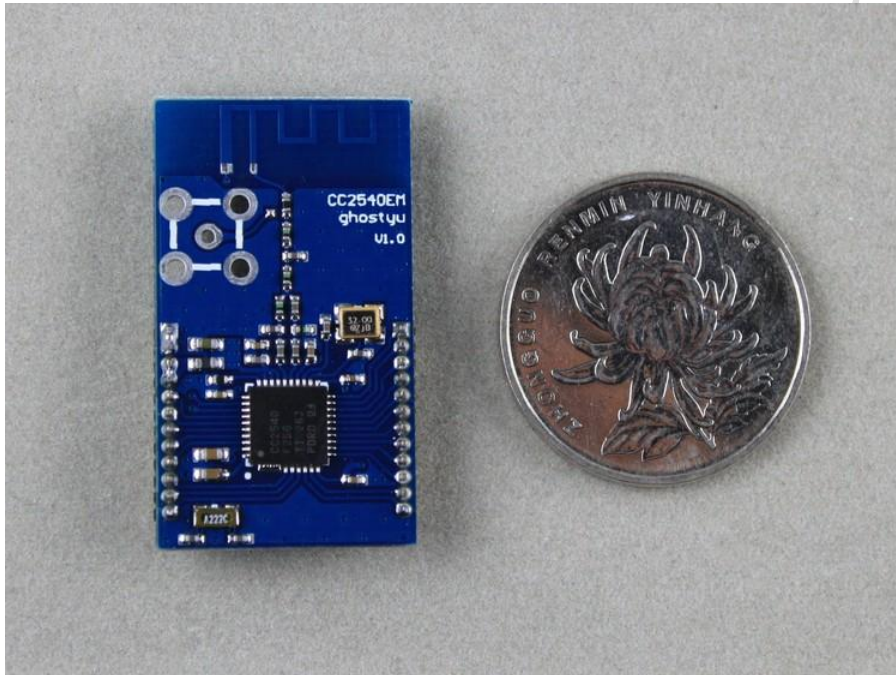




OSAL 编程指南



Ghostyu
2013-03-06



版本

V1.0	2013-03	初始版发布



目的

本文在 BLE 入门指南的基础上进一步阐述 OSAL 的机制和编程方法，希望对 BLE 初学者有所帮助。

OSAL 的具体实现方法不再本文讨论范围，请参见后续文档。

OSAL 层的 API 使用方法，请参见《OSAP API 用户手册》

一、几个小问题。

1、OSAL 是什么

操作系统抽象层，可以理解为运行在 CC2540 上的操作系统，并且把协议栈的代码、硬件处理的代码，用户程序的代码等分别放到了 OSAL 层的不同任务函数中去，各任务函数通过消息、事件的方式来通信。

2、什么是 EVENT 事件

OSAL 为每个任务函数分配了一个 16 位的事件变量，每一位代表一个时间，最高位 0x8000 保留为系统事件 SYS_EVENT_MSG。其余的 15 位留给用户自定义需要的事件。通常事件由定时器启动，比如两秒后我要点亮 LED1，这就需要发送一个点亮 LED1 的事件，然后等待，当 2 秒后接收到点亮 LED1 事件的时候调用 hal 层开关 LED1 的函数开启 LED1。

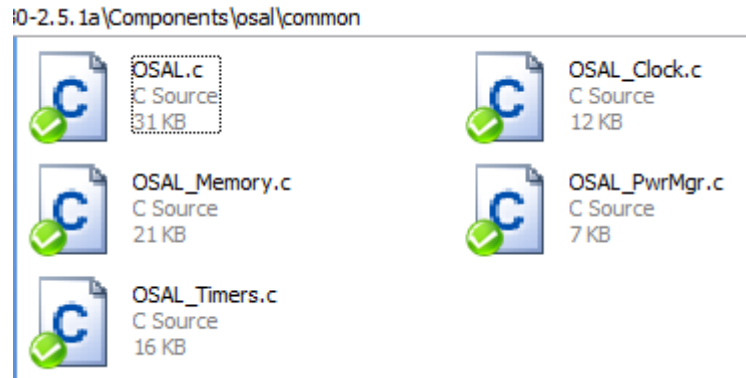
3、什么是 MSG 消息

MSG 是比 EVENT 事件更具体并且可以携带数据的一种通信方式。而且 MSG 的标记是按数值，而不是按位。比如 0x02 和 0x03 是两个不同的消息，但对于事件 0x03 则是 0x01 事件和 0x02 事件的组合。MSG 收发使用 `osal_msg_send()` 和 `osal_msg_receive()`；当调用 `osal_msg_send()` 发送一个 msg 的同事会在 EVENT 列表中触发一个 message ready event。



二、OSAL 的组成。

OSAL 层的源码如下图，这五个文件的分别的作用为



OSAL.c

内存数据复制、比较等函数和 msg 消息的发送接收等函数的实现。

OSAL_Clock.c

系统滴答时钟

OSAL_Memory.c

系统“动态内存”管理，这里是打了引号的动态内存管理，其实是分配的一块静态内存池，然后通过标记的方式来动态的管理该静态内存，这样即无需复杂的动态内存实现技术又有效的利用了单片机有限的内存空间。实现的功能相当于 PC 上的 malloc 函数。

OSAL_PwrMgr.c

功率管理

OSAL_Timers.c

系统定时器管理，任务函数的中事件定时器 osal_start_timerEx() 就来自该源文件。该源文件维护了一个定时器队列，每当用户调用 osal_start_timerEx()，就会再该队列上添加一个成员，当定时事件到时，通知上层，并删除相应的成员。

三、OSAL 使用范例分析

在这一节，我们分析来分析应用程序使用 OSAL 层的两种最常见的范例分析，将它的来龙去脉展现给用户。

以 SimpleBLEPeripheral 为例，一种是用用户事件的启动和接收，第二种是消息的发送和接收。

3.1 用户事件的启动和接收

在 Z-Stack 协议栈的用户程序中，为了实现特定的功能，除了使用系统事件外，用户自定义一些用户事件是非常有必要的。比如要周期性的去检测某个 GPIO 的状态，或者每隔一段时间发送一个命令等，都需要自定义事件来完成。

在 SimpleBLEPeripheral 中，常见的用户事件

```
// Simple BLE Peripheral Task Events
#define SBP_START_DEVICE_EVT           0x0001
#define SBP_PERIODIC_EVT               0x0002
#define SBP_ADV_IN_CONNECTION_EVT     0x0004
```

每个任务函数都类似这样的格式：



```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
{
    if ( events & SYS_EVENT_MSG )
    {
        uint8 *pMsg;
        // return unprocessed events
        return (events ^ SYS_EVENT_MSG);
    }

    if ( events & SBP_START_DEVICE_EVT )
    {
        return ( events ^ SBP_START_DEVICE_EVT );
    }

    if ( events & SBP_PERIODIC_EVT )
    {
        return (events ^ SBP_PERIODIC_EVT);
    }
    // Discard unknown events
    return 0;
} ? end SimpleBLEPeripheral_ProcessEvent ?
```

每一个 if 语句均对应一个 EVENT 事件。首先处理的是 SYS_EVENT_MSG 系统消息事件。其次是用户自定的事件，注意，每个任务函数每次只处理一个事件（注意每个 if 中的 return），也就是说本次处理完了 SYS_EVENT_MSG 事件会直接 return。然后准备下一次 OSAL 层任务调度到该任务时再处理用户自定义的 SBP_START_DEVICE_EVT，如果此时正好又发生了一个 SYS_EVENT_MSG 系统消息事件，那么将继续先处理 SYS_EVENT_MSG，直到空闲的时候才会去处理任务自定义的事件。

留心的读者应该发现了一个问题，这个 if 框架是处理事件的，也就是说事件已经发了才会来处理，那么事件是在哪发起的呢？

我们先寻找 SBP_START_DEVICE_EVT 事件的启动。

在 SimpleBLEPeripheral_Init 任务初始化函数结尾处启动了该事件。

```
void SimpleBLEPeripheral_Init( uint8 task_id )
{
    simpleBLEPeripheral_TaskID = task_id;
    ...
    // Setup a delayed profile startup
    osal_set_event( simpleBLEPeripheral_TaskID, SBP_START_DEVICE_EVT );
```

osal_set_event() 函数是事件启动函数，参数 1 是需要启动定时器的任务 ID，第二个参数是启动的事件 ID，调用该函数后会立刻出发启动的事件，接着在该任务函数中就会接收到 SBP_START_DEVICE_EVT 事件，进而进入下列代码中执行：

```
if ( events & SBP_START_DEVICE_EVT )
{
    // Start the Device
    VOID GAPRole_StartDevice( &simpleBLEPeripheral_PeripheralCBs );

    // Start Bond Manager
    VOID GAPBondMgr_Register( &simpleBLEPeripheral_BondMgrCBs );

    // Set timer for first periodic event
    osal_start_timerEx( simpleBLEPeripheral_TaskID, SBP_PERIODIC_EVT,

    return ( events ^ SBP_START_DEVICE_EVT );
}
```



对该事件作出何种相应，完全由用户决定。注意，定时器定时均为一次性的，不会自动装载。

至于 `osal_set_event` 和 `osal_start_timerEx` 函数具体做了什么，我们将在后续的文档开发资料用详细的阐述，本文档只为 OSAL 编程做指导。OSAL 实现的细节不在讨论范围内。

3.2 MSG 消息的发送与接收

在 `SimpleBLEPeripheral` 任务初始化函数中有这样一条代码：

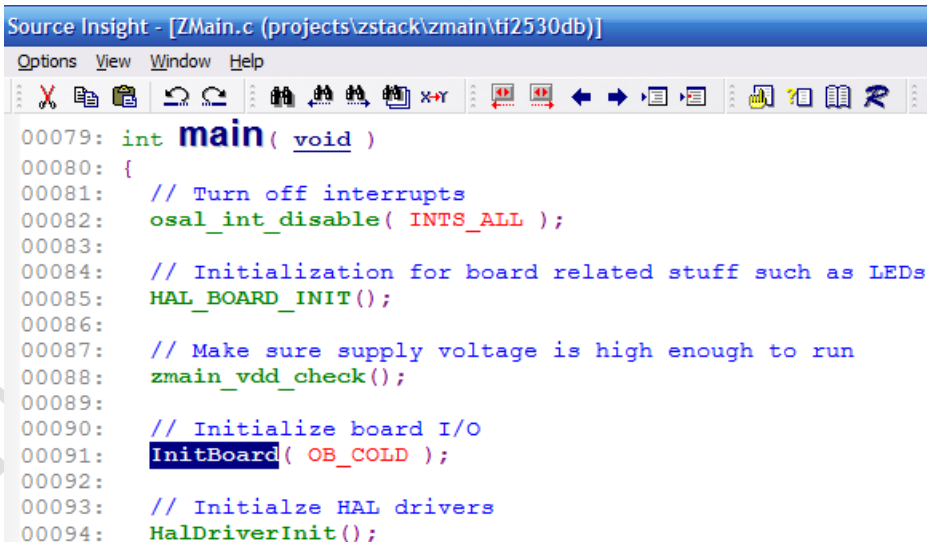
```
// Register for all key events - This app will handle all key events
RegisterForKeys( simpleBLEPeripheral_TaskID );
```

这个函数来自 `OnBoard.c` 源文件中

```
00188: uint8 RegisterForKeys( uint8 task_id )
00189: {
00190:     // Allow only the first task
00191:     if ( registeredKeysTaskID == NO_TASK_ID )
00192:     {
00193:         registeredKeysTaskID = task_id;
00194:         return ( true );
00195:     }
00196:     else
00197:         return ( false );
00198: }
```

向一个全局变量中赋值自己的任务 ID，从代码中可以看出，只能第一个调用该函数的任务才能成功注册到按键服务。那这个全局变量在何时使用呢？

我们暂且放下该问题，在 `main.c` 初始化过程有如下代码：



```
Source Insight - [ZMain.c (projects\zstack\zmain\ti2530db)]
Options View Window Help
00079: int main( void )
00080: {
00081:     // Turn off interrupts
00082:     osal_int_disable( INTS_ALL );
00083:
00084:     // Initialization for board related stuff such as LEDs
00085:     HAL_BOARD_INIT();
00086:
00087:     // Make sure supply voltage is high enough to run
00088:     zmain_vdd_check();
00089:
00090:     // Initialize board I/O
00091:     InitBoard( OB_COLD );
00092:
00093:     // Initialize HAL drivers
00094:     HalDriverInit();
```

该函数的实现如下：



```
00124: void InitBoard( uint8 level )
00125: {
00126:     if ( level == OB_COLD )
00127:     {
00128:         // IAR does not zero-out this byte below the XSTACK.
00129:         *(uint8 *)0x0 = 0;
00130:         // Interrupts off
00131:         osal_int_disable( INTS_ALL );
00132:         // Check for Brown-Out reset
00133:         ChkReset();
00134:     }
00135:     else // !OB_COLD
00136:     {
00137:         /* Initialize Key stuff */
00138:         HalKeyConfig(HAL_KEY_INTERRUPT_DISABLE, OnBoard_KeyCallback);
00139:     }
00140: }
```

该函数通过 HalKeyConfig 接口注册了一个按键回调函数 OnBoard_KeyCallback，首先看 HalKeyConfig 函数的实现：

将上述的回调函数的地址复制给了函数指针变量。通过跟踪发现该函数指针变量在按键的轮询函数中调用，如下图：

```
00318: void HalKeyPoll (void)
00319: {
00320:     uint8 keys = 0;
00321:
00322:     if ((HAL_KEY_JOY_MOVE_PORT & HAL_KEY_JOY_MOVE_BIT)) /*
00323:     {
00324:         keys = halGetJoyKeyInput();
00325:     }
00326:
00327:     /* If interrupts are not enabled, previous key status an
00328:     * are compared to find out if a key has changed status.
00329:     */
00330:     if (!Hal_KeyIntEnable)
00331:     {
00332:         if (keys == halKeySavedKeys)
00333:         {
00334:             /* Exit - since no keys have changed */
00335:             return;
00336:         }
00337:         /* Store the current keys for comparison next time */
00338:         halKeySavedKeys = keys;
00339:     }
00340:     else
00341:     {
00342:         /* Key interrupt handled here */
00343:     }
00344:
00345:     if (HAL_PUSH_BUTTON1())
00346:     {
00347:         keys |= HAL_KEY_SW_6;
00348:     }
00349:
00350:     /* Invoke Callback if new keys were depressed */
00351:     if (keys && (pHalKeyProcessFunction))
00352:     {
00353:         (pHalKeyProcessFunction) (keys, HAL_KEY_STATE_NORMAL);
00354:     }
00355: } ? end HalKeyPoll ?
```




回调函数的一个非常的作用是可以隔离每一层，而且还有一点点面向对象的开发方式，在这里底层的按键查询函数调用一个函数指针，而非具体的函数，这样就将处理按键的接口留给了上层，上层应用中，只需解析函数指针传入的参数 1: keys 就知道是哪个按键被按下了。

我们再回到刚才的 OnBoard_KeyCallback 回调函数处，该回调函数代码如下：

```
00242: void OnBoard_KeyCallback ( uint8 keys, uint8 state )
00243: {
00244:     uint8 shift;
00245:     (void)state;
00246:
00247:     shift = (keys & HAL_KEY_SW_6) ? true : false;
00248:
00249:     if ( OnBoard_SendKeys( keys, shift ) != ZSuccess )
00250:     {
00251:         // Process SW1 here
00252:         if ( keys & HAL_KEY_SW_1 ) // Switch 1
00253:         {
00254:         }
00255:         // Process SW2 here
00256:         if ( keys & HAL_KEY_SW_2 ) // Switch 2
00257:         {
00258:         }
00259:         // Process SW3 here
00260:         if ( keys & HAL_KEY_SW_3 ) // Switch 3
00261:         {
00262:         }
00263:         // Process SW4 here
00264:         if ( keys & HAL_KEY_SW_4 ) // Switch 4
00265:         {
00266:         }
00267:         // Process SW5 here
00268:         if ( keys & HAL_KEY_SW_5 ) // Switch 5
00269:         {
00270:         }
00271:         // Process SW6 here
00272:         if ( keys & HAL_KEY_SW_6 ) // Switch 6
00273:         {
00274:         }
00275:     } ? end if OnBoard_SendKeys(keys... ?
00276: } ? end OnBoard KeyCallback ?
```

该函数又将按键值向下传递到 OnBoard_SendKeys() 函数里，通过名字应该能够看出来，应该是发送了按键消息。果然是调用了 osal_msg_send 发送了一个消息，并且在消息中附加了消息事件名称 KEY_CHANGE，按键值，和状态，按键的处理权又向下传递，最后发送到谁第一次注册按键服务端任务函数中去。

```
00210: uint8 OnBoard_SendKeys ( uint8 keys, uint8 state )
00211: {
00212:     keyChange_t *msgPtr;
00213:
00214:     if ( registeredKeysTaskID != NO_TASK_ID )
00215:     {
00216:         // Send the address to the task
00217:         msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
00218:         if ( msgPtr )
00219:         {
00220:             msgPtr->hdr.event = KEY_CHANGE;
00221:             msgPtr->state = state;
00222:             msgPtr->keys = keys;
00223:
00224:             osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
00225:         }
00226:         return ( ZSuccess );
00227:     }
00228:     else
00229:         return ( ZFailure );
00230: } ? end OnBoard SendKeys ?
```

注册了按键服务的任务ID



最后又回到了 SimpleBLEPeripheral 任务函数中。最终用户按了哪个按键，如何响应该按键在系统事件 SYS_EVENT_MSG 中处理。疑问又来了，为什么通过 osal_msg_send 发送的消息会出现在 SYS_EVENT_MSG 中呢？这个疑问，我们将在下一节中解答。

```
static void simpleBLEPeripheral_ProcessOSALMsg( osal_event_t *pMsg)
{
    switch ( pMsg->event )
    {
        #if defined( CC2540_MINIDK )
        case KEY_CHANGE:
            simpleBLEPeripheral_HandleKeys( ((keyChange_t *)pMsg)->state,
            break;
        #endif // #if defined( CC2540_MINIDK )

        default:
            // do nothing
            break;
    }
}
```

在 SimpleBLEPeripheral 中，对按键的响应如下：joystick left(SW2) 出发广播的开启和关闭

```
if ( keys & HAL_KEY_SW_2 )
{
    SK_Keys |= SK_KEY_RIGHT;

    // if device is not in a connection, pressing the right key should toggle
    // advertising on and off
    if( gapProfileState != GAPROLE_CONNECTED )
    {
        uint8 current_adv_enabled_status;
        uint8 new_adv_enabled_status;

        //Find the current GAP advertisement status
        GAPRole_GetParameter( GAPROLE_ADVERT_ENABLED, &current_adv_enabled_status )

        if( current_adv_enabled_status == FALSE )
        {
            new_adv_enabled_status = TRUE;
        }
        else
        {
            new_adv_enabled_status = FALSE;
        }

        //change the GAP advertisement status to opposite of current status
        GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8 ), &new_adv_enabled_status )
    }
}
```

以上，整个按键消息的发送和处理就结束了，现在读者应该比较清晰的理解的按键消息的处理过程，阅读协议栈代码比较重要的一点是：只跟踪函数肯定会跟丢，要求全局分析。Osal 层中使用了很多非常好的编程机制，值得我们好好的学习。

四、发送和接收自定义的 MSG 消息

启动和处理用户自定的 EVENT 事件是比较简单、容易理解的，那么如何发送和接收用户自定义的消息呢？上一节中也遗留了一个问题，也同样需要在这里回答，上一节的问题是为什么通过 osal_msg_send 发送的消息会出现在 SYS_EVENT_MSG 中呢？如果先前留意代码的读者应该知道了该问题的答案，看 osal_msg_send() 函数的注释中色线标记的区域，该函数同时会向目标任务列表中发起一个 message ready event 事件，所以会出现在 SYS_EVENT_MSG 中。在第一节中的第三个问题就描述这个问题。



```
00475: /*****
00476:  * @fn      osal_msg_send
00477:  *
00478:  * @brief
00479:  *
00480:  * This function is called by a task to send a command message to
00481:  * another task or processing element. The sending_task field must
00482:  * refer to a valid task, since the task ID will be used
00483:  * for the response message. This function will also set a message
00484:  * ready event in the destination tasks event list.
00485:  *
00486:  *
00487:  * @param    uint8 destination task - Send msg to? Task ID
00488:  * @param    uint8 *msg_ptr - pointer to new message buffer
00489:  * @param    uint8 len - length of data in message
00490:  *
00491:  * @return    SUCCESS, INVALID_TASK, INVALID_MSG_POINTER
00492:  */
00493: uint8 osal_msg_send( uint8 destination_task, uint8 *msg_ptr )
00494: {
00495:     if ( msg_ptr == NULL )
00496:         return ( INVALID_MSG_POINTER );
00497:
00498:     if ( destination_task >= tasksCnt )
```

那么如何发送和接收用户自定义的消息呢？

在 ZComDef.h 头文件中定义了先前看到的消息命令的宏定义

```
00382: #define AF_DATA_CONFIRM_CMD          0xFD    // Data confirmation
00383: #define AF_INCOMING_MSG_CMD          0x1A    // Incoming MSG type message
00384: #define AF_INCOMING_KVP_CMD          0x1B    // Incoming KVP type message
00385: #define AF_INCOMING_GRP_KVP_CMD      0x1C    // Incoming Group KVP type message
00386:
00387: // #define KEY_CHANGE                  0xC0    // Key Events
```

然后在稍微下面一点有这样的注释代码：OSAL 系统消息保留给用于应用程序的 IDs 和 EVENTS，为 0xE0~0xFC。

```
00402: #define ZCL_OTA_CALLBACK_IND          0x36    // ZCL OTA Completion Indication
00403:
00404:
00405: // OSAL System Message IDs/Events Reserved for applications (user applications)
00406: // 0xE0 ~ 0xFC
```

这里要说明一点，消息往往用于不同任务函数之间的数据传递，在同一任务函数中使用全局函数或者自动以用户事件完全能够胜任。如果用户是在需要自定义用户消息，可以参考下列操作。

- 4.1 在 xxxApp.h 中宏定义用户 MSG 消息，注意取值范围为 0xE0~0xFC
- 4.2 使用 osal_msg_allocate 函数创建所要发送的消息的结构体空间，例如 KEY_CHANGE 消息：

```
msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
```

- 4.3 填充消息结构体成员，例如 KEY_CHANGE

```
msgPtr->hdr.event = KEY_CHANGE;
msgPtr->state = state;
msgPtr->keys = keys;
```

- 4.4 使用 osal_msg_send 函数向目标任务 ID 发送该消息



```
osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
```

4.5 最后在目标任务的任务处理函数中的处理刚才填充消息

```
MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( SampleApp_TaskID );
while ( MSGpkt )
{
    switch ( MSGpkt->hdr.event )
    {
        // Received when a key is pressed
        case KEY_CHANGE:
            SampleApp_HandleKeys( ((keyChange_t *)MSGpkt)->state, ((keyChange_t *)MSGpkt)->key);
            break;
        case USR_MSG:
            //do something here
            break;
    }
}
```

此时，应该完全体会到了 EVENT 与 MSG 消息的区别。

EVNET 用于同一任务函数传递命令，而 MSG 则用于不同的任务函数传递命令。

未完，待续。。。