TI CC2540/41
BLE 软件开发指南

Ghostyu
2013-03-06

版本

V1.0	2013-03	初始版发布

目的

本文在 OSAL 入门指南和 BLE 开发简介的基础上进一步阐述低功耗蓝牙 BLE 的软件开发，本手册将对 TI BLE 作相对全面的描述，希望帮助读者入门 BLE 的软件开发

阅读本文档前，请先阅读下列文档

TI BLE 简要说明

OSAL 编程指南

1 Bluetooth

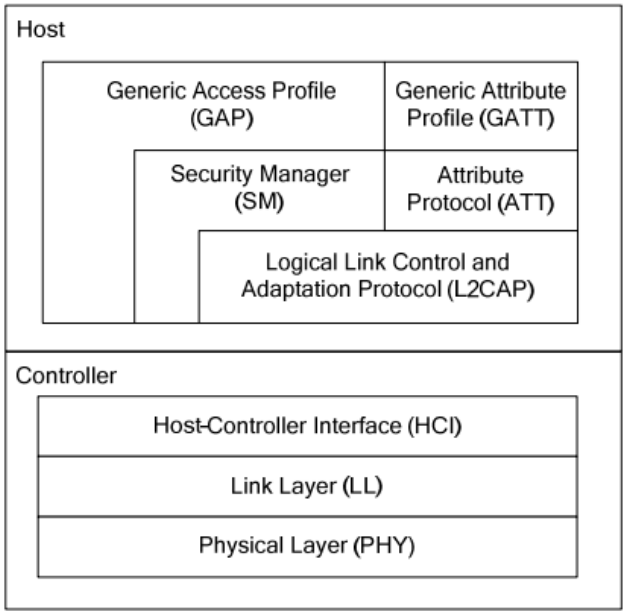
蓝牙 4.0 是 2012 年公布的最新标准，目标是更省电，通信距离更长，成本更低，TI CC2540 便是一颗低功耗蓝牙 4.0 的芯片。

蓝牙 4.0 标准定义了两种无线技术，1：BasicRate（BR），2：Bluetooth Low energy（BLE） 第二种无线技术，BLE 系统旨在每次传输非常小的数据包，这样消耗更低的电量。

同时支持 BR 和 BLE 的设备为 dual-mode 设备（Bluetooth SMART READY），通常，我们常用的智能手机、笔记本电脑都是 dual-mode 设备，仅支持 BLE 的为 single-mode 设备（Bluetooth SMART）。Single-mode 设备通常使用纽扣电池供电，这也代表这 single-mode 设备消耗的电能非常小。

1.1 BLE 协议栈

BLE 协议栈如下图所示：



该协议栈有两部分组成 Host 和 Controller，这种分离的主机和控制器追溯到标准的蓝牙 BR/EDR

（Enhance Data Rate）设备。就是说蓝牙 4.0 之前的版本，这两部分是分开的。

所有的 profile(暂且理解为一种配置)和应用程序都建立在协议栈的 GAP 和 GATT 之上，在接下来的 TI BLE 协议栈应用程序开发中，我们调用的 api 函数也大多数也来自 GAP 和 GATT，我们先记住他们的名字，具体作用接下来会描述。

PHY 层，最底层，1Mbps 自适应调频技术，运行在免证的 2.4GHz。

LL 层，RF 控制层，控制芯片工作在 standby（准备）、advertising（广播）、scanning（监听/扫描），initiating（发起连接）、connected（已连接）这五个状态中的一种。五种状态的切换描述为：advertising（广播）不需要连接就可以发送数据（告诉所有人，我来了），scanning（监听/扫描）来自广播的数据，initiator（发起人）将携带 connection request（连接请求）来相应广播者，如果 advertiser（广播者）同意该请求，那么广播这和发起者都会进入已连接状态，发起连接的设备变为 master（主机），接收连接请求的设备变为 slave（从机）。

HCI 层，通信层，向 host 和 controller 提供一个标准化的接口。该层可以由软件 api 实现或者使用硬件接口 uart、spi、usb 来控制。

L2CAP 层，相当于快递，将数据打包，可以让客户点对点的通信。

SM 层，安全服务层，提供配对和密钥的分发，实现安全连接和数据交换。

ATT 层，

GATT 层，

2 TI BLE 软件开发平台

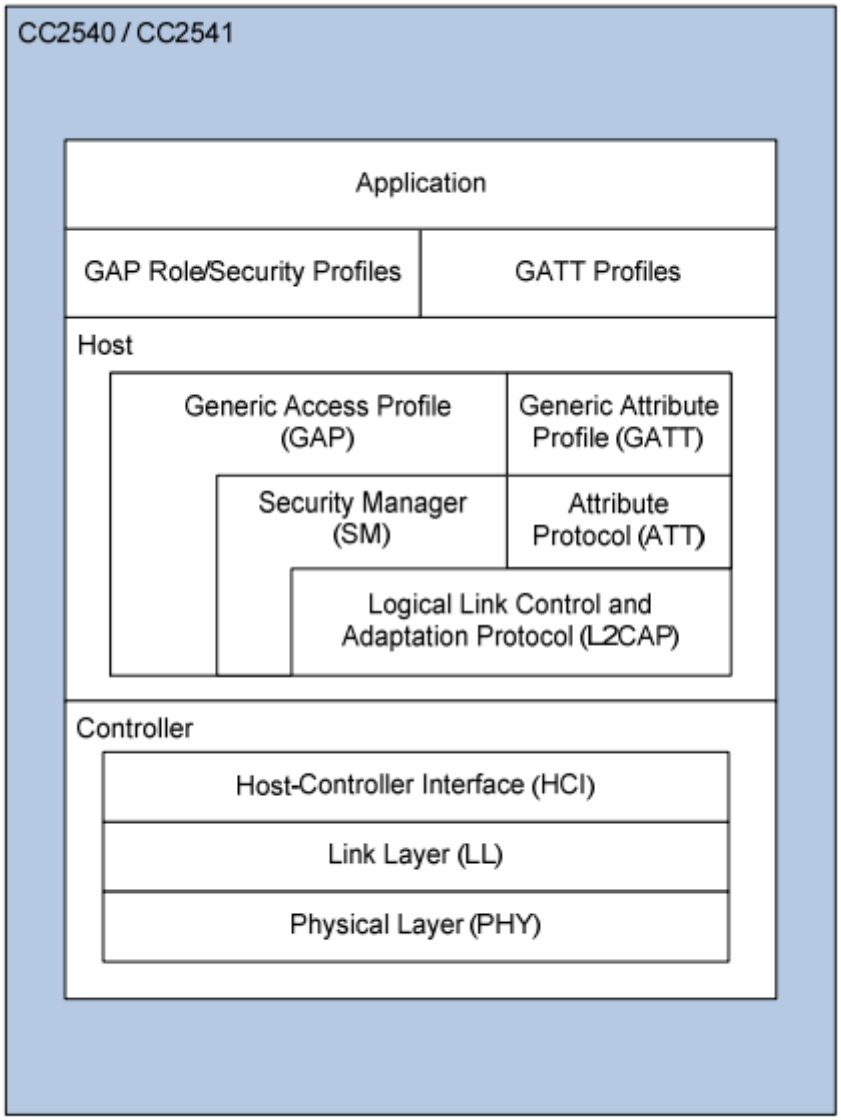
TI 免税版的 BLE 软件开发套件是一套完整的，用来开发 single-mode BLE 应用程序的软件开发平台。该 BLE 基础 TI 的 SoC 蓝牙芯片：CC2540/41，CC2540/41 集成 RF 收发器，处理器，集成的 256K 的内部 flash 和 8K 的 RAM 组成，并且还有一些列的外设

CC2540 与 CC2541 的区别是 CC2540 集成 USB，CC2541 集成 I2C。

2.1 Configurations（配置）

TI BLE 软件平台支持两种不同的协议栈/应用程序配置

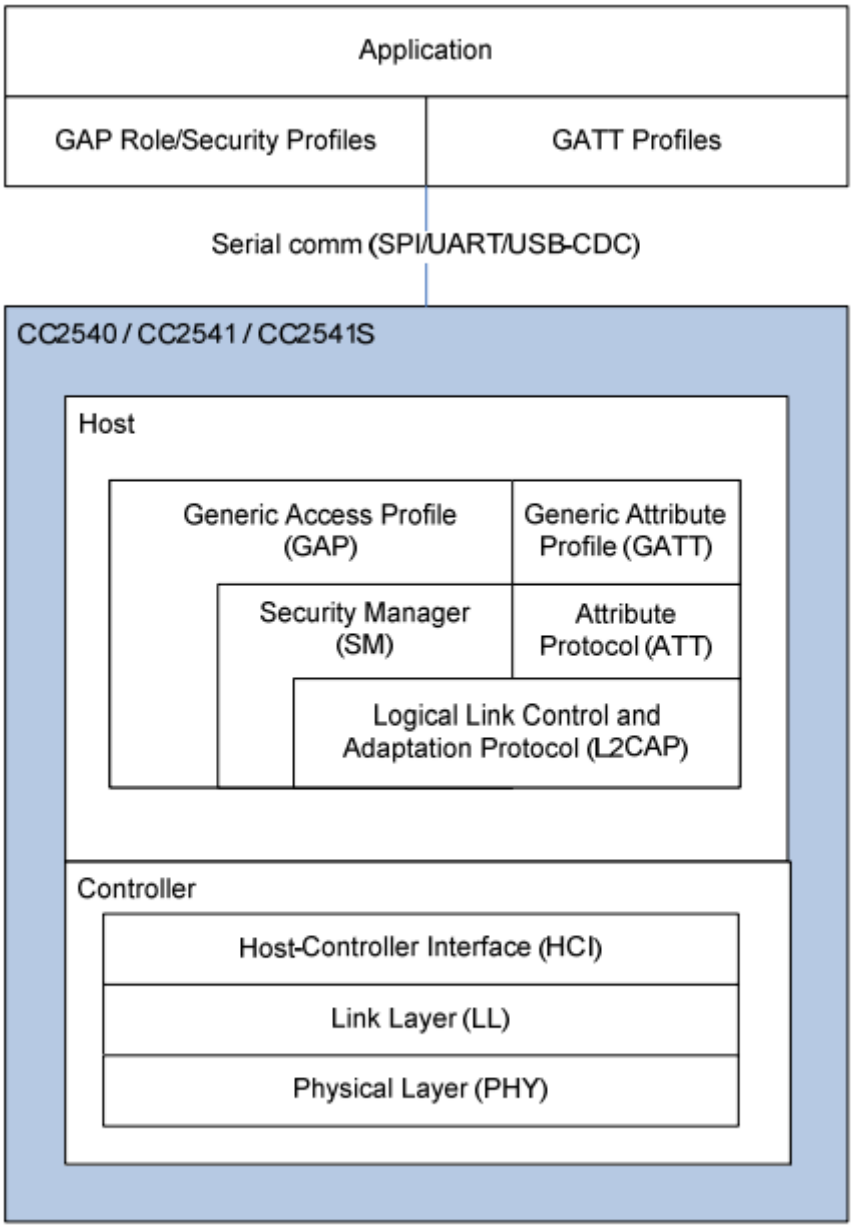
- Single-Device：controller（控制器），host（主机），profiles（标准配置）全部集成在 CC2540/41 单 SoC 芯片中。这也是使用 CC2540/41 最简单和最通用的配置。TI BLE 协议栈中的例程，绝大多数也是使用该配置。具有更高的效率和更低的功耗表现。工程中的 SampleBLEPeripheral 和 SimpleBleCentral 两示例程序为 single-device 的典型应用。



Single-Device Configuration

- Network Processor（网络处理器）：主机和控制器在 CC2540 内部实现，但是 profiles 和应用程序在外面实现，通过 SPI 或者 UART 接口通信。这种方案也是非常有用的，当使用外部处理器或者直接连接 PC，在这种情况下，应用程序可以在外部实现，而 CC2540 任然运行 BLE 协议栈。TI BLE 协

议栈中的 HostTestRelease 示例程序即为此方案中的 CC2540/41 端的 BLE 协议栈程序。



Network Processor Configuration

2.2 Projects（示例代码工程）

SimpleBLEPeripheral 工程由一些示例代码组成的一个非常简单的 single-device 配置，该示例可以用来开发 slave(从设备)应用程序。运行此程序的设备可以被 iphone 搜索到。

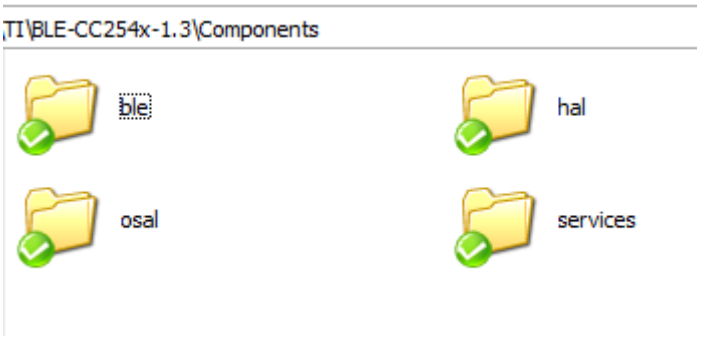
SimpleBLECentral 工程也是类似的，但是他实现的是 single-device 主机端的配置，用来开发 host(主机)程序。

HostTestRelease 工程用来创建 CC2540/41 的 BLE 网络处理器，包含的配置包括 master（主机）和 slave（从机）角色。

其他示例工程不在此介绍，请参见相应的说明书。

3 TI BLE 软件预览

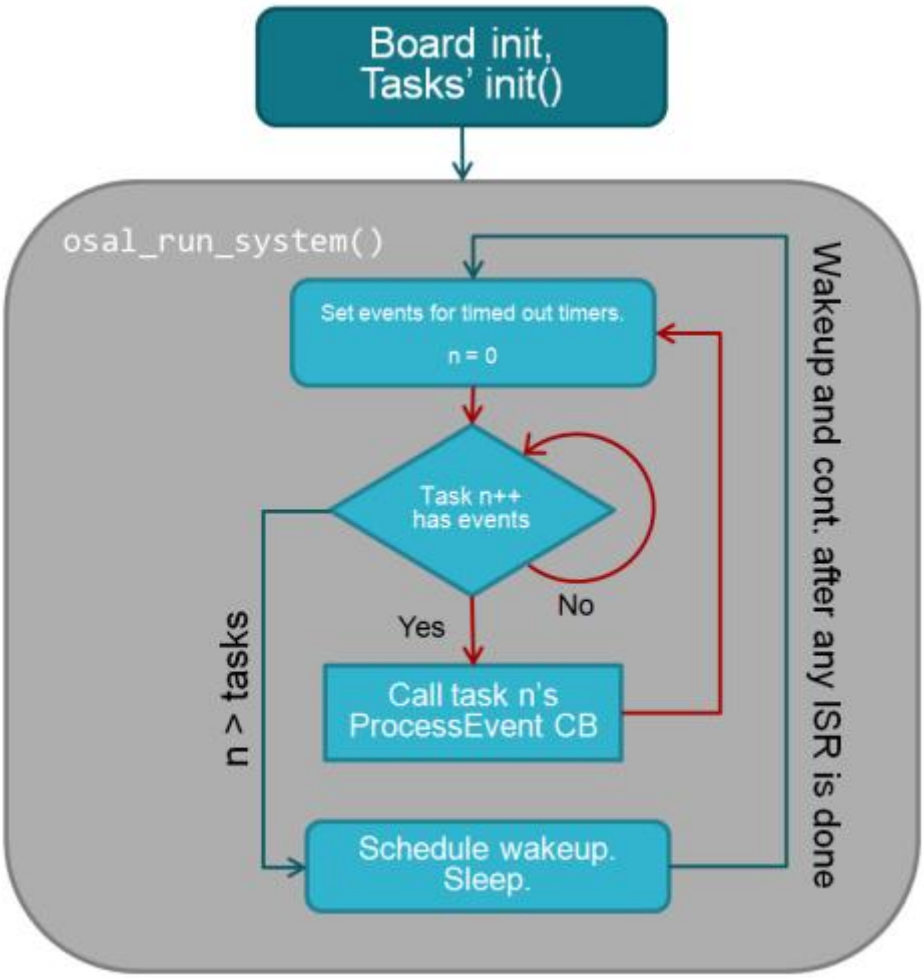
BLE 软件开发和 zigbee 非常类似，他们使用相同的 OSAL 和 HAL 层，不同的是协议栈部分。BLE 软件分部如下图：



ble：BLE 协议栈头文件，注意，只是头文件，TI 的 BLE 协议栈只提供封装好的 lib 库，不提供源代码。

hal：硬件抽象层，与 z-stack 类似，全部是基础 CC2540/41 的硬件驱动程序

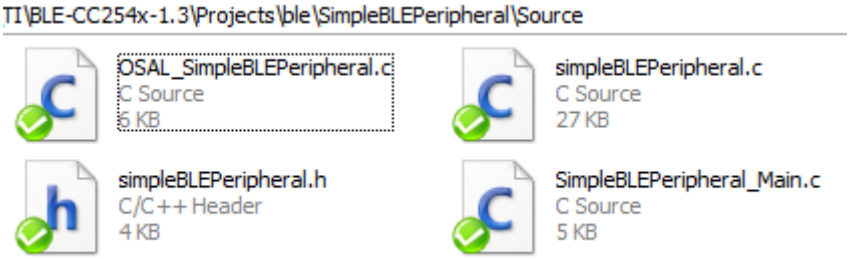
osal：系统抽象层，与 zigbee 的相同，是整个软件运行的基础，关于 osal 的详情，参见《osal 编程指南》，下图是 OSAL 的任务调度的基本框架，osal 的能力非常强大，掌握 osal 的编程方法将大大促进协议栈应用程序的理解和开发。



OSAL task loop

3.1 OSAL（系统抽象层）

下面简要描述 BLE 示例程序中的 OSAL 的操作，以 SimpleBLEPeripheral 示例代码为例。该示例程序应用程序部分有四个代码源文件，分别是，如图：



OSAL_SimpleBLEPeripheral.c：OSAL 任务回调函数数据的定义，和任务初始化函数的定义。在该文件中，作为 OSAL 的外部全局变量，被 OSAL 的任务调度代码使用。

```
00088: const pTaskEventHandlerFn tasksArr[] =
00089: {
00090:   LL_ProcessEvent,
00091:   Hal_ProcessEvent,
00092:   HCI_ProcessEvent,
00093: #if defined ( OSAL_CBTIMER_NUM_TASKS )
00094:   OSAL_CBTIMER_PROCESS_EVENT( osal_CbT
00095: #endif
00096:   L2CAP_ProcessEvent,
00097:   GAP_ProcessEvent,
00098:   GATT_ProcessEvent,
00099:   SM_ProcessEvent,
00100:   GAPRole_ProcessEvent,
00101:   GAPBondMgr_ProcessEvent,
00102:   GATTServApp_ProcessEvent,
00103:   SimpleBLEPeripheral_ProcessEvent
00104: };
```

simpleBLEPeripheral.c：BLE 应用程序的实现代码。该源文件中调用一系列的 BLE API 函数，来完成复杂的蓝牙协议。

```
00278: void SimpleBLEPeripheral_Init( uint8 task_id )
00279: {
00280:   simpleBLEPeripheral_TaskID = task_id;
00281:
00282:   // Setup the GAP Peripheral Role Profile
00283:   {
00284:
00285:     #if defined( CC2540_MINIDK )
00286:       // For the CC2540DK-MINI keyfob, device doesn't star
00287:       uint8 initial_advertising_enable = FALSE;
00288:     #else
00289:       // For other hardware platforms, device starts adver
00290:       uint8 initial_advertising_enable = TRUE;
00291:     #endif
00292:
00293:     // By setting this to zero, the device will go into th
00294:     // being discoverable for 30.72 second, and will not b
00295:     // until the enabler is set back to TRUE
00296:     uint16 gapRole_AdvertOffTime = 0;
00297:
00298:     uint8 enable_update_request = DEFAULT_ENABLE_UPDATE_RE
00299:     uint16 desired_min_interval = DEFAULT_DESIRED_MIN_CONN
00300:     uint16 desired_max_interval = DEFAULT_DESIRED_MAX_CONN
00301:     uint16 desired_slave_latency = DEFAULT_DESIRED_SLAVE_L
00302:     uint16 desired_conn_timeout = DEFAULT_DESIRED_CONN_TIM
00303:
00304:     // Set the GAP Role Parameters
00305:     GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof(
00306:     GAPRole_SetParameter( GAPROLE_ADVERT_OFF_TIME, sizeof(
00307:
00308:     GAPRole_SetParameter( GAPROLE_SCAN_RSP_DATA, sizeof (
00309:     GAPRole_SetParameter( GAPROLE_ADVERT_DATA, sizeof( adv
00310:
00311:     GAPRole_SetParameter( GAPROLE_PARAM_UPDATE_ENABLE, siz
00312:     GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizec
```

simpleBLEPeripheral.h：一些常量的定义。和蓝牙应用程序任务初始化和任务回到函数的声明。

```
00057: // Simple BLE Peripheral Task Events
00058: #define SBP_START_DEVICE_EVT                              0x0001
00059: #define SBP_PERIODIC_EVT                                  0x0002
00060: #define SBP_ADV_IN_CONNECTION_EVT                         0x0004
00061:
00062: /*********************************************************************
00063:  * MACROS
00064:  */
00065:
00066: /*********************************************************************
00067:  * FUNCTIONS
00068:  */
00069:
00070: /*
00071:  * Task Initialization for the BLE Application
00072:  */
00073: extern void SimpleBLEPeripheral_Init( uint8 task_id );
00074:
00075: /*
00076:  * Task Event Processor for the BLE Application
00077:  */
00078: extern uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
```

SimpleBLEPeripheral_Main.c：OSAL 的 main 函数，在 main 函数中，初始化必要的硬件、协议栈，最后进入任务调度循环。

```
00074: int main(void)
00075: {
00076:   /* Initialize hardware */
00077:   HAL_BOARD_INIT();
00078:
00079:   // Initialize board I/O
00080:   InitBoard( OB_COLD );
00081:
00082:   /* Initialze the HAL driver */
00083:   HalDriverInit();
00084:
00085:   /* Initialize NV system */
00086:   osal_snv_init();
00087:
00088:   /* Initialize LL */
00089:
00090:   /* Initialize the operating system */
00091:   osal_init_system();
00092:
00093:   /* Enable interrupts */
00094:   HAL_ENABLE_INTERRUPTS();
00095:
00096:   // Final board initialization
00097:   InitBoard( OB_READY );
00098:
00099:   #if defined ( POWER_SAVING )
00100:     osal_pwrmgr_device( PWRMGR_BATTERY );
00101:   #endif
00102:
00103:   /* Start OSAL */
00104:   osal_start_system(); // No Return from here
00105:
00106:   return 0;
00107: } ? end main ?
```

以上是对 BLE 协议栈应用程序源码文件和文件结构的简单介绍，下面将描述 OSAL 任务的执行流程。

3.1.1 Task Initialization（任务初始化）

任务初始化，是在 OSAL 任务函数调度循环之前执行的，在上图中的 osal_init_system()函数中。

```
01041: uint8 osal_init_system( void )
01042: {
01043:   // Initialize the Memory Allocation System
01044:   osal_mem_init();
01045:
01046:   // Initialize the message queue
01047:   osal_qHead = NULL;
01048:
01049:   // Initialize the timers
01050:   osalTimerInit();
01051:
01052:   // Initialize the Power Management System
01053:   osal_pwrmgr_init();
01054:
01055:   // Initialize the system tasks.
01056:   osalInitTasks();
01057:
01058:   // Setup efficient search for the first free
01059:   osal_mem_kick();
01060:
01061:   return ( SUCCESS );
01062: } ? end osal_init_system ?
```

任务初始化的函数定义在文件 OSAL_SimpleBLEPeripheral.c 中，该函数包含该应用程序用到的所有任务的任务初始化函数的调用。不同协议层通常使用单独的任务函数，因此就对应了许多任务初始化函数。每一个任务初始化函数调用时，一个 8bit 的“tast id”（任务 id），被传入到任务中保存，“tast id”数值越大，该任务的优先级越低。例如 osalInitTasks()函数中

```
00122: void osalInitTasks( void )
00123: {
00124:   uint8 taskID = 0;
00125:
00126:   tasksEvents = (uint16 *)osal_mem_alloc(
00127:   osal_memset( tasksEvents, 0, (sizeof( u
00128:
00129:   /* LL Task */
00130:   LL_Init( taskID++ );
00131:
00132:   /* Hal Task */
00133:   Hal_Init( taskID++ );
00134:
00135:   /* HCI Task */
00136:   HCI_Init( taskID++ );
00137:
00138: #if defined ( OSAL_CBTIMER_NUM_TASKS )
00139:   /* Callback Timer Tasks */
00140:   osal_CbTimerInit( taskID );
00141:   taskID += OSAL_CBTIMER_NUM_TASKS;
00142: #endif
00143:
00144:   /* L2CAP Task */
00145:   L2CAP_Init( taskID++ );
00146:
00147:   /* GAP Task */
00148:   GAP_Init( taskID++ );
00149:
00150:   /* GATT Task */
00151:   GATT_Init( taskID++ );
00152:
00153:   /* SM Task */
00154:   SM_Init( taskID++ );
00155:
00156:   /* Profiles */
00157:   GAPRole_Init( taskID++ );
00158:   GAPBondMgr_Init( taskID++ );
00159:
00160:   GATTServApp_Init( taskID++ );
00161:
00162:   /* Application */
00163:   SimpleBLEPeripheral_Init( taskID );
00164: } /* end osalInitTasks */
```

LL 层任务函数的优先级最高，而应用程序任务函数的优先级最低。

3.1.2 Task Event（任务事件）和 Event Processing（事件处理）

OSAL 为每一个任务函数分配了一个 16 位的 EVENT 事件，每一位代表一个事件，其中最高位代表的事件为 SYS_EVENT_MSG，这个事件被 OSAL 系统保留，其他的 15 位可以由用户定义，OSAL 主循环里每次都会检查每个任务函数的是否有事件发生（事件置位），如果有时间发生，将通过 taskid 来调用发生事件的任务函数，并将发生的事件传递到该函数中去，由任务函数相应对应的事件。

```
void osal_run_system( void )
{
  uint8 idx = 0;

#ifndef HAL_BOARD_CC2538
  osalTimeUpdate();
#endif

  Hal_ProcessPoll();

  do {
    if (tasksEvents[idx])  // Task is highest priority that i
    {
      break;
    }
  } while (++idx < tasksCnt);

  if (idx < tasksCnt)
  {
    uint16 events;
    halIntState_t intState;

    HAL_ENTER_CRITICAL_SECTION(intState);
    events = tasksEvents[idx];
    tasksEvents[idx] = 0;  // Clear the Events for this task.
    HAL_EXIT_CRITICAL_SECTION(intState);

    activeTaskID = idx;
    events = (tasksArr[idx])( idx, events );
    activeTaskID = TASK_NO_TASK;

    HAL_ENTER_CRITICAL_SECTION(intState);
    tasksEvents[idx] |= events;  // Add back unprocessed eve
    HAL_EXIT_CRITICAL_SECTION(intState);
```

如果有事件发生，则

将运行下面的代码

通过id调用任务函数

如，SimpleBLEPeripheral 从机程序相应 SYS_EVENT_MSG 的代码，如下图：

```
uint16 SimpleBLEPeripheral_ProcessEvent( uint8 task_id,
{

  VOID task_id; // OSAL required parameter that isn't used in this funct

  if ( events & SYS_EVENT_MSG )
  {
    uint8 *pMsg;

    if ( (pMsg = osal_msg_receive( simpleBLEPeripheral_TaskID )) )
    {
      simpleBLEPeripheral_ProcessOSALMsg( (osal_event_hdr_t *)pMs

      // Release the OSAL message
      VOID osal_msg_deallocate( pMsg );
    }

    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }

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

系统消息事件

其他事件

SimpleBLEPeripheral 应用程序在 simpleBLEPeripheral.h 头文件中定义了一个用户事件：SBP_START_DEVICE_EVT(0x0001)，上图中的其他事件，这个事件表示初始化已经成功，可以开始应用程序，用户不可以定义值为 0x8000 的事件，因为 SYS_EVENT_MSG 已经占用该位，每当任务之间有 msg 传递时就会触发 SYS_EVENT_MSG 事件，告知应用程序开始接收消息。

在 SimpleBLEPeripheral 应用程序，开发者需要编程的任务函数为 SimpleBLEPeripheral_ProcessEvent，任务函数每次处理完事件后，需手动将已处理的事件标志位清零，否则将一直运行，直到标志位被清零。

在 osal 的各层中，可以为其他任务函数设置事件，当然也包括自己，在 osal 层中提供了这样的设置事件的函数：osal_set_event()，（在 OSAL.h），该函数会直接调度一个事件，如果需要一段时间后产生某个事件，则使用 osal_start_timerEx()，(在 OSAL_Timers.h 中)，传给该函数的第一个参数为 TaskID，第二个参数为需要启动的事件，第三个参数为多久以后启动参数二中的事件。

3.1.3 Heap Manager（堆空间管理）

OSAL 提供了基础的内存管理函数：osal_mem_alloc，（类似标准 c 中的 malloc 动态内存分配），给该函数传递需要分配的字节数，然后返回 void 类型的指针，如果没有足够的空间，则返回 NULL。有内存分配，当然也就有内存回收 osal_mem_free()，被 free 后的内存可以重新被 alloc 使用。（该技术使用静态的内存池，即一长串的静态数组作为存储空间来分配）。

3.1.4 OSAL Messages（消息）

OSAL 为不同任务函数提供了一种可以携带任意数据的通信机制，这就是 msg，发送消息前用 osal_msg_allocate()函数，为消息分配内存空间，然后填充合适的数据，然后调用 osal_msg_send 将消息发送到指定的任务函数中区，OSAL 会告知接收端的任务函数，有新消息来，告知的方式就是设置 SYS_EVENT_MSG 事件。然后接收端就可以使用 osal_msg_reveive()将消息接收过来了，消息接收完成后续使用 osal_msg_deallocate()函数来回收当前消息所占用的内存。OSAL 推荐在任务函数中使用独立的消息接收函数来处理消息，例如 simpleBLEPeripheral_ProcessOSALMsg()函数。

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

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

3.2 Hardware Abstraction Layer（HAL，硬件抽象层）

CC2540/41 的硬件抽象层，为应用程序和协议栈提供上层的提供统一的硬件接口，屏蔽了具体的硬件处理细节，当开发者设计了新的电路，需要相应的修改 hal 层。现有的 hal 已经集成了一些列的现有的硬件平台，如下：

- SmartRF05EB+CC2540EM
- SmartRF05EB+CC2541EM
- CC2540 Keyfob
- CC2541 Keyfob

- CC2541 SensorTag
- CC2540 USB Dongle

详细的 hal 函数说明，请参见文档《hal api》

3.3 BLE Protocol Stack（TI BLE 协议栈）

整个 BLE 协议栈以 lib 库的形式提供，ti 描述说是由于政策的原因，主要还是处于知识产权的保护。学习 TI 蓝牙 4.0 的开发的重点就是学习 BLE 协议栈,BLE 协议栈中的重点则是与具体蓝牙应用相关的 GAP 和 GATT，这两层直接与应用程序打交道，所以也必须理解，而上面所述的 OSAL 编程是学习 BLE 协议栈的基础。下面将详细的描述 GAP 和 GATT。对于新的知识，很难一次、通过一个文档就学会，请读者多看多多分析。

3.3.1 Generic Access Profile (GAP)

暂且将 GAP 翻译成“通用访问规范”，Profile 可以理解为共同约定的配置或规范，只要你我遵守同一个 profile，就可以相互打招呼。

该 Profile 保证不同的 Bluetooth 产品可以互相发现对方并建立连接，GAP 定义了蓝牙设备如何发现和建立与其他设备的安全或不安全的连接（具体包括，设备发现，创建连接，终止连接，安全结构的初始化，和设备配置）。

GAP 层总是工作在一下角色中的一种：

- Broadcaster 广播员，表明我存在，但你只能看到我，不可以连接我
- Observer 观察者，看看谁在，我只是观察，不连接
- Peripheral 外设（从机），大多数的蓝牙设备都工作在这个角色，我存在，并且谁要我，我就跟谁走。
- Central 中心（主机），包括智能手机工作在此角色，看看谁在，并且愿意跟我走的就带他走，工作在单层或多层的连接。

BLE 规定允许同时可具备多种角色，但是 TI 的 BLE 协议栈中的示例程序默认均只支持 Peripheral 角色。

典型的低功耗蓝牙系统中同时包含外设和主机，两者的连接过程如下

Peripheral 外设角色向外发送自己的信息（设备地址，名字等）让别人知道我可以连接，Central 主机角色一旦接收到该广播，将想 Peripheral 外设角色发送“Scan Request”扫描请求，随之外设角色会以“scan response”扫描相应来回应主机角色的扫描请求，以上就是设备发现的处理过程，现在主机角色已经知道外设此时可连接，主机可以向外设发送建立连接的请求，主机的连接请求包含一系列的连接参数。如下：

Connection Interval 通信间隙：蓝牙通信是间断的调频的，每次连接都可能选择不同的子频带，调频的好处是避免频道拥堵，间断连接的好处是节省功耗，通信间隙就是指两次连接之间的时间间隔，这个间隔以 1.25ms 为基本单位，最小 6 单位 7.5ms 最大 3200 单位 4.0s，间隙越小通信月及时，间隙越大功耗月底。

Slave Latency 从机延时：外设与主机建立连接以后，没事的时候主机会定期发送问候信息到外设，但是外设可以对这些问候消息不予理会，继续休眠，以保证节能，主机的每次问候都是一次连接事件，从机可以忽略连接事件，最大忽略的事件数不超过 499 个，另外时间最大不超过 32 秒。

Supervision Timeout 监管超时：这是两个成功的连接事件的最大时间间隔，如果超过了这个时间，设备会认为连接已丢失，然后返回未连接状态，该参数的单位 10ms，监管超时可以设置 100ms 到 32 秒之间，该超时设置的时间必须大于 effective connection interval，该值可以使用下列公式计算

Effective Connection Interval = (Connection Interval) * (1 + (Slave Latency))

如下面的例子：

Connection Interval：80（100ms）

Slave Latency：4

Effective Connection interval：100ms*（1+4）=500ms

所以，如果从机没有数据向主机发送，从机则每隔 500ms 传输一次 connection event。

在大多数的应用中，从机会忽略一个最大计数的 connection event，因此，使用 effective connection interval 是非常有用的用来选择连接参数，

以下是一些参数设置

减小 connection interval 时：

- 增加双方的功耗
- 增强处理能力
- 减少每次每次发送的数据量

增加 connection interval 时：

- 与上述相反

减小 slave latency 时：

- 增加从机的功耗