针

[输出] pbuf 存放读取数据的缓冲区指针

[输入] size 最多要读取的字节个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 2

注解:

功能：用户调用此函数可以通过中断的方式接收串口固定字节的数据，并且支持接收完成回调，用户可以在 static void redone_cb(void * context){}自定义回调函数体

2.2.1.7 hal_status_e hal_uart_rx_it_to(uart_hdl_t *uart, u8*pbuf);

参数:

[输入] uart 串口实例指针

[输出] pbuf 存放读取数据的缓冲区指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

注解:

<猜测功能>：在使用中断读取设备数据时，读取一个字节的数据，过了一个设定时间后未再接到数据，则认为接收到一个完整数据包，用户可以在 static void

rxtimeout_cb(void *context)实现对接收到的数据包进行解析

2.2.1.8 hal_status_e hal_uart_tx_dma(uart_hdl_t *uart, u8 *pbuf, uint16_t size);

参数:

[输入] uart 串口实例指针

[输入] pbuf 要写入设备的数据缓冲区指针

[输入] size 要写入的字节个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 3

注解:

功能：通过 dma 方式向外设写入固定字节的数据

2.2.1.9 hal_status_e hal_uart_rx_dma(uart_hdl_t *uart, u8 *pbuf, uint16_t size);

参数:

[输入] uart 串口实例指针

[输出] pbuf 存放读取数据的缓冲区指针

[输入] size 要读出的字节个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 3

注解:

功能：通过 dma 方式从外设读出固定字节的数据

2.2.1.10 u32 hal_uart_rx_dma_adddress(uart_hdl_t *uart);

参数:

[输入] uart 串口实例指针

返回:

返回一个外设用于 dma 收发的地址

示例:

example 3

注解:

调用此函数可以获取外设的地址

2.2.1.11 hal_status_e hal_uart_dma_txstop(uart_hdl_t *uart);

参数:

[输入] uart 串口实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4

注解:

功能：可以失能串口 dma 发送功能

2.2.1.12 hal_status_e hal_uart_dma_rxstop(uart_hdl_t *uart);

参数:

[输入] uart 串口实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4

注解：

功能：调用此函数可以失能串口 dma 接收功能

2.2.2 GPIO

- hal_status_e hal_gpio_init(gpio_hdl_t *gpio, gpio_init_t *init);
- hal_status_e hal_gpio_deinit(gpio_hdl_t *gpio);
- hal_status_e hal_gpio_write(gpio_hdl_t *gpio, gpio_status_e status);
- gpio_status_e hal_gpio_read(gpio_hdl_t *gpio);
- hal_status_e hal_gpio_togglepin(gpio_hdl_t *gpio);
- void hal_gpio_it_start(gpio_hdl_t *gpio, hal_int_cb cb, void *context);
- hal_gpio_it_stop(gpio_hdl_t *gpio);

2.2.2.1 hal_status_e hal_gpio_init(gpio_hdl_t *gpio, gpio_init_t *init);

参数：

[输入] gpio IO 口实例指针

[输入] init IO 配置参数指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 10 & example 11

注解：

功能：用户调用此函数实现对 IO 口的初始化

配置参数主要有

1，端口号

2,工作模式(mode)主要有：

GPIO_MODE_INPUT ---------- 输入

GPIO_MODE_OUTPUT ---------- 输出

GPIO_MODE_INT_RISING ---------- 上升沿促发

GPIO_MODE_INT_FALLING ---------- 下降沿促发

GPIO_MODE_INT_LEVEL_H ---------- 高电平促发

GPIO_MODE_INT_LEVEL_L ---------- 低电平促发

3,IO 口电平状态配置主要有三种状态：

GPIO_NOPULL ---------- 浮空

GPIO_PULLUP ---------- 上拉

GPIO_PULLDOWN ---------- 下拉

2.2.2.2 hal_status_e hal_gpio_deinit(gpio_hdl_t *gpio);

参数：

[输入] gpio IO 口实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 8

注解：

功能：用户调用此函数将某 IO 恢复为缺省值

2.2.2.3 hal_status_e hal_gpio_write(gpio_hdl_t *gpio, gpio_status_e status);

参数：

[输入] gpio IO 口实例指针

[输入] status 对 IO 口电平设定值

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 10 & example 11

注解：

功能：用户调用此函数对设定引脚 pin 的输出值（pin 需要被配置为 GPIO 输出功能）

status 只能是以下值：

GPIO_PIN_RESET　　代表低电平

GPIO_PIN_SET　　代表高电平

2.2.2.4　gpio_status_e hal_gpio_read(gpio_hdl_t *gpio);

参数：

[输入]　　gpio　　IO 口实例指针

返回:

返回只能是 GPIO_PIN_RESET　or　GPIO_PIN_SET

示例:

example 10 & example 11

注解：

功能：用户调用此函数读取某 IO 口的电平状态

2.2.2.5　hal_status_e　hal_gpio_togglepin(gpio_hdl_t *gpio);

参数：

[输入]　　gpio　　IO 口实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 10 & example 11

注解：

功能：用户调用此函数翻转指定 IO 电平值

2.2.2.6 void hal_gpio_it_start(gpio_hdl_t *gpio, hal_int_cb cb, void *context);

参数:

[输入] gpio IO 口实例指针

[输入] cb 中断触发的回调函数

[输入] context 任意类型的参数指针

返回:

无

示例:

example 11

注解:

功能：用户调用此函数可以开启某 IO 口的外部中断

用户可以在 void gpio_user_isr(void *context){}函数体中自定义回调功能

调用此函数前需要在 hal_status_e hal_gpio_init(gpio_hdl_t *gpio, gpio_init_t *init)配置中对 init->mode 配置一个外部中断的触发条件

init->mode 可配置的触发条件如下：

GPIO_MODE_INT_RISING 上升沿触发

GPIO_MODE_INT_FALLING 下降沿触发

GPIO_MODE_INT_LEVEL_H 高电平触发

GPIO_MODE_INT_LEVEL_L 低电平触发

2.2.2.7 hal_gpio_it_stop(gpio_hdl_t *gpio);

参数:

[输入] gpio IO 口实例指针

返回:

无

示例:

example 11

注解:

功能：用户调用此函数可以关闭某 IO 口的外部中断

2.2.3 SPI

- hal_status_e hal_spi_init(spi_hdl_t *spi);
- hal_status_e hal_spi_deinit(spi_hdl_t *spi);
- int hal_spi_master_txrx(spi_hdl_t *spi, void *txbuf, void *rxbuf, u16 xfer_size, uint32_t ms);
- int hal_spi_slaver_tx(spi_hdl_t* spi, void *txbuf, u16 xfer_size, uint32_t timeout);
- int hal_spi_slaver_rx(spi_hdl_t* spi, void *rxbuf, u16 xfer_size, uint32_t timeout);
- hal_status_e hal_spi_master_xfer_interrupt(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_spi_master_recv_interrupt(spi_hdl_t *spi, void *pbuf, u16 xfer_size);
- hal_status_e hal_spi_slaver_xfer_interrupt(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_spi_slaver_recv_interrupt(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_spi_master_xfer_dma(spi_hdl_t *spi, void *pbuf, u16 xfer_size);
- hal_status_e hal_spi_master_recv_dma(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_spi_slaver_xfer_dma(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_spi_slaver_recv_dma(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);

2.2.3.1 hal_status_e hal_spi_init(spi_hdl_t *spi);

参数:

[输入] spi SPI 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 7 & example 8 & example 9

注解:

功能：用户调用此函数初始化一个 SPI 主设备或者从设备

2.2.3.2 hal_status_e hal_spi_deinit(spi_hdl_t *spi);

参数:

[输入] spi SPI 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 7

注解:

功能：用户调用此函数将某 SPI 设备恢复位缺省状态

2.2.3.3 int hal_spi_master_txrx(spi_hdl_t *spi, void *txbuf, void *rxbuf, u16 xfer_size, uint32_t ms);

参数:

[输入] spi SPI 实例指针

[输入] txbuf 待写入数据在内存中的地址

[输出] rxbuf 接收数据的内存地址

[输入] xfer_size 要发送的数据的长度

[输入] ms 延时

返回:

实际要发送或接收数据的长度

示例:

见例

注解:

功能：用户调用此函数可以实现 SPI 主设备向 SPI 从设备发送固定字节，且主设备也收到从设备相同字节的数据

2.2.3.4 int hal_spi_slaver_tx(spi_hdl_t* spi, void *txbuf, u16 xfer_size, uint32_t timeout);

参数:

[输入]	spi	SPI 实例指针
[输入]	txbuf	待写入数据在内存中的地址
[输入]	xfer_size	要发送的数据的长度
[输入]	ms	发送延时

返回:

已成功发送出去的数据长度

示例:

见例

注解:

功能：用户调用此函数可以实现 SPI 从设备向 SPI 主设备发送固定字节数据

2.2.3.5 int hal_spi_slaver_rx(spi_hdl_t* spi, void *rxbuf, u16 xfer_size, uint32_t timeout);

参数:

[输入]	spi	SPI 实例指针
[输出]	rxbuf	接收数据的内存地址
[输入]	xfer_size	要接收的数据的长度

[输入]　　ms　　延时

返回:

实际接收 SPI 主设备数据的长度

示例:

见例 10

注解:

功能：用户调用此函数可以实现从设备接收主设备固定字节的数据，并返回实际接收的数据的长度

2.2.3.6　hal_status_e hal_spi_master_xfer_interrupt(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);

参数:

[输入]　　spi　　SPI 实例指针

[输入]　　pbuf　　待写入数据在内存中的地址

[输入]　　xfer_size　　要写入数据的长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 8

注解:

功能：用户调用此函数实现 SPI 主设备向从设备发送固定字节的数（通过中断）

2.2.3.7　hal_status_e hal_spi_master_recv_interrupt(spi_hdl_t *spi, void *pbuf, u16 xfer_size);

参数:

[输入]　　spi　　SPI 实例指针

[输出] pbuf 待接收数据的地址

[输入] xfer_size 要接收数据的长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 8

注解：

功能：用户调用此函数实现主设备接收从设备固定字节的数据（通过中断）

2.2.3.8 hal_status_e hal_spi_slaver_xfer_interrupt(spi_hdl_t *spi, u8 *pbuf,

u16 xfer_size);

参数：

[输入] spi SPI 实例指针

[输入] pbuf 待写入数据在内存中的地址

[输入] xfer_size 要写入的数据长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 10

注解：

功能：用户调用此函数实现从设备向主设备发送固定字节的数据（通过中断）

2.2.3.9 hal_status_e hal_spi_slaver_recv_interrupt(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);

参数：

[输入] spi SPI 实例指针

[输出] pbuf 待接收数据的地址

[输入] xfer_size 要接收数据的长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 10

注解:

功能：用户调用此函数实现从设接收主设备固定字节的数据（通过中断）

2.2.3.10 hal_status_e hal_spi_master_xfer_dma(spi_hdl_t *spi, void *pbuf, u16 xfer_size);

参数:

[输入] spi SPI 实例指针

[输入] pbuf 待写入数据在内存中的地址

[输入] xfer_size 要写入数据的长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example9

注解:

功能：用户调用此函数实现 SPI 主设备向从设备发送固定字节的数（通过 DMA）

2.2.3.11 hal_status_e hal_spi_master_recv_dma(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);

参数:

[输入] spi SPI 实例指针

[输出] pbuf 待接收数据的地址

[输入] xfer_size 要接收数据的长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example9

注解:

功能：用户调用此函数实现主设备接收从设备固定字节的数据（通过 dma）

2.2.3.12 hal_status_e hal_spi_slaver_xfer_dma(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);

参数:

[输入]	spi	SPI 实例指针
[输入]	pbuf	待写入数据在内存中的地址
[输入]	xfer_size	要写入的数据长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 10

注解:

功能：用户调用此函数实现从设备向主设备发送固定字节的数据（通过 dma）

2.2.3.13 hal_status_e hal_spi_slaver_recv_dma(spi_hdl_t *spi, u8 *pbuf, u16 xfer_size);

参数:

[输入]	spi	SPI 实例指针
[输入]	pbuf	待接收数据的地址
[输入]	xfer_size	要接收数据的长度

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 10

注解:

功能：用户调用此函数实现从设接收主设备固定字节的数据（通过 dma）

2.2.4 I2C

- hal_status_e hal_i2c_init(i2c_hdl_t *i2c);
- hal_status_e hal_i2c_deinit(i2c_hdl_t *i2c);
- hal_status_e hal_i2c_master_xfer(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);
- hal_status_e hal_i2c_master_recv(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);
- hal_status_e hal_i2c_slavor_xfer(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);
- hal_status_e hal_i2c_slavor_recv(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);
- hal_status_e hal_i2c_master_xfer_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_master_recv_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_slavor_xfer_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_slavor_recv_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_master_xfer_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_master_recv_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_slavor_xfer_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);
- hal_status_e hal_i2c_slavor_recv_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

2.2.4.1 hal_status_e hal_i2c_init(i2c_hdl_t *i2c);

参数:

[输入] i2c i2c 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4 & example 5 & example 6

注解:

功能：用户调用此函数实现对 I2C 设备的初始化

2.2.4.2 hal_status_e hal_i2c_deinit(i2c_hdl_t *i2c);

参数:

[输入] i2c i2c 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 11

注解:

功能：用户调用此函数可将 I2C 设备恢复为缺省值

2.2.4.3 hal_status_e hal_i2c_master_xfer(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);

参数:

[输入] i2c i2c 实例指针

[输入] pbuf 数据缓冲区，要写的数据存放于此

[输入] xfer_size 要写入的数据个数

[输入] ms 写入延时

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4

注解:

功能：用户调用此函数实现 I2C 主设备向从设备发送固定字节的数据，支持发送延时

2.2.4.4 hal_status_e hal_i2c_master_recv(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);

参数：

[输入]	i2c	i2c 实例指针
[输出]	pbuf	数据缓冲区，读取的数据存放于此
[输入]	xfer_size	要读出的数据个数
[输入]	ms	读取延时

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4

注解：

功能：用户调用此函数实现 I2C 主设备读取从设备固定字节的数据

2.2.4.5 hal_status_e hal_i2c_slavor_xfer(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);

参数：

[输入]	i2c	i2c 实例指针
[输入]	pbuf	数据缓冲区，要写的数据存放于此
[输入]	xfer_size	要写入的数据个数
[输入]	ms	写入延时

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

参照 example 4 中 hal_i2c_master_xfer 使用

注解：

功能：用户调用此函数实现 I2C 从设备向主设备发送固定字节的数据，并支持写入延

时

2.2.4.6 hal_status_e hal_i2c_slavor_recv(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size, uint32_t ms);

参数:

[输入]	i2c	i2c 实例指针
[输出]	pbuf	数据缓冲区，读取的数据存放于此
[输入]	xfer_size	要读出的数据个数
[输入]	ms	读出延时

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

参照例 4 中 hal_i2c_master_recv 的使用

注解:

功能：用户调用此函数实现 I2C 从设备读取主设备固定字节的数据

2.2.4.7 hal_status_e hal_i2c_master_xfer_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入]	i2c	i2c 实例指针
[输入]	pbuf	数据缓冲区，要写的数据存放于此
[输入]	xfer_size	要写入的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4

注解:

功能：用户调用此函数实现 I2C 主设备向从设备发送固定字节的数据（通过中断）

2.2.4.8 hal_status_e hal_i2c_master_recv_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入] i2c i2c 实例指针

[输出] pbuf 数据缓冲区，读取的数据存放于此

[输入] xfer_size 要读出的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 4

注解:

功能：用户调用此函数实现 I2C 主设备读取从设备固定字节的数据（通过中断）

2.2.4.9 hal_status_e hal_i2c_slavor_xfer_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入] i2c i2c 实例指针

[输入] pbuf 数据缓冲区，要写的数据存放于此

[输入] xfer_size 要写入的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

参照例 5 中 hal_i2c_master_xfer_interrupt 使用

注解:

功能：用户调用此函数实现 I2C 从设备向主设备发送固定字节的数据（通过中断）

2.2.4.10 hal_status_e hal_i2c_slavor_recv_interrupt(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入] i2c i2c 实例指针

[输出] pbuf 数据缓冲区，读取的数据存放于此

[输入] xfer_size 要读出的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

参照例 5 中 hal_i2c_master_recv_interrupt 使用

注解:

功能：用户调用此函数实现 I2C 从设备读取主设备固定字节的数据（通过中断）

2.2.4.11 hal_status_e hal_i2c_master_xfer_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入] i2c i2c 实例指针

[输入] pbuf 数据缓冲区，要写的数据存放于此

[输入] xfer_size 要写入的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 6

注解:

功能：用户调用此函数实现 I2C 主设备向从设备发送固定字节的数据（通过 DMA）

2.2.4.12 hal_status_e hal_i2c_master_recv_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入]	i2c	i2c 实例指针
[输出]	pbuf	数据缓冲区，读取的数据存放于此
[输入]	xfer_size	要读出的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 6

注解:

功能：用户调用此函数实现 I2C 主设备读取从设备固定字节的数据（通过 DMA）

2.2.4.13 hal_status_e hal_i2c_slavor_xfer_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入]	i2c	i2c 实例指针
[输入]	pbuf	数据缓冲区，要写的数据存放于此
[输入]	xfer_size	要写入的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

参照例 6 中 hal_i2c_master_xfer_dma 使用

注解:

功能：用户调用此函数实现 I2C 从设备向主设备发送固定字节的数据（通过 DMA）

2.2.4.14 hal_status_e hal_i2c_slavor_recv_dma(i2c_hdl_t *hi2c, u8 *pbuf, u16 xfer_size);

参数:

[输入]	i2c	i2c 实例指针
[输出]	pbuf	数据缓冲区，读取的数据存放于此
[输入]	xfer_size	要读出的数据个数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

参照例 6 中 hal_i2c_master_recv_dma 使用

注解:

功能：用户调用此函数实现 I2C 从设备读取主设备固定字节的数据（通过 DMA）

2.2.5 ADC

- hal_status_e hal_adc_init(adc_hdl_t *adc);
- hal_status_e hal_adc_deinit(adc_hdl_t *adc);
- hal_status_e hal_adc_start(adc_hdl_t *adc);
- hal_status_e hal_adc_stop(adc_hdl_t *adc);
- hal_status_e hal_adc_poll_oneshot(adc_hdl_t *adc, u32 timeout);
- hal_status_e hal_adc_poll_continous(adc_hdl_t *adc);
- hal_status_e hal_adc_interrupt_oneshot(adc_hdl_t *adc,
  hal_int_cb cb, void *arg);
- hal_status_e hal_adc_interrupt_continous(adc_hdl_t *adc,
  hal_int_cb cb, void *arg);

2.2.5.1 hal_status_e hal_adc_init(adc_hdl_t *adc);

参数:

[输入] adc ADC 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 17

注解:

功能：用户调用此函数对 ADC 接口初始化

2.2.5.2 hal_status_e hal_adc_deinit(adc_hdl_t *adc);

参数:

[输入] adc ADC 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 17

注解:

功能：用户调用此函数将 ADC 接口恢复为缺省值

2.2.5.3 hal_status_e hal_adc_start(adc_hdl_t *adc);

参数:

[输入] adc ADC 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 17

注解：

功能：用户调用此函数将开启 ADC 采样

2.2.5.4 hal_status_e hal_adc_stop(adc_hdl_t *adc);

参数：

[输入] adc ADC 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 17

注解：

功能：用户调用此函数将关闭 ADC 采样

2.2.5.5 hal_status_e hal_adc_poll_oneshot(adc_hdl_t *adc, u32 timeout);

参数：

[输入] adc ADC 实例指针

[输入] timeout 延时

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 17

注解：

功能：用户调用此函数采集一次 adc 数据（本系统支持双 adc 采集，该函数可以一次采集两个 adc 通道的数据）

2.2.5.6 hal_status_e hal_adc_poll_continous(adc_hdl_t *adc);

参数:

[输入] adc ADC 实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 18

注解:

功能：用户调用此函数持续采集 adc 数据

使用此功能前需要将 adc 配置中的 oneshot.enable 位置 0

2.2.5.7 hal_status_e hal_adc_interrupt_oneshot(adc_hdl_t *adc, hal_int_cb cb, void *arg);

参数:

[输入] adc ADC 实例指针

[输入] cb 回调函数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 19

注解:

功能：用户调用此函数采集一次 adc 数据，并会开启采集完成回调处理

用户可以在 static void adc_oneshot_poll_isr(void *context){}在对采集到的数据进行处理

2.2.5.8 hal_status_e hal_adc_interrupt_continous(adc_hdl_t *adc, hal_int_cb cb, void *arg);

参数:

[输入] adc ADC 实例指针

[输入] cb 回调函数

[输入] arg 任意类型指针参数

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 19

注解:

功能：用户调用此函数连续采集 adc 数据，用户可以在回调函数 static void adc_read_isr(void *context){}函数体中对采集的数据进行处理

2.2.6 TIMER

- u32 hal_timer_get_counter(timer_hdl_t *tim);
- hal_status_e hal_timer_base_init(timer_hdl_t *tim);
- hal_status_e hal_timer_base_deinit(timer_hdl_t *tim);
- hal_status_e hal_timer_start_base(timer_hdl_t *tim);
- hal_status_e hal_timer_stop(timer_hdl_t *tim);
- hal_status_e hal_timer_set_period(timer_hdl_t *tim, u32 period);
- hal_status_e hal_timer_pwm_init(timer_hdl_t *tim);
- hal_status_e hal_timer_pwm_deinit(timer_hdl_t *tim);
- hal_status_e hal_timer_start_pwm(timer_hdl_t *tim);
- hal_status_e hal_timer_stop_pwm(timer_hdl_t *tim);
- hal_status_e hal_timer_start_pwm_dma(timer_hdl_t *tim);

- hal_status_e hal_timer_stop_pwm_dma(timer_hdl_t *tim);
- hal_status_e hal_timer_pwm_set_ccr(timer_hdl_t *tim, u32 ccr,

u8 channel);
- hal_status_e hal_timer_capture_init(timer_hdl_t *tim);
- hal_status_e hal_timer_start_capture(timer_hdl_t *tim);
- hal_status_e hal_timer_start_capture_dma(timer_hdl_t *tim);
- hal_status_e hal_timer_stop_capture_dma(timer_hdl_t *tim);

2.2.6.1 u32 hal_timer_get_counter(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回值为定时器的即时的计数值

示例:

见例 5

注解：

功能：获取定时器的当前计数值

2.2.6.2 hal_status_e hal_timer_base_init(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 3

注解：

用户调用此函数可以初始化一个普通定时器

调用之前需要定义一个 timer_hdl_t 类型的定时器实例

然后需要配置的初始化信息有：

tim->config.prescaler = ***； //分频系数

tim->config.period ***； //预装值

tim->config.int_enable = ***; //中断使能位 0 or 1

tim->it.basic_user_cb.func = timer_basic_interrupt;

tim->it.basic_user_cb.context = tim;

（tim 为指向该定时器实例 timer_hdl_t 类型指针）

如需求定时器中断，在配置时注意 **tim->config.int_enable = 1**

可以在中断处理函数 **void timer_basic_interrupt(void *context){}**函数体中实现用户需求

最后调用 **hal_timer_base_init** tim 作为参数传入，将以上配置信息写入寄存器

最后的最后再调用 hal_timer_start_base 开启定时器

2.2.6.3 hal_status_e hal_timer_base_deinit(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 5

注解：

调用此函数将定时器恢复为缺省值

2.2.6.4 hal_status_e hal_timer_start_base(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 3

注解:

调用此函数开启定时器

2.2.6.5 hal_status_e hal_timer_stop(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 3

注解:

调用此函数关闭定时器

2.2.6.6 hal_status_e hal_timer_set_period(timer_hdl_t *tim, u32 period);

参数:

[输入] tim 定时器实例指针

[输入] period 定时器预装值

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 5

注解:

用户调用此函数可以对定时器预装值设置

2.2.6.7 hal_status_e hal_timer_pwm_init(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解:

用户调用此函数可以实现定时器 PWM 输出初始化

2.2.6.8 hal_status_e hal_timer_pwm_deinit(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解:

用户调用此函数将 tim pwm 恢复为缺省值

2.2.6.9 hal_status_e hal_timer_start_pwm(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解：

用户调用此函数开启 tim+pwm 正常模式

2.2.6.10 hal_status_e hal_timer_stop_pwm(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解：

用户调用此函数关闭 tim+pwm 正常模式

2.2.6.11 hal_status_e hal_timer_start_pwm_dma(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解：

用户调用此函数开启 tim+pwm+dma 模式

2.2.6.12 hal_status_e hal_timer_stop_pwm_dma(timer_hdl_t *tim);

参数：

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解:

用户调用此函数关闭 tim+pwm+dma 模式

2.2.6.13 hal_status_e hal_timer_pwm_set_ccr(timer_hdl_t *tim, u32 ccr, u8 channel);

参数:

[输入] tim 定时器实例指针

[输入] ccr pulse 的值

[输入] channel 定时器 PWM 模式

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 6

注解:

用户调用此函数设定 pulse 值，和 PWM 模式

2.2.6.14 hal_status_e hal_timer_capture_init(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 7

注解:

用户调用此函数可以实现定时器的输入捕获初始化

2.2.6.15 hal_status_e hal_timer_start_capture(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 7

注解:

用户调用此函数开启输入捕获模式

2.2.6.16 hal_status_e hal_timer_start_capture_dma(timer_hdl_t *tim);

参数:

[输入] tim 定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 7

注解:

开启定时器的输入捕获 dma 模式

2.2.6.17 hal_status_e hal_timer_stop_capture_dma(timer_hdl_t *tim);

参数:

[输入]　　tim　　定时器实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

见例 7

注解:

用户调用此函数停止定时器输入捕获 dma 功能

2.2.7　RTC

- hal_status_e hal_rtc_init(rtc_hdl_t *rtc);
- hal_status_e hal_rtc_deinit(rtc_hdl_t *rtc);
- void hal_rtc_get_time(rtc_hdl_t *rtc);
- void hal_rtc_get_alarm(rtc_hdl_t *rtc);
- void hal_rtc_set_unixtime(rtc_hdl_t *rtc);
- void hal_rtc_set_basictime(rtc_hdl_t *rtc);

2.2.7.1　hal_status_e hal_rtc_init(rtc_hdl_t *rtc);

参数:

[输入]　　rtc　　时钟实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 12

注解:

功能：用户调用此函数初始化本地时钟（调用此函数前必须设置所在地理时区）

2.2.7.2 hal_status_e hal_rtc_deinit(rtc_hdl_t *rtc);

参数:

[输入] rtc 时钟实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

示例:

example 13

注解:

功能：用户调用此函数将时钟恢复为缺省值

2.2.7.3 void hal_rtc_get_time(rtc_hdl_t *rtc);

参数:

[输入] rtc 时钟实例指针

返回:

无

示例:

example 12

注解:

功能:用户调用此函数读取 RTC 时间

2.2.7.4 void hal_rtc_get_alarm(rtc_hdl_t *rtc);

参数:

[输入] rtc 时钟实例指针

返回:

无

示例:

example 12

注解：

功能：用户调用此函数获取闹钟时间

2.2.7.5 void hal_rtc_set_unixtime(rtc_hdl_t *rtc);

参数：

[输入] rtc 时钟实例指针

返回:

无

示例:

参照 hal_rtc_set_basictime 使用

注解：

功能：用户调用此函数设置 unix 时间

```
typedef struct system_time_
{
    u16 year; //from 1900
    u8 month; //1-12
    u8 day;    //1-31
    u8 hour;   //0~23
    u8 min;    //0~59
    u8 sec;    //0~59
    u8 week;   //0~6   sunday 0 monday 1 ...
    u32 hw_time;//unix time
}system_time_t;
```

hw_time 代表 unix 时间

在调用该函数设置之前需要对 rtc 下的 hw_time 赋值

2.2.7.6 void hal_rtc_set_basictime(rtc_hdl_t *rtc);

参数：

[输入] rtc 时钟实例指针

返回:

无

示例:

example 12

注解:

功能：用户调用此函数设置 RTC 时间

2.2.8 WDG

- hal_status_e hal_wdg_deinit(wdg_hdl_t *wdg);
- hal_status_e hal_wdg_init(wdg_hdl_t *wdg);
- void hal_wdg_start(wdg_hdl_t *wdg);
- void hal_wdg_stop(wdg_hdl_t *wdg);
- void hal_wdg_start_it(wdg_hdl_t *wdg);
- void hal_wdg_refresh(wdg_hdl_t *wdg);

2.2.8.1 hal_status_e hal_wdg_init(wdg_hdl_t *wdg);

参数:

[输入] wdg 看门狗实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

3.5 SDK 中不可见返回形态,在头文件中没声明该函数

示例:

见例

注解:

功能：用户调用此函数初始化看门狗

2.2.8.2 hal_status_e hal_wdg_start(wdg_hdl_t *wdg);

参数:

[输入] wdg 看门狗实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

3.5 SDK 中不可见返回形态,在头文件中没声明该函数

示例:

见例

注解:

功能：用户调用此函数可开启看门狗

2.2.8.3 hal_status_e hal_wdg_stop (wdg_hdl_t *wdg);

参数:

[输入] wdg 看门狗实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

3.5 SDK 中不可见返回形态,在头文件中没声明该函数

示例:

见例

注解:

功能：用户调用此函数可关闭看门狗

2.2.8.4 hal_wdg_start_it(&wdg_hdl);

参数:

[输入]　　wdg　　看门狗实例指针

返回:

返回 HAL_OK 代表操作成功，其它值代表失败，失败原因查看 ERRNO.

3.5 SDK 中不可见返回形态,在头文件中没声明该函数

示例:

见例

注解:

功能：用户调用此函数可开启看门狗中断

2.2.9 sleep

- hal_status_e hal_sleep_wake_config(ps_wake_up_mode_e event,u32 val);
- hal_status_e hal_sleep_event_clear(ps_wake_up_mode_e event);
- hal_status_e hal_sleep_enter(void);
- hal_status_e hal_dsleep_wake_config(dsleep_wake_up_mode_e event,u32 val);
- hal_status_e hal_deepsleep_enter(void);
- hal_status_e hal_wake_handle(void);

2.2.9.1 hal_status_e hal_sleep_wake_config(ps_wake_up_mode_e event,u32 val)

```
/**
  * @brief     sleep mode wake source config
  * @detail    Config sleep mode wake event and interrupt
  * @param     event: chose wake event.
  * @param     val: config val in different event.
  * @retval    HAL_OK          = 0x00,
               HAL_ERROR       = 0x01,
               HAL_BUSY        = 0x02,
               HAL_TIMEOUT     = 0x03
```

说明：配置 sleep 唤醒模式

参数：　　event：用于选择 wake event

　　val：用于不同 wake event 下的配置参数

返回： 0 ： success
1 ： error
2 ： busy
3 ： timeout

2.2.9.2 hal_status_e hal_sleep_event_clear(ps_wake_up_mode_e event)

```
/**
  * @brief     sleep mode clear wake event
     * @detail   This func only used in sleep mode，when the device was waked，this function
        should be called to clear the wake event parameters.
  * @param    event: chose wake event.
  * @retval   HAL_OK          = 0x00,
              HAL_ERROR       = 0x01,
              HAL_BUSY        = 0x02,
              HAL_TIMEOUT     = 0x03

  */
```

说明：清除已经配置的 wake event 参数，这个函数只在 sleep 模式下调用，当设备被唤醒时，需要调用此函数。
参数： event : 用于选择 wake event
返回： 0 ： success
1 ： error
2 ： busy
3 ： timeout

2.2.9.3 hal_status_e hal_sleep_enter(void)

```
/**
  * @brief     Enter sleep mode
     * @detail   This function is used to enter sleep mode after wake config.
  * @param    null
  * @retval   HAL_OK          = 0x00,
              HAL_ERROR       = 0x01,
              HAL_BUSY        = 0x02,
              HAL_TIMEOUT     = 0x03
  */
```

说明：此函数用于进入 sleep mode。必须在成功执行 wake config 之后运行。
参数：无

返回：0 ： success
1 ： error
2 ： busy
4 ： timeout

2.2.9.4 hal_status_e hal_dsleep_wake_config(dsleep_wake_up_mode_e event,u32 val)

```
/**
  * @brief      deepsleep mode wake source config
    * @detail     Config deepsleep mode wake event and interrupt
  * @param    event: chose wake event.
    * @param    val: config val in different event.
  * @retval     HAL_OK          = 0x00,
                HAL_ERROR       = 0x01,
                HAL_BUSY        = 0x02,
                HAL_TIMEOUT     = 0x03
  */
```

说明：配置 sleep 唤醒模式
参数： event : 用于选择 wake event
val : 用于不同 wake event 下的配置参数
返回： 0 ： success
1 ： error
2 ： busy
3 ： timeout

2.2.9.5 hal_status_e hal_deepsleep_enter(void)

```
/**
  * @brief      Enter deepsleep mode
    * @detail     This function is used to enter deepsleep mode after wake
                  config.
  * @param    null
  * @retval     HAL_OK          = 0x00,
                HAL_ERROR       = 0x01,
                HAL_BUSY        = 0x02,
                HAL_TIMEOUT     = 0x03
  */
```

说明：当配置完必要参数后，调用此函数，可以使设备进入 deepsleep mode
参数：*sleep 用于存放配置参数

返回：0 ： success
1 ： error
2 ： busy
3 ： timeout

2.2.9.6 hal_status_e hal_wake_handle(void)

```
/**
  * @brief      System resume
      * @detail    This func is called to resume system when it wake up from sleep mode.
  * @param    null
  * @retval    HAL_OK          = 0x00,
               HAL_ERROR       = 0x01,
               HAL_BUSY        = 0x02,
               HAL_TIMEOUT     = 0x03
  */
```

说明：当设备从 sleep 模式唤醒时，此函数用来恢复系统运行。
参数：null
返回：0 ： success
1 ： error
2 ： busy
3 ： timeout

2.3 WLAN

- int s907x_wlan_on(int mode);
- int s907x_wlan_off();
- int is907x_wlan_is_running(int s907x_device_id) ;
- int s907x_wlan_get_mode(void);
- int s907x_wlan_set_netfunc(void *hook);
- int s907x_wlan_send(int id, struct eth_drv_sg *sg_list, int sg_len, int total_len);
- void s907x_wlan_recv(int id, struct eth_drv_sg *sg_list, int sg_len);
- void s907x_wlan_set_netif(int id, unsigned int netif);
- int s907x_wlan_event_reg(s907x_event_e event, s907x_event_callback event_callback, void

*context);

- int s907x_wlan_event_unreg(s907x_event_e event,s907x_event_callback event_callback);
- int s907x_wlan_start_ap(s907x_ap_init_t *init);
- int s907x_wlan_stop_ap(void);
- int s907x_wlan_ap_deauth_sta(u8 *hw_addr);
- int s907x_wlan_ap_get_client_nums(void);
- int s907x_wlan_ap_get_client_infor(ap_client_infor_t *infor, int max_client);
- int s907x_wlan_add_ie(u8 s907x_device_id, u8 *ie, int len);
- int s907x_wlan_del_id(u8 s907x_device_id);
- int s907x_wlan_start_sta(s907x_sta_init_t *init);
- int s907x_wlan_stop_sta(void);
- int s907x_wlan_scan(s907x_scan scan_cb, int max_ap_nums, void *context);
- int s907x_wlan_scan_ssid(const char *ssid,int ssid_len,s907x_scan_result_t *result);
- int s907x_wlan_get_link_infor(s907x_link_info_t *link_infor);
- int s907x_wlan_tx_mgt_frame(u8 s907x_device_id, u8 *pbuf, int len);
- s907x_wlan_enable_autoreconn(u8 cnt, u16 interval, int static_ip);
- int s907x_wlan_disable_autoreconn(void);
- int s907x_wlan_start_monitor(s907x_monitor_t *pmonitor);
- int s907x_wlan_stop_monitor(void);
- int s907x_wlan_set_channel(u8 s907x_device_id,u8 ch);
- u8 s907x_wlan_get_channel(u8 s907x_device_id);
- int s907x_wlan_set_mac_address(u8 s907x_device_id, char *mac);
- int s907x_wlan_get_mac_address(u8 s907x_device_id, char *mac);
- int s907x_wlan_set_phy_mode(u8 mode);
- int s907x_wlan_get_phy_mode(void);
- int s907x_wlan_set_country(int country_code);
- int s907x_wlan_get_country (void);

- void s907x_set_rx_mgnt_callback (wifi_rx_mgnt_callback cb);
- void s907x_wlan_enable_sleep(u8 sleep);

2.3.1 int s907x_wlan_on(int mode)

说明：启动 wifi，并设置工作模式
参数：mode 取值

```
typedef enum {
    S907X_MODE_NONE = 0,
    S907X_MODE_STA,
    S907X_MODE_AP,
    S907X_MODE_STA_AP,
    S907X_MODE_MONITOR,
    S907X_MODE_P2P
}s907x_mode_e;
```

返回：0 : No error
-1: Error

2.3.2 int s907x_wlan_off()

说明：关闭 wifi
参数：无
返回：0 : No error
-1: Error

2.3.3 int is907x_wlan_is_running(int s907x_device_id)

说明： 判断 wlanX （s907x_device_id）是否在运行
参数：s907x_device_id : 0/1
返回：0 : wlanX 没有运行
1: wlanX 正在运行

2.3.4 int s907x_wlan_get_mode(void)

说明：获取当前 wifi 的工作模式
参数：无
返回：工作模式

2.3.5 int s907x_wlan_set_netfunc(void *hook)

说明：设置网络回调函数
参数：无
返回：0

2.3.6 int s907x_wlan_send(int id, struct eth_drv_sg *sg_list, int sg_len, int total_len)

说明：发送 IP 数据包函数
参数：id:发送接口 id（0/1）
sg_list:发送数据缓存列表

```
struct eth_drv_sg {
    unsigned int buf;//缓存数据地址
    unsigned int     len; //缓存数据长度
};
```

sc_len:每个缓存数据的长度
totlen:总的数据长度
返回：0 : No error
-1: Error

2.3.7 void s907x_wlan_recv(int id, struct eth_drv_sg *sg_list, int sg_len)

说明：接收 IP 数据包函数
参数：id:发送接口 id（0/1）
sg_list:接收数据缓存列表
sc_len:每个缓存数据的长度
返回：无

2.3.8 void s907x_wlan_set_netif(int id, unsigned int netif)

说明：设置网络接口参数
参数：id:发送接口 id（0/1）
netif:网络参数地址
返回：无

2.3.9 int s907x_wlan_event_reg(s907x_event_e event,s907x_event_callback event_callback, void *context)

说明：注册事件回调函数
参数：event:事件

```
typedef enum
{
        S907X_EVENT_STAMODE_SCAN_DONE = 0,
        S907X_EVENT_STAMODE_CONNECTED,
        S907X_EVENT_STAMODE_DISCONNECTED,
        S907X_EVENT_STAMODE_AUTHCHANGE,
        S907X_EVENT_APMODE_STA_CONNECTED,
        S907X_EVENT_APMODE_STA_DISCONNECTED,
        S907X_EVENT_APMODE_PROBE_REG_RECEIVED,
    S907X_EVENT_MAX
}s907x_event_e;
```

event_callback:回调函数指针

```
typedef void (*s907x_event_callback)(s907x_event_data *event_data, void *context);
```

context:参数指针
返回：0

2.3.10 int s907x_wlan_event_unreg(s907x_event_e event,s907x_event_callback event_callback)

说明：注销事件回调函数
参数：event:事件
event_callback: 调用 s907x_wlan_event_reg 注册的回调函数指针

返回：0

2.3.11 int s907x_wlan_start_ap(s907x_ap_init_t *init)

说明：启动 AP
参数： init ： AP 的配置参数

```
typedef struct s907x_ap_init_
{
char *ssid;
    int     ssid_len;
    char *password;
    int     password_len;
    int     channel;
    int     is_hidded_ssid;
    s907x_ap_security security;
    }s907x_ap_init_t;
```

返回： 0 : No error
-1: Error

2.3.12 int s907x_wlan_stop_ap(void)

说明：断开所有 sta 的连接，并关闭 ap
参数：无
返回：0 : No error
-1: Error

2.3.13 int s907x_wlan_ap_deauth_sta(u8 *hw_addr)

说明：断开指定 mac 地址的 sta 连接
参数：hw_addr:sta 的 mac 地址
返回：0 : No error
-1: Error

2.3.14 int s907x_wlan_ap_get_client_nums(void)

说明：ap 模式下，获取连接的 sta 数量
参数：无
返回：sta 数量

2.3.15 int s907x_wlan_ap_get_client_infor(ap_client_infor_t *infor, int max_client)

说明：ap 模式下，获取连接的 sta 信息
参数：infor：指向存储 sta 信息的 buffer
Max_client：最多 sta 数

```
typedef struct s907x_ap_client_info_
{
    u8   hw_addr[6];
    u8   rsvd[2];
    s32 rssi;
}s907x_ap_client_info_t;
```

返回：0 : No error
-1: Error

2.3.16 int s907x_wlan_add_ie(u8 s907x_device_id, u8 *ie, int len)

说明：指定 wlan 接口添加定制的 IE
参数：s907x_device_id ： wlan 接口 id
Ie： 指向添加的 IE buffer

```
struct _cus_ie{
    u8 *ie;        // IE + Len + data
    u8 type; // beacon、probe_rsp、probe_req
};
```

Len : IE 数据长度
返回：0 : No error
-1: Error

2.3.17 int s907x_wlan_del_id(u8 s907x_device_id)

说明：删除指定 wlan 接口的全部定制的 IE
参数：s907x_device_id ： wlan 接口 id
返回：0 : No error
-1: Error

2.3.18 int s907x_wlan_start_sta(s907x_sta_init_t *init)

说明：启动 sta 连接 ap
参数：init：

```
typedef struct s907x_auto_conn_
{
    int     enable;
    int     interval_ms;
    int     cnt;
}s907x_auto_conn_t;

typedef struct s907x_conn_type_
{
    u8      mode;          //block or unblock
    u32*    result;       //should be value
    u32     blocking_timeout;
    sema_t coutom_async_sema;
}s907x_conn_type_t;

typedef struct s907x_sta_init_
{
    char *ssid;
    int     ssid_len;
    char *password;
    int     password_len;
    int     channel;
    u8      security;
    s907x_conn_type_t     conn;
    s907x_auto_conn_t     auto_conn;
}s907x_sta_init_t;
```

返回：0 : No error

-1: Error

2.3.19 int s907x_wlan_stop_sta(void)

说明：sta 断开连接。
参数：无
返回：0 : No error
-1: Error

2.3.20 int s907x_wlan_scan(s907x_scan scan_cb, int max_ap_nums, void *context)

说明：sta 开启扫描
参数：scan_cb :

```
typedef void (*s907x_scan) (s907x_scan_result_t *presult);
```

max_ap_nums:支持扫描最大的 ap 数
context:可选参数指针，指向 s907x_scan_result_t 中的 context，可在 scan_cb 中访问。

```
typedef struct s907x_scan_info_
{
    char ssid[33];
    int  ssid_len;
    u8   bssid[6];
    u8   channel;
    u8   security;
    s32  rssi;
    void *object;
    u16  rsvd[2];
}s907x_scan_info_t;

typedef struct s907x_scan_result_
{
    int max_nums;
    int id;
    void *context;
    s907x_scan_info_t scan_info;
```

```
    }s907x_scan_result_t;
```

返回：0 : No error

-1: Error

2.3.21 int s907x_wlan_scan_ssid(const char *ssid,int ssid_len,s907x_scan_result_t *result)

说明：sta 扫描指定 ssid 的 ap

参数：ssid:路由的 ssid

ssid_len:ssid 的长度

result:扫描结果指针

```
typedef struct s907x_scan_result_{
    int max_nums;//扫描到的 ap 数量
    int id;//扫描到的 ap 的 id
    void *context;//传入的参数指针
    s907x_scan_info_t scan_info;
}s907x_scan_result_t;
```

返回：0 : No error

-1: Error

2.3.22 int s907x_wlan_get_link_infor(s907x_link_info_t *link_infor)

说明：获取连接状态

参数：link_infor：指向存储连接状态的 buffer

```
typedef struct s907x_link_info_
{
    int is_connected;
    int rssi;
    int channel;
    u8   ssid[33];
    u8   bssid[6];
}s907x_link_info_t;
```

返回：0 : No error

-1: Error

2.3.23 int s907x_wlan_tx_mgt_frame(u8 s907x_device_id, u8 *pbuf, int len)

说明：指定 wlan 接口发送管理帧
参数：s907x_device_id：wlan 接口 id
pbuf：指向要发送的管理帧 buffer
len :要发生的管理帧数据长度
返回：0 : No error
-1: Error

2.3.24 int s907x_wlan_enable_autoreconn(u8 cnt, u16 interval, int static_ip)

说明：使能 sta 自动重连
参数：cnt:重连次数
interval:重连的时间间隔（秒）
static_ip:静态 ip（0 代表启动 dhcp 获取 ip）
返回：0

2.3.25 int s907x_wlan_disable_autoreconn(void)

说明：关闭 sta 自动重连
参数：无
返回：0

2.3.26 int s907x_wlan_start_monitor(s907x_monitor_t *pmonitor)

说明：启动 wifi monitor
这个函数会关闭 station 和 ap
参数：pmonitor：指向 monitor 相关参数的 buffer，包括回调函数，以及使能，packet 过滤类型

```
typedef enum
{
    m_disable,
    m_bcmc,        //b/m packets only
    m_all,      //802.11 all packkets
}monitor_mode_e;
```

```
typedef void (*s907x_monitor_cb) (u8 *pbuf, int len, void *context);

typedef struct s907x_monitor_
{
    monitor_mode_e mode;
    s907x_monitor_cb cb;
    void *coustom_context;
}s907x_monitor_t;
```

返回：0 : No error
-1: Error

2.3.27 int s907x_wlan_stop_monitor(void)

说明：关闭 monitor 模式
参数：无
返回：0 : No error
-1: Error

2.3.28 int s907x_wlan_set_channel(u8 s907x_device_id,u8 ch)

说明：设置 wlanx(s907x_device_id)信道
参数：s907x_device_id: wlan 接口 id（0/1）
ch:信道
返回：0 : No error
-1: Error

2.3.29 u8 s907x_wlan_get_channel(u8 s907x_device_id)

说明：获取 wlan 端口(s907x_device_id)的信道
参数：s907x_device_id: wlan 接口 id（0/1）
返回：当前信道

2.3.30 int s907x_wlan_set_mac_address(u8 s907x_device_id, char *mac)

说明：设置指定 wlan 接口 mac 地址
参数：s907x_device_id：wlan 接口 id
mac:指向存储 mac 地址的 buffer
返回：0 : No error
-1: Error

2.3.31 int s907x_wlan_get_mac_address(u8 s907x_device_id, char *mac)

说明：获取指定 wlan 接口 mac 地址
参数：mac:指向存储 mac 地址的 buffer
返回：0 : No error
-1: Error

2.3.32 int s907x_wlan_set_phy_mode(u8 mode)

说明：设置 PHY 模式
参数：mode:phy 模式（b/bg/bgn）

```
typedef enum {
    S907X_PHYMODE_B = 0,
    S907X_PHYMODE_BG,
    S907X_PHYMODE_BGN
}s907x_phymode_e;
```

返回：0 : No error
-1: Error

2.3.33 int s907x_wlan_get_phy_mode(void)

说明：获取 PHY 模式
参数：无
返回：phy 模式（b/bg/bgn）

2.3.34 int s907x_wlan_set_country(int country_code)

说明：设置国家码
参数：country_code:国家码信息

```
typedef    enum {
    S907X_COUNTRY_CN,//1~13
    S907X_COUNTRY_US,//1~13
    S907X_COUNTRY_JP,//1~14
    S907X_COUNTRY_FR,//1~13
    S907X_COUNTRY_AU,//1~13
    S907X_COUNTRY_EU,//1~13
    S907X_COUNTRY_MAX//1~13
} s907x_country_code_e;
```

返回：0 : No error
-1: Error

2.3.35 int s907x_wlan_get_country (void)

说明：读取国家码
参数：无
返回：国家码信息

2.3.36 void s907x_set_rx_mgnt_callback (wifi_rx_mgnt_callback cb)

说明：设置接收管理帧回调函数
参数：cb：回调函数指针

```
typedef void (*wifi_rx_mgnt_callback)(u8 *buf,int buf_len,int type,char rssi);
```

返回：无

2.3.37 void s907x_wlan_enable_sleep(u8 sleep)

说明：设置 rtos powersleep 状态
参数：sleep：0 – 禁止 ps
1 – 使能 ps
返回：无

2.3.38 API 使用实例

2.3.38.1 Station 模式

设置模块工作模式 1（sta:1 ; ap:2 ; sta_ap:3 ; promisc:4 ; p2p:5）
初始化扫描相关的变量
初始化结构体
调用 sta 启动函数
启动 dhcp client
注：非阻塞模式/阻塞模式下

2.3.38.2 AP 模式

设置模块工作模式 2（sta:1 ; ap:2 ; sta_ap:3 ; promisc:4 ; p2p:5）

初始化结构体 network_InitTypeDef_st init；

调用 ap 启动函数

启动 dhcp server

注：返回 0，则代表 ap 成功启动，之后可以用电脑或手机等 sta 扫描到该 ap（ssid:test），输入密码：12345678 即可连接成功，可以看到 sta 分配到的 ip 地址。Ap 自身的 ip 状态可。

2.3.38.3 Add custom ie

```
uint8 custom_ie[]={0x07,0x01,0x33,};
struct _cus_ie{
  u8 *ie;
  u8 type;
};
struct _cus_ie cus_ie;
cus_ie.type = BIT(2);//beacon 或 probe_rsp
cus_ie.ie = custom_ie;
mico_wlan_custom_ie_add(Soft_AP,(u8*)&test_cus_ie,3);
```

custom ie 组成：IE + Len + data

cus_ie.type:添加到哪种类型包里进行发送

本例中，调用之后,通过抓包软件，可以看到，在 beacon 中多了一个 country ie

2.3.38.4 Delete custom ie

本例中，调用之后，通过抓包软件可以看到，2.4.3 例中 beacon 中添加的 country ie 已经不存在了

3 版本信息

日期	版本	更新内容	作者
2019-3-20	1.0	文档草案	tqian
2019-4-20	1.1	部分接口修订	tqian