Detailed explanation of WCH TMOS usage

Original post: https://www.cnblogs.com/debugdabiaoge/p/15775521.html

In order to connect with multiple devices and achieve multi-function and multi-tasking, Bluetooth has a scheduling problem. Although the software and protocol stack can be expanded, after all, there is only one bottom-level execution department. In order to realize multi-event and multi-task switching, it is necessary to correspond events and tasks, and a TMOS name operating system abstraction layer is set up for this application.

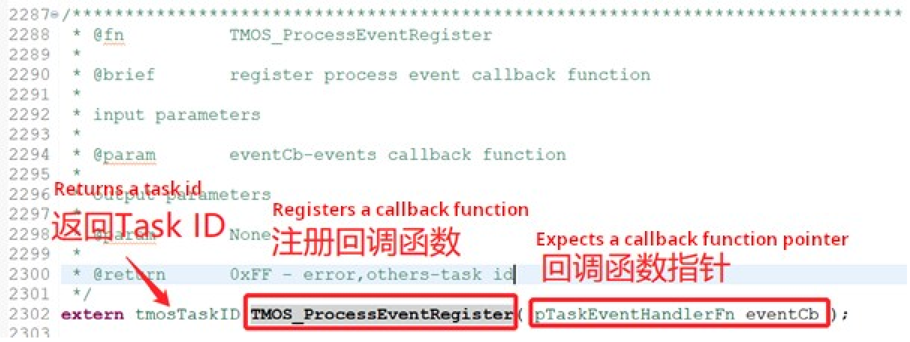
TMOS is the scheduling core, and the BLE protocol stack, profile definition, and all applications are implemented around it. TMOS is not the traditional operating system that everyone uses, but a cycle that allows software to create and execute events.

In the multi-task management mode, only one task is actually running, but multiple tasks can be scheduled using the task scheduling strategy, and each task takes a certain amount of time (exclusive, exit after executing the current task, and continue to query other executable tasks) , all tasks are processed by time slicing.

The **TMOS** system clock unit is **625us**, and all the required system time is obtained based on the RTC.

Software functions are realized by task events. Creating a task event requires the following work:

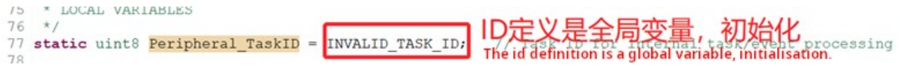
1. Create task identifier task ID



```
/*********************************************************************************
 * @fn          TMOS_ProcessEventRegister
 *
 * @brief       register process event callback function
 *
 * input parameters
 *
 * @param       eventCb-events callback function
 *
 * output parameters
 *
 * @param       None
 *
 * @return      0xFF - error,others-task id
 */
extern tmosTaskID TMOS_ProcessEventRegister( pTaskEventHandlerFn eventCb );
```

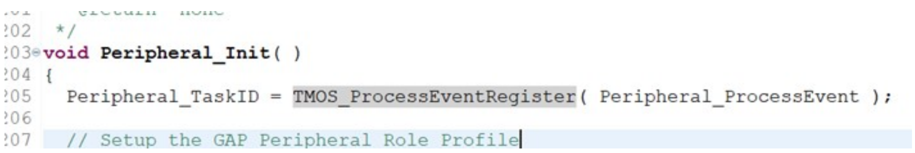
Returns a task id
返回Task ID
Registers a callback function
注册回调函数
Expects a callback function pointer
回调函数指针

For example:



```
 * LOCAL VARIABLES
 */
static uint8 Peripheral_TaskID = INVALID_TASK_ID;   // Task ID for internal task/event processing
```

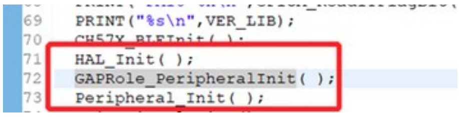
ID定义是全局变量，初始化
The id definition is a global variable, initialisation.



```
 */
void Peripheral_Init( )
{
  Peripheral_TaskID = TMOS_ProcessEventRegister( Peripheral_ProcessEvent );

  // Setup the GAP Peripheral Role Profile
```

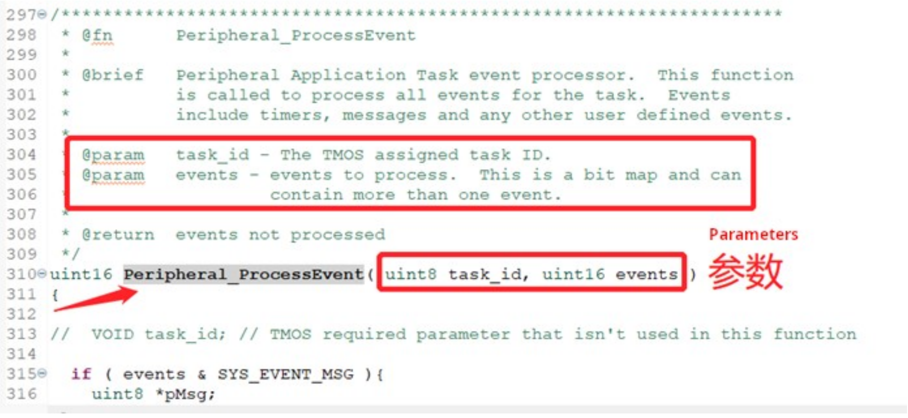
2. Write a task initialization routine process and add it to the TMOS initialization process

Which means that the function cannot be dynamically added after the system starts (new Task ID).



```
PRINT("%s\n",VER_LIB);
CH57X_BLEInit( );
HAL_Init( );
GAPRole_PeripheralInit( );
Peripheral_Init( );
```

3. Write a task handler



```
/*********************************************************************
 * @fn      Peripheral_ProcessEvent
 *
 * @brief   Peripheral Application Task event processor.  This function
 *          is called to process all events for the task.  Events
 *          include timers, messages and any other user defined events.
 *
 * @param   task_id - The TMOS assigned task ID.
 * @param   events - events to process.  This is a bit map and can
 *                   contain more than one event.
 *
 * @return  events not processed
 */
uint16 Peripheral_ProcessEvent( uint8 task_id, uint16 events )
{

//  VOID task_id; // TMOS required parameter that isn't used in this function

  if ( events & SYS_EVENT_MSG ){
    uint8 *pMsg;
```

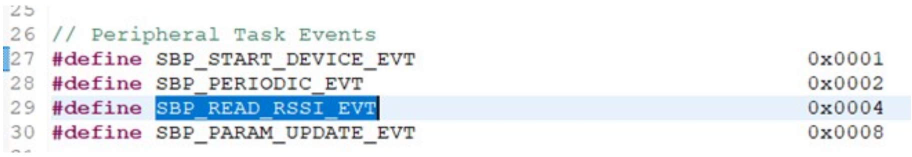
Parameters

参数

4. Define task events and write user function codes

The event name is defined by bit, each **taskID** contains at most 1 message event and 15 task events (total 16 bits). For example:

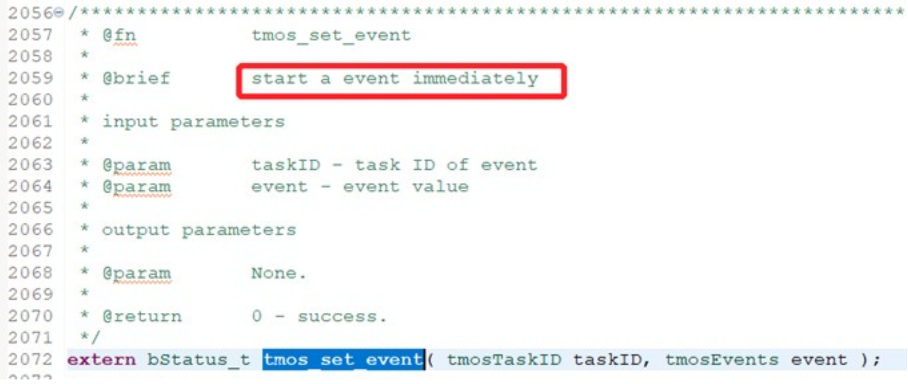
Define EVT task events bit by bit, as shown in the figure below:



```
// Peripheral Task Events
#define SBP_START_DEVICE_EVT                                  0x0001
#define SBP_PERIODIC_EVT                                      0x0002
#define SBP_READ_RSSI_EVT                                     0x0004
#define SBP_PARAM_UPDATE_EVT                                  0x0008
```

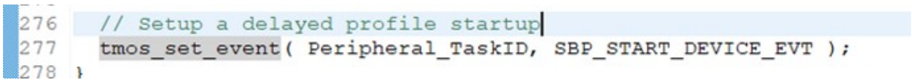
There are two ways to start the task event (the task is only executed once after the task is started, if it is executed repeatedly, the task needs to be restarted):

1) The task is started immediately, and the event time is executed immediately after the call



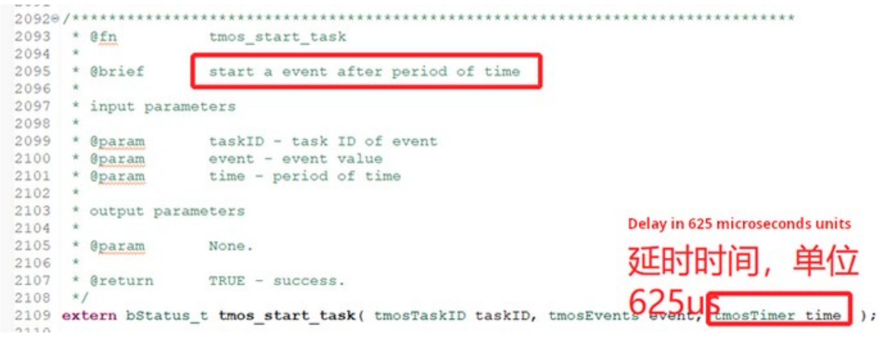
```
/*********************************************************************
 * @fn          tmos_set_event
 *
 * @brief       start a event immediately
 *
 * input parameters
 *
 * @param       taskID - task ID of event
 * @param       event - event value
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      0 - success.
 */
extern bStatus_t tmos_set_event( tmosTaskID taskID, tmosEvents event );
```

For example:



```
  // Setup a delayed profile startup
  tmos_set_event( Peripheral_TaskID, SBP_START_DEVICE_EVT );
}
```

2) Set a delay to start a task, and start timing after the setting is completed



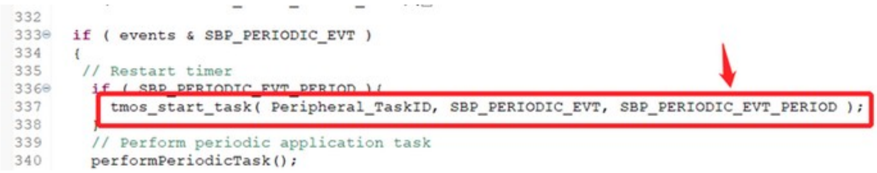
```
/*********************************************************************************
 * @fn          tmos_start_task
 *
 * @brief       start a event after period of time
 *
 * input parameters
 *
 * @param       taskID - task ID of event
 * @param       event - event value
 * @param       time - period of time
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      TRUE - success.
 */
extern bStatus_t tmos_start_task( tmosTaskID taskID, tmosEvents event, tmosTimer time );
```

Delay in 625 microseconds units

延时时间，单位

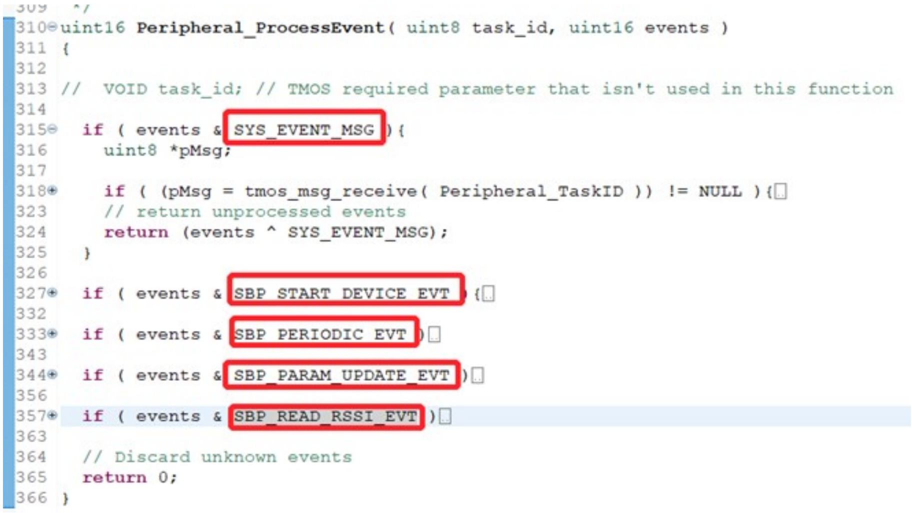
625us

For example: the custom **SBP_PERIODIC_EVT** task under the **Peripheral_TaskID** function runs after a delay of (**SBP_READ_RSSI_EVT_PERIOD***625)us.



```
  if ( events & SBP_PERIODIC_EVT )
  {
   // Restart timer
    if ( SBP_PERIODIC_EVT_PERIOD ){
      tmos_start_task( Peripheral_TaskID, SBP_PERIODIC_EVT, SBP_PERIODIC_EVT_PERIOD );
    }
    // Perform periodic application task
    performPeriodicTask();
```

User code function. When generating **TaskID**, you need to register the EVT processing function pointer with TMOS. After the EVT execution conditions are met, TMOS will automatically call this function, as shown in the following figure:



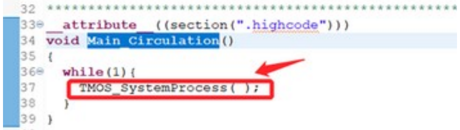
```
uint16 Peripheral_ProcessEvent( uint8 task_id, uint16 events )
{

//  VOID task_id; // TMOS required parameter that isn't used in this function

  if ( events & SYS_EVENT_MSG ){
    uint8 *pMsg;

    if ( (pMsg = tmos_msg_receive( Peripheral_TaskID )) != NULL ){□
    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }

  if ( events & SBP_START_DEVICE_EVT ){□

  if ( events & SBP_PERIODIC_EVT )□

  if ( events & SBP_PARAM_UPDATE_EVT )□

  if ( events & SBP_READ_RSSI_EVT )□

  // Discard unknown events
  return 0;
}
```

extern bStatus_t tmos_stop_task(tmosTaskIDtaskID, tmosEvents event);

This function will stop a task named event that will take effect at the **taskID** layer. After calling this function, the **event** task will not take effect.

5. The main loop calls TMOS_SystemProcess continuously to query executable events

If **HAL_SLEEP** starts, the chip turns on low-power sleep mode, TMOS will turn on the RTC wake-up function, and it will automatically wake up before the event is executed, and run the event code.



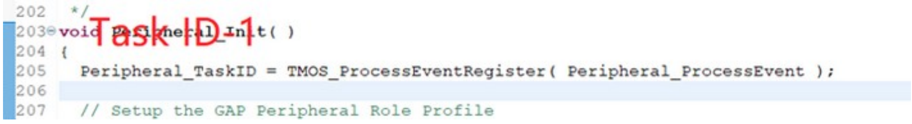
```
***************************************************
__attribute__((section(".highcode")))
void Main_Circulation()
{
  while(1){
    TMOS_SystemProcess( );
  }
}
```

Precautions for using the task scheduling function:

1. It is forbidden to call in interrupt
2. It is recommended not to execute tasks that exceed half the connection interval in a single task, otherwise it will affect Bluetooth communication
3. In the same way, it is recommended not to perform tasks that exceed half the connection interval during the interruption, otherwise it will affect the Bluetooth communication
4. When the delayed execution function is called in the code executed by the event, the delay time is offset from the current event effective time point, so there is no requirement for the position of the delayed execution function called in the executed code.
5. Tasks have priority, which is determined according to the sequence of judgments in the **xxx_ProcessEvent** function. Tasks that are effective at the same time are executed first and judged first, and then judged after execution. Note that after executing the first-judgment event task, the last-judgment event task will not be executed until the task scheduling system takes turns.
6. The event name is defined by bit. Each **taskID** contains at most 1 message event and 15 task events (total 16 bits)

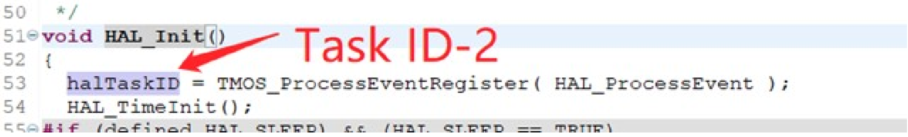
I talked about the application of a Task ID earlier. In order to reduce the coupling between C files or functions, it is generally better to put the same function or similar events under the same Task ID, which creates a problem; different Task IDs may have data that needs to be interacted with, and TMOS provides functions for data interaction between different Task IDs.

For example: Take two of the Task IDs in the peripheral as examples, **halTaskID** and **Peripheral_TaskID**, assuming that data interaction between these two Tasks is required.



```
 */
void Peripheral_Init( )
{
  Peripheral_TaskID = TMOS_ProcessEventRegister( Peripheral_ProcessEvent );

  // Setup the GAP Peripheral Role Profile
```

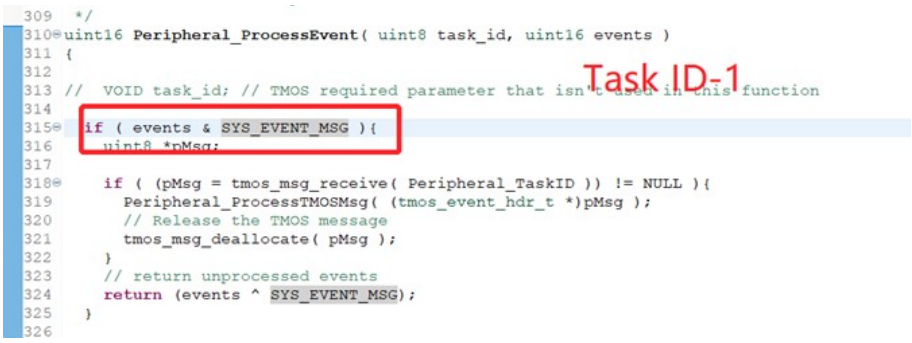
Task ID-1



```
 */
void HAL_Init()
{
  halTaskID = TMOS_ProcessEventRegister( HAL_ProcessEvent );
  HAL_TimeInit();
#if (defined HAL_SLEEP) && (HAL_SLEEP == TRUE)
```

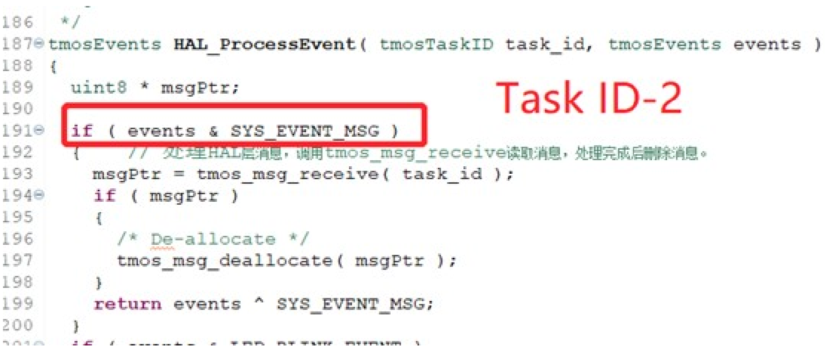
Task ID-2

It was said before that each Task ID has 1 message **event**



```
 */
uint16 Peripheral_ProcessEvent( uint8 task_id, uint16 events )
{

//  VOID task_id; // TMOS required parameter that isn't used in this function

  if ( events & SYS_EVENT_MSG ){
    uint8 *pMsg;

    if ( (pMsg = tmos_msg_receive( Peripheral_TaskID )) != NULL ){
      Peripheral_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg );
      // Release the TMOS message
      tmos_msg_deallocate( pMsg );
    }
    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }

```

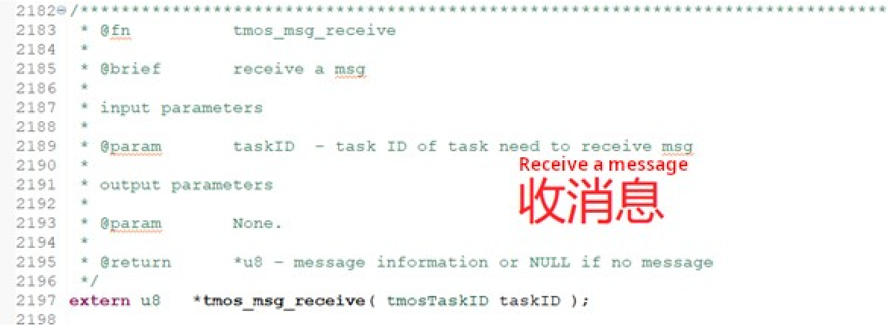
Task ID-1



```
 */
tmosEvents HAL_ProcessEvent( tmosTaskID task_id, tmosEvents events )
{
  uint8 * msgPtr;

  if ( events & SYS_EVENT_MSG )
  {     // 处理HAL层消息，调用tmos_msg_receive读取消息，处理完成后删除消息。
    msgPtr = tmos_msg_receive( task_id );
    if ( msgPtr )
    {
      /* De-allocate */
      tmos_msg_deallocate( msgPtr );
    }
    return events ^ SYS_EVENT_MSG;
  }
  if ( events & LED_BLINK_EVENT )
```

Task ID-2

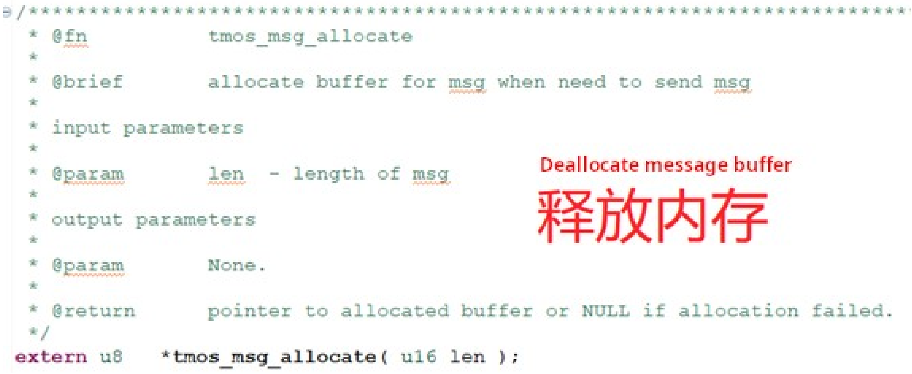
The above figure demonstrates receiving messages, which mainly uses 2 functions:



```
/*********************************************************************
 * @fn          tmos_msg_receive
 *
 * @brief       receive a msg
 *
 * input parameters
 *
 * @param       taskID  - task ID of task need to receive msg
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      *u8 - message information or NULL if no message
 */
extern u8   *tmos_msg_receive( tmosTaskID taskID );

```

Receive a message

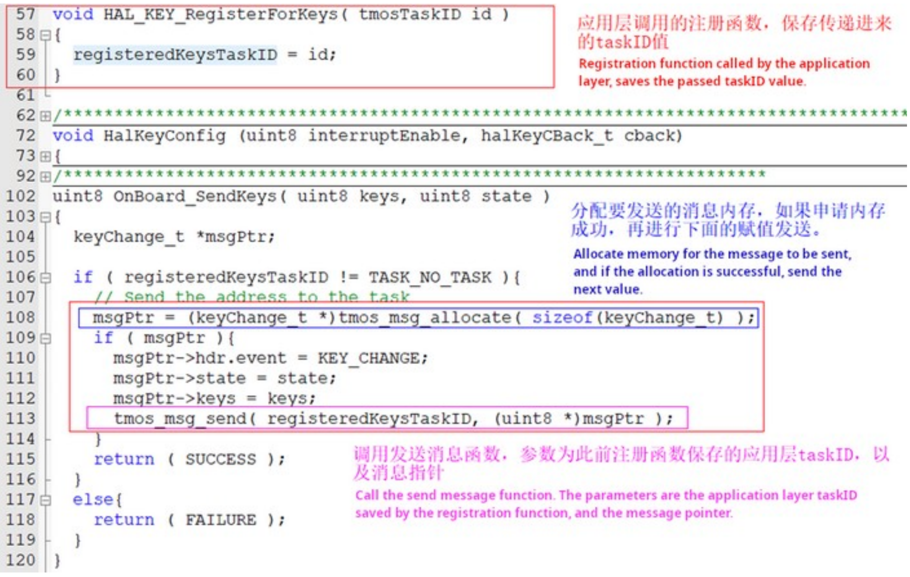
收消息



```
/*********************************************************************
 * @fn          tmos_msg_allocate
 *
 * @brief       allocate buffer for msg when need to send msg
 *
 * input parameters
 *
 * @param       len   - length of msg
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      pointer to allocated buffer or NULL if allocation failed.
 */
extern u8   *tmos_msg_allocate( u16 len );
```

Deallocate message buffer

释放内存



```
void HAL_KEY_RegisterForKeys( tmosTaskID id )
{
  registeredKeysTaskID = id;
}

/******************************************************************************
void HalKeyConfig (uint8 interruptEnable, halKeyCBack_t cback)
{
/*******************************************************************
uint8 OnBoard_SendKeys( uint8 keys, uint8 state )
{
  keyChange_t *msgPtr;

  if ( registeredKeysTaskID != TASK_NO_TASK ){
    // Send the address to the task
    msgPtr = (keyChange_t *)tmos_msg_allocate( sizeof(keyChange_t) );
    if ( msgPtr ){
      msgPtr->hdr.event = KEY_CHANGE;
      msgPtr->state = state;
      msgPtr->keys = keys;
      tmos_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
    }
    return ( SUCCESS );
  }
  else{
    return ( FAILURE );
  }
}
```

应用层调用的注册函数，保存传递进来的taskID值

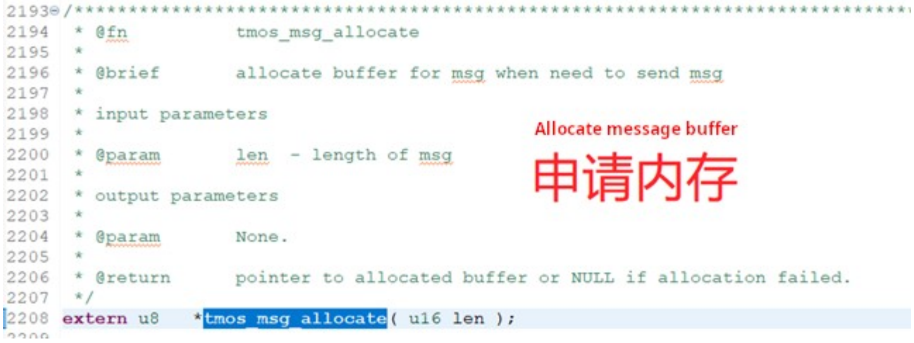
Registration function called by the application layer, saves the passed taskID value.

分配要发送的消息内存，如果申请内存成功，再进行下面的赋值发送。

Allocate memory for the message to be sent, and if the allocation is successful, send the next value.

调用发送消息函数，参数为此前注册函数保存的应用层taskID，以及消息指针

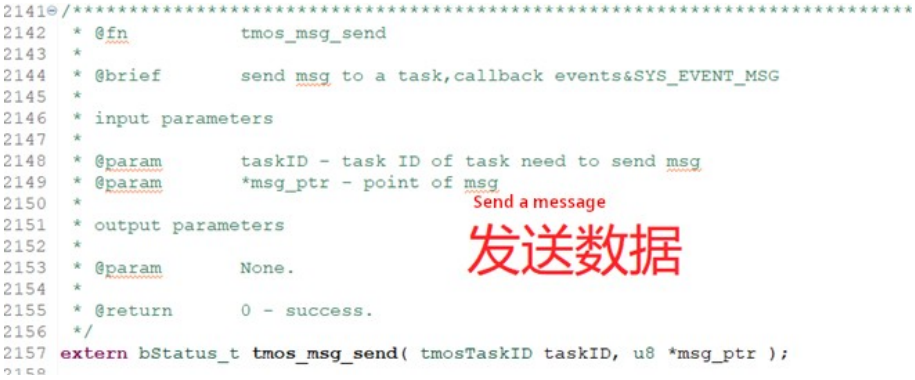
Call the send message function. The parameters are the application layer taskID saved by the registration function, and the message pointer.



```
/*******************************************************************************
 * @fn          tmos_msg_allocate
 *
 * @brief       allocate buffer for msg when need to send msg
 *
 * input parameters
 *
 * @param       len  - length of msg
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      pointer to allocated buffer or NULL if allocation failed.
 */
extern u8   *tmos_msg_allocate( u16 len );
```

Allocate message buffer

申请内存



```
/*******************************************************************************
 * @fn          tmos_msg_send
 *
 * @brief       send msg to a task,callback events&SYS_EVENT_MSG
 *
 * input parameters
 *
 * @param       taskID - task ID of task need to send msg
 * @param       *msg_ptr - point of msg
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      0 - success.
 */
extern bStatus_t tmos_msg_send( tmosTaskID taskID, u8 *msg_ptr );
```

Send a message

发送数据