

基于 ARM 架构的超低功耗 OS 可行性报告

崔天一 曹焕琦 邓翔 李泽昌

2016 年 3 月 20 日

摘要

本文简要论证了将 uC/OS 移植到蓝牙 DA14580 并实现低功耗且正常使用 DA14580 蓝牙协议栈的可行性，并描述了该种做法的创新点。本文依次介绍了 uC/OS 移植的一般步骤，外部中断的处理方法，休眠的实现方法。

目录

1	uC/OS II 的移植 [2]	1
1.1	移植条件	1
1.2	移植细节	2
1.2.1	OS_CPU.H	2
1.2.2	OS_CPU_C.C	4
1.2.3	OS_CPU_A.ASM	6
2	中断处理程序 (ISR) 的编写	7
2.1	中断服务程序简介	7
2.2	uC/OS II 中应用程序使用中断的方式	8
2.3	uC/OS II 中中断处理程序的一般执行步骤	10
2.4	我们的移植方法	11
3	蓝牙协议栈的移植	12
4	休眠功能的实现	12
5	创新点	15

1 uC/OS II 的移植 [2]

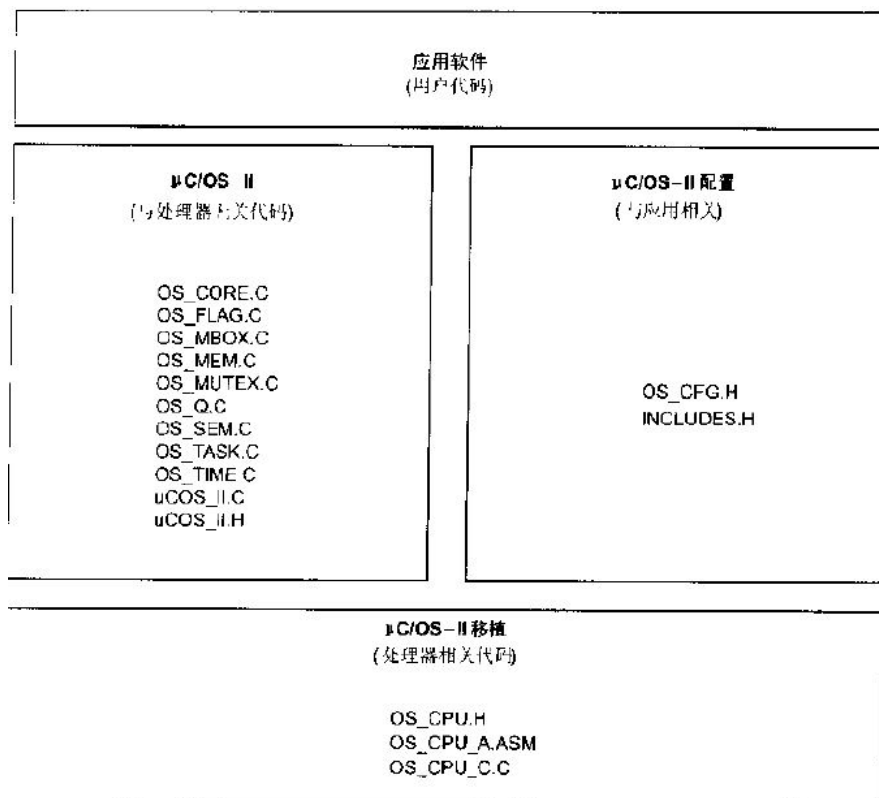
1.1 移植条件

要移植 uC/OS，处理器需满足以下条件。

1. 处理器的 C 编译器能够产生可重入型的代码
2. 处理器支持定时中断
3. 可以使用 C 语言开关中断
4. 处理器支持一定数量的数据存储硬件堆栈，可能需要几 KB
5. 处理器上的指令支持将堆栈指针以及其他 CPU 寄存器中的内容读出，并存储到相应的内存或堆栈中

开发过程中将使用 Keil 作为开发工具，编译器为 ARMCC，满足以上编译器的需求。开发版支持 32KHZ 的实时钟，且拥有 42KB 的 System SRAM，也有相应的存取指令，满足要求。

1.2 移植细节



在 uC/OS 的移植中，图中所示的与处理器无关的代码可以不用修改，移植到开发板上之后就可以利用里面支持的一些 uC/OS 的特性。可以看到，有三个文件 OS_CPU.H、OS_CPU_A.ASM 和 OS_CPU_C.C 是需要我们根据处理器修改的。

1.2.1 OS_CPU.H

不同的处理器支持的字长不同，所以需要定义不同的数据类型。

Type	Size in bits	Natural alignment in bytes
char	8	1 (byte-aligned)
short	16	2 (halfword-aligned)
int	32	4 (word-aligned)
long	32	4 (word-aligned)
long long	64	8 (doubleword-aligned)
float	32	4 (word-aligned)
double	64	8 (doubleword-aligned)
long double	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)
bool (C++ only)	8	1 (byte-aligned)
_Bool (C only [a])	8	1 (byte-aligned)
wchar_t (C++ only)	16	2 (halfword-aligned)

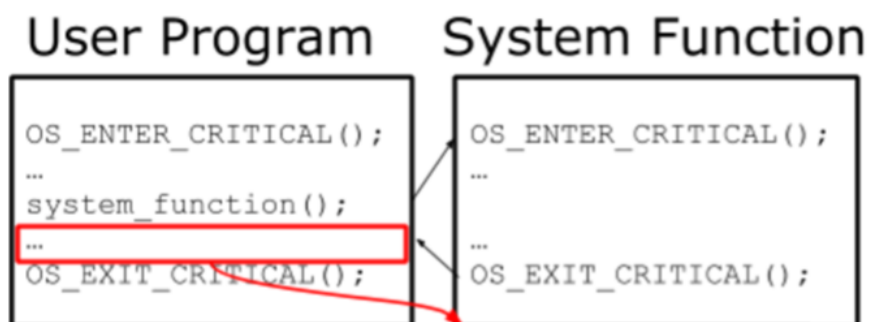
根据编译器支持的数据类型，定义相应的 uC/OS 数据类型，如整形，浮点型及任务堆栈的数据类型，如果使用了一种特定的开关中断的方式，还需要声明 CPU 的状态寄存器的数据类型。

除此之外，还需要定义 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 这两个宏，uC/OS 使用这两个宏来开关中断。具体实现有三种。

第一种方法是通过调用处理器的指令来开关中断。但在这种情况下，关闭中断后调用功能函数，返回时中断可能已被开启。比如，在一个中断子程序中，先关闭了中断，在退出时再开启中断。但是在中间的用户代码里调用了功能函数，该功能函数在进入时关闭中断，退出时开启中断，这样在退出中断子程序前中断就被开启了。

第二种方法与第一种的区别是，在功能函数关中断前先将中断状态保存到堆栈，最后功能函数退出时不是开启中断，而是恢复原来的保存到堆栈中的中断状态，这样调用功能函数后就不会改变中断状态了。但如果编译器对插入汇编优化的得不够好的话，编译器可能不会知道堆栈指针已经因为保存了中断状态而改变，这样的话，所有变量的栈偏移量可能会偏差。

第三种方法是利用编译器的扩展功能，将当前处理器状态字的值保存在局部变量里，最后用这个局部变量来恢复。



若OS_EXIT_CRITICAL()简单地
打开中断, 此时中断便处于打开
状态, 不符合期望的功能;

而若OS_EXIT_CRITICAL()恢复
OS_ENTER_CRITICAL()时的中
断状态, 就不会出现此种问题。

变量 OS_STK_GROWTH 表示堆栈使用方向, 设置为 1 表示高地址往低地址递减, 0 则相反。

从低优先级任务切换到高优先级任务时需要调用 OS_TASK_SW() 宏。处于就绪态的任务的寄存器被保存在堆栈中, 就像刚发生了中断一样。执行就绪态的任务, 也就意味着恢复一系列的寄存器, 并执行中断返回指令。这个宏就是用来模拟中断的产生。一般的处理器支持软中断或 TRAP 来完成这项操作, 做法是让中断向量指向一个在 OS_CPU_A.ASM 中定义的任务级切换函数 OSCtxSW(), 或者调用它。

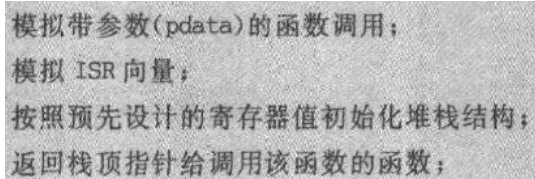
1.2.2 OS_CPU_C.C

这个文件中需要编写 10 个 c 语言函数。

```
OSTaskStkInit();  
OSTaskCreateHook();  
OSTaskDelHook();  
OSTaskSwHook();  
OSTaskIdleHook();  
OSTaskStatHook();  
OSTimeTickHook();  
OSInitHookBegin();  
OSInitHookEnd();  
OSTCBInitHook();
```

但必须编写的只有 OSTaskStkInit() 函数，其余只要声明，不必包含内容。

该函数初始化堆栈结构，完成如下工作。



模拟带参数(pdata)的函数调用;
模拟 ISR 向量;
按照预先设计的寄存器值初始化堆栈结构;
返回栈顶指针给调用该函数的函数;

以下是一个具体例子 [3]

```

void *OSTaskStkInit (void (*task) (void *pd), void *pdata, void *ptos, INT16U opt)
{
    unsigned int *stk;
    opt = opt; /* 因为 'opt' 变量没有用到, 防止编译器产生警告 */
    stk = (unsigned int *) ptos; /* 装载堆栈指针 */
    /* 为新任务创建上下文 */
    *--stk = (unsigned int) task; /* pc */
    *--stk = (unsigned int) task; /* lr */
    *--stk = 0; /* r12 */
    *--stk = 0; /* r11 */
    *--stk = 0; /* r10 */
    *--stk = 0; /* r9 */
    *--stk = 0; /* r8 */
    *--stk = 0; /* r7 */
    *--stk = 0; /* r6 */
    *--stk = 0; /* r5 */
    *--stk = 0; /* r4 */
    *--stk = 0; /* r3 */
    *--stk = 0; /* r2 */
    *--stk = 0; /* r1 */
    *--stk = (unsigned int) pdata; /* r0 */
    *--stk = (SVC32MODE|0x0); /* cpsr IRQ, 关闭FIQ */
    *--stk = (SVC32MODE|0x0); /* spsr IRQ, 关闭FIQ */
    return ((void *) stk);
}

```

1.2.3 OS_CPU_A.ASM

这里需要编写 4 个汇编函数。

OSStartHighRdy() 用于使任务优先级最高的任务运行。该函数首先调用 OSTaskSwHook(), 这由用户在 OS_CPU_C.c 中自己定义。然后设置 OSRunning 为 TRUE。接着获取优先级最高的任务的堆栈指针, 恢复其所有的寄存器。

OSCtSw() 是一个任务级的任务切换函数。首先保存寄存器到堆栈, 再把堆栈指针存到当前任务的 TCB 里, 然后调用 OS_CPU_C.c 中自己定义的 OSTaskSwHook() 函数。接着指向当前任务的指针的值改变, 指向要恢复的任务, 获取优先级, 堆栈指针, 按与保存时相反的方向恢复寄存器。

OSTickISR() 函数里由用户提供周期性地时钟源。可以在 OSStart() 运行后的第一个任务中初始化节拍中断。这个函数需要同其他中断服务子程序一样, 设置 OSIntNesting 等等, 它的用户代码就是调用 OSTimeTick()。

最后一个是 OSIntCtxSw()。该函数在 ISR 中被 OSIntExit() 调用，因此假定所有寄存器都已保存，比起 OSCtxSw() 只是少了保存处理器的代码。

2 中断处理程序 (ISR) 的编写

2.1 中断服务程序简介

在移植过程中，我们不可避免的需要完成一些中断服务程序的编写 (Interrupt Service Routine)。中断服务程序是一系列中断发生时 CPU 会跳转到的函数。如下为一个时钟中断处理函数：

```
void SysTick_Handler(void)
{
    //set_pxact_gpio();

    if(app_stream_check_for_button_release && GPIO_GetPinStatus(GPIO_PORT_0, GPIO_PIN_6))
    {
        app_stream_button_released();
    }

    if (started)
    {
        request_more_data_if_possible();
    }
}
```

事实上，开发板上所有具有的中断都在其官方提供的 SDK 包中的启动初始化代码 boot_vector.s 中有所体现，如下图所示：

__Vectors	DCD	__initial_sp	; Top of Stack
	DCD	Reset_Handler	; Reset Handler
	DCD	NMI_Handler	; NMI Handler
	DCD	HardFault_Handler	; Hard Fault Handler
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	SVC_Handler	; SVCaLL Handler
	DCD	0	; Reserved
	DCD	0	; Reserved
	DCD	PendSV_Handler	; PendSV Handler
	DCD	SysTick_Handler	; SysTick Handler

所需的中断处理程序也可分为两部分：由 ARM Cortex-M0 定义的 handler 和由开发厂商定义的 handler，上图所示的就是由 Cortex-M0 所定义的 handler。

通过观察 boot_vector.s 我们发现，它本身为全部的 handler 定义了符号位置，但导出到全局时仅是 WEAK 导出，因此当有 C 语言程序中包含了初始化汇编程序所定义的 Handler 的同名函数时，链接时对应的符号就会

被填充上对应的函数的地址，否则会指向 boot_vector.s 中的符号位置，此处仅有一条 B 指令和一条 ENDP 指令，不响应中断直接跳回原位。

```
Default_Handler PROC
    EXPORT BLE_WAKEUP_LP_Handler    [WEAK]
    EXPORT BLE_FINETGTIM_Handler    [WEAK]
    (部分省略)
    EXPORT GPIO4_Handler            [WEAK]

    BLE_WAKEUP_LP_Handler
    BLE_FINETGTIM_Handler
    (部分省略)
    GPIO4_Handler
    B .
    ENDP
```

因此，我们在移植完 uC/OS II 操作系统之后，只需定义对应名称的中断函数即可实现对应的中断功能。

2.2 uC/OS II 中应用程序使用中断的方式

通常在嵌入式系统中，每一个进程都是一个死循环，而在进程的主要循环中，其必须调用一个或多个 OSFlagePend() OSQPend() OSSemPend() 等函数从而等待某些事件的发生从而将 CPU 使用权让出给其余进程使用。如下图是一个典型的 uC/OS II 应用程序的结构。

```

void MyTask (void *p_arg)
{
    /* Local variables                                     */
    /* Do something with "p_arg"                           */
    /* Task initialization                                   */
    while (DEF_ON) { /* Task body, as an infinite loop.    */
        :
        /* Task body ... do work!                          */
        :
        /* Must call one of the following services:        */
        /* OSFlagPend()                                     */
        /* OSMutexPend()                                    */
        /* OSPendMulti()                                    */
        /* OSQPend()                                        */
        /* OSSemPend()                                     */
        /* OSTimeDly()                                      */
        /* OSTimeDlyHMSM()                                  */
        /* OSTaskQPend()                                    */
        /* OSTaskSemPend()                                  */
        /* OSTaskSuspend() (Suspend self)                  */
        /* OSTaskDel()   (Delete self)                     */
        :
        /* Task body ... do work!                          */
        :
    }
}

```

而在一个应用程序等待某个中断发生时，使用的是 EVENT FLAG 机制。Even Flags 是一系列的真值变量，应用程序可以调用 OSFlagPend() 函数等待一组真值变量中的任意一个变为真 (OR) 或等待所有变量均变为真 (AND)，而作为一个中断处理程序 ISR，可以调用 OSFlagPost() 设置一组变量中的一个为真或假。因此，如果一个应用程序希望等待一个中断的发生，其可以设置一组事件标志组 (EVENT FLAG)，并等待其变为真。而对应的中断处理程序可以设置对应的 Event Flag 从而让应用程序进入 Ready List 并得知中断已经发生。如下图为一个典型的例子：

```

#define BATT_LOW (OS_FLAGS)0x0002
#define SW_PRESSED (OS_FLAGS)0x0004

OS_FLAG_GRP MyEventFlagGrp;                                (2)

void main (void)
{
    OS_ERR err;

    OSInit(&err);
    :
    OSFlagCreate(&MyEventFlagGrp,                            (3)
                 "My Event Flag Group",
                 (OS_FLAGS)0,
                 &err);
    /* Check 'err' */
    :
    OSStart(&err);
}

void MyTask (void *p_arg)                                    (4)
{
    OS_ERR err;
    CPU_TS ts;

    while (DEF_ON) {
        OSFlagPend(&MyEventFlagGrp,                            (5)
                   TEMP_LOW + BATT_LOW,
                   (OS_TICK )0,
                   (OS_OPT)OS_OPT_PEND_FLAG_SET_ANY,
                   &ts,
                   &err);
        /* Check 'err' */
        :
    }
}

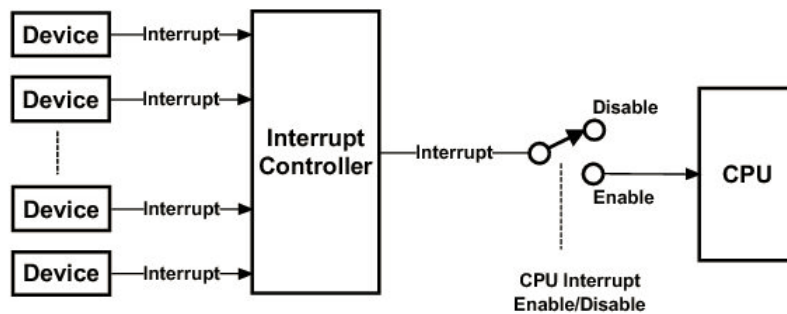
void MyISR (void)                                            (6)
{
    OS_ERR err;
    :
    OSFlagPost(&MyEventFlagGrp,                                (7)
               BAT_LOW,
               (OS_OPT)OS_OPT_POST_FLAG_SET,
               &err);
    /* Check 'err' */
    :
}

```

在该任务中，如果发生了电池电量低的事件，MyTask 就会从 OSFlagPend() 中返回并进行相应的处理。

2.3 uC/OS II 中中断处理程序的一般执行步骤

根据 uC/OS 官方提供的文档 [1]，如下图为一般的嵌入式系统中中断的硬件连接。



由于操作系统的存在，为了避免中断发生时发生进程上下文切换等事件的发生以及中断处理过程中又有中断发生，一般需要按照如下流程对中断进行处理。

```

MyISR:
    Disable all kernel aware interrupts;           (1)
    Save the CPU registers;                        (2)
    OSIntNestingCtr++;                             (3)
    if (OSIntNestingCtr == 1) {                    (4)
        OSTCBCurPtr->StkPtr = Current task's CPU stack pointer register value;
    }
    Clear interrupting device;                      (5)
    Re-enable kernel aware interrupts (optional);   (6)
    Call user ISR;                                 (7)
    OSIntExit();                                   (8)
    Restore the CPU registers;                      (9)
    Return from interrupt;                         (10)

```

首先需要关闭所有内核所能知晓的中断（对于有优先级的中断控制器，使用该种方法相当于没有使用优先级相关功能），以防止中断程序再次被中断。接下来保存 CPU 的所有寄存器在当前的任务栈上，标识自己进入了中断处理函数。如果自己是第一层中断，还需要保存当前任务的栈指针。截至将刚刚请求中断的设备的中断信号撤销，接着调用用户的中断程序（向其他程序发送事件标志 Event Flag），接着就是一系列的退出动作，接着从中断处理程序中返回至操作系统。

2.4 我们的移植方法

在 DA14580 的 SDK 中，其已提供了部分中断处理程序，但由于其并没有操作系统的支持，所以在移植的过程中我们需按 uC/OS II 的要求进行一定的封装，即 SDK 提供的中断相当于 user ISR，而我们则需在其基础上添加上关闭其他中断，保存 CPU 状态等内容即可。我们希望能从仅实现一个时钟中断开始（uC/OS 需要），逐步向上添加蓝牙、UART、GPIO 等相

关内容。

3 蓝牙协议栈的移植

根据 DA14580SDK 提供的相关内容，其内涵一个 RivieraWaves 提供的 ip 核，并用.h 导出了相关接口，如下图所示：

```
static const struct ke_msg_handler app_gap_process_handlers[]=
{
    {GAPM_DEVICE_READY_IND,                (ke_msg_func_t)gapm_device_ready_ind_handler},
    {GAPM_CMP_EVT,                          (ke_msg_func_t)gapm_cmp_evt_handler},
    {GAPC_CMP_EVT,                          (ke_msg_func_t)gapc_cmp_evt_handler},
    {GAPC_CONNECTION_REQ_IND,               (ke_msg_func_t)gapc_connection_req_ind_handler},
    {GAPC_DISCONNECT_IND,                   (ke_msg_func_t)gapc_disconnect_ind_handler},
    {APP_MODULE_INIT_CMP_EVT,               (ke_msg_func_t)app_module_init_cmp_evt_handler},
    {GAPM_ADV_REPORT_IND,                   (ke_msg_func_t)gapm_adv_report_ind_handler},
};
```

上图描述了处理 Generic Access Profile 所需的函数，而这些函数最后在 SDK 的包装之下给用户提供了一个良好的接口，用户只需编写一个回调函数即可，如下图所示：

```
void user_app_connection(uint8_t connection_idx, struct gapc_connection_req_ind const *param)
{
    if (app_env[connection_idx].conidx != GAP_INVALID_CONIDX)
    {
        app_connection_idx = connection_idx;

        // Stop the advertising data update timer
        app_easy_timer_cancel(app_adv_data_update_timer_used);

        // Check if the parameters of the established connection are the preferred ones.
        // If not then schedule a connection parameter update request.
        if ((param->con_interval < user_connection_param_conf.intv_min) ||
            (param->con_interval > user_connection_param_conf.intv_max) ||
            (param->con_latency != user_connection_param_conf.latency) ||
            (param->sup_to != user_connection_param_conf.time_out))
        {
            // Connection params are not these that we expect
            app_param_update_request_timer_used = app_easy_timer(APP_PARAM_UPDATE_REQUEST_TO, param_update_request_timer_cb);
        }
    }
    else
    {
        // No connection has been established, restart advertising
        user_app_adv_start();
    }
}
```

而我们所需完成的是将上述蓝牙代码中的核心代码在 uC/OS II 上重新编写并将其中包含的原有 SDK 中的内容（如消息传递等）用 uC/OS II 中提供的机制实现。

4 休眠功能的实现

根据 Dialog Semiconductor 提供的资料，如下图所示，是能够使 CPU 进入休眠的主循环的一般模式：

```

void main_loop(void)
{
    while(1)
    {
        schedule();

        WFI();
    }
}

```

但为了进入功耗更低的 Extended Sleep Mode 和 Deep Sleep Mode, 我们需要更详细的处理来替代简单地调用 WFI()。WFI() 为一个宏, 指向一个函数 __WFI(), 该函数调用 ARM 汇编指令 wfi, 其作用为使 CPU 停止运作直至新中断出现。因此, 为了进入不同的休眠模式, 我们需要在调用该宏前配置好各个组件的行为。具体需要执行的操作如下图:

```

GLOBAL __INT_STOP();
if (lp_clk_sel == LP_CLK_RCX20)
    set_sleep_delay();
sleep_mode = rwip_sleep();
if (sleep_mode == mode_sleeping) {
    if (to_ext_sleep()) sleep_mode = mode_ext_sleep;
    else sleep_mode = mode_deep_sleep;

    SetBits16(PMU_CTRL_REG, RADIO_SLEEP, 1);
    if( (sleep_mode == mode_deep_sleep) && ke_mem_is_empty(KE_MEM_NON_RETENTION) )
        func_check_mem_flag = 1;
    else
        sleep_mode = mode_ext_sleep;
    SCB->SCR |= 1<<2;
    SetBits16(SYS_CTRL_REG, PAD_LATCH_EN, 0);
    SetBits16(PMU_CTRL_REG, PERIPH_SLEEP, 1);
    if (sleep_mode == mode_ext_sleep) {
        SetBits16(SYS_CTRL_REG, RET_SYSRAM, 1);
        SetBits16(SYS_CTRL_REG, OTP_COPY, 0);
    } else {
        SetBits16(SYS_CTRL_REG, RET_SYSRAM, 0);
        otp_prepare(0x1FC0);
    }
    SetBits16(CLK_16M_REG, XTAL16_BIAS_SH_DISABLE, 0);
    WFI();
    SCB->SCR &= ~(1<<2);
}
else if (sleep_mode == mode_idle)
    WFI();
GLOBAL __INT_START();

```

上图是从 SDK 中的 arch_main.c 中截下并整理的部分代码, 应使用它们取代简单地调用 WFI()。其中的宏和函数在 SDK 的其它部分均有定义。我们假设该设备同时只与一个蓝牙设备连接。

大致过程分如下几步:

1. 关闭中断。
2. 特殊情况处理: 当低功耗时钟为 RCX 时钟时, 为避免温度变化引起的时钟信号漂移, 设定休眠延迟。

3. 调用 RivieraWaves 提供并由 Dialog 修改后的函数 `rwip_sleep()`，获取当前蓝牙模块状态所能允许的休眠模式：当蓝牙模块可停机时，为 `mode_sleeping`，否则为 `mode_idle`。
4. 当蓝牙模块可停机时，根据应用需求选择 Extended Sleep Mode 或 Deep Sleep Mode。否则跳转至 12。
5. 关闭无线电信号。
6. 当蓝牙协议栈内核的非保留堆不为空时，将 Deep Sleep Mode 改为 Extended Sleep Mode 以保留数据，否则标记 `func_check_mem_flag` 以便其它部分查看。
7. 设置 System Control Register 以保证 WFI() 调用时进入休眠。
8. 启动 PAD 锁存器，关闭外设电源。
9. 根据不同的睡眠模式，关闭或保留 SysRAM 电源，准备恢复时 OTP 存储器读入 SysRAM 或关闭 OTP 存储器。
10. 关闭时钟信号，调用 WFI() 进入休眠。
11. 因中断被唤醒时，恢复 System Control Register。跳转至 13。
12. 蓝牙模块不可停机，直接调用 WFI() 停止 CPU 工作。
13. 重新启用中断。

这其中需要注意的是，虽然调用 WFI 时全局中断均被禁用，但 `wfi` 指令本身无条件接收任何中断，因此不会导致无法被唤醒。而若在 `wfi` 前先启用中断，则在启用中断和调用 `wfi` 之间可能有中断的进入，导致不可预料的行为，这是不可行的。

但考虑到 uC/OS II 的设计并不存在一个主循环体，而是各个任务均为一个死循环，显然在每个任务中均人工添加休眠代码是不可行的——这会导致应用开发的不便。因此，我们考虑将上下文切换时当作主循环中两次 `schedule` 间的时机，在此时进行休眠。上下文切换的调用函数位于 `os_cpu_c.c` 中，如下图所示：


```

/*
*****
TASK SWITCH HOOK
*
* Description: This function is called when a task switch is performed. This allows you to perform other
*              operations during a context switch.
*
* Arguments   : none
*
* Note(s)    : 1) Interrupts are disabled during this call.
*              2) It is assumed that the global pointer 'OSTCBHighRdy' points to the TCB of the task that
*                  will be 'switched in' (i.e. the highest priority task) and, 'OSTCBCur' points to the
*                  task being switched out (i.e. the preempted task).
*****
*/
#if (OS_CPU_HOOKS_EN > 0) && (OS_TASK_SW_HOOK_EN > 0)
void OSTaskSwHook (void)
{
    #if OS_APP_HOOKS_EN > 0
        App_TaskSwHook();
    #endif
}
#endif

```

只需在该函数体内加入上述代码，即可完成休眠工作。

唯一的问题在于休眠模式的选择。这一点我们选择先行实现 Extended Sleep Mode，并将会在实际开发初步完成后通过测定功耗来确定是否需要考虑 Deep Sleep Mode 以通过降低效率来进一步降低功耗。

5 创新点

我们小组的工作的主要创新点主要有如下几点：

1. 向 DA14580 芯片中移植 uC/OS

传统上，开发者基于由硬件厂商提供的 SDK 开发应用，这种开发方式有很多弊端。首先，直接将写好的应用烧录至开发板中导致只能运行单一应用，使得板子的功能单一。如果要将多种功能综合于同一应用中，必将大大增加代码的复杂性与开发难度，并且如果应用代码编辑不善极易使得系统资源利用不充分甚至导致各类错误的发生。此外这种开发方式使得扩展和更新应用变得困难并且延长了开发周期。

我们首先将 uC/OS 移植于 DA14580 蓝牙开发板，使其承担进程调度等相关功能。从而使得多应用并行运行成为可能。使得能够充分利用系统资源实现较为复杂的功能。通过 uC/OS 封装一部分底层接口，从而降低了应用开发难度，使得能够较快的实现功能扩展与更新。

2. 休眠功能的实现

当今制约蓝牙与许多嵌入式设备应用的关键点往往在于功耗问题。因此为了尽可能降低功耗，许多此类设备上并未安装完整的操作系统。我们将 uC/OS 移植进 DA14580 后，主要通过尽可能多的进入低功耗休眠模式从而使得平均功耗基本与传统方式相同。因为 uC/OS 的设计中不存在主循环体而原 SDK 通过在循环体中调用相关函数实现。为避免重复在各个应用循环中添加相关调用造成的不便，我们创新性的提出了在上下文切换时处理休眠相关任务的解决方案。并且能够在 Extended Sleep Mode 和 Deep Sleep Mode 中进行合理选择，在功耗与性能之间达到更好的平衡。

3. 移植蓝牙协议栈

DA14580 原设计内部存在一 IP 核提供蓝牙协议栈相关功能，通过将相关代码移植入 uC/OS，可以通过操作系统来完成这些辅助工作。这样一方面能够更加高效的利用资源，同时也有利于后续的更新升级。

参考文献

- [1] `µc/os-iii user manual - µc/os-iii documentation 3.05 - doc`. <https://doc.micrium.com/pages/viewpage.action?pageId=15706203>. Accessed: 2016-3-19.
- [2] Jean J Labrosse. *MicroC/OS-II: the real-time kernel*. CMP Media, Inc., 2002.
- [3] maochengtao. cos-ii 移植到 arm 处理器上的几个要点. <http://m.blog.csdn.net/article/details?id=41709271>.