RT-Thread 实时操作系统函数查询手册

内核版本：version 1.1.0

文档版本号：0.0.1

日期：2013/4/20

修订记录

2013/4/20	barryzxy@163.com	初始版本

1 线程管理

在操作系统中，程序代码已不再是以 main()函数为主体去编写程序，是以线程(任务)为单位的设计。这里介绍 RT-Thread 中与线程相关的函数。

- 线程挂起：rt_err_t rt_thread_suspend (rt_thread_t thread)
- 线程删除：rt_err_t rt_thread_delete(rt_thread_t thread)
- 线程脱离：rt_err_t rt_thread_detach(rt_thread_t thread)
- 线程恢复：rt_err_t rt_thread_resume(rt_thread_t thread)
- 线程挂起：rt_err_t rt_thread_suspend(rt_thread_t thread)
- 线程控制 rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg)
- 返回线程句柄：rt_thread_t rt_thread_self(void)
- 线程让出处理器：rt_err_t rt_thread_yield(void)

1.1 rt_thread_init 线程初始化

代码 1-1 rt_thread_init 原型声明

```
/* thread.c */
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)
```

功能：

rt_thread_init 函数用来初始化静态线程对象。而线程句柄（或者说线程控制块指针），线程栈由用户提供。静态线程是指，线程控制块、线程运行栈一般都设置为全局变量，在编译时就被确定、被分配处理，内核不负责动态分配内存空间。需要注意的是，用户提供的栈首地址需做系统对齐（例如 ARM 上需要做 4 字节对齐）。

参数：

*thread　　线程句柄。线程句柄由用户声明(struct rt_thread led_thread;)，并指向对应线程控制块的内存地址，声明的线程句柄就是线程一个代号，所有对线程的操作者是句柄进行的。

*name　　线程名称。

(*entry)　　线程函数入口，也是线程建立后执行的代码。

*parameter　线程入口函数的参数。

*stack_start　线程堆栈的首地址

stack_size　　线程堆栈的大小，字节为单位。

priority　　线程优先级。优先级范围由系统的配置情况决定，（rtconfig.h 中的 RT_THREAD_PRIORITY_MAX 宏定义），数值越小优先组越高，0 代表最高优先级，如果最大是 32，一般在程序设计中开始时从 16 开始，后续线程可根据其重要性把优先级按排在其前后。

tick　　线程时间片大小。时间片的单位是操作系统的时钟节拍（在 rtconfig.h 中#define RT_TICK_PER_SECOND 100 一秒钟有 100 个节拍，即每个节拍为 0.01 秒）。当系统中存在相同优先级时，线程一次调度能够运行的最大时间长度，当时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行。

返回值：

RT_EOK　　线程初始化成功

RT_ERROR　线程创建失败

中断例程不可调用。

范例：

代码 1-2 rt_thread_init 示例

```
/* 线程句柄声明 */
static struct rt_thread led_thread;
/* 线程堆栈声明 */
static rt_uint8_t led_stack[ 512 ];
/* 线程入口 */
static void led_thread_entry(void* parameter)
{
    ...
}
/* 用户应用入口 */
int rt_application_init()
{
    rt_err_t result;
    /* 初始化 led_thread */
    result = rt_thread_init(&led_thread, //
                        "led",                      //线程名为 led
                        led_thread_entry,           //线程函数入口
                        RT_NULL,                    //没有参数
                        (rt_uint8_t*)&led_stack[0],
                        sizeof(led_stack),
                        20,
                        5);
    if (result == RT_EOK)           //如果返回正确，启动线程 led_thread
        rt_thread_startup(&led_thread);
    else
        return -1;

    return 0;
}
```

本例中，创建线程 led_thread，线程入口函数为 led_thread_entry，传递的堆栈的首地址为&led_stack[0]，确定线程的优先级以及相同优先级下分到的时间片。

1.2 rt_thread_startup 线程启动

代码 1-3 rt_thread_startup 原型声明

```
/* thread.c */
rt_err_t rt_thread_startup(rt_thread_t thread)
```

功能：

创建（初始化）的线程对象的状态处于初始态，并未进入就绪线程的调度队列，我们可以调用这个函数接口启动这个线程。

当调用这个函数时，将把线程的状态更改为就绪状态，并放到相应优先级队列中等待调度。如果新启动的线程优先级比当前线程优先级高，将立刻切换到这个线程。

参数：

thread 线程句柄。线程句柄由用户声明(struct rt_thread led_thread;)，并指向对应线程控制块的内存地址，声明的线程句柄就是线程一个代号，所有对线程的操作者是句柄进行的。

返回值：

RT_EOK 线程初始化成功

RT_ERROR 线程创建失败

注意：

中断例程不可调用。

范例：例程见代码 1-2 rt_thread_init 示例

1.3 rt_thread_delay、rt_thread_sleep 线程睡眠

在为了 RT-Thread 的 API 的友好，线程睡眠有两种形式 rt_thread_delay、rt_thread_sleep。

代码 1-4 线程睡眠原型声明

```
/* thread.c */
/* rt_thread_delay 原型声明 */
rt_err_t rt_thread_delay(rt_tick_t tick)
/* rt_thread_sleep 原型声明 */
rt_err_t rt_thread_sleep(rt_tick_t tick)
```

功能：

在实际应用中，我们有时需要让运行的当前线程延迟一段时间，在指定的时间到达后重新运行，这就叫做“线程睡眠”。线程睡眠可使用上面这两个函数接口之一。

这两个函数接口的作用相同，调用它们可以使当前线程挂起一段指定的时间，当这个时间过后，线程会被唤醒并再次进入就绪状态。这个函数接受一个参数，该参数指定了线程的休眠时间（单位是 OS Tick 时钟节拍）。

参数：

tick　　线程睡眠时间

返回值：

RT_EOK　　线程初始化成功

RT_ERROR　　线程创建失败

注意：

中断例程不可调用。

范例：

代码 1-5 线程睡眠示例

```
/*  x 线程函数入口 */
static void x_thread_entry(void* parameter)
{
    while (1)
    {
        ...
        rt_thread_delay( 50 ); /* 线程延时 50 个时钟节拍 */
        ...
    }
}
```