

博客链接：原文：[https://blog.csdn.net/Sanjay\\_Wu/article/details/86582759](https://blog.csdn.net/Sanjay_Wu/article/details/86582759)

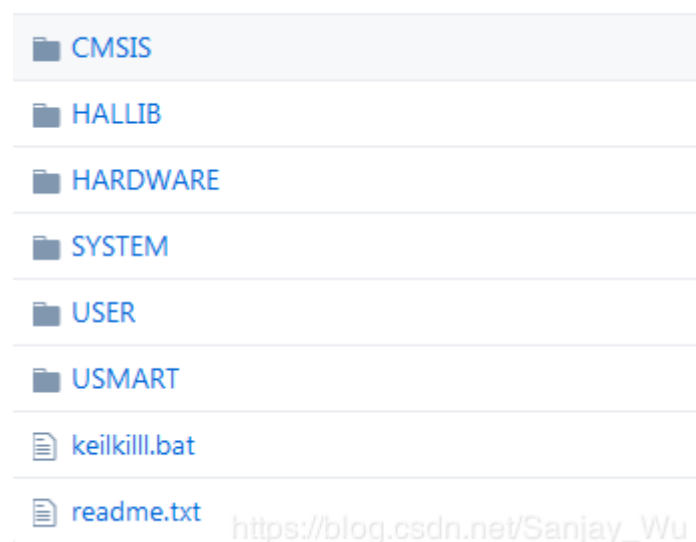
博客的格式更好，大家可以点开看下

## 前言

从本文开始，记录自己的 RT-Thread 学习笔记，基于 STM32L475VET6 讲解，相关开发板用 RTT&正点原子的潘多拉 IoT Board 开发板。本文先从 Nano 开始学起，个人觉得对于初学者，还是先学会 Nano 的移植，把内核部分向学一遍，再去学组件和设备驱动以及其他的東西，这里包括 RT-Thread 的内核移植、FinSH 移植，相关代码到 GitHub 下载：[https://github.com/sanjaywu/STM32L475\\_PANDORA\\_RT-Thread\\_DEMO](https://github.com/sanjaywu/STM32L475_PANDORA_RT-Thread_DEMO)

## 一、获取裸机工程

1、裸机工程可到 GitHub 下载：[https://github.com/sanjaywu/STM32L475\\_PANDORA\\_DEMO](https://github.com/sanjaywu/STM32L475_PANDORA_DEMO)，下载完成之后，打开工程文件夹，可以发现如下文件：



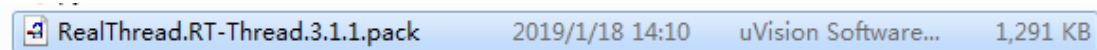
2、接着我们把 HARDWARE、SYSTEM 和 USMART 这三个文件删除，HARDWARE 文件夹是裸机的外设驱动，在讲解移植的时候不需要用到，SYSTEM 文件夹有 delay 延时、串口驱动和相关类型宏定义，在移植 RT-Thread 的时候，我们会重新实现 delay 延时和串口驱动以及类型宏定义。

## 二、下载 RT-Thread Nano 源码

1、RT-Thread Master 的源码可从 RT-Thread GitHub 仓库下载，Nano 就是从里面扣出来的，去掉了一些组件和各种开发板的 BSP，保留了 OS 的核心功能，但足够我们使用。RT-Thread 官方并没有将抠出来的 Nano 放到他们的官方网站，而是作为一个 Package 放在了 KEIL 网站：<http://www.keil.com/dd2/pack/>，目前最新的是 3.1.1 版本，打开这条连接，然后拉到下面找到 RT-Thread 的 Package：

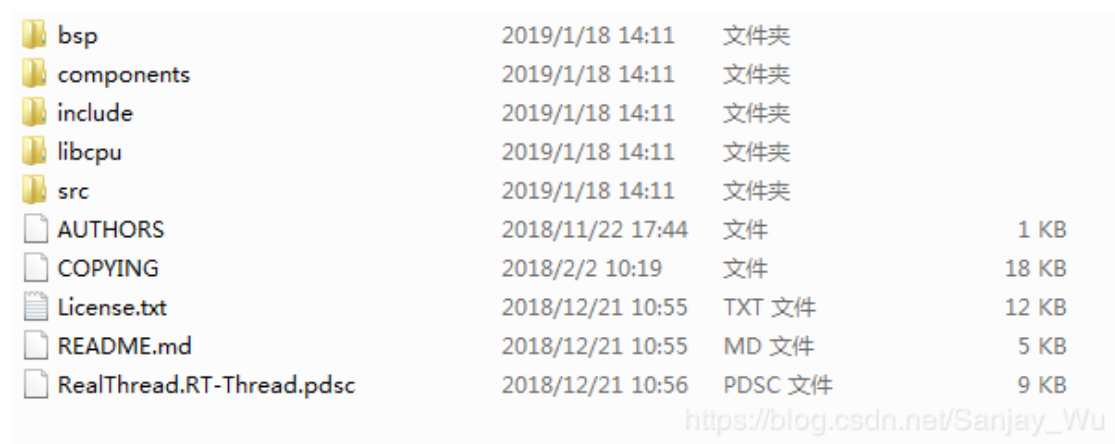


2、点击箭头下载，弹出窗口点击 OK，然后开始下载：



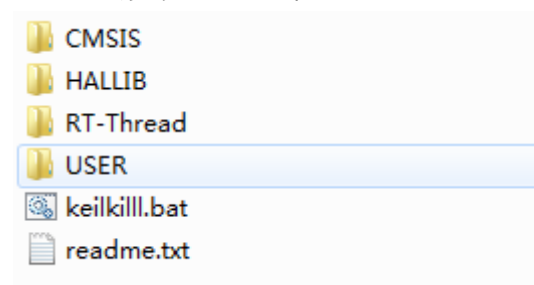
3、下载完成之后双击安装这个 pack，安装的路径和你安装 MDK5 的时候是一样的，我安装的是默认路径。

4、安装完成之后，找到你安装 MDK5 的路径，然后按这个路径找到 RT-Thread 的源码：  
C:\Keil\_v5\ARM\PACK\RealThread\RT-Thread\3.1.1:



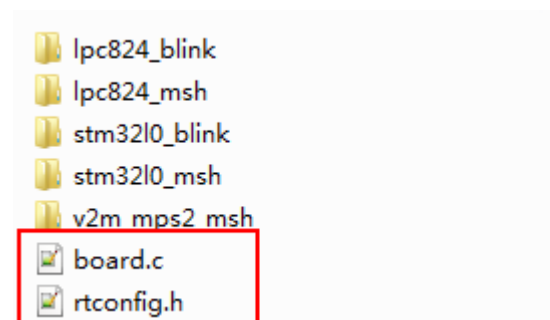
### 三、往裸机工程添加 RT-Thread 源码

1、在前面下载好的裸机工程里，再新建一个文件夹为 RT-Thread 的，然后将上面下载好的 Nano 版源码拷贝到这个文件：



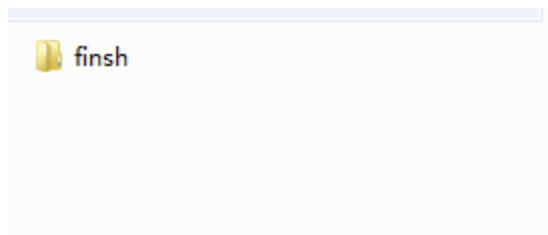
2、对于 Nano 源码各个文件内容删减：

(1) 打开 bsp，这里 RT-Thread 是放底层驱动的东西：

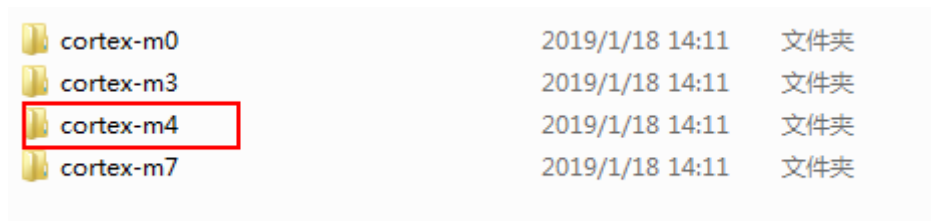


除了 board.c 和 rtconfig.h 这两个文件，其他都删除，然后再新建一个 board.h 头文件。

(2) 打开 components, RT-Thread 组件放置的地方, 只有一个 finsh, 保留它, 这个 finsh 非常好用:

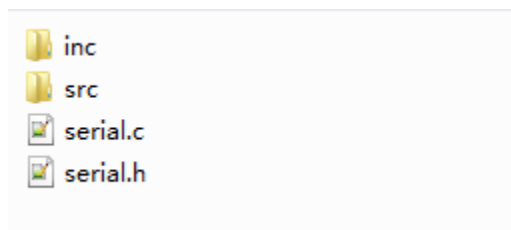


(3) 打开 libcpu —> arm, 因为用的是 STM32L4xx, 是 cortex-m4, 所以只需保留 cortex-m4 即可, 其它都删除:

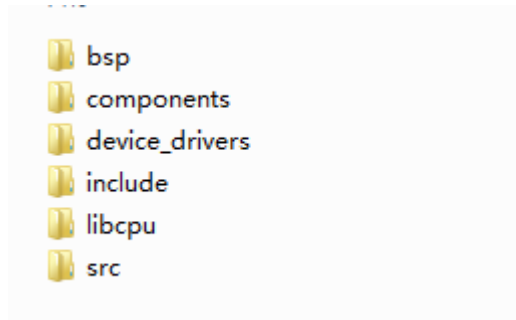


(4) 剩下的 include 和 src 文件设 RT-Thread 的头文件和内核源码, 不能删除, 保留完整。

(5) 接着, 新建一个文件夹来放设备驱动, 命名 device\_drivers。这里为什么要用 RT-Thread 设备驱动呢, 因为 RT-Thread 的 finsh 功能实现需要串口, 这里就用先只设备驱动里面的串口驱动来实现, 自己从 RT-Thread 的 master 版本中整理出来, 代码可以看工程 ([https://github.com/sanjaywu/STM32L475\\_PANDORA\\_RT-Thread\\_DEMO](https://github.com/sanjaywu/STM32L475_PANDORA_RT-Thread_DEMO)) 里面的, 这样既能实现 finsh 也能实现 rt\_kprintf。当然你也可以自己写一个串口驱动, 只不过后面一直 finsh 就会很麻烦, 读写函数都要改掉, 而且容易出错。



最终移植整理好之后, RT-Thread 的文件如下:



#### 四、修改 rtconfig.h

```
#ifndef __RTTHREAD_CFG_H__  
#define __RTTHREAD_CFG_H__
```

```
/* RT-Thread 内核部分 */
```

```

#define RT_NAME_MAX 8 //内核对象名称最大长度，大于该长度的名称
多余部分会被自动裁掉
#define RT_ALIGN_SIZE 4 //字节对齐时设定对齐的字节个数。常使用
ALIGN(RT_ALIGN_SIZE)进行字节对齐
#define RT_THREAD_PRIORITY_MAX 32 //定义系统线程优先级数；通常用
RT_THREAD_PRIORITY_MAX-1 定义空闲线程的优先级
#define RT_TICK_PER_SECOND 1000 //定义时钟节拍，为 1000 时表示 1000 个 tick
每 秒，一个 tick 为 1ms
#define RT_USING_OVERFLOW_CHECK //检查栈是否溢出，未定义则关闭
#define RT_DEBUG //定义该宏开启 debug 模式，未定义则关闭
#define RT_DEBUG_INIT 0 //开启 debug 模式时：该宏定义为 0 时表示
关闭打印组件初始化信息，定义为 1 时表示启用
#define RT_DEBUG_THREAD 0 //开启 debug 模式时：该宏定义为 0 时表示
关闭打印线程切换信息，定义为 1 时表示启用
#define RT_USING_HOOK //定义该宏表示开启钩子函数的使用，未定义
则关闭
#define IDLE_THREAD_STACK_SIZE 256 //定义了空闲线程的栈大小

/*
线程间同步与通信部分，
该部分会使用到的对象有信号量、
互斥量、事件、邮箱、消息队列、信号等
*/
#define RT_USING_SEMAPHORE //定义该宏可开启信号量的使用，未定义则关
闭
#define RT_USING_MUTEX //定义该宏可开启互斥量的使用，未定义则关
闭
#define RT_USING_EVENT //定义该宏可开启事件集的使用，未定义则关
闭
#define RT_USING_MAILBOX //定义该宏可开启邮箱的使用，未定义则关闭
#define RT_USING_MESSAGEQUEUE //定义该宏可开启消息队列的使用，未定义则
关闭
#define RT_USING_SIGNALS //定义该宏可开启信号的使用，未定义则关
闭

/* 内存管理部分 */
#define RT_USING_MEMPOOL //定义该宏可开启静态内存池的使用，未定义
则关闭
#define RT_USING_MEMHEAP //定义该宏可开启两个或以上内存堆拼接的
使用，未定义则关闭
#define RT_USING_SMALL_MEM //定义该宏可开启小内存管理算法，未定
义则关闭
// #define RT_USING_SLAB //定义该宏可开启 SLAB 内存管理算法，未定
义则关闭

```

```

#define RT_USING_HEAP //定义该宏可开启堆的使用，未定义则关闭

/* 内核设备对象 */
#define RT_USING_DEVICE //表示开启了系统设备的使用，使用设备驱动
#define RT_USING_CONSOLE //定义该宏可开启系统控制台设备的使用，未定义则关闭
#define RT_CONSOLEBUF_SIZE 128 //定义控制台设备的缓冲区大小
#define RT_CONSOLE_DEVICE_NAME "uart1" //控制台设备的名称

/* 自动初始化方式 */
#define RT_USING_COMPONENTS_INIT //定义该宏开启自动初始化机制，未定义则关闭
#define RT_USING_USER_MAIN //定义该宏 开启设置应用入口为 main 函数
#define RT_MAIN_THREAD_STACK_SIZE 2048 //定义 main 线程的栈大小

/* FinSH */
#define RT_USING_FINSH //定义该宏可开启系统 FinSH 调试工具的使用，未定义则关闭
#ifdef RT_USING_FINSH
#define FINSH_THREAD_NAME "tshell" //开启系统 FinSH 时：将该线程名称定义为 tshell
#define FINSH_USING_HISTORY //开启系统 FinSH 时：使用历史命令
#define FINSH_HISTORY_LINES 5 //开启系统 FinSH 时：对历史命令行数的定义
#define FINSH_USING_SYMTAB //开启系统 FinSH 时：定义该宏开启使用 Tab 键，未定义则关闭
#define FINSH_THREAD_PRIORITY 20 //开启系统 FinSH 时：定义该线程的优先级
#define FINSH_THREAD_STACK_SIZE 4096 //开启系统 FinSH 时：定义该线程的栈大小
#define FINSH_CMD_SIZE 80 //开启系统 FinSH 时：定义命令字符长度
#define FINSH_USING_MSH //开启系统 FinSH 时：定义该宏开启 MSH 功能
#define FINSH_USING_MSH_DEFAULT //开启系统 FinSH 时：开启 MSH 功能时，定义该宏默认使用 MSH 功能
#define FINSH_USING_MSH_ONLY //开启系统 FinSH 时：定义该宏，仅使用 MSH 功能
#endif

/* 关于 MCU */
#define STM32L475VE //定义该工程使用的 MCU 为 STM32L475VE
#define RT_HSE_VALUE 8000000 //定义时钟源频率
#define RT_USING_LED //定义该宏开启 LED 的使用
#define RT_USING_SERIAL //定义该宏开启串口的使用
#define BSP_USING_UART1 //定义该宏开启 UART1 的使用

```

```
#endif/* __RTTHREAD_CFG_H__ */
```

## 五、修改 board.c

1、添加系统时钟初始函数，这里使用 HAI 库将系统初始化为 80MHz:

```
void _Error_Handler(char *file, int line)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    while(1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

/*
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct;
    RCC_ClkInitTypeDef RCC_ClkInitStruct;

    __HAL_RCC_PWR_CLK_ENABLE();

    /* Initializes the CPU, AHB and APB busses clocks */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 1;
    RCC_OscInitStruct.PLL.PLLN = 20;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    /* Initializes the CPU, AHB and APB busses clocks */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK |
    RCC_CLOCKTYPE_PCLK1| RCC_CLOCKTYPE_PCLK2;
```

```

RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/* Configure the main internal regulator output voltage */
if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) !=
HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

```

2、修改 Tick 相关函数，初始化 SysTick，：

```

/**
 * HAL adaptation
 */
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /* Return function status */
    return HAL_OK;
}

uint32_t HAL_GetTick(void)
{
    return rt_tick_get() * 1000 / RT_TICK_PER_SECOND;
}

void HAL_SuspendTick(void)
{
    return ;
}

void HAL_ResumeTick(void)
{
    return ;
}

void HAL_Delay(__IO uint32_t Delay)

```

```

{
    return ;
}

void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}

```

### 3、修改 rt\_hw\_board\_init 函数，初始化 SysTick:

```

/**
 * This function will initial your board.
 */
void rt_hw_board_init()
{
    /* 使用 HAL 库，初始化 HAL */
    HAL_Init();

    /* 初始化系统时钟和 SysTick */
    SystemClock_Config();
    SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND);

    /* 硬件 BSP 初始化统统放在这里，比如 LED，串口，LCD 等 */
#ifdef RT_USING_LED
    led_init();
#endif

#ifdef RT_USING_SERIAL
    stm32_hw_usart_init();
#endif

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#ifdef RT_USING_CONSOLE && defined(RT_USING_DEVICE)

```



```

    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif

#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
    rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
#endif
}

```

4、增加 RT-Thread 堆空间大小，因为 finsh 需要和其他线程需要，我这里先修改为 16K，后期使用具体看 MCU 的 RAM 和实际需要调节：

```

- #if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
+ #define RT_HEAP_SIZE 4*1024
+ static uint32_t rt_heap[RT_HEAP_SIZE]; // heap default size: 16K(1024 * 4 * 4)

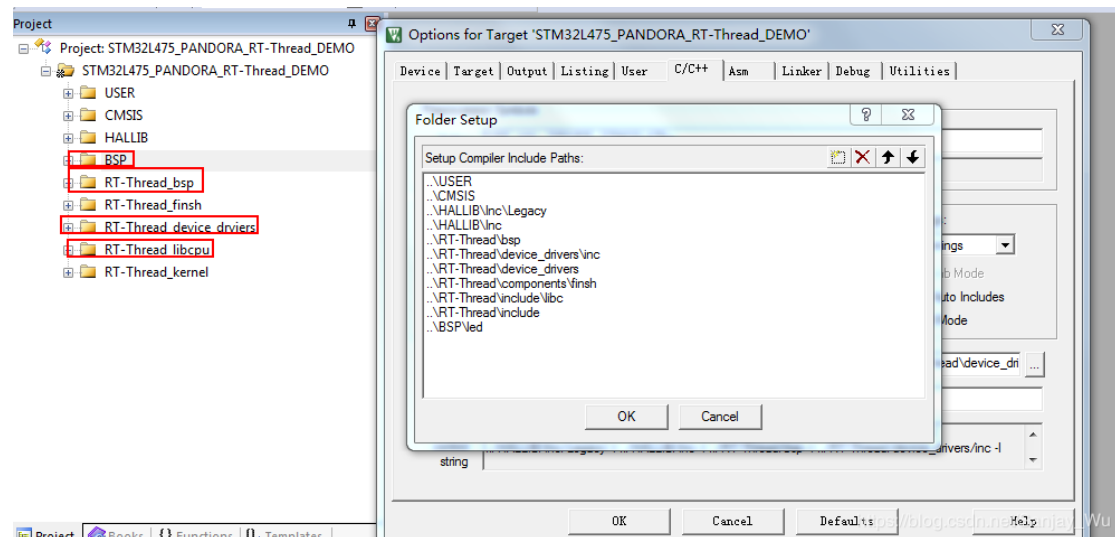
```

## 六、新建 MDK 工程

1、新建一个 BSP 文件夹用于放自己写的外设驱动：

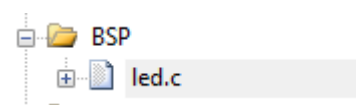
BSP	20
CMSIS	20
HALLIB	20
RT-Thread	20
USER	20
keilkill.bat	20
readme.txt	20

2、然后新建一个 MDK 工程，往工程里面加入相关文件，如下：



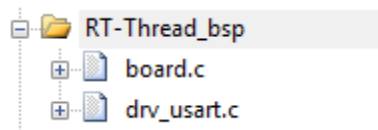
这里我们重点关注几个文件夹： BSP、RT-Thread\_bsp、RT-Thread\_device\_drivers、RT-Thread\_libcpu。

(1) BSP，放用户自己写的驱动，如 LED 驱动、LCD 驱动等：

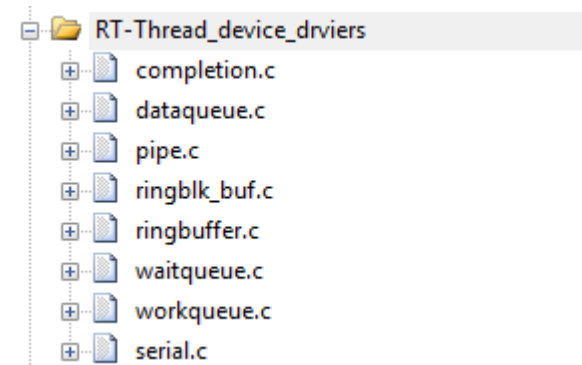


(2) RT-Thread\_bsp：放 RT-Thread 做的 BSP、board.c、board.h、rt\_config.h，放 RT-Thread

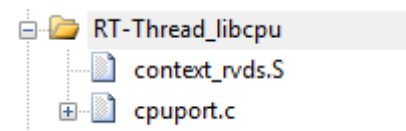
做的 BSP 可以是串口驱动等：



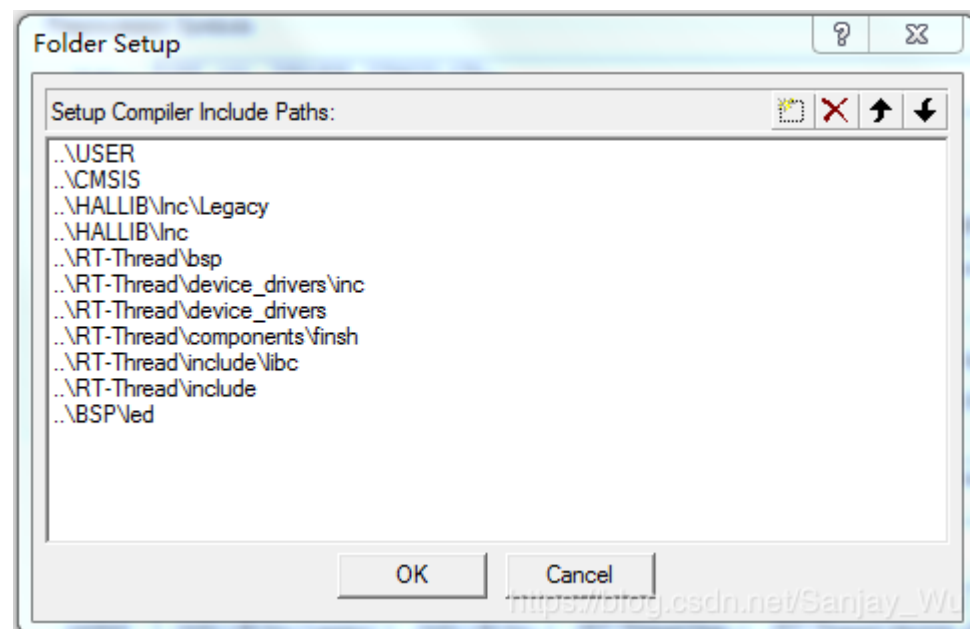
(3) RT-Thread\_device\_drivers: 放 RT-Thread 的设备驱动框架，如串口、I2C、SPI 等，我们目前只先用串口：



(4) RT-Thread\_libcpu: 放 CPU 架构，我们用的是 SMT32L4xx，因此这里是 cortex-m4，在添加 cpuport.c 和 context\_rvds.S：



3、添加相关头文件到工程：



## 七、修改 main.c

1、将 main 函数修改如下：

```
#include "main.h"
```

```

#include "board.h"
#include "rtthread.h"

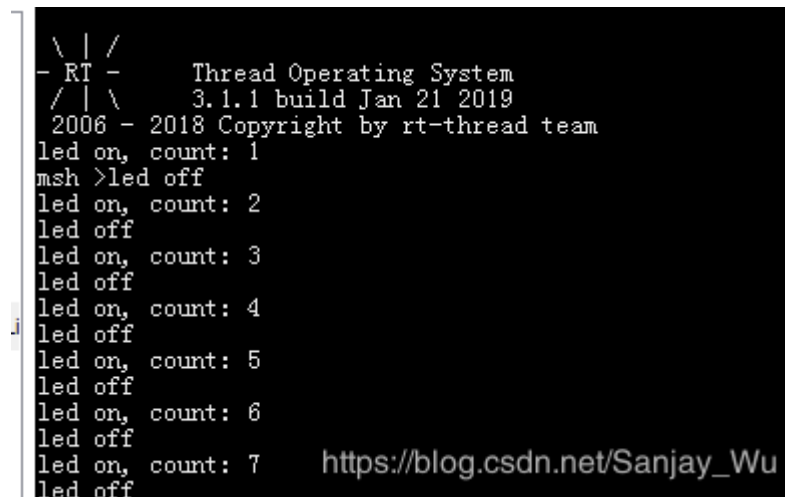
int main(void)
{
    u32 count = 1;

    while(count > 0)
    {
        LED_R(0);
        rt_kprintf("led on, count: %d\r\n", count);
        rt_thread_mdelay(500);
        LED_R(1);
        rt_thread_mdelay(500);
        rt_kprintf("led off\r\n");
        count++;
    }

    return 0;
}

```

2、保存工程，然后编译工程，下载到开发板,观察 LED 灯情况，会亮 500ms 后再灭 500ms，同时串口打印相关信息：



```

\ | /
- RT -      Thread Operating System
/ | \      3.1.1 build Jan 21 2019
2006 - 2018 Copyright by rt-thread team
led on, count: 1
msh >led off
led on, count: 2
led off
led on, count: 3
led off
led on, count: 4
led off
led on, count: 5
led off
led on, count: 6
led off
led on, count: 7
led off

```

[https://blog.csdn.net/Sanjay\\_Wu](https://blog.csdn.net/Sanjay_Wu)

3、测试 finsh 功能：

为更加直观看 finsh 相关功能，将 main 函数的串口打印代码注释掉，然后重新编译，下载到开发板：

```

#include "main.h"
#include "board.h"
#include "rtthread.h"

```

```

int main(void)
{
    u32 count = 1;

    while(count > 0)
    {
        LED_R(0);
        //rt_kprintf("led on, count: %d\r\n", count);
        rt_thread_mdelay(500);
        LED_R(1);
        rt_thread_mdelay(500);
        //rt_kprintf("led off\r\n");
        count++;
    }

    return 0;
}

```

接着打开串口，打印如下信息：

```

  \ | /
- RT -   Thread Operating System
 / | \   3.1.1 build Jan 21 2019
2006 - 2018 Copyright by rt-thread team
msh >

```

按 tab 键：

```

msh >
RT-Thread shell commands:
version
list_thread
list_sem
list_event
list_mutex
list_mailbox
list_msgqueue
list_memheap
list_mempool
list_timer
list_device
help
ps
time
free

```

输入 list\_thread：

```

msh >list_thread
thread pri  status      sp      stack size max used left tick  error
-----
tshell  20   ready  0x00000084 0x00001000    07%  0x00000006 000
tidle   31   ready  0x00000044 0x00000100    32%  0x0000001a 000
main    10   suspend 0x00000090 0x00000800    10%  0x00000013 000
msh >

```

更多 finsh 的讲解到 RT-Thread 官方看相关文档。

## 八、RT-Thread 启动流程

当你拿到一个移植好的 RT-Thread 工程的时候, 你去看 main 函数, 只能在 main 函数里面看到创建线程和启动线程的代码, 硬件初始化, 系统初始化, 启动调度器等信息都看不到。那是因为 RT-Thread 拓展了 main 函数, 在 main 函数之前把这些工作都做好了。

1、在 components.c 中有如下代码:

```
/* $Sub$$main 函数 */  
int $Sub$$main(void)  
{  
    rtthread_startup();  
    return 0;  
}
```

在这里 \$Sub\$\$main 函数仅仅调用了 rtthread\_startup() 函数, 在 components.c 的代码中找到 rtthread\_startup() 函数, 如下:

```

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();

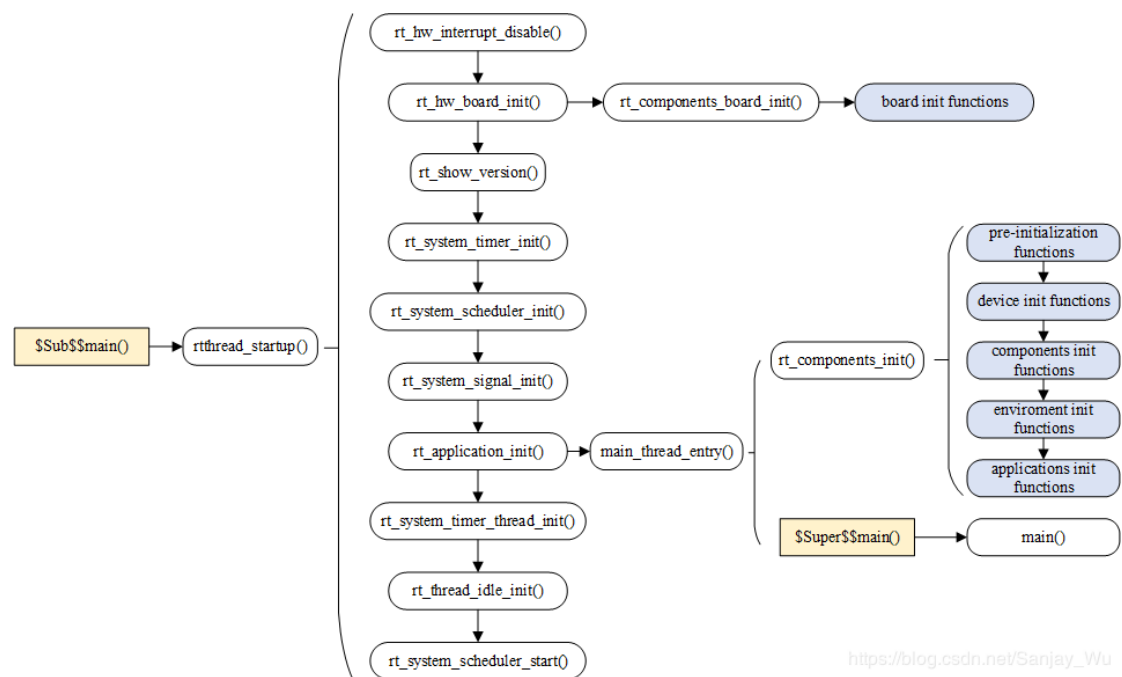
    /* start scheduler */
    rt_system_scheduler_start();

    /* never reach here */
    return 0;
} « end rtthread_startup »

```

[https://blog.csdn.net/Sanjay\\_Wu](https://blog.csdn.net/Sanjay_Wu)

启动流程如下（图片来源 RT-Thread 编程指南）：



这部分启动代码，大致可以分为四个部分：

- (1) 初始化与系统相关的硬件；
- (2) 初始化系统内核对象，例如定时器、调度器、信号；
- (3) 创建 main 线程，在 main 线程中对各类模块依次进行初始化；
- (4) 初始化定时器线程、空闲线程，并启动调度器。

rt\_hw\_board\_init() 中完成系统时钟设置，为系统提供心跳、串口初始化，将系统输入输出终端绑定到这个串口，后续系统运行信息就会从串口打印出来。

参考文献：

- 1、[野火®]《RT-Thread 内核实现与应用开发实战—基于 STM32》
- 2、RT-THREAD 编程指南



## RT-Thread

让物联网终端的开发  
变得简单、快速，芯  
片的价值得到最大化  
发挥。Apache2.0协  
议，可免费在商业产  
品中使用，不需要公  
布源码，无潜在商业  
风险。

长按二维码，关注我们