密级：公开资料

TTC BLE SDK V3.0
RTOS 简介

文件版本：V1.0

适用 SDK 版本：V3.0

深圳市昇润科技有限公司

2017 年 03 月 28 日

版本	修订日期	修订人	审稿人	修订内容
1.0	2017-03-28	郭高亮	张眼	初稿

目　　录

1. TI RTOS 概述

TI-RTOS 是一个抢占式、多线程的实时操作系统。可以执行硬件中断、软件中断、任务、睡眠，且优先级依次降低。下面依次介绍事件、任务、定时任务、消息、掉电存储及动态内存管理六个概念，并以 TTC_CC2640_SDK 工程为示例，进行简要说明。

1.1. 事件(Event)

在 TI-RTOS 中，事件可用于同步两个任务的操作，如协调应用程序与 BLE 协议栈两个任务对共享资源的访问。通过置起事件可以唤醒任务。

1.1.1. 相关函数

(1) Event_pend()

调用此函数后，线程将会挂起直到设定的挂起超时时间到达，或者直到 Event_post()函数被调用。ICALL_TIMEOUT_FOREVER 表示不会超时。

(2) Event_post()

更高优先级的线程，比如硬件/软件中断可以通过调用 Event_post()置起特定事件，以唤醒线程。

1.1.2. 示例说明

以 TTC_CC2640_SDK 工程中 TTCDemoSystemClock.c 为例，解释事件的用法。

(1) 定义线程事件句柄 syncEvent (实际是一个指针)：

```
<TTCBlePeripheralTask.c>
static ICall_SyncHandle  syncEvent;          //线程事件句柄，用于唤醒线程
```

(2) 将事件句柄 syncEvent 传入 TTCDemoSystemClcokInit 初始化函数：

```
<TTCBlePeripheralTask.c>
static void TTCBlePeripheralTaskInit(void){
    //…
    TTCDebugInit(syncEvent,
                 appMsgQueue,
                 &TTCBlePeripheralTaskCls);
    //…
```

```
<appCommParam.c>
void TTCDebugInit(Event_Handle eventHandle,
                  Queue_Handle queueHandle,
                  TTCSdkClass_t *pCB) {
    //…
    TTCDemoInit(pCB,
                eventHandle,
                queueHandle);                 //驱动初始化
    //…
```

```
<TTCDemo.c>
void TTCDemoInit( TTCSdkClass_t *pCB,
                  Event_Handle eventHandle,
                  Queue_Handle queueHandle){
    //…
    TTCDemoSystemClcokInit(eventHandle);
    //…
```

(3) 定义事件句柄，TTCDemoSystemClcokInit 初始化。

```
<TTCDemoSystemClock.c>
static Event_Handle sysClockEventHandle;
-------------------------------------------------------------------------------
void TTCDemoSystemClcokInit(Event_Handle eventHandle){
      //…
      sysClockEventHandle = eventHandle;
      //…
    }
```

(4) 软件中断中(具体为何会进入软件中断，见1.3 节定时任务)，调用 Event_post()函数置起事件。

注意，以下函数置起了两个事件：一个为应用程序的事件 arg，另一个是用于唤醒线程所需要的事件 UTIL_QUEUE_EVENT_ID.

```
<TTCDemoSystemClock.c>
static void TTCDemoSystemClockCB(UArg arg){
    //…
    sysClockEvt |= arg;
    //…
    Event_post(sysClockEventHandle, UTIL_QUEUE_EVENT_ID);
    //…
```

(5) Event_pend()等待事件唤醒线程

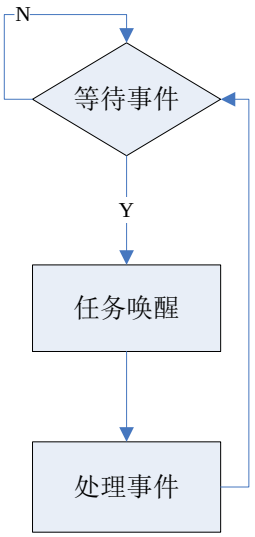
```
<TTCBlePeripheralTask.c>
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
    //…
    for(;;)
        //…
        events = Event_pend(syncEvent,
                            Event_Id_NONE,
                            TTC_ALL_EVENTS,
                            ICALL_TIMEOUT_FOREVER);
        //…
```

1.2. 任务(Task)

系统每时每刻有且只有一个任务正在执行，也可能是空任务。每个任务执行结束后，处理器再去执行就绪状态中优先级最高的任务。当优先级相等时，则按照“先就绪先执行”的原则执行。

在从机工程中，BLE 协议栈任务优先级最高，应用程序任务优先级最低。任务大部分时间处于挂起状态，一旦事件被置起，任务被唤醒，处理相应事件，如下图所示。

应用程序的种种需要处理的事件，均可以在已有的任务中处理，无需创建新的任务。以 TTC_CC2640_SDK 工程中 TTCDemoSystemClock.c 为例，解释在应用程序任务中事件处理的用法。



1.2.1. 相关函数

(1) Task_Params_init()

初始化任务相关参数。

(3) Task_construct()

创建任务。

1.2.2. 示例说明

以从机角色为例，说明任务创建，及如何处理用户事件。

(1) 创建任务：在 TI-RTOS 启动之前创建任务

```
<main.c>
int main() {
  //…
  TTCBleGAPRoleCreateTask();                    //从机任务
  //…
  BIOS_start();                                 //TI-RTOS 启动
  //…
```

初始化任务相关参数，创建任务：

```
<TTCBlePeripheralTask.c>
Task_Struct  sbcTask;                                    //线程配置
void TTCBlePeripheralCreateTask(void){
      Task_Params taskParams;                            //任务相关参数
      Task_Params_init(&taskParams);                     //初始化
      taskParams.stack     = sbcTaskStack;               //任务对应的堆栈
      taskParams.stackSize = SBC_TASK_STACK_SIZE;        //任务对应的堆栈大小
      taskParams.priority  = SBP_TASK_PRIORITY;          //任务优先级
      Task_construct(&sbcTask,                           //创建任务
                     TTCBlePeripheralTaskFxn,            //任务函数
                     &taskParams,
                     NULL);
  }
```

(2)任务函数：等待唤醒，若唤醒则处理用户事件

```
<TTCBlePeripheralTask.c>
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
    //…
    for(;;)
        //…
        events = Event_pend(syncEvent,                   //syncEvent 无需用户清 0
                            Event_Id_NONE,
                            TTC_ALL_EVENTS,
                            ICALL_TIMEOUT_FOREVER);
        //…
        TTCDebugEvent();                                 //事件唤醒后，处理用户事件
        //…
  }
```

(3)处理用户事件

```
<appCommParam.c>
void TTCDebugEvent(void) {
    #ifdef TTC_DEBUG
      TTCDriverUartEvent();
      TTCDemoEvent();
    #endif
}
```

```
<TTCDemo.c>
void TTCDemoEvent(void){
  #ifdef TTCDRIVER_SYSCLOCK
```

```
    TTCDemoSystemClockEvt();
  #endif //TTCDRIVER_SYSCLOCK
}
```

处理相应的用户事件，并清 0.

```
<TTCDemoSystemClock.c>
void TTCDemoSystemClockEvt(void){
    if(sysClockEvt & SYSTEM_CLOCK_EVENT){                    //判断用户事件是否置起
        ICall_CSState key = ICall_enterCriticalSection();
        sysClockEvt &= ~SYSTEM_CLOCK_EVENT;                  //相应用户事件清 0
        ICall_leaveCriticalSection(key);
        TTCDriverIOSetOutputVaule(&systemClockPinHandle,
                                  Board_SysClock,
                                  ~TTCDriverIOGetOutputValue(Board_SysClock));
    }
}
```

1.3. 定时任务(Clock)

定时任务可以定时启动相应事件处理，当定时到达时触发软件中断，进入定时任务回调函数。比如,在回调函数中可以调用 TTCSdkSetEvent()函数，以便置起相应事件。

1.3.1. 相关函数

(1) Util_constructClock()
创建一个定时任务。
(2) Util_startClock()
启动定时任务。
(3) Util_restartClock()
重启定时任务，改变定时时间。
(4) Util_isActive()
判断定时任务是否在运行。
(5) Util_stopClock()
终止定时任务。
(6) Util_rescheduleClock()
重新配置定时任务。

1.3.2. 示例说明

以 TTC_CC2640_SDK 工程使用 TTCDemoSystemClock.c 为例，解释定时任务的用法。

(1) 创建定时任务，即定义一个定时时钟变量 sysClock：

```
<TTCDemoSystemClock.c>
static Clock_Struct sysClock;
```

(2) 在初始化函数 TTCDemoSystemClcokInit()中，初始化定时任务

```
<TTCDemoSystemClock.c>
Util_constructClock(&sysClock,                    // 软件定时器
                    TTCDemoSystemClockCB,         // 回调函数名
                    sysClockPeriod,               // 定时时间(ms)
                    sysClockPeriod,               // 定时时间(ms)
                    false,                        // 初始化后立即启动
                    SYSTEM_CLOCK_EVENT);          // 回调函数带的参数
```

备注：此定时任务在初始化后不会立刻开启。开启定时器后，当 sysClockPeriod(即 1000ms)到达时，调用回调函数并带上参数如 TTCDemoSystemClockCB(SYSTEM_CLOCK_EVENT)；具体见以下入口参数说明。

< util.c> Util_constructClock 原型，对应入口参数说明：

```
Clock_Handle Util_constructClock(Clock_Struct *pClock,
                                 Clock_FuncPtr clockCB,
                                 uint32_t clockDuration,
                                 uint32_t clockPeriod,
                                 uint8_t startFlag,
                                 UArg arg)
```

- pClock：时钟变量地址
- clockCB：定时时间到达时，进入此定时任务回调函数
- clockDuration：第一次启动定时任务的定时时间(ms)
- clockPeriod：重复周期(ms),0 则不重复，非 0 则重复。
  (1) 值为 0 时，定时任务为单周期，开启定时任务后到达 clockDuration 设置的时间，调用回调函数后停止。
  (2) 值非 0 时，定时任务为重复周期，开启定时任务后，第一次定时时间为 clockDuration，后续定时时间为 clockPeriod，且定时任务自动周期性执行。
- startFlag：立即开启标志
  (1)true：无需调用开启函数，立即自行开启；
  (2)false：不会自行开启，若需启动定时任务，则需调用开启函数；
- arg：回调函数的参数

(3) 定时任务启动后到达定时时间,进入回调函数,置 SYSTEM_CLOCK_EVENT 事件(事件处理方式见 1.1 节）。

```
<TTCDemoSystemClock.c>
-------------------------------------------------------------------------------

static void TTCDemoSystemClockCB(UArg arg){                    //定时任务的回调函数
    ICall_CSState key = ICall_enterCriticalSection();
    sysClockEvt |= arg;
```

```
    ICall_leaveCriticalSection(key);
    Event_post(sysClockEventHandle, UTIL_QUEUE_EVENT_ID);
}
```

(4) 重新启动定时任务：

```
<TTCDriverUARTDemo.c>
-------------------------------------------------------------------------
CMDState_t TTCDemoSystemClockCfg(
    //…
     sysClockPeriod = getPeriod;
     Util_rescheduleClock(&sysClock,sysClockPeriod);     //重新设置定时器的定时时间
     if(!Util_isActive(&sysClock)){                      //判断定时任务是否在运行
          Util_startClock(&sysClock);                    //保持定时时间不变，启动定时任务
        }
    //…
```

(5) 停止定时任务：定时任务开启后可以在任何时候停止

```
Util_stopClock(&sysClock);
```

1.4. 消息(Measege)

当需要处理的事件对时序要求严格，可以使用定时任务处理；而当事件必须按照先后顺序执行，消息则为更好的办法，因为消息队列按照“先进先出”的原则处理。消息不仅可以用于处理同一线程内部的事件；也可用于线程同步，如 Task A 中发消息，可在 Task B 中处理消息。

1.4.1. 相关函数

TI-RTOS 消息处理相关函数均在 util.c 中，包含以下三个函数：

(1) Util_constructQueue()

消息初始化函数，创建一个消息队列。

(2) Util_enqueueMsg()

创建一个消息节点，并将消息“压入”消息队列中，如上图中”put”操作。

(3) Util_dequeueMsg()

“取出”最先进入消息队列的消息节点，如上图中”get”操作。

1.4.2. 示例说明

以 TTC_CC2640_SDK 工程中串口(UART)处理 AT 指令的过程为例，介绍如何发消息、处理消息。

(1) 定义线程消息相关变量：

```
<TTCBlePeripheralTask.c>
-------------------------------------------------------------------------------

static Queue_Struct appMsg;                          //消息结构
static Queue_Handle appMsgQueue;                     //消息句柄
```

(2) 创建消息队列：

应用程序任务中已经创建好对应的消息队列，无需再自行新建消息队列。

```
<TTCBlePeripheralTask.c>
-------------------------------------------------------------------------------

static void TTCBlePeripheralTaskInit(void){
    ICall_registerApp(&selfEntity, &sem);
    appMsgQueue = Util_constructQueue(&appMsg);     // 创建消息队列
    //…
    }
```

(3) 串口初(URAT)初始化，将事件句柄、消息句柄传入串口初始化函数：

```
<TTCDriverUARTDemo.c>
-------------------------------------------------------------------------------

Event_Handle     pTTCSDKDebugaEvent;                 //事件句柄（指针变量）
Queue_Handle     pTTCSDKDebugQueueHandle;            //消息句柄（指针变量）
-------------------------------------------------------------------------------

//串口初始化
TTCDebugInit(syncEvent,                              //事件句柄
             appMsgQueue,                            //消息句柄
             &TTCBlePeripheralTaskCls);
-------------------------------------------------------------------------------

//函数原型
void TTCDebugInit( Event_Handle eventHandle,
                   Queue_Handle queueHandle,
                   TTCSdkClass_t *pCB) {
    //…
    pTTCSDKDebugaEvent      = eventHandle;
    pTTCSDKDebugQueueHandle = queueHandle;
    //…
```

(4) 发消息，消息“压入”消息队列。CC2640 模组接收到 UART AT 指令后，会进入串口回调函数 TTCDebugReadCB()：

```
<TTCDriverUARTDemo.c>
-------------------------------------------------------------------------------

//发消息
```

```
static void TTCDebugReadCB(void * handle, u8 * buffer, u16 len){
    TTCMsg_t *pMsg;
    if ((pMsg = ICall_malloc( sizeof(TTCMsg_t) + sizeof(u8)*len ))) {
        //…
        pMsg->hdr.event = TTCSDK_MSG_DRIVER_UART_EVENT; //发消息所携带的事件
        pMsg->hdr.state = len;                          //串口数据长度
        pMsg->pValue    = (u8*)(&pMsg->pValue + 1);     //串口数据
        memcpy(pMsg->pValue, buffer, len);
        Util_enqueueMsg(pTTCSDKDebugQueueHandle,
                        pTTCSDKDebugaEvent,
                        (u8*)pMsg);
    }
}
```

需要注意，SDK V3.0 的 TI-RTOS 在发消息的同时，也会置起一个特定的事件 UTIL_QUEUE_EVENT_ID，用于唤醒线程：

```
<util.c>
-------------------------------------------------------------------------------
//发消息函数原型
uint8_t Util_enqueueMsg(Queue_Handle msgQueue,
                        Event_Handle event,
                        uint8_t *pMsg)
    //…
    Queue_put(msgQueue, &pRec->_elem);              //发消息
    //…
    Event_post(event, UTIL_QUEUE_EVENT_ID);         //自动置起事件
    //…
{
```

(5) 唤醒线程，取出消息队列中的消息，并处理：

注意：TTC_QUEUE_EVT 即 UTIL_QUEUE_EVENT_ID

```
<TTCBlePeripheralTask.c>
-------------------------------------------------------------------------------
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
  //…
  for (;;){
    uint32_t events;
    events = Event_pend(syncEvent,
                        Event_Id_NONE,
                        TTC_ALL_EVENTS,
                        ICALL_TIMEOUT_FOREVER);
    if (events) {
```

```
        //…
        if(events & TTC_QUEUE_EVT) {
           while (!Queue_empty(appMsgQueue)){          //判断消息队列是否为空
              TTCMsg_t *pMsg = (TTCMsg_t *)Util_dequeueMsg(appMsgQueue);
              if (pMsg){
                 TTCBlePeripheralTaskProcessAppMsg(pMsg);      //处理消息
                 ICall_free(pMsg);                             //释放内存
               }
    //…
}
-------------------------------------------------------------------------------
static void TTCBlePeripheralTaskProcessAppMsg(TTCMsg_t *pMsg){
    switch (pMsg->hdr.event){
        //…
         case TTCSDK_MSG_DRIVER_UART_EVENT:{          //发消息所携带的事件
               //处理 AT 指令
              }break;
        //…
    }
}
```

1.5. 掉电存储

SNV 为 CC2640 内部 flash 中的一部分，根据 SDK 工程的配置不同，SNV 的大小可以为 0 或者 4Kbyte.可用于存储绑定信息，或存储少量用户数据，实现数据掉电保存。

osal_snv_read()及 osal_snv_write()分别为 SNV 的读写函数，所需要参数为 NV ID、长度及数据指针。

```
//函数原型：
uint8 osal_snv_read( osalSnvId_t id,      //NV ID
                     osalSnvLen_t len,    //数据长度
                     void *pBuf)          //读数据指针

uint8 osal_snv_write( osalSnvId_t id,     //NV ID
                      osalSnvLen_t len,   //数据长度
                      void *pBuf)         //写数据指
//参数说明
(1) NV ID：用户可用 NV ID 范围是 0x80~0x8F
    #define BLE_NVID_CUST_START 0x80
    #define BLE_NVID_CUST_END   0x8F
(2) 数据长度：最长为 252 字节
数据指针
```

1.6. 动态内存管理

为了灵活合理应用内存，**应用程序可以动态申请内存，用完后必须释放申请的内存，否则导致内存泄露！**以下为内存申请、内存释放的简要示例：

(1) 发送消息时，ICall_malloc()函数为消息数据申请缓存：

```
<appCommParam.c>
-------------------------------------------------------------------------------
static void TTCDebugReadCB(void * handle, u8 * buffer, u16 len){
  TTCMsg_t *pMsg;
   //…
  if ((pMsg = ICall_malloc( sizeof(TTCMsg_t) + sizeof(u8)*len ))) {
        pMsg->hdr.event = TTCSDK_MSG_DRIVER_UART_EVENT;
        pMsg->hdr.state = len;
        pMsg->pValue    = (u8*)(&pMsg->pValue + 1);
        memcpy(pMsg->pValue, buffer, len);
        Util_enqueueMsg(pTTCSDKDebugQueueHandle,
                        pTTCSDKDebugaEvent,
                        (u8*)pMsg);
    }
}
```

(2) 从消息队列中取出消息，消息数据使用完毕后，调用 ICall_free()释放内存：

```
<TTCBlePeripheralTask.c>
-------------------------------------------------------------------------------
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
  //…
  for (;;){
    uint32_t events;
    events = Event_pend(syncEvent,
                        Event_Id_NONE,
                        TTC_ALL_EVENTS,
                        ICALL_TIMEOUT_FOREVER);
    if (events) {
       //…
       if(events & TTC_QUEUE_EVT) {
          while (!Queue_empty(appMsgQueue)){
             TTCMsg_t *pMsg = (TTCMsg_t *)Util_dequeueMsg(appMsgQueue);
             if (pMsg){
                TTCBlePeripheralTaskProcessAppMsg(pMsg);      //消息处理
                ICall_free(pMsg);                             //释放内存
              }
    //…
}
```

2.联系我们

深圳市昇润科技有限公司
ShenZhen ShengRun Technology Co.,Ltd.
Tel: 0755-86233846 Fax: 0755-82970906
官网地址：www.tuner168.com
阿里巴巴网址：http://shop1439435278127.1688.com
E-mail: marketing@tuner168.com
地址： 广东省深圳市南山区西丽镇龙珠四路金谷创业园 B 栋 6 楼 601-602