# rt-thread之realtouch 学习笔记 01

记录学习过程中的点点滴滴

## Table of Contents



## 1 开发环境的搭建

推荐开发使用操作系统:Window系列。目前仅尝试过WindowXP。 推荐原因：因为我要使用JLINK进行程序的烧录，而JLINK的Linux驱动是无法在盗版JLINK上使用的。

使用开发IDE：MDK ARM。 版本：4.12是不行的，要用新的，因为老版本不支持新的CPU架构。我使用的是v4.54。 说明：新的MDK的JLINK驱动会要求更新JLINK固件，SEGGER官网上的新固件会识别盗版JLINK。因此不建议更新。

使用JLINK驱动：J-Link ARM V4.40c 原因：最新的驱动程序会检查盗版JLINK从而退出程序，太老的驱动又不支持新的CPU架构。

使用交叉编译工具链：arm-2011.03-42-arm-none-eabi.exe 安装默认在C盘，使用gcc编译。

使用python版本： python-2.7.1.msi

使用scons版本：scons-2.2.0-setup.exe

说明：gcc+python+scons就可以完成编译工作，Keil是作为开发IDE来使用的，使用Keil的时候我不确信是否使用的是gcc的工具链。也不确认是否会用到python和scons。

软件安装完毕之后，下载rt-thread realtouch的代码，解压。

- bin
- programs
- realtouch
- sdcard
- README.txt

得到上面的目录结构。

我推荐使用github.com上的for windows软件来管理代码，觉得git挺好用的。

进入realtouch目录，打开project.uvproj，Keil打开了整个工程。

build，没有error，有warning。

更改Keil设置:

Target Options… -> Debug-> Use J-LINK/J-Trace Cortex Settings-> 选择SW模式

Target Options… -> Utilities->Settings->Programming Algorithm选择Add STM32F4xx Flash 1M的算法

然后连接JLINK和realtouch板，应该就能正常的烧录了。

我也试过用scons编译完了之后，用J-Link ARM把rtthread.bin文件烧录进去，主要需要选择好CPU的类型。

烧写过程中realtouch需要在外接电源上电状态。

烧写完毕，程序启动，触摸屏校准程序启动。接上串口板，串口上有输出信息。

至此，表明，基本的开发环境搭建完毕。

## 2 开始研究代码的目录结构

首先要看的是Keil的Project窗口，里面列出了为划分层次而人为建立的目录结构:

| RTThread |
| --- |
| Applications |
| Drivers |
| STM32_StdPeriph |
| ui |
| Kernel |
| CORTEX-M4 |
| Filesystem |
| DeviceDrivers |
| jpeg |

| RTThread |
| --- |
| finsh |
| Components |
| LwIP |
| pthreads |
| RTGUI |

打开各个折叠的目录看了一下,主要关注两个目录: `Application` 和 `STM32_StdPeriph`

| Application |
| --- |
| application.c |
| libc_export.c |
| setup.c |
| startup.c |

在startup.c中有着C语言的入口函数main()

| STM32_StdPeriph |
| --- |
| system_stm32f4xx.c |
| startup_stm32f4xx.s |

`startup_stm32f4x.s` 文件里面是汇编代码,是最开始的程序。

下面看一下汇编里面的关键代码

```
; Reset handler
Reset_Handler PROC
              EXPORT Reset_Handler [WEAK]
      IMPORT  SystemInit
      IMPORT  __main

              LDR     R0, =SystemInit
              BLX     R0
              LDR     R0, =__main
              BX      R0
              ENDP
```

系统在复位之后，调用了两个函数。

```
/**
  * @brief  Setup the microcontroller system
  *         Initialize the Embedded Flash Interface, the PLL and update the
  *         SystemFrequency variable.
  *         设置MCU，初始化嵌入式Flash接口，PLL和升级系统时钟频率变量
  * @param  None
  * @retval None
  */

void SystemInit(void)
{
  /* FPU settings 设置FPU */
  #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
    SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and CP11 Full Access CPACR:协处理器访问控制寄存器 */
/*
0b00：拒绝访问.任何访问尝试都会产生一个NOCP的UsageFault.
0b01：优先级访问.非优先级访问产生一个NOCP fault.
0b10：保留.访问的结果未预期
0b11：全访问
*/
  #endif

  /* Reset the RCC clock configuration to the default reset state
     复位RCC时钟配置为复位后默认状态 */

  /* Set HSION bit 内部高速时钟使能*/
  RCC->CR |= (uint32_t)0x00000001;

  /* Reset CFGR register 时钟配置寄存器 */
  RCC->CFGR = 0x00000000;

  /* Reset HSEON, CSSON and PLLON bits */
  RCC->CR &= (uint32_t)0xFEF6FFFF;

  /* Reset PLLCFGR register */
  RCC->PLLCFGR = 0x24003010;

  /* Reset HSEBYP bit */
  RCC->CR &= (uint32_t)0xFFFBFFFF;

  /* Disable all interrupts 时钟中断寄存器 */
  RCC->CIR = 0x00000000;

#ifdef DATA_IN_ExtSRAM
  SystemInit_ExtMemCtl();
#endif /* DATA_IN_ExtSRAM */

  /* Configure the System clock source, PLL Multiplier and Divider factors,
     AHB/APBx prescalers and Flash settings ----------------------------------*/
  SetSysClock();

  /* Configure the Vector Table location add offset address ------------------*/
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM */
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH */
#endif
}
```

```
int main(void)
{
    /* disable interrupt first */
    rt_hw_interrupt_disable();

    /* startup RT-Thread RTOS */
    rtthread_startup();

    return 0;
```

关注 `rtthread_startup()` 函数，就在startup.c中，这是rt-thread的入口。

```
/*This function will startup RT-Thread RTOS.*/
void rtthread_startup(void)
{
    /*init board*/
    rt_hw_board_init();

    /*show version*/
    rt_show_version();

    /*init tick*/
    rt_system_tick_init();

    /*init kernel object*/
    rt_system_object_init();

    /*init timer system*/
    rt_system_timer_init();

#if STM32_EXT_SRAM
    ext_sram_init();
    rt_system_heap_init((void*)STM32_EXT_SRAM_BEGIN,
                        (void*)STM32_EXT_SRAM_END);
#else
    rt_system_heap_init((void*)STM32_SRAM_BEGIN, (void*)STM32_SRAM_END);
#endif /* STM32_EXT_SRAM */

    /* init scheduler system */
    rt_system_scheduler_init();

    /* init all device */
    rt_device_init_all();

    /*init application*/
    rt_application_init();

    /*init timer thread*/
    rt_system_timer_thread_init();

    /*init idle thread*/
    rt_thread_idle_init();

    /*start scheduler*/
    rt_system_scheduler_start();

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

这段代码做了不少的事情，而且每一个事情的背后都隐藏着很多的知识储备。

- 初始化板子
- 打印版本信息
- 初始化tick
- 初始化内核对象
- 初始化定时器系统
- 初始化外部SRAM(在定义的情况下)
- 初始化调度系统
- 初始化所有的设备
- 初始化应用程序
- 初始化定时器线程
- 初始化空闲进程
- 开始调度

上面这些，大部分还都是与内核紧密相连的，没什么办法，一个一个的看看

### 2.1 rt_hw_board_init()

```
/*This function will initial STM32 board.*/
void rt_hw_board_init()
{
    /*NVIC Configuration 设置中断向量表*/
    NVIC_Configurationa();

    /*Configure the SysTick 配置系统tick定时器和它的中断，并且启动tick定时器*/
    SysTick_Config(SystemCoreClock/RT_TICK_PER_SECOND);

    rt_hw_usart_init();/*根据定义的宏，配置对应的串口，并且注册了UART1*/
#ifdef RT_USING_CONSOLE
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);/*设置一个设备作为控制台设备。所有的输出都会被重定向到这个设备*/
#endif

    fsmc_gpio_init();/*配置液晶屏控制器*/

    mco_config();/*配置晶振*/
}
```

### 2.2 void rt_show_version(void)

这个函数就是用 `rt_kprintf` 打印了一些信息。

### 2.3 void rt_system_tick_init(void)

这个函数在新的版本中不使用了

### 2.4 void rt_system_object_init(void)

这个函数在新的版本中不使用了

### 2.5 void rt_system_timer_init(void)

这个函数在新的版本中不使用了

### 2.6 void rt_system_scheduler_init(void)

这个函数将会初始化系统调度器

```
void rt_system_scheduler_init(void)
{
    register rt_base_t offset;

    rt_scheduler_lock_nest = 0;

    RT_DEBUG_LOG(RT_DEBUG_SCHEDULER,
      ("start scheduler: max priority 0X%02x\n", RT_THREAD_PRIORITY_MAX));

    for (offset = 0; offset < RT_THREAD_PRIORITY_MAX; offset++) {
        rt_list_init(&rt_thread_priority_table[offset]);
    }

    rt_current_priority = RT_THREAD_PRIORITY_MAX - 1;
    rt_current_thread = RT_NULL;

    /*initialize ready priority group*/
    rt_thread_ready_priority_group = 0;

#if RT_THREAD_PRIORITY_MAX > 32
    /*initialize ready table */
    rt_memset(rt_thread_ready_table, 0, sizeof(rt_thread_ready_table));
#endif

    /*initialize thread defunct*/
    rt_list_init(&rt_thread_defunct);
}
```

这里面比较主要的一个调用是 `rt_list_init()` 函数

跟踪一下代码

rtservice.h

```
rt_inline void rt_list_init(rt_list_t *l)
{
    l->next = l->prev = l;
}
/*一个指向自己的链表，还是双向链表*/
```

```
struct rt_list_node
{
    struct rt_list_node *next;
    struct rt_lsit_node *prev;
};
typedef struct rt_list_node rt_list_t;
/*定义了一个链表中的一个节点*/
```

目前，我们接触到了一个数据结构，就是双向链表，而且是跟调度相关的。

暂时能想到的关联就是用这个双向链表管理所有的进程。

### 2.7 rt_device_init_all() rt_application_init()

`rt_device_init_all()` 和 `rt_application_init()` 暂时先不分析。

device里面出现了object，而且是个抽象的设备层，这个放到后面再研究。

application则是创建线程，也先不管。

### 2.8 rt_system_timer_thread_init()

这个函数将会初始化所有的系统定时器线程

```
void rt_system_timer_thread_init(void)
{
#ifdef RT_USING_TIMER_SOFT
    rt_list_init(&rt_soft_timer_list);

    /*start software timer thread*/
    rt_thread_init(&timer_thread,
             "timer",
             rt_thread_timer_entry, RT_NULL,
             &timer_thread_stack =[0]=, sizeof(timer_thread_stack),
             RT_TIMER_THREAD_PRIO, 10);

    /*startup*/
    rt_thread_startup(&timer_thread);
#endif
}
#END_SRC
```

**定义了一个软定时器双向链表** `=rt_soft_timer_list=`

**又开了一个线程** `=rt_thread_timer_entry=`

`=rt_thread_startup=` **是用来启动一个线程，并且把它放到系统**`ready queue`**里面去。**

**对于** `=rt_thread_init=` **解释是这个函数用来初始化一个线程，通常它被用来初始化一个静态**`thread object`**。**

**并且调用了** `=rt_object_init=` **和** `=_rt_thread_init=` **两个函数**

**我对**`object`**暂时还不想碰，先深究以下** `=_rt_thread_init=` **函数**

```
#+BEGIN_SRC c
static rt_err_t _rt_thread_init(struct rt_thread *thread,
                                const char *name,
                                void (*entry)(void *parameter),
                                void *parameter,
```

```
                                void *stack_start,
                                rt_uint32_t stack_size,
                                rt_uint8_t priority,
                                rt_uint32_t tick)
{
    /*init thread list*/
    rt_list_init(&(thread->tlist));

    thread->entry = (void*)entry;
    thread->parameter = parameter;

    /*stack init*/
    thread->stack_addr = stack_start;
    thread->stack_size = stack_size;

    /*init thread stack*/
    rt_memset(thread->stack_addr, '#', thread->stack_size);
    thread->sp = (void*)rt_hw_stack_init(thread->entry, thread->parameter,
                                         (void*)((char*)thread->stack_addr
                                        +thread->stack_size - 4),
                                         (void*)rt_thread_exit);

    /*priority init*/
    RT_ASSERT(priority < RT_THREAD_PRIORITY_MAX);
    thread->init_priority = priority;
    thread->current_priority = priority;

    /* tick init */
    thread->init_ick = tick;
    thread->remaining_tick = tick;

    /*error and flags*/
    thread->error = RT_EOK;
    thread->stat = RT_THREAD_INIT;

    /*initialize cleanup function and user data*/
    thread->cleanup = 0;
    thread->user_data = 0;

    /*init thread timer*/
    rt_timer_init(&(thread->thread_tiemr),
                  thread->name,
                  rt_thread_timeout,
                  thread,
                  0,
                  RT_TIMER_FLAG_ONE_SHOT);

    return RT_EOK;
}
```

这里面，初始化了一个双向链表，填充了thread结构体，初始化了一个定时器。

下面可以看一下thread结构体

```
struct rt_thread
{
    /*rt object*/
    char name[RT_NAME_MAX]; /* 线程的名字 */
    rt_uint8_t type;        /* 对象的类型 */
    rt_uint8_t flags;       /* 线程的标志 */

#ifdef RT_USING_MODULE
    void *module_id;        /* 应用模块的id */
#endif

    rt_list_t list;         /* 对象列表 */
    rt_list_t tlist;        /* 线程列表 */

    /* 栈指针和入口 */
    void    *sp;            /*栈指针*/
    void    *entry;         /*入口*/
    void    *parameter;     /*参数*/
    void    *stack_addr;    /*栈地址*/
    rt_uint16_t stack_size; /*栈大小*/

    /*error code*/
    rt_err_t error;         /*错误码*/

    rt_uint8_t stat;        /*线程状态*/

    /*属性*/
    rt_uint8_t current_priority;    /*当前优先级*/
    rt_uint8_t init_priority;       /*初始化优先级*/
#if RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t number;
    rt_uint8_t high_mask;
#endif
    rt_uint32_t number_mask;

#if defined(RT_USING_EVENT)
    /*thread event*/
    rt_uint32_t event_set;
    rt_uint8_t  event_info;
#endif

    rt_ubase_t init_tick;  /*线程的初始化的tick*/
    rt_ubase_t remaining_tick; /*剩下的tick*/

    struct rt_timer thread_timer; /*内建线程定时器*/

    void (*cleanup)(struct rt_thread *tid); /* 当线程退出的时候的清理函数 */

    rt_uint32_t user_data; /* 这个线程的私有用户数据 */
};
typedef struct rt_thread *rt_thread_t;
```

### 2.9 rt_thread_idle_init()

初始化空闲线程

启动线程 rt_thread_idle_entry,然后执行了 rt_thread_idle_excute() 函数

### 2.10 rt_system_scheduler_start()

开始调度

scheduler.c

这个函数将开始执行调度，它将会选择一个具有最高优先级的线程，然后交换到该线程

```
void rt_system_scheduler_start(void)
{
    register struct rt_thread *to_thread;
    register rt_ubase_t highest_ready_priority;

#if RT_THREAD_PRIORITY_MAX == 8
    highest_ready_priority = rt_lowest_bitmap[rt_thread_ready_priority_group];
#endif
    register rt_ubase_t number;
    /*find out the highest priority task*/
    if (rt_thread_ready_priority_group & 0xff) {
        number = rt_lowest_bitmap[rt_thread_ready_priority_group & 0xff];
    } else if (rt_thread_ready_priority_group & 0xff00) {
        number = rt_lowest_bitmap[(rt_thread_ready_priority_group >> 8) & 0xff + 8;
    } else if (rt_thread_ready_priority_group & 0xff0000) {
       number = rt_lowest_bitmap[(rt_thread_ready_priority_group >> 16) & 0xff]+ 16;
    } else {
       number = rt_lowest_bitmap[(rt_thread_ready_priority_group >> 24) & 0xff] + 24;
    }

#if RT_THREAD_PRIORITY_MAX > 32
    highest_ready_priority = (number << 3) + rt_lowest_bitmap[rt_thread_ready_table[number]];
#else
    highest_ready_priority = number;
#endif
#endif

    /* get switch to thread */
    to_thread = rt_list_entry(rt_thread_priority_table[highest_ready_priority].next,
                              struct rt_thread, tlist);

    rt_current_thread = to_thread;

    /* switch to new thread */
    rt_hw_context_switch_to((rt_uint32_t)&to_thread->sp);

    /* never come back */
}
```

`rt_list_entry` 做的事情是通过地址偏移量计算出来某个结构体的首地址

`rt_hw_context_switch_to` 是个汇编的过程 `context_xxx.S` 里面

r0用来存放to参数，这个函数用来处理第一个线程交换。没太看懂这个汇编过程的主要用意，主要还是对线程的上下文交换需要作哪些事情不是很清楚。

那么在此时，优先级别最高的线程是哪个呢？这是个问题！

---

## 3 优先级探索

根据 `void rt_system_scheduler_start(void)` 函数中的内容，我们可以看到，跟优先级有关系的有下面这么几个变量

- `register rt_ubase_t highest_ready_priority`
- `register rt_ubase_t number`
- `rt_thread_ready_priority_group`

打印一下信息可以得到

```
rt_thread_ready_priority_group=80000400
number=a
highest_ready_priority=a
```

因此，我们要研究一下，它们是怎么被赋值的。

先关注一下 `rt_thread_ready_priority_group` 这个变量，它在 `rt_system_scheduler_init` 函数里面已经被初始化为0了。

在这之间，还有三个函数需要查看 `rt_application_init()`, `rt_system_timer_thread_init()`, `rt_thread_idle_init()`, 这三个里面都建立了线程。

```
rt_thread_create("init",                        //线程的名字
                 rt_init_thread_entry,          //线程的入口函数
                 RT_NULL,                       //入口函数的参数
                 2048,                          //线程栈的大小
                 RT_THREAD_PRIORITY_MAX/3,      //线程的优先级
                 20);                           //同样优先级情况下的分配的时间片
```

```
rt_thread_init(&timer_thread,                   //静态线程对象
               "tiemr",                         //线程名字
               rt_thread_timer_entry,           //线程函数入口
               RT_NULL,                         //线程函数参数
               &timer_thread_stack[0],          //线程栈开始地址
               sizeof(timer_thread_stack),      //线程栈大小
               RT_TIMER_THREAD_PRIO,            //线程优先级
               10)                              //时间片
```

```
rt_thread_init(&idle,
               "tidle",
               rt_thread_idle_entry,
               RT_NULL,
               &rt_thread_stack[0],
               sizeof(rt_thread_stack),
               RT_THREAD_PRIORITY_MAX-1,
               32);
```

那么我们打印一下init和create的线程信息，看一下优先级是怎样的

```
[create]thread name:init priority:a
[init]thread name:tidle priority:1f
[init]thread name:tshell priority:14
[init]thread name:erx priority:e
[init]thread name:etx priority:e
[create]thread name:tcpip priority:c
[create]thread name:rtgui priority:f
```

```
[create]thread name:touch priority:e
[init]thread name:mmcsd_detect priority:f
[create]thread name:key priority:e
[create]thread name:app_mgr priority:14
[create]thread name:cali priority:14
```

而且是，在 `rt_system_scheduler_start()` 的时候，只有init和tidle两个线程被注册了。

还有一个timer的线程不知道为什么没有打印出来，也是通过 `rt_thread_init` 注册的。

现在，先看一下这三个线程所注册的函数

init-> `rt_init_thread_entry`

timer-> `rt_thread_timer_entry`

tidle-> `rt_thread_idle_entry`

`rt_init_thread_entry` 是最麻烦的，先来看一下它

```
void rt_init_thread_entry(void *parameter)
{
#ifdef RT_USING_COMPONENTS_INIT
    /* initialization RT-Thread Components */
    rt_components_init();
#endif
    rt_platform_init();
    /* Filesystem Initialization */
#ifdef RT_USING_DFS
    /*mount sd card fat partition 1 as root directory */
    if (dfs_mount("sd0", "/", "elm", 0, 0)== 0) {
        rt_kprintf("File System initialized!\n");
    } else  {
        rt_kprintf("File System initialzation failed!\n");
    }
#endif
#ifdef RT_USING_RTGUI
    realtouch_ui_init();
#endif
}
```

调用了三个函数做了不少事情。

看一下 `rt_thread_timer_entry`

```
static void rt_thread_timer_entry(void *parameter)
{
    rt_tick_t next_timeout;

    while (1) {
        next_timeout = rt_timer_list_next_timeout(&rt_soft_timer_list);
        if (next_timeout == RT_TICK_MAX) {
            rt_thread_suspend(rt_thread_self());
            rt_schedule();
        } else {
            rt_thread_delay(next_timeout);
        }

        /*lock scheduler*/
        rt_enter_critical();
        /*check software timer*/
        rt_soft_timer_check();
        /*unlock scheduler*/
        rt_exit_critical();
    }
}
```

`rt_thread_idle_entry`

```
static void rt_thread_idle_entry(void *parameter)
{
    while (1) {
        #define RT_USING_HOOK
        if (rt_thread_idle_hook != RT_NULL) {
            rt_thread_idle_hook();
        }
        #endif
        rt_thread_idle_excute();
    }
}
```

`rt_thread_idle_excute()` 函数是比较复杂的

这里面出现了两个while(1)的死循环。

timer主要做的事情是检查timeout，将超时的线程挂起，然后重新调度。

idle主要是把不需要的线程从链表里删除

这个具体执行调度的地方我还是不是很清楚，没有说找到特别明确的代码片段。

---

# 4 发现DEBUG

rtdebug.h里面有好多的宏开关，我们应该尝试打开一下，看一下输出信息。

```
 \ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Sep  6 2012
 2006 - 2012 Copyright by rt-thread team
mem init, heap begin address 0x60000000, size 524264
start scheduler: max priority 0x20
malloc size 136
thread (NULL) take sem:heap, which value is: 1
thread (NULL) releases sem:heap, which value is: 0
allocate memory at 0x6000000c, size: 148
malloc size 2048
thread (NULL) take sem:heap, which value is: 1
thread (NULL) releases sem:heap, which value is: 0
allocate memory at 0x600000a0, size: 2060
[create]thread name:init priority:a
```

```
startup a thread:init with priority:10
thread resume:  init
insert thread[init], the priority: 10
[init]thread name:tidle priority:1f
startup a thread:tidle with priority:31
thread resume:  tidle
insert thread[tidle], the priority: 31
rt_thread_ready_priority_group=80000400
number=a
highest_ready_priority=a
irq coming..., irq nest:0
timer check enter
timer check leave
irq leave, irq nest:1
irq coming..., irq nest:0
timer check enter
timer check leave
irq leave, irq nest:1
irq coming..., irq nest:0
timer check enter
timer check leave
irq leave, irq nest:1
irq coming..., irq nest:0
timer check enter
timer check leave
irq leave, irq nest:1
```

debug全开之后的信息如上所示，以后的信息就更加混乱了，试图从上面的信息里面分析一些东西出来吧。

在scheduler.c中的每个函数入口和出口的地方都加入一个打印信息，得到了另外的一组调试数据。

```
 \ | /
- RT -     Thread Operating System
 / | \     1.1.0 build Sep  6 2012
 2006 - 2012 Copyright by rt-thread team
[scheduler] enter rt_system_scheduler_init
start scheduler: max priority 0x20
[scheduler] leave rt_system_scheduler_init
[create]thread name:init priority:a
[scheduler] enter rt_schedule_insert_thread
insert thread[init], the priority: 10
[scheduler] leave rt_schedule_insert_thread
[init]thread name:tidle priority:1f
[scheduler] enter rt_schedule_insert_thread
insert thread[tidle], the priority: 31
[scheduler] leave rt_schedule_insert_thread
[scheduler] enter rt_system_scheduler_start
rt_thread_ready_priority_group=80000400
number=a
highest_ready_priority=a
[scheduler] switch-to init
[init]thread name:tshell priority:14
[scheduler] enter rt_schedule_insert_thread
insert thread[tshell], the priority: 20
[scheduler] leave rt_schedule_insert_thread
[scheduler] enter rt_schedule
[scheduler] leave rt_schedule
[scheduler] enter rt_enter_critical
[scheduler] leave rt_enter_critical
[scheduler] enter rt_exit_critical
[scheduler] enter rt_schedule
[scheduler] leave rt_schedule
[scheduler] leave rt_exit_critical
[scheduler] enter rt_enter_critical
[scheduler] leave rt_enter_critical
[scheduler] enter rt_exit_critical
[scheduler] enter rt_schedule
[scheduler] leave rt_schedule
[scheduler] leave rt_exit_critical
[create]thread name:rtgui priority:f
[scheduler] enter rt_schedule_insert_thread
insert thread[rtgui], the priority: 15
[scheduler] leave rt_schedule_insert_thread
[scheduler] enter rt_schedule
[scheduler] leave rt_schedule
[scheduler] enter rt_enter_critical
[scheduler] leave rt_enter_critical
[scheduler] enter rt_exit_critical
[scheduler] enter rt_schedule
[scheduler] leave rt_schedule
[scheduler] leave rt_exit_critical
[scheduler] enter rt_enter_critical
[scheduler] leave rt_enter_critical
```

研究一下，在create init线程的时候，为什么会进入 `rt_system_scheduler_init` 函数

```
rt_system_timer_thread_init();
```

为了看的更加详细些，建议在thread.c里面的每个函数里面也都加入调试信息->

```
rt_thread_init();
->enter rt_thread_create();
->enter _rt_thread_init();
->leave _rt_thread_init();
->leave rt_thread_create();
rt_thread_startup();
->enter rt_thread_startup();
->print "startup a thread: init with priority:10"
```

然后代码改变了线程的状态

```
thread->stat = RT_THREAD_SUSPEND;

rt_thread_resume(thread);
```

从而导致了

```
->enter rt_thread_resume();
->从suspend list中移除，然后添加到schedule ready list中去
->enter rt_schedule_insert_thread();
->leave rt_schedule_insert_thread();
->leave rt_thread_resume();
```

```
->enter rt_thread_self();
因为还没有执行过rt_system_scheduler_start();所以这里不执行rt_schedule();
->leave rt_thread_self();

->leave rt_thread_startup();
```

在用同样的流程创建完idle线程之后才开始调用 `rt_system_scheduler_start`

在这个函数里面寻找优先级最高，也就是数最小的那个线程，然后把 `rt_current_thread` 赋值。

执行 `rt_hw_context_switch_to` 函数，进行切换

`rt_hw_context_switch_to` 是个汇编代码，在 `context_rvds.S` 中

里面主要用到了

- `rt_interrupt_to_thread`
- `rt_interrupt_from_thread`
- `rt_thread_switch_interrupt_flag`
- `NVIC_SYSPRI2`
- `NVIC_PENDSV_PRI`
- `NVIC_INT_CTRL`
- `NVIC_PENDSVSET`

最后触发了 `context_rvds.S` 里面的 `PendSV_Handler` 过程

然后，一开始注册的init线程才开始工作，在此之前只有timer线程和idle线程工作。

现在，简单的归纳一下：

一开始从 `rt_applicaton_init()` 中注册了init线程，并没有启动。

因为这个时候init线程的$rt_{currentthread}$还是RTNULL

然后注册了timer线程和idle线程，也没有启动

一直等到 `rt_system=scheduler_start()` 函数执行，才开始唤醒线程。

由于没有定义 `RT_USING_TIMER_SOFT` 所以timer线程没有被注册！

## 5 定时器 SysTick

现在，回过头来看一下 `_rt_thread_init()` 这个函数

最后有一段

```
/*init thread timer -> timer.c*/
rt_timer_init(&(thread->thread_timer), //静态定时器对象
              thread->name,            //定时器的名字
              rt_thread_timeout,       //超时函数
              thread,                  //超时函数的参数
              0,                       //定时器的tick
              RT_TIMER_FLAG_ONE_SHOT); //定时器的标志位
```

找了很多文件，发现没有一个集中的说明问题的……

《Cortex-M4 Technical Reference Manual》

4.2 Register summary

System control reigsters

| Address | Name | Type | Reset | Description |
|---|---|---|---|---|
| 0xE000E010 | STCSR | RW | 0x00000000 | SysTick Control and Status Register |
| 0xE000E014 | STRVR | RW | Unknown | SysTick Reload Value Register |
| 0xE000E018 | STCVR | RW clear | Unkndow | SysTick Current Value Register |
| 0xE000E01C | STCR | RO | STCALIB | SysTick Calibration Value Register |

《STM32F4xxx Cortex-M4 programming manual》

4.5 SysTick timer (STK)

The processor has a 24-bit system timer, SysTick, that counts down from the reload value to zero, reloads (wraps to) the value in the $STK_{LOAD}$ register on the next clock edge, then counts down on subsequent clocks.

When the processor is halted for debugging the counter does not decrement.

| **$STK_{CTRL}$ SysTick control and status register** | |
|---|---|
| COUNTFLAG | Returns 1 if timer counted to 0 since last time this was read |
| CLKSOURCE | Clock source selection<br>0: AHB/8<br>1: Processor clock (AHB) |
| TICKINT | SysTick exception request enable<br>0: Counting down to zero does not assert the SysTick excepton request<br>1: Counting down to zero to asserts the SysTick exception request<br>Note:Software can use COUNTFLAG to determine if SysTick has ever counted to zero |
| ENABLE | Counter Enable<br>0: Counter disabled<br>1: Counter enabled |

Enables the counter. When ENABLE is set to 1, the counter loads the RELOAD value from the LOAD register and then counts down.

On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT.

It then loads the RELOAD value again, and begins counting.

| $STK_{LOAD}$ | SysTick reload value register |
|---|---|
| RELOAD | RELOAD value<br>The LOAD register specifies the start value to load into the $STK_{VAL}$ register when the counter is enabled<br>and when it reaches 0. |

Calculating the RELOAD value

The RELOAD value can be any value in the range 0x00000001-0x00FFFFFF.

A start value of 0 is possible, but has no effect because the SysTick exception request and COUNTFLAG are activated when count from 1 to 0.

The RELOAD value is calculated according to its use:

- To generato a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. For example, if the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.
- To deliver a single SysTick interrupt after a delay of N processor clock cycles, use a RELOAD of value N. For example, if a SysTick interrupt is required after 100 clock pulses, set RELOAD to 99.

| $STK_{VAL}$ | SysTick current value regiser |
|---|---|
| CURRENT | Current counter value |

The VAL register contains the current value of the SysTick counter.

Reads return the current value of the SysTick counter.

A write of any value clears the field to 0, and also clears the COUNTFLAG bit in the $STK_{CTRLregister}$ to 0.

| $STK_{CALIB}$ | SysTick calibration value register |
|---|---|
| NOREF | NOREF flag.<br>Reads as zero. Indicates that a separate reference clock is provided. The frequency of this clock is HCLK/8 |
| SKEW | SKEW falg.<br>Indicates whether the TENMS value is exact.<br>Reads as one. Calibration value for the 1 ms inexact timing is not knownbecause TENMS is not known.<br>This can affect the suitablity of SysTick as a software real time clock. |
| TENMS | Calibration value.<br>Indicates the calibration value when the SysTick counter run on HCLK max/8 as external clock.<br>The value is product denpendent, please refer to the Product Reference Manual, SysTick Calibration Value section.<br>When HCLK is programmed at the maximum frequency, the SysTick period is 1ms.<br>If calibration information is not known,<br>calculate the calibration value required from the frequency of the processor clock or external clock. |

SysTick design hits and tips

The SysTick counter runs on the processor clock.

If this clock signal is stopped for low power mode, the SysTick couner stops.

Ensure software uses aligned word accesses to access the SysTick registers.

The SysTick counter relaod and current value are undefined at reset, the correct initializaton sequence for the SysTick counter is :

1. Program reload value.
2. Clear current value.
3. Program Countrol and Status register.

看完了数据手册，对应的看一下代码 `core_cm4.h`

```
/* System Tick Configration
   This function initialises the system tick timer and its interrupt and start the system tick tiemr.
   Counter is in free running mode to generate periodical interrupts.
 */
static __INLINE uint32_t SysTick_Confit(uint32_t ticks)
{
    if (ticks > SysTick_LOAD_RELOAD_Msk) { /* Reload value impossible */
        return(1);
    }
    SysTick->LOAD = (tick & SysTick_LOAD_RELOAD_Msk) - 1; /*set reload register*/
    NVIC_SetPriority(SysTick_IRQn, (1<<__NVIC_PRIO_BITS)-1);/*set Priority for Coterx-M0 System Interrupts*/
    SysTick->VAL = 0;
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk   |
                    SysTick_CTRL_ENABLE_Msk; /*Enable SysTick IRQ and SysTick Timer*/
}
```

```
/*SysTick Reload Register Definitions*/
#define SysTick_LOAD_RELOAD_Pos 0                                          /*SysTick LOAD: RELOAD Position*/
#define SysTick_LOAD_RELOAD_Msk (0xFFFFFFUL<<SysTick_LOAD_RELOAD_Pos) /*SysTick LOAD: RELOAD Mask*/
```

基本上做的事情是：

先赋值，设置中断，清零，设置控制位

这个 `SysTick_IRQn` 在stm32f4xx.h里面

typedef enum IRQn-> `SysTick_IRQn` = -1 *Cortex-M4 System Tick Interrupt*

然后让我们跟踪一下这个中断吧

`startup_stm32f4xx.s` 里面写到

```
DCD SysTick_Handler

SysTick_Handler PROC
                EXPORT SysTick_Handler [WEAK]
                B       .
                ENDP
```

实现在board.c里面

```
void SytTick_Handler(void)
{
    rt_interrupt_enter();
    rt_tick_increase();
    rt_interrupt_leave();
}
```

```
void rt_tick_increase(void)
{
    struct rt_thread *thread; //定义一个线程结构体

    ++rt_tick; //这是一个静态全局变量，应该是用来存tick计数的

    thread = tr_thread_self(); //获取当前在运行的线程

    -- thread->remainning_tick; //线程剩余时间减少
    if (thread->remaining_tick == 0) { //看看是不是减少到零了
        thread->remaining_tick = thread->init_tick; //准备重新开始
        rt_thread_yield(); //主动让出CPU
    }
    rt_timer_check(); //timer.c 检查timer list，如果一个超时事件发生，相关的超时函数将会被调用
                      // 这个函数应该在操作系统时间中断内被调用
}
```

timer.c

```
void rt_timer_check(void)
{
    struct rt_timer *t;
    rt_tick_t current_tick;
    register rt_base_t level;

    current_tick = rt_tick_get();

    /*disable interrupt*/
    level = rt_hw_interrupt_disable();

    while(!rt_list_isempty(&rt_timer_list)) {
        t = rt_list_entry(rt_timer_list.next, struct rt_timer, list);

        /*
            It supposes that the new tick shall less than the half duration ofo tick max.
        */
        if ((current_tick - t->timeout_tick) < RT_TICK_MAX/2) {
            RT_OBJECT_HOOK_CALL(rt_timer_timeout_hook, (t));

            /*remove timer from timer list firstly*/
            rt_list_remove(&(t->list));

            /*call timeout fucntion */
            t->timeout_func(t->parameter);

            /*re-get tick*/
            current_tick = rt_tick_get();

            if ((t->parent.flag & RT_TIMER_FLAG_PERIODIC) &&
                (t->parent.flag & RT_TIMER_FLAG_ACTIVATED)) {
                /*start it*/
                t->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;
                rt_timer_start(t);
            } else {
                /*stop timer*/
                t->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;
            }
        } else
            break;
    }

    /* enable interrupt */
    rt_hw_interrupt_enable(level);
}
```

牵一发而动全身，现在又从硬件寄存器的控制转移到该死的链表操作里面去了。

现在模糊的理解了定时器是怎么集成到操作系统里面去的。

在每个线程被初始化的时候，都会产生一个以线程名字命名的定时器。

而系统的SysTick定时器是已经在运行中的了，每隔一定的时间就会产生硬件中断，然后执行中断服务程序，遍历所有的定时器链表，修改当前线程的timeout值。

而我们知道，线程的调度是通过优先级和时间片共同的作用来的。因此……是不是我们可以先看一下优先级所起到的作用。

# 6 线程 优先级

线程的优先级被定义在rtdef.h中

```
/*thread state definitions*/
#define RT_THREAD_INIT          0x00                // 初始状态 线程不参与调度
#define RT_THREAD_READY         0x01                // 就绪状态
#define RT_THREAD_SUSPEND       0x02                // 挂起状态 线程不参与调度
#define RT_THREAD_RUNNING       0x03                // 运行状态
#define RT_THREAD_BLOCK         RT_THREAD_SUSPEND   // 阻塞状态 线程不参与调度
#define RT_THREAD_CLOSE         0x04                // 关闭状态 线程不参与调度
```

## 6.1 RT_THREAD_INIT

只有在 `_rt_thread_init` 函数里面将thread->stat赋值为 `RT_THREAD_INIT`

## 6.2 RT_THREAD_READY

在 `rt_schedule_insert_thread` 函数中改变状态

## 6.3 RT_THREAD_SUSPEND

`rt_thread_startup` 函数将挂起线程

`rt_thread_suspend` 函数会挂起线程

### 6.4 RT_THREAD_RUNNING

都没搜到哪里用到了

### 6.5 RT_THREAD_BLOCK

也没有地方用到

### 6.6 RT_THREAD_CLOSE

`rt_thread_exit` 函数会关闭线程

`rt_thread_detach` 函数会关闭线程

`rt_thread_delete` 函数会关闭线程

### 6.7 kernel如何找到当前优先级最高的线程的

blog.csdn.net/prife/articale/details/7077120 文章中 Prife写的非常的详尽和清楚明白

主要是进行了位图调度算法分析

先说白了就是一个哈希表+双向链表的结构，不过这个哈希表是个位图，也就是个二位数组，映射起来很方便。

rt-thread的调度算法是基于优先级和时间片的，时间片部分前面大概分析了以下，不过还不是很清晰。

优先级这部分，通过上面的文章介绍，倒是很清晰了。很感谢Prife开放自己的思想。

这个问题可以这么描述：

现在有一堆优先级不同的线程，优先级是一个数，这个数越小表示这个线程的优先级越高，现在要用一种简单的方法来找出来这一堆线程里面谁的优先级最高。

现在先要定义一个数组来表示优先级

```
rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];
```

这其中 `RT_THREAD_PRIORITY_MAX` = 256

我们的目的在于找出当前最高的优先级是多少来，然后根据 **`rt_thread_priority_table[最高优先级]`** 来找到对应的进程。

实际的代码是

```
to_thread = rt_list_entry(rt_thread_priority_table[highest_ready_priority].next, xxxx);
```

因此，如何确定 `hightest_ready_priorit` 就关键了

这里存在这么一个问题，如果使用for循环是肯定可以找到的，不过时间消耗没有办法确定，我们需要的是一个时间消耗恒定的算法。

因此，就想，不就是256个优先级么，当前这个优先级上要不有ready进程，要不没有。

因此，可以用一个256bits的数来表示，bit为1就表示这一个级别的优先级上有ready进程，否则，就是空的。

从C语言的角度来看，定义一个数组来表示比较合适

```
rt_uint8_t rt_thread_ready_table[32];
```

为什么是uint8也就是8位的呢，因为，8位的的确是比较好算，好处理。

这样相当于给256个bit分了32组，每组8个优先级。

现在，我们就单独的只看8个优先级，这8个bit。

8个bit可以形成一个byte。它的取值范围是0x00到0xFF。

为什么这么说，因为，每当新建一个进程的时候，都会赋予它一个优先级，对应的，这个byte对应的位就要置1。

比如，现在有A,B,C三个进程，优先级分别为1，3，4，则，这个byte=0b00011010=26=0x1a。

这也是为什么我们用8位来解释问题，因为32位所代表的数实在是太大了，$2^{32}$=4G，没法表示～(这里说的是 `rt_thread_ready_priority_group` 这个32位的数)

这32位表示了 `rt_thread_ready_table` 的32个组。这样考虑的，以8位为单位进行最小位图的展开，这样相当于形成了一个二级的映射关系。

256bit-—>32group-—>8bit位图

把这0x00~0xFF表示成一个数

那么，这8位优先级中找到最高的优先级的问题，就变成了，这个数里面最低位为1的问题。

是这么理解的：

| bit7 | bit6 | bit5 | bit4 | bit3 | bit2 | bit1 | bit0 | 十进制 | 最高优先级 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 0 |
| .. | .. | .. | .. | .. | .. | .. | .. | .. | .. |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 255 | 0 |

研究一下上表，会发现，可以事先确定出来，最高优先级是几。

因此可以得到一个数组

```
const rt_uint8_t rt_lowest_bitmap[] =
{
    /* 00 */ 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 10 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 20 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 30 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 40 */ 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 50 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 60 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
```

```
    /* 70 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 80 */ 7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 90 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* A0 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* B0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* C0 */ 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* D0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* E0 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* F0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

那，我们看看，进程在创建的时候，优先级是怎么来影响个个变量的呢

```
// 一开始的时候，要用到thread结构体
thread->init_priority = priority;
// 在starup函数里面，转换优先级，纯数学
thread->number = thread->current_priority >> 3; //相当于prioroity/8取得落在哪一个组里面
thread->number_mask = 1L << thread_number;
thread->high_mask = 1L << (thread->current_priority & 0x07); //相当于priority%8，余数，取的这一个组中，8bit中的第几位
// 置一的目的在于方便位操作
// insert进程的时候置位
rt_thread_ready_table[thread->number] |= thread->high_mask; // 对应组中的对应位，置一
rt_thread_ready_priority_group |= thread->number_mask; // 对应组中的标记，置一
```

现在可以看出来， `rt_thread_ready_table[]` 是对256个优先级的一一映射。

而 `rt_thread_ready_priority_group` 是对这个表的一个抽象，分组。

看到这儿，我觉得我可以理解schedule的代码了

256位的十六进制就是0xFFFF FFFF。

主要使用 `rt_thread_ready_priority_group` 来进行操作，每次比较8位，相当于把32个组又分成了8x4份来进行比较。

因此，会有

```
rt_thread_ready_priority_group & 0x000000FF
rt_thread_ready_priority_group & 0x0000FF00
rt_thread_ready_priority_group & 0x00FF0000
rt_thread_ready_priority_group & 0xFF000000
```

这样的代码比较，主要是比较出来，哪8位的里面有1的存在，而且是从低位到高位比较的，这样，也遵循了低位表示高优先级的原则

假设0xff的就中了，那么

```
number = rt_lowest_bitmap[rt_thread_ready_priority_group & 0xff];
```

就表示了，在这8位中，查表，得到了，最低优先级的位数，赋值给number。此时number表示，在32个组中，第几个组里面有最后的结果。

然后这个组里的第几位要通过

```
rt_lowest_bitmap[rt_thread_ready_table[number]];
```

来查表，找出了，具体的第几位。

最后再把它们拼接起来，所以就有了

```
highest_ready_priority = (number << 3) + rt_lowest_bitmap[rt_thread_ready_table[number]]
```

整个相当于查了四次表

- 第一次： `rt_thread_ready_priority_group` 中第几组8位中存在1
- 第二次： 这个8位的组中，最低位是1的是第几位,代表了32个组中第几个组中有最高优先级
- 第三次： `rt_thread_ready_table` 中这一组中的内容取出来
- 第四次： 再用一边 `rt_lowest_bitmap` 的查表，确认这一组中第几位是1，表示了最高优先级

最后把“组信息“和“组内的位信息“还原回表示整个256位中第几位的数值

一个先抽象运算再还原结果的过程。其中这个 `rt_lowest_bitmap` 的位图被使用了两次，算法是一样的，其意义是不一样的。

虽然都是为了找8位中最低位为1的位数，只是，这个位数所表示的含义是不一样的。

通过这一系列的折腾，就找到了当前最高的优先级的进程，而且是在ready队列里面的，接下来就准备调用它了。

不在ready队列里面的是不会被调度的。

Author: Wizard.Yang <xblandy@gmail.com>

Date: 2012-09-11 09:18:00 CST