基于TI-RTOS的CC2650DK开发（5）---线程概览

再学下去就必须理解RTOS中的线程了，这是必须要跨过的坎，好在TI有详细的资料：《 SYS/BIOS(TI-RTOS Kernel) v4.64 User's Guide》这本PDF手册里对线程有详细的介绍，我就翻译跟线程相关的章节吧。

3.1 SYS/BIOS 启动顺序

SYS/BIOS启动顺序在逻辑上划分为两个阶段---先于main()函数发生的操作和后于main()函数发生的操作。启动函数中的每一种启动顺序都在不同地方提供了控制指针供用户操作。

“先main()”启动顺序完全由XDCtools运行时包来管理。有关“先main()”启动顺序的更多信息请参考wiki：http://rtsc.eclipse.org/docs-tip/Using_xdc.runtime_Startup。XDCtools运行时启动顺序如下：

1. CPU重启之后，立即执行具体目标/设备的CPU初始设定（从c_int00开始）。请参考《Assembly Language Tools User’s Guide 》中的 "Program Loading and Running"这一章。
2. 在cinit()之前运行“reset functions”表（ xdc.runtime.Reset模块提供这个hook）。Reset.fxns[] 数组中的函数被调用。这些重启函数仅在重启时在运行一个程序之前被平台调用。
3. 运行cinit()对C运行时环境进行初始化。
4. 运行用户提供的第一个函数（xdc.runtime.Startup模块提供此hook）。
5. 运行所有模块初始化函数。
6. 运行用户提供的最后一个函数（xdc.runtime.Startup模块提供此hook）。
7. 运行pinit()。
8. 运行main()。

后main()启动顺序由SYS/BIOS管理，它通过应用程序main()方法最后的BIOS_start()方法显示调用完成初始化。BIOS_start()调用后，SYS/BIOS启动顺序运行如下：

1. Startup Fuctions：运行用户提供的“startup functions”（见BIOS.startupFxns）。如果系统支持计时器，所有静态创建的计时器在此时使用它们的静态配置进行初始化。如果计时器配置为自动启动，它也在此开始计时。
2. Enable Hardware Interrupts（使能硬件中断）。
3. Enable Software Interrupts（使能软件中断）。如果系统支持软件中断（Swis见BIOS.swiEnabled），那么SYS/BIOS启动顺序在此时使能Swis。
4. Task Startup。如果系统支持任务（见BIOS.taskEnabled），那么任务调试始于此处。如果系统无静态或动态创建的任务，那么运行直接进入空循环。

以下摘录的配置脚本在启动顺序的每个可能的控制点上安装了一个用户提供启动方法。配置脚本的文件扩展名为“.cfg”，常用于配置模块和对象。

```
/* get handle to xdc Reset module */
Reset = xdc.useModule('xdc.runtime.Reset');
/* install a "reset function" */
Reset.fxns[Reset.fxns.length++] = '&myReset';
/* get handle to xdc Startup module */
var Startup = xdc.useModule('xdc.runtime.Startup');
/* install a "first function" */
Startup.firstFxns[Startup.firstFxns.length++] = '&myFirst';
/* install a "last function" */
Startup.lastFxns[Startup.lastFxns.length++] = '&myLast';
/* get handle to BIOS module */
var BIOS = xdc.useModule('ti.sysbios.BIOS');
/* install a BIOS startup function */
BIOS.addUserStartupFunction('&myBiosStartup');
```

3.2 线程模块概览

很多实时应用运行时必须同时执行一些看似不相关的方法，它们常常用于响应外部事件，诸如数据可用性或控制信号存在与否。方法的执行和何时执行都非常重要。

这些方法叫做线程，不同的系统对线程的定义要么严密，要么宽泛。在SYS/BIOS中，此术语定义非常宽泛，包含任意由处理器执行的指令流。线程是可由方法调用或中断服务程序（ISR）触发的控制点。

SYS/BIOS将你应用程序中的所有模块化方法组织为一个线程集合。多线程应用程序运行于单个处理器之上时，允许高优先级线程抢占低优先级线程，并在线程间允许各种类型的交互，包括阻塞、通信和同步。

实时应用程序以如下模块化方式进行组织-----集中轮询环，相对于单个运行，它更易于设计、实现和维护。

SYS/BIOS提供了几种拥有不同优先级的线程的支持。每个线程类型都拥有不同的运行方式和优先特性。线程类型包括（优先级从高到低）：

- Hardware interrupts (Hwi)：包含计时器功能
- Software interrupts (Swi)：包含时钟功能
- Tasks （Task）
- Background thread（Idle）

这些线程类型将在下面进行简单描述，更为详细的讨论将在剩余章节进行。

3.2.1 线程类型

SYS/BIOS程序中的四个主要线程类型是：

- Hardware interrupt (Hwi) threads（硬件中断线程）：Hwi线程（也叫中断服务程序-ISRs）在SYS/BIOS应用程序中是拥有最高优先权的线程。Hwi线程常用于执行对时间有严格要求的任务，它们在实时环境中回应外部异步事件（中断）时触发。Hwi线程常常一直运行至结束，但也可能被其它中断所触发的Hwi线程抢占。
- Software interrupt (Swi) threads（软件中断线程）：软件中断线程在Hwi线程和Task线程之间提供额外的优先权等级。和Hwi 的由硬件中断不一样，Swis通过调用某些Swi模块APIs，从而以编程方式来触发。Swis处理程序也受到时间限制，从而避免象tasks一样运行，但它们对时间的要求不如硬中断那样苛刻。和Hwi一样，Swi线程也总是一直运行到结束。Swis使得Hwis可以将某些不太关键的处理延迟为低优先级线程，禁用某些Hwis，从而最小化中断服务程序的CPU消耗。当Tasks为每个线程使用单独的栈时，Swis仅需要足够的空间保存每个中断优先等级关联即可。
- Task threads（任务线程）：Task线程的优先级比后台空闲线程高，比软件中断低。Tasks和软件中断不同之处在于它们在执行期间可以等待（阻塞），直到必要的资源可以使用为止。Tasks需要为每个线程配一个栈。SYS/BIOS提供了许多机制可用于任务间的通信和同步。这些机制包含Semaphores（信号量）、事件、消息队列和 Mailboxes（邮箱）。
- Idle Loop thread（空闲循环线程）：在SYS/BIOS应用程序中，Idle线程运行于最低优先级，它们在一个连续的循环（the Idle Loop）内依次执行。在main方法返回后，一个SYS/BIOS应用程序会调用每个SYS/BIOS模块的启动程序，进而进入Idle Loop。每个线程都必须等到所有其它线程执行完毕后才能再次被调用。Idle Loop会连续不断地运行，除非它被更高优先级的线程抢占。只有那些没有硬时限的方法才会放到Idle Loop内执行。

其它的线程类型，时钟线程运行于Swi线程中，并由重复计时器外部中断所调用的Hwi线程触发。

3.2.2 如何选择使用哪种线程类型

在应用程序中，线程类型和优先级的选择受在预定时间内完成还是正确执行的影响。SYS/BIOS静态配置使得线程类型转换变得非常容易。

一个程序可以使用多种类型的线程。以下规则可决定在程序中所运行的每个线程使用何种类型的对象。

- Swi或Task对比Hwi：在硬件中断服务程序中仅执行重要处理。Hwis应考虑处理那些耗费时间不超过5微秒的硬件中断（IRQs），尤其是那些期限未到达数据就已写入的情况。Swis或Tasks就考虑拥有更长时间期限的事件---约100微秒或更长。Hwi方法应将低优先级处理交由Swis和tasks去处理。使用低优先级线程可将中断处理时间（中断延迟）最小化，从而允许其它硬件中断发生。
- Swi对比Task：如果方法拥有相对简单的相互依存关系和数据共享需求，则使用Swis。如果需求更为复杂，则使用tasks。当高优先级线程可抢占低优先级线程，只有tasks可以等待其它事件，譬如资源可用性。Tasks相比Swis，在使用共享数据时拥有更多选项。当程序提交Swi时，所有Swi方法所需要的输入都应准备就绪。Swi对象的触发结构体提供了一种方法决定资源何时可用。Swis更具内存效率，因为它们全运行于一个单一的栈。
- Idle：创建Idle线程用于在没有其它处理任务时执行非关键管理任务。Idle线程通常没有硬期限，它们往往在处理器空闲时运行。Idle线程按同一优先级顺序执行。你可以使用Idle线程在其它处理未执行的情况下来降低供电系统的需求，这样在电源不足时就无需依赖于管理任务。
- Clock：当希望一个方法运行速率是某个时钟周期操纵的外设中断率的倍数时，可以使用Clock方法。Clock方法可配置为周期性执行或只执行一次。这些方法作为Swi方法运行。
- Clock对比Swi：所有Clock方法的优先级和Swi相同，所以Clock方法无法抢占其它线程。但，Clock方法可为冗长处理提交低优先级Swi线程，这确保了当下个系统周期出现以及Clock Swi再次提交时，Clock Swi可以抢占这些方法。
- Timer：Timer线程运行于Hwi线程背景中。因此，它继承了相应Timer中断的优先级。它们以编程计时器周期的速率调用。Timer线程应当完成那些时间需求最少的任务，如果需要更长的处理时间，可以考虑提交给Swi去做，或提交一个信号量，稍后由task处理，从而更为有效地管理CPU时间。

3.2.3 线程特性对比

表3-1提供了SYS/BIOS所支持的线程类型对比。

3.2.4 线程优先级

在SYS/BIOS中，硬件中断拥有最高优先级。Hwi对象集的优先级并非通过SYS/BIOS隐式维护。Hwi优先级仅适用于在给定CPU周期内多个中断将要触发时进行排序。除非中断被全面禁用或指定中断被单独禁用，否则硬件中断将被其它中断抢占。

Swis的优先级低于Hwis。Swis的优先级可到32（默认为16）。MSP430和C28x的最大优先级序号为16。Swis可被更高优先级的Swi和任意Hwi抢占。

Tasks的优先级低于Swis。task的优先级可到32（默认为16）。MSP430和C28x的最大优先级序号为16。Tasks可被任意更高优先级线程抢占。在等待资源可用及低优先级线程时，Tasks可阻塞。

对于Swis和Tasks来说，更高序号相当于更高优先级。在Swis集和Tasks集中，0是最低优先级。

在所有优先级中，后台空闲循环（Idle Loop）的优先级最低。在CPU不忙于其它线程时运行此循环。当tasks可用，Idle Loop被当成优先级为0的task运行。当tasks被禁用，Idle Loop将在main()方法之后调用。

3.2.5 Yielding 和 Preemption

（Yield为让出控制权的意思，Preemption为抢占的意思）

除非以下情况发生，否则SYS/BIOS线程调度器将运行准备就绪的线程中拥有最高优先级的那一个：

- 正在运行的线程通过Hwi_disable()或Hwi_disableInterrupt()临时关闭某些或所有硬件中断，以防止硬件ISRs运行。
- 正在运行的线程通过Swi_disable()来临时禁用 Swis。这防止了任何更高优先级的Swi抢占当前线程。
- 正在运行的线程通过Task_disable()来临时禁用任务调度器。这防止了任何更高优先级task抢占当前task。但这无法阻止Hwis和Swis抢占当前task。
- 如果低优先级task与高优先级task共享同一门控资源并改变其状态为pending，高优先级task可能将其优先级设置为和低优级task相同。这叫Priority Inversion（优先反转），在4.3.3节描述。

Hwis和Swis都可与SYS/BIOS任务调度器互动。当task被阻塞，通过是由于等待未获得的信号量。信号量可被Hwis和Swis以及其它tasks提交。如果Hwis或Swis提交了信号量使得等待task停止阻塞，当此task比当前运行的task拥有更高优先级时，处理器会切换到此task（必须在Hwi或Swi完成后）。

当同时运行Hwi和Swi时，SYS/BIOS会使用专用系统中断栈，也叫system stack（系统栈，有时也叫ISR栈）。每个task使用自己的私有栈，因此，如果系统中没有task，所有线程共享相同的系统栈。由于性能的原因，将系统栈放入precious fast memory是有利的。请阅读3.5.3节了解系统栈尺寸，3.6.3节了解任务栈尺寸。

表示3-2演示了当一个类型线程正在运行，另一线程准备运行时所发生的状况。显示的是新提交（准备运行）线程的动作。

（表格中*号处解释）：在某些targets中，硬件中断可以被单独可用及禁止，并非所有target可用。同时一些targets拥有控制器支持硬件中断优先级，这种情况下，Hwi仅能被更高优先级Hwi抢占。

注意，表3-2所显示的结果是在线程类型可以被提交的情况下发生的。如果那个线程类型被禁用（例如通过Task_disable），线程将无法在任何情况下运行，直到它的线程类型再次可用。

图3-2演示了Swis和Hwis可用（默认）场景下的执行图。演示了当中断发生，Hwi提交了一个比当前运行Swi优先级更高的Swi时所发生的情况。还演示了第一个ISR正在运行，第二个Hwi发生并抢占第一个ISR的情形。

图3-2中，低优先权Swi被Hwis异步抢占。第一个Hwi提交了一个更高优先级的Swi，它们在两个Hwis完成后才得以执行。

以下是图3-2的简单伪代码描述：

```
backgroundThread()
{
  Swi_post(Swi_B) /* priority = 5 */
}
Hwi_1 ()
{
  ...
}
Hwi_2 ()
{
  Swi_post(Swi_A) /* priority = 7 */
}
```

3.2.6 Hooks（钩子）

Hwi、Swi和Task线程可在线程的生命周期内提供一些点，用于仪器、监视或统计收集等目的。这些代码点被称为“hook”，这些hook所对应的函数称为“钩子函数”。

下列为各种线程类型可设置的hook函数：

Hooks被声明为一组hook函数叫“hook sets”。你无需在一个集合中定义所有hook函数，只需要定义那些应用需要的。

Hook函数只能被静态声明（在配置脚本中），所以它们在调用时非常有效率，不使用时则无运行时开销。

除了寄存器hook外，所有hook函数在调用时都使用对应线程关联对象的句柄做为参数（也就是Hwi对象、Swi对象、Task对象）。其它参数用于具体线程对象的hook函数。

你可为你的应用定义尽可能多的hook集。当定义的hook集超过一个，每个hook集中单独的hook函数通过hook ID序号来调用，hook ID序号用于特定hook类型。如在Task_create()运行期间，为每个被调用的Task hook集所创建的hook的序号，就是Task hook集在最初定义时的序号。

线程的寄存器hook（只被调用一次）的参数是一个索引号（hook ID），它指示在hook集中相关hook函数的调用顺序。

每个hook函数集都有唯一关联的“hook context pointer”。这些普通目的指针可自行保持hook集专用信息，或者如果特别应用需要更多空间，它们会被初始化为指向在一个hook集中创建hook函数所分配内存块。

单个hook函数通过以下指定线程类型API来获取其关联的context pointer：Hwi_getHookContext()，Swi_getHookContext()，和Task_getHookContext()。对应的用于初始化context pointer的API为：Hwi_setHookContext()，Swi_setHookContext()，和Task_setHookContext()。所有这些API都将hook ID做为其参数。

下图显示了一个应用的三个Hwi hook集：

hook context pointers可通过 Hwi_getHookContext()使用三个寄存器hook函数所对应的索引号来进行访问。

在调用ISR函数之前，请使用以下命令调用启动Hook函数：
1. beginHookFunc0();
2. beginHookFunc1();
3. beginHookFunc2();

同样地，返回ISR函数之前，通过以下命令结束Hook函数：
1. endHookFunc0();
2. endHookFunc1();
3. endHookFunc2();

3.3 在SMP系统中使用SYS/BIOS

SYS/BIOS可被用于对称多处理系统（SMP），如现存的几个TI SoC设备，双核ARM Cortes-M3/M4和多核ARM Cortex-A15子系统。这些系统混合了两个或多个相同的处理器核心，共享相同的内存和外设。SMP/BIOS是SYS/BIOS的一种运转模式，它支持这样的系统。

在BIOS、Core、Task、Idle和Hwi模块中配置APIs参数可让你在SMP/BIOS中控制多核运转。详情请参见：http://processors.wiki.ti.com/index.php/SMP/BIOS。