RT-Thread 环境快速搭建入门教程

-基于正点原子STM32 开发平台

本教程和配套源码由正点原子团队和 RT-Thread 团队联合编写，为大家学习 RT-Thread 搭建一个基于 STM32 的工程模板，开启大家学习 RT-Thread 大门。正点原子团队后续将联合 RT-Thread 团队推出详细的 RT-Thread 测试源码和学习教程，敬请耐心等待。

该文档由于时间仓促，难免有不足的地方，如有发先错误，请在 www.openedv.com 开源电子网 RT-Thread 实时系统板块发帖指出，我们会及时修正。谢谢支持。

本教程配套源码请到开源电子网下载，连接如下：
http://www.openedv.com/thread-230240-1-1.html

何为 RT-Thread Nano？大家知道，Keil5 以后采用 pack 形式管理芯片及各种相关组件的。RT-Thread Nano 就是通过 Keil pack 方式发布，在保持原有 RT-Thread 基本功能的情况下，实现了极小的 Flash 和 Ram 占用。默认配置下，Flash 可小至 2.5K， Ram 可以小至 1K。

目前 pack 包含有 kernel、shell（msh）、device drivers 三部分功能，这 3 个功能可按实际使用情况按需加载。

一、RT-Thread Nano Pack 下载安装

1. 打开正点原子任何一款开发板的跑马灯实验库函数版本（F1 和 F407 为标准库，F429 和 F767 为 HAL 库）。
2. 在 MDK5 主界面上点击“Pack Install”按钮，即可进入 Pack Install 界面

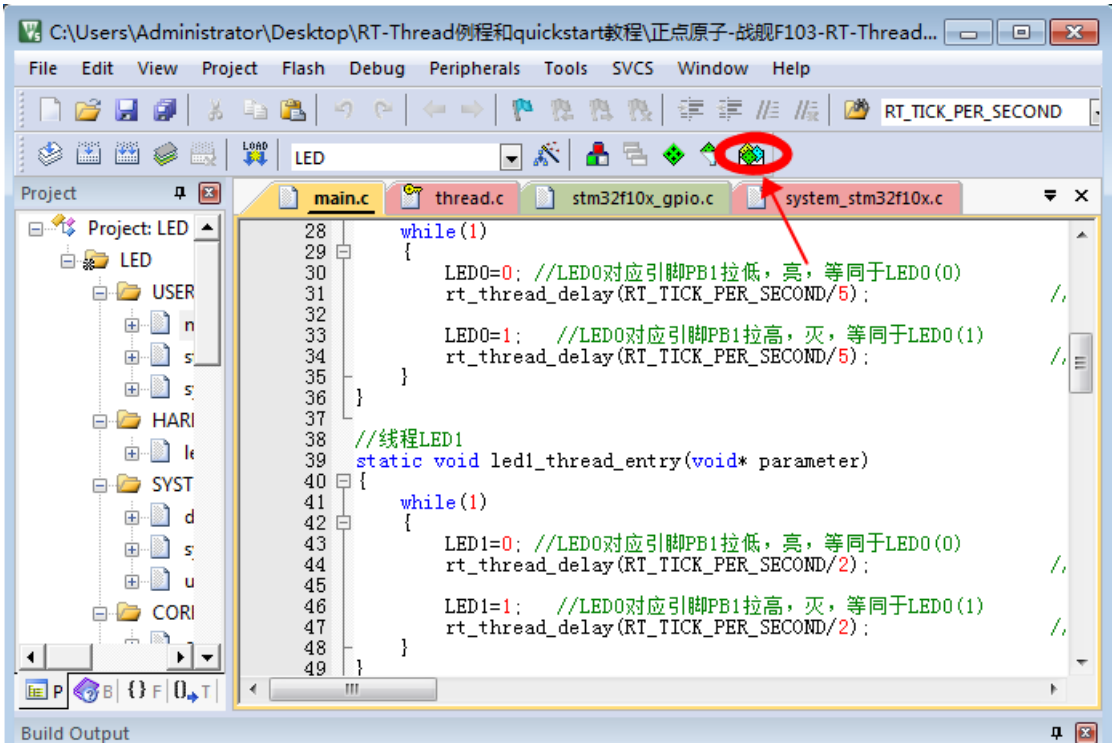

图 1：MDK5 主界面

3. 在 Pack Install 界面下，RT-Thread Pack 在右边栏中。如未下载，可点击“Install”下载；如已安装，版本有更新，将提示“Update”可更新。

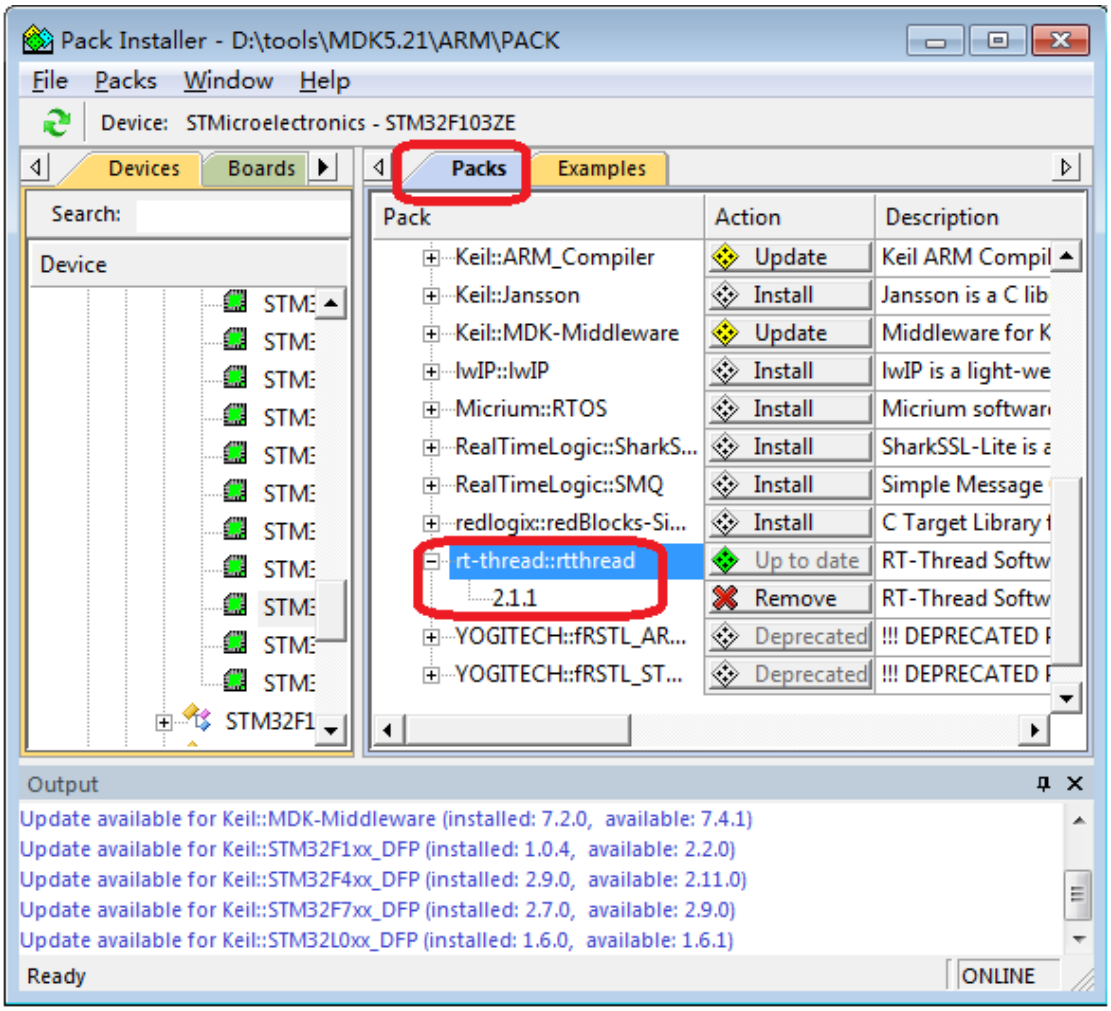

图 2：RT-ThreadPack 下载

4. 如果在图 2 界面“Packs”栏中未发现“RT-Thread”，我们可以通过两种方法获取 RT-Thread Pack。第一种方法是直接从链接 http://www.rt-thread.org/download/mdk/rt-thread.rtthread.2.1.1.pack 下载 2.1.1 版本的的 RT-Thread Pack，然后双击完成安装。第二种方法是在菜单“Packs”下点击“Check for Updates”，Update 需要一定的时间。Update 完成后，将可看到 RT-Thread Pack。Pack 下载完成后 Keil 将自动弹出 Pack 安装界面，按步骤依次完成安装。

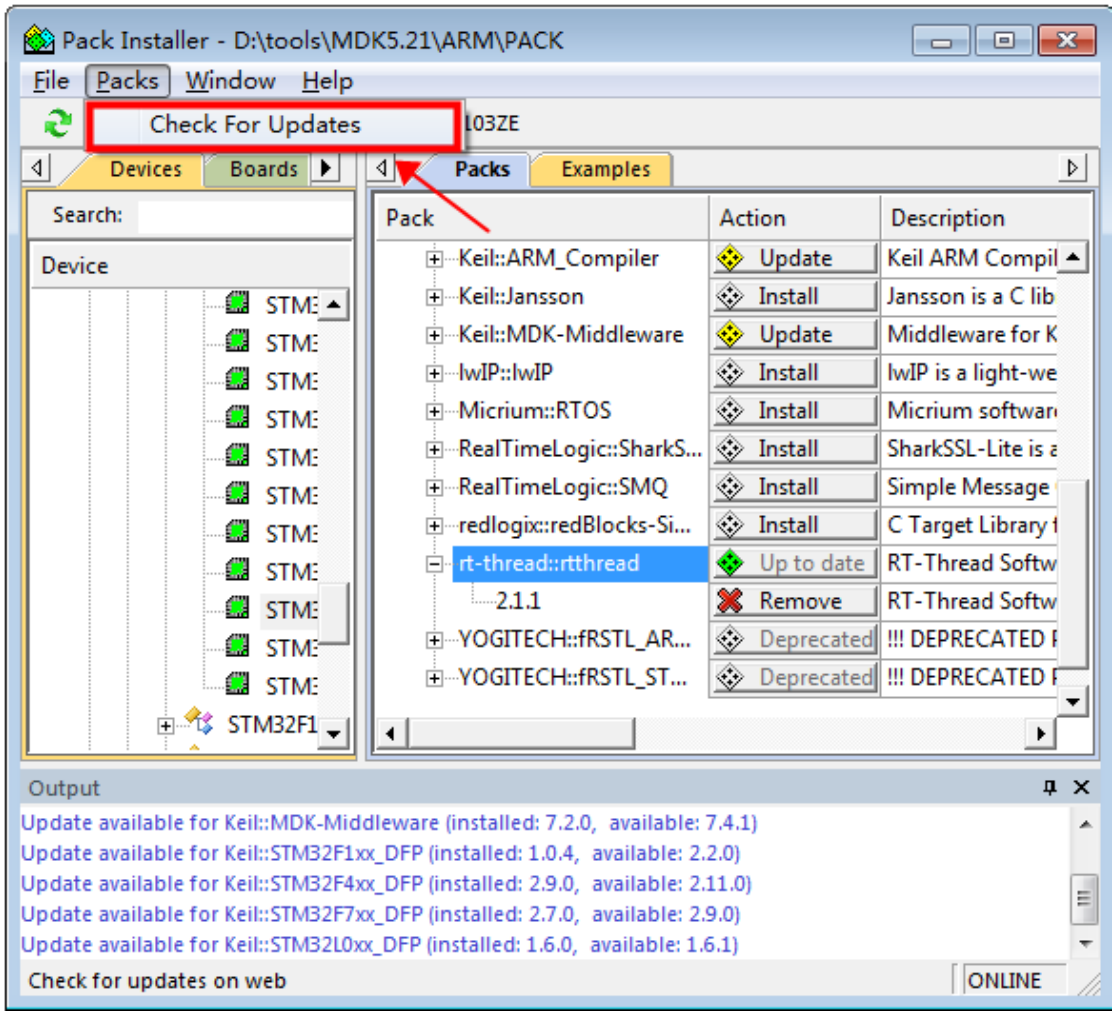

图 3：Pack Update

二、kernel 加载与应用

1. 加载 RT-Thread Kernel:在主界面点击“ManageRun-Time Environment”按钮即可进入加载页。

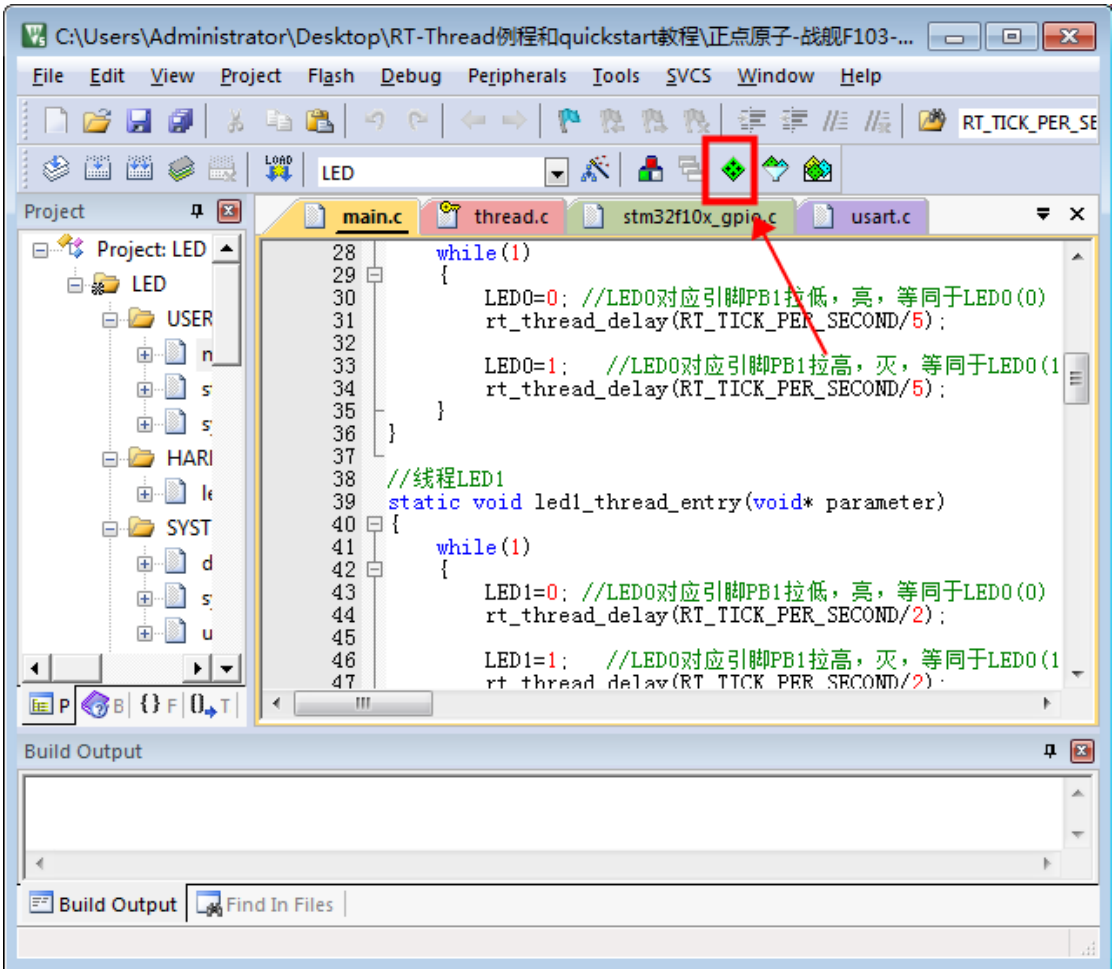

图 4：ManageRun-Time Environment

在“RTOS”一栏中选中“RT-Thread”，并在列表中选中“kernel”，当前版本为 2.1.2。

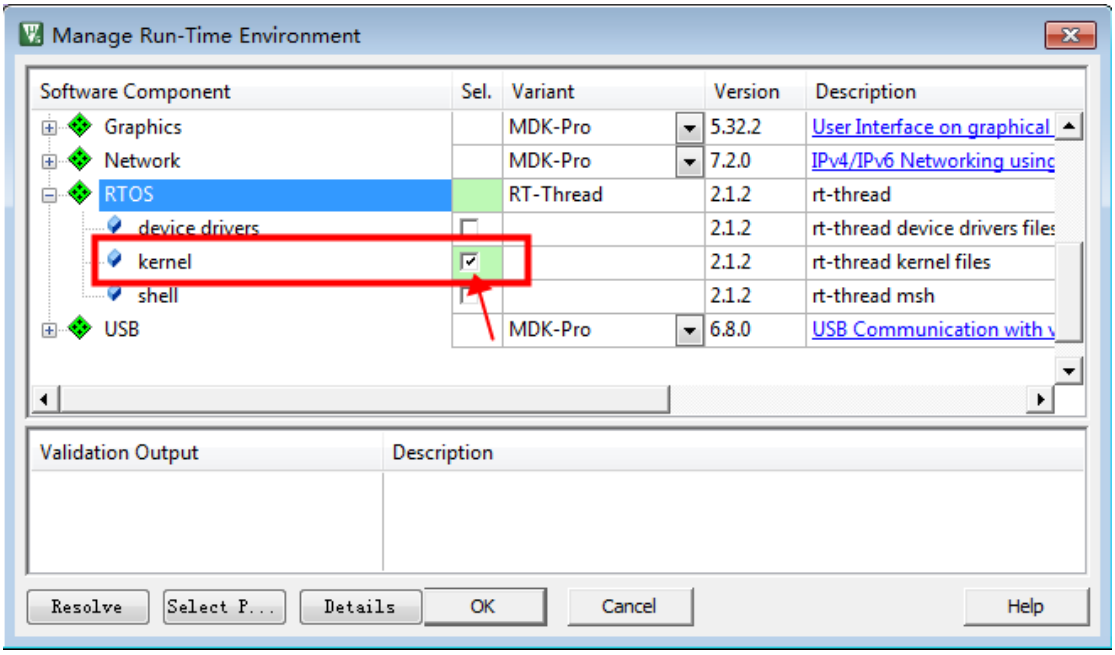

图 5：RT-Thread kernel 选择

2. 确定后，keil 界面上会加载 RT-Thread 的 kernel 文件，会根据当前选择芯片类型加入已移植完成的芯片内核代码、配置文件等。

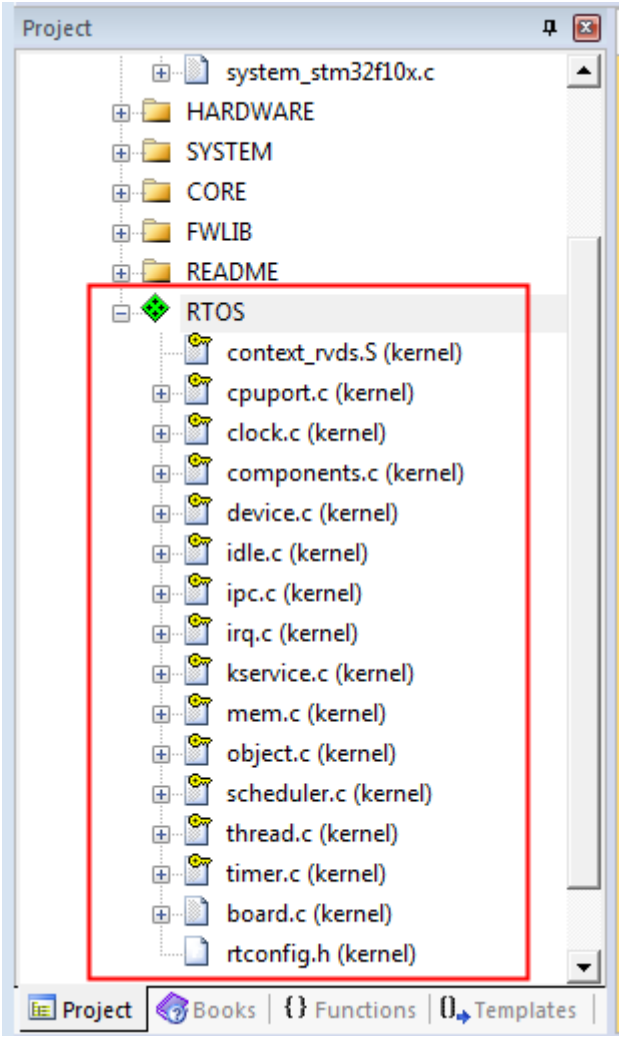

图 6：RT-Thread kernel 文件

其中：

Kernel 文件包括：
clock.c
components.c
device.c
idle.c
ipc.c
irq.c
kservice.c
mem.c
object.c
scheduler.c
thread.c
timer.c
Cortex-M 芯片内核移植代码：

cpuport.c
context_rvds.s
应用代码及配置文件：
board.c
rtconfig.h

三、修改源码适配开发板

1. 此时再次编译工程，编译器会提示有函数被重复定义了。需按照如下方式做一些修改：

1）修改 stm32fXXXX_it.c 文件（stm32f103 对应的文件为 stm32f10x_it.c，stm32f4 对应的文件为 stm32f4xx_it.c，stm32f7 对应的文件为 stm32f7xx_it.c），删除如下函数：

```
void HardFault_Handler(void);
void PendSV_Handler(void);
void SysTick_Handler(void);
```

2）按照应用代码 board.c 上的说明，依次修改相关代码：

修改 24 行：
如果是 F10，则#include “stm32f10x.h”
如果是 F4， 则#include “stm32f4xx.h”
如果是 F7， 则#include “stm32f7xx.h”

修改 48 行：在 rt_hw_board_init()函数内开启 Systick：

```
SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND);
```

修改 66 行：引入 Systick 中断服务函数**（注意，整个函数的注释都取消）。**

```
void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();
    rt_tick_increase();
    /* leave interrupt */
    rt_interrupt_leave();
}
```

这里特别提醒，对于使用 HAL 库的阿波罗 F429 或者 F767，我们还应该在 rt_hw_board_init 函数开头部分初始化 HAL 库和系统时钟：

```
HAL_Init();                          //初始化 HAL 库
Stm32_Clock_Init(360,25,2,8);        //设置时钟,180Mhz   这里是以 F429 为例
```

具体代码请参考对应的工程。

2. 修改启动文件，设置堆大小为 0。因为后续我们将采用 RT-Thread 管理内存堆。打开芯片对应的启动文件，例如是 stm32f103，启动文件为：
startup_stm32f10x.s

```
35  Stack_Size      EQU     0x00000400
36
37                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
38  Stack_Mem       SPACE   Stack_Size
39  __initial_sp
40
41  ; <h> Heap Configuration
42  ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
43  ; </h>
44
45  Heap_Size       EQU     0x00000000
46
47                  AREA    HEAP, NOINIT, READWRITE, ALIGN=3
48  __heap_base
49  Heap_Mem        SPACE   Heap_Size
50  __heap_limit
```

图 7 启动文件栈和堆修改

3. 最后，修改 main.c 文件，加入 RT-Thread 测试代码：

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

static struct rt_thread led0_thread;//线程控制块
static struct rt_thread led1_thread;//线程控制块
ALIGN(RT_ALIGN_SIZE)
static rt_uint8_t rt_led0_thread_stack[1024];//线程栈
static rt_uint8_t rt_led1_thread_stack[1024];//线程栈

//线程 LED0
static void led0_thread_entry(void* parameter)
{
    while(1)
    {
        LED0=0; //注意：F7 不支持位带操作，LED 操作请参考代码修改
        rt_thread_delay(RT_TICK_PER_SECOND/5);    //延时

        LED0=1;
        rt_thread_delay(RT_TICK_PER_SECOND/5);    //延时
    }
}

//线程 LED1
static void led1_thread_entry(void* parameter)
{
    while(1)
    {
        LED1=0;
        rt_thread_delay(RT_TICK_PER_SECOND/2);    //延时
```

```
        LED1=1;
        rt_thread_delay(RT_TICK_PER_SECOND/2);    //延时
    }
}


int main(void)
{


    LED_Init();     //初始化 LED

    // 创建静态线程
    rt_thread_init(&led0_thread,                    //线程控制块
                   "led0",                          //线程名字，在 shell 里面可以看到
                   led0_thread_entry,               //线程入口函数
                   RT_NULL,                         //线程入口函数参数
                   &rt_led0_thread_stack[0],        //线程栈起始地址
                   sizeof(rt_led0_thread_stack),    //线程栈大小
                   3,                               //线程的优先级
                   20);                             //线程时间片

    rt_thread_startup(&led0_thread);                 /启动线程 led0_thread，开启调度

    rt_thread_init(&led1_thread,                    //线程控制块
                   "led1",                          //线程名字，在 shell 里面可以看到
                   led1_thread_entry,               //线程入口函数
                   RT_NULL,                         //线程入口函数参数
                   &rt_led1_thread_stack[0],        //线程栈起始地址
                   sizeof(rt_led1_thread_stack),    //线程栈大小
                   3,                               //线程的优先级
                   20);

    rt_thread_startup(&led1_thread);                //启动线程 led1_thread，开启调度

}
```

该例程通过 rt_thread_init 函数创建两个静态线程 led0 和 led1，对应线程入口函数分别为 led0_thread_entry 和 led1_thread_entry，这两个函数分别控制 LED0 和 LED1 的状态翻转。然后程序通过 rt_thread_startup 函数分别启动这两个线程。

最后我们通过 ST-LINK 将程序下载到开发板，可以看到 LED0 和 LED1 轮流翻转。如有实验现象不一致，请参考对应开发板配套源码。

四、RT-Thread 程序执行流程分析

4.1 RT-Thread 入口

我们可以在 components.c 文件的 140 行看到#ifdef RT_USING_USER_MAIN 宏定义判断，这个宏是定义在 rtconfig.h 文件内的，而且处于开启状态。同时我们可以在 146 行看到#if defined (__CC_ARM)的宏定义判断，__CC_ARM 就是指 keil 的交叉编译器名称。

我们可以在这里看到定义了 2 个函数：$$Sub$$main()和$$Super$$main()函数；这里通过$$Sub$$main()函数在程序就如主程序之前插入一个例程，实现在不改变源代码的情况下扩展函数功能。链接器通过调用$$Sub$$Main()函数取代 main()，然后通过$$Super$$main 再次回到 main()

```
#if defined (__CC_ARM)
extern int $Super$$main(void);
/* re-define main function */
int $Sub$$main(void)
{
        rt_hw_interrupt_disable();
        rtthread_startup();
        return 0;
}
```

在$$Sub$$main 函数内调用了 rt_hw_interrutp_disable()和 rtthread_startup()两个函数。熟悉 RT-Thread 开发流程的，一看就知道这是标准的 RT-Thread 的启动入口。

其中：

rt_hw_interrupt_disable()：关中断操作，

rtthread_startup()：完成 systick 配置、timer 初始化/启动、idle 任务创建、应用线程初始化、调度器启动等工作。

```
int rtthread_startup(void)
{
                rt_hw_interrupt_disable();

        /* board level initalization
           * NOTE: please initialize heap insideboard initialization.
```

```
    */
    rt_hw_board_init();

    /* show RT-Thread version */
    rt_show_version();

    /* timer system initialization */
    rt_system_timer_init();

    /* scheduler system initialization */
    rt_system_scheduler_init();

    /* create init_thread */
    rt_application_init();

    /* timer thread initialization */
    rt_system_timer_thread_init();

    /* idle thread initialization */
    rt_thread_idle_init();

    /* start scheduler */
    rt_system_scheduler_start();

    /* never reach here */
    return 0;
}
```

rt_hw_board_init()：该函数定义在 board.c 文件内，需要修改 systick 配置
rt_system_timer_init()/rt_system_timer_thread_init()：timer 初始化/启动
rt_thread_idle_init()：idle 任务创建
rt_application_init()：应用线程初始化
rt_system_scheduler_start()：调度器启动

4.12 应用线程入口

```
rt_application_init()
void rt_application_init(void)
{
    rt_thread_t tid;

#ifdef RT_USING_HEAP
    tid = rt_thread_create("main",main_thread_entry, RT_NULL,
          RT_MAIN_THREAD_STACK_SIZE,RT_THREAD_PRIORITY_MAX / 3, 20);
```

```
        RT_ASSERT(tid != RT_NULL);
#else
        rt_err_t result;
        tid = &main_thread;
        result = rt_thread_init(tid,"main", main_thread_entry, RT_NULL,main_stack,
sizeof(main_stack),RT_THREAD_PRIORITY_MAX / 3, 20);
        RT_ASSERT(result == RT_EOK);
#endif
        rt_thread_startup(tid);
}
```

在这里，我们可以看到应用线程创建了一个名为 main_thread_entry 的任务，并且已经启动了该任务。我们再次来看一下 man_thread_entry 任务。

```
/* the system main thread */
void main_thread_entry(void*parameter)
{
        extern int main(void);
        extern int $Super$$main(void);

        /* RT-Thread components initialization */
        rt_components_init();

        /* invoke system main function */
#if defined (__CC_ARM)
        $Super$$main(); /* for ARMCC. */
#elif defined(__ICCARM__) ||defined(__GNUC__)
        main();
#endif
}
```

man_thread_entry 任务完成了 2 个工作：调用 rt_components_init()、进入应用代码真正的 main 函数。

在这里我们看到了$$Super$$main()的调用，在前面我们讲了调用该函数可用来回到 main()的。

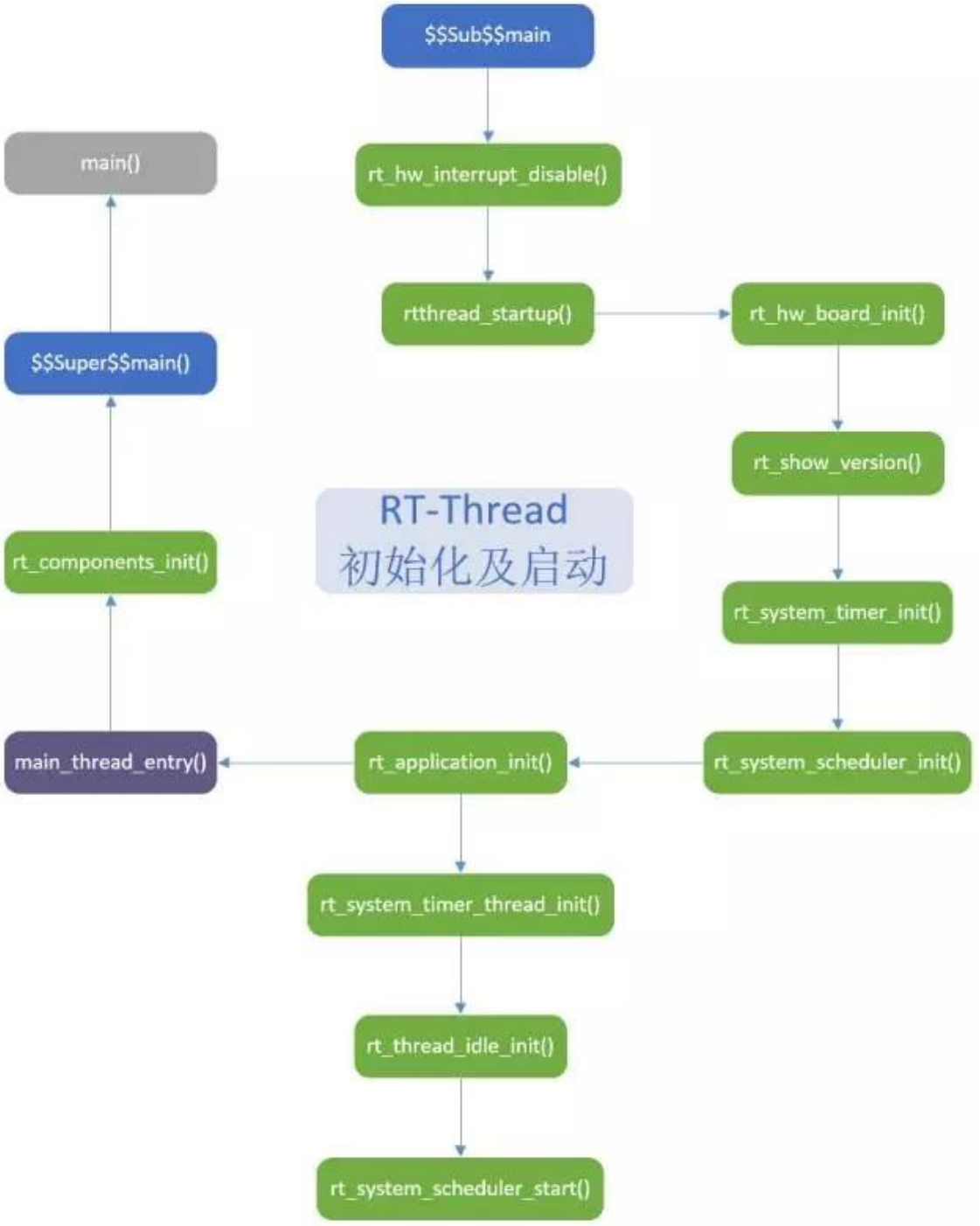

图 8 RT-Thread 初始化及启动流程

从以上分析可以，正是由于在 rtconfig.h 内开启了 RT_USING_USER_MAIN 选项，编译器在 main 之前插入了$$Sub$$main()，完成了 RT-Thread 初始化及调度器启动工作。并且通过创建 main_thread_entry 任务，并通过$$Super$$main()回到 main()函数。这样看来 main()函数其实只是 RT-Thread 的一个任务，该任务的优先级为 RT_THREAD_PRIORITY_MAX / 3，任务栈为 RT_MAIN_THREAD_STACK_SIZE。

```
25  // </c>
26  // <c1>Using user main
27  //   <i>Using user main
28  #define RT_USING_USER_MAIN
29  // </c>
30  // <o>the size of main thread<1-4086>
31  //   <i>Default: 512
32  #define RT_MAIN_THREAD_STACK_SIZE     256
```

图 9 RT_USING_USER_MAIN 选项

正点原子&RT-Thread