事件的基本使用

RealTouch 评估板 RT-Thread 入门文档

版本号：1.0.0

日期：2012/8/12

修订记录

日期	作者	修订历史
2012/8/12	Prife	创建文档

实验目的

- 了解事件的基本用法
- 熟练使用事件实现多个线程间同步

硬件说明

本实验使用 RT-Thread 官方的 Realtouch 开发板作为实验平台。涉及到的硬件主要为：

- 串口 3，作为 rt_kprintf 输出
  需要连接 JTAG 扩展板，具体请参见《Realtouch 开发板使用手册》

实验原理及程序结构

实验设计

本实验演示在 RT-Thread 中使用事件（EVENT）实现多线程间同步和通信。

事件是一个 32bit（4 个字节）的变量，其中每一个位可以表示代表一种事件。接收事件的线程既可以在多个事件同时发生后（即多个 bit 位同时置 1）触发，正如本例中线程 1 中第一条语句所演示的那样。

也可以多个事件任意一个发生后（即多个 bit 位任意一个置位）就可以触发。

主程序中创建三个线程，线程 1 接收事件标志。线程 2 和线程 3 则向发送事件标志。

本实验同样使用静态事件作为演示，涉及静态事件初始化/脱离。动态事件创建/删除类似，不再赘述。

源程序说明

本实验对应 kernel_event_basic。

系统依赖

在 rtconfig.h 中需要开启

- #define RT_USING_EVENT
  此项必须，开启此项即可使用事件机制。
- #define RT_USING_HEAP
  此项可选，开启此项可以创建动态线程和动态邮箱，如果使用静态线程和静态信号量，则此项不是必要的。
- #define RT_USING_CONSOLE
  此项必须，本实验使用 rt_kpriintf 向串口打印按键信息，因此需要开启此项

主程序说明

在 applications/application.c 中定义静态消息队列控制块、存放消息的缓冲区。如下所示

定义全局变量代码

```
/* 事件控制块 */
static struct rt_event event;
```

在 applications/application.c 中的 int rt_application_init()函数中，初始化静态事件。

初始化静态事件代码

```
    rt_err_t result;

    /* 初始化事件对象 */
    rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init event failed.\n");
        return -1;
    }
```

在 int rt_application_init()初始化名为“thread1”的 thread1 的静态线程，如下所示。

创建线程 1 代码

```
    rt_thread_init(&thread1,
                   "thread1",
                   thread1_entry,
                   RT_NULL,
```

```
                &thread1_stack[0],
                sizeof(thread1_stack),8,50);
rt_thread_startup(&thread1);
```

其线程入口函数如下所示，线程 1 调用 rt_event_recv 函数，等待 event 事件上 bit3 和 bit5 代表的事件发生；第三个参数中的 RT_EVENT_FLAG_AND 表示事件标志采用与，即等待的多个事件同时发生此函数才返回，否则继续等待事件；RT_EVENT_FLAG_CLEAR 表示接收到事件后将事件相关位清除；RT_WAITING_FOREVER 表示如果等待的时间没有发生，则永远等待下去。

线程 1 先等待 bit3 和 bit5 表示的事件，若都发生后则向串口打印信息，否则永远等待下去。之后使用 rt_thread_delay 延时 1 秒钟。之后再次等待 bit3 和 bit5 表示的事件，这一次使用的是 RT_EVENT_FLAG_OR，这表示 bit3 和 bit4 任意一个事件发生都可以。

线程 1 代码

```
ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
static char thread1_stack[1024];  //设置线程堆栈为 1024Bytes
struct rt_thread thread1;       //定义静态线程数据结构
/* 线程 1 入口 */
/* 线程 1 入口函数 */
static void thread1_entry(void *param)
{
    rt_uint32_t e;

    /* receive first event */
    if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
        RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
        RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: AND recv event 0x%x\n", e);
    }

    rt_kprintf("thread1: delay 1s to prepare second event\n");
    rt_thread_delay(RT_TICK_PER_SECOND);

    /* receive second event */
```

```
    if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
        RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
        RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: OR recv event 0x%x\n", e);
    }
    rt_kprintf("thread1 leave.\n");
}
```

在 int rt_application_init()初始化名为”thread2”的 thread2 的静态线程，如下所示。

初始化线程 2 代码

```
    rt_thread_init(&thread2,
                "thread2",
                thread2_entry,
                RT_NULL,
                &thread2_stack[0],
                sizeof(thread2_stack),9,50);
rt_thread_startup(&thread2);
```

其线程入口函数如下所示,线程 2 调用 rt_event_send 发送 bit3 事件，之后调用 rt_kprintf 打印结束信息后，线程函数运行结束。

线程 2 代码

```
ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
static char thread2_stack[1024];  //设置线程堆栈为 1024Bytes
struct rt_thread thread2;       //定义静态线程数据结构
/* 线程 2 入口 */
static void thread2_entry(void *param)
{
    rt_kprintf("thread2: send event1\n");
    rt_event_send(&event, (1 << 3));
    rt_kprintf("thread2 leave.\n");
}
```

在 int rt_application_init()初始化名为”thread3”的 thread3 的静态线程，如下所示。

初始化线程 3 代码

```
    rt_thread_init(&thread3,
                   "thread3",
                   thread3_entry,
                   RT_NULL,
                   &thread3_stack[0],
                   sizeof(thread3_stack),10,50);
    rt_thread_startup(&thread3);
```

其线程入口函数如下所示，线程 2 首先调用 rt_event_send 发送 bit5 事件，延时 20 个 tick 之后，再次发送 bit5 表示的事件，线程函数运行结束。

```
ALIGN(RT_ALIGN_SIZE)        //设置下一句线程栈数组为对齐地址
static char thread3_stack[1024];  //设置线程堆栈为 1024Bytes
struct rt_thread thread3;       //定义静态线程数据结构

/* 线程 3 入口函数 */
static void thread3_entry(void *param)
{
    rt_kprintf("thread3: send event2\n");
    rt_event_send(&event, (1 << 5));

    rt_thread_delay(20);

    rt_kprintf("thread3: send event2\n");
    rt_event_send(&event, (1 << 5));

    rt_kprintf("thread3 leave.\n");
}
```

编译调试及观察输出信息

编译请参见《RT-Thread 配置开发环境指南》完成编译烧录，参考《Realtouch 开发板使用手册》完成硬件连接，连接扩展板上的串口和 jlink。

运行后可以看到如下信息：

串口信息

```
\ | /
```

```
- RT -     Thread Operating System
 / | \     1.1.0 build Aug  9 2012
 2006 - 2012 Copyright by rt-thread team
thread2: send event1
thread2 leave.
thread3: send event2
thread1: AND recv event 0x28
thread1: delay 1s to prepare second event
thread3: send event2
thread3 leave.
thread1: OR recv event 0x20
thread1 leave.
```

结果分析

整个程序运行过程中各个线程的状态变化：

rt_application_init 中创建了三个线程，thread1（线程 1）、thread2（线程 2）、thread3（线程 3），线程 1 的优先级最高，线程 2 的优先级次之，线程 3 优先级最低。

线程 1 首先运行，其线程处理函数中调用 rt_event_recv，以‘AND’方式接收 event 上 bit3 和 bit5 表示的事件，如果这两个事件中只要有任何一个没有发生，则线程 1 就会被挂起到 event 事件上， 即只有当 bit3 和 bit5 表示的事件都发生后，线程 1 才被唤醒。显然，线程 1 被挂起后，调度器会重新调度，线程 2 被调度运行，它会打印

```
thread2: send event1
```

然后向事件 event 发送 bit3 代表的事件。之后线程 2 退出。此时依然不满足线程 1 的等待的事件条件，bit3 置位，bit5 依然为 0，因此线程 1 依然被挂起在事件 event 上。调度器继续调度，线程 3 被调度运行，线程 3 打印

```
thread3: send event2
```

然后使用 rt_event_send(&event, (1 << 5))向事件 event 发送 bit5 代表的事件，此时线程 1 等待的条件满足，线程 1 的状态由挂起转换成就

绪。线程 1 的优先级高于线程 3，因此在下一个系统 tick 中断后，线程 1 被调度运行。线程 1 接收事件 event 后，会将 event 的 bit3 和 bit5 清 0，接下来向串口打印：

```
thread1: AND recv event 0x28
thread1: delay 1s to prepare second event
```

之后，线程 1 再次接收事件 bit3 和 bit5，这次是以‘OR’的方式等待事件，即 bit3 和 bit5 中任意一个发生则线程 1 等待的条件满足，否则被挂起在事件上。

此时线程 1 再次被挂起，等待 bit3 或 bit5 代表的事件发生。内核再次调度线程 3 运行，线程 3 中调用 rt_event_send(&event, (1 << 5))，这会将线程 1 从挂起态转换成就绪态，但并不会立刻执行状态切换，线程切换会发生在下一次系统 tick 中断中。

因此线程 3 继续运行，打印

```
thread3 leave.
```

在之后的系统 tick 中断中，线程 1 被调度运行，打印

```
thread1: OR recv event 0x20
thread1 leave.
```

线程 1 的处理函数也运行完毕后退出，之后内核调度运行 IDLE 线程。

以上就是整个实验中，各个线程的状态转换过程。

总结

本实验演示了 RT-Thread 中事件作为多线程通信的用法，以静态事件控制块为例，动态事件的用法类似，只是创建/删除需要使用 rt_event_create/rt_event_delete 函数，读者可以使用动态事件重复本实验。