1.在stm32上启动

1.初始状态

使用rt-thread 5.02版本中bsp中stm32f407-rt-spark进行调试

该例程仿真启动是从内部flash启动，即在复位状态之后，从0x00000000映射到0x08000000处取MSP初始值，之后会自动获取下一个32位地址取出复位中断入口向量地址即0x08000004从而获取PC初始值，继而跳转执行复位中断服务程序。

打开map文件后可查到0x80000000处是中断向量表的初始地址：

```
    __Vectors_Size                          0x00000130   Number
    __Vectors                               0x08000000   Data
    __Vectors_End                           0x08000130   Data
```

MSP在*_Vectors获取到_initial_sp，并设置SP = _initial_sp* 。
在Map文件中查询 _initial_sp得知为其值为0x20000fd8，也就是栈顶指针;

```
STACK                                   0x20000bd8   Section
__initial_sp                            0x20000fd8   Data
```

在0x08000004地址获取得到的PC是中断向量表的的Reset_Handler：

```
__Vectors       DCD     __initial_sp
                DCD     Reset_Handler
                DCD     NMI_Handler
                DCD     HardFault_Handler
                DCD     MemManage_Handler
```

Reset_Handler的地址在map文件中查询得知为0x080003B5， 也就是进行debug后，执行startup_stm32f407xx.s文件的Reset_Handler函数并开始执行第一条语句，此时的寄存器状态是：

```
R12                0x00000000
R13 (SP)           0x20000FD8
R14 (LR)           0xFFFFFFFF
R15 (PC)           0x080003B4
```

SP设置为_initial_sp即0x20000fd8，PC设置为0x0800003B4(**map显示Reset_Handler地址是0x080003B5而PC设置为0x080003B4可能进行debug后还并未真正执行Reset_Handler，而进行函数调用的时候PC指针会自动+1，所以执行到Reset_Handler时候PC会被设置为0x080003B5**)

2.Reset_Handler

Reset_Handler函数内主要有SystemInit和__main进行执行；

```
Reset_Handler   PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  __main
                IMPORT  SystemInit
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

1.进入SystemInit函数：

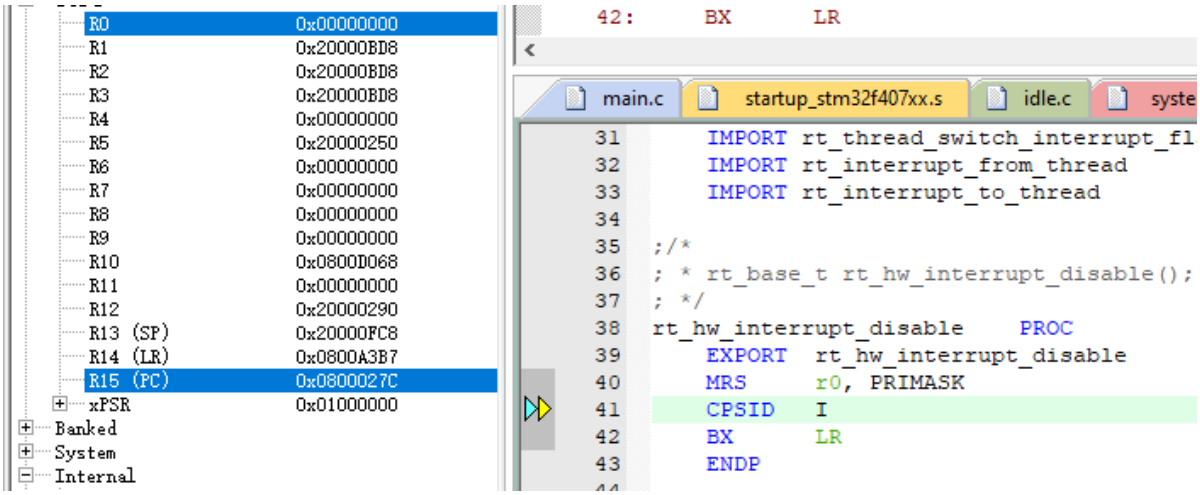
FPU以及系统时钟初始化配置

2.进入_main()：

在$Sub$$main函数打断点并全速执行，会发现在执行完SystemInit()函数后会执行$Sub$$main()函数

```
142  int $Sub$$main(void)
143  {
144      rtthread_startup();
145      return 0;
146  }
```

先执行$Sub$$main函数中的 rt_hw_interrupt_disable()函数：



在这个函数内，会关闭系统中断，使用 MRS 指令将 PRIMASK 寄存器的值保存到 r0 寄存器里，然后使用“CPSID I” 指令关闭全局中断，最后使用 BX 指令返回。上图中可以看到在执行MRS语句后R0寄存器的值为0；关闭中断之后执行rtthread_startup()：

在这个rtthread_startup()函数中会进行一系列的初始化：

```
     */
    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 /* RT_USING_SIGNALS */

    /* create init_thread */
    rt_application_init();

    /* timer thread initialization */
    rt_system_timer_thread_init();

    /* idle thread initialization */
    rt_thread_idle_init();

#ifdef RT_USING_SMP
    rt_hw_spin_lock(&_cpus_lock);
#endif /* RT_USING_SMP */

    /* start scheduler */
    rt_system_scheduler_start();

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

在rt_hw_board_init函数内会使能I_CACHE，D_CACHE,Hal库，堆栈等初始化
执行rt_show_version函数会打印系统信息；

之后的函数分别进行初始化任务定时器，初始化任务调度器，初始化系统信号，在rt_application_init()创建主线程，之后创建定时器线程，和空闲线程。在开启调度器之前系统并未开始执行线程，各创建好的线程会处于就绪状态，会在开启rt_system_scheduler_start()即系统调度之后开始调度就绪状态的线程。
RTT在MDK中使用了扩展功能 `$Sub$$` 和 `$Super$$`，在rt_application_init函数中创建了一个main主线程;

```
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_MAIN_THREAD_PRIORITY, 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_thread_stack, sizeof(main_thread_stack), RT_MAIN_THREAD_PRIORITY, 20);
    RT_ASSERT(result == RT_EOK);

    /* if not define RT_USING_HEAP, using to eliminate the warning */
    (void)result;
#endif /* RT_USING_HEAP */

    rt_thread_startup(tid);
}
```

该主线程被创建之后即被开始调度(调度器未开始，就绪状态)，在主线程的入口函数中可以看到，如果使用的是ARM汇编器的话就会执行$Super$$main()。
执行rt_system_scheduler_start();在开启调度时，系统中处于就绪状态的线程有main_thread_entry，rt_thread_timer_entry，rt_thread_idle_entry。

在完成rtthread_startup()后会跳转到main()函数进行执行：

```
18  int main(void)
19  {
20      rt_pin_mode(GPIO_LED_R, PIN_MODE_OUTPUT);
21
22      while (1)
23      {
24          rt_pin_write(GPIO_LED_R, PIN_HIGH);
25          rt_thread_mdelay(500);
26          rt_pin_write(GPIO_LED_R, PIN_LOW);
27          rt_thread_mdelay(500);
28      }
29  }
```