# 第三章：使用WICED实时操作系统（RTOS）与调试器

## 目标

在完成第3章的学习后，您将对WICED RTOS在构建WICED项目中的作用建立基本的认识。您将能使用WICED RTOS抽象层来创建和使用线程、信号标、互斥、队列和定时器。此外，您还将掌握如何配置和运行调试器。

## 时间：1.5小时

## 基础

### RTOS介绍

RTOS的目的是为了降低嵌入式固件的编写复杂性，这种固件拥有资源需求相互重叠的多异步快速时间响应任务。例如，您可能具有一台正在向连接网络读写数据、向外部文件系统读写数据、从多个外设读写数据的设备。既要确保满足响应网络请求的时间要求，同时又要不断为外设提供支持会相当复杂，因而也非常容易出错。通过使用RTOS，您就能将系统功能划分成单独的任务（称为线程），并以相对独立的方式对它们进行开发。

RTOS保有一个线程列表，其中列出了待机、停止或运行状态的线程，以及接下来运行哪个任务（按优先级）和开始运行的时间。我们将RTOS中的这个功能称为调度器。有两种主要的方案可用于管理操作系统中处于活跃状态的线程/任务/进程：先占式与协作式。

在先占式多任务处理中，CPU可完全控制运行中的任务，并能根据要求相应停止和启动任务。在该方案中，调度器可通过CPU保护模式从活动任务中抢夺控制权，停止它们，并移动到下一个任务。先占式多任务处理是用于Windows、Linux等中的方案。

在协作性多任务处理中，每个进程都是好公民，会将控制权交还给RTOS。有一些机制可用于交回控制权，比如rtos\_delay、信号标、互斥和队列（我们将在本文档的后续部分进行讨论）等。WICED RTOS均为协作属性，因而您需要精心处理。

### WICED RTOS抽象层

目前，WICED Studio可支持多种RTOS，其中包括：

* ThreadX，提供方为Express Logic
* FreeRtos，提供方为FreeRTOS

但是，由于可在在WICED芯片上的ROM中构建许多ThreadX，因而一般来说ThreadX是最佳选择。为了简化多个RTOS的使用，WICED SDK具有一个内置的抽象层，可为基础RTOS功能提供统一接口。您能在API指南（Guide）🡪组件（Components）🡪RTOS下查找到关于WICED RTOS API的技术文档。



### 使用RTOS的问题

所有这些听起来都非常好，但无论形容词有多么完美，并非事事都会顺心顺意、完美无缺。使用这些类型的系统会很容易产生三大严重的缺陷，而且这种缺陷很难被发现。这些缺陷都是线程之间在进行交互的过程中产生的副作用。这三大缺陷是：

* 循环依赖性，可能导致死锁；
* 与共享内存和共享外设时发生资源冲突，可导致不稳定的非确定性行为；
* 给执行进程间通信带来困难。

但是并非没有解决的希望。WICED RTOS可为您提供众多解决这些问题的机制，具体包括互斥、信号标、队列和定时器等。通常情况下，所有这些功能都以相同的方式工作。其基本流程如下：

1. 从创建正确类型的数据结构开始（例如*wiced\_mutex\_t*等）；
2. 调用RTOS初始化函数（例如*wiced\_rtos\_init\_mutex()*）。为其提供可指向第一步所创建结构的指针。这是一个供其他函数使用的“句柄”；
3. 通过使用其中的一个访问函数来访问该数据结构（例如*wiced\_rtos\_lock\_mutex()*）；
4. 使用相应的去初始化函数来去除您的数据结构（例如*wiced\_rtos\_deinit\_mutex()*）；。

所有这些函数都需要能够访问该数据结构，因此我通常把这些“共享的”资源称为它们所使用文件中的静态全局变量。

### 线程

正如我们之前讨论过的，线程是RTOS的核心。通过下列变量调用函数*wiced\_rtos\_create\_thread()* 就能非常容易地创建新的线程：

* *wiced\_thread\_t\* thread* – 指向线程句柄数据结构的指针。可将这个句柄用于为其它线程函数识别该线程；
* *uint8\_t priority* – 这是该线程的优先级；
  + 优先级可分为0到31级，0是最高优先级；
  + 如果调度器知道有两个线程都符合运行条件，它就将运行具有较高优先级的线程；
  + WICED Wi-Fi驱动器(WWD)可运行的优先级是3。
* *char \*name* – 线程的名称；
* *wiced\_thread\_function\_t \*thread* – 指向该线程函数的函数指针
* *uint32\_t stack size* – 线程堆栈中应有多少个字节（这里您应该注意的是，堆栈溢出会导致不稳定且难以调试的行为）；
* *void \*arg* – 将传递给该线程的通用变量；
  + 如果您不需要向该线程传递任何变量，则只需使用NULL即可。

该线程函数必须匹配类型*wiced\_thread\_function\_t*。其必须选取类型*wiced\_thread\_arg\_t的单参数，但必须返回一个void。*

线程体看起来像是“main”的无限循环。例如：

void mySpecialThread(wiced\_thread\_arg\_t arg)

{

const int delay=100; //

while(1)

{

processData();

wiced\_rtos\_delay\_milliseconds(delay);

}

}

注意：您应该（大多数情况下）始终在每个线程中放置一定量的*wiced\_rtos\_delay\_milliseconds()或wiced\_rtos\_delay\_microseconds()，*才能给其它线程运行的机会*。*此外，这也适用于主应用*while(1)*循环，因为主应用就是另一个线程。

关于操作线程的函数介绍，请参阅API指南的“组件（Component）🡪RTOS🡪线程（Threads）”部分。



### 信号标

信号标是线程之间的信令机制。名称信号标（源于帆船的信号旗）由Dijkstra在一篇关于同步顺序进程的论文中应用于计算机。在WICED SDK中，可将信号标作为一个简单的无符号整数进行实施。在您“设置”信号标时，它会增加信号标的值。在您“获得”信号标时，它又会相应减少信号标的值。但如果这个值为0，那么线程就会自己**挂起**，直到对信号标进行设置为止。因此，您可以使用信号标在线程之间进行信号传输，提示某事已经完备就绪。例如，您有一个“sendToCloud”线程和一个“collectDataThread”线程。sendToCloud将“获取”信号标，其将让线程挂起**直到**有新的需要发送到云端的数据时且collectDataThread“设置”好信号标后，挂起才会解除。

获取函数需要超时参数。超时参数用于设置函数返回前的等待时间，单位为毫秒。如果您想让线程（几乎）无限等待直至对信号标进行设置，而不是在一个特定延迟之后继续执行，那么就可以使用WICED\_WAIT\_FOREVER。

有关信号标函数的介绍，请参阅“组件（Components）🡪RTOS🡪信号标（Semaphores）”目录下的技术文档。



在启动任何使用信号标的线程之前，您都应始终先对信号标进行初始化。否则，您可能会看到不可预测的行为。

### 互斥

互斥是“相互排斥（Mutual Exclusion）”的缩写。互斥是某个特定资源上的锁——如果您请求在某个已被另一线程锁定的资源上进行互斥，那么您的线程就会进入休眠状态，直到该锁定被解除为止。在本章的练习中，您将为WPRINT\_APP\_INFO函数创建互斥。该函数用于在可变时间量内将字节经由UART流出。如果同时有一个以上的线程使用该函数写入到UART，就会发生故障。您可以通过使用互斥来进行自我保护。

关于互斥函数的介绍，请参阅“组件（Components）🡪RTOS🡪互斥（Mutex）”目录下的技术文档。



您在启动任何使用信号标的线程之前，您都应始终先对信号标进行初始化。否则，您可能会看到不可预测的行为。

### 队列

队列就是向另一个线程发送数据的线程安全机制。该队列是一种FIFO队列 - 您从前面读取，向后面写入。如果您想要读取一个空队列，您的线程会挂起，直至有内容写入到该队列里面。队列里的有效载荷（每个条目的大小）和队列的大小（条目的数量）在创建队列时，用户可以自行配置。

*wiced\_rtos\_push\_to\_queue()*需要一个超时参数*。*这样就可设置当队列满时该函数在返回前的等待时间，单位为毫秒。如果您想让线程（基本上）无限地等待队列中的空间，而不是在特定延迟后继续执行，那么就可以使用WICED\_WAIT\_FOREVER。类似地，*wiced\_rtos\_pop\_from\_queue()*函数需要一个超时参数来指定如果队列为空时线程应等待的时间长度。如果您想让线程（基本上）无限等待队列中的某个值，而不是在特定延迟后继续执行，那么可使用WICED\_WAIT\_FOREVER。

关于队列函数的介绍，请参阅“组件（Components）🡪RTOS🡪队列（Queues）”目录下的技术文档。



在启动任何使用信号标的线程之前，您都应始终先对信号标进行初始化。否则，您可能会看到不可预测的行为。

您队列中允许的最小消息大小为4个字节。指定少于4个字节的消息可能会导致不可预测的行为。使用uint32\_t作为最小尺寸变量是一种非常好的实践做法（因为ARM内核处理器是32位，而且该做法对所有变量都是如此）。

### 定时器

RTOS定时器允许您调度某个函数以指定间隔运行，例如每隔10秒向云端发送一次数据。

当您设置定时器时，您应指定您想要运行的函数以及函数运行的频次。定时器调用的这个函数可选取类型为*void\* arg*的单参数。如果该函数不要求任何变量，那么您就能在定时器初始化函数中指定NULL。

注意：每当定时器失效时，该函数就会执行一次线程，而不是连续不断地执行。因此该函数通常**不会**使用while(1)环路，它只是在每次时钟调用它时才运行，然后退出。

关于定时器函数的介绍，请参阅“组件（Components）🡪RTOS🡪RTOS定时器（Timers）”目录下的技术文档。



### 调试

本部分包含了使用调试器的背景信息。在您进行真正练习的时候，您就可能想要尝试这些步骤，以便能够更轻松地掌握操作方法。

#### 生成目标

为了使用调试器，可更改您的生成目标，这样就能将-debug添加在平台名称的后面（不空格），并从目标末端去除run。也就是说，目标应该是这样：

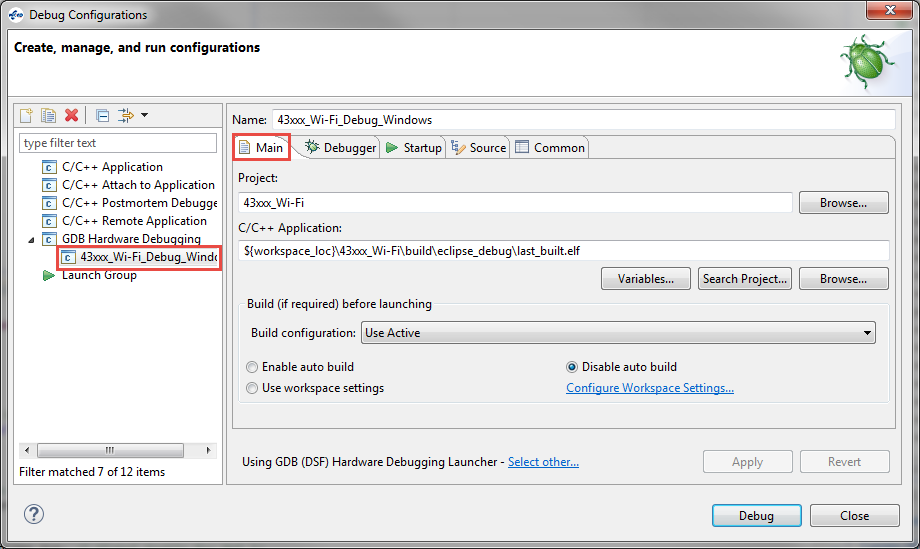
*<folder1>.[<folder2>…].<project>-<platform>-debug download*

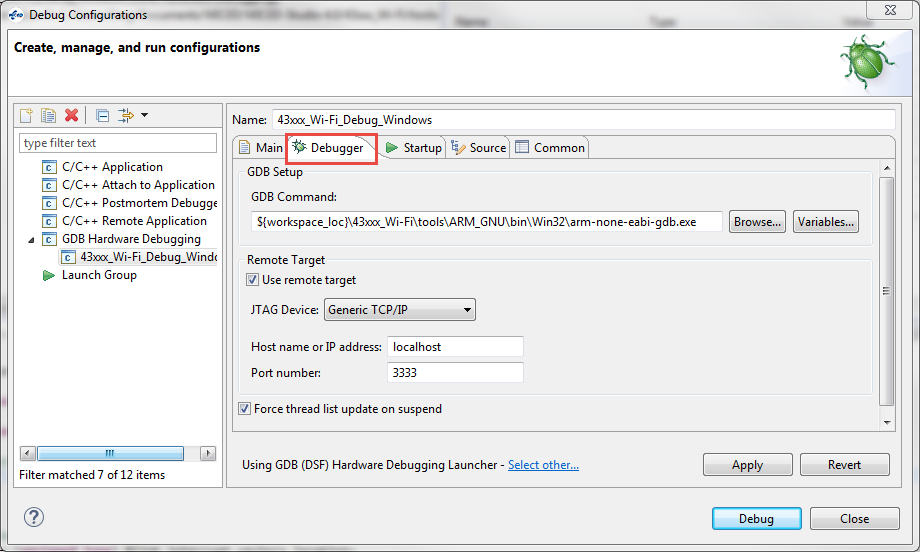
例如，对来自上一章的01\_blinkled项目而言，生成目标应为：

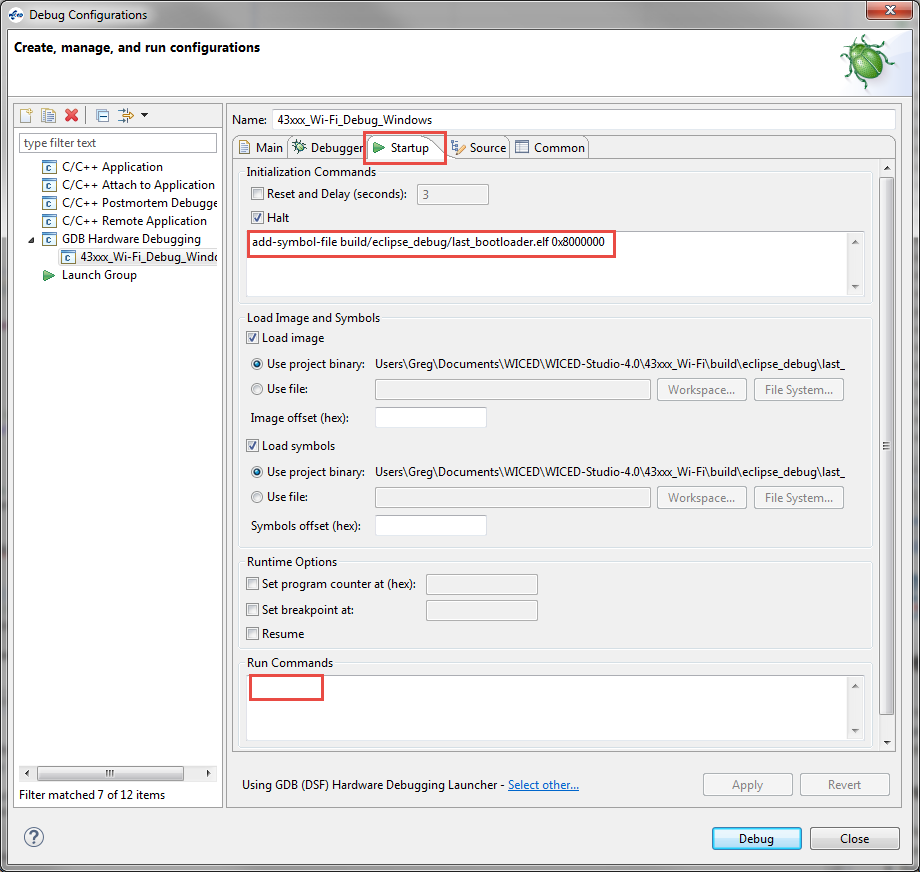
*ww101.02.01\_blinkled-BCM94343W\_AVN-debug download*

#### 设置

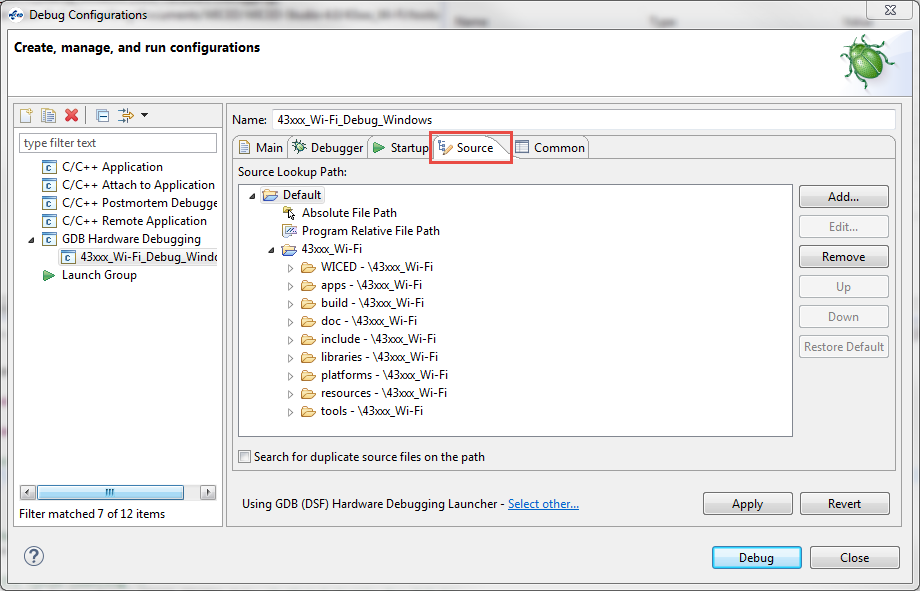
在启动调试器前，我们需要验证它是否对其进行了正确的设置。在WICED Studio中点击绿色虫子图标旁的向下箭头，并选择“调试设置……（Debug Configurations…）”。然后再从左边的窗口中选择“GDB硬件调试（Hardware Debugging）>43xxx\_Wi-Fi Debug\_Windows”。设置下图所示的各种选项卡。您应只需要对“启动（Startup）”和“常见（Common）”选项卡进行更改，为完整起见这里显示了所有的选项卡。

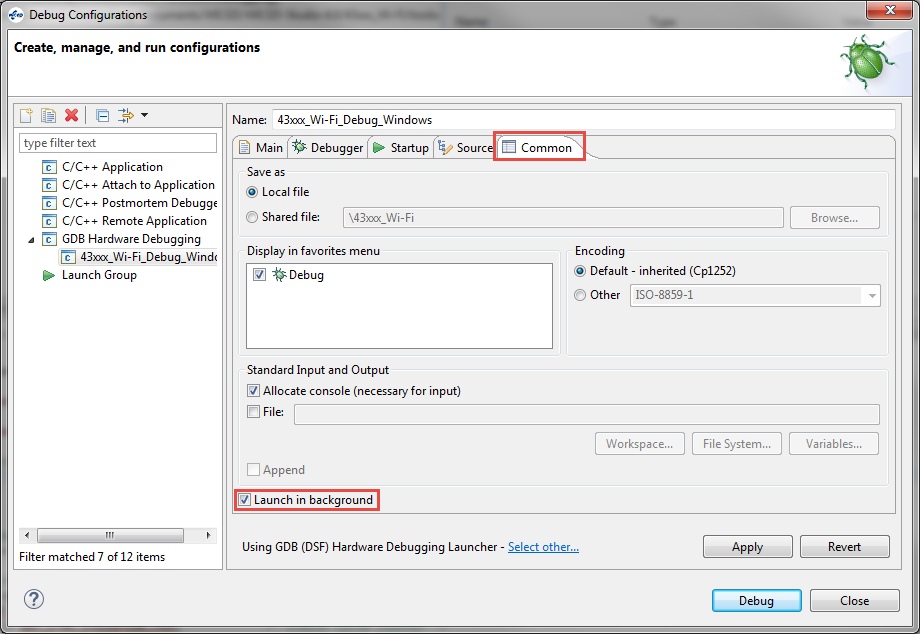






注意：上面方框中的文本是：***add-symbol-file build/eclipse\_debug/last\_bootloader.elf 0x8000000***





#### 运行调试器

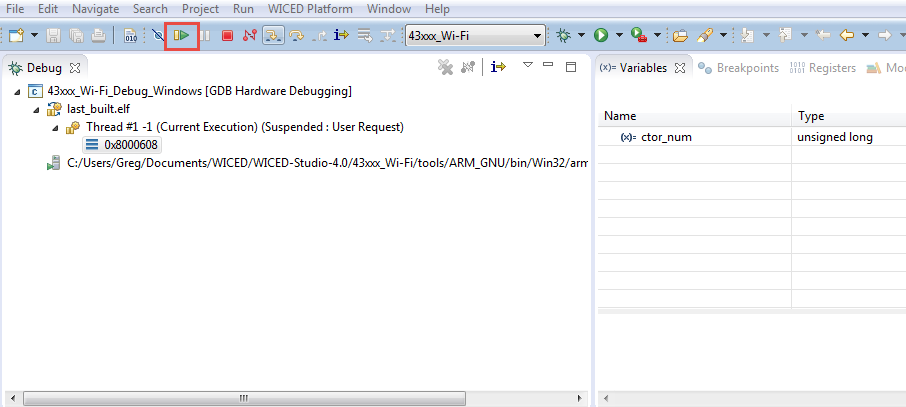
设置完成后，执行生成目标将程序下载到开发板。项目下载完成后，点击绿色虫子图标旁的向下箭头,并选择“43xxx\_Wi-Fi调试窗口（Debug\_Windows）”。如果您看到一个消息询问您是否想打开调试视图，就点击“是的（Yes）”。您可以点击勾选框，告知该工具在将来自动切换视图。



注意：如果您在试图启动调试器时出现报错，您可能需要终止现有的调试进程。启动Windows任务管理器（Windows Task Manager），选择进程(Process)选项卡，点击“映像名称（Image Name）”，按进程名称进行排序并终止当前所有“arm-none-eabi-gdb”进程。



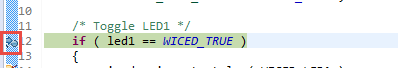
调试器启动后，顶栏将呈现下图状态：



点击“恢复（Resume）”按钮数次（如上图所示）直至程序继续运行，应注意，调试窗口中会出现额外线程以及与之相关的信息。



双击源代码中行号左边的列，您就可以切换断点或者您也可以点击右键并选择“切换断点（Toggle Breakpoint）”来实现。断点符号会显示在行号的左边，如下所示。



一旦断点导致线程挂起，您将看到那一行代码以绿色高亮显示，如上图所示，同时您在调试窗口中也将看到该线程因断点而挂起，如下图所示。



点击红色“终止（Terminate）”按钮停止调试。终止调试器后，您可以点击右上角的按钮切换回C/C++视图。



## 练习

### 01 (线程)创建一个让LED每500毫秒闪烁一次的线程

1. 在ww101文件夹下生成一个新文件夹，称为03。将02/03\_blinkled项目复制到03文件夹中。重命名该项目为01\_thread。更新生成文件，并创建一个生成目标。
2. 设置一个新的线程，让LED每隔500毫秒以开/关形式闪烁一次。
   1. 提示：将代码从03\_blinkled项目的主应用循环中移至线程的循环中。
   2. 提示：主应用循环需要一个例如*wiced\_rtos\_delay\_milliseconds(1)*这样的延迟*。*如果您没有纳入此类延迟，那么LED线程将没有机会运行，因为主应用线程永远不会挂起。
3. 将您的项目编程到开发板。

### 02 (信号标) 创建一个程序，让主线程识别按下按键的行为，然后使用信号标与切换LED的线程通信

1. 将01\_thread复制到02\_semaphore。更新生成文件并创建一个生成目标。
2. 创建一个新的信号标。
3. 在主应用线程中识别按键按下，并在按键按下时设置信号标。
   1. 提示：您可以使用引脚中断来检测按键是否按下并设置信号标。
   2. 提示：确保您在主线程中添加了延迟，使其它线程有机会运行。
4. 在LED线程内使用*wiced\_rtos\_get\_semaphore()*，这样它会永远等待信号标，然后切换LED，而不是让LED持续闪烁。
   1. 提示：如果该线程的名称中含有“blink”字样，您应将其重命名，让它与现在的用途保持一致。

问答题：

在LED线程中，您是否需要*wiced\_rtos\_delay\_millisecon*ds()？给出需要或不需要的原因。

如果您使用100作为型号表超时数值，会发生什么情况？为什么？

### 03 (高级) (互斥)如果两个线程同时向它写入，WPRINT\_APP\_INFO可能会陷入混乱.这时可以使用互斥锁定打印。

1. 将01\_thread复制到03\_mutex。更新生成文件，并创建一个生成目标。
2. 添加第二个线程，让LED2以498毫秒的速率闪烁。
   1. 提示：在一个线程中使用250毫秒的延迟，另一个线程中使用249毫秒的延迟。
3. 用不同消息为每个线程添加一个WPRINT\_APP\_INFO。
4. 将您的项目编程到开发板。
5. 打开终端，查看消息的打印方式。您是否看出了任何问题？
6. 在项目中添加一个互斥，使每个线程都能正确打印。

问答题：

如果您忘记解锁其中一个线程中的互斥，会发生什么情况？为什么？

LED是否仍会闪烁？为什么？

### 04 (高级) (队列)使用队列发送消息，以指示LED闪烁的次数。

1. 将02\_semaphore复制到04\_queue。更新生成文件，并创建一个生成目标。
2. 从项目中移除信号标，并创建一个队列来取代。
3. 给ISR添加一个每次按下按键时都会形成增量的静态变量。将该值推到队列上，使LED线程能够访问该值。
4. 在LED线程中，从队列中发出指令值，确定LED闪烁的次数。
5. 将您的项目编程到开发板。按下按键数次，以观察闪烁次数随每次按下按键而增加的情况。

### 05 (高级)(定时器)使用定时器实现LED闪烁

1. 将01\_thread复制到05\_timer。更新生成文件，并创建一个生成目标。
2. 更新LED线程函数，使它成为一个仅用于切换LED的简单函数，既没有*while(1)*循环，也没有*wiced\_rtos\_delay\_milliseconds()。*
   1. 提示：用于记忆LED状态的变量必须是静态变量，因为每次完成时该函数就会退出，而不会像线程那样无限运行。
3. 移除线程创建函数调用，设置一个每隔250毫秒就会调用LED函数的RTOS定时器来取代创建函数。
4. 将您的项目编程到开发板。

问答题：

在这种情况下，主应用循环能否为空？为什么？

如果对闪烁LED的函数中的*while(1)*循环您不采取移除措施，会发生什么情况？为什么？

运行调试器。这个应用中有多少个线程？

## 相关示例“应用”

|  |  |
| --- | --- |
| **App名称** | **函数** |
| snip.thraed\_monitor | 示范使用系统监测器API来监测应用线程的运行。 |
| snip.stack\_overflow | 示范堆栈溢出状况。 |

## 已知勘误 + 强化 + 点评

您如何知道一个给定线程所需堆栈的大小？