

无锡谷雨电子有限公司

# TI-RTOS 开发教程

---

CC26xx 平台

2017-06-29

## 目录

更改记录.....	3
第一章 TI-RTOS.....	4
1.1 什么是 TI-RTOS.....	4
1.2 TI-RTOS 组件.....	4
1.3 下载并安装.....	5
1.4 创建 RTOS 工程（IAR）.....	8
1.5 开发资料详情.....	11
第二章 任务.....	13
简介.....	13
2.1 任务类型.....	13
2.2 任务优先级.....	14
2.3 任务状态.....	15
2.4 任务堆栈.....	16
2.5 创建任务.....	17
例 1 创建任务实验.....	20
2.5 删除任务.....	23
例 2 任务删除实验.....	23
2.6 任务抢占.....	24
例 3 任务抢占实验.....	25
例 4 任务优先级更改.....	28
第三章 任务间同步.....	29
3.1 信号量（Semaphores）.....	29
3.1.1 创建/删除信号量.....	30
3.1.2 信号申请与释放.....	31
例 5 利用二值信号量对任务控制.....	32
例 6 利用二值信号进行互斥访问.....	34
3.2 事件（Event）.....	35
3.2.1 创建/删除事件对象.....	37
3.2.2 事件 pend/post.....	38
例 7 事件创建实验.....	38
3.3 队列（Queues）.....	41
3.3.1 队列创建.....	42
3.3.2 队列操作.....	42
3.3.3 遍历队列.....	44
3.3.4 插入删除元素.....	44
例 8 队列实验.....	45
3.4 邮箱（MailBoxes）.....	45
3.4.1 创建邮箱.....	45
3.4.2 发送接收邮件.....	47
例 9 邮箱实验.....	47
第四章 HWI.....	50
4.1 创建 HWI.....	50
4.2 硬件中断嵌套和系统堆栈大小.....	50

4.3 Hwi hooks .....	51
4.4 Register 函数.....	51
4.5 Create/Delete 函数.....	52
4.6 Begin/End 函数.....	52
第五章 SWI .....	52
5.1 创建 SWI 对象.....	52
5.2 SWI 对象优先级及系统堆栈大小 .....	54
5.3 SWI 中断执行 .....	54
例 10 SWI 实验 .....	58
5.4 同步 Swi 函数.....	60
5.5 Swi hooks .....	60
5.6 Register 函数.....	61
5.7 Create/Delete 函数.....	61
5.8 Ready/Begin/End 函数.....	61
例 11 SWI hooks 例子.....	62
第六章 门(Gates).....	67
6.1 基于抢占 Gate 实现.....	68
6.1.1 GateHwi.....	68
6.1.2 GateSwi .....	69
6.1.3 GateTask .....	69
例 12 Gate 临界保护实验 .....	69
6.2 基于信号量 Gate 实现.....	70
6.2.1 GateMutex.....	70
例 13 Gate 对共享资源互斥访问 .....	70
6.2.2 GateMutexPri .....	73
第七章 时间服务.....	73
7.1 Clock .....	73
例 14 Clock 延时实验 .....	76
7.2 Timer .....	78
7.3 Second.....	78
第八章 RTOS 实验例程说明 .....	79
8.1 实验例程汇总.....	79
第九章 APIs 列表.....	80
9.1 Tasks APIs 函数 .....	80
9.2 Semaphore APIs 函数 .....	82
9.3 Event APIs 函数.....	82
9.4 Queue APIs 函数.....	83
9.5 Mailbox APIs 函数.....	84
声明.....	84
附录 1: 联系方式.....	84

# 更改记录

版本	日期	描述
<b>V1.0</b>	20170614	文档初始版本

# 第一章 TI-RTOS

## 1.1 什么是 TI-RTOS

在操作系统没有出来之前，单处理器就只能运行单个任务。这样用户只能独占 CPU，这样带来的缺点是 CPU 利用效率非常低。为了解决这个缺点人们提出了操作系统，这样即使只有一个 CPU 也可以运行多个任务，且每个任务互不影响，感觉上还是独占 CPU。由于这个操作系统的出现，使 CPU 使用效率大大提高。

不同的多任务系统侧重点会有所不同。

对于 PC 用户可以独占一个或多个 CPU。而这类操作系统的调度算法则设计为让用户可以同时运行多个应用程序，而计算机也不会反应迟钝。例如用户可能同时运行一个 word 处理程序，一个 web 浏览器，或者更多，并且期望每个应用程序任何时候都能对输入有足够快的响应时间。

对于服务器操作系统，则是为了支持多用户。这类系统的调度算法侧重于让每个用户公平享用 CPU 时间。

实时嵌入式系统则更侧重于实时性，即任务必须在给定的时间限制内完成。例如危险的错误发生时，如果不能在限定的时间内做出反应，那么就有可能发生严重的后果。

大多数嵌入式系统不仅满足硬实时要求，也能满足软实时要求。这样才能称这实时操作系统。

TI-RTOS 是与其他 RTOS 一样，是对系统资源进行管理与调度。只不过 TI-RTOS 是针对 TI 的可编程器件的一个 RTOS。TI-RTOS 是一个可裁剪，可剥夺，一站式嵌入式工具型的一个实时操作系。它由一个具有多任务管理功能 SYS/BIOS 内核组件，和其它一些辅助组件与设备驱动组成，其 SYS/BIOS 是其主要的，不可或缺的核心组件。SYS/BIOS 负责任务启动，调度，抢占，任务间同步。

由于 TI-RTOS 避免了从头开始创建基本系统软件功能的必要，所以加快了开发步伐。TI-RTOS 可从实时多任务内核（之前称为 SYS/BIOS 的 TI-RTOS 内核）扩展为完整的 RTOS 解决方案，包括附加中间件组件、器件驱动程序和电源管理。通过结合 TI-RTOS 电源管理和 TI 的超低功耗 MCU，开发人员能够设计出电池寿命更长的应用。TI-RTOS 提供经预测试和预集成的必要系统软件组件，使开发人员能够专注于设计最与众不同的应用。

TI-RTOS 构建于经过检验的现有软件组件基础之上，确保了可靠性和质量。除此之外，还提供了适用于多任务开发和集成测试的文档、额外示例以及 API，用于验证所有组件能否协调工作。TI-RTOS 经过 Code Composer Studio™ 集成开发环境 (CCS IDE) 的充分测试。针对一些微控制器平台，TI-RTOS 包含可以与 IAR 嵌入式工作平台 IDE 和 GCC 配合使用的库。

## 1.2 TI-RTOS 组件

TI-RTOS 包含自己的源文件，预编译的库和相应的例子。TI-RTOS 的组件在安装目录下的“products”子目录下。一些组件不是适用所有的 TI 器件。在每个组件下有相应的 doc 目录，里面文档说细说明了当前组件。在下面表格里列举的组件是我们后面将要重点介绍与编程使用的组件。

TI-RTOS 组件	名称	PDF 原文档
TI-RTOS 内核	SYS/BIOS	SYS/BIOS(TI-RTOS Kernel) User's Guide
TI-RTOS 驱动与板级初始化	Drivers,CC26xxWare	--
XDCtools	XDCtools	<Xdctool_install>/docs/目录下

**TI-RTOS Kernel – SYS/BIOS** SYS/BIOS 是一个实时内核。它被具有实时性要求的应用程序使用。SYS/BIOS 具有可抢占的多任务，硬件抽象，实时分析，可工具配置，它对内存与 CPU 要求低。在 CC26xx 内部 Flash 中，固化的 SYS/BIOS 内核。

**Drivers** TI-RTOS 包含了多个外设驱动。驱动在<tirtos\_install>/packages/ti/drivers 目录下。这些驱动具有线程安全特性。

**CC26xxWare** 是 CC26xx 外设的寄存器级别的驱动软件。

**XDCtools** 是一个单独的软件组件，它提供了对 TIRTOS 的配置与编译。XDCtools 提供了 XGCONF 配置编辑和脚本语言对.cfg 文件进行编辑。它也提供了工具对.cfg 文件进行编译。只要你的工程里有.cfg 后缀的文件，XDCtools 都会自动对其进行编译，从而产生源码参与程序的编译与链接。XDCtools 中的软件模块不做具体介绍，在后面的工程中用到的时候，再进行说明。

## 1.3 下载并安装

在使用 TI-RTOS 之前，要在自己的电脑上安装相应的 TI-RTOS 包。这些资源工具的获取，最好在 TI 的官网下载。TI-RTOS 的官网下载地址为 <http://www.ti.com/tool/ti-rtos-mcu>。

在浏览器中输入上述地址并进入 TI 官网。其网站的部分内容截图如下。可能随着时间变化，其网页的内容和布局会有所变化，一切以 TI 官网为准。或者在其官网的搜索框中搜索 **TI-RTOS** 关键字，同样可以找到。

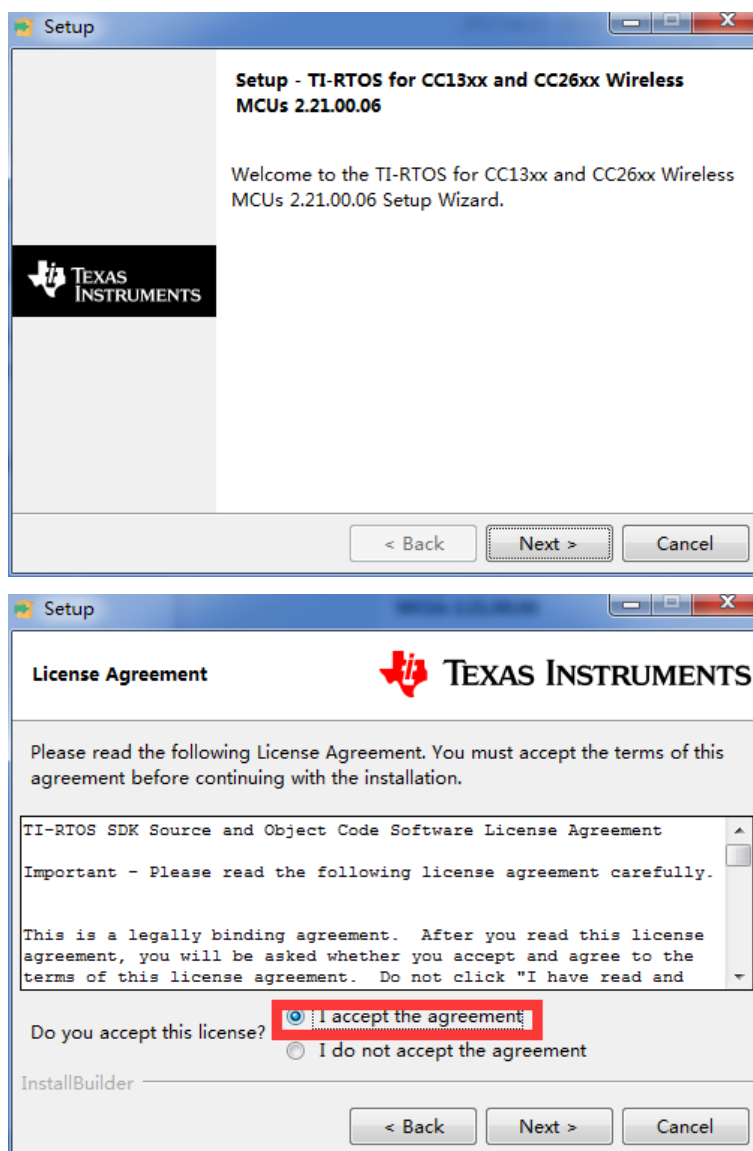
Part Number	Buy from Texas Instruments or Third Party	Alert Me	Status	Current Version
TI-RTOS: Real-Time Operating System (RTOS) provided by Texas Instruments	Free <a href="#">Get Software</a>	<a href="#">Alert Me</a>	ACTIVE	Current and previous versions

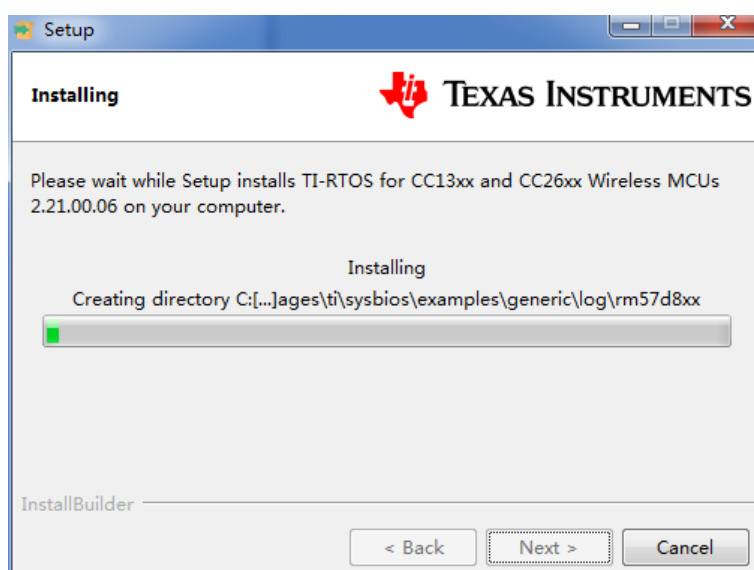
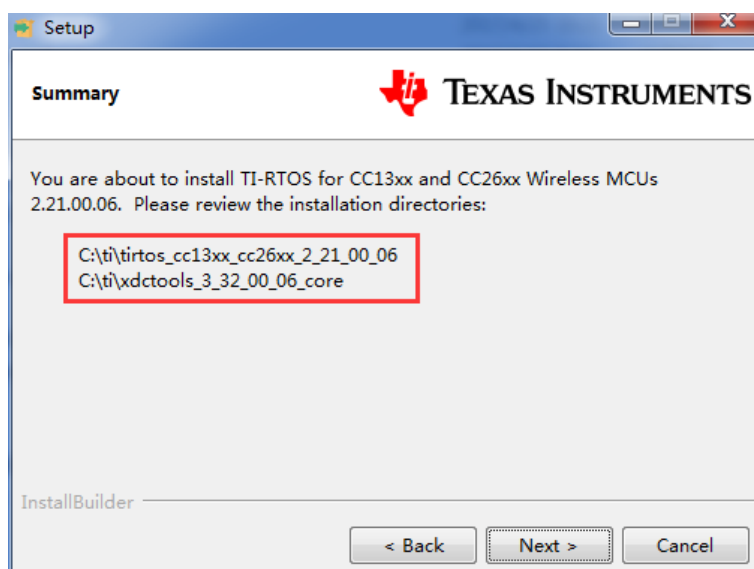
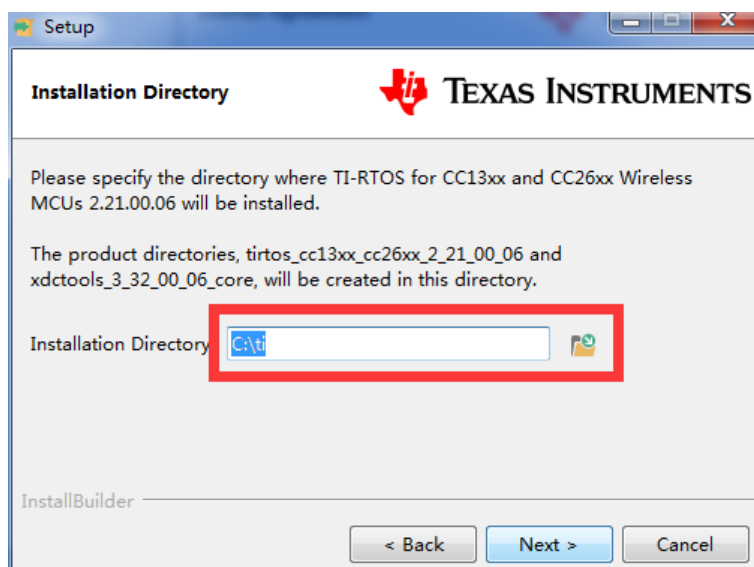
点击 **Get software**，便可进入 TI-RTOS 详细分支页面。

TI-RTOS 2.16.xx and above Product Releases						
Version	Description	C2000	MSP43x	Tiva C	CC13xx/CC26xx	CC32xx
2.21.00.06 13 Sep 2016	See release notes for a list of enhancements and bugs fixed.				<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	
2.20.01.08 20 Oct 2016	Updated Crypto driver, CC26xx driverlib and SYS/BIOS.				<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	
2.20.00.06 22 Jun 2016	See release notes for a list of enhancements and bugs fixed.		<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>		<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	
2.18.00.03 10 Jun 2016	Updated CC13xx/CC26xx DriverLib and updated RF examples generated settings to use RF Studio 2.4.0				<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	
2.16.01.14 22 Apr 2016	Updated CC13xx/CC26xx DriverLib and RF driver. Updated MacOS installers.	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>
2.16.00.08 25 Feb 2016	Release for TI-RTOS products. For use with CCS 6.1.0 and above.	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>	<a href="#">Windows</a> <a href="#">Linux</a> <a href="#">MacOS (Beta)</a> <a href="#">Rel Notes</a>

用户根据自己需求，点击下载相应版本的 RTOS。这里我们以 CC26xx 平台为例，进行 RTOS 的安装。如果用户的 PC 上安装了 BLE 协议栈，可以不用单独下载安装 RTOS。因为在安装 BLE 协议栈时，已经安装了相应的 RTOS 和 XDCTools。

TI-RTOS 是以安装包形式提供，所以用户只要双击.exe 文件，并一直点击**下一步**直至安装完成。强烈建议不要修改安装的路径。下图是安装过程的部分截图。





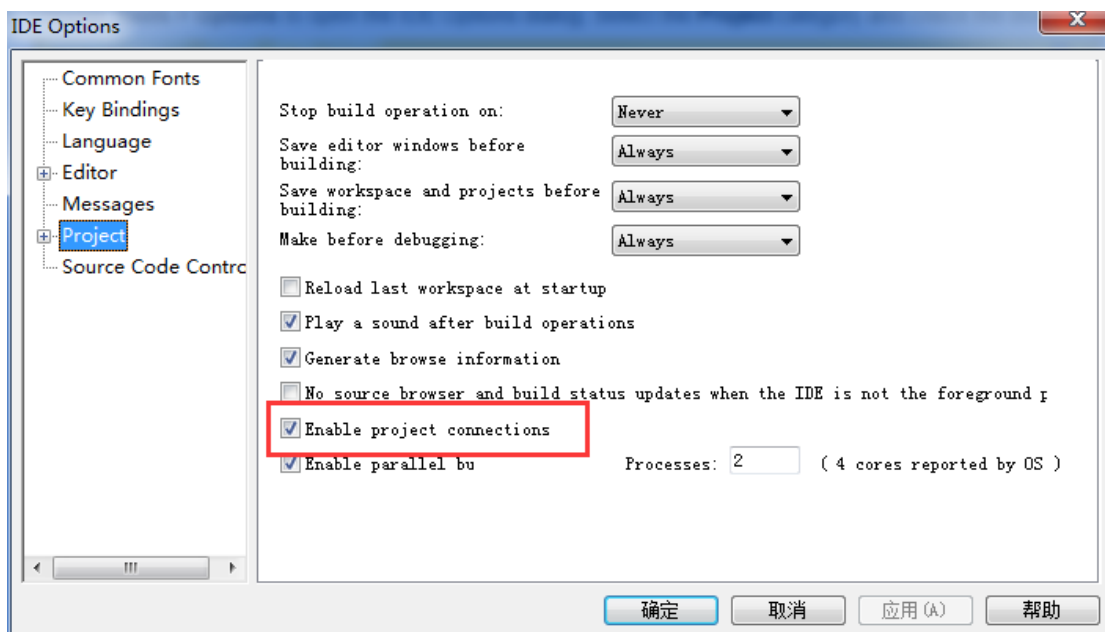


## 1.4 创建 RTOS 工程（IAR）

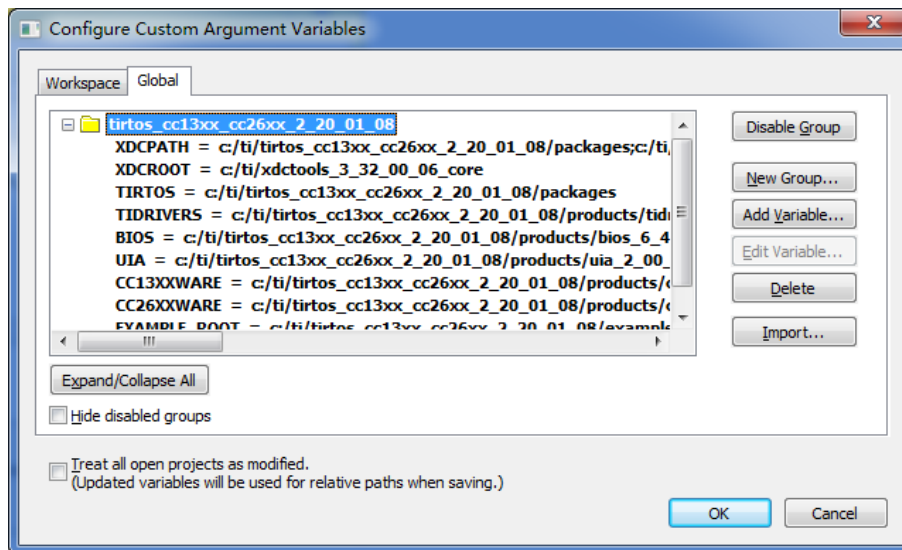
在 TI-RTOS 安装完成之后,便可以 IAR 中运用 TI-RTOS 的创建具体的应用。使用 TI-RTOS, 不像其他的嵌入式操作系统。其它的操作系统大使用之前要进行移植等相应工作, 这些在 TI-RTOS 中已经为我们做好了基础工作, 不需要我们从零开始。这样带来一个限制只能在 TI 相应的 MCU 上运行。由于 TI-RTOS 帮我们做了底的工作, 那么我们在创建工程时, 一定是基于一定的基础之上的。下面将详细说明, 如何在 IAR 中, 创建 TI-RTOS 工程。然后在创建的工程的基础上进行用户应用逻辑修改。

为了使 IAR 能够发现 TI-RTOS, 用户必须做以下操作。

1. 打开 IAR Embedded Workbench
2. 点击 IAR 菜单栏上的 **Tools**, 选择 **Options**, 打开 IDE Options 对话框。点击 **Project**, 并将 **Enable project connections** 勾选, 然后点击 OK。

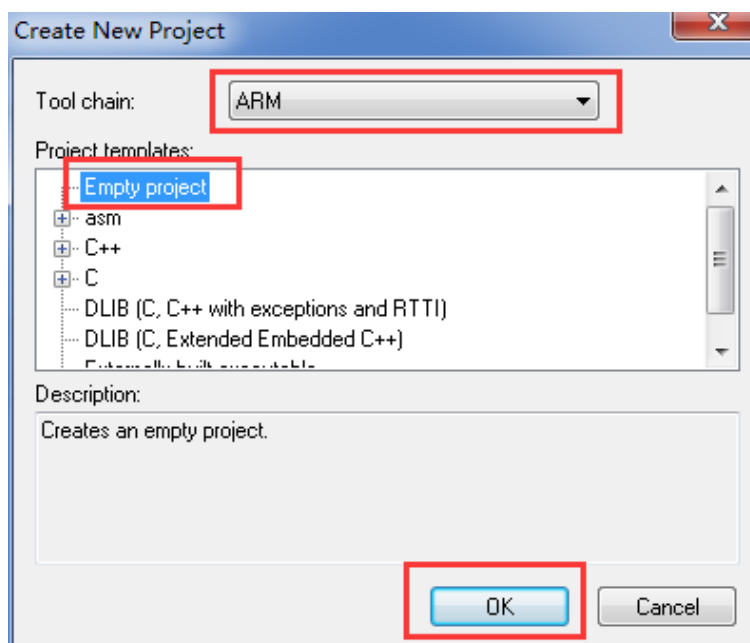


3. 选择 Tools 菜单下的 **Configure Custom Argument Variables**。这时打开了一个对话框, 这个对话框将允许用户定义路径, 将 TI-RTOS 与 IAR Embedded Workbench 进行整合。
4. 在对话框中选择 **Global** 页, 并点击 **Import** 按钮。
5. 对打开对话框中将路径定位到 TI-RTOS 的例子 IAR 子目录下, 先择参数变量文件。即 TI 安装目录下的 `\tirtos_cc13xx_cc26xx_xx_xx_xx_xx\examples\IAR\tirtos_cc13xx_cc26xx_xx_xx_xx.custtom_argvars` 文件, 然后点击**打开**按钮。
6. 此时 **Configure Custom Argument Variables** 对话框中, 就会出现类似于环境变量信量。

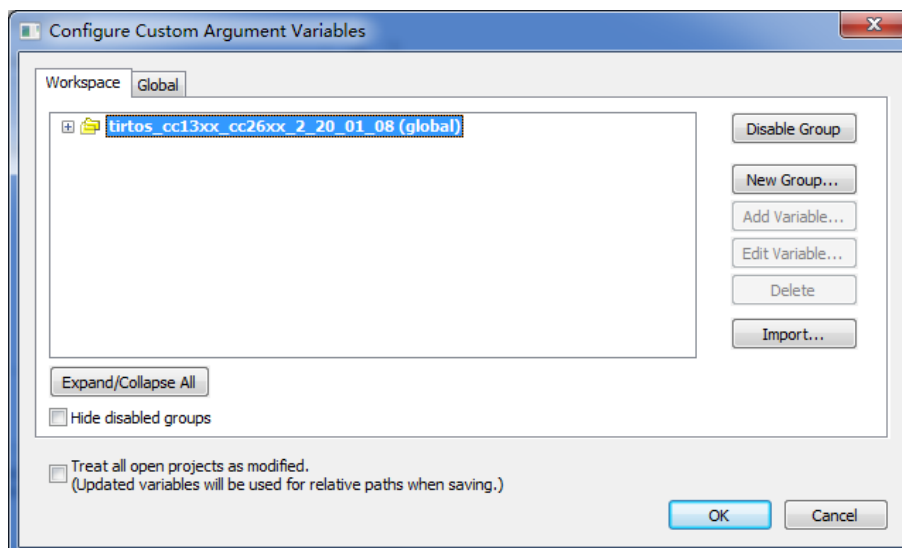


用 IAR Embedded Workbench 创建 TI-RTOS 工程例子。

1. 在 IAR 中创建一个新工程
2. 选择正确的 **Tool chain** 和 **Empty project**，并点击 OK。



3. 选择指定的目录，保存工程文件。（注意：路径不能包含空格）
4. 保存工作空间，**File->Save Workspace**。
5. 确定自定义参数对这个工程是有效的。选择 **Tools->Configure Custom Argument Variables**。在 **Configure Custom Argument Variables** 对话框，上面步骤导入的自定义变量在 **Workspace** 页下有效。



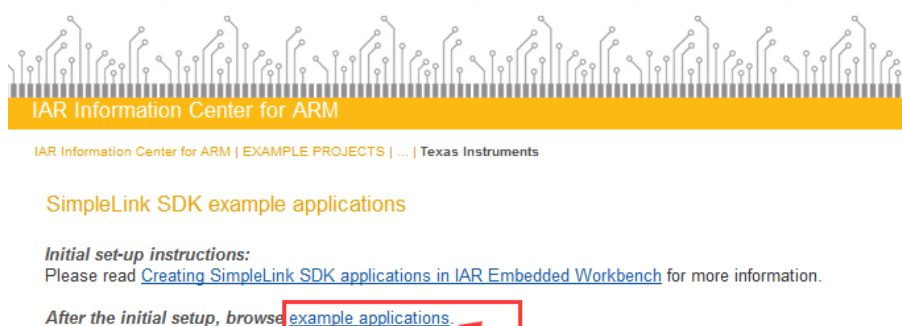
6. 点击 IAR 工程 **Information Center** 中 **Integrated Solution**。如果 Information Center 没有打开，可以在 Help -> Information Center 打开。



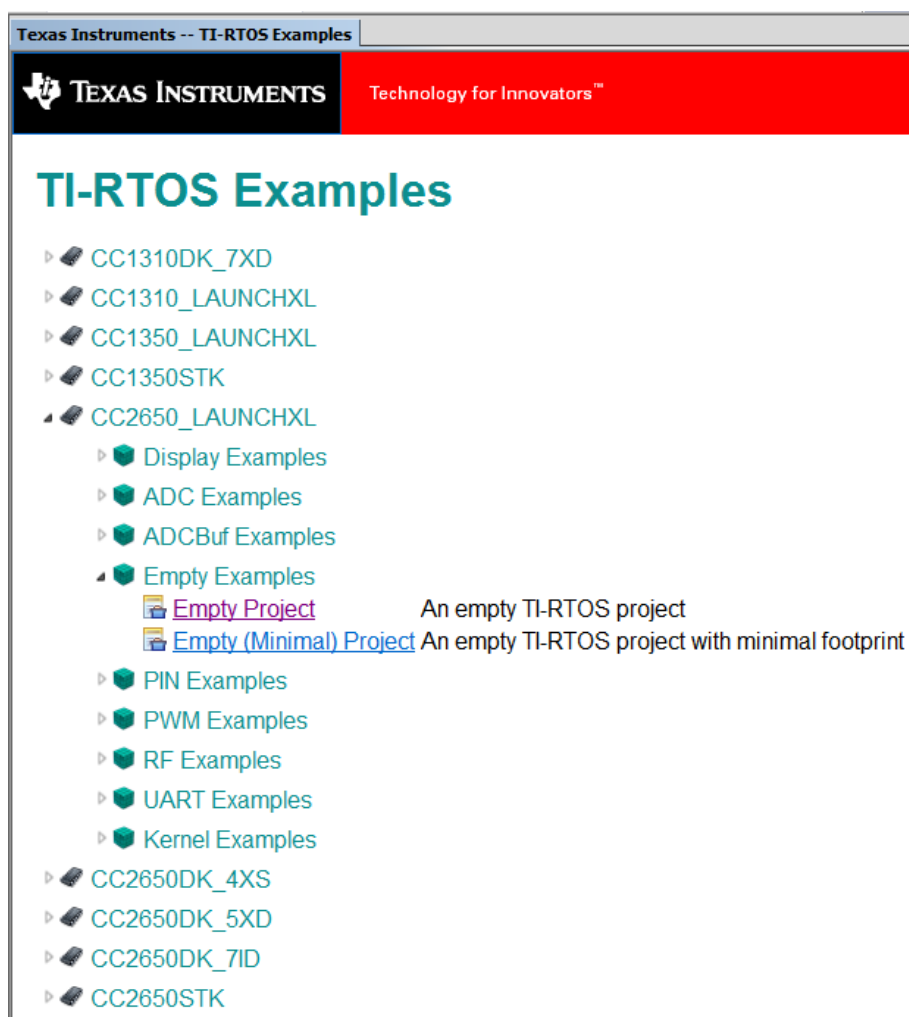
7. 在 Partners 中，选择 Texas Instrument，点击 Example projects。



8. 点击 example application，打开 Texas Instrument 的 TI-RTOS 例子页。



9. 展开例子列表。点击一个例子将例子工程添加到自己的工程中。下图展示了 CC26xx/CC13xx 例子。



通过上述步骤的操作，一个基于 TI-RTOS 的工程便可创建完成。用户只要在工程的基础这上进行修改和添加自己的逻辑即可。下面的章节将详细介绍，TI-RTOS 的内容，并进行相应的例子实验编写，将帮助用户了解 TI-RTOS。

## 1.5 开发资料详情

最新开发资料下载连接：<http://bbs.iotxx.com/thread-6-1-1.html>

资料根目录截图如下，套件资料仍然处理快速完善中，因此截图与实际内容可能有出入。

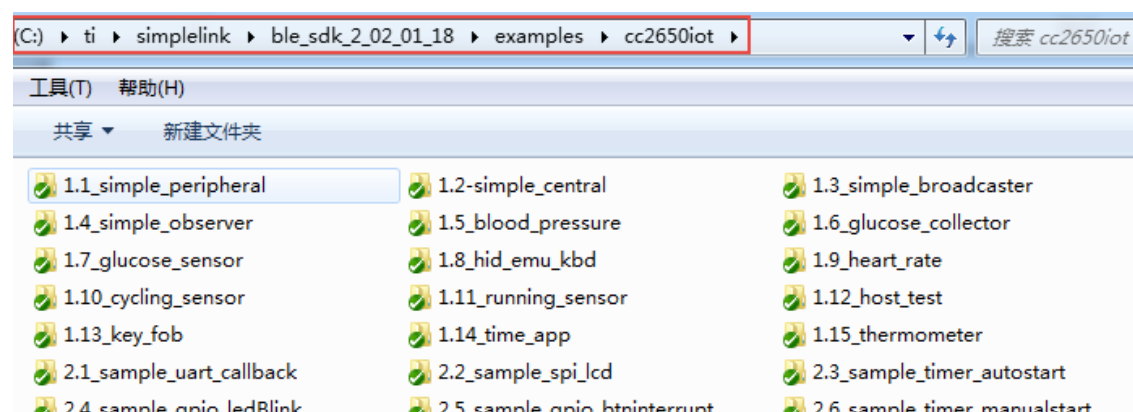
名称	修改日期	类型
0-套件软硬件资料	2017/8/14 14:49	文件夹
1-蓝牙入门教程	2017/8/14 14:49	文件夹
2-硬件外设教程	2017/8/14 14:49	文件夹
3-RTOS操作系统教程	2017/8/14 14:49	文件夹
4-协处理器教程	2017/8/14 14:49	文件夹
5-视频教程	2017/8/14 14:49	文件夹
A-##首先读我##.txt	2017/8/14 14:49	文本文档
B-快速用户手册.pdf	2017/8/14 14:49	看图王 PDF 文件
C-联系我们.pdf	2017/8/14 14:49	看图王 PDF 文件
D-访问官网.html	2017/8/14 14:49	HTML 文件
E-访问官方店铺.html	2017/8/14 14:49	HTML 文件
F-访问开发者论坛.html	2017/8/14 14:49	HTML 文件

**0-套件软硬件资料：**开发过程中的所有软硬件资料，例如开发板原理图，协议栈 sdk，开发板 sdk，以及开发环境 IAR、烧写软件等。

子目录截图如下：

名称	修改日期	类型
0-协议栈sdk	2017/8/14 14:51	文件夹
1-开发板sdk	2017/8/14 14:50	文件夹
2-软件工具	2017/8/14 14:51	文件夹
3-硬件原理图	2017/8/14 14:49	文件夹
4-相关手册	2017/8/14 14:49	文件夹
5-其他资料	2017/8/14 14:49	文件夹

协议栈 sdk 由 TI 提供的 exe 可执行文件，安装到默认路径，开发板 sdk 是由我们提供的基于开发板的例程压缩包，需要解答到协议栈 sdk 安装后的得到的 example 目录下。解压后得到如下目录结构，压缩包解压时请注意不要产生两个 cc2650iot 目录层次。



**1-蓝牙入门教程：**我们精心编写的《CC2640 低功耗蓝牙入门教程》，教程内容近 200 页，教程中提到的协议栈自带例程，已经统一打包在：开发板 sdk 目录下 cc2650iot 压缩包中。

**2-硬件外设教程：**《CC2640 外设教程》，几乎涵盖了 CC2640 系列芯片的所有外设编程。教程中涉及的例程，已经统一打包在：开发板 sdk 目录下 cc2650iot 压缩包中。

**3-RTOS 操作系统教程：**《TI-RTOS 实时操作系统教程》，教程中涉及的例程，已经统一打包在：开发板 sdk 目录下 cc2650iot 压缩包中。虽然 TI-RTOS 例程独立于蓝牙协议栈，但由于例程中不能包含中文路径，因此也统一打包在：开发板 sdk 目录下 cc2650iot 压缩包中。

**4-协处理器教程：**《SensorController 协处理器教程》，教程中涉及的**蓝牙协议栈例程**，已经统一打包在：开发板 sdk 目录下 cc2650iot 压缩包中。注意，SensorController Studio 的示例工程（可以理解成 SensorController 协议处理的程序编译软件）直接存放在该目录下，软件支持中文路径，可直接打开工程开发。详情请参考《SensorController 协处理器教程》

**5-视频教程：**开发板视频教程，目前正在紧张录制中，预计 9 月 1 日全部完成！

## 第二章 任务

### 简介

人们在实际生活中处理一个大而复杂问题时，常用的方法就是把一个大问题分解成多个相对简单，比较容易解决的小问题。如果小问题逐个解决了，大问题也就随之解决了。同样，在设计一个较为复杂的程序时，也通常把一个大型任务分解成多个小任务，然后在计算机中通过运行这些小任务，最终达到完成大任务的目的。

在 TI-RTOS 中，与上述小任务对应的程序实体我们称之为“任务”，TI-RTOS 就是对这些小任务进行管理和调度的程序。在 C 代码中，任务就是内部含用“死循环”的一个函数。

TI-RTOS 是多任务，可剥夺实时操作系统。当任务被切换后，以便继续运行，就需要一个内存空间来保存现场环境。当被切换任务获得 CPU 权限时，可以恢复之前运行环境，继续执行。

当然，TI-RTOS 在进行任务调度时，需要一定依据。在 TI-RTOS 中，采用任务优先级方法对任务进调度控制，高优先级的任务优先运行，也可以抢占低优先级任务。这样采能保证 TI-RTOS 优先处理紧急任务。

通过上述说明，在 TI-RTOS 系统中，任务包含任务程序代码，任务堆栈和任务优先级三个主要部分。

### 2.1 任务类型

TI-RTOS 的内核 SYS/BIOS 定义了多个不同类型的任务，且任务的优先级也不同。每种类型任务都是不同执行方式和抢占特性。下面列出的 SYS/BIOS 支持的任务类型（优先级从高到低）：

- 硬中断（HWI），包括 Timer
- 软中断（SWI），包括 Clock

- 用户创建的任务 (TASK)
- 空闲任务 (IDLE)

HWI 也称为中断服务例程或 ISR，在 SYS / BIOS 应用程序中具有最高优先级的线程。Hwi 任务用于执行时间受到严格的限期的关键任务。在实时环境中，它是响应外部触发的异步事件（中断）。详细的硬件中断说明，将在后面进行说明。

SWI 也称为软件中断服务例程。在 SYS/BIOS 中，SWI 任务优先级在 HWI 任务和用户任务之间。不像 HWI 由硬件中断触发，SWI 是通过调用 SWI 模块 API 方式触发。SWI 允许 HWI 将较少的关键处理推迟到较低优先级的任务中，这样可以最小化 CPU 在中断服务程序中花费的时间。SWI 需要足够的空间来保存每个 SWI 中断优先级的上下文。

TASK 用户任务，此类型的任务优先级高于系统空闲任务，低于 SWI。TASK 不同于 SWI，它们在运行过程中，可以被阻塞直到必要的资源可用。TASK 要求每个任务要有自己独立的栈空间。SYS/BIOS 提供了一些机制用于 TASK 间的同步和通信，它们有信号量 (Semaphore)，事件 (Events)，队列 (Queue) 和邮箱 (Mailboxes)。

系统任务之空闲任务。空闲任务运行在最低优先级下。它于用户任务没有什么区别，只是它由 TI-RTOS 自己创建。当 SYS/BIOS 没有比空闲任务更高的任务运行时，SYS/BIOS 就会执行空闲任务，并且是连接执行，直到有更高的任务进入就绪状态。

## 2.2 任务优先级

TI-RTOS 是一个可剥夺型操作系统，其任务的优先级是任务调度算法的基准。所以任务优先级是比较重要的。

在 SYS/BIOS 中，HWI 有最高优先级。其优先级不能被 SYS/BIOS 维护，它是 CPU 特性决定。硬件中断也可以被其也的中断抢占，除非全局中断禁用或特定的中能被单独禁用。

SWI 的优先级低于 HWI。它有优先级可以达到 32 级，默认只有 16 级。SWI 可以被更高优先级的 SWI 和 HWI 抢占，但是 SWI 是不会阻塞的。

Task 的优先级低行 SWI。它的优先级也可以达到 32 级，默认是 16 级。Task 优先级数值越大，其优先级就越高，0 是优先级最低的。Task 可以被更高优先级 Task 抢占。当 Task 等待资源时，Task 可以被阻塞。Task 的优先级可以通过 Task\_setPri 函数动态的更改。新的优先级必须在 1 到 TnumPriorities - 1 之间。当新优先级数低于当前优先级时，可能会发生任务切的换。

空闲任务优先级在所有任务中是最低的。当 CPU 不忙时，空闲任务就会被执行。其运行在 Task 任务 0 优先级上。

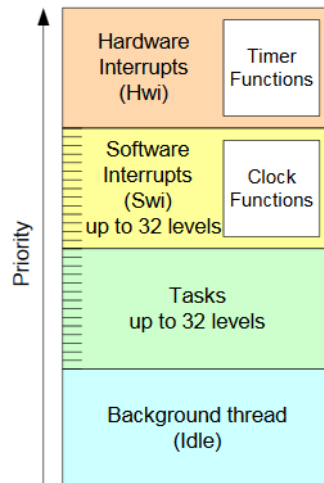


图 1 任务优先级

## 2.3 任务状态

SYS/BIOS 中 Task 有五种状态，分别 RUNNING, READY, BLOCKED, TERMINATED, INACTIVE 五种。

- **RUNNING**

SYS/BIOS 正在执行的状态

- **READY**

任务处于就绪状态，等待 SYS/BIOS 调度。

- **BLOCKED**

任务处在阻塞状态，即任务不能被执行，可能等待个别事件发生。

- **TERMINATED**

任务处于终断状态，即不能被 SYS/BIOS 再次调用

- **INACTIVE**

任务非激活状态，即任务的优先级等于-1。任务刚被创建时就处于这个状态。可以调用 Task\_setPri() API 进行设定。

任务被调度执行是根据任务的优先级。同一时间只能有一个任务处于 RUNNING。非就绪状态任务优先级大于当前 RUNNING 状态任务优先级，只要此任务处于 READY 状态就会抢占当前任务。

最大优先级是 Task\_numPriorities-1（默认=15，最大=31）。最小优先级是 1。如果优先级小于 0，此任务将会被禁止执行直到其他任务将其优先级提高大于 0。如果任务的优先级大于 Task\_numPriorities-1，此任务不会被其他的 Task 任务抢占。但此任务仍然可以调用 Semaphore\_pend(), Task\_sleep()或其他一些带有阻塞特性的 API 来阻塞自己，这样可以使低优先级任务有机会去运行。一个任务的优先级也可以通过调用 Task\_setPri()进行更改。

一个任务状态的改变可能有多种原因。



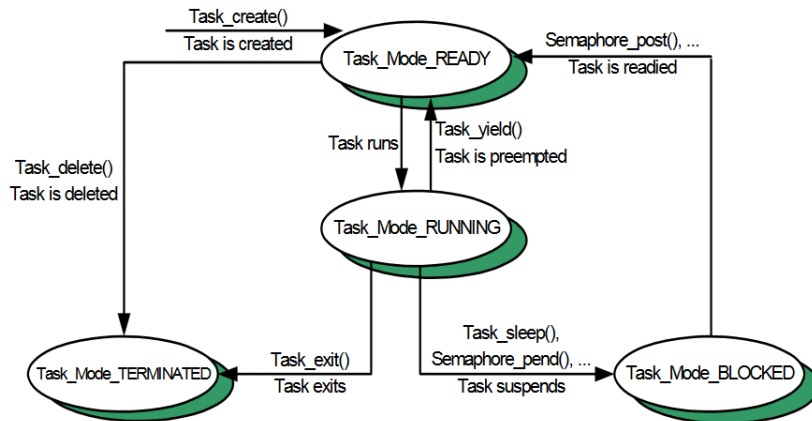


图 2 Task 状态变化图

任务中的函数，Semaphore，Event 和 MailBox 都有可能更改任务的执行状态。阻塞或终止当前执行任务，就绪之前被挂起的任务，重新调度当前任务等等。

在 SYS/BIOS 有且仅有一个任务处于 Task\_Mode\_RUNNING。如果所有 Task 都被阻塞或没有 HWI 和 SWI 运行，SYS/BIOS 将会执行 Task\_Idle 空闲任务。当一个任务被 HWI 或 SWI 抢占，通过调用 Task\_state() 返回的状态还是 Task\_Mode\_RUNNING，因为 SWI 或 HWI 释放之后这个任务还将继续执行。需要注意的地方就是，在空闲任不要执行带有阻塞功能函数，这有可能导致应用程序终端。

当一个 Task\_Mode\_RUNNING 状态任务，变更到另外三个状态，可以控制 SYS/BIOS 进行任务切换任务，去运行就绪状态任务中具有最高优先级的任务。其方法有以下几种。

- 使运行的任务变成 Task\_Mode\_TERMINATED，可以通过调用 Task\_exit()。
- 使运行的任务变成 Task\_Mode\_BLOCKED，可以调用带有阻塞特性函数。例如 Semaphore\_pend() 或 Task\_sleep()，它们有可能挂起当前任务。
- 使运行的任务变成 Task\_Mode\_READY，可以使更高优先级任务变成 READY 状态并执行，便可抢占当前 Task。调用 Task\_setPri() 函数使当前任务降低优先级。也可以调用 Task\_yield() 去放弃当前 CPU 时间，使同优先级任务得到运行。

## 2.4 任务堆栈

任务堆栈是任务重要的组成部分。

所谓堆栈，就是在存储器中按数据 LIFO（后进先出）的原则组织的连续存储空间。为了满足任务切换和响应中断时保存 CPU 寄存器中内容及任务调用其他函数所需要，每个 Task 都应该配有自己的堆栈。

当任务使用堆栈产生溢出，系统将不可预测，存在致命后果。因此对任务堆栈溢出检查就显得非常有意义。

在 SYS/BIOS 里提供函数 Task\_stat()，可以对任务堆栈空间进行观察。此函数返回的结构中包含任务堆栈空间大小和栈空间已使用了大小。见下面代码说明。

```

Task_Stat  statbuf;          /* declare buffer */
Task_stat(Task_self(), &statbuf); /* call func to get status */
If (statbuf.used > (statbuf.stackSize * 9 / 10))
{
    System_printf("Over 90% of task's stack is in user.\n");
}
  
```

TI-RTOS 要求任务最小的堆栈空间为 512 个字节。TI-RTOS 有两种方式对任务进行堆栈分配，第一手动创建数组分配给任务当任务堆栈，第二由任务创建函数动态分配内存，其大小由 `Task_Params.stackSize` 成员变量指定。两种分配方式区别是，创建数组方式其内存是应用程序的堆中分配；而动态分配的内存是在 TI-RTOS 系统堆栈空间中。由于链接文件为 TI-RTOS 分配的堆空间有限，过多的动态分配，将会占用过多的内存空间，而导致其他组件因不能分配到内存而失败。在实际使用过程中，推荐使用手动创建方式为任务分配堆栈空间。

手动创建数组作为任务堆栈方式如下：

```
Char task0Stack[512];  
.....  
taskParams.stackSize = 512;  
taskParams.stack = &task0Stack;  
  
Task_construct(&task0Struct, (Task_FuncPtr)heartBeatFxn, &taskParams,  
NULL);
```

## 2.5 创建任务

根据嵌入式系统任务的工作特点，任务的执行代码通常是一个无限循环结构，当然对于只需执行一次的任务不是这样。同时在这个结构中响应中断。下面给出示例。

```
void UserTask(Uarg a0,Uarg a1)  
{  
    for(;;) //相当于 while(1)  
    {  
        //可以被中断的代码  
        ...  
        ...  
        gateKey = GateHwi_enter(gateHwi);  
        //临界区，不可以被中断代码  
        ...  
        ...  
        GateHwi_leave(gateHwi, gateKey);  
        //可以被中断的代码  
        ...  
        ...  
    }  
}
```

从 C 程序来看，TI-RTOS 的任务就是一个 C 函数。为了可以传递参数，TI-RTOS 的任务函数定义了两个参数，这个参数可以被转换成其他类型数据，包括函数指针。

代码中 `GateHwi_enter` 和 `GateHwi_leave` 是 TI-RTOS 的 `Gates` 的一种，用于保护临界区代码。`Gates` 还有其他类型，后面将一一介绍。这两个函数必须成对使用，否则将导致不可预测后果。在两个函数之间的代码是不会被中断的。

从程序代码的形式上来看，用户任务就是一个 C 函数，但这个函数不是由主函数 `main()`

调用的函数，它们何时被调用又何时被中止是由 TI-RTOS 系统调度的。`main()`是 C 程序的主函数，是程序运行的入口函数，但在其内部可以创建任务并将他们交给系统，由系统负责管理。基于 TI-RTOS 的应用程序代码大体上如下。

```
Task_Handle pTask1Handle;
Task_Handle pTask2Handle;
Task_Handle pTask3Handle;
void MyTask1(Uarg a0, Uarg a1)    //任务 1 的任务函数
{
    //相关初始化代码
    ...
    ...
    for(;;)
    {
        .....
    }
}
void MyTask2(Uarg a0, Uarg a1)    //任务 2 的任务函数
{
    //相关初始化代码
    ...
    ...
    for(;;)
    {
        .....
    }
}
void MyTask3(Uarg a0, Uarg a1)    //任务 3 的任务函数
{
    //相关初始化代码
    ...
    ...
    for(;;)
    {
        .....
    }
}
//主函数
void main()
{
    Task_Params task1Params;
    Task_Params task2Params;
    Task_Params task3Params;
    Error_Block eb1;
    Error_Block eb2;
```

```

Error_Block   eb3;
Task_Param_init(&task1Params);    //初始为默认值
Task_Param_init(&task2Params);
Task_Param_init(&task3Params);
Error_init(&eb1);
Error_init(&eb2);
Error_init(&eb3);

.....

.....

pTask1Handle = Task_create((Task_FuncPtr)MyTask1,
    &task1Param, &eb1);
pTask2Handle = Task_create((Task_FuncPtr)MyTask2,
    &task2Param, &eb2);
pTask3Handle = Task_create((Task_FuncPtr)MyTask3,
    &task3Param, &eb3);
BIOS_start()
}

```

其中 `Task_create()` 是 TI-RTOS 提供的用来创建任务函数; `BIOS_start()` 启动系统, 系统被启动之后, 任务就由操作系统来管理和调度。

#### 1. `Task_Handle Task_create(Task_FuncPtr fxn, const Task_Param *params, Error_Block *eb);`

上述是创建用户任务的一种函数, 是系统提供的 API。

**fxn** – Task Function 是用户的任务函数, 与上述的 `MyTask` 函数相一致。

**params** – 创建任务的配置参数, 也可以设置为 `NULL`。 `NULL` 表示使用默认参数。

**eb** – 错误处理结构。如果为 `NULL` 表示选择默认。

返回值

- 任务对象句柄 `Task_Handle`  
如果创建任务成功, 将返回任务对象句柄
- `NULL`  
创建失败, 返回 `NULL`, 并中止执行
- 详细说明

**fxn** 参数是使用 `Task_FuncPtr` 类型的函数指针传递给任务去运行。如果 **fxn** 是用户自己去编写, 你的 C 代码在创建任务对象之后去调用这个函数, 则可以遵行下面的代码调用方式。

```

Task_Params    taskParams;

//Create task with priority 15
Task_Params_init(&taskParams);
taskParams.stackSize = 512;
taskParams.stack = &taskStack;
taskParams.priority = 15;
Task_create((Task_FuncPtr)myFxn, &taskParams, &eb);

```

如果是在配置文件(.cfg)中, 使用静态方式去创建任务。

```
var params = new Task.Params;  
params.instance.name = "task0";  
params.stackSize = 512;  
params.priority = 15;  
Task.create('&myFxn',params);
```

## 2. void Task\_construct(Task\_Struct \*structP, Task\_FuncPtr fxn, const Task\_Params \*params, Error\_Block \*eb);

**structP** – 是任务控制块结构的指针, 所以在使用此函数进行任务创建时必须先创建 Task\_Struct 变量。

**fxn** – Task Function 是用户的任务函数。

**params** – 创建任务的配置参数, 也可以设置为 NULL。NULL 表示使用默认参数。

**eb** – 错误处理结构。如果为 NULL 表示选择默认。

返回值—none

➤ 详细说明

Task\_construct 函数与 Task\_create 函数在使用上的区别, 只是多了一个先创建 Task\_Struct 结构体变量, 其他没有什么区别。

```
Task_Params    taskParams;  
Task_Struct    taskStruct;  
  
//Create task with priority 15  
Task_Params_init(&taskParams);  
taskParams.stackSize = 512;  
taskParams.priority = 15;  
Task_construct(&taskStruct, (Task_FuncPtr)myFxn, &taskParams,  
&eb);
```

Task\_Handle 与 Task\_Struct 可以相互转换, 所使用的函数如下。

```
Task_Handle Task_handle(Task_struct *structP);  
Task_Struct *Task_struct(Task_Handle handle);
```

了解了 TI-RTOS 的任务创建方法之后, 便可以在应用程序中创建任务。一般来说, 任务可在调用函数 BIOS\_start()启动任务调度之前来创建, 也可在任务中来创建。但是在嵌入式操作系统中, 在调用启动任务函数 BIOS\_start()之前必须已经创建了至少一个任务。特别注意, TI-RTOS 中不能在中断函数中创建任务。

## 例 1 创建任务实验

本例演示了不同方式创建任务的必要步骤。这个两个任务周期性的控制 LED 闪烁, 并通过 UART 向终端输出字符串。两个任务在创建时指定相同优先级。下面程序清单分别对两种方式创建任务。一种是采用静态方式创建 MyTask1, 一种采用动态方式 (Task\_create()) 创建 MyTask2。具体代码工程见 3.1\_Task\_Create/目录下。

例 1.cfg 文件创建任务程序清单

```
var params = new Task.Params();
```

```
params.priority = 1;
params.arg0 = 1000000/Clock.tickPeriod;
Program.global.userTask = Task.create('&MyTask1',params);
```

在例子中，使用.cfg 配置文件创建任务，只是为了演示.cfg 文件中也可以创建任务且与动态创建的任务一样，没有什么不同，但在.cfg 文件中创建的任务不能被删除的。强烈建议创建任务使用动态方式创建，即使用函数 `Task_create()` 或 `Task_construct()`。因为在.cfg 文件中编写指令太容易出错，且操作不友好。在后面的例子代码中，所有任务将使用动态方式创建。

#### 例 1 动态创建任务程序清单

```
void CreateUserTask(void)
{
    Task_Params taskParams;

    /* Construct heartBeat Task thread */
    Task_Params_init(&taskParams);
    taskParams.arg0 = 1000000 / Clock_tickPeriod;
    taskParams.stackSize = TASKSTACKSIZE;
    taskParams.stack = &task0Stack;
    taskParams.priority = 1;
    Task_create((Task_FuncPtr)MyTask2, &taskParams, NULL);
}
```

图 1 例 1 程序运行

从图 1 上可以看出，两个任务在同时运行，但是它们运行在同一个处理器上，所以它们不是真正意义上的同时运行。事实上，这两个任务在进入运行状态后，从串口打印信息后便退出运行进入阻塞状态，让出 CPU 的控制权。再让出 CPU 控制权之后，BIOS 此时会运行具有最高优先权就绪任务，此时 MyTask2 进行运行状态。

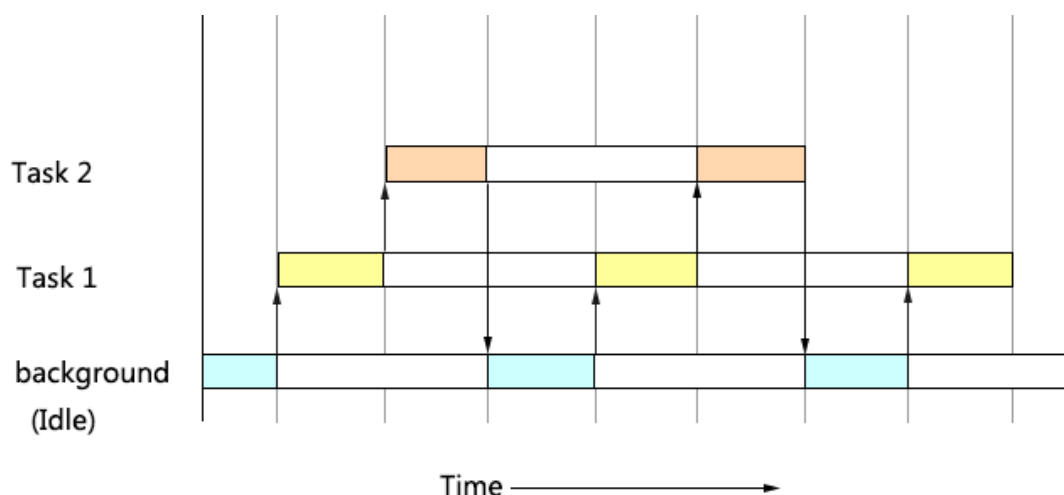


图 2 任务运行时序图

## 2.5 删除任务

删除一个任务,就是把任务置于 **TERMINATED** 状态。在 TI-RTOS 中,可以通过 `Task_delete()` 函数来删除动态任务。如果调用删除任务成功,被任务占用的内存与堆栈将会被系统回收,同时将任务从内部的队列中去除。

删除任务时,任务不要占用任何信号量或其他资源,否则将会导致应用程序不可预测的问题。所以在删除任务之前,一定要将所有占用的资源都释放掉。

### 1. `Void Task_delete(Task_Handle *handleP);`

`handleP` – 是指向任务实例指针

此删除函数是针对使用 `Task_Create` 函数创建的任务

### 2. `Void Task_destruct(Task_Struct *structP);`

`structP` – 是指向任务 `Task_Struct` 结构指针

此删除函数是针对使用 `Task_construct` 函数创建的任务

其例子代码见 `example/Ex_Task_Create/` 目录下。`MainTask` 任务是在 `main` 中被创建,在其被 `SYS/BIOS` 调用之后,运行一定代码之后,便删除自己。

## 例 2 任务删除实验

本例实验在例 1 实验基础上创建一个最高优先级任务。在此任务启动之后,对串口、显示屏驱动进行初始化,并创建一个用户任务。在完成相应工作之后,任务自己将调用 `Task_delete` 函数将自己删除,即将自己从 **RUNNING** 状态置成 **TERMINATED** 状态。具体代码工程见 `3.2_Task_Del/` 目录下。

```
void CreateMainTask(void)
{
    Task_Params taskParams;
    /* Construct heartBeat Task thread */
    Task_Params_init(&taskParams);
    taskParams.priority = Task_numPriorities - 1;
    Task_construct(&mainTaskStruct, (Task_FuncPtr)MainTask, &taskParams,
    NULL);
}
```

最高优先级任务函数

```
Void MainTask(UArg arg0,UArg arg1)
{
    UARTDriverInit();
    //open the LCD
    dispHandle = Display_open(Display_Type_LCD,NULL);

    //display the title
    Display_print0(dispHandle,0,4,"TI-RTOS ");
    Display_print0(dispHandle,1,0,"Create Task");
}
```



```
//init the com
CreateUserTask();

//del ownself
Task_destruct(&mainTaskStruct);
}
```

## 2.6 任务抢占

SYS/BIOS 是 TI-RTOS 的内核，它调度任务是根据就绪任务的最高优先级进行的。即它任何时候只执行最高优先级任务。但也有例外的情况，下面列出几种情况。

- 当前正在执行的任务，通过调用 `Hwi_disable()`或 `Hwi_disableInterrupt()`禁用一些或所有硬中断，阻止中断服务程序执行。
- 正在运行的任务，通过 `Swi_disable()`临时禁用软中断。这将阻止具有高优先级的 SWI 去抢占当前任务。但是它不能阻止硬中断去抢占当前任务。
- 正在运行的任务，通过调用 `Task_disable()`函数临时禁止 SYS/BIOS 任务调度。这将阻止任何已经就绪的高优先级任务抢占当前任务。但是它无法阻止 Hwi 和 Swi 抢占当前任务。
- 如果低优先级任务与高优先级任务共享 Gates 资源，且低优先级任务已经占用了 Gates，那么高优先级任务再申请 Gates 就会被阻塞，只能等到低优先级任务释放 Gates 资源后，高优先级任务才会运行。这种现象被称为优先级反转。

Hwi 和 Swi 与 SYS/BIOS 的任务调度器相互作用，相互影响。任务经常因为在请求一个被占用的信号量时而被阻止。但这个信号量可以在 Hwi 或 Swi 的服务程序中被释放，也可以在任务中被释放。如果信号量在 Hwi 或 Swi 中被释放，那么被阻塞的任务就会在 Hwi 或 Swi 服务程序执行完成之后，且成为最高优先级就绪任务时，被 SYS/BIOS 切换为当前执行任务。

在 SYS/BIOS 内核中也可以通过调用 `Task_sleep()`或 `Task_yield()`放弃当前 CPU 控制权，此时 SYS/BIOS 就会执行一次任务切换。

### 1. UInt Task\_disable();

返回值要进行保存，用于后序恢复使用。

#### ➤ 详细说明

此函数是禁止 SYS/BIOS 调度器进行任务切换，相当于给调度器加上了锁。此时任何高优先级任务都无法进行抢占，但是 HWI 和 SWI 仍可以抢占。`Task_disable` 与 `Task_restore` 成对使用。两函数可以保证它们之间的代码可以一起执行，而不被高优先级任务打断。`Task_disable()`返回值需要传递给 `Task_restore()`进行恢复。示例代码如下。

```
Key = Task_disable();
// 临界区代码
...
...
Task_restore(key);
```

### 2. Void Task\_restore(UInt key)

key – 恢复调度器的状态，由 `Task_disable()`函数返回。

此函数与 `Task_disable()`成对使用。用于保护临界区域，不被其他任务抢占。

注意

不要在 `disable/restore` 块之间，调用可能会导致当前任务阻塞函数。例如 `Semaphore_pend()` (timeout 参数为非零), `Task_sleep()`, `Task_yield()`, `Memery_alloc` 等。

### 3. `Void Task_sleep(UInt32 nticks)`

`nticks` – 系统 tick 数

➤ 详细说明

`Task_sleep` 函数会将当前任务状态从 `RUNNING` 变成 `BLOCKED`，并延迟执行系统时钟的 `nticks` 增量。由于系统计时的间隔，延迟的实际时间可以达到 1 个系统时钟，小于 `nticks`，每个 tick 的时间由 `Clock_tickPeriod` 决定。在指定的时间段过去之后，任务将恢复到 `READY` 状态。调用 `Task_sleep` 时会出现任务切换。

注意：

`Task_sleep` 不要在 `Hwi`，`Swi` 和 `disable/restore` 结构中调用。

`Task_sleep` 不要在 `main` 函数中调用。

`Task_sleep` 不要在 `Idle` 空闲任务中调用。

### 4. `Void Task_yield();`

➤ 详细说明

`Task_yield` 将处理器控制权让给处于相同优先级的另一个任务。如果有等待优先级任务准备运行，则在您调用 `Task_yield` 时会发生任务切换。更高优先级的任务抢占当前运行的任务，不需要调用 `Task_yield`。如果在调用 `Task_yield` 时只有较低优先级的任务可以运行，则当前任务将继续运行。控制不会传递到较低优先级的任务。

## 例 3 任务抢占实验

调度器总是在可运行的任务中，选择具有最高优先级的任务，使其进入运行状态。在本例子中创建三个任务。演示 `SYS/BIOS` 任务抢占作用，即运行最高优先级任务。在例子中，`MainTask` 在 `main` 函数中创建，并具有最高优先级，而且只运行一次。`MyTask2` 任务在 `MainTask` 任务函数中创建，且具有最低优先级。而 `MyTask1` 任务在 `MyTask2` 任务函数中创建，其优先级高于 `MyTask2`。例 2 代码工程在 `3.3_Task_Preempt/` 目录下。

### 例 2 程序部分清单

```
void main()
{
    ...
    ...
    ...
    CreateMainTask();
    BIOS_start();
}
void CreateMainTask(void)
{
    Task_Params taskParams;

    Task_Params_init(&taskParams);
    taskParams.priority = Task_numPriorities - 1; //最高优先级
    Task_construct(&taskStruct, (Task_FuncPtr)MainTask, &taskParams,
    NULL);
```

```
}
Void MainTask(UArg arg0,UArg arg1)
{
    //辅助外设驱动初始
    ...
    ...
    UARTDriverInit(); //串口驱动初始化
    //创建用户任务
    CreateUserTask(); //创建 MyTask2
    UART_write(pUartHandle,"MainTask print \r\n",strlen("MainTask print
\r\n")); //串口打印
    Task_destruct(&taskStruct);
}
void CreateUserTask(void)
{
    Task_Params taskParams;
    //Error_Block eb;
    /* Construct heartBeat Task thread */
    Task_Params_init(&taskParams);
    taskParams.stackSize = TASKSTACKSIZE;
    taskParams.stack = &task2Stack;
    taskParams.priority = 1;
    myTask2Handle = Task_create((Task_FuncPtr)MyTask2, &taskParams,
NULL);
    if(myTask2Handle == NULL)
    {
        System_printf("Create Task2 occur error \n");
    }
}
Void MyTask2(UArg arg0, UArg arg1)
{
    CreateHighTask(); //创建 MyTask1

    .....

    UART_write(pUartHandle,"MyTask2 print \r\n",strlen("MyTask2 print
\r\n"));
    Task_delete(&myTask2Handle);
}
void CreateHighTask()
{
    Task_Params taskParams;
    Task_Params_init(&taskParams);
    taskParams.stackSize= TASKSTACKSIZE;
```

```
taskParams.stack = &task1Stack;
taskParams.priority = 2;
myTask1Handle = Task_create((Task_FuncPtr)MyTask1, &taskParams,
NULL);
}
Void MyTask1(UArg arg0, UArg arg1)
{
    //相关代码
    .....
    UART_write(pUartHandle,"MyTask1 print \r\n",strlen("MyTask1 print
\r\n"));
    Task_delete(&myTask1Handle);
}
```

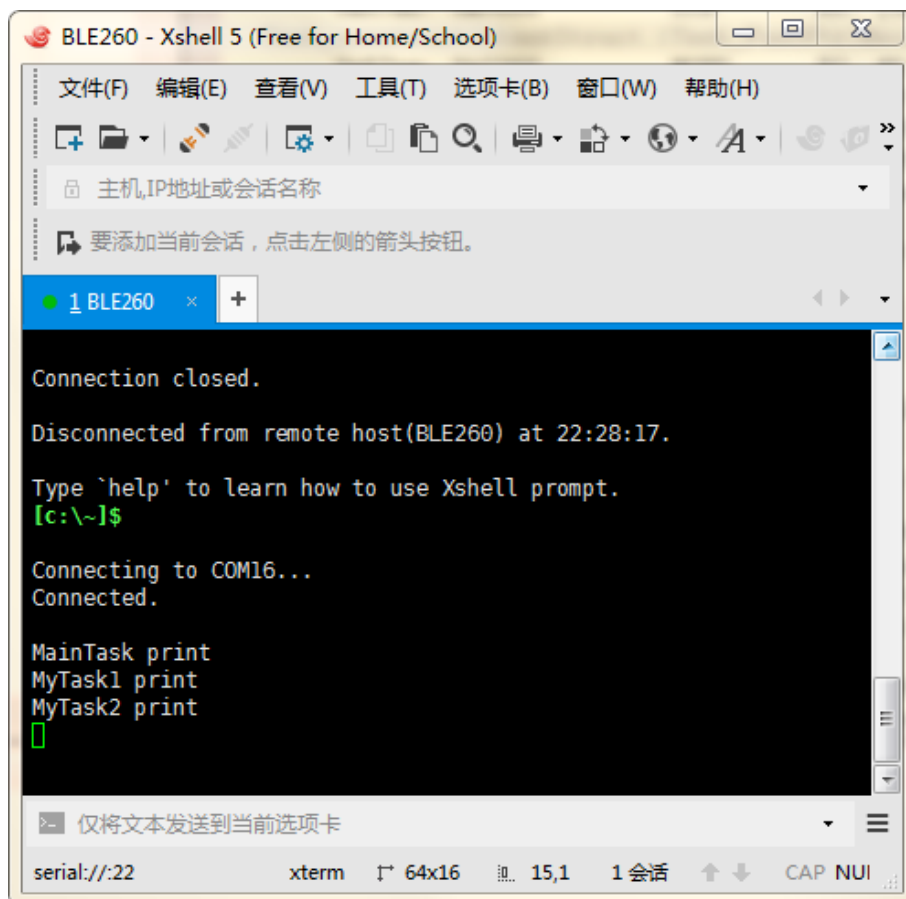


图 3 例 2 任务打印信息

根据图 3 信息, SYS/BIOS 首先运行 MainTask 任务, 因为 MainTask 任务具有最高优先级。在 MainTask 任务函数中, 创建了 MyTask2 任务, MyTask2 任务进入 READY 状态, 但其优先级低于当前任务, 所以调度器不会发生任务切换工作。MainTask 任务继续执行, 并向串口打印 “MainTask print”, 直到其退出进入 TERMINATED 状态。由于 MainTask 任务被删除, 现在可运行任务变成了 MyTask2, 所以调度器发生一次任务切换, MyTask2 进入 RUNNING 状态。MyTask2 任务开始执行, 在其内部创建了 MyTask1, 并使其进入 READY 状态, 由于其优先级高于 MyTask2, 调度器发生一次任务切换, MyTask1 进入 RUNNING 状态, MyTask2 被抢

占而进入 READY 状态。MyTask1 获得 CPU 控制权，开始执行其内部代码，并向串口打印 “MyTask1 print”，执行完之后，任务删除自己使其退出进入 TERMINATED 状态。由于当前任务放弃 CPU 控制权，系统调度器发生一次任务切换，此时 MyTask2 成为当前可运行最高优先级任务。MyTask2 从 READY 状态进入 RUNNING 状态。继续从上次被抢占的地方执行，并向串口打印 “MyTask2 print”。执行完之后，便退出进入 TERMINATED 状态。

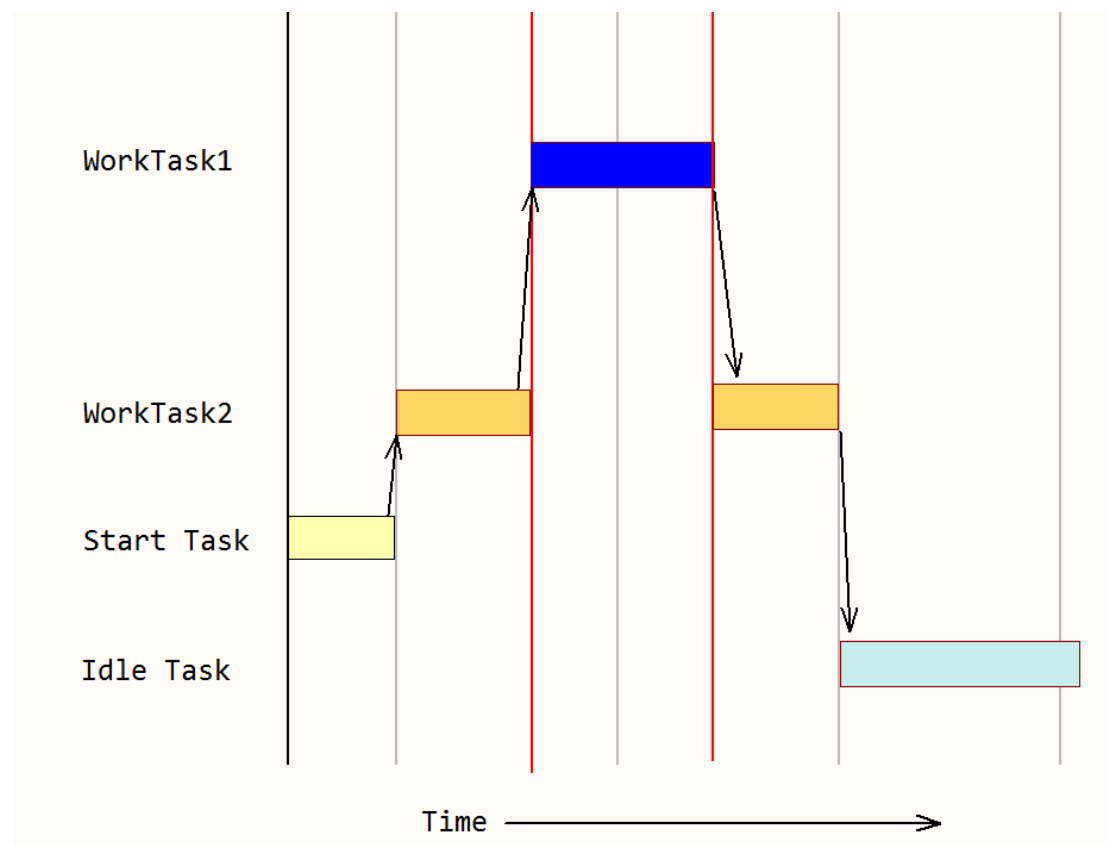


图 4 例 2 任务运行时序图

## 例 4 任务优先级更改

在实际使用过程中，有时需要提升各别任务优先级，来满足实际需求。在本例实验中，通过调用 Task\_setPri() API 函数将任务的优先级提高。本例子是基于例 1 的代码工程进行修改的，所以大部分代码与例 1 工程相似。下面的示意代码只贴出部分代码。具本的工程代码见 3.4\_Task\_Pri/目录下。

```
Void MyTask1(UArg arg0, UArg arg1)
{
    //初始化串口
    UChar count = 0;
    .....
    .....

    while(1)
    {
        .....
    }
}
```

```
if(count++ == 50)
{
    Task_setPri(Task_self(), TASK_NEW_PRI);
}
else if(count == 200)
{
    Task_setPri(Task_self(), TASK_OLD_PRI);
}
}
```

代码中变量 `count` 通过简单的记数，来模拟一些事件。当记数达到 50 时，将自己的任务优先级提高到新的优先级，然后在新的优先级下运行一段时间，再恢复到当前优先级。用户在实际使用中，可以将 `count` 改成相应的事件等。

## 第三章 任务间同步

TI-RTOS 是一个多任务嵌入式系统，各任务必须通过彼此之间的有效合作，来完成一项大规模的工作。在运行过程中，经常需要无冲突地访问一个共享资源，或者需要互相支持和依赖，甚至有时还要互相加以制约，才能保证任务顺序运行。因此，操作系统必须具有对任务的运行进行协调的能力，从而使任务之间流畅运行。就像人们通过交流，沟通一起完成一项工作一样。

为了实现各个任务之间的合作和无冲突的运行，在多任务操作系统中应该要解决两个问题：一是各个任务间应该具有一种互斥关系，即对于共享资源访问。如果一个任务正在使用，则其他任务只能等待，等到该任务释放该资源后，等待的任务之一才能使用它；二是相关任务在执行上要有先后顺序，一个任务要等其伙伴发来通知，或建立了某个条件后才继续执行，否则只能等待。任务之间的这种制约性的合作运行机制统称为任务间同步。

在 TI-RTOS 操作系统中提供了以下几种方法用于任务间同步。

- 信号量 (Semaphores)
- 事件 (Events)
- 门 (Gates) (见第六章)
- 邮箱 (MailBoxes)
- 队列 (Queues)

### 3.1 信号量 (Semaphores)

SYS / BIOS 基于信号量为任务间同步和通信提供了一套基本功能函数。信号量通常用于协调对共享资源的访问一套竞争的任务。SYS/BIOS 内核中提供了操作信号量相关函数，且通过 `Semaphore_Handle` 类型句柄对其进行访问。

信号量可以被声明为二值信号量或记数信号量。默认为记数信号量。记数型信号量记数值与系统内部资源数保持一致。当记数值大于 0 时，任务在申请获取信号量时是不会被阻塞的。记数型信号量的最大数值的决定应该根据实际资源数来确定。而二值信号量只有两种状态，要么可用，要么不可用。它们的值不能超过 1。因此，二值信号值通常用于协调访问共

享资源，且它的性能也优于记数值信号量。

在 SYS/BIOS 中，信号量模式具体有四种，且可以被分二值信号量与记数信号两类。

- Semaphore\_Mode\_COUNTING
- Semaphore\_Mode\_BINARY
- Semaphore\_Mode\_COUNTING\_PRIORITY
- Semaphore\_Mode\_BINARY\_PRIORITY

Semaphore\_Mode\_COUNTING 与 Semaphore\_Mode\_BINARY 是具有 FIFO 特性。主要体现在任务在请求信号时，是采用 FIFO 方式进行排队等待。先请求的任务，可以优先获得信号量，后来的任务只能等待。而 Semaphore\_Mode\_COUNTING\_PRIORITY 与 Semaphore\_Mode\_BINARY\_PRIORITY 是基于任务优先级进行排队等待。任务优先级高的优先获得信号资源。

#### 注意

使用优先级信号量会增加系统中断延时。

配置信号量模式可以通过参数 mode 域进行配置。见下面代码示意。

```
semParams.mode = Semaphore_Mode_BINARY;
或
semParams.mode = Semaphore_Mode_COUNTING;
或
semParams.mode = Semaphore_Mode_BINARY_PRIORITY;
或
semParams.mode = Semaphore_Mode_COUNTING_PRI
```

### 3.1.1 创建/删除信号量

SYS/BIOS 中可以通过系统提供的 Semaphore\_create()和 Semaphore\_delete()两个 API 函数分别创建和删除信号量。也可以通过 Semaphore\_construct()和 Semaphore\_destruct()进行创建和删除。

#### 1. Semaphore\_Handle Semaphore\_create(

Int count,  
Semaphore\_Params \*attrs,  
Error\_Block \*eb);

count – 初始化的信号量值。一般情况，它应该被设为资源数

attrs – 信号量的配置参数。如果为 NULL，则系统会选择默认参数

eb – 错误处理块，如果为 NULL，则系统会选择默认方法

返回值 -- 如果成功创建，则返回信号量对象句柄。如果失败，则返回 NULL。

#### 2. Void Semaphore\_delete(Semaphore\_Handle \*handleP);

handle – 是信号量对象句柄指针

delete 函数会结束，释放信号量对象实例，并将 handleP 设成 NULL。

#### 3. Void Semaphore\_Params\_init(Semaphore\_Params \*params);

params – 是 Semaphore\_Params 型变量指针

此函数会将信号量配置参数初始化为默认参数。（此函数调用在 create 函数之前）

示例代码：创建二值信号量

```
Semaphore_Handle handleSema;  
Semaphore_Params paramsSema;
```

```

Semaphore_Params_init(&paramSema);
paramSema.mode = Semaphore_Mode_BINARY;
handleSema = Semaphore_create(1, &paramSem, NULL);
if(handleSema == NULL)
{
    //进行相应处理
    .....
}

```

#### 4. Void Semaphore\_construct(

```

        Semaphore_Struct    *structP,
        Int                  count,
        const Semaphore_Params *params);

```

structP – Semaphore\_Struct 结构变量指针

count – Semaphor 初始化的 count 值

params – Semaphore 配置信息参数指针

此 construct 函数与 create 函数功能相似，都是创建信号量函数。

#### 5. Void Semaphore\_destruct(Semaphore\_Struct \*structP)

structP – 信号量 Semaphore\_Struct 结构信息指针

些函数与 delete 函数一样，是删除指定的信号量，并释放占用的资源。

#### 6. Semaphore\_Handle Semaphore\_handle(Semaphore\_Struct \*structP)与 Semaphore\_Struct \*Semaphore\_struct(Semaphore\_Handle handle)

Semaphore\_handle 函数是从 Semaphore\_Struct 结构中获得 Semaphore\_Handle 信息，而 Semaphore\_struct 函数是从 Semaphore\_Handle 中获得 Semaphore\_Struct 结构指针。

### 3.1.2 信号申请与释放

系统中通过 Semaphore\_pend(), 与 Semaphore\_post()两函数对信号量进行获取与释放。Semaphore\_pend 是申请一个信号量。如果信号量的 count 大于 0, Semaphore\_pend()会将 count 值减 1 并返回。否则 Semaphore\_pend()只能等待直到信号量被 Semahore\_post()函数释放。

#### 1. Bool Semaphore\_pend(Semaphore\_Handle handle, UInt32 timeout);

handle – 创建的信号量返回的句柄指针

timeout – 等待超时时间。单位是 System tick。timeout 只有在信号量的 count 值为 0 时，才会起作用。如果 timeout 时间内，还是没有可用信号量，函数也将返回，并给出超时。

返回值 – TRUE 表示成功

FALSE 表示超时

##### ➤ 详细说明

如果信号量的 count 大于 0 (available), pend 将 count 值减 1, 并返回 TRUE。如果信号量的 count 等于 0 (unavailable), 这个函数挂起当前任务直到 post()被调用或超时发生。

timeout 是 BIOS\_WAIT\_FOREVER 值, 任务一直等待一个可用的信号量

timeout 是 BIOS\_NO\_WAIT 值, 函数会立即返回。

#### 2. Void Semaphore\_post(Semaphore\_Handle handle);



handle – 已创建信号量对象指针

➤ 详细说明

此函数将会就绪第一个等待信号量任务。如果没有等待任务，此函数将信号量 count 加 1 并返回。至于二值信号量，count 最大只能为 1。

### 3. Int Semaphore\_getCount(Semaphore\_Handle handle);

handle – 已创建信号量对象指针

返回值 – 当前信号的 count 值。

## 例 5 利用二值信号量对任务控制

例 3 中，创建两个工作任务 workTask1 和 workTask2，任务优先级分别 1 和 2。在 workTask2 中一直在请求一个二值信号量，如果请求成功，则向 PC 打印 “workTask2 pend sem ok”。在低优先级任务 workTask1 中，每三秒中向信号量 post 一次。这样 workTask2 任务就会从 pend 中返回执行下面的代码，然后继续 pend 信号量，由于信号量不可用，此任务就会进入阻塞状态，等待信号量可用。具体代码工程在 3.5\_Semaphore\_Test 目录下。

### 例 3 程序部分清单

```
//main 函数中创建具有最高优先级启动任务
int main(void)
{
    //相关驱动初始化
    ...
    ...
    //创建启动任务
    USER_createStartTask();
    /* Start BIOS */
    BIOS_start();
    return (0);
}

void USER_createStartTask(void)
{
    System_printf("System Create StartTask \n");
    System_flush();
    Task_Params taskParams;
    Task_Params_init(&taskParams);

    taskParams.priority = Task_numPriorities - 1; //指定任务为最高优先级
    taskParams.stackSize = TASK_STACK_SIZE + 128; //指定任务堆栈空间大小
    Task_construct(&startTask, (Task_FuncPtr)StartTaskFunc,
                  &taskParams, NULL);
}
```

在启动任务中创建 workTask1 和 workTask2 任务，初始值为 1 的二值信号量，最后自己删除自己。

```
static Void StartTaskFunc(UArg a0,UArg a1)
```

```
{
    .....
    .....
    //创建二值信号量
    SEM_Create();
    //创建两个工作任务
    USER_createWorkTask(1,workTask1Func,&workTask1);
    USER_createWorkTask(2,workTask2Func,&workTask2);
    .....
    .....
    Task_destruct(&startTask);
}

//任务 workTask1Func 与任务 workTask2Func
static Void workTask1Func(UArg a0,UArg a1)
{
    System_printf("work1Task run,pri = %d \n",Task_getPri(Task_self()));
    System_flush();

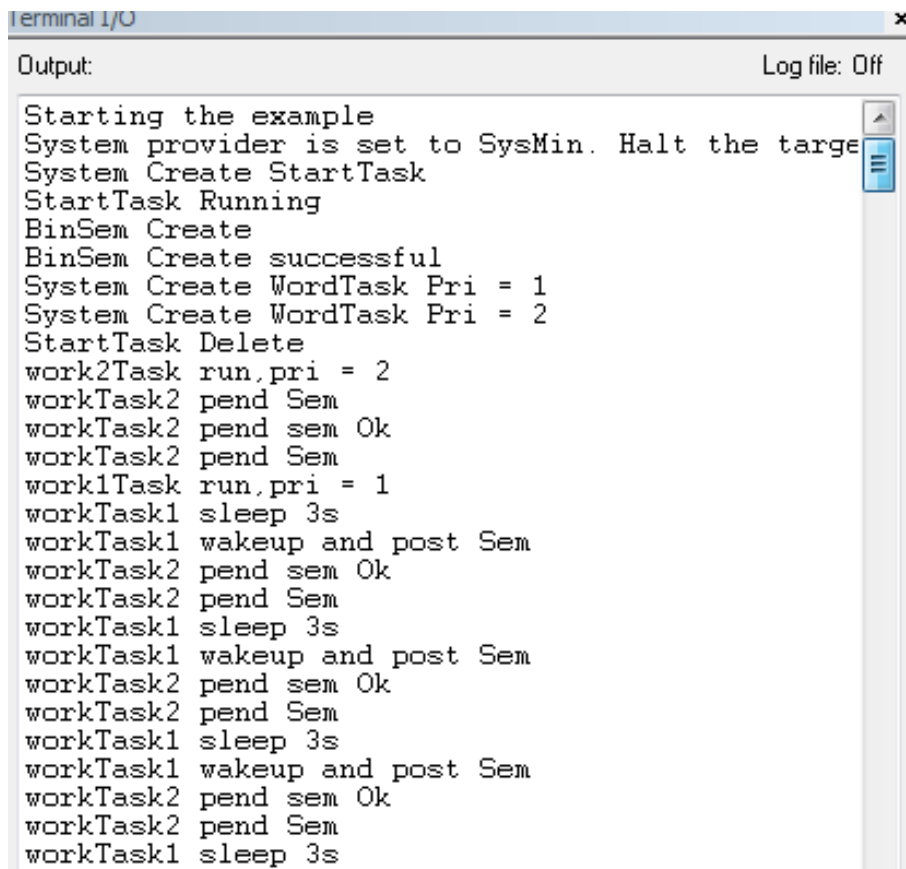
    for(;;)
    {
        .....
        Task_sleep(300000); //sleep time = 3s

        .....
        Semaphore_post(semBinHandle); //post 信号量
    }
}

static Void workTask2Func(UArg a0,UArg a1)
{
    .....
    for(;;)
    {
        ...
        Semaphore_pend(semBinHandle,BIOS_WAIT_FOREVER); //请求信号量,直到可用
        .....
    }
}
```

下图是例 3 实验程序打印的信息。程序开始运行时，由 main 函数创建 StartTask 任务。之后启动系统之后，由系统调用当前最高优先级任务 StartTask。并在其内部创建了一个信号量和两个任务。当 StartTask 任务删除之后，系统便会运行优先级为 2 的 workTask2 任务，因为它是就绪任务中优先级最高的。workTask2 运行之后，请求信号量。信号量开始为可用状态，所以 pend 函数会立即返回。当继续请求信号量时，发现此时的信号量已经不可用，pend 函数阻塞当前任务。调度器开始运行已经就绪任务 workTask1，workTask1 任务一开始就睡

眠 3 秒，任务再一次被阻塞。此时系统只能运行 idle 空闲任务。直到过了 3 秒，workTask1 醒来，抢占当前空闲任务，并向信号量 post 了一个信号，此时一直处在等待状态的 workTask2 获得了信号量，成为当最高优先级任务，便会抢占 workTask1 任务。两个任务在信号量的协调下一直循环运行。但是高优先级的 workTask2，迟于低优先级 workTask1 后运行，这个现象称为优先级返转。



```
Starting the example
System provider is set to SysMin. Halt the target
System Create StartTask
StartTask Running
BinSem Create
BinSem Create successful
System Create WordTask Pri = 1
System Create WordTask Pri = 2
StartTask Delete
work2Task run, pri = 2
workTask2 pend Sem
workTask2 pend sem Ok
workTask2 pend Sem
work1Task run, pri = 1
workTask1 sleep 3s
workTask1 wakeup and post Sem
workTask2 pend sem Ok
workTask2 pend Sem
workTask1 sleep 3s
workTask1 wakeup and post Sem
workTask2 pend sem Ok
workTask2 pend Sem
workTask1 sleep 3s
workTask1 wakeup and post Sem
workTask2 pend sem Ok
workTask2 pend Sem
workTask1 sleep 3s
```

图 5 例 3 打印信息

## 例 6 利用二值信号进行互斥访问

在 RTOS 中，常常需要对共享资源进行互斥访问。即对其进行独占式的访问。这样的操作是有必要的。具体的模型与令牌特别相似。如果一个任务要合法地访问资源，其必须获得对该资源访问的令牌，有了令牌任务就可以访问该资源，当任务使用完成之后，必须马上归还令牌。没有获得令牌的任务只能等待，直到其他任务归还令牌，并获得令牌，否则只能等待（即阻塞）。本例子中利用二值信号实现对全局变量的互斥访问。下面代码只是例子中部分代码，具体的工程代码见 3.6\_Semaphore\_Mutex/目录下。

```
//用户任务函数 1
static Void workTask1Func(UArg a0,UArg a1)
{
    for(;;)
    {
        Semaphore_pend(semBinHandle, BIOS_WAIT_FOREVER);
```

```
PIN_setOutputValue(ledPinHandle, Board_LED1,
                  !PIN_getOutputValue(Board_LED1));

g_Count++;
Task_sleep(100000); //sleep 1s

LCD_Print("g_Count=%d", (char*)&g_Count);

PIN_setOutputValue(ledPinHandle, Board_LED1,
                  !PIN_getOutputValue(Board_LED1));
Semaphore_post(semBinHandle);
}
}
//用户任务 2 对 g_Count 进行有条件复位
static Void workTask2Func(UArg a0,UArg a1)
{

    for(;;)
    {
        Semaphore_pend(semBinHandle, BIOS_WAIT_FOREVER);
        PIN_setOutputValue(ledPinHandle, Board_LED0,
                          !PIN_getOutputValue(Board_LED0));
        if(g_Count > 10)
        {
            g_Count = 0;
            LCD_Print("Reset g_Count", NULL);
        }
        PIN_setOutputValue(ledPinHandle, Board_LED0,
                          !PIN_getOutputValue(Board_LED0));
        Semaphore_post(semBinHandle);
        Task_sleep(10000); //sleep 1s
    }
}
```

上述代码中，在 **workTask1Func** 任务函数中，利用信号量对 **g\_Count** 变量进行自加处理，并在随后用 **Task\_sleep** 交出 CPU 权限。而在 **workTask2Func** 任务函数中，访问 **g\_Count**，并判断如果大于 10 将复位 **g\_Count**。

## 3.2 事件（Event）

SYS/BIOS 中事件是任务与其它线程之间或任务与其他 SYS/BIOS 对象之间通信方式。其他 SYS/BIOS 对象包括信号量，邮箱，消息队列等。只有任务可以等待事件，而任务，Hwi，Swi 或 SYS/BIOS 对象可以发布事件。

为了能从 SYS/BIOS 对象中接收通知事件，首先该对象必须先注册事件对象。为了让每个 SYS/BIOS 对象支持此功能，SYS/BIOS 提供了单独的 API 函数。事件本质上是任务同步，这

意味着接收任务将在等待事件发生时会产生阻塞或挂起。当接收到所需事件时，挂起的任务继续执行。例如调用 `Semaphore_pend()` 之后。

任务还可以等待未链接到其他 SYS/BIOS 对象事件。这些事件从其他线程（如任务，Hwi 或 Swi）显式发布。如果任务不注册接收这些事件，发送线程只是简单地将其事件发布到任务正在等待的事件对象上。这种情况类似于使用 ISR 发布信号量。

任务可以等待来自多个资源或线程的事件，因此它可以等待信号量发布，和等待消息到达消息队列或 ISR 线程发布信号通知事件。事件是二值的，类似于二值信号量。事件可以通过调用 `Event_post()` 使相应的事件 ID 成为可用；也可以根据 `andMask` 或 `orMask` 调用 `Event_pend()` 使相应的事件 ID 成为不可用。与信号量不同，只有一个任务可以挂在 Event 对象上。`pend` 用于等待事件，`andMask` 各 `orMask` 确定在从 `pend` 返回之前必须发生哪些事件。超时参数允许任务等待直到超时，永不超时，或者完全不等待三种。返回值为 0 表示发生超时，非 0 返回值是在任务解除阻止时处于活动状态的事件集。

`andMask` 定义一组事件，必须全部发生才能使 `pend` 返回。

`orMask` 定义一组事件，如果发生任何事件，都可以导致 `pend` 返回。

从 `pend` 中返回时，`orMask` 中存在的所有活动事件将被消耗（即从事件对象中删除）。只有当存在于 `andMask` 中的所有事件都处于活动状态时，才会在从 `pend` 返回时消耗。

如果 `andMask` 或 `orMask` 条件满足时，则 `pend` 会立即返回。`post` 用于发布事件。如果任务正在等待事件并且所有事件都满足条件，则 `post` 会就绪等待的任务。如果没有任务正在等待，那么 `post` 只是向事件对象中注册事件并返回。

单个事件对象实例，可以管理 32 个事件，每个事件可以由事件 ID 表示。事件 ID 是一个简单的位标志，每个事件占 `UInt` 类型数据的一位，如下定义的事件 ID 常量。所以从单事件 ID 看，每个事件 ID 就像一个二值信号量。最后需要注意的是，只有任务能调用 `Event_pend()`，而 `Event_post()` 是所有线程都可以调用。

SYS/BIOS 事件模块中定义事件 ID 号如下

```
#define Event_Id_00 (UInt)0x1
#define Event_Id_01 (UInt)0x2
#define Event_Id_02 (UInt)0x4
#define Event_Id_03 (UInt)0x8
#define Event_Id_04 (UInt)0x10
#define Event_Id_05 (UInt)0x20
#define Event_Id_06 (UInt)0x40
#define Event_Id_07 (UInt)0x80
#define Event_Id_08 (UInt)0x100
#define Event_Id_09 (UInt)0x200
#define Event_Id_10 (UInt)0x400
#define Event_Id_11 (UInt)0x800
#define Event_Id_12 (UInt)0x1000
#define Event_Id_13 (UInt)0x2000
#define Event_Id_14 (UInt)0x4000
#define Event_Id_15 (UInt)0x8000
#define Event_Id_16 (UInt)0x10000
#define Event_Id_17 (UInt)0x20000
#define Event_Id_18 (UInt)0x40000
#define Event_Id_19 (UInt)0x80000
```

```
#define Event_Id_20 (UInt)0x100000
#define Event_Id_21 (UInt)0x200000
#define Event_Id_22 (UInt)0x400000
#define Event_Id_23 (UInt)0x800000
#define Event_Id_24 (UInt)0x1000000
#define Event_Id_25 (UInt)0x2000000
#define Event_Id_26 (UInt)0x4000000
#define Event_Id_27 (UInt)0x8000000
#define Event_Id_28 (UInt)0x10000000
#define Event_Id_29 (UInt)0x20000000
#define Event_Id_30 (UInt)0x40000000
#define Event_Id_31 (UInt)0x80000000
#define Event_Id_NONE (UInt)0
```

### 3.2.1 创建/删除事件对象

SYS/BIOS 提供了相应的 API 函数，用于创建删除事件对象。下面将详细介绍这些 API 函数的使用。

1. **Void Event\_construct(Event\_Struct \*structP, const Event\_Params \*params)**  
structP – 是用于存放事件对象信息的 Event\_Struct 结构变量指针  
params – 是事件对象配置参数指针
2. **Void Event\_destruct(Event\_Struct \*structP)**  
structP – 是已经创建的事件对象 Event\_Struct 结构指针  
此函数将释放实例对象在系统内部占用的资源
3. **Event\_Handle Event\_create(const Event\_Params \*params, Error\_Block \*eb)**  
params --是事件对象配置参数指针，在创建之前应该设置好相应的函数.如果为 NULL，将采用默认参数配置事件对象。  
eb – 错误处理模块，如果为 NULL，将选择默认处理  
返回值 – 如果创建事件对象成功，将返回事件对象句柄；如果失败将返回 NULL
4. **Void Event\_delete(Event\_Handle \*handleP)**  
handleP – 是已经创建的事件对象句柄指针  
此函数将结束和释放事件对象占用的资源，同时将 handle 设为 NULL。

通过上述四个函数，完全可以完成事件对象的创建和删除工作。下面给出示意代码。

```
Event_Handle myEvent;
Error_Block eb;
Error_init(&eb);

//使用默认配置参数
myEvent = Event_create(NULL, &eb);
if(myEvent == NULL)
{
    //创建事件对象失败
    .....
```

```
}
```

删除事件对象

```
Event_delete(&myEvent)
```

或

```
Event_destruct(&myEventStruct);
```

Event\_Handle 与 Event\_Struct 结构指针可以相互转换。具体转换可以通过下面的 API 函数进行。

1. **Event\_Handle Event\_handle(Event\_Struct \*structP)**
2. **Event\_Struct \*Event\_struct(Event\_Handle handle)**

### 3.2.2 事件 pend/post

SYS/BIOS 中事件请求与发布是通过 Event\_pend(), Event\_post()函数进行。下面将详细介绍这两个函数使用方法。

1. **UInt Event\_pend(Event\_Handle handle, UInt andMask, UInt orMask, UInt32 timeout)**

handle – 已经创建的事件实例对象句柄

andMask – 所请求的事件必须全部发生，才能从 pend 中返回

orMask – 所请求的事件只要有一个发生，就从 pend 中返回

timeout – 如果不能从 pend 立即返回，最长阻塞多长时间，时间单位是系统 tick。timeout 有三种可能情况，BIOS\_WAIT\_FOREVER，有具体的超时时间，BIOS\_NO\_WAIT。

返回值 – 如果超时，返回 0；如果条件满足，返回所有余下的事件集

此函数通过 andMask 和 orMask 请求事件

2. **Void Event\_post(Event\_Handle handle, UInt eventMask)**

handle -- 已经创建的事件实例对象句柄

eventMask – 事件 ID 号。如果有多个事件可以通过 “+” 或 “|” 进行连接

此函数向指定的事件对象上，发布指定的事件

3. **UInt Event\_getPostedEvents(Event\_Handle handle)**

handle – 已经创建的事件实例对象句柄

返回值 – 返回指定事件对象当前事件集

## 例 7 事件创建实验

在例 7 中，创建一个事件对象，一个用户任务。在用户任务中调用 Event\_pend()请求 Event\_Id\_00 和 Event\_Id\_01 事件。而这个两个事件分别在两个按钮 ISR 服务程序中调用 Event\_post()进行发布或注册。在实验中，为了能通过屏幕打印信息，采用了迂回方式实现。ISR 服务程序不直接注册事件，而是通过信号量向另一个任务通知按钮按下事件，然后在这个任务中处理按钮事件，并在这个任务中调用 Event\_post()，来发布 Event\_Id\_00 或 Event\_Id\_01 事件。而在任务 2 中，利用 Event\_pend()的 API 函数，采用 AndMode 方式请求 Event\_Id\_00 和 Event\_Id\_01 事件，详细的工程代码见 **3.10\_Event\_Create\_AndMode/**目录下。而在 **3.11\_Event\_Create\_OrMode** 工程中采用 OrMode 方式请求 Event\_Id\_00 和 Event\_Id\_01 事件。AndMode 与 OrMode 区别是，AndMode 要求所有请求的事件都发生，Event\_pend()才会返回，

而 OrMode 只要请求事件中一个发生，Event\_pend()都会返回。

### 例 7 部分程序清单

```
int main(void)
{
    .....
    .....
    USER_createStartTask();
    /* Start BIOS */
    BIOS_start();
    return (0);
}

void USER_createStartTask(void)
{
    .....
    .....
    Task_Params taskParams;
    Task_Params_init(&taskParams);

    taskParams.priority = Task_numPriorities - 1;
    taskParams.stackSize = TASK_STACK_SIZE + 128;

    Task_construct(&startTask, (Task_FuncPtr)StartTaskFunc, &taskParams, N
ULL);
}

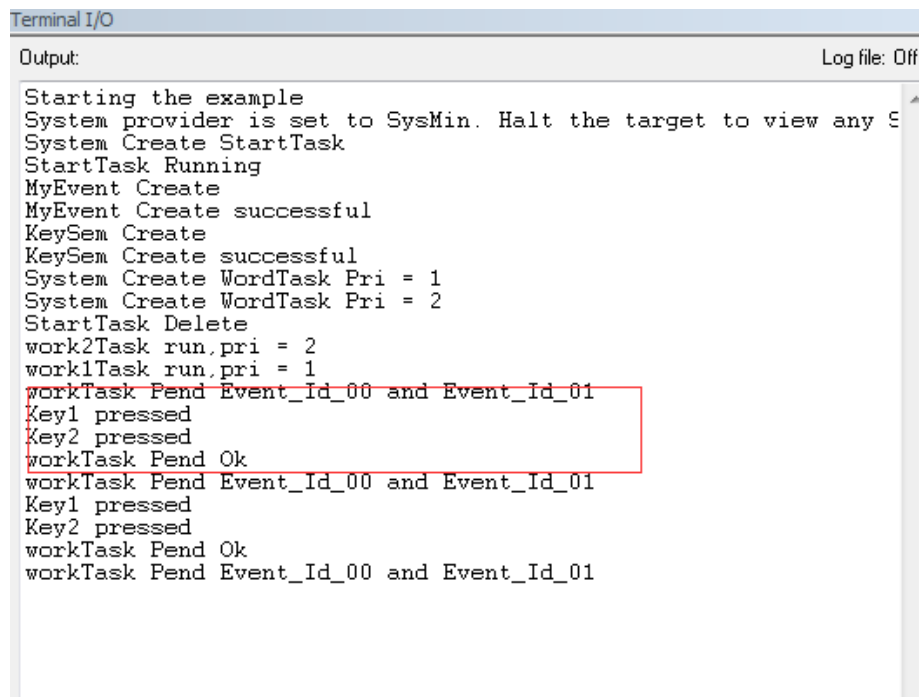
static Void StartTaskFunc(UArg a0, UArg a1)
{
    .....
    .....
    EVENT_Create();           //创建事件
    SEM_Create();             //创建信号量
    //创建工作任务
    USER_createWorkTask(1, workTask1Func, &workTask1);
    USER_createWorkTask(2, workTask2Func, &workTask2);

    .....
    Task_destruct(&startTask); //删除任务
}

//任务 1 函数，主要进行事件获取
Void workTask1Func(UArg a0, UArg a1)
{
    .....
    for(;;)
    {
        .....
    }
}
```



```
    Event_pend(myEventHandle, Event_Id_00|Event_Id_01,Event_Id_NONE,
    BIOS_WAIT_FOREVER);
    .....
}
}
//任务2 函数，主要处理按键和事件发布
Void workTask2Func(UArg a0,UArg a1)
{
    .....
    for(;;)
    {
        Semaphore_pend(keySemHandle,BIOS_WAIT_FOREVER);
        eventId = Event_Id_NONE;
        if(gKeys & KEY_BTN1)
        {
            eventId |= Event_Id_00;
            keyInfo = "Key1 pressed";
        }
        if(gKeys & KEY_BTN2)
        {
            eventId |= Event_Id_01;
            keyInfo = "Key2 pressed";
        }
        .....
        .....
        //发布按钮事件
        Event_post(myEventHandle,eventId);
    }
}
```



```

Terminal I/O
Output:
Starting the example
System provider is set to SysMin. Halt the target to view any S
System Create StartTask
StartTask Running
MyEvent Create
MyEvent Create successful
KeySem Create
KeySem Create successful
System Create WordTask Pri = 1
System Create WordTask Pri = 2
StartTask Delete
work2Task run, pri = 2
work1Task run, pri = 1
workTask Pend Event_Id_00 and Event_Id_01
Key1 pressed
Key2 pressed
workTask Pend Ok
workTask Pend Event_Id_00 and Event_Id_01
Key1 pressed
Key2 pressed
workTask Pend Ok
workTask Pend Event_Id_00 and Event_Id_01
  
```

图 6 例 7 调试信息

### 3.3 队列 (Queues)

SYS/BIOS 队列模块提供了一组函数来处理队列对象，函数通过类型为 `Queue_Handle` 的句柄访问队列对象。每个队列包含通过类型为 `Queue_Elem` 的变量引用的零个或多个元素的链接序列，它们作为结构中的第一个字段嵌入。在队列 API 描述中，在修改队列之前禁用中断的 API 被标记为“原子”，而不禁用中断的 API 是“非原子的”。队列被表示为双向链接列表，因此对 `Queue_next` 或 `Queue_prev` 的调用可以在 `Queue` 上连续循环。以下代码演示了一种从头到尾遍历队列一次的方法。在这个例子中，“myQ”是一个 `Queue_Handle`。

```

Queue_Elem *elem;

for (elem = Queue_head(myQ);
     elem != (Queue_Elem *)myQ;
     elem = Queue_next(elem)) {
    ...
}
  
```

为了向队列中添加一个结构，则元素结构的第一个域类型必须为 `Queue_Elem`。下面的代码将展现了一个可以被添加到队列中的结构。队列对象有一个“head”，它指向的队列链表的第一个元素。`Queue_enqueue()`将元素添加队列的尾部，`Queue_dequeue()`从 head 处将元素从队列中删除并返回。所以队列与 FIFO 的数据结构十分相似。

```

typedef struct Rec
{
    Queue_Elem _elem;
    Int data;
} Rec;
  
```

### 3.3.1 队列创建

SYS/BIOS 中提供了相应的 API 函数用于创建、删除队列。接下来，将详细介绍队列对象的创建。

- 1. Queue\_Handle Queue\_create(const Queue\_Params \*params, Error\_Block \*eb);**  
params – 是创建队列的配置参数，如果为 NULL，将选择默认值。  
eb – 错误处理块，如果为 NULL，将选择默认方式。  
返回值 – 如果创建成功，将返回队列对象的句柄；如果失败，将返回 NULL。  
此函数的作用是分配并初始化新的实例对象并返回对象句柄。
- 2. Void Queue\_delete(Queue\_Handle \*handleP)**  
handleP – 指向队列对象句柄  
返回值 – NONE  
此函数的作用，与 create 函数相反。它用于删除对象实例，并释放实例占用的空间。
- 3. Void Queue\_construct(Queue\_Struct \*structP, const Queue\_Params \*params)**  
structP – 是 Queue\_Struct 类型变量指针  
params – 是创建队列的配置参数，如果为 NULL，将选择默认值。  
返回值 – 空  
此函数也是用于创建队列对象。作用与 Queue\_create()一样。
- 4. Void Queue\_destruct(Queue\_Struct \*structP);**  
structP – 是 Queue\_Struct 类型变量指针  
此函数也是删除实例对象，并释放其占用的资源。
- 5. Void Queue\_Params\_init(Queue\_Params \*params)**  
params – 是 Queue\_Params 类型变量指针  
此函数是将传入的配置参数，初始化为默认值。
- 6. Queue\_Handle Queue\_handle(Queue\_Struct \*structP)**
- 7. Queue\_Struct \*Queue\_struct(Queue\_Handle handle)**  
Queue\_handle()与 Queue\_struct()是将 Queue\_Handle 与 Queue\_Struct 结构进行互相转换。

### 3.3.2 队列操作

通过上述的 API 函数创建了队列对象之后，便可以利用队列句柄对其进行操作。SYS/BIOS 提供一系列函数用于对队列操作。下面将介绍常用的 API 函数。

- 1. Void Queue\_enqueue(Queue\_Handle handle, Queue\_Elem \*elem)**  
handle – 队列实例对象句柄  
elem – 加入队列元素的 Queue\_Elem 成员地址  
此函数将指定的元素加入到队列后面
- 2. Ptr Queue\_dequeue(Queue\_Handle handle)**  
handle – 队列实例对象句柄  
返回值 – 指向元素的指针  
➤ 详细说明  
此函数从队列上删除第一个元素并返回。但是如果队列为空时，dequeue()也会返回一

个非 NULL 值，这是队列的实现形式决定的。所以在调用 Queue\_dequeue()之前，使用 Queue\_empty()进行判断，再决定是否调用 Queue\_dequeue()。注意，此函数不会导致任务阻塞。

### 3. Bool Queue\_empty(Queue\_Handle handle)

handle – 队列实例对象句柄

返回值 – TRUE 此队列为空

FALSE 此队列不为空

此函数只是检测指定的队列对象是否为空。

### 4. Ptr Queue\_head(Queue\_Handle handle)

handle – 队列实例对象句柄

返回值 – 返回指向队列第一个元素的指针

➤ 详细说明

此函数返回队列中第一个元素，但不会从队列中将其删除。但是如果队列为空时，dequeue()也会返回一个非 NULL 值，这是队列的实现形式决定的。所以在调用 Queue\_head()之前，使用 Queue\_empty()进行判断，再决定是否调用 Queue\_head()。

示例代码

演示了如何创建队列，并加入两个元素，然后取出元素直到队列为空

```
#include <xdc/std.h>
#include <xdc/runtime/System.h>

#include <ti/sysbios/knl/Queue.h>

typedef struct Rec
{
    Queue_Elem _elem;
    Int data;
} Rec;

Int main(Int argc, Char *argv[])
{
    Queue_Handle q;
    Rec r1, r2;
    Rec* rp;

    r1.data = 100;
    r2.data = 200;

    // 创建一个队列实例 'q'
    q = Queue_create(NULL, NULL);

    //加入两个队列元素
    Queue_enqueue(q, &r1._elem);
```

```
Queue_enqueue(q, &r2._elem);

// 取出队列元素，直到队列为空
while (!Queue_empty(q))
{
    rp = Queue_dequeue(q);
    System_printf("rec: %d\n", rp->data);
}
System_exit(0);
return (0);
}
```

### 3.3.3 遍历队列

队列提供了两个函数分别为，Queue\_prev()和 Queue\_next()。

**1. Ptr Queue\_prev(Queue\_Elem \*qelem)**

qelem – 队列中的元素指针。

返回值 – 前一个元素指针

此函数返回 qelem 之前元素指针。

**2. Ptr Queue\_next(Queue\_Elem \*qelem)**

qelem – 队列中的元素指针。

返回值 – qelem 后面一个元素指针

此函数返回 qelem 之后元素指针。

利用上述两个函数对指定队列进行遍历。假设 myQ 是已经创建好队列的句柄。示意代码如下。示例中方法只是其中一种，读者可以用另一个函数进行遍历也是可以的。

```
Queue_Elem *elem;
for (elem = Queue_head(myQ); elem != (Queue_Elem *)myQ;
    elem = Queue_next(elem))
{
    ...
}
```

### 3.3.4 插入删除元素

Queue\_insert()函数可以将元素插入到队列的任何位置，而 Queue\_remove()可以删除队列中任何位置的元素。

**1. Void Queue\_insert(Queue\_Elem \*qelem, Queue\_Elem \*elem)**

qelem – 队列中的元素指针，即插入点

elem – 待插入元素地址

此函数将 elem 元素插入到 qelem 元素的前面。

**2. Void Queue\_remove(Queue\_Elem \*qelem)**

qelem – 队列中的元素地址

此函数将从队列中删除指定的元素

示例代码

```
Queue_enqueue(myQ, &(r1.elem));  
/* Insert r2 in front of r1 in the Queue. */  
Queue_insert(&(r1.elem), &(r2.elem));  
/* Remove r1 from the Queue. Note that Queue_remove() does not  
 * require a handle to myQ. */  
Queue_remove(&(r1.elem));
```

## 例 8 队列实验

例 8 中，将例 7 传递按钮信息方式换成采用队列方式。例 7 中传递按钮信息是用全局变量临时保存，但是使用全局变量会将函数变得不可重入，所以在本例实验是将其改成队列方式。但是在使用队列，发现其本身不提供传递信息内存空间，所以这部分空间需要我们自己分配。具体的 IAR 工程代码见 **3.12\_Queue\_Create/**目录下。此处不贴代码了，具体的说明可以看一下例 7 实验代码说明。

## 3.4 邮箱（MailBoxs）

SYS/BIOS 内核提供一组函数来操作邮箱对象，这些函数通过类型为 Mailbox\_Handle 的句柄访问邮箱对象并对其进行操作。邮箱可以将数据从一个任务传给另一个任务。其在创建时会指定消息大小及最大能存放多少个消息（即邮箱大小）。

Mailbox\_pend ()用于等待来自邮箱的邮件。Mailbox\_pend 的 timeout 参数允许任务等待直到超时。如果 timeout 是 BIOS\_WAIT\_FOREVER，Mailbox\_pend 使任务会无限期地等待消息。如果 timeout 是 BIOS\_NO\_WAIT，Mailbox\_pend 会立即返回。Mailbox\_pend 的返回值指示邮箱是否成功发信号。返回 TRUE 表示成功，返回 FALSE 表示超时。如果 Mailbox\_pend 收到邮件，其会从邮箱中将消息拷贝出来。

当邮箱注册了一个 readerEvent 事件对象并且任务已经从 Event\_pend() 返回，BIOS\_NO\_WAIT 会传递给 Mailbox\_pend()以检索该消息。

注意：由于邮箱只有一个 readerEvent 事件对象可以注册，所以配置了 readerEvent 事件对象的邮箱不支持多个读取者。

Mailbox\_post()用于向邮箱发送邮件。Mailbox\_post 的 timeout 参数指定了如果邮箱已满，则调用任务等待的时间。

当邮箱注册了 writerEvent 事件对象，并且一个任务已经从 Event.post()返回，那么 BIOS\_NO\_WAIT 应该被传递给 Mailbox\_post()，指示消息被成功发布。

注意：由于邮箱只有一个 writerEvent 事件对象可以注册，所以配置了一个 writerEvent 事件对象的邮箱不支持多个写入者。

### 3.4.1 创建邮箱

SYS/BIOS 提供了创建邮箱 API 函数。下将一一介绍具体 API 函数使用。

**1. Mailbox\_Handle Mailbox\_create(SizeT msgSize, UInt numMsgs, const Mailbox\_Params \*params, Error\_Block \*eb)**

msgSize – 一个消息大小（所占用的空间大小）

numMsgs – 最大可以存放多少条消息，即邮箱长度

params – 配置邮箱参数指针。如果为 NULL，将采用默认参数

eb – 错误处理块指针。如果为 NULL，将采用默认方法

返回值 – 如果成功，则返回邮箱对象指针；如果失败，则返回 NULL。

➤ 详细说明

Mailbox\_create 创建邮箱对象，并将分配邮箱空间。空间大小为（msgSize \* numMsgs）

**2. Void Mailbox\_delete(Mailbox\_Handle \*handleP)**

handleP – 邮箱对象解句柄指针

**3. Void Mailbox\_construct(Mailbox\_Struct \*structP, SizeT msgSize, UInt numMsgs, const Mailbox\_Params \*params, Error\_Block \*eb)**

structP – Mailbox\_Struct 类型变量地址

msgSize – 一个消息大小（所占用的空间大小）

numMsgs – 最大可以存放多少条消息，即邮箱长度

params – 配置邮箱参数指针。如果为 NULL，将采用默认参数

eb – 错误处理块指针。如果为 NULL，将采用默认方法

➤ 详细说明

此函数与 Mailbox\_create 作用相同

**4. Void Mailbox\_destruct(Mailbox\_Struct \*structP)**

structP – 已经创建的邮箱对象 Struct 结构指针

**5. Void Mailbox\_Params\_init(Mailbox\_Params \*params)**

params – 邮箱对象配置参数指针

➤ 详细说明

此函数将传入的配置参数，初始为默认参数

下面示例代码，演示利用上述函数创建邮箱对象。

```
typedef struct msg
{
    UInt id;
    Char buf[10]
}msg;
Mailbox_Handle mbox = NULL;
Mailbox_Params mboxParams;
Error_Block eb;
Error_init(&eb);
Mailbox_Params_init(&mboxParams);
// 创建邮箱对象，并将邮箱长度初始化为 10
mbox = Mailbox_create(sizeof(msg),10,&mboxParams,eb);
if(mbox == NULL)
{
    //创建邮箱对象失败
    .....
}
```

### 3.4.2 发送接收邮件

邮箱是在多任务间传递消息，所以消息的发布与获取是邮箱对象最基本操作，下面将介绍其相应的 API 函数。

#### 1. Bool Mailbox\_pend(Mailbox\_Handle handle, Ptr msg, UInt32 timeout)

handle – 邮箱对象句柄

msg – 消息指针

timeout – 最大超时时间

返回值 – TRUE 表示获得消息成功

FALSE 表示超时

➤ 详细说明

Mailbox\_pend 是从邮箱获得消息。如果邮箱不空，此函数将邮箱第一个消息拷贝出来，返回 TRUE。否则，此函数将挂起当前任务直到邮箱有消息或者发生超时事件。

timeout = BIOS\_WAIT\_FOREVER，将无限期挂起任务，直到有消息

= BIOS\_NO\_WAIT，不管邮箱是否有消息，函数都将立刻返回。通过返回值来判断获取消息是否成功。

#### 2. Bool Mailbox\_post(Mailbox\_Handle handle, Ptr msg, UInt32 timeout)

handle – 邮箱对象句柄

msg – 消息指针

timeout – 最大超时时间

返回值 – TRUE 表示获得消息成功

FALSE 表示超时

➤ 详细说明

Mailbox\_post 在将消息拷贝到邮箱之前会对邮箱时行检查，看是否有空的位置。如果邮箱有等待的任务，则 post 就绪第一个等待的任务并根据优先级决定是否发生任务切换。如果当前邮箱满时，timeout 参数将会起作用。如果在 timeout 时间内邮箱有空位置出现，post 消息拷贝到邮箱，并返回 TRUE，否则只能超时，返回 FALSE。

#### 3. SizeT Mailbox\_getMsgSize(Mailbox\_Handle handle)

handle – 邮箱对象句柄

返回值 – 邮箱对象单个消息所占用的空间大小

#### 4. Int Mailbox\_getNumFreeMsgs(Mailbox\_Handle handle)

handle – 邮箱对象句柄

返回值 – 邮箱对象当前空闲消息个数

#### 5. Int Mailbox\_getNumPendingMsgs(Mailbox\_Handle handle)

handle – 邮箱对象句柄

返回值 – 邮箱对象当前内部消息个数，即被占用了多少个消息

## 例 9 邮箱实验

例 9 实验中，利用邮箱对象在任务间传递信息。其中任务 2 等待按钮按下事件，并根据具体按钮处理按钮记数值。随后将按钮按下记数值通过邮箱对象传递给另任务 1。其中



Button1 会将按钮记数加 1，而 Button2，会将按钮记数减 1。具体的代码见 **3.13\_Mailbox\_Create/**目录下。

```
int main(void)
{
    .....
    .....

    USER_createStartTask();    //创建启动任务
    /* Start BIOS */
    BIOS_start();

    return (0);
}

//启动任务函数
static Void StartTaskFunc(UArg a0,UArg a1)
{
    .....
    Board_initKeys(ButtonHandleFunc);
    //初始化串口驱动
    UART_DriverInit();
    .....
    EVENT_Create();
    Mailbox_Create();
    //创建一个工作任务
    USER_createWorkTask(1,workTask1Func,&workTask1);
    USER_createWorkTask(2,workTask2Func,&workTask2);

    Task_destruct(&startTask);
}

//按钮回调函数
static void ButtonHandleFunc(UChar keys)
{
    UInt eventId = Event_Id_NONE;
    if(keys & KEY_BTN1)
    {
        eventId |= Event_Id_00;
    }
    if(keys & KEY_BTN2)
    {
        eventId |= Event_Id_01;
    }
    Event_post(myEventHandle,eventId);
}
```

```
}  
//任务1 函数  
static Void workTask1Func(UArg a0,UArg a1)  
{  
    .....  
    for(;;)  
    {  
        .....  
        Mailbox_pend(myMailHandle, &tmpMsg ,BIOS_WAIT_FOREVER);  
        .....  
    }  
}  
static Void workTask2Func(UArg a0,UArg a1)  
{  
    .....  
    for(;;)  
    {  
        //andMask = None  
        eventId = Event_pend(myEventHandle,Event_Id_NONE,  
Event_Id_00|Event_Id_01, BIOS_WAIT_FOREVER);  
  
        if(eventId & Event_Id_00)  
        {  
            keyInfo = "Key1 pressed";  
            keys.key++;  
        }  
        if(eventId & Event_Id_01)  
        {  
            keyInfo = "Key2 pressed";  
            keys.key--;  
        }  
        if(keyInfo)  
        {  
            .....  
            Mailbox_post(myMailHandle,&keys,BIOS_NO_WAIT);  
        }  
        keyInfo = NULL;  
    }  
}
```

## 第四章 HWI

由于 TI 将硬件封装，HWI 部分用户很难直接使用，所以这里只作介绍不详细说明。

硬件中断（Hwi）处理应用程序必须响应的外部异步事件。在 SYS / BIOS 中的 Hwi 模块用于管理硬件中断。在典型的嵌入式系统中，硬件中断由设备外设或者通过触发处理器外部的设备。在这两种情况下，中断使处理器跳向量中的 ISR 地址。

如果其在 ISR 有机会执行之前，有多次相同 Hwi 触发，那么 ISR 只会运行一次。执行 Hwi 运行的代码，应该最小化执行的代码量。如果中断全局启用，即通过调用 Hwi\_enable() - 可以通过任何方式抢占 ISR。通过创建 Hwi 对象来将 ISR 功能与特定中断相关联。

### 4.1 创建 HWI

Hwi 模块维护一个指向 Hwi 对象的指针表，其中包含有关每个 Hwi 的信息由调度员管理。要动态创建 Hwi 对象，请使用类似于以下内容的调用：

```
Hwi_Handle hwi0;
Hwi_Params hwiParams;
Error_Block eb;
Error_init(&eb);
Hwi_Params_init(&hwiParams);
hwiParams.arg = 5;
hwi0 = Hwi_create(id, hwiFunc, &hwiParams, &eb);
if (hwi0 == NULL) {
    System_abort("Hwi create failed");
}
```

这里，hwi0 是创建的 Hwi 对象的句柄，id 是定义的中断号，hwiFunc 是与 Hwi 相关联的函数的名称，hwiParams 是包含 Hwi 实例的结构参数（启用/恢复掩码，Hwi 函数参数等）。这里，hwiParams.arg 设置为 5。如果将 NULL 传递给实际的 Hwi\_Params 结构体的指针，使用默认的参数集。“eb”是一个错误块，可用于处理 Hwi 对象创建过程中可能发生的错误。相应的静态配置 Hwi 对象创建语法是：

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params;
hwiParams.arg = 5;
Program.global.hwi0 = Hwi.create(id, '&hwiFunc', hwiParams);
```

在这里，“hwiParams = new Hwi.Params”语句相当于创建和初始化 hwiParams 结构具有默认值。在静态配置世界中，不需要错误块（eb）为“创建”功能。“Program.global.hwi0”名称成为一个运行时可访问的句柄（符号 name = “hwi0”）到静态创建的 Hwi 对象。

### 4.2 硬件中断嵌套和系统堆栈大小

当 Hwi 运行时，使用系统栈调用其功能。在最坏的情况下，每个 Hwi 都可以产生在调

度功能的嵌套中（即，最低优先级 Hwi 被抢占次高优先级的 Hwi，反过来又被下一个最高级别抢占，...）。这导致每一个 Hwi 优先级实际使用堆栈大小要增加。

默认系统堆栈大小为 4096 字节。您可以通过添加以下内容来设置系统堆栈大小到你的配置脚本：

```
Program.stack = yourStackSize;
```

## 4.3 Hwi hooks

Hwi 模块支持以下 Hook 功能：

- **Register**

函数会在任何静态创建的 Hwis 初始化之前被调用。注册的 hook 函数在启动时间进入 main 函数之前且使能中断之前被调用。

- **Create**

函数在 Hwi 创建时调用。这包括静态创建或使用 Hwi\_create() 动态创建。

- **Begin**

在运行 Hwi ISR 功能之前调用的函数。

- **End**

在 Hwi ISR 功能完成之后调用的函数。

- **Delete**

Hwi 在 Hwi\_delete() 运行时被删除时调用的函数。

以下 HookSet 结构类型定义封装了 Hwi 支持的钩子函数：

```
typedef struct Hwi_HookSet
{
    Void (*registerFxn) (Int); /* Register Hook */
    Void (*createFxn) (Handle, Error.Block *); /* Create Hook */
    Void (*beginFxn) (Handle); /* Begin Hook */
    Void (*endFxn) (Handle); /* End Hook */
    Void (*deleteFxn) (Handle); /* Delete Hook */
};
```

Hwi Hook 功能只能静态配置。

## 4.4 Register 函数

注册函数提供了允许钩子集存储其相应的挂钩 ID。此 ID 可以传递给 Hwi\_setHookContext() 和 Hwi\_getHookContext() 来设置或获取特定于钩子的上下文。如果 hook 实现需要使用 Hwi\_setHookContext() 或者 Hwi\_getHookContext()，则必须指定注册函数。

registerFxn 钩子函数，在启用中断之前，在系统初始化期间调用。

```
Void registerFxn (Int id);
```

## 4.5 Create/Delete 函数

Create 函数与 Delete 函数在 Hwi 对象被创建或删除时被调用。createFxn 和 deleteFxn 函数在启用中断的情况下被调用（除非在启动时调用或 from main（））。

```
Void createFxn(Hwi_Handle hwi, Error_Block *eb);  
Void deleteFxn(Hwi_Handle hwi);
```

## 4.6 Begin/End 函数

Begin 和 End hook 函数在中断全局禁用的情况下调用。结果，任何 hook 处理函数增加整个系统中断响应延迟。为了最小化这个影响，仔细考虑在 Hwi beginFxn 或 endFxn 钩子功能中花费的处理时间。

在调用 ISR 函数之前调用 beginFxn。之后立即调用 endFxn 从 ISR 功能返回。

```
Void beginFxn(Hwi_Handle hwi);  
Void endFxn(Hwi_Handle hwi);
```

# 第五章 SWI

SYS / BIOS 中的 Swi 模块提供了一个软件中断能力。通过对 SYS / BIOS API 的调用程序触发软件中断如 Swi\_post()。软件中断的优先级高于任务但低于硬件中断。

注意：Swi 模块不应该与许多处理器的 SWI 指令混淆。SYS / BIOS Swi 模块独立于任何目标/设备特定的模块软件中断功能。

Swi 线程适用于处理以较慢速率发生的应用程序任务或对实时性要求低于 Hwis。

SYS/BIOS 中可以触发或发布 Swi 的 API 函数有以下几个：

- Swi\_andn()
- Swi\_dec()
- Swi\_inc()
- Swi\_or()
- Swi\_post()

Swi 管理器控制所有 Swi 函数的执行。当应用程序调用其中一个 API，Swi 管理器调度与指定的 Swi 相对应的函数执行。Swi 管理器使用 Swi 对象，去执行 Swi 函数。如果一个 Swi 被触发，它只能在所有被挂起的 Hwis 运行完之后运行。正在进行的 Swi 函数可以任何时候补 Hwi 中断；另一方面，Swi 函数总是抢占任务，即使最高优先级任务。实际上，Swi 就像一个优先级大于所有普通任务的任务。

## 5.1 创建 SWI 对象

与许多其他 SYS / BIOS 对象一样，您可以动态地创建 Swi 对象，通过调用 Swi\_create() 或静态配置。动态创建的 Swis 也可以被删除。要将新的 Swi 添加到配置中，请在配置脚本中创建一个新的 Swi 对象。为每个 Swi 对象设置函数属性，此函数将在应用程序触发对象时运行。你也可以最多配置两个参数以传递给每个 Swi 函数。与其他模块实例一样，你可以

决定 Swi 对象内存分配。当 Swis 被触发时，Swi 管理器通过 Swi 对象执行相应的中断函数。

动态创建 Swi 对象，可以使用下述方法：

```
Swi_Handle swi0;
Swi_Params swiParams;
Error_Block eb;
Error_init(&eb);
Swi_Params_init(&swiParams);
swi0 = Swi_create(swiFunc, &swiParams, &eb);
if (swi0 == NULL) {
    System_abort("Swi create failed");
}
```

这里，swi0 是创建的 Swi 对象的句柄，swiFunc 是与该对象关联的函数的名称 Swi 和 swiParams 是包含 Swi 实例参数的 Swi\_Params 类型的结构（priority, arg0, arg1 等）。如果传递 NULL 而不是指向实际的 Swi\_Params 结构体的指针，则为默认值使用一组参数。“eb”是一个错误块，可用于处理在 Swi 对象创建期间可能发生的错误。

在配置文件中创建 SWI 对象，可以使用下述代码：

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var swiParams = new Swi.Params();
program.global.swi0 = Swi.create(swiParams);
```

### 1. Swi\_Handle Swi\_create(Swi\_FuncPtr swiFxn, const Swi\_Params \*params, Error\_Block \*eb)

swiFxn – 是 swi 对象中断函数指针。其函数指针类型为 typedef Void (\*Swi\_FuncPtr)(UArg,UArg)。

params – swi 对象配置参数指针，如果为 NULL 将采用默认参数

eb – 错误处理模块，如果为 NULL 则使用默认错误处理

返回值—如果成功，将返回 SWI 对象实例句柄。如果失败返回 NULL。

此函数将创建一个 Swi 对象。

```
Swi_Params swiParams;
Swi_Params_init(&swiParams);
swiParams.arg0 = 1;
swiParams.arg1 = 0;
swiParams.priority = 2;
swiParams.trigger = 0;
swi0 = Swi_create(swi0Fxn, &swiParams, NULL);
```

### 2. Void Swi\_delete(Swi\_Handle \*handleP)

handleP – swi 对象句柄指针

此函数将删除 swi 对象实例

```
Swi_delete(&swi0);
```

### 3. Void Swi\_construct(Swi\_Struct \*structP, Swi\_FuncPtr swiFxn, const Swi\_Params \*params, Error\_Block \*eb)

structP – 是 Swi\_Struct 结构指针。

其他参数与 Swi\_create()函数相同。

此函数与 `Swi_create()` 函数相当，具有相同的作用，都是创建一个 `swi` 对象。

#### 4. `Void Swi_destruct(Swi_Struct *structP)`

`structP` – `swi` 对象 `Struct` 结构指针。

此函数与 `construct` 函数成对使用。

#### 5. `Swi_Handle Swi_handle(Swi_Struct *structP);`

与

`Swi_Struct *Swi_struct(Swi_Handle handle);`

此两个函数可以将 `Swi_Handle` 与 `Swi_Struct` 结构指针进行转换。

## 5.2 SWI 对象优先级及系统堆栈大小

`swi` 对象有不同优先级。你可以为每个优先级创建一个 `swi` 对象，只要你的内存不受限制。你可以为 `swi` 对象选择高优先级，去处理实时性要求高的任务，而低优先级可以处理实时性要求不高的任务。

应用程序内支持的 `Swi` 优先级最多可配置 32 个。默认优先级为 16 最低优先级为 0。因此，默认情况下，最高优先级为 15。同时，不能以优先级对 `Swis` 进行排序。他们按照他们触发的顺序进行执行。

当 `Swi` 触发时，使用系统堆栈调用其关联的 `Swi` 函数。虽然你可以拥有在一些目标上最多可达 32 个 `Swi` 优先级别，请记住，在最坏的情况下，每个 `Swi` 优先级别可以导致 `Swi` 调度功能的嵌套（即，最低优先级的 `Swi` 被抢占下一个最高优先级的 `Swi`，反过来又被下一个最高优先级抢占，...）。这导致增加实际使用的每个 `Swi` 优先级的堆栈大小要求。

默认系统堆栈大小为 4096 字节。在配置文件中，您可以通过添加以下内容来设置系统堆栈大小。

```
Program.stack = yourStackSize;
```

## 5.3 SWI 中断执行

可以调用 `Swis` 对象来执行 `Swi_andn()`，`Swi_dec()`，`Swi_inc()`，`Swi_or()` 和 `Swi_post()`。这些函数几乎可以在程序的任何地方使用，比如 `Hwi` 函数，`Clock` 函数，空闲函数或其他 `Swi` 函数。

当 `Swi` 被触发时，`Swi` 管理器将其添加到待执行的已触发的 `Swis` 列表中。该 `Swi` 管理器检查是否启用了 `Swis`。如果他们不是，`Swi` 管理器将控制权返回给当前线程。如果启用了 `Swis`，则 `Swi` 管理器根据优先级检查已触发的 `Swi` 对象与当前正在运行的线程的优先级。如果当前正在运行的线程是背景空闲循环，一个任务，或者较低优先级的 `Swi`，则 `Swi` 管理器会将 `Swi` 对象从已触发的列表中删除，从当前线程中剥夺 CPU 控制权执行触发的 `Swi` 函数。如果当前运行的线程是具有相同或更高优先级的 `Swi`，则 `Swi` 管理器将控制权返回给当前线程，并且执行之前触发的高优先级或相同优先级的 `Swi` 函数。如果触发的多个 `Swi` 对象具有相同的优先级，则按照它们触发的顺序执行。

有两个重要的事情：

- 当 `Swi` 开始执行时，必须保证它顺利运行完成而不会被阻止。
- 当从硬件 `ISR` 中调用时，代码调用任何可以触发或发布的 `Swi` 函数，`Swi` 必须由 `Hwi` 调度程序调用。就是说，`Hwi` 对象调用可以触发 `Swi` 函数。

**Swi** 函数数可以被高优先级的线程抢占，例如 **Hwi** 或高优先级的 **Swi**。不管怎样，**Swi** 函数不能被阻塞。你不能挂起 **Swi**，去等待一些事情，像设备就绪。在 **Swi** 管理器从触发列表中删除 **Swi** 之前，**Swi** 已经被多次触发，那么 **Swi** 函数只被执行一次。

**Swi** 操作函数有以下个：

**1. Void Swi\_post(Swi\_Handle handle)**

handle – swi 对象句柄

此函数无条件触发一次软中断。它不会修改 **Swi** 对象的触发值。

**2. Void Swi\_or(Swi\_Handle handle, UInt mask)**

handle – swi 对象句柄

mask – 与 swi 对象相或的值

此函数用于触发软中断，并通过 mask 参数将触发值进行相或。**Swi\_or** 触发软中断是不管触发值的结果的。其逻辑表达式为  $trigger = trigger | mask$ 。

**3. Void Swi\_inc(Swi\_Handle handle)**

handle – swi 对象句柄

函数将触发值加 1，并触发一次软中断。**Swi\_inc** 在触发软中断是不管触发值的。

**4. Void Swi\_andn(Swi\_Handle handle, UInt mask)**

handle – swi 对象句柄

mask – 取反值将与触发值相与

此函数将复位 swi 对象触发值的指定位，并在触发值变为 0 后，触发软中断。所以 **Swi\_andn** 是一个有条件的触发。其逻辑运算为  $trigger = trigger \&(\sim mask)$ ;

**5. Void Swi\_dec(Swi\_Handle handle)**

handle – swi 对象句柄

函数将触发值减 1，如果触发值变成了 0 将触发一次软中断。

**6. UInt Swi\_getTrigger()**

返回当前执行的 **Swi** 对象的触发值。

下面将演示 **Swi\_inc** 函数与 **Swi\_getTrigger** 函数

通过使用 **Swi\_inc()** 函数触发了一个 **Swi**，一旦 **Swi** 管理器执行了 **Swi** 函数，可以通过 **getTrigger** 知道在 **Swi** 函数执行之前已经被触发了多少次。知道了触发次数后，可以把同一函数连续执行相应次数。如下图所示



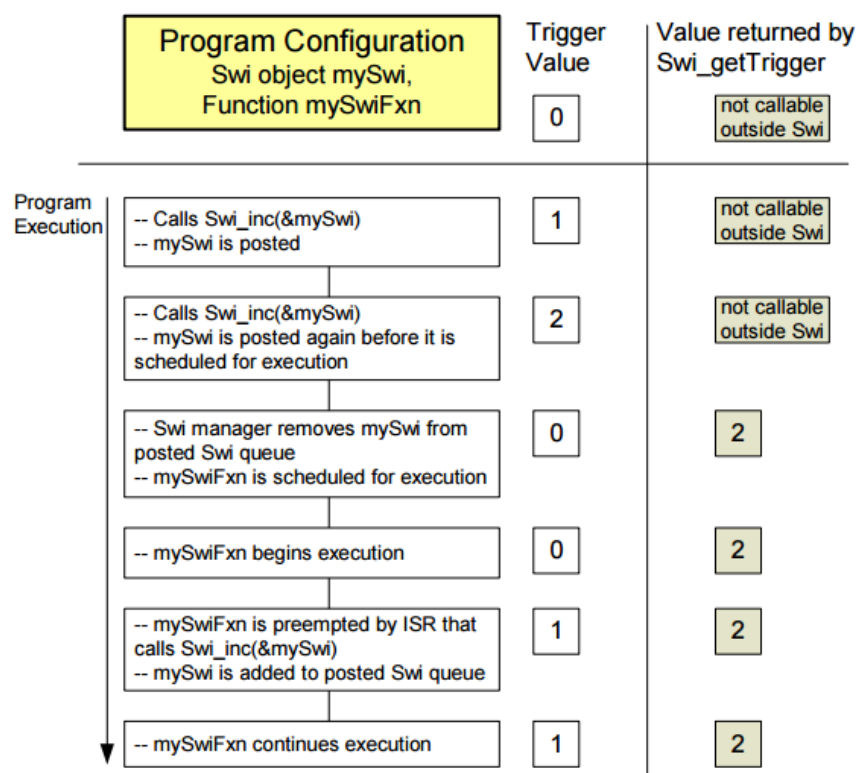


图 6 Swi\_inc 触发 Swi

```
mySwiFxn()
{
    . . .
    repetitions = SWI_getTrigger();
    while (repetitions --)
    {
        //run Swi function
    }
    . . .
}
```

如果多个事件发生后才会触发 Swi，那么应该使用 Swi\_andn()函数去触发相应 Swi 对象。如图 7 所示。例如，如果 Swi 必须等待来自两个不同设备的数据输入才能继续。那么 Swi 对象被配置时，其触发器应该有两个位。当提供输入数据的两个函数都完成了任务时，他们应该都使用 Swi\_andn()，以清除在 Swi 触发器默认值中设置的每个位值。因此，只有当两个进程的数据准备就绪时，才会发布 Swi。

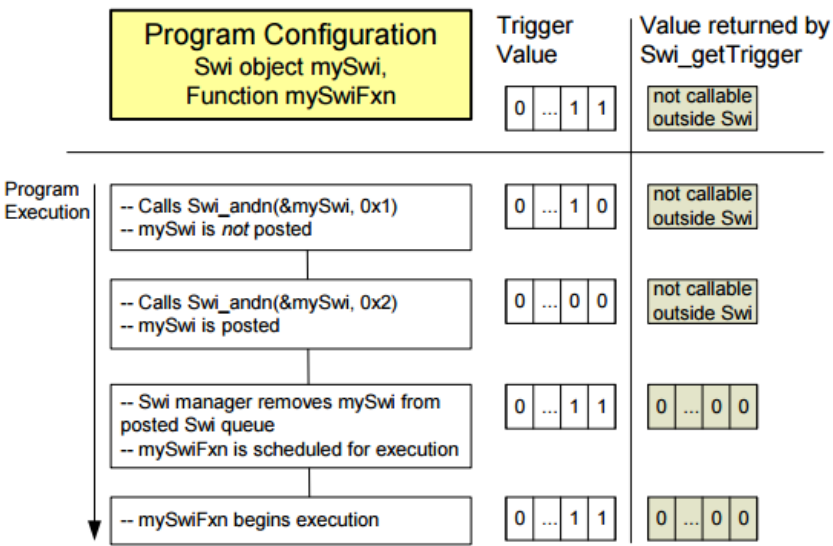


图 7 Swi\_andn()触发 Swi

如果程序执行需要同一事件的多次发生才会触必 Swi，那么 Swi\_dec()被用着触发 Swi，如图 8 所示。通过配置 Swi 触发器等于在 Swi 发布和调用之前事件的发生次数，每次事件发生时，Swi\_dec()将触发值减 1，只有在触发器达到 0 之后才会发布 Swi。

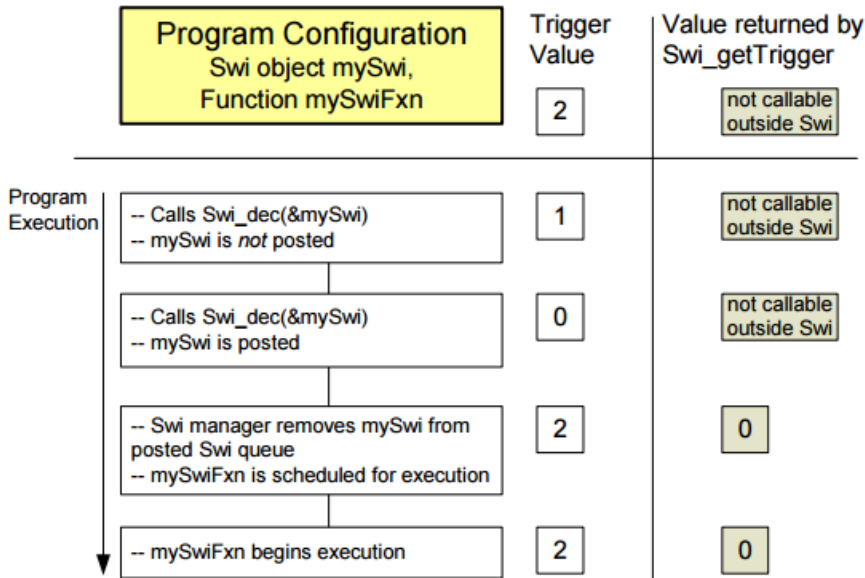


图 8 Swi\_dec 触发 Swi

在某些情况下，Swi 函数可以根据发布的事件调用不同的函数。在那个程序中可以使用 Swi\_or()对事件发生时无条件地发布 Swi 对象。如图 9 所示。Swi\_or()使用的位掩码的值对触发值进行修改操作。Swi 触发值可以被 Swi 函数用作标识事件的标志用于选择执行的功能。

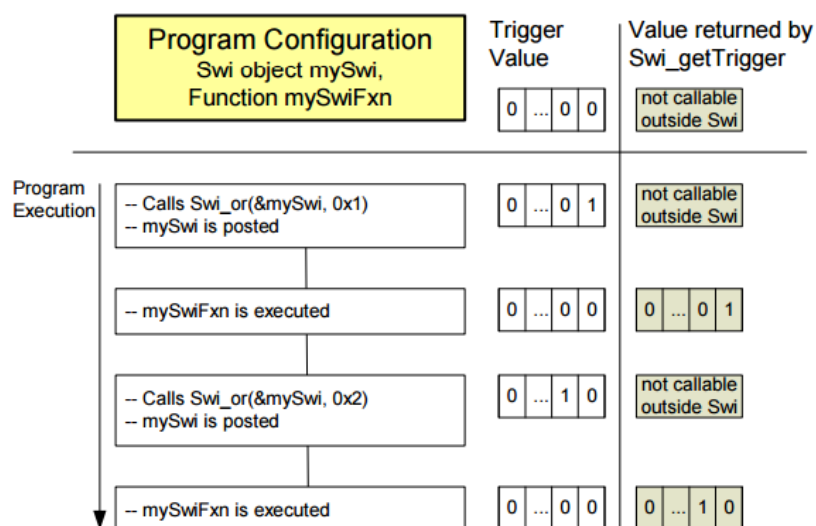


图 9 Swi\_or 触发 Swi

```

mySwiFxn()
{
    ...
    eventType = Swi_getTrigger();
    switch (eventType)
    {
        case '0x1': 'run alg 1'
        case '0x2': 'run alg 2'
        case '0x3': 'run alg 3'
    }
    ...
}

```

## 例 10 SWI 实验

本例中，利用 Swi\_andn, Swi\_dec, Swi\_post API 函数功能，演示了这些函数之间的区别。并在 swi 中断回调函数中，对中断触发方式控制，使任务中调用不同的 API 触发函数。同时在任务中，依据不同的触发方式控制 LaunchIOT 平台外设。详细的工程代码在 **3.16\_SWI/**目录下。

下面示例代码只是工程中部分代码，这里只是对 swi 相关代码进行说有。

```

//创建一个 SWI 对象实例
void SWI_Create(void)
{
    Swi_Params swiParams;
    Swi_Params_init(&swiParams);
    swiParams.arg0 = 0;
    swiParams.arg1 = 0;
    swiParams.trigger = 3; //默认 trigger 值为 3
    swiParams.priority = 2;
}

```

```
Swi_construct(&swiStruct, swi_Fxn, &swiParams, NULL);
}

//用户任务函数
Void heartBeatFxn(UArg arg0, UArg arg1)
{
    UInt mask = 0;
    SWI_Create(); //创建一个 swi 对象

    while (1)
    {
        Task_sleep((UInt)arg0); //利用 task_sleep 进行延时
        //trigger a swi
        if(swi_trigger == POST)
        {
            mask = 0;
            PIN_setOutputValue(ledPinHandle, Board_LED0,
                               !PIN_getOutputValue(Board_LED0));
            //无条件触发, 不会改变 trigger 值
            Swi_post(Swi_handle(&swiStruct));

        }
        else if(swi_trigger == DEC)
        {
            mask = 0;
            PIN_setOutputValue(ledPinHandle, Board_LED1,
                               !PIN_getOutputValue(Board_LED1));
            //将 trigger 值减 1, 如果 trigger 值变成 0, 将触发中断
            Swi_dec(Swi_handle(&swiStruct));

        }
        else
        {
            mask++;
            PIN_setOutputValue(ledPinHandle, Board_MON,
                               !PIN_getOutputValue(Board_MON));
            //mask 对 trigger 位值进行复位, 如果 trigger 变成 0, 触发中断
            Swi_andn(Swi_handle(&swiStruct), mask);

        }

    }
}

//中断回调函数
```

```
static void swi_Fxn(UArg a0, UArg a1)
{

    //对触发方式进行更改
    if(swi_trigger == POST)
    {
        swi_trigger = DEC;
    }
    else if(swi_trigger == DEC)
    {
        swi_trigger = ANDN;
    }
    else
    {
        swi_trigger = POST;
    }
}
```

## 5.4 同步 Swi 函数

在空闲任务，或 SWI 函数中，你可以通过调用 `Swi_disable()` 临时禁止被高优先级 Swi 对象抢占，它将失能所能 Swi 抢占。重新使能抢占，可以调用 `Swi_restore()`。Swi 使能与失能是作为一个组。一个独立的 Swi 是不能使能或失能它自己。

当 SYS/BIOS 完成初始化，且在第一个任务被调用之前，Swi 已经使能。如果应用程序希望禁用 SWI，可以调用 `Swi_disable()`。如下：

```
key = Swi_disable();
```

相应的使能函数是 `Swi_restore()`，它的参数 `key` 是 `Swi_disable()` 返回值。

```
Swi_restore(key);
```

`Swi_disable()/Swi_enable()` 允许嵌套调用。因为只有最外层的 `Swi_restore()` 调用实际上使 Swis。换句话说，任务可以禁用并启用 Swis，而无需确定是否已经在别处调用了 `Swi_disable()`。当 Swis 被禁用，触发函数不会立刻运行。中断被“锁定”在软件中，只有 Swis 被使能之后才会运行。

## 5.5 Swi hooks

SWI 模块支持以下 Hook 函数功能：

- **Register**  
函数会在任何静态创建的 Swis 初始化之前被调用。注册的 hook 函数在启动时间进入 main 函数之前且使能中断之前被调用。
- **Create**  
函数在 Swi 创建时调用。这包括静态创建或使用 `Hwi_create()` 动态创建。
- **Begin**  
在运行 Hwi ISR 功能之前调用的函数。

- **End**

在 Swi ISR 功能完成之后调用的函数。

- **Delete**

Swi 在 Swi\_delete()运行时被删除时调用的函数。

以下 Swi\_HookSet 结构类型定义封装了 Swi 支持的钩子函数：

```
typedef struct Hwi_HookSet
{
    Void (*registerFxn) (Int); /* Register Hook */
    Void (*createFxn) (Handle, Error.Block *); /* Create Hook */
    Void (*readyFxn) (Handle); /* Ready Hook */
    Void (*beginFxn) (Handle); /* Begin Hook */
    Void (*endFxn) (Handle); /* End Hook */
    Void (*deleteFxn) (Handle); /* Delete Hook */
};
```

Swi Hook 功能只能静态配置。

当多个 HookSet 被定义，通用类型的独立 hook 函数以 hook Id 顺序调用。

## 5.6 Register 函数

注册函数提供了允许钩子集存储其相应的挂钩 ID。此 ID 可以传递给 Swi\_setHookContext() 和 Swi\_getHookContext() 来设置或获取特定于钩子的上下文。如果 hook 实现需要使用 Swi\_setHookContext() 或者 Swi\_getHookContext()，则必须指定注册函数。

registerFxn 钩子函数，在启用中断之前，在系统初始化期间调用。

```
Void registerFxn(Int id);
```

## 5.7 Create/Delete 函数

Create 函数与 Delete 函数在 Swi 对象被创建或删除时被调用。

createFxn 和 deleteFxn 函数在启用中断的情况下被调用（除非在启动时调用或 from main()）。

```
Void createFxn(Swi_Handle hwi, Error_Block *eb);
Void deleteFxn(Swi_Handle hwi);
```

## 5.8 Ready/Begin/End 函数

Ready, Begin 和 End hook 函数在中断使能时被调用。readyFxn 函数在 Swi 被触发且准备就绪运行时被调用；beginFxn 函数在 swi 函数运行之前补调用；endFxn 函数是在 Swi 函数返回之后被调用。

readyFxn 和 beginFxn 两函数被提供，主要是因为 Swi 已经触发且已经就绪但仍然被挂起

等待高优先级线程完成。

```
Void readyFxn(Swi_Handle swi);
Void beginFxn(Swi_Handle swi);
Void endFxn(Swi_Handle swi);
```

## 例 11 SWI hooks 例子

例子中应用程序使用了两个 Swi hook。它们演示了如何去读取和写入每个 Hook 关联的上下文。其中用到了两个 Swi 上下文操作的函数。

### 1. Ptr Swi\_getHookContext(Swi\_Handle handle, Int id)

handle – swi 对象句柄

id – 是 SWI 模块在注册 hook 函数时分配的 hook ID

返回值 – SWI 的 hook 实例上下文指针

此函数是从 SWI 中获得 hook 实例上下文指针。

下面的例子中，获得 hookContext，并且打印，同时再设置一个新的值

```
Ptr pEnv;
Swi_Handle mySwi;
Int myHookSetId1;

pEnv = Swi_getHookContext(swi, myHookSetId1);

System_printf("myEnd1: pEnv = 0x%lx, time = %ld\n",
              (ULong)pEnv, (ULong)Timestamp_get32());

Swi_setHookContext(swi, myHookSetId1, (Ptr)0xc0de1);
```

### 2. Void Swi\_setHookContext(Swi\_Handle handle, Int id, Ptr hookContext)

handle – swi 对象句柄

id --是 SWI 模块在注册 hook 函数时分配的 hook ID

hookContext – 写入上下文值

此函数是将新的值写入到 swi 对象 hook 上下文中。

swi hook 例子代码如下。

```
/* ===== SwiHookExample.c =====
 * This example demonstrates basic Swi hook usage */
#include <xdc/std.h>
#include <xdc/runtime/Error.h>
#include <xdc/runtime/System.h>
#include <xdc/runtime/Timestamp.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/hal/Timer.h>
#include <ti/sysbios/knl/Swi.h>
Swi_Handle mySwi;
Int myHookSetId1, myHookSetId2;
/* HookSet 1 functions */
```

```
/* ===== myRegister1 =====
 * invoked during Swi module startup before main
 * for each HookSet */
Void myRegister1(Int hookSetId)
{
    System_printf("myRegister1: assigned hookSet Id = %d\n", hookSetId);
    myHookSetId1 = hookSetId;
}

/* ===== myCreatel =====
 * invoked during Swi_create for dynamically created Swis */
Void myCreatel(Swi_Handle swi, Error_Block *eb)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId1);
    /* pEnv should be 0 at this point. If not, there's a bug. */
    System_printf("myCreatel: pEnv = 0x%x, time = %d\n",
        pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xdead1);
}

/* ===== myReady1 =====
 * invoked when Swi is posted */
Void myReady1(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId1);
    System_printf("myReady1: pEnv = 0x%x, time = %d\n",
        pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xbeef1);
}

/* ===== myBegin1 =====
 * invoked just before Swi func is run */
Void myBegin1(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId1);
    System_printf("myBegin1: pEnv = 0x%x, time = %d\n",
        pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId1, (Ptr)0xfeeb1);
}

/* ===== myEnd1 =====
 * invoked after Swi func returns */
Void myEnd1(Swi_Handle swi)
{
    Ptr pEnv;
```



```
pEnv = Swi_getHookContext(swi, myHookSetId1);
System_printf("myEnd1: pEnv = 0x%x, time = %d\n",
pEnv, Timestamp_get32());
Swi_setHookContext(swi, myHookSetId1, (Ptr)0xc0de1);
}
/* ===== myDelete1 =====
 * invoked upon Swi deletion */
Void myDelete1(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId1);
    System_printf("myDelete1: pEnv = 0x%x, time = %d\n",
pEnv, Timestamp_get32());
}
/* HookSet 2 functions */
/* ===== myRegister2 =====
 * invoked during Swi module startup before main
 * for each HookSet */
Void myRegister2(Int hookSetId)
{
    System_printf("myRegister2: assigned hookSet Id = %d\n", hookSetId);
    myHookSetId2 = hookSetId;
}
/* ===== myCreate2 =====
 * invoked during Swi_create for dynamically created Swis */
Void myCreate2(Swi_Handle swi, Error_Block *eb)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId2);
    /* pEnv should be 0 at this point. If not, there's a bug. */
    System_printf("myCreate2: pEnv = 0x%x, time = %d\n",
pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xdead2);
}
/* ===== myReady2 =====
 * invoked when Swi is posted */
Void myReady2(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId2);
    System_printf("myReady2: pEnv = 0x%x, time = %d\n",
pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xbeef2);
}
```

```
/* ===== myBegin2 =====
 * invoked just before Swi func is run */
Void myBegin2(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId2);
    System_printf("myBegin2: pEnv = 0x%x, time = %d\n",
        pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xfeeb2);
}
/* ===== myEnd2 =====
 * invoked after Swi func returns */
Void myEnd2(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId2);
    System_printf("myEnd2: pEnv = 0x%x, time = %d\n",
        pEnv, Timestamp_get32());
    Swi_setHookContext(swi, myHookSetId2, (Ptr)0xc0de2);
}
/* ===== myDelete2 =====
 * invoked upon Swi deletion */
Void myDelete2(Swi_Handle swi)
{
    Ptr pEnv;
    pEnv = Swi_getHookContext(swi, myHookSetId2);
    System_printf("myDelete2: pEnv = 0x%x, time = %d\n",
        pEnv, Timestamp_get32());
}
/* ===== mySwiFunc ===== */
Void mySwiFunc(UArg arg0, UArg arg1)
{
    System_printf("Entering mySwi.\n");
}
/* ===== myTaskFunc ===== */
Void myTaskFunc(UArg arg0, UArg arg1)
{
    System_printf("Entering myTask.\n");
    System_printf("Posting mySwi.\n");
    Swi_post(mySwi);
    System_printf("Deleting mySwi.\n");
    Swi_delete(&mySwi);
    System_printf("myTask exiting ...\n");
}
```

```
/* ===== myIdleFunc ===== */
Void myIdleFunc()
{
    System_printf("Entering myIdleFunc().\n");
    System_exit(0);
}
/* ===== main ===== */
Int main(Int argc, Char* argv[])
{
    Error_Block eb;
    Error_init(&eb);
    System_printf("Starting SwiHookExample...\n");
    /* Create mySwi with default params
     * to exercise Swi Hook Functions */
    mySwi = Swi_create(mySwiFunc, NULL, &eb);
    if (mySwi == NULL)
    {
        System_abort("Swi create failed");
    }
    BIOS_start();
    return (0);
}
```

配置脚本如下:

```
/* pull in Timestamp to print time in hook functions */
xdc.useModule('xdc.runtime.Timestamp');
/* Disable Clock so that ours is the only Swi in the application */
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.clockEnabled = false;
var Idle = xdc.useModule('ti.sysbios.knl.Idle');
Idle.addFunc('&myIdleFunc');
/* Create myTask with default task params */
var Task = xdc.useModule('ti.sysbios.knl.Task');
var taskParams = new Task.Params();
Program.global.myTask = Task.create('&myTaskFunc', taskParams);
/* Define and add two Swi Hook Sets */
var Swi = xdc.useModule("ti.sysbios.knl.Swi");
/* Hook Set 1 */
Swi.addHookSet({
    registerFxn: '&myRegister1',
    createFxn: '&myCreate1',
    readyFxn: '&myReady1',
    beginFxn: '&myBegin1',
    endFxn: '&myEnd1',
```

```
deleteFxn: '&myDelete1'
});
/* Hook Set 2 */
Swi.addHookSet({
  registerFxn: '&myRegister2',
  createFxn: '&myCreate2',
  readyFxn: '&myReady2',
  beginFxn: '&myBegin2',
  endFxn: '&myEnd2',
  deleteFxn: '&myDelete2'
});
```

应用程序打印输出信息如下:

```
myRegister1: assigned hookSet Id = 0
myRegister2: assigned hookSet Id = 1
Starting SwiHookExample...
myCreate1: pEnv = 0x0, time = 315
myCreate2: pEnv = 0x0, time = 650
Entering myTask.
Posting mySwi.
myReady1: pEnv = 0xdead1, time = 1275
myReady2: pEnv = 0xdead2, time = 1678
myBegin1: pEnv = 0xbeef1, time = 2093
myBegin2: pEnv = 0xbeef2, time = 2496
Entering mySwi.
myEnd1: pEnv = 0xfeeb1, time = 3033
myEnd2: pEnv = 0xfeeb2, time = 3421
Deleting mySwi.
myDelete1: pEnv = 0xc0de1, time = 3957
myDelete2: pEnv = 0xc0de2, time = 4366
myTask exiting ...
Entering myIdleFunc().
```

## 第六章 门(Gates)

**Gates** 是设备禁止并发访问临界区代码。不同的 **Gates** 是在锁定临界区实现方式不同。

线程可以被其也高优先级的线程抢占,但一些代码段需要在一个线程中完成之后,才能被其他线程执行。代码修改全局变量是临界区典型的例子,它需要通过 **Gate** 保护。**Gates** 通常是失能一些级别的抢占,比如失能任务切换或甚至是硬中断,或使用 **Semaphore**。

所有 **Gates** 通过“key”支持嵌套。禁用抢占功能的 **Gates**,可能是多个线程调用 **Gate\_enter()**,直到所有任务调用 **Gate\_leave()**,抢占才会是恢复。这个功能的实施是通过使用 **key**。调用 **Gate\_enter()**返回一个 **key**,它必须被回传给 **Gate\_leave()**。只有最外层的

Gate\_enter()返回的 key 被传给 Gate\_leave(), 抢占才会恢复。

创建相应的 Gates 的 API, 可以查看 **SYS/BIOS API 说明文档**。

下面的示例代码, 用 Gate 保护临界区。在这个代码中, 使用了 GateHwi, 它禁止与使能中断作为锁定的机制。

```
UInt gateKey;
GateHwi_Handle gateHwi;
GateHwi_Params prms;
Error_Block eb;
Error_init(&eb);
GateHwi_Params_init(&prms);
gateHwi = GateHwi_create(&prms, &eb);
if (gateHwi == NULL) {
    System_abort("Gate create failed");
}
/* Simultaneous operations on a global variable by multiple
 * threads could cause problems, so modifications to the global
 * variable are protected with a Gate. */
gateKey = GateHwi_enter(gateHwi);
myGlobalVar = 7;
GateHwi_leave(gateHwi, gateKey);
```

## 6.1 基于抢占 Gate 实现

下面 Gates 的实现使用一些禁用抢占形式

- **ti.sysbios.gates.GateHwi**
- **ti.sysbios.gates.GateSwi**
- **ti.sysbios.gates.GateTask**

### 6.1.1 GateHwi

GateHwi 禁用和使能中断作为锁定机制。这样的 Gates 保证独家访问 CPU。当临界区域由 Task, Swi 或 Hwi 线程共享时, 可以使用该 Gate。enter 和 leave 之间的持续时间应尽可能短以最小化 Hwi 延迟。

```
UInt key;
GateHwi_Params params;
GateHwi_Params_init(&params);
GateHwi_construct(&myGates, &params);

key = GateHwi_enter(GateHwi_handle(&myGates));
.....
.....
```

```
GateHwi_leave(GateHwi_handle(&myGates),key);
```

### 6.1.2 GateSwi

GateSwi 禁用和使能软中断作为锁定机制。当临界区域由 Task 和 Swi 线程共享时，可以使用该 Gate。enter 和 leave 之间的持续时间应尽可能短以最小化 Swi 延迟。

```
UInt key;
GateSwi_Params params;
GateSwi_Params_init(&params);
GateSwi_construct(&myGates, &params);

key = GateSwi_enter(GateSwi_handle(&myGates));
.....
.....
GateSwi_leave(GateSwi_handle(&myGates),key);
```

### 6.1.3 GateTask

GateTask 禁用和使能 Task 作为锁定机制。当临界区域由 Task 共享时，可以使用该 Gate。enter 和 leave 之间的持续时间应尽可能短以最小化 Task 延迟。

```
UInt key;
GateTask_Params params;
GateTask_Params_init(&params);
GateTask_construct(&myGates, &params);

key = GateTask_enter(GateTask_handle(&myGates));
.....
.....
GateTask_leave(GateTask_handle(&myGates),key);
```

## 例 12 Gate 临界保护实验

本例中，利用 Gate 提供不同级别的临界保护。临界保护是对在不同对象之间共享数据进行独占式的访问。具体的代码工程见 3.17\_GATES\_Create/目录下。工程中部分代码如下：

```
//创建 Gate 对象
void Gate_Create(void)
{
    GateHwi_Params params;
    GateHwi_Params_init(&params);
    GateHwi_construct(&myGates, &params);
```

```
// GateSwi_Params params;  
// GateSwi_Params_init(&params);  
// GateSwi_construct(&myGates, &params);  
//  
// GateTask_Params params;  
// GateTask_Params_init(&params);  
// GateTask_construct(&myGates, &params);  
}
```

对数据进行临界访问

```
gateKey = GateHwi_enter(GateHwi_handle(&myGates));  
DutyValue += 10;  
if(DutyValue > 2500)  
{  
    DutyValue = 0;  
}  
GateHwi_leave(GateHwi_handle(&myGates), gateKey);
```

## 6.2 基于信号量 Gate 实现

下列的 Gates 实现使用了信号量

- **ti.sysbios.gates.GateMutex**
- **ti.sysbios.gates.GateMutexPri**

### 6.2.1 GateMutex

GateMutex 使用一个二值信号量作为锁定机制。每个 GateMutex 实例有它自己唯一的信号量。因为这个 Gate 可能会阻塞，它不能用于 SWI 或 HWI 线程中，只能用于 Task 线程中。

### 例 13 Gate 对共享资源互斥访问

本例中，利用 GateMutex 对共享资源进行互斥访问。它与信号量实现的互斥访问具有相同的作用。实验中创建两个任务，任务 1 每隔 3 秒对共享变量 DutyValue 进行修改，并以事件的形式向另一个任务通知事件。任务 2 则主要是对事件进行请求，并设置最大超时时间。如果事件发生，则进行相应事件处理。同时利用 PWM 顺序点亮 RGB 灯。具体的工程见 **3.18\_GATES\_Mutex/**目录下。

```
//Create the GateMutex object  
void Gate_Create(void)  
{  
    GateMutex_Params params;  
    GateMutex_Params_init(&params);  
    GateMutex_construct(&myGates, &params);  
}
```

```
}  
//the key callback  
static void ButtonHandleFunc(UChar keys)  
{  
    UInt eventId = Event_Id_NONE;  
    if(keys & KEY_BTN1)  
    {  
        eventId |= Event_Id_00;  
    }  
    if(keys & KEY_BTN2)  
    {  
        eventId |= Event_Id_01;  
    }  
    if(eventId)  
    {  
        Event_post(myEventHandle,eventId);  
    }  
}  
  
//任务1 任务函数  
Void heartBeatFxn(UArg arg0, UArg arg1)  
{  
    UInt gateKey;  
    .....  
    .....  
  
    //Create the Event object  
    EVENT_Create();  
  
    //Create the Gate object  
    Gate_Create();  
  
    //Init pwm driver  
    HwPWMInit();  
  
    //创建用户任务  
    UserTaskCreate();  
    while (1)  
    {  
        .....  
        gateKey = GateMutex_enter(GateMutex_handle(&myGates));  
        DutyValue += 1000;  
        GateMutex_leave(GateMutex_handle(&myGates),gateKey);  
    }  
}
```



```
LCD_Print("Duty=%d",&DutyValue);
Event_post(myEventHandle,Event_Id_02);
.....
}
}
//任务2 任务函数
Void TaskKeyFxn(UArg arg0, UArg arg1)
{
    UInt eventId = 0;
    UInt pwmIndex = PWM_R;
    UInt gateKey;
    //指示 RGB 显示方向
    UChar dirFlag = FALSE ;

    while (1)
    {
        //启动 PWM, 即点亮 RGB 亮
        HwPWMStart(pwmIndex);

        //请求事件, 并阻塞在这里
        eventId =
Event_pend(myEventHandle,Event_Id_NONE,Event_Id_00|Event_Id_01,10000)
;

        HwPWMStop(pwmIndex);

        //改变点亮顺序
        if(eventId == Event_Id_00)
        {
            dirFlag = FALSE;
        }
        if(eventId == Event_Id_01)
        {
            dirFlag = TRUE;
        }
        //更改 PWM 占空比
        if(eventId == Event_Id_02)
        {
            gateKey = GateMutex_enter(GateMutex_handle(&myGates));
            HwPWMSetDuty(pwmIndex,DutyValue);
            GateMutex_leave(GateMutex_handle(&myGates),gateKey);
        }

        .....
    }
}
```

```
}
```

### 6.2.2 GateMutexPri

GateMutexPri 是一个互斥 Gate（一次只能被一个线程占有），它实现了优先级继承，防止优先级反转。

## 第七章 时间服务

时间相关操作也是嵌入式操作系统主要的一部分。在 SYS/BIOS 和 XDCtools 中有几个与计时和时间相关的模块。

- **ti.sysbios.knl.Clock 模块**

此模块是利用系统内核 tick 进行时间跟踪。所有的 SYS/BIOS API 会将超时参数解释成系统的 tick 时间。Clock 模块以指定系统 tick 时间间隔调度其函数。默认情况下，Clock 模块使用 hal.Timer 模块获取基于硬件的 tick。当然 Clock 模块也可以配置为应用程序提供的 tick 源。

- **ti.sysbios.hal.Timer 模块**

此模块提供了使用 timer 外设的标准接口。它隐藏了所有 timer 外设底层硬件细节。你可以使用 Timer 模块选择一个 timer，当定时时间到时，让其调用指定函数。

- **ti.sysbios.hal.Seconds 模块**

Second 模块提供了维护当前时间与日期的方法。它起始时间是 1970 年 1 月 1 日 00:00:00 以来的秒数。

### 7.1 Clock

在 SYS/BIOS 中，系统 Clock 管理所有的时间服务。系统的 tick 也是由 Clock 产生。超时和周期是所有 Clock 实例所需要的参数。Clock 模块默认是使用 ti.sysbios.hal.Timer 模块去创建定时器并产生系统 tick，即周期性的调用 Clock\_tick()。Clock 模块也可以通过配置文件配置 Clocks.tick 时钟源不使用硬件定时器。如下：

```
ti.sysbios.knl.Clock.tickSource = Clock.tickSource_USER
or
ti.sysbios.knl.Clock.tickSource = Clock.tickSource_NULL
```

tick 周期决定系统的心跳。可以通过配置文件对 Clock.tickPeriod 参数进行修改。Clock.tickPeriod 时间单位是微秒。

Clock\_tick()和 tick 周期使用如下：

- **tickSource = Clock.tickSource\_TIMER**

这是默认 tick 时钟源设置，即通过硬件定时器产生系统 tick。Clock 模块通过 Clock.tickPeriod 创建定时器。通过改变 Clock.timerId 值，改变 Clock 使用不同的定时器。

- **tickSource = Clock.tickSource\_USER**

如果这样设置，那么应用程序必须在用户自己的中断里调用 `Clock_tick()`。并根据用户中断频率设置 `tickPeriod` 大概值。

- **tickSource = Clock.tickSource\_NULL**

用户将不能调用所有带有超时参数的 SYS/BIOS API 函数，也不能调用 Clock API 函数。但是用户仍然可以使用 Task 模块 API，除了有超时要求 API。比如 `Task_sleep()`。在这个配置方式中，`Clock.tickPeriod` 值是无效的。

`Clock_getTicks()` 返回自启动以来系统 tick 值。系统 tick 值是 32 位，如果达到最大值后，将回到零继续计数。Clock 模块提供 APIs 去启动，暂停和配置 tick。这些 APIs 可以改变 tick 频率。但是以下三个 API 不是可重入，在调用时需要使用 Gates 对其进行保护。

- **Clock\_tickStop()**

暂停定时器去停止 tick

- **Clock\_tickReconfig()**

此函数将在内部调用 `Timer_setPeriodMicroseconds()` 重新配置定时器。如果定时在当前 CPU 频率在不支持 `Clock.tickPeriod` 定时时间将返回错误。

- **Clock\_tickStart()**

通过调用 `Timer_start()`，重新启动定时器产生 tick。

与系统 tick 相关的 API 函数如下：

1. **UInt32 Clock\_getTicks(void)**

返回值 - 当前 tick

2. **Void Clock\_tick(void)**

此函数将 tick 计数值加 1。如果 `tickSource` 设置是 `TickSource_TIMER`，系统将自动调用此函数。而设置成 `TickSource_USER`，需要在应用程序中周期性的调用。

3. **Bool Clock\_tickReconfig();**

此函数将用新的 CPU 频率，去配置用于产生系统 tick 定时器。如果返回 `TRUE`，则配置成功。函数调用只能在 tick 定时器暂停之后进行。即在 `Clock_tickStop()` 与 `Clock_tickStart()` 两函数之间调用。不是所有的定时器都支持 `Reconfiguration`。

利用 `Clock_getTick()` 测试 `Task_sleep` 真正睡眠时间

```
UInt32 time1, time2;
...
System_printf("task going to sleep for 10 ticks... \n");
time1 = Clock_getTicks();
Task_sleep(10);
time2 = Clock_getTicks();
System_printf("...awake! Delta time is: %lu\n", (ULong) (time2 - time1));
```

利用 `Clock_tickReconfig()` 函数重新配置系统 tick

```
BIOS_getCpuFreq(&cpuFreq);
cpuFreq.lo = cpuFreq.lo / 2;
BIOS_setCpuFreq(&cpuFreq);
key = Hwi_disable();
Clock_tickStop();
Clock_tickReconfig();
Clock_tickStart();
Hwi_restore(key);
```

#### 4. Void Clock\_tickStart()

函数将启动系统 tick 定时器。注意，此函数是不可重入，调用性需要进行保护。

#### 5. Void Clock\_tickStop()

函数将暂停系统 tick 定时器。注意，此函数是不可重入，调用性需要进行保护。

上述是 Clock 模块为系统 tick 创建的 Clock 实例。然而 Clock 模块允许开发者创建自己的 Clock 对象实例，并且在超时时间发生时调用超时函数。所有的 Clock 函数在 Swi 上下文中运行。那是因为 Clock 模块自己创建 Swi 对象去使用并且运行 Clock 函数。Clock 使用的 swi 对象优先级是 Clock.swiPriority。开发者可以使用 Clock\_create() 函数动态创建 Clock 对象实例。Clock 实例有一次和连续两种方式。Clock 实例可以在创建好就运行或者创建好之后通过 Clock\_start() 调用运行，决定上述两种行为的是通过配置参数 startFlag。需要注意 Clock\_create() 只能在 Task 上下文或 main 函数中调用。如果一个超时回调函数和一个非零 Timeout 传给 Clock\_create() 时，回调函数将在 Timeout 时间后被调用。Timeout 时间是第一次回调函数被调用时间。对于一次性计时方式，Timeout 是用于计时第一次回调时间，而 period 是零。对于连续模式，除了 Timeout 用于计时第一次回调时间外，period 将计时周期回调时间。Clock 的一次性与连续方式如下图所示。

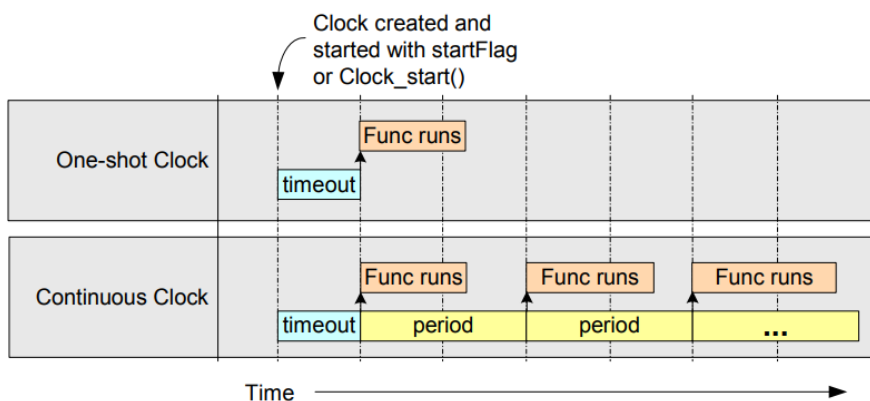


图 11 一次性与连续模式

Clock 实例可以通过调用 Clock\_start() 重新启动，调用 Clock\_stop() 暂停计时。在被暂停之后，可以通过 Clock\_setPeriod(), Clock\_setTimeOut(), Clock\_stop() 等 API 对其进行相应属性修改。

下面将列出 Clock 模块主要的 API 函数及重要的参数结构。

```
typedef struct Clock_Params
{
    // Instance config-params structure

    IInstance_Params *instance; //实例指针
    UArg arg; //传递给函数参数
    UInt32 period; //实例周期
    Bool startFlag; //启动标志
} Clock_Params;
```

Clock 实例超时回调函数指针类型

```
typedef Void (*Clock_FuncPtr) (UArg);
```

**1. Clock\_Handle Clock\_create(Clock\_FuncPtr clockFxn, UInt timeout, const Clock\_Params \*params, Error\_Block \*eb)**

clockFxn – 用户超时回调函数指针

timeout – 第一次超时时间，时间单位系统 tick。毫秒单位转换成系统 tick 可以用下述方法： $ms * 1000 / \text{Clock\_tickPreiod}$ 。

params – Clock 实例配置参数指针，如果为 NULL，则采用默认参数。

eb – 错误处理模块，如果为 NULL，将采用默认方法处理

返回值 – TRUE 表示成功创建

FALSE 表示创建失败

此函数用于创建用户自己的 Clock 实例

**2. Void Clock\_delete(Clock\_Handle \*handleP)**

handle – Clock 实例句柄指针

此函数将删除 Clock 实例对象

**3. Void Clock\_construct(Clock\_Struct \*structP, Clock\_FuncPtr clockFxn, UInt timeout, const Clock\_Params \*params)**

structP – Clock\_Struct 变量指针

clockFxn – 用户超时回调函数指针

timeout – 第一次超时时间，时间单位系统 tick。毫秒单位转换成系统 tick 可以用下述方法： $ms * 1000 / \text{Clock\_tickPreiod}$ 。

params – Clock 实例配置参数指针，如果为 NULL，则采用默认参数。

eb – 错误处理模块，如果为 NULL，将采用默认方法处理

此函数用于创建用户自己的 Clock 实例，此函数与 create 函数区别在于，所占的内存空间有应用程序提供，而 create 函数创建实例内存是从系统堆栈中分配。

**4. Void Clock\_destruct(Clock\_Struct \*structP)**

structP – 是 Clock 实例 struct 指针。

删除 Clock 对象，并释放所占用的资源。

**5. Void Clock\_Params\_init(Clock\_Params \*params)**

params – 配置参数变量指针

将配置参数初始为默认值

**6. Clock\_Handle Clock\_handle(Clock\_Struct \*structP)**

从 Clock\_Struct 结构信息中获得 Clock\_Handle 信息

**7. Clock\_Struct \*Clock\_struct(Clock\_Handle handle)**

从 Clock\_Handle 结构信息中获得 Clock\_Struct 信息

## 例 14 Clock 延时实验

在例七中，按钮的处理没有进行消抖操作。导致在按钮按下时，会产生多次按钮按下事件。在本例中，利用 Clock 的一次性定时特性，对按钮进行延时消抖操作。有效减小按钮的抖动产生的误操作。具体的工程代码见 **3.15\_Clock\_Create/**目录下。

下面代码，演示了 Clock 对象实例创建。并用延时对按钮进行消抖。

```
//按钮 Clock 对象创建
static void Key_TimerCreate(UInt timeout, UInt period)
{
```

```
//将超时数值，转换成系统 tick 个数
UInt timeoutTick = timeout *1000 /Clock_tickPeriod;

Clock_Params params;
Clock_Params_init(&params);

//period => system tick
params.period = period * 1000 / Clock_tickPeriod;
params.startFlag = false; //不立即启动

myClockHandle = Clock_create(Timer_Callback,
                             timeoutTick,&params,NULL);
if(myClockHandle == NULL)
{
    System_printf("Create Clock failed");
    System_flush();
}
}
```

在按钮的中断回调函数中，启动 Clock 对象记时操作。

```
static void Board_keyCallback(PIN_Handle hPin, PIN_Id pinId)
{
    if(myClockHandle != NULL)
    {
        Clock_start(myClockHandle); //启动 clock 记时
    }
}
```

Clock 对象回调函数中，对按钮按下操作进行甄别。

```
static void Timer_Callback(UArg a0)
{
    keysPressed = 0;
    //判断按钮按下
    if ( PIN_getInputValue(Board_BTN1) == 0 )
    {
        keysPressed |= KEY_BTN1;
    }

    if ( PIN_getInputValue(Board_BTN2) == 0 )
    {
        keysPressed |= KEY_BTN2;
    }

    //如果有按钮按下，且回调函数加为空
    if(appKeyChangeHandler != NULL && keysPressed)
    {

```

```
    (*appKeyChangeHandler) (keysPressed) ;  
}  
}
```

## 7.2 Timer

系统的 `ti.sysbios.hal.Timer` 模块为定时器外设提供了标准接口。它的详细描述可以查看硬件抽象层的 `Timer` 说明。用户可以使用 `Timer` 模块创建定时器，并配置超时回调函数，而不需要去关心定时器外设的细节。

根据定时器实例的周期 `period` 参数，定时器有一次性（只产生一次）和连续两种行为。

## 7.3 Second

`SYS/BIOS` 中 `ti.sysbios.hal.Second` 模块提供了一种方式去设置和获取从 1970 年 1 月 1 日 00: 00: 00 以来的秒数。如果硬件系统里有相应的秒发生器（如 `RTC`），则 `Second` 模块维护时间是通过 `RTC` 来完成的；如果没有，`Second` 模块是通过 `SYS/BIOS` 中的 `ti.sysbios.hal.SecondClock` 模块来完成。其主要的 API 函数：

### 1. `Void Second_set(UInt32 seconds)`

`seconds` – 从 1970 年以来，发生的秒数

### 2. `UInt32 Seconds_get(Void)`

返回值 – 获取从 1970 年 1 月 1 日 00: 00: 00 以来的秒数

应用程序调用 `Second_set()` 函数去初始化发生的秒数。如果有需要可以多次调用 `Second_set` 函数，去更新，复位记数。在调用 `Second_get()` 之前函数，应用程序至少调用过一次 `Second_set()` 函数，否则 `Second_get()` 返回的数值没有意义。

在 `Second` 模块中，包含了一个 `time()` 函数。它内部调用了 `Second_get()` 函数。它是重写了 C 标准库中的 `time()` 函数。用户可以将 `time()` 函数与其它 C 标准头文件 `time.h` 中包含的函数合用，去查看当前可读格式的日期时间。

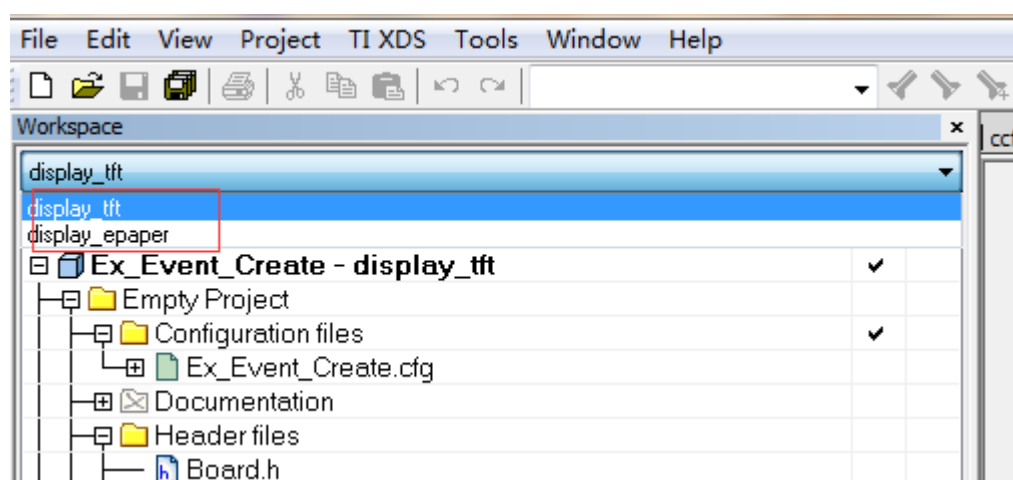
下面的例子代码中使用 `Second` 模块 API 函数，去设置日期，并读取当前日期时间，最后以人类可读方式进行显示。

```
#include <time.h>  
#include <ti/sysbios/hal/Seconds.h>  
UInt32 t;  
time_t t1;  
struct tm *ltm;  
char *curTime;  
/* set to today's date in seconds since Jan 1, 1970 */  
Seconds_set(1412800000); /* Wed, 08 Oct 2014 20:26:40 GMT */  
/* retrieve current time relative to Jan 1, 1970 */  
t = Seconds_get();  
/*  
 * Use overridden time() function to get the current time.  
 * Use standard C RTS library functions with return from time().  
*/
```

```
* Assumes Seconds_set() has been called as above
*/
t1 = time(NULL);
ltm = localtime(&t1);
curTime = asctime(ltm);
System_printf("Time(GMT): %s\n", curTime);
```

## 第八章 RTOS 实验例程说明

RTOS 所有例子工程 Workspace 中都有两个配置分支,分别为 display\_tft、display\_epaper。如下图所示。其中 display\_tft 是支持 128x128 的 tft 真彩显示屏,而 display\_epaper 是支持 200x200 的电子墨水屏。用户只要根据自己身边的显示屏选择相应的分支工程即可。



### 8.1 实验例程汇总

类别	工程目录	相关说明
Task	3.1_Task_Create	创建用户任务实验
	3.2_Task_Del	创建用户任务并且任务实现删除自己,即使自己处于 <b>TERMINNATED</b>
	3.3_Task_Preempt	创建多个不同优先级任务,实现高优先级任务抢占低优先级任务
	3.4_Task_Pri	根据相应事件,动态更改用户任务的优先级
Semaphore	3.5_Semaphore_Test	创建二值信号量,利用信号量控制一个任务的运行
	3.6_Semaphore_Mutex	创建二值信号量,实现对共享资源的互斥访问
	3.7_Semaphore_Count	创建记数值型信号量,控制任



		任务的运行
	3.8_Semaphore_Binary_Pri	创建基于优先级排队的二值信号量。即高优先级任务优先获得信号量
	3.9_Semaphore_Count_Pri	创建基于优先级排队的记数信号量。即高优先级任务优先获得信号量
Event	3.10_Event_Create_AndMode	创建事件对象，并以 And(与)方式请求相应事件，即请求的事件都要发生，才会返回，否则继续阻塞
	3.11_Event_Create_OrMode	创建事件对象，并以 Or(或)方式请求相应事件，即请求的事件只要有一个发生，都会返回，否则继续阻塞
Queue	3.12_Queue_Create	创建一个队列，实现任务间数据通信
Mailbox	3.13_Mailbox_Create	创建邮箱，实现任务间通信
	3.14_Mailbox_Event	创建事件型邮箱，即发布一个邮件的同时也发布一个事件，或请求一个邮件的同时也请求了一个事件
Clock	3.15_Clock_Create	创建一个 Clock 定时器，实现按钮消抖
SWI	3.16_SWI	软件中断
GATES	3.17_GATES_Create	GateHwi, GateSwi 和 GateTask 创建，实现临界访问
	3.18_GATES_Mutex	Gate 互斥访问，实现共享资源保护
HWI	3.19_HWI_Critical	硬中断的临界访问
Memery	3.20_Memery_Create	HeapBuf 和 HeapMem 管理内存，实现 Memery_alloc 和 Memery_free 动态内存操作

## 第九章 APIs 列表

### 9.1 Tasks APIs 函数

1	Void Task_construct(	
---	----------------------	--

	Task_Struct *structP, Task_FuncPtr fxn, const Task_Params *params, Error_Block *eb);	
2	Task_Handle Task_create( Task_FuncPtr fxn, const Task_Params *params, Error_Block *eb);	
3	Void Task_delete(Task_Handle *handleP);	
4	Void Task_destruct(Task_Struct *structP);	
5	UInt Task_disable();	
6	Void Task_exit();	
7	UInt Task_getAffinity(Task_Handle handle);	
8	Ptr Task_getEnv(Task_Handle handle);	
9	Task_FuncPtr Task_getFunc( Task_Handle handle, UArg *arg0, UArg *arg1);	
10	Ptr Task_getHookContext( Task_Handle handle, Int id);	
11	Task_Handle Task_getIdleTask();	
12	Task_Handle Task_getIdleTaskHandle(UInt coreId);	
13	Task_Mode Task_getMode(Task_Handle handle);	
14	Int Task_getPri(Task_Handle handle);	
15	Void Task_Params_init(Task_Params *params);	
16	Void Task_restore(UInt key);	
17	Task_Handle Task_self();	
18	Task_Handle Task_selfMacro();	
19	UInt Task_setAffinity( Task_Handle handle, UInt coreId);	
20	Void Task_setEnv( Task_Handle handle, Ptr env);	
21	Void Task_setHookContext( Task_Handle handle, Int id, Ptr hookContext);	
22	UInt Task_setPri( Task_Handle handle, Int newpri);	
23	Void Task_sleep(UInt32 nticks);	

24	Void Task_stat( Task_Handle handle, Task_Stat *statbuf);	
25	Void Task_yield();	

## 9.2 Semaphore APIs 函数

1	Void Semaphore_construct( Semaphore_Struct *structP, Int count, const Semaphore_Params *params);	
2	Semaphore_Handle Semaphore_create( Int count, const Semaphore_Params *params, Error_Block *eb);	
3	Void Semaphore_delete(Semaphore_Handle *handleP);	
4	Void Semaphore_destruct(Semaphore_Struct *structP);	
5	Int Semaphore_getCount(Semaphore_Handle handle);	
6	Void Semaphore_Params_init( Semaphore_Params *params);	
7	Bool Semaphore_pend( Semaphore_Handle handle, UInt32 timeout);	
8	Void Semaphore_post(Semaphore_Handle handle);	
9	Void Semaphore_registerEvent( Semaphore_Handle handle, Event_Handle event, UInt eventId);	
10	Void Semaphore_reset( Semaphore_Handle handle, Int count);	

## 9.3 Event APIs 函数

1	Void Event_construct( Event_Struct *structP, const Event_Params *params);	
2	Event_Handle Event_create ( const Event_Params *params, Error_Block *eb);	
3	Void Event_delete(Event_Handle *handleP);	

4	VoidEvent_destruct(Event_Struct *structP);	
5	UInt Event_getPostedEvents(Event_Handle handle);	
6	Void Event_Params_init(Event_Params *params);	
7	UInt Event_pend( Event_Handle handle, UInt andMask, UInt orMask, UInt32 timeout);	
8	Void Event_post( Event_Handle handle, UInt eventMask);	

## 9.4 Queue APIs 函数

1	Void Queue_construct( Queue_Struct *structP, const Queue_Params *params);	
2	Queue_Handle Queue_create( const Queue_Params *params, Error_Block *eb);	
3	Void Queue_delete(Queue_Handle *handleP);	
4	Ptr Queue_dequeue(Queue_Handle handle);	
5	Void Queue_destruct(Queue_Struct *structP);	
6	Bool Queue_empty(Queue_Handle handle);	
7	Void Queue_enqueue( Queue_Handle handle, Queue_Elem *elem);	
8	Ptr Queue_get(Queue_Handle handle);	
9	Ptr Queue_getTail(Queue_Handle handle);	
10	Ptr Queue_head(Queue_Handle handle);	
11	Void Queue_insert( Queue_Elem *qelem, Queue_Elem *elem);	
12	Ptr Queue_next(Queue_Elem *qelem);	
13	Void Queue_Params_init(Queue_Params *params);	
14	Ptr Queue_prev(Queue_Elem *qelem);	
15	Void Queue_put( Queue_Handle handle, Queue_Elem *elem);	
16	Void Queue_putHead( Queue_Handle handle, Queue_Elem *elem);	
17	Void Queue_remove(Queue_Elem *qelem);	

## 9.5 Mailbox APIs 函数

1	<pre>VoidMailbox_construct(     Mailbox_Struct *structP,     SizeT msgSize,     UInt numMsgs,     const Mailbox_Params *params,     Error_Block *eb);</pre>	
2	<pre>Mailbox_Handle Mailbox_create(     SizeT msgSize,     UInt numMsgs,     const Mailbox_Params *params,     Error_Block *eb);</pre>	
3	<pre>Void Mailbox_delete(Mailbox_Handle *handleP);</pre>	
4	<pre>Void Mailbox_destruct(Mailbox_Struct *structP);</pre>	
5	<pre>SizeT Mailbox_getMsgSize(Mailbox_Handle handle);</pre>	
6	<pre>Int Mailbox_getNumFreeMsgs(Mailbox_Handle handle);</pre>	
7	<pre>Int Mailbox_getNumPendingMsgs(     Mailbox_Handle handle);</pre>	
8	<pre>Void Mailbox_Params_init(Mailbox_Params *params);</pre>	
9	<pre>Bool Mailbox_pend(     Mailbox_Handle handle,     Ptr msg,     UInt32 timeout);</pre>	
10	<pre>Bool Mailbox_post(     Mailbox_Handle handle,     Ptr msg,     UInt32 timeout);</pre>	

## 声明

本教程中关于 TI-RTOS 原理性知识均参考 TI 官方手册：《Bios\_User\_Guide》。

API 函数说明参考《Bios\_APIs.html》，即在 TI-RTOS 系统安装目录下  
tirtos\_cc13xx\_cc26xx\_xx\_xx\_xx\products\bios\_xx\_xx\_xx\_xx\docs\

## 附录 1：联系方式

公司：无锡谷雨电子有限公司

地址：江苏无锡市滨湖区山水城科技工业园南湖中路 28-11 二号楼 3 楼

网址：<http://www.iotxx.com>

固话：0510-8518-7650

企业 QQ：400-670-7650

客服电话：400-670-7650

