密级：公开资料

TTC BLE SDK
RTOS 简介

文件版本：V1.0

深圳市昇润科技有限公司

2016 年 12 月 05 日

版本	修订日期	修订人	审稿人	修订内容
1.0	2016-12-05	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，优先级依次降低。下面依次介绍信号量、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 信号量(Semaphore)

在 TI-RTOS 中，信号量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。通过置起信号量可以唤醒任务。以 TTC_CC2640_SDK 工程中串口 demo 为例，解释信号量的用法。

(1) 定义线程信号量 sem(无需再自行定义)：

```
<TTCBlePeripheralTask.c>
static ICall_Semaphore sem;          //线程信号量，用于唤醒线程
```

(2) 将信号量地址&sem 传入串口初始化函数：

```
<TTCBlePeripheralTask.c>
#ifdef TTCDRIVER_UART
   TTCDriverDemoUARTInit(&TTCBlePeripheralTaskCls,&sem,&appMsgQueue);
#endif //TTCDRIVER_UART
```

(3) 定义信号量指针 UartSem，串口初始化成功后，该指针则指向在 TTCBlePeripheralTask.c 定义的信号量 sem.

```
<TTCDriverUARTDemo.c>
static ICall_Semaphore * UartSem;
------------------------------------------------------------------------
void TTCDriverDemoUARTInit(TTCSdkClass_t *appCallbacks,
                           ICall_Semaphore * sem,
                            Queue_Handle * appMsgQueue){
        //…
        UartSem = sem;
        //…
    }
```

(4) Semaphore_post()函数置起信号量

```
<TTCDriverUARTDemo.c>
static void TTCSDKDriverUARTSetEvent(UArg arg){
    events |= arg;
    Semaphore_post(*UartSem); //置起信号量，唤醒线程
   }
```

(5) ICall_wait()等待信号量唤醒线程

```
<TTCBlePeripheralTask.c>
```

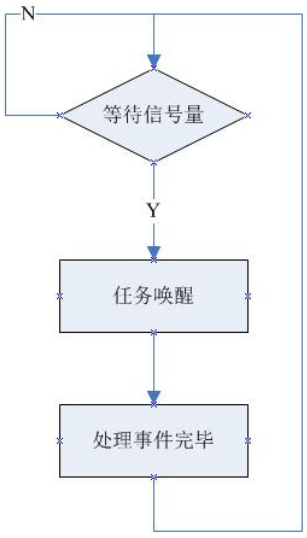
```
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
    TTCBlePeripheralTaskInit();
    for (;;){
        ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);
        //…
    }
```

1.2 任务及事件处理(Task & Event)

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，一旦信号量被置起，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 工程中串口 demo 为例，解释在应用程序任务中事件处理的用法。



(1) 定义事件变量 events：

```
<TTCDriverUARTDemo.c>
static u16 events;                                  //本地事件
#define TTCBLE_SDK_UART_EVN     0x0001              //串口事件
```

(2) 定时任务回调中置起事件 TTCBLE_SDK_UART_EVN，注意事件宏定义方式为位域方式，及 0x01,0x02,0x04,0x08,0x10,0x20,依次类推，不可重复。

```
<TTCDriverUARTDemo.c>
Util_constructClock(&uClock,
                    TTCDriverDemoUARTClockHandler,    //回调函数
                    1000,  0,  true,
```

```
                        TTCBLE_SDK_UART_EVN);              //回调函数参数,串口事件
-------------------------------------------------------------------------------
static void TTCDriverDemoUARTClockHandler(UArg arg){
    TTCSDKDriverUARTSetEvent(arg);
}
-------------------------------------------------------------------------------
static void TTCSDKDriverUARTSetEvent(UArg arg){      //arg 为 TTCBLE_SDK_UART_EVN
    events |= arg;                                    //置起串口事件
    Semaphore_post(*UartSem);                         //唤醒线程
}
```

(3) 有置起事件，也需要有相应的事件处理函数 TTCSDKDriverUARTEvent()

```
<TTCDriverUARTDemo.c>
void TTCSDKDriverUARTEvent(void){
    if(events & TTCBLE_SDK_UART_EVN){                 //查询串口事件是否发生
       events &= ~TTCBLE_SDK_UART_EVN;                //清除串口事件
       TTCDriverUartWrite(&uartHandle,"TTCDriverUART Test\r\n",
                          strlen("TTCDriverUART Test\r\n"));
       Util_startClock(&uClock);
    }
}
```

(4) 事件置起时,同时也置起了信号量。线程唤醒后,调用 TTCSDKDriverUARTEvent()

```
<TTCBlePeripheralTask.c>
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
    TTCBlePeripheralTaskInit();
    for (;;){
       ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);

        #ifdef TTCDRIVER_UART
          TTCDriverUartEvent();
          TTCSDKDriverUARTEvent();                    //处理串口事件
        #endif //TTCDRIVER_UART
        //…
     }
}
```

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，以便置起相应事件及信号量。以 TTC_CC2640_SDK 工程使用串口 demo 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 uClock：

```
<TTCDriverUARTDemo.c>
Clock_Struct uClock;
```

(2) 在串口初始化函数 TTCDriverDemoUARTInit()中，初始化定时任务

```
<TTCDriverUARTDemo.c>
Util_constructClock(&uClock,                          // 软件定时器
                    TTCDriverDemoUARTClockHandler,    // 回调函数名
                    1000,                             // 定时时间(ms)
                    0,                                // 单周期定时任务
                    true,                             // 初始化后立即启动
                    TTCBLE_SDK_UART_EVN);             // 回调函数带的参数
```

备注：此定时任务在初始化后立刻开启，到 1000ms 后，调用回调函数时带上参数如 TTCDriverDemoUARTClockHandler(TTCBLE_SDK_UART_EVN)；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 TTCBLE_SDK_UART_EVN 事件（事件处理方式见 1.2 节）。

```
<TTCBlePeripheralTask.c>
static void TTCDriverDemoUARTClockHandler(UArg arg){  //串口事件
    TTCSDKDriverUARTSetEvent(arg);
}
```

(4) 重新启动定时任务,方法一：

```
<TTCDriverUARTDemo.c>
void TTCSDKDriverUARTEvent(void){
    if(events & TTCBLE_SDK_UART_EVN){
       events &= ~TTCBLE_SDK_UART_EVN;
       TTCDriverUartWrite(&uartHandle,"TTCDriverUART Test\r\n",
                          strlen("TTCDriverUART Test\r\n"));
       //再次启动定时任务，定时时间仍为 1000ms
       Util_startClock(&uClock);
    }
}
```

(5) 重新启动定时任务，方法二（可更改定时时间）：

```
<TTCDriverUARTDemo.c>
void TTCSDKDriverUARTEvent(void){
    if(events & TTCBLE_SDK_UART_EVN){
       events &= ~TTCBLE_SDK_UART_EVN;
       TTCDriverUartWrite(&uartHandle,"TTCDriverUART Test\r\n",
                          strlen("TTCDriverUART Test\r\n"));
       //再次启动定时任务，定时更改为 2000ms
       Util_restartClock(&uClock, 2000);
    }
}
```

(6) 停止定时任务，定时任务开启后可以在任何时候停止

```
Util_stopClock(&uClock);
```

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，如下图。可以在 Task A 中发消息，在 Task B 中处理消息。

上图中”put”操作可以通过 TTCSdkTaskEnqueueMsg()函数实现，即将消息“压入”消息队列中；

”get”操作可以通过 Util_dequeueMsg()函数实现，即“取出”最先进入消息队列的消息，并在 TTCBlePeripheralTaskProcessAppMsg()函数中进行相应事件处理。

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。以 TTC_CC2640_SDK 工程中串口 demo 为例，介绍如何发消息、处理消息。

(1) 定义线程消息句柄：

```
<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：

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

(3) 注册发消息函数，并将消息句柄地址&appMsgQueue 传入串口初始化函数：

```
<TTCBleSDKConfig.h>
typedef struct {                               //TTC SDK 公共函数类
    TTCSdkSetEvent_t            pfnTTCSdkSetEvent;
    TTCSdkTaskEnqueueMsg_t      pfnTTCSdkTaskEnqueueMsg;
    TTCSdkDriverCB_t            pfnTTCSdkDriverCB;
    TTCSdkBleCB_t               pfnTTCSdkBleCB;
}TTCSdkClass_t;
-------------------------------------------------------------------------
<TTCBlePeripheralTask.c>
TTCSdkClass_t TTCBlePeripheralTaskCls = {
    TTCSdkSetEvent,
    TTCSdkTaskEnqueueMsg,
    TTCSdkDriverCB,
};
-------------------------------------------------------------------------
#ifdef TTCDRIVER_UART
   TTCDriverDemoUARTInit(&TTCBlePeripheralTaskCls, &sem, &appMsgQueue);
#endif //TTCDRIVER_UART
```

(4) 定义串口应用程序事件、消息句柄指针 UartMsgQueue、回调函数指针 UartCallbacks，串口初始化成功后，该 UartMsgQueue 指针则指向在 TTCBlePeripheralTask.c 定义的消息句柄 appMsgQueue. UartCallbacks 指针指向 TTCBlePeripheralTask.c 定义的 TTCBlePeripheralTaskCls.

```
<TTCBleSDKConfig.h>
#define TTCSDK_MSG_APP_UART_EVENT        0x000C    //TTCSDK 串口应用程序事件
-------------------------------------------------------------------------
<TTCDriverUARTDemo.c>
static Queue_Handle * UartMsgQueue = NULL;        //消息句柄指针
```

```
static TTCSdkClass_t * UartCallbacks = NULL;      //回调注册
-------------------------------------------------------------------------------
void TTCDriverDemoUARTInit(TTCSdkClass_t *appCallbacks,
                           ICall_Semaphore * sem,
                           Queue_Handle * appMsgQueue){
    if(appCallbacks == NULL || sem == NULL || appMsgQueue == NULL){
       return;
    }
    UartSem       = sem;
    UartMsgQueue  = appMsgQueue;
    UartCallbacks = appCallbacks;
    //…
  }
```

(5) 利用回调发消息，置起信号量，消息“压入”消息队列。

```
<TTCDriverUARTDemo.c>
-------------------------------------------------------------------------------
void TTCSDKDriverUARTEvent(void){
    if(events & TTCBLE_SDK_UART_EVN){
        events &= ~TTCBLE_SDK_UART_EVN;
        TTCDriverDemoUARTSendMsg(TTCSDK_MSG_APP_UART_EVENT, 0, NULL);
    }
}
-------------------------------------------------------------------------------
static void TTCDriverDemoUARTSendMsg(uint16_t event,
                                     uint16_t len, uint8_t *pData){
    if ( UartCallbacks != NULL &&
        UartCallbacks->pfnTTCSdkTaskEnqueueMsg != NULL){
        UartCallbacks->pfnTTCSdkTaskEnqueueMsg(*UartSem,
                                               *UartMsgQueue,
                                               event,
                                               NULL,
                                               NULL);
    } else {
    }
  }
  备注：通过回调实际调用是 TTCSdkTaskEnqueueMsg();
```

(6) 唤醒线程，取出消息队列中的消息，并处理

```
<TTCBlePeripheralTask.c>
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
  //…
```

```
  if (errno == ICALL_ERRNO_SUCCESS){
      TTCBlePeripheralProcessStack(selfEntity);          //协议栈消息处理
      while (!Queue_empty(appMsgQueue)){                 //接收消息
         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_APP_UART_EVENT:{
             // TTCSDK 串口应用程序事件处理
             }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！**按照 SDK 默认设置，缓存大小默认为 2672Byte，用户申请内存建议不超过 700Byte；如需动态申请更多内存空间，请修改宏 HEAPMGR_SIZE 的大小(Project->Options->C/C++Complier->Defined Symbols)。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：

```
static u8 TTCSdkTaskEnqueueMsg(ICall_Semaphore sem,
                               Queue_Handle queueHandle,
                               u16 event,
                               u16 state,
                               void * pValue){
      u8 status = FALSE;
      TTCMsg_t *pMsg;
      if ((pMsg = ICall_malloc(sizeof(TTCMsg_t)))) {
          pMsg->hdr.event = event;
          pMsg->hdr.state = state;
          pMsg->pValue = pValue;
          status = Util_enqueueMsg(queueHandle, sem, (u8*)pMsg);
      }
      return status;
}
```

(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：

```
static void TTCBlePeripheralTaskFxn(UArg a0, UArg a1){
  //…
  if (errno == ICALL_ERRNO_SUCCESS){
      TTCBlePeripheralProcessStack(selfEntity);          //协议栈消息处理
      while (!Queue_empty(appMsgQueue)){                 //接收消息
         TTCMsg_t *pMsg = (TTCMsg_t *)Util_dequeueMsg(appMsgQueue);
         if (pMsg){
              TTCBlePeripheralTaskProcessAppMsg(pMsg); //处理消息
              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