

# 实验六：RT-Thread 操作系统认识实验

## 一、实验目的

- 1.了解RT-Thread操作系统的构成；
- 2.学会利用工程模板建立新工程的方法；
- 3.学习利用移植的方法建立新工程的方法；
- 4.掌握基于RT-Thread操作系统工程的调试方法。

## 二、实验仪器与设备

计算机、RT-Thread 操作系统、相关软件

## 三、实验内容

### (1) RT-Thread 操作系统简介

RT-Thread 是一个集实时操作系统（RTOS）内核、中间件组件和开发者社区于一体的技术平台，由熊谱翔先生带领并集合开源社区力量开发而成，RT-Thread 也是一个组件完整丰富、高度可伸缩、简易开发、超低功耗、高安全性的物联网操作系统。RT-Thread 具备一个 IoT OS 平台所需的所有关键组件，例如 GUI、网络协议栈、安全传输、低功耗组件等等。经过 13 年的累积发展，RT-Thread 已经拥有一个国内最大的嵌入式开源社区，同时被广泛应用于能源、车载、医疗、消费电子等多个行业，累积装机量超过两千万台，成为国人自主开发、国内最成熟稳定和装机量最大的开源 RTOS。

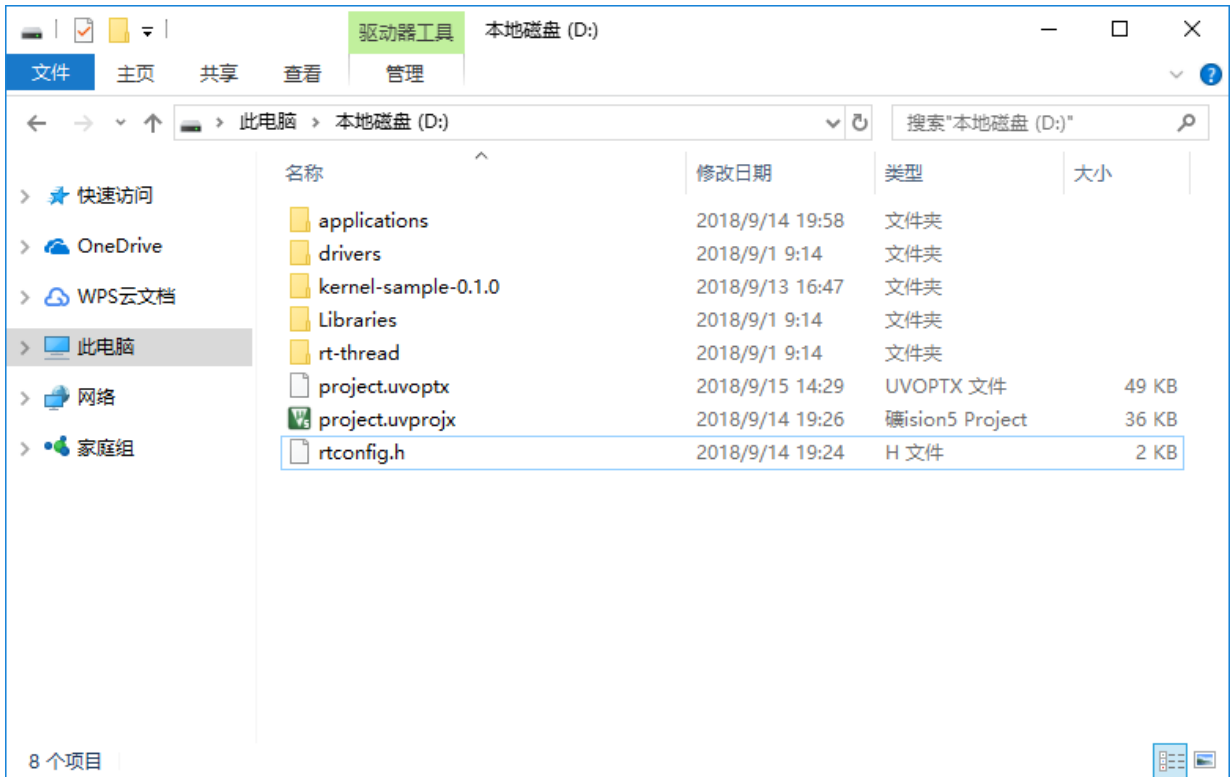
RT-Thread 拥有良好的软件生态，支持市面上所有主流的编译工具如 GCC、Keil、IAR 等，工具链完善、友好，支持各类标准接口，如 POSIX、CMSIS、C++应用环境、Javascript 执行环境等，方便开发者移植各类应用程序。商用支持所有主流 MCU 架构，如 ARM Cortex-M/R/A, MIPS, X86, Xtensa, C-Sky, RISC-V，几乎支持市场上所有主流的 MCU 和 Wi-Fi 芯片。

### (2) RT-Thread 操作系统文件结构

作为一个操作系统，RT-Thread 的代码规模怎么样呢？在弄清楚这些之前，我们先要做的就是获得与本文相对应的 RT-Thread 的例子，这份例子可以从以下链接获得：

**RT-Thread Simulator 例程**

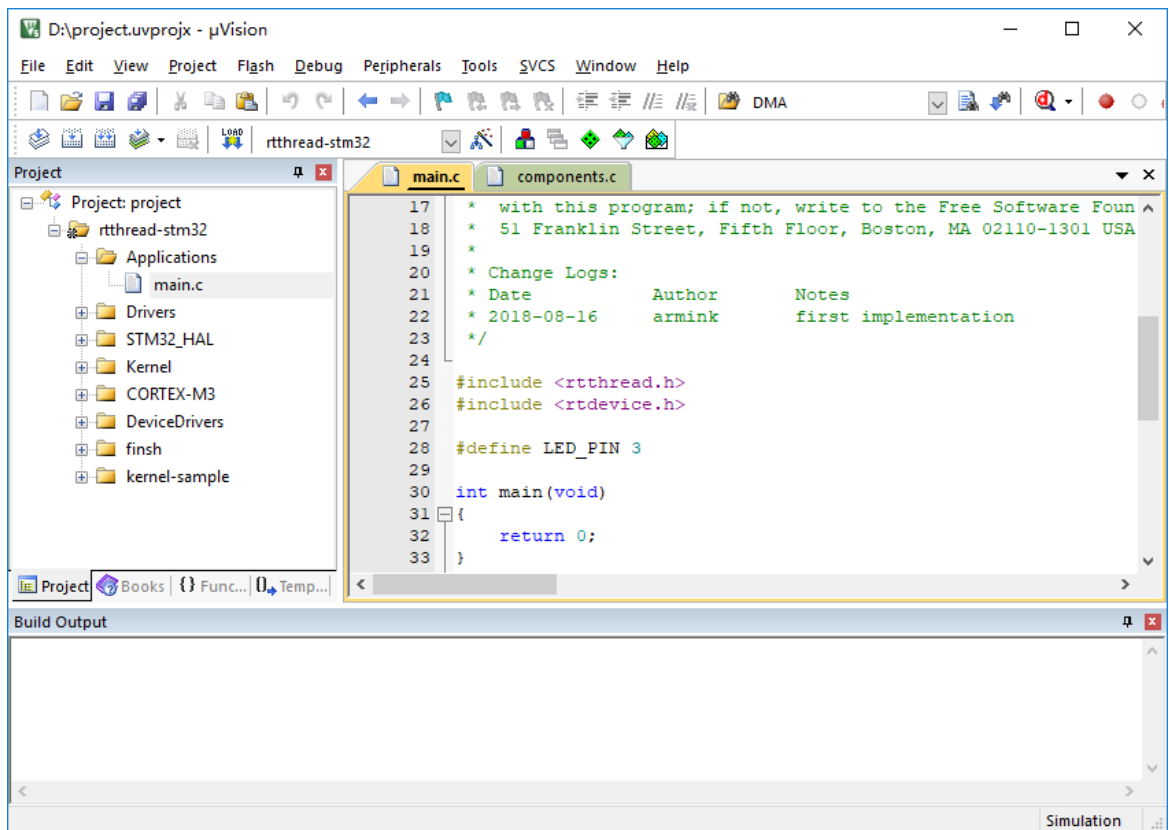
这个例子是一个压缩包文件，将它解压，我们这里解压到 D:/。解压完成后的目录结构如下图所示：



各个目录所包含的文件类型的描述如下表所示：

目录名	描述
applications	RT-Thread 应用程序。
rt-thread	RT-Thread 的源文件。
- components	RT-Thread 的各个组件目录。
- include	RT-Thread 内核的头文件。
- libcpu	各类芯片的移植代码，此处包含了 STM32 的移植文件。
- src	RT-Thread 内核的源文件。
- tools	RT-Thread 命令构建工具的脚本文件。
drivers	RT-Thread 的驱动，不同平台的底层驱动具体实现。
Libraries	ST 的 STM32 固件库文件。
kernel-sample-0.1.0	RT-Thread 的内核例程。

在目录下，有一个 project.uvprojx 文件，它是本文内容所引述的例程中的一个 MDK5 工程文件，双击“project.uvprojx”图标，打开此工程文件：



在工程主窗口的左侧“Project”栏里可以看到该工程的文件列表，这些文件被分别存放到如下几个组内，分别是：

目录组	描述
Applications	对应的目录为 rtthread_simulator_v0.1.0/applications，它用于存放用户应用代码。
Drivers	对应的目录为 rtthread_simulator_v0.1.0/drivers，它用于存放 RT-Thread 底层的驱动代码。
STM32_HAL	对应的目录为 rtthread_simulator_v0.1.0/Libraries/CMSIS/Device/ST/STM32F1xx，它用于存放 STM32 的固件库文件。
kernel-sample	对应的目录为 rtthread_simulator_v0.1.0/kernel-sample-0.1.0，它用于存放 RT-Thread 的内核例程。
Kernel	对应的目录为 rtthread_simulator_v0.1.0/src，它用于存放 RT-Thread 内核核心代码。
CORTEX-M3	对应的目录为 rtthread_simulator_v0.1.0/rt-thread/libcpu，它用于存放 ARM Cortex-M3 移植代码。
DeviceDrivers	对应的目录为 rtthread_simulator_v0.1.0/rt-thread/components/drivers，它用于存放 RT-Thread 驱动框架源码。
finsh	对应的目录为 rtthread_simulator_v0.1.0/rt-thread/components/finsh，它用于存放 RT-Thread 命令行 finsh 命令行组件。

### （3）利用工程模板建立新工程的方法

一般了解一份代码大多从启动部分开始，同样这里也采用这种方式，先寻找启动的源头。以 MDK-ARM 为例，MDK-ARM 的用户程序入口为 `main()` 函数，位于 `main.c` 文件中。系统启动后先从汇编代码 `startup_stm32f103xe.s` 开始运行，然后跳转到 C 代码，进行 RT-Thread 系统功能初始化，最后进入用户程序入口 `main()`。

下面我们来看看在 `components.c` 中定义的这段代码：

```
//components.c 中定义
/* re-define main function */
int $Sub$$main(void)
{
    rt_hw_interrupt_disable();
    rtthread_startup();
    return 0;
}
```

在这里 `$Sub$$main` 函数仅仅调用了 `rtthread_startup()` 函数。RT-Thread 支持多种平台和多种编译器，而 `rtthread_startup()` 函数是 RT-Thread 规定的统一入口点，所以 `$Sub$$main` 函数只需调用 `rtthread_startup()` 函数即可。例如采用 GNU GCC 编译器编译的 RT-Thread，就是直接从汇编启动代码部分跳转到 `rtthread_startup()` 函数中，并开始第一个 C 代码的执行的。在 `components.c` 的代码中找到 `rtthread_startup()` 函数，我们将可以看到 RT-Thread 的启动流程：

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

    /* board level initialization
     * NOTE: please initialize heap inside board 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();

#ifdef RT_USING_SIGNALS
    /* signal system initialization */
    rt_system_signal_init();
#endif

    /* 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;  
}
```

这部分启动代码，大致可以分为四个部分：

- 初始化与系统相关的硬件；
- 初始化系统内核对象，例如定时器，调度器；
- 初始化系统设备，这个主要是为 RT-Thread 的设备框架做的初始化；
- 初始化各个应用线程，并启动调度器。

## 用户入口代码

上面的启动代码基本上可以说都是和 RT-Thread 系统相关的，那么用户如何加入自己的应用程序的初始化代码呢？RT-Thread 将 main 函数作为了用户代码入口，只需要在 main 函数里添加自己的代码即可。

```
int main(void)  
{  
    /* user app entry */  
    return 0;  
}
```

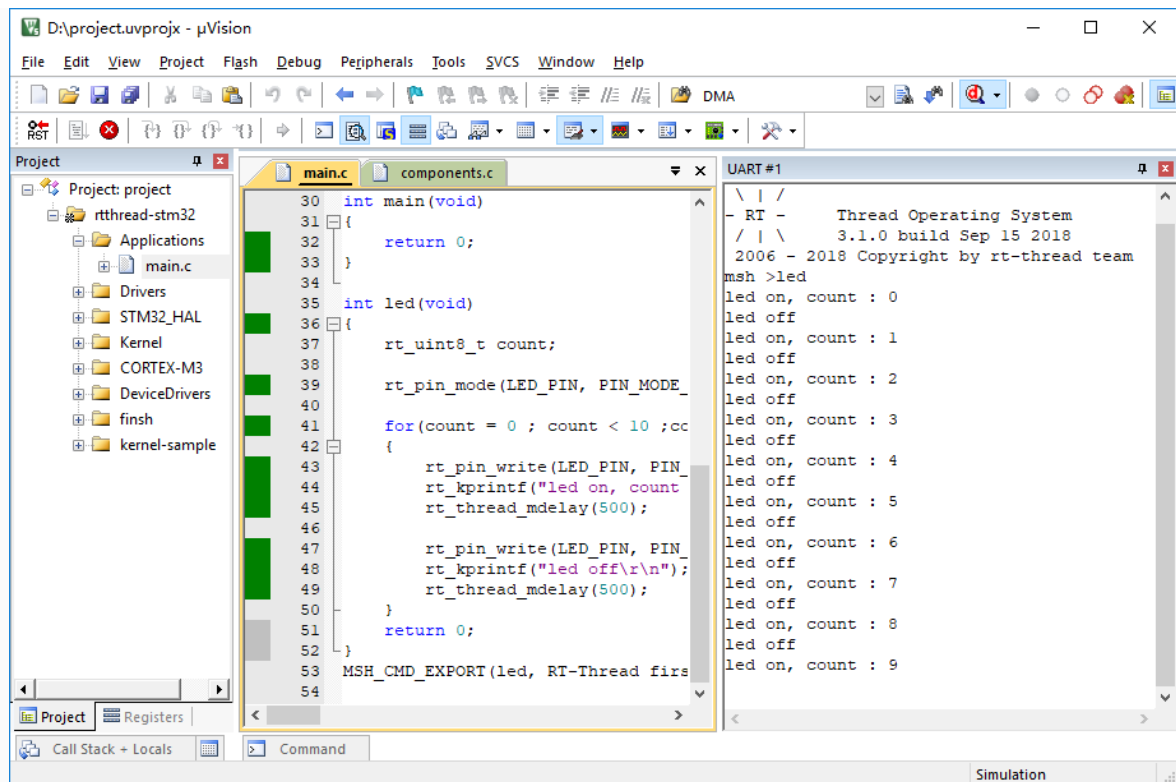
### 提示

注：为了在进入 main 程序之前，完成系统功能初始化，可以使用 `$sub$$` 和 `$super$$` 函数标识符在进入主程序之前调用另外一个例程，这样可以让用户不用去管 main() 之前的系统初始化操作。详见 [ARM® Compiler v5.06 for μVision® armlink User Guide](#)。

## 跑马灯的例子

对于从事电子方面开发的技术工程师来说，跑马灯大概是最简单的例子，就类似于每种编程语言中程序员接触的第一个程序 Hello World 一样，所以这个例子就从跑马灯开始。让它定时地对 LED 进行更新（关或灭）。

我们 UART#1 中输入 msh 命令：led 然后回车就可以运行起来了，如图所示：



## 跑马灯例子

```

/*
 * 程序清单：跑马灯例程
 *
 * 跑马灯大概是最简单的例子，就类似于每种编程语言中程序员接触的第一个程序
 * Hello World 一样，所以这个例子就从跑马灯开始。创建一个线程，让它定时地对
 * LED 进行更新（关或灭）
 */

```

```

int led(void)
{
    rt_uint8_t count;

    rt_pin_mode(LED_PIN, PIN_MODE_OUTPUT);

    for(count = 0 ; count < 10 ;count++)
    {
        rt_pin_write(LED_PIN, PIN_HIGH);
        rt_kprintf("led on, count : %d\r\n", count);
        rt_thread_mdelay(500);

        rt_pin_write(LED_PIN, PIN_LOW);
        rt_kprintf("led off\r\n");
        rt_thread_mdelay(500);
    }
    return 0;
}
MSH_CMD_EXPORT(led, RT-Thread first led sample);

```

#### (4) 利用内核移植建立工程

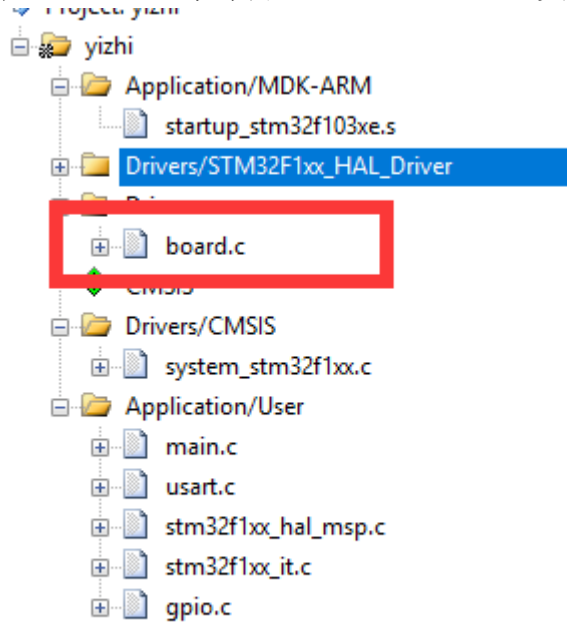
第一步：复制给定的，MDK5 工程

第二步：参考 0-bare-metal 完成 board.c、board.h 文件

0-bare-metal	2018/8/14 17:53	文件夹	
1-basic	2018/8/14 17:53	文件夹	
2-os-tick-porting	2018/8/14 17:53	文件夹	
3-console-porting	2018/8/14 17:53	文件夹	
4-heap-init	2018/8/14 17:53	文件夹	
5-finish	2018/8/14 17:53	文件夹	
rt-thread-source-code	2018/8/14 17:53	文件夹	
README.md	2018/8/14 17:53	MD 文件	0 KB

<https://blog.csdn.net/spu20134823091>

在 Drivers 组下添加 board.c、board.h 文件



添加 board.c 文件

实现 board.c 里面关于关于时钟配置、串口初始化、GPIO 初始化等函数

```
void bsp_uart_init(void)
{
    MX_USART1_UART_Init();
}
```

```
void bsp_uart_send(char c)
{
```

```

        while ((__HAL_UART_GET_FLAG(&huart1, UART_FLAG_TXE) ==
RESET));
        huart1.Instance->DR = c;
    }
void bsp_led_init(void)
{
    MX_GPIO_Init();
}
void bsp_led_on(void)
{
    HAL_GPIO_WritePin(LED_GPIO_Port,LED_Pin, GPIO_PIN_RESET);
}
void bsp_led_off(void)
{
    HAL_GPIO_WritePin(LED_GPIO_Port,LED_Pin, GPIO_PIN_SET);
}
void rt_hw_board_init()
{
    /* HAL_Init() function is called at the beginning of program after reset and
before      *      the clock configuration. */
    HAL_Init();
    /* Clock Config:
    * System Clock : 80M
    * HCLK : 80M
    * PCLK1 : 80M
    * PCLK2 : 80M
    * SDMMC1 : 48M
    * USART1 : PCLK2 */
    SystemClock_Config();
    bsp_led_init();
    bsp_uart_init();
}
void SystemClock_Config(void)

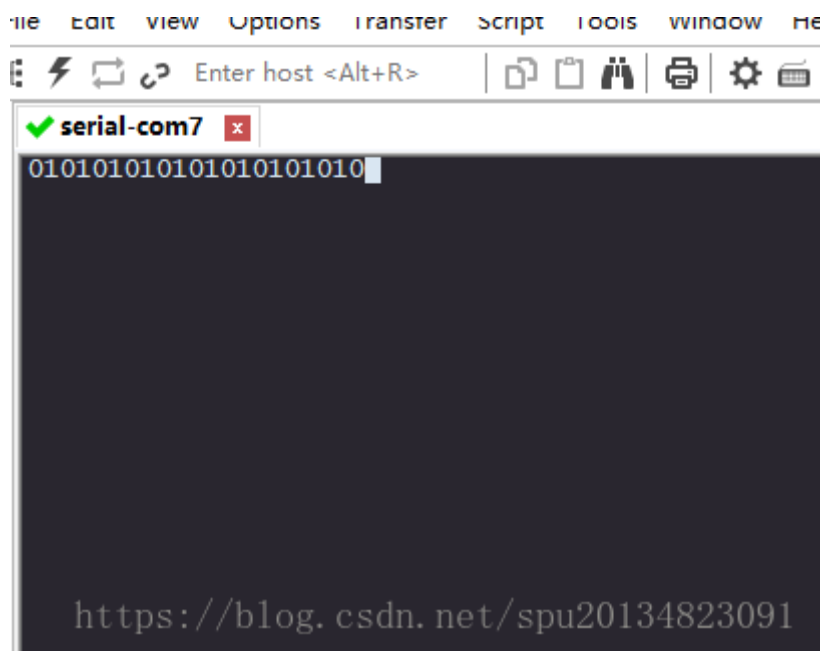
void _Error_Handler(char *file, int line)

```

以上两个函数采用 cubemx 工具生成的函数，main 函数和例程中一样

现象：



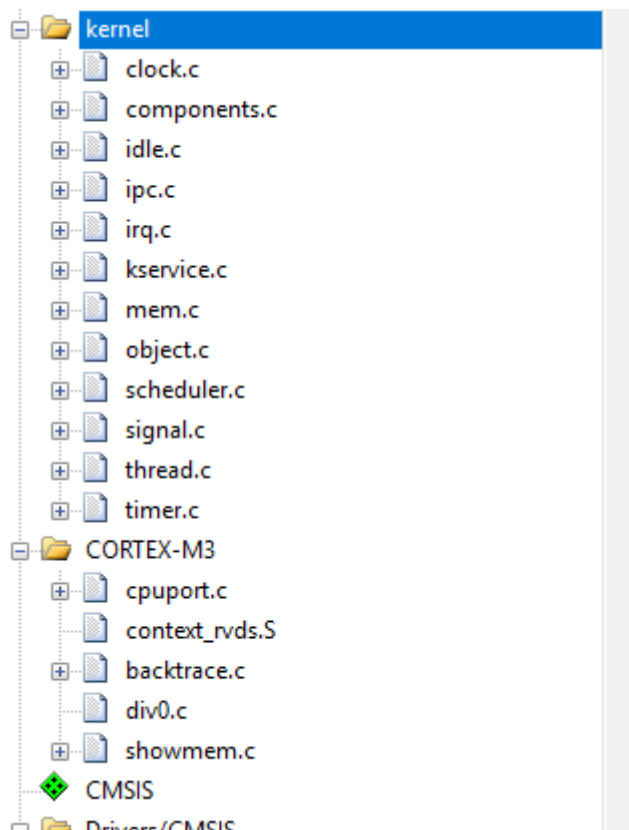


### 第三步：添加 RT-Thread 源码

拷贝 RT-Thread 源代码到工程路径下，当然也可以不移动，就是指定路径的时候长一点，没关系

添加两个新组一个叫做 `kernel`，防止内核代码，另一个叫做 `CROTEX-M3` 放置平台相关代码，我用的是 STM32F103ZET6 使用的是 `cortex-m3` 内核。

添加相关文件到对应的组下面，添加完如图，对了还有一个 `rtconfig.h` 文件，用于配置和裁剪内核。



添加代码之后别忘了添加头文件，rtconfig.h 文件放在了工程的根目录之下，别忘了添加头文件。

在编译之后有了有符号重定义，注释掉原有的函数

在编译之后有了有符号重定义，注释掉原有的函数

```
//void HardFault_Handler(void)
//{
// /* USER CODE BEGIN HardFault_IRQn 0 */
// /* USER CODE END HardFault_IRQn 0 */
// while (1)
// {
// /* USER CODE BEGIN W1_HardFault_IRQn 0 */
// /* USER CODE END W1_HardFault_IRQn 0 */
// }
// /* USER CODE BEGIN HardFault_IRQn 1 */
// /* USER CODE END HardFault_IRQn 1 */ //}

//void PendSV_Handler(void)
//{
// /* USER CODE BEGIN PendSV_IRQn 0 */
// /* USER CODE END PendSV_IRQn 0 */
// /* USER CODE BEGIN PendSV_IRQn 1 */
// /* USER CODE END PendSV_IRQn 1 */
//}
```

编译下载正常运行

第四步：console 移植，实现控制台输出功能

实现函数

```
void rt_hw_console_output(const char *str)
{
    RT_ASSERT(str != RT_NULL);
    while (*str != '\0')
    {
        if (*str == '\n')
        {
            bsp_uart_send('\r');
        }
        bsp_uart_send(*str++);
    }
}
```

这个函数主要是实现 bsp\_uart\_send,此函数在第二步已经实现。

接下来修改 SysTick\_Handler 函数

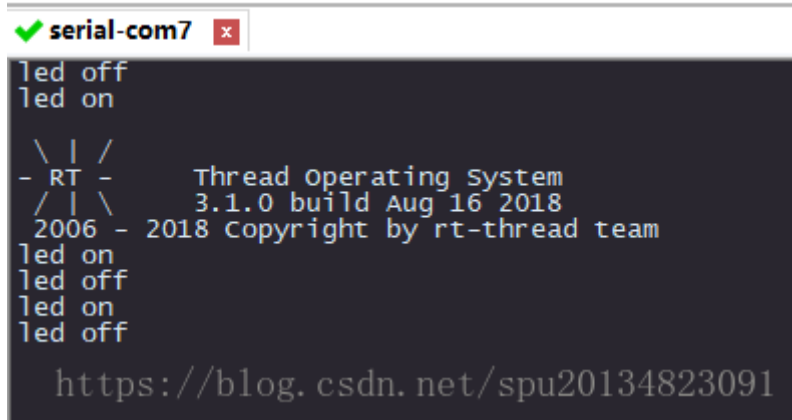
```
/** * @brief This function handles System tick timer. */
void SysTick_Handler(void)
{
    rt_interrupt_enter();
    rt_tick_increase();
    rt_interrupt_leave();
}
```

注意包含 rtthread.h 头文件

修改 main 函数如下

```
int main(void)
{
    while (1)
    {
        /* USER CODE END WHILE */
        /* USER CODE BEGIN 3 */
        rt_kprintf("led on\n");
        rt_thread_delay(RT_TICK_PER_SECOND);
        rt_kprintf("led off\n");
        rt_thread_delay(RT_TICK_PER_SECOND);
    }
    /* USER CODE END 3 */
}
```

编译下载输出如图



```
✓ serial-com7 x
led off
led on

  \ | /
- RT -   Thread Operating System
  / | \   3.1.0 build Aug 16 2018
2006 - 2018 Copyright by rt-thread team
led on
led off
led on
led off

https://blog.csdn.net/spu20134823091
```

第五步：内存堆配置，实现动态内存管理功能

在 `rt_config.h` 中添加宏

```
//#define RT_USING_NOHEAP
#define RT_USING_SMALL_MEM
#define RT_USING_HEAP
```

注意：注释掉 `RT_USING_NOHEAP`，如果有的话

在 `rt_hw_board_init` 函数中添加动态内存管理初始化函数

```
void rt_hw_board_init()
{
    static uint8_t heap_buf[10 * 1024];
    .....
    rt_system_heap_init(heap_buf, heap_buf + sizeof(heap_buf) - 1);
}
```

在 `main` 函数中初始化一个线程函数

```
#define THREAD_PRIORITY 25
#define THREAD_STACK_SIZE 512
#define THREAD_TIMESLICE 5
void test_thread_entry(void *parameter)
{
    while (1)
    {
        rt_kprintf("enter test thread\n");
        rt_thread_delay(RT_TICK_PER_SECOND);
    }
}
/** * @brief The application entry point. * * @retval None */
int main(void)
{
```

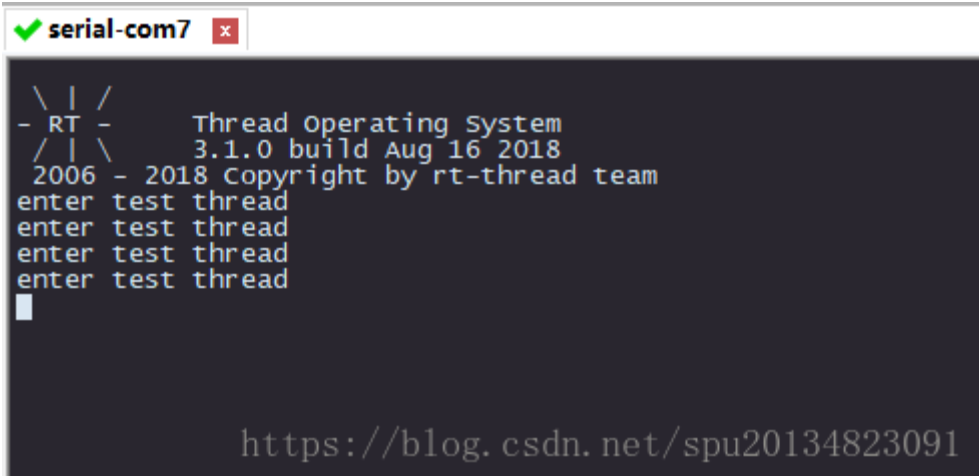
```

rt_thread_t tid;
tid = rt_thread_create("test",
                        test_thread_entry,
                        RT_NULL,
                        THREAD_STACK_SIZE,
                        THREAD_PRIORITY,
                        THREAD_TIMESLICE);

if (tid != RT_NULL)
{
    rt_thread_startup(tid);
    return 0;
}
else
{
    return -1;
}
}

```

编译下载，输出如图



```

✓ serial-com7 x
\  |  /
- RT -   Thread Operating System
/  |  \   3.1.0 build Aug 16 2018
2006 - 2018 Copyright by rt-thread team
enter test thread
enter test thread
enter test thread
enter test thread
█

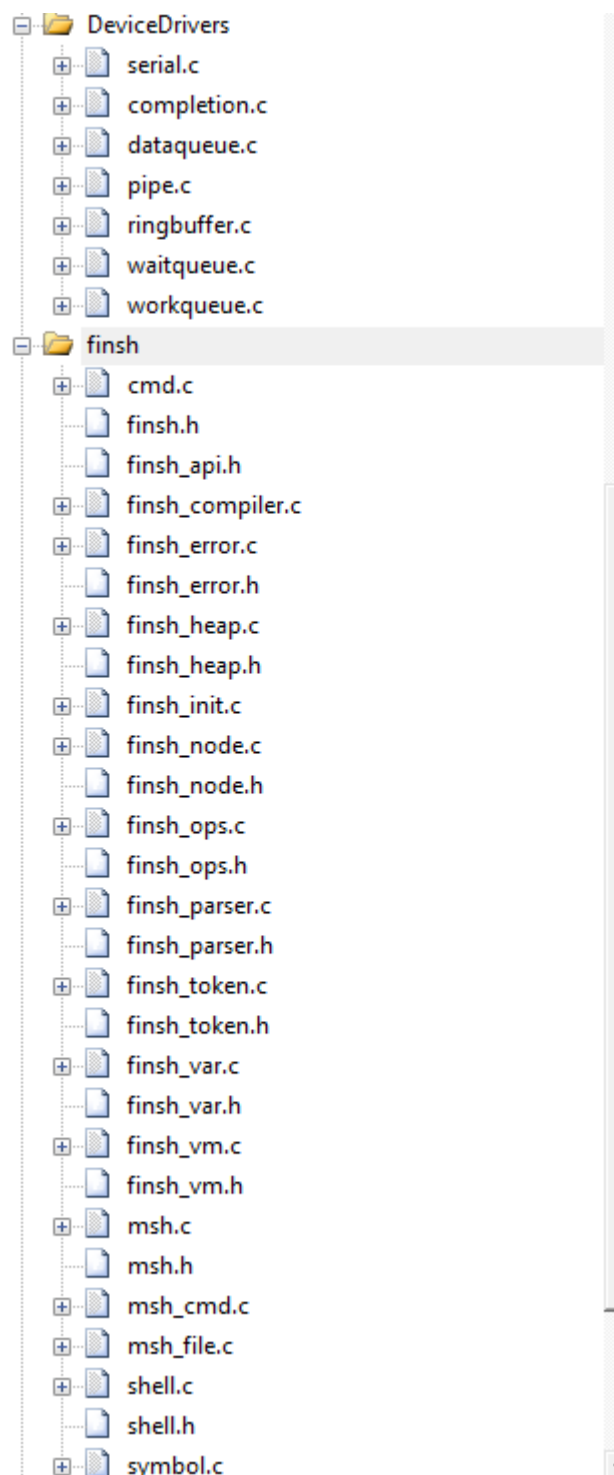
https://blog.csdn.net/spu20134823091

```

第六步：实现 Finsh Shell 功能

将 components 组件文件夹下的 drivers 和 finsh 文件添加到工程

新建 finsh 组和 DeviceDrivers 组，添加文件



在前几步添加文件的时候在 **Kernel** 组莫名少添加了 **device.c** 文件在调试的时候浪费很长时间，

在官方的 1-basic 的 **rt-thread\src** 文件下也没有，感受到来自 NPC 的调戏。

添加代码之后先修改 **board.c** 文件里的 **rt\_hw\_board\_init** 函数，

```

/** * This function will initial STM32 board. */
void rt_hw_board_init()
{
    static uint8_t heap_buf[10 * 1024];
    /* HAL_Init() function is called at the beginning of program after reset and
before      * the clock configuration. */
    HAL_Init();
    /* Clock Config:
    * System Clock : 80M
    * HCLK : 80M
    * PCLK1 : 80M
    * PCLK2 : 80M
    * SDMMC1 : 48M
    * USART1 : PCLK2
    */
    SystemClock_Config();
    /* Configure the SysTick interrupt time */
    HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()
RT_TICK_PER_SECOND);
    /* Configure the SysTick */
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
    /* SysTick_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
    //bsp_led_init();
    stm32_hw_usart_init();

#ifdef RT_USING_CONSOLE
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif
#ifdef RT_USING_HEAP
    rt_system_heap_init(heap_buf, heap_buf + sizeof(heap_buf) - 1);
#endif
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif
}
stm32_hw_usart_init();

```

用来初始化并注册串口

屏蔽掉 board.c 文件里面串口初始化函数、串口发送函数、控制台输出函数，工作交给 drv\_usart.c。

接下来修改 rtconfig.h 中的宏定义

添加#define BSP\_USING\_UART1 等

添加后 rtconfig.h 如下

```
#ifndef RT_CONFIG_H__
#define RT_CONFIG_H__
/* Automatically generated file; DO NOT EDIT. */
/* RT-Thread Configuration */
/* RT-Thread Kernel */
#define RT_NAME_MAX 8
#define RT_ALIGN_SIZE 4
#define RT_THREAD_PRIORITY_32
#define RT_THREAD_PRIORITY_MAX 32
#define RT_TICK_PER_SECOND 1000
#define RT_USING_OVERFLOW_CHECK
#define RT_USING_HOOK
#define RT_IDLE_HOOK_LIST_SIZE 4
#define IDLE_THREAD_STACK_SIZE 256
#define RT_DEBUG
/* Inter-Thread communication */
#define RT_USING_SEMAPHORE
#define RT_USING_MUTEX
#define RT_USING_EVENT
#define RT_USING_MAILBOX
#define RT_USING_MESSAGEQUEUE
/* Memory Management */
// #define RT_USING_NOHEAP
#define RT_USING_SMALL_MEM
#define RT_USING_HEAP
/* Kernel Device Object */
#define RT_USING_DEVICE
#define RT_USING_CONSOLE
#define RT_CONSOLEBUF_SIZE 128
#define RT_CONSOLE_DEVICE_NAME "uart1"
/* RT-Thread Components */
#define RT_USING_COMPONENTS_INIT
#define RT_USING_USER_MAIN
#define RT_MAIN_THREAD_STACK_SIZE 2048
#define RT_MAIN_THREAD_PRIORITY 10
/* C++ features */
/* Command shell */
#define RT_USING_FINSH
#define FINSH_THREAD_NAME "tshell"
#define FINSH_USING_HISTORY
#define FINSH_HISTORY_LINES 5
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION
#define FINSH_THREAD_PRIORITY 20
#define FINSH_THREAD_STACK_SIZE 4096
```



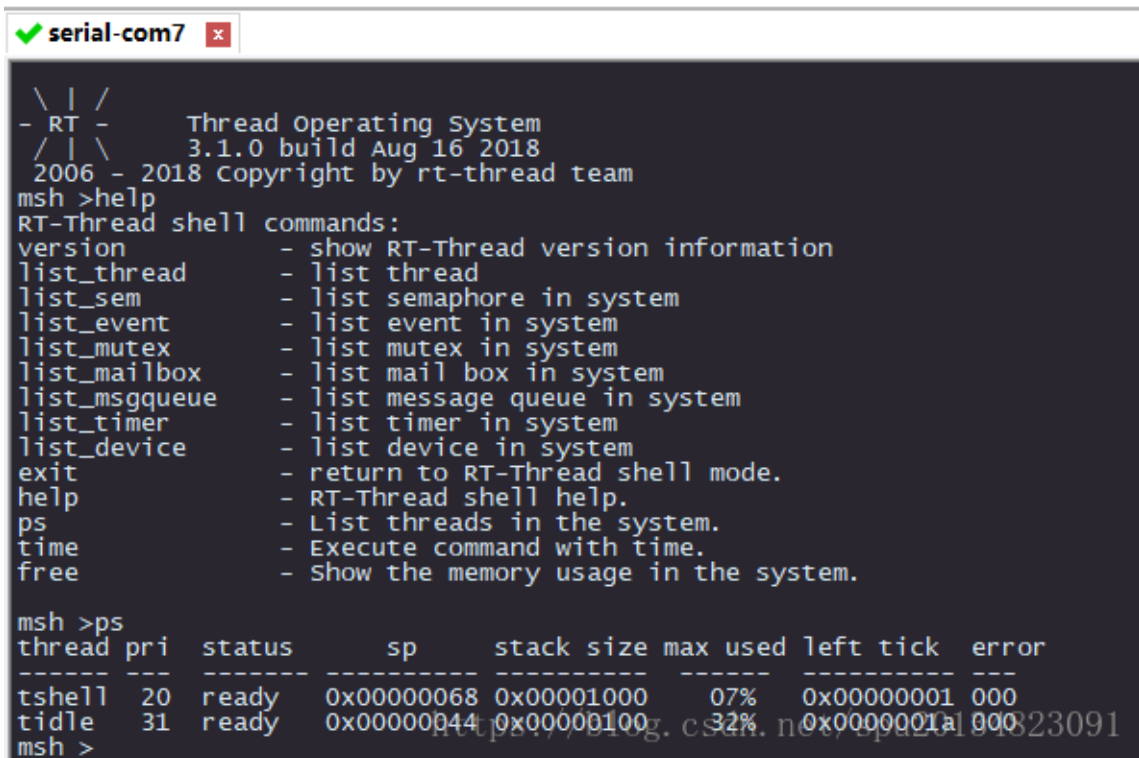
```

#define FINSH_CMD_SIZE 80
#define FINSH_USING_MSH
#define FINSH_USING_MSH_DEFAULT
#define FINSH_ARG_MAX 10
/* Device virtual file system */
/* Device Drivers */
#define RT_USING_DEVICE_IPC
#define RT_PIPE_BUFSZ 512
#define RT_USING_SERIAL
/* Env config */
#define SYS_PKGS_DOWNLOAD_ACCELERATE
#define BSP_USING_UART1
#endif

```

之前 mian 函数中的初始化函数去掉

编译下载如图



```

\ | /
- RT -   Thread operating system
/ | \   3.1.0 build Aug 16 2018
2006 - 2018 Copyright by rt-thread team
msh >help
RT-Thread shell commands:
version          - show RT-Thread version information
list_thread      - list thread
list_sem         - list semaphore in system
list_event       - list event in system
list_mutex       - list mutex in system
list_mailbox     - list mail box in system
list_msgqueue    - list message queue in system
list_timer       - list timer in system
list_device      - list device in system
exit             - return to RT-Thread shell mode.
help            - RT-Thread shell help.
ps              - List threads in the system.
time            - Execute command with time.
free            - Show the memory usage in the system.

msh >ps
thread pri  status      sp      stack size max used left tick  error
-----
tshell   20  ready      0x00000068 0x00001000    07%  0x00000001 000
tidle    31  ready      0x00000044 0x00000100    32%  0x0000001a 000
msh >

```

成功运行，

#### 四、实验步骤

按要求逐步操作以上内容，并记录运行结果。

#### 五、要求

编写实验报告。