TI-RTOS概述

TI-RTOS是CC2640R2F设备上低功耗蓝牙项目的运行环境。TI-RTOS内核是传统SYS/BIOS内核的定制版本，是一个具有驱动程序，同步和调度工具的实时抢占式多线程操作系统。

线程模块

TI-RTOS内核管理线程执行的四个不同的任务级别，如图21所示。线程模块列表如下图所示，按照优先级降序排列。

- 硬件中断
- 软件中断
- 任务
- 后台空闲功能的空闲任务

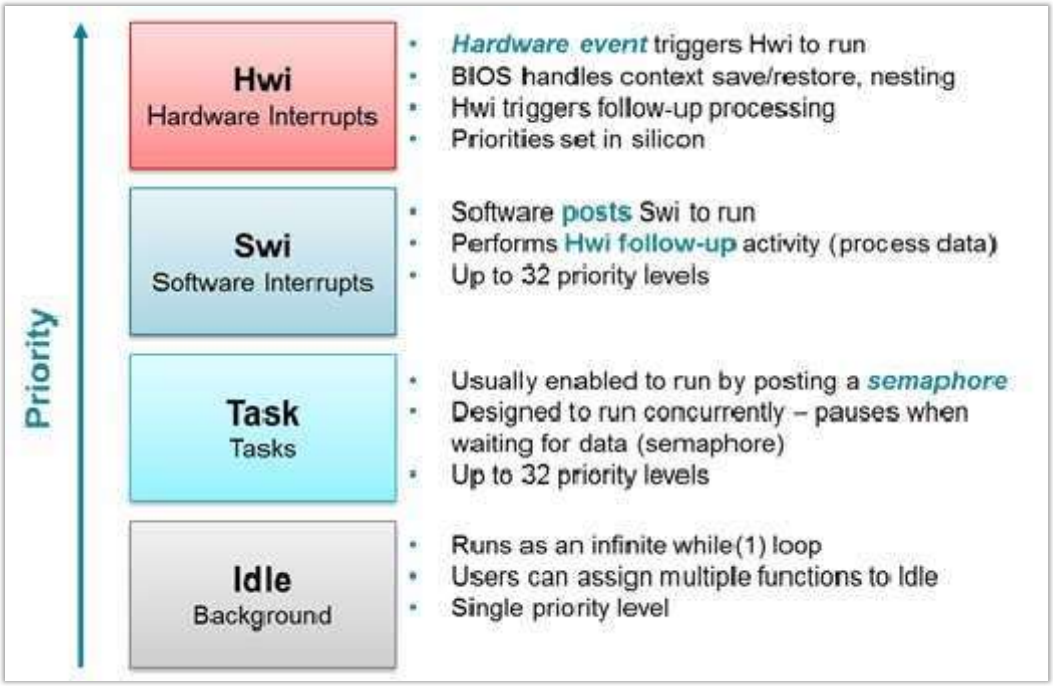

图21. TI-RTOS执行线程

这一节将要介绍四个执行线程以及整个TI-RTOS中用于消息传递和同步的各种结构。在大多数情况下，TI-RTOS函数在util.c（Util）中已被抽象为更高级别的函数。较低级的TI-RTOS函数在**TI-RTOS Kernel API Guide**中有描述，你可以在TI-RTOS内核用户指南中查看。本文档还定义了TI-RTOS包含的软件包和模块。

硬件中断（Hwi）

Hwi线程（也称为中断服务程序或ISR）是TI-RTOS应用程序中具有最高优先级的线程。Hwi线程用于执行有严格时间限制的关键任务。它们被触发以响应在实时环境中发生的外部异步事件（中断）。Hwi线程总是运行至完成，但是如果有其他的Hwi中断使能，它也可以暂时地被其他中断触发的Hwi线程抢占，这就是所谓的中断嵌套。有关中断嵌套，向导和功能的具体信息，可以在CC26XX技术参考手册中查看。

一般来说中断服务程序运行时间较短，不影响硬实时系统的要求。另外，由于Hwis总是运行至完成，所以在在其上下文中不会调用阻塞API。

中断的TI-RTOS驱动程序将初始化分配的外设所需的中断。有关详细信息，请参阅外设驱动。

> 注意:
> 外部资源（External Resources）提供了使用GPIO和Hwis的示例。虽然SDK包含一个外设驱动程序库抽象了对硬件寄存器的访问，但建议使用线程>安全的TI-RTOS驱动程序，如外设驱动中所述。

CC2640R2F的Hwi模块还支持**零延迟中断**。这些中断不通过TI-RTOS Hwi调度程序，因此比标准中断更灵敏，但是该功能禁止其中断服务程序直接调用任何TI-RTOS内核API。ISR要保护自己的上下文防止它干扰内核的调度程序。

为了能让低功耗蓝牙协议栈满足RF严格的时序要求，所有应用程序定义的Hwis都要以最低优先级执行。TI向系统添加新的Hwis时，建议不要修改默认的Hwi优先级。为了不破坏低功耗蓝牙协议栈中TI-RTOS的严格时序，应用程序中定义了临界段代码。执行临界段代码时中断会被关闭，在临界段代码执行完毕之后才会重新启用中断。

软件中断（Swi）

软件中断线程（Swis）是在Hwi线程和任务线程之间提供的一个额外的优先级。与Hwis由硬件中断触发不同，Swis是通过在程序编写过程中调用某些Swi模块的API来触发中断。由于时间限制，Swis中断服务程序不能作为任务来运行，其截止时间不如硬件中断服务程序那么严格。Swi也总是运行至完成，它允许Hwis将不太重要的中断处理放到较低优先级的线程来处理，从而最小化CPU在中断服务程序中花费的时间。Swis需要足够的空间来保存每个Swi中断优先级的上下文，它的每个线程都使用单独的堆栈。

与Hwis类似，Swis需要保持简短，它不应该包含任何阻塞API的调用。这保证了诸如无线协议栈等高优先级任务能根据需要执行。在Swis中建议发布某些TI-RTOS同步对象，然后把具体处理放在任务上下文中。图22说明了这种用例。

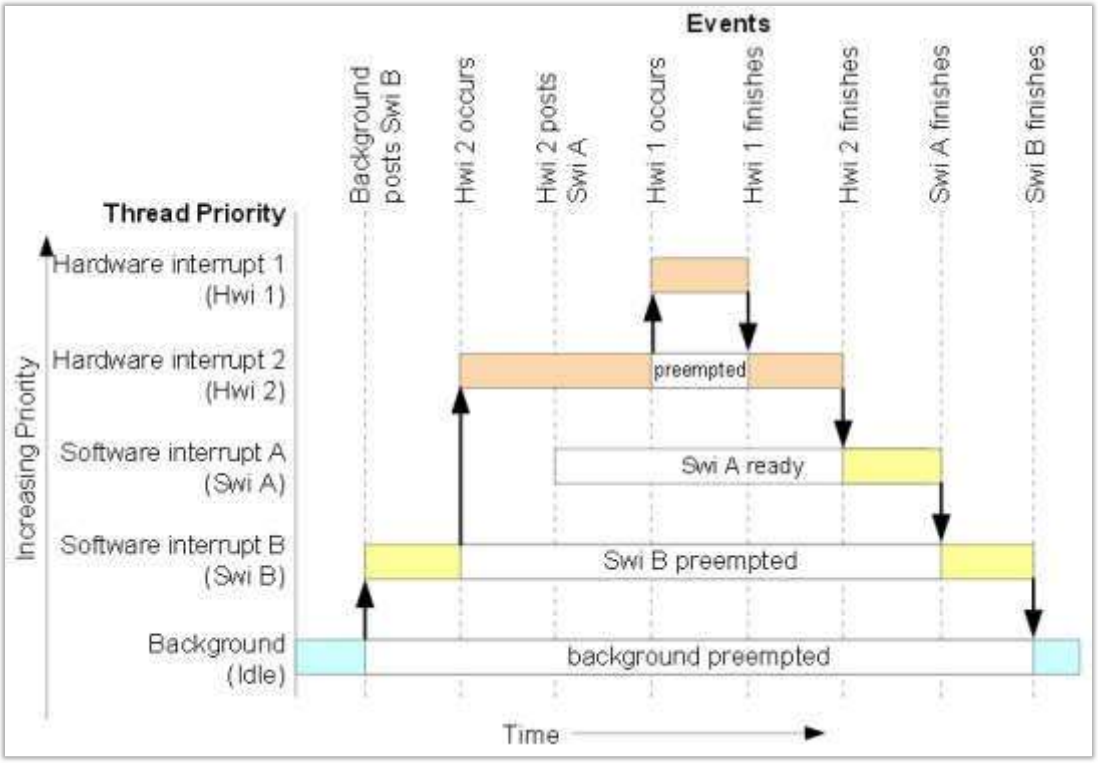

图22. 抢占情景

Swi上下文中常常会由时钟模块调用，对于Swi服务函数，不调用阻塞API，保证较短的执行时间是很重要的。

任务

任务线程的优先级高于空闲任务线程，低于软件中断。任务与软件中断的不同之处在于，任务可以在执行期间等待（阻塞），直到有必要的资源可用。每个任务线程都有一个独立的堆栈。TI-RTOS提供了可用于任务间通信和同步的多种机制。这些包括信号量，事件，消息队列和邮箱。
有关详细信息，可以在本文后面的**任务**中查看。

空闲任务

空闲任务线程在TI-RTOS应用程序中优先级最低，它会执行一个空闲循环。在主函数返回之后，TI-RTOS应用程序会调用每个TI-RTOS模块的启动程序，然后进入空闲循环。每个线程在被再次调用之前都必须等待所有其他线程执行完成。空闲循环在没有更高优先级的任务需要执行的时候会一直执行。只有没有严格截止期限的函数才能在空闲循环中执行。

对于CC2640R2F设备，空闲任务支持电源策略管理器设置为允许的最低功率节省模式。

内核配置

TI-RTOS应用程序可以使用工程中的`.cfg`文件来配置TI-RTOS内核。在**IAR**和**CCS**工程中，配置文件在应用程序项目工作区中的`TOOLS`文件夹下。

该配置通过选择性地包括或使用可用于内核的RTSC模块来实现。`.cfg`中通过调用`xdc.useModule()`函数来设置TI-RTOS内核用户指南中定义的各种选项来启用一个模块。

> 注意:
> BLE5-Stack中的项目（如`simple_peripheral`）通常会包含一个app_ble.cfg配置文件。

可以在`.cfg`文件中配置的一些选项（但不限于这些）：

- 启动选项
- Hwi，Swi和任务优先级的数量
- 异常和错误处理
- 系统刻度的持续时间（TI-RTOS内核中最基本的时间单位）。
- 定义应用程序的入口点和中断向量
- TI-RTOS堆和堆栈（不要与其他堆管理器混淆!）
- 包括预编译的内核和TI-RTOS驱动程序库
- 系统配置（for `System_printf()`）

一旦`.cfg`文件发生变动时，您需要重新运行XDCTools的`configuro`工具。在IAR和CCS提供的示例中这一步作为预构建步骤已经为您提供。

> 注意:
> `.cfg`的名称并不重要。但是项目只能包含一个`.cfg`文件。

对于CC2640R2F，TI-RTOS内核存储在ROM中。通常为了节省Flash的访问足迹，`.cfg`也会包含在内核的ROM模块中，如清单1所示。

清单1. 如何把TI-RTOS内核包含到ROM中

```
/ * ================ ROM configuration================ * /
/ *
*To use BIOS in flash, comment out the code block below.
* /
if （typeof NO_ROM == 'undefined' ||（typeof NO_ROM ! = 'undefined' && NO_ROM == 0 ））
{
        var ROM = xdc.useModule（'ti.sysbios.rom.ROM'）;
        if（program.cpu.deviceName .match（/CC26/））{
                ROM.romName = ROM.CC2640R2F;
        }
        else if（Program.cpu.deviceName.match（/CC13/））{
                ROM.romName = ROM.CC1350;
        }
}
```

ROM中的TI-RTOS内核针对性能进行了优化。如果您的应用程序需要额外的工具（通常用于调试），则必须将TI-RTOS内核包含在Flash中，这将增加Flash消耗。下面显示的是在`ROM`中使用TI-RTOS内核的简要列表。

- `BIOS.assertsEnabled`必须设置为`false`
- `BIOS.logsEnabled`必须设置为`false`
- `BIOS.taskEnabled`必须设置为`true`
- `BIOS.swiEnabled`必须设置为`true`
- `BIOS.runtimeCreatesEnabled` 必须设置为`true`
- BIOS必须使用该`ti.sysbios.gates.GateMutex`模块
- `Clock.tickSource`必须设置为`Clock.TickSource_TIMER`
- `Semaphore.supportsPriority`一定是`false`
- Swi，Task和Hwi hooks不容许
- Swi，Task和Hwi name instances不容许
- 任务堆栈检查被禁用
- `Hwi.disablePriority`必须设置为`0x20`
- `Hwi.dispatcherAutoNestingSupport`必须设置为`true`
- 默认的Heap instance必须设置为`ti.sysbios.heaps.HeapMem`管理者

有关上述列表外的其他文档，可以在TI-RTOS内核用户指南中查看。

创造与构建

大多数TI-RTOS模块通常都有`_create()`和`_construct()`APIs用来初始化最初的例程。这两个API之间运行时的主要差异是内存分配和错误处理。
在初始化之前，**创建**API会从默认的TI-RTOS堆执行内存分配。因此，应用程序必须在继续之前检查有效句柄的返回值。

清单2. 创建一个信号量

```
1 Semaphore_Handle sem;
2 Semaphore_Params semParams;
3 Semaphore_Params_init（&semParams）;
4 sem = Semaphore_create（0，&semParams，NULL）;/*Memory allocated in here*/
5 if (sem == NULL) /* Check if the handle is valid */
6 {
7       System.abort("Semaphore could not be created");
```

```
 8 }
 9
10
```

构造API提供了一个用于存储实例变量的数据结构。由于内存已被预先分配给实例，构建后不再需要进行错误检查。

清单3. 构造一个信号量

```
1 Semaphore_Handle  SEM ;
2 Semaphore_Params  semParams ;
3 Semaphore_Struct structSem; /* Memory allocated at build time */
4 Semaphore_Params_init(&semParams);
5 Semaphore_construct(&structSem, 0, &semParams);
6 /* It's optional to store the handle */
7 sem = Semaphore_handle(&structSem);
8
9
```

线程同步

TI-RTOS内核提供了几个诸如信号量，事件和队列用于同步任务的模块。以下部分讨论这些常见的TI-RTOS同步模块。

信号量

信号量通常用于TI-RTOS应用中的任务同步和互斥。图23.显示了信号量的功能。信号量可以分为计数信号量或二进制信号量。程序通过`Semaphore_post()`来释放信号量，计数信号量会记录跟踪信号量发布的次数。当一组资源在任务之间共享时，信号量很有用。在使用这些资源之前，任务会调用`Semaphore_pend()`来查看资源是否可用，只有共享资源被释放出来之后处于等待状态的任务得到该资源才能执行。二进制信号量只能有两种状态：可用（count = 1）和不可用（count = 0）。二进制信号量可用于在任务之间共享一个资源，或者用于基本信令机制，可以多次发布信号量。二进制信号不跟踪计数， 他们只跟踪信号量是否已经发布。

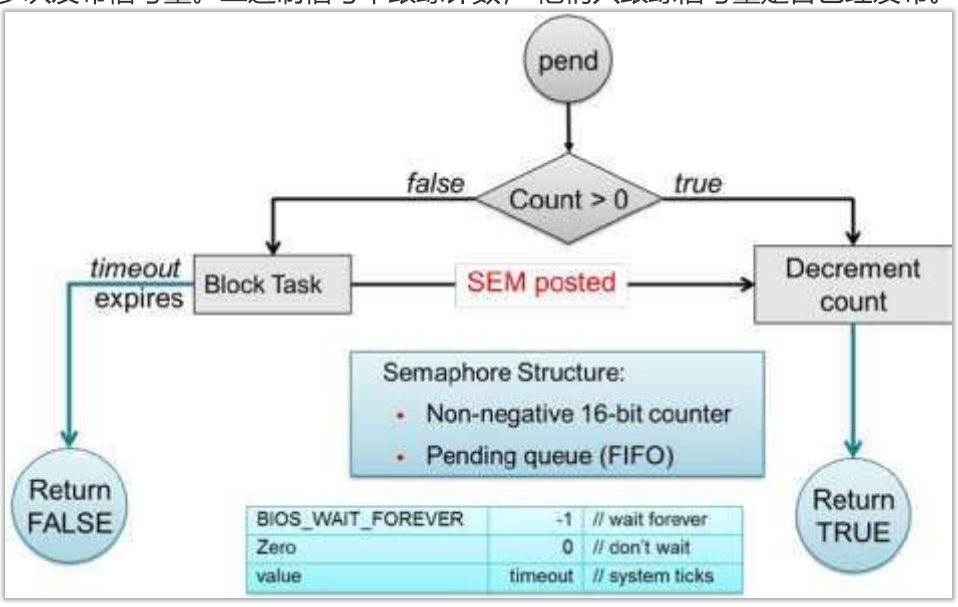

图23. 信号量功能

初始化信号量

以下代码描述了如何在TI-RTOS中初始化信号量。信号量可以创建和构造，如本文上面的**创建与构造**中所述。
有关如何创建信号量，请参见清单2。
有关如何构造信号量，请参见清单3。

等待信号量

`Semaphore_pend()`是一个阻塞函数调用。它只能在任务中调用。当任务调用此阻塞函数后将会等待信号量的释放（post），这时就绪的低优先级任务可以执行。调用`Semaphore_pend()`如果计数器为0，任务将阻塞，否则计数器会递减1，任务执行。在另一个线程调用`Semaphore_post()`释放信号量或者提供的系统滴答时钟超时之前，任务都会保持阻塞状态。通过读取其返回值`Semaphore_pend()`可以区分信号量是否发布或超时。

清单4. 等待一个信号量

```
1 bool isSuccessful;
2 uint32_t timeout = 1000 * (1000/Clock_tickPeriod);
3 /* Pend (approximately) up to 1 second */
4 isSuccessful = Semaphore_pend(sem, timeoutInTicks);
5 if (isSuccessful)
6 {
7               System_printf("Semaphore was posted");
8 }
9 else
10 {
11              System_printf("Semaphore timed out");
12 }
13
14
```

> 注意:
> 默认的TI-RTOS系统滴答时钟周期为1毫秒。在CC26xx和CC13xx设备中通过设置`.cfg`文件中的`Clock.tickPeriod = 10`，将此默认值重新配置为10微秒。
> 给定一个10微秒的系统滴答时钟，`timeout`在清单4中约为1秒。

发布信号量

信号量的发布是通过调用`Semaphore_post()`完成的。在此发布的信号量上挂起的任务将从阻塞状态转换到就绪状态。如果此时正好没有更高优先级的线程准备运行，得到该信号量的任务将会运行。如果信号量上没有挂起任务，调用Semaphore_post()将时信号量计数器加1。二进制信号量的最大计数为1。
清单5. 发布信号量

```
1 Semaphore_post （sem ）;
```

事件

信号量提供了线程之间的基本同步。有些情况下，信号量本身就足够了解什么进程需要触发。然而有些同步的特定原因也需要跨线程传递。为了实现这一点，可以使用TI-RTOS 事件（Event）模块。事件类似于信号量，每个事件实际上都包含一个信号量。使用事件的附加优点在于可以以线程安全的方式向任务通知特定事件。

初始化事件

创建和构建事件同样遵循本文上面的**创建与构建**中说明的准则。如清单6所示，是一个关于如何构造Event实例的例子。

清单6. 构造事件

```
1 Event_Handle event;
2 Event_Params eventParams;
3 Event_Struct structEvent; /* Memory allocated at build time */
4 Event_Params_init(&eventParams);
5 Event_construct(&structEvent, 0, &eventParams);
6 /* It's optional to store the handle */
7 event = Event_handle(&structEvent);
8
9
```

事件等待

类似于`Semaphore_pend()`，任务线程会在调用`Event_pend()`时阻塞， 直到事件通过一个`Event_post()`发布或者等待超时。清单7展示了一个任务等待下面显示的3个示例事件ID中的任意一个发布的代码片段。`BIOS_WAIT_FOREVER`参数设置表示不会等待超时，只要时间没发布会永远等待下去。因此， Event_pend()将在返回的位掩码值中发布一个或多个事件。

`Event_pend()`返回的每个事件会以线程安全的方式在事件实例中自动清除。因此，对于发布的事件只需保留本地副本。有关使用`Event_pend()`的详细介绍 ， 可以在TI-RTOS内核用户指南中查看。

清单7. 事件挂起

```
1 #define START_ADVERTISING_EVT         Event_Id_00
2 #define START_CONN_UPDATE_EVT         Event_Id_01
3 #define CONN_PARAM_TIMEOUT_EVT        Event_Id_02

4 void TaskFxn(..)
5 {
6     /* Local copy of events that have been posted */
7    uint32_t events;

8     while(1)
9     {
10        /* Wait for an event to be posted */
11        events = Event_pend(event,
12                            Event_Id_NONE,
13                            START_ADVERTISING_EVT |
14                            START_CONN_UPDATE_EVT |
15                            CONN_PARAM_TIMEOUT_EVT,
16                            BIOS_WAIT_FOREVER);

17        if (events & START_ADVERTISING_EVT)
18        {
19            /* Process this event */
20        }
```

```
21        if (events & START_CONN_UPDATE_EVT)
22        {
23            /* Process this event */
24        }

25        if (events & CONN_PARAM_TIMEOUT_EVT)
26        {
27            /* Process this event */
28        }
29    }
30 }
```

事件发布

事件可以从一些TI-RTOS内核任务中发布（Post），通过调用`Event_post()`就可以发布事件实体和事件ID。清单8.显示了高优先级线程（如Swi）如何发布特定事件。

清单8. 发布事件

```
1 #define START_ADVERTISING_EVT         Event_Id_00
2 #define START_CONN_UPDATE_EVT         Event_Id_01
3 #define CONN_PARAM_TIMEOUT_EVT        Event_Id_02

4 void SwiFxn(UArg arg)
5 {
6     Event_post(event, START_ADVERTISING_EVT);
7 }
8
```

队列

TI-RTOS队列是一个基于先入先出（FIFO），线程安全的单向消息传递模块。队列通常用于高优先级线程将消息传递给较低优先级的线程以进行延迟处理;因此允许低优先级任务阻塞直到需要运行。

在图24中，队列被配置为从任务A到任务B的单向通信。任务A将消息**放入**到队列中，任务B从队列**获取**消息。

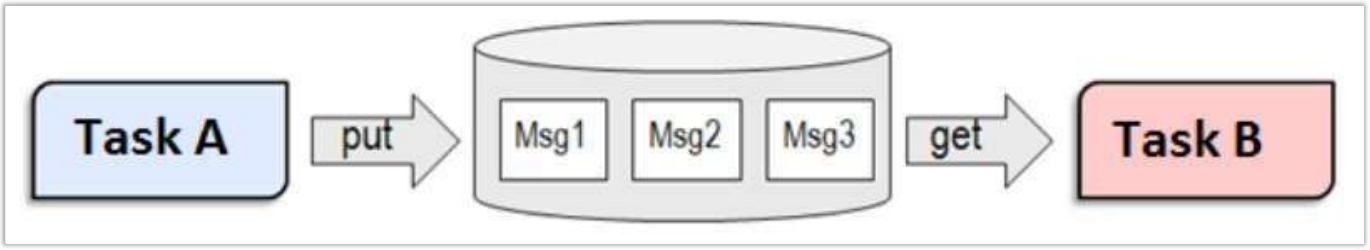

图24. 队列消息传递过程

在BLE5-Stack中，TI-RTOS队列功能在`util.c`中已经被抽象为接口函数，在TI-RTOS内核用户指南的队列模块文档中可以查看有关的基础功能 。`util.c`中的函数结合队列模块中的队列与事件模块中的事件来实现线程之间消息传递。

在CC2640R2F软件中，ICall使用来自各自模块的队列和事件在应用程序和协议栈任务之间传递消息。高优先级任务，Hwi或Swi将消息发布到队列中传递给应用程序任务。然后，当没有其他高优先级线程运行时，应用程序任务将在其上下文中处理此消息。

`util`模块包含一组抽象的TI-RTOS队列功能，如下所示：

- Util_constructQueue（）创建一个队列。
- Util_enqueueMsg（）将项目放入队列。
- Util_dequeueMsg（）从队列中获取项目。

功能实例

图25和图26说明了队列如何将按键消息从Hwi排入队列（到Swi中的板卡按键模块），然后在任务上下文中发布后处理。这个实例来自于BLE5-Stack中的simple_central工程。

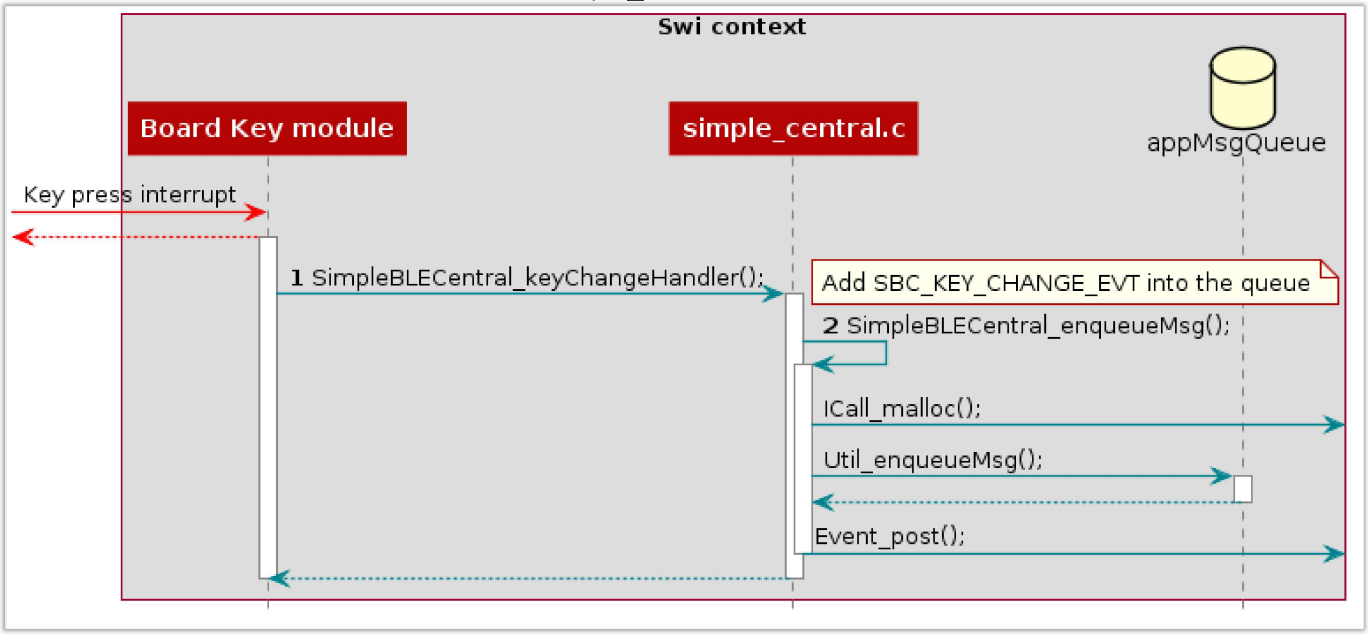

图25. 消息加入队列的时序图

在中断使能的情况下，引脚中断可能会在Hwi上下文中异步发生 。为了使中断尽可能短，与中断相关的工作推迟到任务来处理。在BLE5-Stack中的simple_central示例中，引脚中断通过板卡按键模块进行抽象。该模块通过Swi注册回调通知函数 。在这种情况下，`SimpleBLECentral_keyChangeHandler`是注册的回调函数。

图25中的步骤1：展示了当键按下的时候回调`SimpleBLECentral_keyChangeHandler`。该事件被放入应用程序的队列中等待处理。

清单9. 定义SimpleBLECentral_keyChangeHandler（）

```
1 void SimpleBLECentral_keyChangeHandler(uint8 keys)
2 {
3       SimpleBLECentral_enqueueMsg(SBC_KEY_CHANGE_EVT, keys, NULL);
4 }
```

图25中的步骤2：显示了按键消息是如何被压入`simple_central`任务的队列中的。首先通过`ICall_malloc（）`分配内存，以便消息可以放入队列中。一旦消息添加进队列，Util_enqueueMsg（）将发布一个`UTIL_QUEUE_EVENT_ID`事件来通知应用程序进行处理。

清单10. 定义SimpleBLECentral_enqueueMsg（）

```
1 static uint8_t SimpleBLECentral_enqueueMsg(uint8_t event, uint8_t state, uint8_t *pData)
2 {
3   sbcEvt_t *pMsg = ICall_malloc(sizeof(sbcEvt_t));
```

```
4   // Create dynamic pointer to message.
5   if (pMsg)
6   {
7     pMsg->hdr.event = event;
8     pMsg->hdr.state = state;
9     pMsg->pData = pData;

10     // Enqueue the message.
11     return Util_enqueueMsg(appMsgQueue, sem, (uint8_t *)pMsg);
12   }
13   return FALSE;
14 }
```

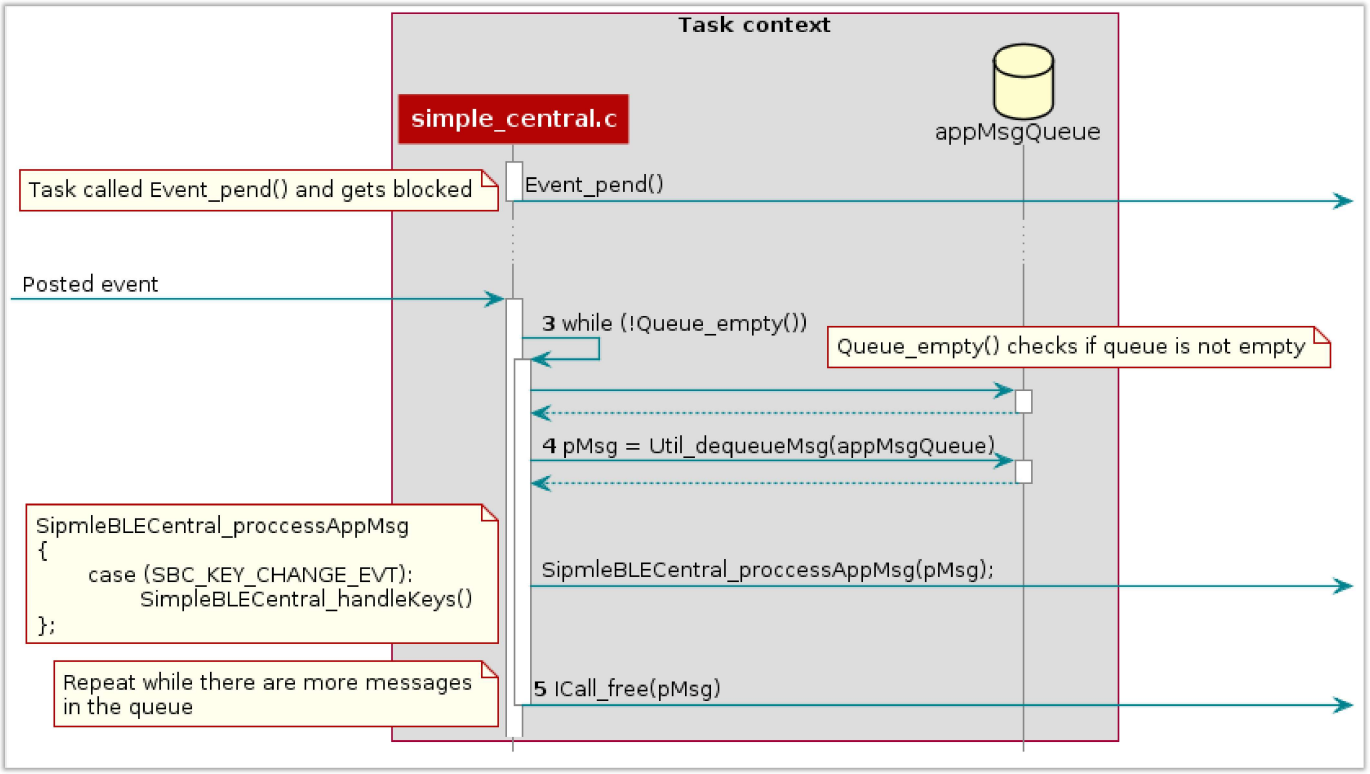

图26. 消息出队的时序图

图26中的步骤3:`simple_central`应用程序一直在检查是否有消息被放置在队列中需要进行处理，当`UTIL_QUEUE_EVENT_ID`事件发布之后，它也就被解除阻塞，开始处理消息。

清单11. 处理应用程序消息

```
1 // If TI-RTOS queue is not empty, process app message
2 while (!Queue_empty(appMsgQueue))
3 {
4        sbcEvt_t *pMsg = (sbcEvt_t *)Util_dequeueMsg(appMsgQueue);
5    if (pMsg)
6    {
7          // Process message
8          SimpleBLECentral_processAppMsg(pMsg);

9          // Free the space from the message
10    ICall_free(pMsg);
11   }
12      }
13
```

图26中的步骤4：`simple_central`应用取出队列中的消息并对其进行处理。

清单12. 处理按键中断消息

```
1 static void SimpleBLECentral_processAppMsg(sbcEvt_t *pMsg)
2 {
3  switch (pMsg->hdr.event)
4  {
5    case SBC_KEY_CHANGE_EVT:
6      SimpleBLECentral_handleKeys(0, pMsg->hdr.state);
7      break;
8    //...
9   }
10}
```

图26中的步骤5：`simple_central`应用程序处理完消息后可以释放在**步骤2**中所分配的内存。

任务

TI-RTOS任务（或称为线程）就是一段简单的程序，通常是一段死循环。实际上，将处理器从一个任务切换到另一个任务有助于实现并发。每个任务总是处于以下运行状态之一：

- **运行**：任务当前正在运行
- **就绪**：任务准备好等待执行
- **阻塞**：任务被暂停执行
- **终止**：任务终止执行
- **无效**：任务处于无效列表中（还不受TI-RTOS管理）

有且只有一个任务总是在CPU中运行，当然它有可能只是空闲任务（见图21.）。当前运行的任务可以被某些唤醒的高优先级任务以及其他模块（如Semaphores）的功能阻止执行。当前任务也可以自行终止执行。在任一情况下，处理器都切换到准备运行的最高优先级任务。有关这些功能的更多信息，请参见**TI-RTOS内核用户指南**中TI.sysbios.knl软件包中的任务模块。

每个任务都会分配相应的优先级，多个任务可以具有相同的优先级。任务是从最高优先级向最低优先级执行的;相同优先级的任务按照到达顺序进行执行。当前运行的任务的优先级永远不会低于任何就绪任务的优先级。当有更高优先级的任务就绪时，正在运行的任务才会被抢占并重新安排执行。

在`simple_peripheral`应用中，低功耗蓝牙协议栈任务被给予最高优先级（5），应用任务被给予最低优先级（1）。

初始化任务

初始化任务时，会给它分配独立的运行堆栈，用于存储局部变量以及进一步嵌套函数调用。在单个程序中执行的所有任务共享一组通用的全局变量，根据C语言标准规范中的函数访问范围进行访问。分配的运行堆栈就是任务的上下文。以下是正在构建的应用程序任务的示例。

清单13. TI-RTOS任务

```
1 #include <xdc/std.h>
2 #include <ti/sysbios/BIOS.h>
```

```
3 #include <ti/sysbios/knl/Task.h>

4 /* Task's stack */
5 uint8_t sbcTaskStack[TASK_STACK_SIZE];

6 /* Task object (to be constructed) */
7 Task_Struct task0;

8 /* Task function */
9 void taskFunction(UArg arg0, UArg arg1)
10 {
11     /* Local variables. Variables here go onto task stack!! */

12     /* Run one-time code when task starts */

13     while (1) /* Run loop forever (unless terminated) */
14     {
15         /*
16          * Block on a signal or for a duration. Examples:
17          *  ``Sempahore_pend()``
18          *  ``Event_pend()``
19          *  ``Task_sleep()``
20          *
21          * "Process data"
22          */
23     }
24 }

25 int main() {

26    Task_Params taskParams;

27    // Configure task
28    Task_Params_init(&taskParams);
29    taskParams.stack = sbcTaskStack;
30    taskParams.stackSize = TASK_STACK_SIZE;
31    taskParams.priority = TASK_PRIORITY;

32    Task_construct(&task0, taskFunction, &taskParams, NULL);

33    BIOS_start();
34 }
```

在启动TI-RTOS内核的调度程序`BIOS_start()`之前，在main（）中需要完成所有任务的创建。任务在调度程序启动后按其分配的优先级执行。

TI建议尽量使用现有的任务应用程序进行特定应用的处理。在工程中添加额外的任务时，必须在TI-RTOS配置文件（`.cfg`）中定义的TI-RTOS优先级范围内为该任务分配优先级。

> 提示：
> 尽量减少任务优先级别数量，以在TI-RTOS配置文件（.cfg）中节省出额外的RAM：
> `Clock.tickPeriod = 6;`

不要添加具有等于或高于低功耗蓝牙协议栈任务优先级的任务。有关系统任务层次结构的详细信息，可以在**标准工程任务层次结构**中查看。

确保任务的堆栈大小至少为512字节的预定义内存。必须分配足够大的堆栈空间来保证程序的正常运行以及任务抢占上下文的存储。任务抢占上下文是当一个任务由于中断产生或者被较高优先级任务抢占而被保存的上下文。使用IDE的TI-RTOS分析工具，可以分析任务以确定任务堆栈使用情况的峰值。

> 注意：
> 这里用术语讲了如何去构建任务。在TI-RTOS中实际是如何构建任务的你可以在本文前面的**创建与构造**中查看。

任务功能

当任务被初始化时，任务函数的函数指针会传递给`Task_construct`函数。当任务第一次有机会运行时，这个函数就是由TI-RTOS管理运行的函数了。清单13.显示了此任务函数的一般拓扑关系。

就典型的使用情况而言，任务大部分时间都通过调用称为`_pend()`的API（例如`Semaphore_pend()`）而处于阻塞状态。通常，高优先级线程（例如Hwis或Swis）使用`_post()`API（例如`Semaphore_post()`）来解除某些任务的阻塞。

时钟

时钟实例是可以在一定数量的系统时钟节拍之后运行的函数。时钟实例可以是单次或周期性的。这些实例可以配置为在创建后立即开始或者在一段延迟后启动，并可以随时停止。所有的时钟实例当它们在Swi的上下文中时钟溢出就会被执行。以下示例显示了TI-RTOS配置文件（`.cfg`）中设置的TI-RTOS时钟周期的最小分辨率。

> 注意：
> 默认的TI-RTOS内核刻度周期为1毫秒。对于CC2640R2F器件，在TI-RTOS配置文件（`.cfg`）中重新配置：
> Clock.tickPeriod = 10 ;

每个系统时钟节拍会启动一个时钟对象来为节拍计数，然后再和一系列的时钟进行比较以确定相关函数是否该运行。对于需要更高分辨率的定时器，TI建议使用16位硬件定时器通道或传感器控制器来做。有关这些功能的更多信息，请参见TI-RTOS内核用户指南的`TI.sysbios.knl`软件包中的Clock模块。

您可以直接在应用程序中使用内核的Clock API，`Util`模块中提供了一组抽象的TI-RTOS时钟功能，如下所示：

- `Util_constructClock()` 创建一个Clock对象。
- `Util_startClock()` 启动一个现有的Clock对象。
- `Util_restartClock()` 停止，重新启动一个现有的Clock对象。
- `Util_isActive()` 检查Clock对象是否正在运行。
- `Util_stopClock()` 停止一个现有的Clock对象。
- `Util_rescheduleClock()` 重新配置一个现有的Clock对象。

功能实例

以下示例来自BLE5-Stack中的simple_peripheral项目。

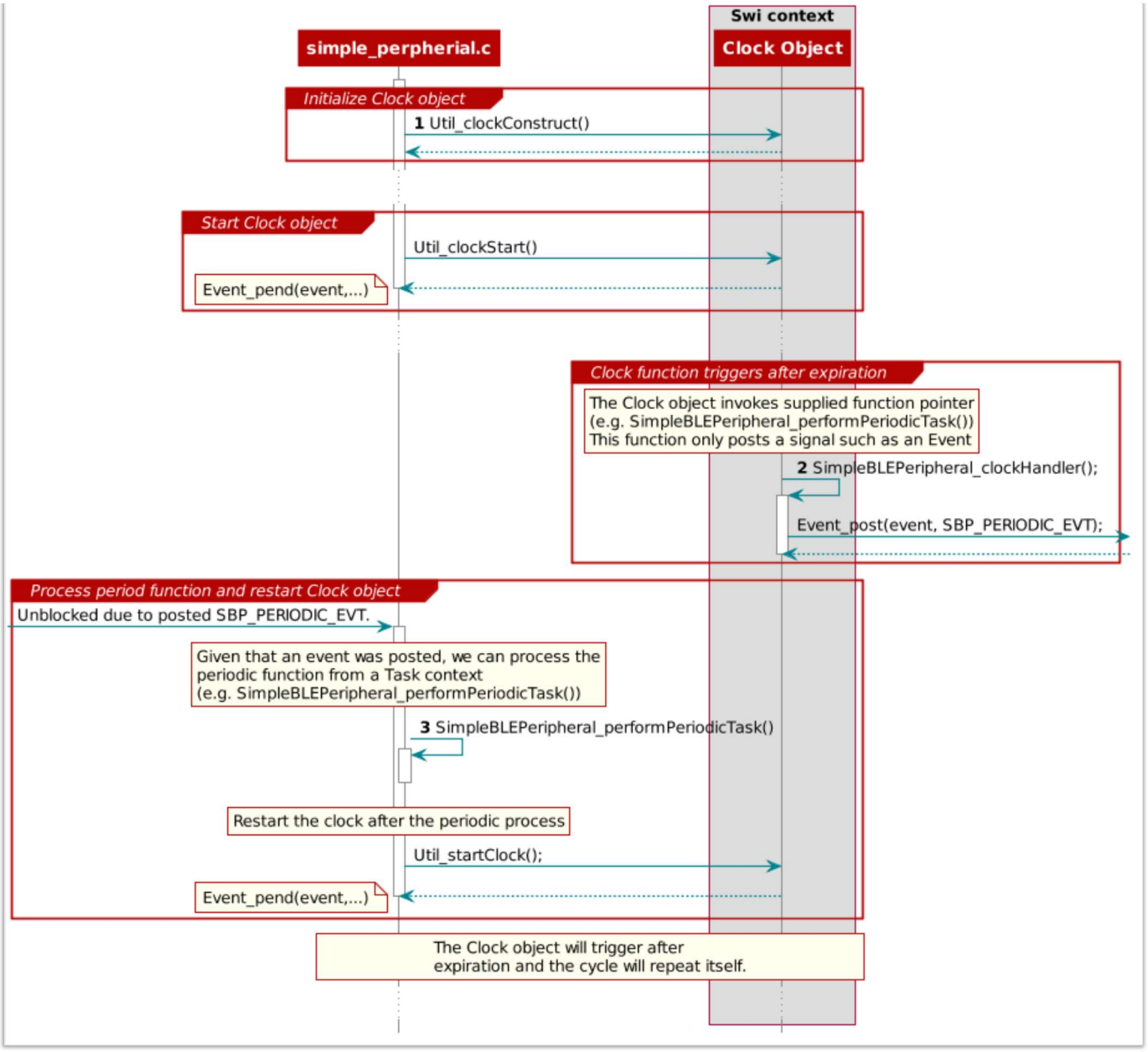

图27. 触发时钟对象

步骤1：在图27中，通过`Util_constructClock()`来构建时钟对象。在示例进入连接状态后，它将通过`Util_startClock()`启动Clock对象。

清单14. 在`simple_peripheral`中的构建`periodicClock`Clock对象

```
// Clock instances for internal periodic events.
static Clock_Struct periodicClock;

// Create one-shot clocks for internal periodic events.
Util_constructClock(&periodicClock, SimpleBLEPeripheral_clockHandler,
    SBP_PERIODIC_EVT_PERIOD, 0, false, SBP_PERIODIC_EVT);
```

步骤2：在图27中，Clock对象的计时器到期后，它将在Swi上下文中执行`SimpleBLEPeripheral_clockHandler()`。由于此调用不能被阻止并能阻止所有的任务，所以调用`simple_peripheral`中的`Event_post(SBP_PERIODIC_EVT)`来进行事件发布以尽可能保证其处理过程足够短暂。

清单15. 定义SimpleBLEPeripheral_clockHandler（）

```
static void SimpleBLEPeripheral_clockHandler(UArg arg)
{
    /* arg is passed in from Clock_construct() */
    Event_post(events, arg);
}
```

> 注意:
> 时钟功能不能调用阻塞内核的API或TI-RTOS驱动程序的API！时钟功能的处理时间过长将影响分配给无线协议栈的高优先级任务中的实时约束！

步骤3:在图27中，在`Event_post(SBP_PERIODIC_EVT)`发布了事件之后`simple_peripheral`任务被解除阻塞，然后继续调用`SimpleBLEPeripheral_performPeriodicTask()`函数。最后如果要重新启动此功能的定期执行，需要重新启动`periodicClock`时钟对象。

清单16.维护SBP_PERIODIC_EVT事件

```
if (events & SBP_PERIODIC_EVT)
{
  // Perform periodic application task
  SimpleBLEPeripheral_performPeriodicTask();

  Util_startClock(&periodicClock);
}
```

驱动

TI-RTOS提供了一套可以添加到应用程序的CC26xx外设驱动程序。驱动程序为应用程序提供了与CC26xx板载外设接口以及外部设备通信的机制。这些驱动程序在`DriverLib`中抽象了对寄存器的访问。

有关BLE5-Stack中每个TI-RTOS驱动程序的重要文档及有关具体位置，请参阅`BLE5-Stack release notes`。本节仅概述了驱动程序如何适应软件生态系统。有关可用的功能和驱动程序API的说明，请参阅`TI-RTOS API Reference`。

添加驱动程序

某些驱动程序会作为源文件被添加到工程项目工作区中Drivers文件夹下的相应文件夹中，如图28所示。

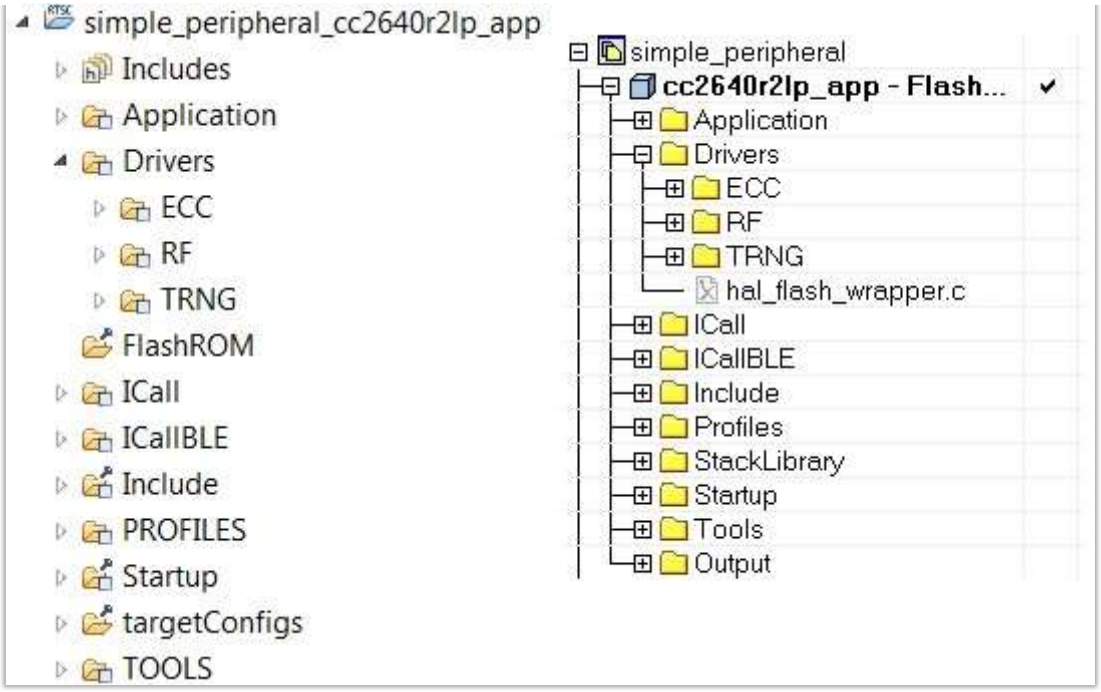

图28. 驱动程序文件夹

驱动的源文件可以在`$TI_RTOS_DRIVERS_BASE$\ti\drivers`的相应文件夹中找到。（`$TI_RTOS_DRIVERS_BASE$`指的是安装位置，可以在IAR Tools\Configure Custom Argument Variables菜单中查看。对于CCS，相应的路径变量在Project Options\ Resource\Linked Resources选项卡中定义）

例如，ECC和TRNG驱动程序是BLE5-Stack而不是TI-RTOS的一部分，它们分别位于`<SDK_INSTALL_DIR>\source\ti\ble5stack\common\cc26xx\ecc`和`<SDK_INSTALL_DIR>\source\ti\ble5stack\hal\src\target\_common`中。

要向工程中添加驱动程序，需要将相应驱动程序的头文件包含在引用驱动程序API的应用程序文件中。

例如，要添加用于读取或控制输出I/O引脚的PIN驱动程序，请添加以下内容：

```
#include <ti/drivers/pin/PINCC26XX.h>
```

还需要将以下TI-RTOS驱动程序文件添加到`Drivers\PIN`文件夹下的项目中：

- PINCC26XX.c
- PINCC26XX.h
- PIN.h

板文件

板文件为将通过设置相应的参数将固定的驱动配置修改为特定的板配置，例如配置PIN驱动程序的GPIO表或者定义将哪些引脚分配给I2C，SPI或UART。

有关板文件的更多信息，以及如何在TI EMs和LPs或本地硬件端口之间切换，请参见**TI提供的板文件部分**部分。

可用驱动程序

本节给大家介绍一些可用的驱动程序，并提供一些基本示例演示如何将驱动程序添加到`simple_peripheral`项目中。有关每个驱动程序的详细信息，可在**TI-RTOS API Reference**中查看。

引脚

PIN驱动程序用于控制GPIO的I/O引脚或着连接在上面的硬件外围设备。如**TI提供的板文件部分**所述，引脚在main（）中必须首先初始化为安全状态（在板文件中配置）。此初始化后，任何模块都可以使用PIN驱动程序配置一组引脚供使用。以下是在simple_peripheral任务中将一个引脚配置成作为中断使用，另一个配置成作为输出使用的示例，在中断发生时交换它们的角色。IOID_x引脚号对应于CC26XX技术参考手册中引用的DIO引脚号 。下表列出了使用的引脚及其在`CC2640R2F LaunchPad`上的映射，这些已在板文件中定义了。

信号名称	针号	CC2640R2F LaunchPad映射
CC2640R2_LAUNCHXL_PIN_RLED	IOID_6	DIO6（红色）
CC2640R2_LAUNCHXL_PIN_BTN1	IOID_13	DIO13（BTN_1）

`simple_peripheral.c`代码需要做以下修改。

1. 包括PIN驱动程序文件：
   `#include <ti / drivers / pin / PINCC26xx.h>`

2. 声明引脚配置表和引脚状态并申明simple_peripheral任务使用的变量：
   清单17. 引脚配置表

```
static PIN_Config SBP_configTable[] =
{
    CC2640R2_LAUNCHXL_PIN_RLED | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVST
    CC2640R2_LAUNCHXL_PIN_BTN1 | PIN_INPUT_EN | PIN_PULLUP | PIN_IRQ_BOTHEDGES | PIN_HYSTERES
    PIN_TERMINATE
};

static PIN_State sbpPins;
static PIN_Handle hSbpPins;
static uint8_t LED_value = 0;
```

3. 声明在Hwi上下文中执行ISR：
   清单18. 声明ISR

```
static void buttonHwiFxn(PIN_Handle hPin, PIN_Id pinId)
{
    SimpleBLEPeripheral_enqueueMsg(SBP_BTN_EVT, 0, NULL);
}
```

4. 在SimpleBLEPeripheral_processAppMsg中，添加一个case来处理上面的事件，并定义事件：
   清单19. ISR事件处理

```
#define SBP_BTN_EVT

static void SimpleBLEPeripheral_processAppMsg(sbcEvt_t *pMsg)
{
    switch (pMsg->hdr.event)
    {
        case SBP_BTN_EVT:
            //toggle red LED
```

```
            if (LED_value)
            {
                PIN_setOutputValue(hSbpPins, CC2640R2_LAUNCHXL_PIN_RLED , LED_value--);
            }
            else
            {
                PIN_setOutputValue(hSbpPins, CC2640R2_LAUNCHXL_PIN_RLED, LED_value++);
            }
            break;
            //...
    }
}
```

5. 打开引脚使用，并在simple_peripheral_init（）中配置中断：
   清单20. 打开引脚并配置中断

```
// Open pin structure for use
hSbpPins = PIN_open(&sbpPins, SBP_configTable);
// Register ISR
PIN_registerIntCb(hSbpPins, buttonHwiFxn);
// Configure interrupt
PIN_setConfig(hSbpPins, PIN_BM_IRQ, CC2640R2_LAUNCHXL_PIN_BTN1 | PIN_IRQ_NEGEDGE);
// Enable wakeup
PIN_setConfig(hSbpPins, PINCC26XX_BM_WaKEUP, CC2640R2_LAUNCHXL_PIN_BTN1|PINCC26XX_WAKEUP_NEGE
```

6. 编译
7. 下载
8. 运行

> 注意：
> 按下CC2640R2F LaunchPad上的BTN-1按钮可切换红色LED，这里没有做去抖动处理。

GPIO

GPIO模块允许您通过简单易用的API来管理通用I/O引脚。GPIO引脚的行为通常是静态配置的，但也可以在运行时进行重新配置。

由于其简单性，GPIO驱动程序不遵循其他TI-RTOS驱动程序类型，其中驱动程序的API具有单独的特定于设备的实现。在GPIOxxx_Config结构中，这种差异最为明显，它不需要您指定特定的函数表或对象,而其他驱动程序在各自的xxx_config中则要指定函数表和对象，你可以在驱动程序的配置文件中查看，例如：CC2640R2_LAUNCHXL.c。

以下是一个如何配置GPIO引脚，并利用注册中断回调函数来控制LED灯的打开关闭的示例。

1. 在`simple_peripheral.c`中包含GPIO驱动程序文件：
   `#include <ti / drivers / GPIO.h>`

在Board.c中做一下添加：

2. 添加一组`GPIO_PinConfig`元素，用于定义应用程序使用的每个引脚的初始配置。引脚类型（即INPUT/OUTPUT），初始状态（即OUTPUT_HIGH或LOW），中断特性（RISING/FALLING边沿等）以

及器件特定引脚标识。以下是对于CC26XX设备`GPIO_PinConfig`数组的特定配置示例：
清单21. 设置GPIO引脚配置数组

```
//
// Array of Pin configurations
// NOTE: The order of the pin configurations must coincide with what was
//       defined in CC2640R2_LAUNCH.h
// NOTE: Pins not used for interrupts should be placed at the end of the
//       array.  Callback entries can be omitted from callbacks array to
//       reduce memory usage.
//
GPIO_PinConfig gpioPinConfigs[] = {
    // Input pins
    GPIOCC26XX_DIO_13 | GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_RISING,  // Button 0
    GPIOCC26XX_DIO_14 | GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_RISING,  // Button 1
    // Output pins
    GPIOCC26XX_DIO_07 | GPIO_CFG_OUT_STD | GPIO_CFG_OUT_STR_HIGH | GPIO_CFG_OUT_LOW,     //
    GPIOCC26XX_DIO_06 | GPIO_CFG_OUT_STD | GPIO_CFG_OUT_STR_HIGH | GPIO_CFG_OUT_LOW,     //
};
```

3. 添加一组`GPIO_CallbackFxn`元素，用于存储配置有中断的GPIO引脚的回调函数指针。这些数组元素的索引对应于`GPIO_PinConfig`数组中定义的引脚。这些函数指针可以通过引用数组元素中的回调函数名来静态定义，也可以通过动态地将数组元素设置为NULL并在运行时使用`GPIO_setCallback（）`来插入回调条目。不用于中断的引脚可以从回调数组中省略，以减少内存使用（如果它们位于`GPIO_PinConfig`数组的末尾）。回调函数语法应符合以下条件：
   `void （* GPIO_CallbackFxn ）（unsigned int index ）;`

index参数与传递给GPIO_setCallback（）的索引相同。这允许通过使用索引来识别哪个GPIO发生了中断，可以将相同的回调函数用于多个GPIO中断。以下是CC26XX设备的GPIO_CallbackFxn数组的特定示例：
清单22. 设置GPIO回调函数数组

```
//
// Array of callback function pointers
// NOTE: The order of the pin configurations must coincide with what was
//       defined in CC2640R2_LAUNCH.h
// NOTE: Pins not used for interrupts can be omitted from callbacks array to
//       reduce memory usage (if placed at end of gpioPinConfigs array).
//
GPIO_CallbackFxn gpioCallbackFunctions[] = {
    NULL,  //  Button 0
    NULL,  //  Button 1
};
```

4. 将前面提到的两个数组以及每个数组的元素数量添加到`GPIOCC26XX_Config`结构体中用于驱动程序接口调用。此处还指定了所有能产生中断的引脚的中断优先级。中断优先级的值是是跟设备有关的，同一个引脚的中断优先级对于不同设备是不同的。在将此参数设置为非默认值之前，应熟悉设备中使用的中断控制器。优先级的默认值用于表示应使用最低可能的优先级。以下是初始化`GPIOCC26XX_Config`结构的示例：
   清单23. 设置GPIO配置结构

```
const GPIOCC26XX_Config GPIOCC26XX_config = {
     .pinConfigs = (GPIO_PinConfig *)gpioPinConfigs,
```

```
    .callbacks = (GPIO_CallbackFxn *)gpioCallbackFunctions,
    .numberOfPinConfigs = sizeof(gpioPinConfigs)/sizeof(GPIO_PinConfig),
    .numberOfCallbacks = sizeof(gpioCallbackFunctions)/sizeof(GPIO_CallbackFxn),
    .intPriority = (~0)
};
```

以下是需要在`simple_peripheral.c`中添加的内容：
5. 按键回调函数：
清单24. 设置按键的回调函数

```
//
//  ======== gpioButtonFxn0 ========
//  Callback function for the GPIO interrupt on CC2640R2_LAUNCHXL_PIN_BTN1.
//
void gpioButtonFxn0(unsigned int index)
{
    // Toggle the LED
    GPIO_toggle(CC2640R2_LAUNCHXL_PIN_BTN1);
}
```

6. 初始化和使用GPIO（将其添加到simple_peripheral_init（））：
   清单25. 初始化和使用的GPIO

```
// Call GPIO driver init function
GPIO_init();

// Turn on user LED
GPIO_write(CC2640R2_LAUNCHXL_PIN_RLED, Board_GPIO_LED_ON);

// install Button callback
GPIO_setCallback(CC2640R2_LAUNCHXL_PIN_BTN1, gpioButtonFxn0);

// Enable interrupts
GPIO_enableInt(CC2640R2_LAUNCHXL_PIN_BTN1);
```

7. 编译
8. 下载
9. 运行

其他驱动程序

TI-RTOS附带的其他驱动程序有：UART，SPI，加密（AES），I2C，PDM，Power，RF和UDMA。由于协议栈使用了电源，RF和UDMA，因此在使用它们的时候，必须特别小心。与其他驱动程序一样，这些驱动程序也在文档中被详细记录，BLE5-Stack中提供了示例。

功耗管理

所有功耗管理功能由外设驱动程序和低功耗蓝牙协议栈进行处理。可以通过包含或移除POWER_SAVING预处理器定义的符号来启用或禁用此功能。当POWER_SAVING被使能时，设备根据低功耗蓝牙事件，外设事件，应用计时器等的要求进入和退出休眠状态。当POWER_SAVING未定义时，设备保持唤醒。有关修改预处理器定义的符号的步骤，请参阅用CCS开发中的**访问预处理器符号**或者用IAR开发中的**访问预处理器符号**。

有关电源管理功能的更多信息，包括API和自定义UART驱动程序的示例，可以在安装的TI-RTOS中包含的CC26xx的TI-RTOS电源管理中找到。只有使用自定义驱动程序时，才需要这些API。

另请参阅`Measuring Bluetooth Smart Power Consumption (SWRA478)`，以分析系统功耗和电池寿命。

加入我们

文章所有代码、工具、文档开源。加入我们**QQ群 591679055**获取更多支持，共同研究CC2640R2F&BLE5.0。