

目录

0 裸机编程引发的思考.....	2
1 RT-Thread 简介	6
1.1 RT-Thread 内核	6
1.2 RT-Thread 与 μ C/OS-II 对比.....	7
1.3 支持平台.....	7
1.4 获取 RT-Thread 最新动态，寻求帮助	8
2 RT-Thread 应用开发	9
2.1 第一个应用 流水灯	9
仿真运行	15
开发板上实际运行	16

RT-Thread 入门指南

写在前面的话：

这篇文档面向的对象，是有一些裸机编程经验，但是对嵌入式操作系统所知很少的开发人员。我希望这篇文档可以引领大家了解实时嵌入式系统，支持国产的 RT-Thread。如果有读者能因为这篇文档收获一些知识，那是我的荣幸，我会很开心。

这篇文档实际上并不能称为 RT-Thread 的入门指南，因为文中对 RT-Thread 的介绍是很少的，至于对 RT-Thread 的内核实现，更是一点没有提及，但是最后还是鼓起勇气起这个名字，我希望这篇文档会不断的发展，完善，内容越来越翔实，可以真正地称为指南。

我会将这篇文档放在 RT-Thread 的论坛上，热切期望大家能把自己在学习中遇到的问题，心得和体会记录下来，反馈上去。众人拾柴火焰高，希望这篇文档不断的丰富下去。

prife goprife@gmail.com

0 裸机编程引发的思考

现在摆在面前你有一个任务：两个 LED 灯，LED1 每隔 1S 闪烁一次，而 LED2 每隔 2S 闪烁一次。仔细想一下，裸机代码中如何实现呢？

我们可以使用一个定时器实现一个时基中断，比如 50 ms 中断一次，那么需要 20 次中断即可以统计 1S，并使用两个全局变量来表示两个 LED 的状态，伪代码如下。

编程模型 1：

```
int main(void)
{
    系统初始化操作
    for (;;)
    {
        //空循环
    }
}

Time_INTR_Hander() //定时器中断函数，这个名字为仅为示意
{
    static int led1_tinc = 0; // 统计第一个定时的定时时间
    static int led2_tinc = 0; // 统计第二个定时的定时时间
    static int led1_status = 0; //标示第一个led的状态
    static int led2_status = 0; //标示第二个led的状态
    //检测是否是定时器中断溢出
    if(led1_tinc++ > 20) // 表示到了s的时间了
    {
        led1_status = !led1_status;
        if(led1_status)
            点亮led1
        else
            熄灭 led1
    }
}
```

```

if(led2_status++ > 40) //表示到达s的时间了
{
    led2_status = !led2_status;
    if(led2_status)
        点亮led2
    else
        熄灭 led2
    //清除定时器中断标志位
}

```

使用裸机编程，我们可以轻易的达到目的。但是这种方式会带来两个问题

- 1) 需要我们来自己管理所有的硬件设备，包括使用系统的定时器。
- 2) led1 和 led2 看起来没有任何的关系，但是我们却要在同一个定时器的中断处理子程序中操作他们。

点亮 LED 这个动作很简单，耗费的 CPU 时间也很短，以至于我们可以把点亮 LED 的代码放在中断中，而 main 函数中只需要一个 for(;;) 死循环即可，main 函数中甚至都不知道有两个 LED 灯在交替闪烁。

点亮两个 LED 灯只是对多任务运行的一种最简单的抽象。

我们来考虑另外一种情况。比如说需要控制两个复杂的外设，比如说 GPRS 模块，和电机模块，其中电机每 1s 中启动一次，控制电机需要花费 0.3s 钟，而 GPRS 每 2s 就要向外部发送一些信息，耗费的时间大约为 0.5s 钟。此时系统中可能还要执行其他的一些控制，比如控制串口 (USRT) 或者同时做 GUI 显示。系统中也可能会有其他的一些中断，而不是单纯是一个定时器中断这么简单。

无疑这种情况是复杂的，至少相对于流水灯而言，我们不能简单的把控制电机的代码和控制 GPRS 的代码简单的放在定时器中断中执行，为什么呢？

因为当系统中中断很多时，中断时间执行过长，将导致比它优先级低的中断被抢占而无法执行。对于不支持中断嵌套的 MCU，采用这种方式，将会导致一个更重要的事件被阻塞。当然，在支持中断嵌套的 MCU 上，这么设计也是可以的，但这不是一个好的设计方案，因为一般来说，同级别的中断是不可以被另外一个同级别的中断抢占的。

从一般意义上来说，中断程序应该尽可能快的执行完毕。

这种编程方式很容易使我们在使系统的各个部分协调工作上耗费大量的时间，而不是专注各个部分的功能。

让我们来考虑第二种写法。把点亮 led 的代码从中断处理子程序中取出放在 main 中，为了实现这种效果，我们需要增加两个变量，用来做事件标志。

编程模型 2

```

static int flag_led1 = 0; //led1事件标志，表示led1的s间隔到
static int flag_led2 = 0; //led2事件标志，表示led2的s间隔到
static int led1_status = 0;
static int led2_status = 0;

int main(void)

```

```

{
    //..完成系统初始化操作
    for(;;)
    {
        if(flag_led1)
        {
            flag_led1 = 0;
            led1_status = !led1_status;
            if(led1_status)
                点亮led1
            else
                熄灭 led1
        }
        if(flag_led2)
        {
            flag_led2 = 0;
            led2_status = !led2_status;
            if(led2_status)
                点亮led2
            else
                熄灭 led2
        }
    }
}

Time_INTR_Hander()//定时器中断函数，这个名字为仅为示意
{
    static int led1_tinc = 0; // 用来统计第一个定时的定时时间
    static int led2_tinc = 0; // 用来统计第二个定时的定时时间
    //检测是否是定时器中断溢出
    if(led1_tinc ++ > 20)// 表示到了s的时间了
    {
        flag_led1 = 1;
    }
    if(led2_tinc ++ > 40)//表示到达s的时间了
    {
        flag_led2 = 1;
    }
    //清楚定时器中断标志位
}

```

看起来这种代码满足上上面所说的原则：

“从一般意义上来说，中断程序应该尽可能快的执行完毕。”

main 函数采用的是轮询方式来处理事件。并且中断中来触发事件标志，这是一种经典的前后台的编程思想。一般来说，这种代码可以工作的很好，并且可以解决我们可以遇到的大部分问题。

不过让我们仔细的分析一下这种方法的问题。

闪烁 led1 和 led2 变成了相同优先级事件了。为什么呢？因为我们采用的轮询方式，没有抢占，这意味着所有的事件都是相同优先级的，main 中依次扫描各个事件标志，当发现某个事件标志为 1，则表示该事件满足，进入事件处理的代码，别忘了，处理之前，先把这个标志清楚为 0，就像

```
flag_led1 = 0;
```

一般来说变成相同优先级没什么问题，并且实践表明，这样处理可以解决很多问题。但是在某些实时要求严格的情况下，这种编程模型不能满足系统的性能。这就是轮询最大的弊端。它将所有的事件混为一潭，不能区分不同事件的轻重级别。因而在某些紧急情况出现时，不能很好的工作。系统的响应时间受限于 for 循环中事件的多寡，以及每个事件函数执行的时间。

此外还有其他的一些编程模型，比如定义一个函数指针数组，然后将各个模块代码各自写成函数，向函数指针数组中填充模块的函数指针，这种编程的方式实际是轮询式编程的一种变形。因为不支持抢占，所以还是轮询的方式，并没有从根本上解决轮询的弊端。

当系统中各个部分复杂起来，各种控制逻辑相互交织的时候，编写一个清晰的裸机程序会耗费一些脑细胞，尤其是当代码量增长到数千行至上万行时，如果没有一个合理并且清晰的程序结构，问题会更糟糕。

让我们来考虑一下桌面系统，比如大名鼎鼎的 windows-XP，这是一个多任务的操作系统，每个应用程序各自工作，它们并不知道其它应用程序处于什么状态，是运行还是关闭，每个应用程序仿佛占用了整个 CPU 一样，由操作系统管理多个应用程序的运行，套用伟大的政治课本上的一句伟大的名言，这极大的解放了我们的生产力。采用 RTOS，多个人可以为多个模块各自编写代码。每个任务占用一个线程，任务在线程中活动，最简单的情况下，它可以不知道其它线程的状态。这种方式，无疑可以大大加快程序的编写，并且减轻程序拼装的难度。RTOS 接管了最令人头痛的环节。

在嵌入式系统中，显然不能运行桌面操作系统，在这种情况下，嵌入式操作系统横空出世。在嵌入式领域，主要使用的嵌入式系统都是实时嵌入式系统，故了描述方便，下面称之为 RTOS。下一节我将隆重介绍国产实时嵌入式系统 RT-Thread。

1 RT-Thread 简介

实时线程操作系统 (RT-Thread) 是国内 RT-Thread 工作室精心打造的稳定的开源实时操作系统 (Real Time Operation System 简称为 RTOS), 历时 4 年, 呕心沥血研发, 力图突破国内没有小型稳定的开源实时操作系统的局面。它不仅仅是一款开源意义的实时操作系统, 也是一款产品级别的实时操作系统, 目前已经被国内十多所企业采用, 被证明是一款能够稳定持续运行的操作系统。

实时线程操作系统 (RT-Thread) 不仅是一个单一的实时操作系统内核, 它也是一个完整的嵌入式系统, 包含了实时嵌入式系统相关的各个组件:

- Finsh Shell
- 文件系统
- 图形界面 RTGUI
- TCP/IP 协议栈

1.1 RT-Thread 内核

RT-Thread 实时操作系统核心是一个高效的硬实时核心, 它具备非常优异的实时性、稳定性、可剪裁性。

当进行最小配置时, 内核体积可以到 3k ROM 占用、1k RAM 占用。

- 内核对象系统

实时线程操作系统内部采用面向对象的方式设计, 内建内核对象管理系统, 能够访问或管理所有内核对象。内核对象包含了内核中绝大部分设施, 而这些内核对象可以是静态分配的静态对象, 也可以是从系统内存堆中分配的动态对象。通过内核对象系统, RT-Thread 可以做到不依赖于具体的内存分配方式, 伸缩性得到 极大的加强。

- 任务/线程调度

支持以线程为基本调度单位的多任务系统。调度算法是基于优先级的全抢占式线程调度, 支持 256 个线程优先级 (亦可配置成 32 个线程优先级), 0 优先级代表最高优先级, 255 优先级留给空闲线程使用; 相同优先级上支持多个线程, 这些相同优先级的线程采用可设置时间片长度的时间片轮转调度; 调度器寻找下一个最高优先级就绪线程的时间是恒定的($O(1)$)。系统不限制线程数量的多少, 只与物理平台的具体内存相关。

- 同步机制

系统支持 semaphore (信号量), mutex (互斥锁) 等线程间同步机制。mutex 采用优先级继承方式以防止优先级翻转。semaphore 释放动作可安全用于中断服务例程中。同步机制支持线程按优先级等待或按先进先出方式获取信号量或互斥锁。

- 通信机制

系统支持 event, mailbox, message queue 通信机制等。event 支持多事件“或触发”及“与触发”, 适合于线程等待多个事件情况。mailbox 中一个 mail 的长度固定为 4 字节, 效率较 messagequeue 高。通信设施中的发送动作可安全用于中断服务例程中。通信机制支持线程按优先级等待或按先进先出方式获取。

- 时钟, 定时器

系统默认使用时钟节拍来完成同优先级任务的时间片轮转调度; 线程对内核对象的时间敏感性是通过系统定时器来实现的; 定时器又分成了硬定时器和软定时器, 一次定时及周期性定时。

- 内存管理

系统支持静态内存池管理及动态内存堆管理。从静态内存池中获取/释放内存块时间恒定，而当内存池空时，可根据申请线程请求把申请线程挂起、立刻返回、或等待一段时间仍未获得返回。当其他线程释内存块到内存池时，将把挂起的线程唤醒。对于系统内存紧张的系统，RT-Thread 也提供了小型的内存管理算法。而对于拥有较大内存的嵌入式系统，RT-Thread 提供了性能非常高效的 SLAB 内存管理系统。

- 诊断

通过系统提供的 FinSH shell 组件，能够查看到线程，信号量，互斥锁，事件，邮箱，消息队列的运行情况，以及各个线程的栈使用情况。

1.2 RT-Thread 与 μ C/OS-II 对比

国内使用的实时嵌入系统，应用比较多的是 μ C/OS-II，那么与 RT-Thread 相比，RT-Thread 表现如何呢？参考下表

	μ C/OS-II	RT-Thread
任务/线程调度	256 个优先级；不允许有相同优先级任务；最大 256 个任务	最大优先级 256/32/8 可配置；允许存在相同优先级线程；线程数不限制；允许动态创建/删除线程
同步互斥机制	semaphore,mutex,mailbox, message queue, event (mailbox 只能存放 1 条消息)	semaphore, mutex, mailbox, message queue,event (mailbox 可存储多条消息)
内存管理	只能使用 OSTimeDly 进行时间间隔处理	挂接到 OS 定时器的硬定时器或软定时器
定时器	只能使用 OSTimeDly 进行时间间隔处理	挂接到 OS 定时器的硬定时器或软定时器
中断嵌套	允许	允许

从上面这个表格可以看出，RT-Thread 实际支持的特性比 μ C/OS-II 要好。

使用 μ C/OS-II 做商业开发需要购买其昂贵的商业许可证，并且这只是一个内核的，还不包括嵌入式系统常用的组件，如 GUI，TCP/IP 协议栈，文件系统等等，这些组件需要另外付费。而 RT-Thread 采用 GPL-V2 发布，并且承诺永久不会针对使用 RT-Thread 收费，用户只需要保留 RT-Thread 的 Logo 既可以免费使用。

1.3 支持平台

目前 RT-Thread 支持的平台已经很多了，目前已支持的平台有(仅列出部分)

核心	平台	编译环境
ARMCortex-M3	STM32F10x	MDK IAR GCC
	STM32F207	MDK IAR GCC
ARM920T	S3C2440	MDK GCC

ARM7TDMI	ATMEL AT91SAM7S	MDK GCC
	LPC2148	MDK GCC
X86	I386	GCC

对更多平台的支持在快速的发展中。对其他平台的信息可以从 Rt-Thread 的官方网站上获取。

这里强调一点，RT-Thread 对主流的 STM32 系列 MCU 支持已经非常完善，官方发布的源码包中就已经包含 STM32 的移植实现。因此 STM32 上使用 RT-Thread 是非常简单的，具体使用将在后面的章节详述。

1.4 获取 RT-Thread 最新动态，寻求帮助

官方网站: <http://www.rt-thread.org/>

此网站中有 wiki 和论坛。wiki 可以查阅资料，论坛可以发贴求助。

获取 RT-Thread 的最新代码:

google 代码管理网址: <http://code.google.com/p/rt-thread/>

最新代码可以从这里下载。另外这里也发布有 RT-Thread 开源产品。

如果你想参与 RT-Thread 的开发，或者在遇到问题时，想要向开发人员请教，又或者是发现了 RT-Thread 中的 BUG，想要提交 BUG 或修复补丁，那么订阅邮件列表就很有必要了。

邮件列表: rt-thread-cnusers@googlegroups.com

最好使用 gmail 的邮箱，给上面这个邮箱发两封邮件，就可以订阅邮件列表。使用邮件列表是最快捷，最方便的方法，这里强烈推荐。

2 RT-Thread 应用开发

下面的章节将以 STM32 平台为例简要介绍如何使用 RT-Thread

更详细的应用开发,请参考 RT-Thread 官方推出的《RT-Thread 实时操作系统编程指南》

首先根据上面提供的地址中下载 RT-Thread 的源码包。解压之后,目录结构如下:

```
RT-Thread-0.3.2
├─bsp          支持的各个平台的工程目录, STM32 目录即在此目录下
├─examples     存放一些测试的例子文件
├─filesystem   文件系统源码目录, 含有文件系统组件代码
├─finsh        FinSH SHELL 源码目录
├─include      头文件目录
├─libc         小型 C 库源码
├─libcpu       存放各个平台核心的引导程序源码, 如 ARM7DMI, ARM920TDMI 等
├─net          网络支持源码目录
├─rtgui        GUI 组件源码目录
└─src          RT-Thread 核心源码目录
```

在 BSP (Board Support Package 即所谓的板级支持包) 目录下, 就有我们将要用到 STM32 目录。

2.1 第一个应用 流水灯

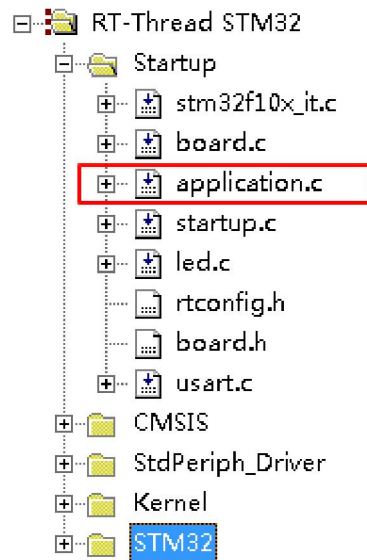
下面我们使用 RT-Thread 实现两个 LED 灯的闪烁效果。

前面已经说明, RT-Thread 对 STM32 的支持非常完善。

在 RT-Thread-0.3.2\bsp\stm3210 目录下已经有了一些工程实例。如下

```
├─Libraries
├─project_107
├─project_filesystem
├─project_finsh
├─project_full
├─project_led
├─project_led_simple
├─└─obj
├─project_lwip
└─project_valueline
```

进入 project_led_simple 目录, 此目录中 project.uv2 文件, 这就是 MDK 的工程文件, 双击它, 就会打开 MDK, 如果你使用的是 IAR, 那么可以双击 project.eww。本文档采用 MDK 编译。

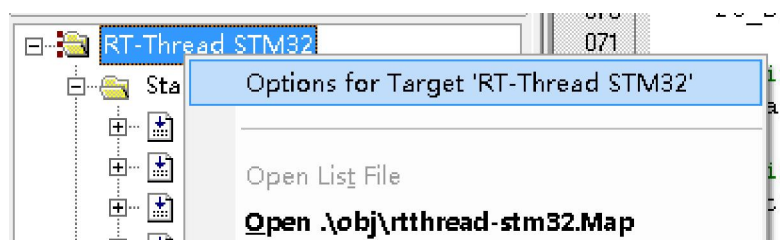


工程结构介绍：

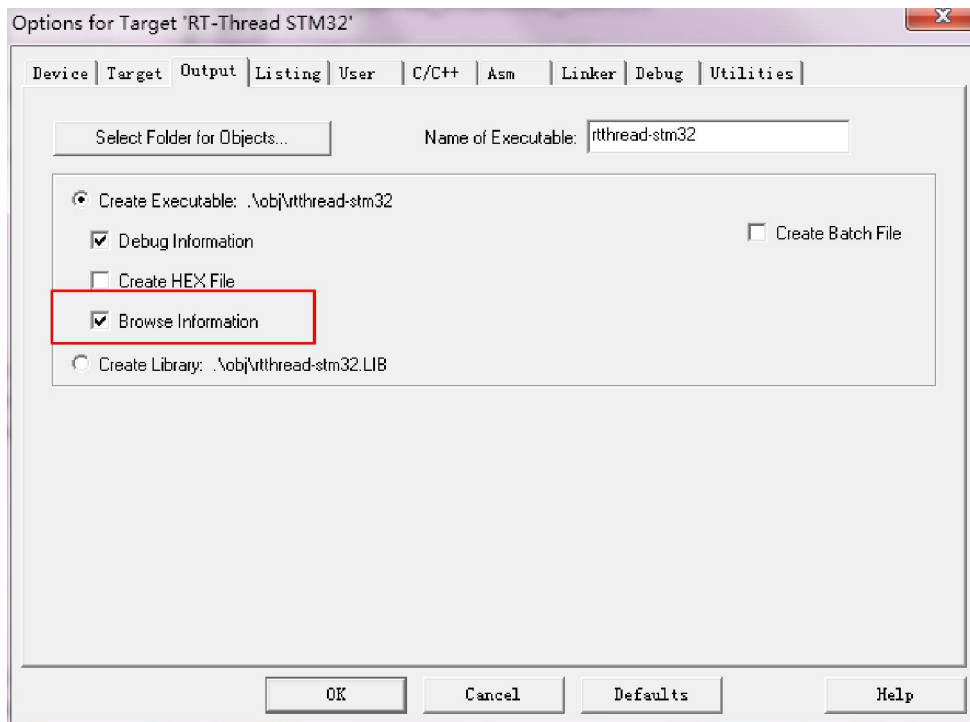
Startup, 存放的使用的 stm32 开发板的相关代码, 包括 STM32 系统时钟配置, NVIC 向量表配置, 涉及的硬件模块(这里就是 LED 占用的 IO 口)的初始化代码, 以及 RT-Thread 的应用程序代码 (application 文件)。Startup 文件, 是我们目前主要关注的地方。

CMSIS	STM32 官方的库文件
StdPeriph_Driver	STM32 官方提供的片上外设的库文件
Kernel	RT-Thread 内核的代码
STM32	STM32 官方代码, 主要启动代码和一些硬件中断的处理代码

为了方便阅读代码, 首先做如下配置。按照下图打开 Option 选项, 下文中再叙述 MDK 的 Option, 都按照如下方式打开。



勾选红色部分 (即浏览信息, 勾选这个之后, 编译工程, MDK 就会生成浏览信息, 我们就可以方便的在 MDK 中查看函数的定义, 并跳转到定义处)

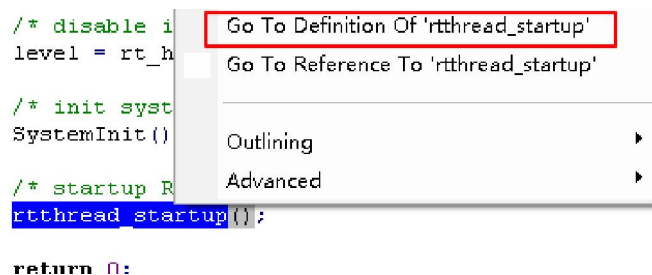


然后编译 MDK 工程。编译完毕，没有任何的错误。

阅读代码，让我们看看这个工程是如何操作 LED 灯的。先明确 LED 灯所在代码找到程序的入口 main 函数，位于 startup.c 中

```
int main(void)
{
    rt_uint32_t UNUSED level;
    /* disable interrupt first */
    level = rt_hw_interrupt_disable();
    /* init system setting */
    SystemInit(); //配置系统时钟
    /* startup RT-Thread RTOS */
    rtthread_startup();
    return 0;
}
```

在 `rtthread_startup()`； 在这个函数上右击，选择 Go To Definition，就会跳转到函数定义出。



我们发现此时代码就已经跳转到定义处。MDK 的代码浏览就提这么多，后面不再赘述。

```

/**
 * This function will startup RT-Thread RTOS.
 */
void rtthread_startup(void)
{
    /* init board */
    rt_hw_board_init(); // 主要实现初始化NVIC，串口初始化也在这里
    /* show version */
    rt_show_version(); // 向串口打印rtt logo和版本号
    /* init tick */
    rt_system_tick_init(); //启动stm32的 sys tick中断
    /* init kernel object */
    rt_system_object_init(); //RTT内核 对象初始化
    /* init timer system */
    rt_system_timer_init(); //RTT内核 定时器初始化
    /* init scheduler system */
    rt_system_scheduler_init(); //RTT内核 调度器初始化
    /* init application */
    rt_application_init(); //应用程序初始化,应用代码会放在这个函数中
    /* init idle thread */
    rt_thread_idle_init(); //RTT内核 idle线程初始化
    /* start scheduler */
    rt_system_scheduler_start(); //RTT内核 启动调度器，开始线程调度
    /* never reach here */
    return ;
}

```

大体浏览各个函数具体实现，即可了解流程，上面红色部分是我们在做应用程序编码时需要改动的函数入口。

`rt_application_init();` // 应用程序线程初始化 我们的线程代码会放在这个函数中

实际上应用中，使用 RT-Thread 编写应用程序线程代码，主要集中在 `board.c` 和 `application.c` 中。

`board.c` 中会编写模块的初始化函数，并将它放在 `rt_hw_board_init()` 内调用。

`application.c` 更多的是存放线程运行时的代码，不过对一些简单的初始化操作也可以放在线程函数中，不建议采用这种方式（这个 led 示例工程是这种方式！）。

对于本例，`led.c` 提供了 led 占用 GPIO 口的初始化操作，包括配置 IO 口为推挽输出，使能 GPIO 口的总线时钟。并且定义了 led 点亮和关闭的两个函数。读者一看便知。

重点来看 `application.c`

```

#include <rtthread.h>
#include "led.h"

char thread_led1_stack[512];
struct rt_thread thread_led1;

```

```

static void rt_thread_entry_led1(void* parameter)
{
    /* init led configuration */
    rt_hw_led_init();

    while (1)
    {
        /* led on */
        rt_kprintf("led1 on\r\n");
        rt_hw_led_on(0);
        rt_thread_delay(50); /* sleep 0.5 second and switch to other
thread */

        /* led off */
        rt_kprintf("led1 off\r\n");
        rt_hw_led_off(0);
        rt_thread_delay(50);
    }
}

char thread_led2_stack[512];
struct rt_thread thread_led2;
void rt_thread_entry_led2(void* parameter)
{
    unsigned int count=0;
    while (1)
    {
        /* led on */
        rt_kprintf("led2 on,count : %d\r\n",count);
        count++;
        rt_hw_led_on(1);
        rt_thread_delay(RT_TICK_PER_SECOND);

        /* led off */
        rt_kprintf("led2 off\r\n");
        rt_hw_led_off(1);
        rt_thread_delay(RT_TICK_PER_SECOND);
    }
}

int rt_application_init()
{
    /* init led1 thread */
    rt_thread_init(&thread_led1,

```

```

        "led1",
        rt_thread_entry_led1,
        RT_NULL,
        &thread_led1_stack[0],
        sizeof(thread_led1_stack),10,10);
rt_thread_startup(&thread_led1);

/* init led2 thread */
rt_thread_init(&thread_led2,
        "led2",
        rt_thread_entry_led2,
        RT_NULL,
        &thread_led2_stack[0],
        sizeof(thread_led2_stack),10,10);
rt_thread_startup(&thread_led2);

return 0;
}

```

上面的代码是非常容易理解的。这里，在 `rt_application_init` 函数中创建了两个线程。线程创建函数为 `rt_thread_init`。关于这个函数的使用，请参考《编程手册》，创建线程之后，`rt_thread_startup(&thread_led1);` 就会启动调度器启动这个线程。`led2` 的线程同理。当我们编写其他的模块代码时，可以对照处理。

第一个线程，使 `led1` 以 0.5s 闪烁一次，第二个线程中 `led2` 以 1s 闪烁一次，这个跟我们的所要实现的效果基本一致。

其中 `led1` 的线程函数如下

```

char thread_led1_stack[512];
struct rt_thread thread_led1;
static void rt_thread_entry_led1(void* parameter)
{
    /* init led configuration */
    rt_hw_led_init();
    while (1)
    {
        /* led on */
        rt_kprintf("led1 on\r\n");
        rt_hw_led_on(0);
        rt_thread_delay(50); /* sleep 0.5 second and switch to other
thread */

        /* led off */
        rt_kprintf("led1 off\r\n");
        rt_hw_led_off(0);
        rt_thread_delay(50);
    }
}

```

```
}
```

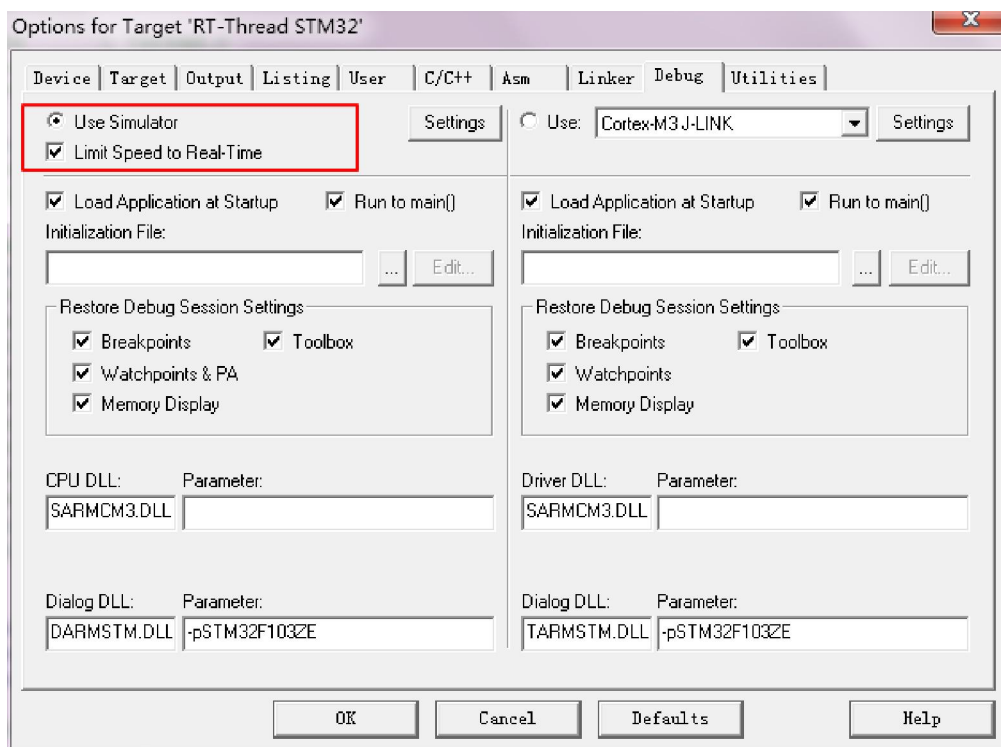
可以看出,这个 led1 的代码跟在裸机中编程几乎是一样的,这个代码中不涉及对 led2 的操作。同样,led2 的线程函数中也只是 led2 的操作,这避免了裸机编程模型 1 中,led1 和 led2 相互交织的情况。显然,这两段代码可以由两个人分别编写。


需要说明一下 `rt_kprintf("led1 on\r\n");` 函数是 RTT 提供的 API, 可以实现向控制台输出功能, 当前会向串口输出, 跟 `printf` 使用是一致的, 也支持打印变量值。

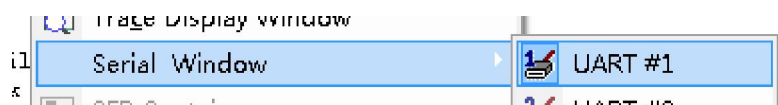
仿真运行

MDK 提供了强大的软件仿真功能, 可以仿真外设, 串口, 定时器, PWM, 等等, 现在我们用 MDK 提供的仿真功能来运行一下程序。因为上面的代码中有串口输出的语句。所以可以通过 MDK 提供的串口仿真, 看到运行的效果。

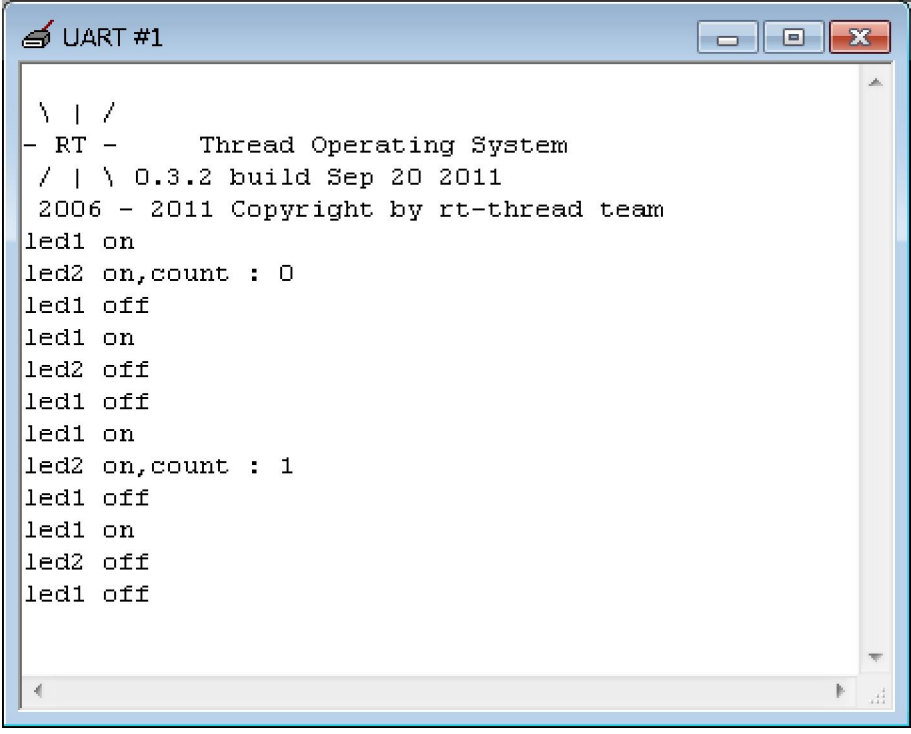
MDK 配置如下



然后点击工具栏的  按钮, 开始仿真运行。此时, 可以看到程序运行光标到达 main 函数入口处, 现在打开串口仿真界面。在菜单栏 view -> serial Windows 下, 打开 UART#1, 如下图所示



然后点击运行, 开始仿真运行, 串口上输出的信息如下。这里说明一点, 仿真模式下, 时间基准是不准确的, 程序中是 led1 0.5S 闪烁一次, led2 1S 闪烁一次, 仿真上运行比实际运行稍慢一些。



```
\ | /
- RT -      Thread Operating System
/ | \ 0.3.2 build Sep 20 2011
2006 - 2011 Copyright by rt-thread team

led1 on
led2 on,count : 0
led1 off
led1 on
led2 off
led1 off
led1 on
led2 on,count : 1
led1 off
led1 on
led2 off
led1 off
```

开发板上实际运行

要想在开发板上实际运行，修改 `board.c` 和 `led.c` 文件中相关代码。根据自己的开发板上的实际接线情况做修改。编译，烧录，运行即可。不再赘述。

更多内容，请参考

更详细的应用开发，请参考 RT-Thread 官方推出的《RT-Thread 实时操作系统编程指南》

推荐阅读书目（待添加）

《C 和指针》

~~《操作系统设计与实现》~~

~~《Linux 内核设计与实现》~~

STM32 相关书籍

《ARM Cortex-M3 权威指南》有中文 PDF (word 生成) 版本，强烈推荐阅读，泛读即可。

STM32 编程相对来说比较简单。STM32 官方提供了完整的库函数，涵盖所有片上外设的操作函数。编写 STM32 程序，推荐的必备参考资料有

1) STM32 参考手册，中文版本

STM32_RM_CH_V10_1.pdf

Stm32-Reference manual-R008-v12-en

注：STM32 官方网站有最新的翻译修订版本，可以去下载最新的版本参考

2) STM32 某系列 MCU datasheet。

因为当需要查阅某个引脚的功能时，需要用到 datasheet

以 103 为例，103 系列的 datasheet 为

stm32f103_en.pdf，建议看英文版本。

3) STM32 应用笔记

STM32 官方提供了丰富的应用笔记，涵盖片上大部分外设的使用，并且配有大量的官方示例代码。善加利用可以大大加快开发进度。

4) Google!

需要掌握的东西

- C 程序运行模型，什么是堆，什么是栈，MDK 中如何配置堆栈大小
- static, volatile 用法
- 1) 全局变量，全局静态变量，局部静态变量
- 2) volatile 用法详解
- 函数指针用法
- 条件编译语句使用
- malloc, free 函数使用
- 多行宏定义
- 内联函数使用
- 嵌入式汇编/内联汇编
- sizeof 用法
- 链表（暂时不要求）

MDK 使用技巧（待整理添加）

重定向 printf，例如，用 printf 向串口打印信息。

更多技巧