

一起来学 RT-Thread 系列教程

作者: jiezhi320

日期: 2013-03-02

QQ 交流群: 258043068 欢迎加入

✧ 目的:

本人是一名电子爱好者, 一次机缘巧合接触到了 RT-thread 这个开源操作系统, 从此便成了 RT-thread 这款 OS 的 fans, 因为它实在是太好使、太可爱了!。在这里我和大家分享一下我在用这款 OS 时积累的一些东西, 希望对那些想要学习 RT-thread 的同学有所帮助。由于本人水平有限、文字拙劣, 文中如果有不对的地方, 欢迎指正、交流。

文章将采用连载的形式, 从内核线程、线程间各种通信机制、各种组件的使用等方面一一做介绍。另外文章主要讲解 RT-thread 的相关使用方法, 即如何应用, 而不是分析 RT-thread 的内部具体实现机制。

✧ 硬件平台

后续例子中所牵扯到的软硬件实验环境如下:

操作系统: Windows XP SP3

开发编译环境: Keil MDK 4.54 版

对应硬件平台: 魔笛 stm32 RT_thread 综合实验平台

仿真器: STlink

RT-Thread 版本: 1.1.0 版、1.2.0 版

如果读者使用别的硬件平台, 请稍改下里面的源码, 使之与自己的目标板对应。

魔笛 RTT 实验平台是建立在原《stm32 网络收音机》基础上的, 详情参见:

<http://www.rt-thread.org/phpBB3/viewtopic.php?f=17&t=2398>

第一篇：认识 RT-thread

日期：2013-03-02

✧ RT-thread 简介

RT-Thread（实时线程操作系统）是国内 RT-Thread 工作室精心打造的稳定的开源实时操作系统，“她”是 RTT 核心成员历时 4 年，呕心沥血研发，力图突破国内没有小型稳定的开源实时操作系统局面的开山之作，曾获得“[第六届中日韩开源软件竞赛](#)”技术优胜奖（其他两个技术优胜奖获得者为淘宝的 OceanBase 和红旗的 Qomo Linux）它不仅仅是一款开源意义的硬实时操作系统（不是软的哦），也是一款产品级别的实时操作系统，目前已经被国内十多家企业采用，被证明是一款能够稳定持续运行的操作系统。

RT-Thread 实时操作系统核心是一个高效的硬实时核心，它具备非常优异的实时性、稳定性、可剪裁性，当进行最小配置时，内核体积可以到 3k ROM 占用、1k RAM 占用。目前 RT-thread 支持的分支和包含的组件如下：

分支：

- ARM Cortex-M3: STM32F1, STM32F2, LPC176xx, LPC18xx, LM3S, EFM32, MB9BF
- ARM Cortex-M4: STM32F4, LM4S, LPC4300
- ARM7TDMI: LPC2478, LPC2148, AT91SAM7S, AT91SAM7X, S3C44B0
- ARM720T: SEP4020
- ARM9: AT91SAM9260, S3C2440
- NIOS-II
- XILINX MicroBlaze
- AVR32
- Blackfin 533
- MIPS: PIC32, Jz47xx
- PPC450: taihu
- x86
- windows simulator (VC++)

组件

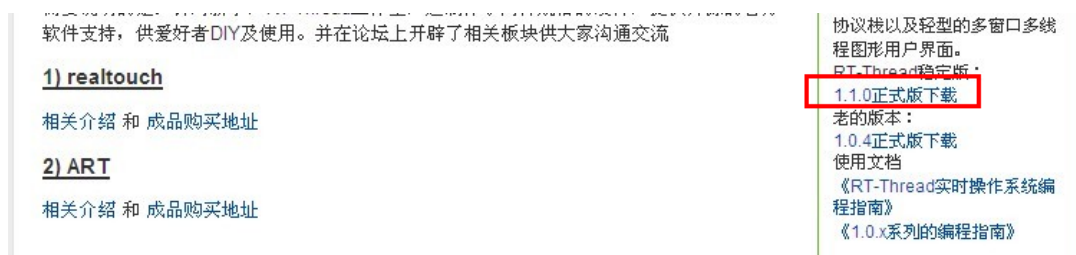
- CMSIS, CMSIS-RTOS
- RT-Thread DFS 文件系统: devfs, ELM FatFs, JFFS2, NFS, romfs, UFFS, YAFFS2
- **finsh shell** (类似命令行的组件, RTT 的亮点哦)
- libc: armlibc (针对 Keil MDK), newlib
- POSIX: pthreads, libdl
- 网络: lwIP 1.4.0
- RT-Thread GUI
- lua
- Device Drivers: IIC, MTD NOR/NAND, RTC, SDIO, serial, SPI Bus/Device, USB device/host

✧ RT-thread 授权

我们使用操作系统, 应该都会考虑一个收费问题, 使用 RT-thread, 我们就不用担心这个问题了。RT-Thread 采用 GPL-V2 发布, 并且承诺永久不会针对使用 RT-Thread 收费, 用户只需要保留 RT-Thread 的 LOGO 既可以免费使用。

✧ 下载 RT-thread 源码、资料

RT-thread 最新稳定版是 1.1.0 版, 我们可登陆 RT-thread 官方网站 <http://www.rt-thread.org>, 点击如下图中红色框中的超链接进行下载, 同时也可以下载到其编程指南。



RT-thread 工作室人员除了进行 RT-thread 操作系统及其组件的开发维护外, 还主导了一些较大开源的软硬件项目, 这些项目在其论坛中都可以找得到, 也可以跟着这些开源项目去深入学习 RT-thread 的使用。

第二篇感受 RT-thread

日期：2013-03-12

✧ RT-thread 源码目录结构介绍

解压源码后，会看到如下的文件和文件夹：



接下来简单说说各文件夹、文件的作用。

Bsp—包含 RT-thread 的各个移植分支；

components —包含 RT-thread 的各中组件：finsh、文件系统、网路协议栈等；

documentation—一些介绍性的文档，包括其代码风格的要求；

examples—各种示例代码，是很好的学习素材；

include—一些头文件；

libcpu—各种 CPU 体系结构下的相关移植；

src—RT-thread 内核核心代码；

tools —使用 scon 自动化创建工具时需要的一些文件；

AUTHORS—RT-thread 开发者列表

COPYING—权限说明

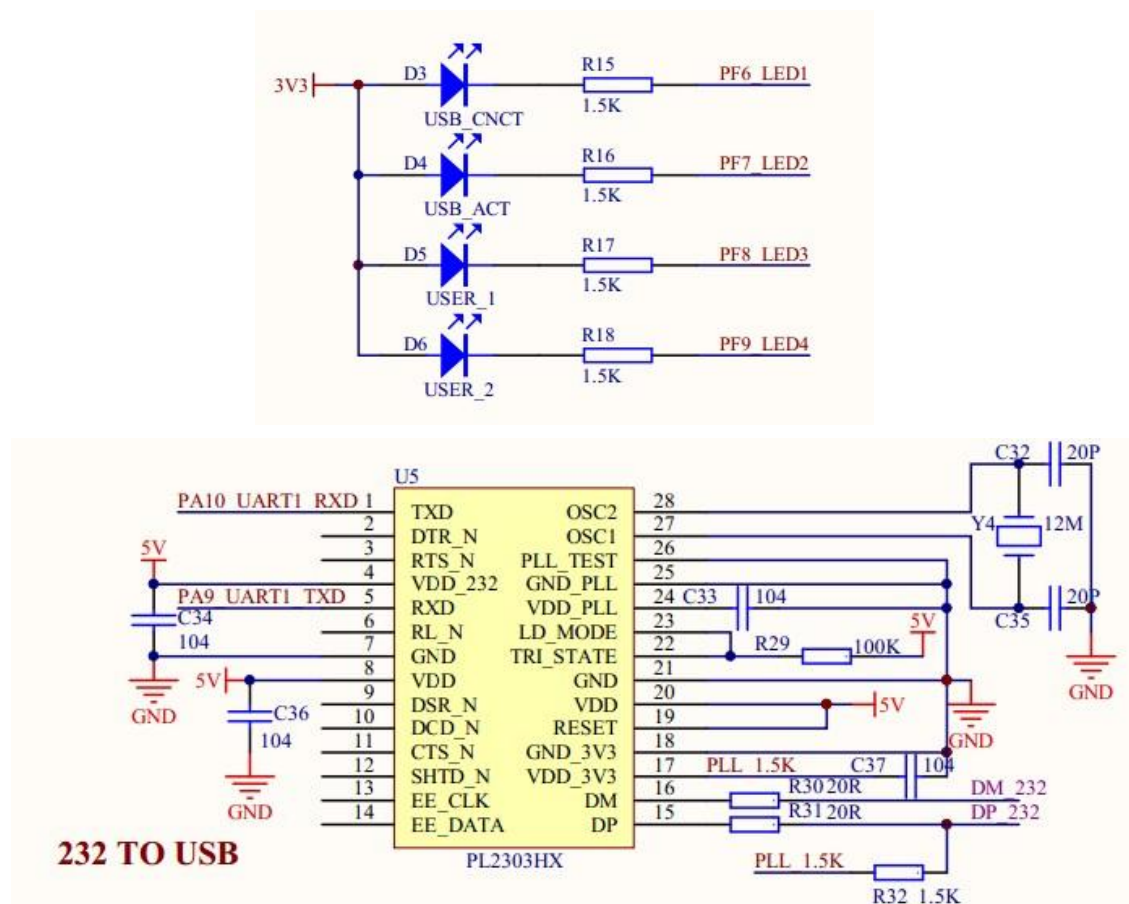
✧ 第一次运行 RT-thread

RT-thread 成员已经为我们做好了各种平台下的移植，我们打开 bsp 目录下 stm32f10x 系列的对应分支，双击 MDK 下工程 project.uvproj，打开工程。

这个示例工程包含了 RT-thread 的内核、finsh 组件这两个最基本的部分，主代码完成了从 RT-thread 的启动到创建一个**闪灯线程**的过程，程序运行时会通过**串口终端打印**运行信息。

我们根据目标板上的 LED 和串口引脚对应关系修改一下程序，使程序能在我们的板子上正常工作。

目标板 LED、串口电路如下：



led.c 中我们用 GPIOF8、GPIOF9 来替换原来的 GPIOE2、GPIOE3：

```
#define led1_rcc          RCC_APB2Periph_GPIOF
#define led1_gpio         GPIOF
#define led1_pin          GPIO_Pin_8

#define led2_rcc          RCC_APB2Periph_GPIOF
#define led2_gpio         GPIOF
#define led2_pin          GPIO_Pin_9
```

程序中默认使用串口 1 作为终端，针对我的目标板就无需改了，如果你的板子串口不是串口 1，则改动下面两处：

board.h 中：

```
#define STM32_CONSOLE_USART    1 //根据目标板自己实际修改
```

rt_config.h 中：

```
#define RT_USING_UART1 //根据目标板实际情况修改
```

编译后下载到目标板运行，我们会看到 D5 LED 灯以 1Hz 的频率闪烁，同时串口输出了系统启动信息：



OK，到这里，RT-Thread 就算是在我们的目标板上真正的运行起来了，是不是很简单呢？赶紧试试吧，心动不如行动！

第三篇 RT-Thread 启动过程

日期: 2013-03-24

✧ RT-Thread 启动过程

我们分析一个系统的启动过程，一般都从 main 函数开始，我们点开 startup.c 找到 main 函数，main 函数就下面 3 行：

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

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

    return 0;
}
```

第一行是关中断的操作，第三行是返回退出（永远都执行不到），那么看来 RT-Thread 的启动就在第二行中进行的，我们贴上 rtthread_startup() 的内容来大致浏览一下其启动过程：

```
void rtthread_startup(void)
{
    /* 初始化硬件平台相关：时钟设置、中断设置、系统滴答设置、串口设置 */
    rt_hw_board_init();
    up_mcu_show();
    /* 打印 RT-Thread 版本信息 */
    rt_show_version();
    /* init tick */
    rt_system_tick_init();
    /* 内核对象初始化 */
    rt_system_object_init();
    /* 系统定时器初始化 */
    rt_system_timer_init();
    /* 如果使用动态内存分配，则配置之 */
#ifdef RT_USING_HEAP
#ifdef STM32_EXT_SRAM
    rt_system_heap_init((void*)STM32_EXT_SRAM_BEGIN, (void*)STM32_EXT_SRAM_END);
#else
#ifdef __CC_ARM
```

```
rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit,(void*)STM32_SRAM_END);
#elif __ICCARM__
    rt_system_heap_init(__segment_end("HEAP"),(void*)STM32_SRAM_END);
#else
    /* init memory system */
    rt_system_heap_init((void*)&__bss_end,(void*)STM32_SRAM_END);
#endif
#endif
#endif
/* 系统调度器初始化 */
    rt_system_scheduler_init();
#ifdef RT_USING_DFS
    /* init sdcard driver */
    #if STM32_USE_SDIO
        rt_hw_sdcard_init();
    #else
        rt_hw_msd_init();
    #endif
#endif
/* 下面可加入用户所需的相关初始化 */

    /* 上面可加入用户所需的相关初始化 */
/* 实时时钟初始化 */
    rt_hw_rtc_init();
/* 系统设备对象初始化 */
    rt_device_init_all();
/* 用户应用初始化 */
    rt_application_init();
#ifdef RT_USING_FINSH
    /* init finsh */
    finsh_system_init();
    finsh_set_device("uart1");
#endif
    list_date();/*显示当前时间 by jiezhi320 */
/* 初始化软件定时器 */
    rt_system_timer_thread_init();
/* 初始化空闲线程 */
    rt_thread_idle_init();
/* 开始线程调度此后便进入各个线程的无限循环 */
    rt_system_scheduler_start();
/* never reach here */
```



```
return;
}
```

上面的函数完成了系统启动前的所有初始化动作,包括必要的硬件初始化、堆栈初始化、系统相关组件初始化、用户应用程序初始化,然后启动调度机制。

1、rt_hw_board_init()

完成中断向量表设置、系统滴答时钟设置,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。

2、rt_system_heap_init()

RT-Thread 提供动态内存管理机制(由 RT_USING_HEAP 宏来选择性开启,默认开启),这个函数用来设置需要系统来管理的内存段地址。对于 stm32f1 这样的芯片这里有两种选择:

一种是除去编译时分配的全局变量、静态局部变量外的其他剩余内存被设置为系统堆空间,被系统管理起来。比如魔笛 F1 板子芯片内部有 64k ram,除去编译后的 RW、ZI 所占去的内存,剩余的就让系统管理起来:

```
rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit, (void*)STM32_SRAM_END);
```

一种是板子有外扩 ram,这整个外扩 ram 可被设置为堆空间,被系统管理起来

```
rt_system_heap_init((void*)STM32_EXT_SRAM_BEGIN, (void*)STM32_EXT_SRAM_END);
```

内存管理设置好以后,应用程序就可以使用 rt_malloc、rt_realloc、re_free 等函数了。

3、rt_application_init()

这个函数是为用户准备的,用户可以在这个函数里创建自己的应用线程。

✧ RT-Thread 的裁剪

我们在上面的启动函数中看到了如下的一些宏定义:

```
#ifdef RT_USING_DFS
#ifdef RT_USING_FINSH
```

这些宏定义就是为了 RT-Thread 的可裁剪性而做的,对于其裁剪配置,我们在 rtconfig.h 中进行,rtconfig.h 列出了每个可配置项,可参考配置项上的英文解释来了解每项的意义。我们在应用过程中,需要根据实际需求情况来选择打开或屏蔽掉某些功能,比如是否用到文件系统、是否用到网络功能、是否用 RTGUI(图形界面)等等。

每次重新配置 rtconfig.h 后往往涉及到一些 c 文件的加入或移除,还有头文件的搜索路径变化,如果手动来添加和移除无疑是一个让人头大的事,为此 RT-Thread 提供用 scons 命令来自动生成 mdk 或 iar 工程的方法,这些我们后续再说,先来学习内核基础比较好些。

第四篇线程基本知识

日期：2013-03-24

✧ 什么叫线程？

RT-Thread 号称实时线程 RTOS，那么什么叫线程？

人们在生活中处理复杂问题时，惯用的方法就是“分而治之”，即把一个大问题分解成多个相对简单、比较容易解决的小问题，小问题逐个被解决了，大问题也就随之解决了。同样，在设计一个较为复杂的应用程序时，也通常把一个大型任务分解成多个小任务，然后通过运行这些小任务，最终达到完成大任务的目的。

在 RT-Thread 中，与上述小任务对应的程序实体就叫做“线程”（或任务），RT-Thread 就是一个能对这些小“线程”进行管理和调度的多“线程”操作系统。

✧ 线程的组成

RT-Thread 中的“线程”一般由三部分组成：线程代码（函数）、线程控制块、线程堆栈，我们来看看《篇 2-例程 1-第一次运行 RTT》中的闪灯线程是如何构造的。

● 线程代码：

```
void led_thread_entry(void* parameter)
{
    unsigned int count=0;

    rt_hw_led_init();

    while(1)
    {
#ifdef RT_USING_FINSH
        rt_kprintf("led on, count : %d\r\n",count);
#endif
        count++;
        rt_hw_led_on(0);
        rt_thread_delay( RT_TICK_PER_SECOND/2 );

#ifdef RT_USING_FINSH
        rt_kprintf("led off\r\n");
#endif
        rt_hw_led_off(0);
        rt_thread_delay( RT_TICK_PER_SECOND/2 );
    }
}
```

```
}
```

上面即是一个典型的线程代码结构—**无限死循环**，当然还有一种线程结构是**顺序执行**的，比如初始化线程，它执行到 `return()`，就会返回，当其返回后，系统会在 `idle` 线程中将其删除，从而使其退出调度队列。一般情况下用户线程都将是一个无限循环结构。

- **线程控制块：**

```
static struct rt_thread led_thread;
```

其中线程控制块 `rt_thread` 结构体具体内容如下：

```
struct rt_thread
{
/* rt object */
    char name[RT_NAME_MAX];/**< the name of thread */
    rt_uint8_t type;/**< type of object */
    rt_uint8_t flags;/**< thread's flags */

#ifdef RT_USING_MODULE
    void*module_id;/**< id of application module */
#endif

    rt_list_t list;/**< the object list */
    rt_list_t tlist;/**< the thread list */

/* stack point and entry */
    void*sp;/**< stack point */
    void*entry;/**< entry */
    void*parameter;/**< parameter */
    void*stack_addr;/**< stack address */
    rt_uint16_t stack_size;/**< stack size */
/* error code */
    rt_err_t error;/**< error code */
    rt_uint8_t stat;/**< thread stat */
/* priority */
    rt_uint8_t current_priority;/**< current priority */
    rt_uint8_t init_priority;/**< initialized priority */
#ifdef RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t number;
    rt_uint8_t high_mask;
#endif
    rt_uint32_t number_mask;

#ifdef defined(RT_USING_EVENT)
/* thread event */
    rt_uint32_t event_set;
```

```

    rt_uint8_t event_info;
#endif

    rt_ubase_t init_tick;/**< thread's initialized tick */
    rt_ubase_t remaining_tick;/**< remaining tick */

    struct rt_timer thread_timer;/**< built-in thread timer */

    void(*cleanup)(struct rt_thread *tid);/**< cleanup function when thread exit
    */
    rt_uint32_t user_data;/**< private user data beyond this thread */
};

```

它记录了线程的各个属性，系统用线程控制块链表对其进行管理。

- 线程堆栈:

```
static rt_uint8_t led_stack[512];
```

线程堆栈是一段连续的内存块，当线程切换后，为了满足线程切换和响应中断时保存 cpu 寄存器中的内容及任务调用其它函数时的准备，每个线程都要配有自己的堆栈

✧ 创建一个我们自己的线程

前面说了这么多，我们还是来自己建立一个线程，这样的话印象更深刻。

RT-Thread 中的线程分为**静态线程**—线程堆栈由编译器静态分配，使用 `rt_thread_init()` 函数创建和**动态线程**—线程堆栈由系统动态分配，使用 `rt_thread_create()` 函数创建。在例程《篇 4-例程 1-自己创建静态、动态线程》中我们分别建立一个静态线程和动态线程，这两个线程的任务都是使板上的 LED 灯以 1Hz 频率闪烁。

```

/* 静态线程的线程堆栈*/
static rt_uint8_t led1_stack[512];
/* 静态线程的线程控制块 */
static struct rt_thread led1_thread;

void demo_thread_creat(void)
{
    rt_err_t result;
    /* 动态线程的线程控制块指针 */
    rt_thread_t led2_thread;

    rt_hw_led_init();

    /* 创建静态线程：优先级 20，时间片 2 个系统滴答 */
    result = rt_thread_init(&led1_thread, "led1",
                           static_thread_entry, RT_NULL,

```

```

        (rt_uint8_t*)&led1_stack[0],
        sizeof(led1_stack),20,2);

if(result == RT_EOK)
{
    rt_thread_startup(&led1_thread);
}

/* 创建动态线程: 堆栈大小 512 bytes , 优先级 21 , 时间片 2 个系统滴答 */
led2_thread =rt_thread_create("led2",
                                dynamic_thread_entry, RT_NULL,
                                512,21,2);

if(led2_thread != RT_NULL)
    rt_thread_startup(led2_thread);
}

```

编译此例程下载到板子中运行，我们看到 D3、D4 灯以同样的节奏闪烁，同时串口也打印出了如下的运行信息：

```

finsh>> 静态线程运行
动态线程运行
静态线程运行
动态线程运行
静态线程运行
动态线程运行

```

◇ 静态线程 VS 动态线程

从上例可看出，静态、动态线程在做同样的事情时，从效果上看，是没有任何差别的！那么，我们在实际中如何抉择？

使用静态线程时，必须先定义静态的线程控制块，并且定义好堆栈空间，然后调用 `rt_thread_init()` 来完成线程的初始化工作。采用这种方式，线程控制块和堆栈占用的内存会放在 `RW/ZI` 段，这段空间在编译时就已经确定，它不是可以动态分配的，所以不能被释放，而只能使用 `rt_thread_detach()` 函数将该线程控制块从对象管理器中脱离。

使用动态定义方式 `rt_thread_create()` 时，RT-Thread 会动态申请线程控制块和堆栈空间。在编译时，编译器是不会感知到这段空间的，只有在程序运行时，RT-Thread 才会从系统堆中申请分配这段内存空间，当不需要使用该线程时，调用 `rt_thread_delete()` 函数就会将这段申请的内存空间重新释放到内存堆中。

这两种方式各有利弊，静态定义方式会占用 `RW/ZI` 空间，但是不需要动态分配内存，运行时效率较高，实时性较好。动态方式不会占用额外的 `RW/ZI` 空间，占用空间小，但是运行时需要动态分配内存，效率没有静态方式高。

总的来说，这两种方式就是空间和时间效率的平衡，可以根据实际环境需求选择采用具体的分配方式。

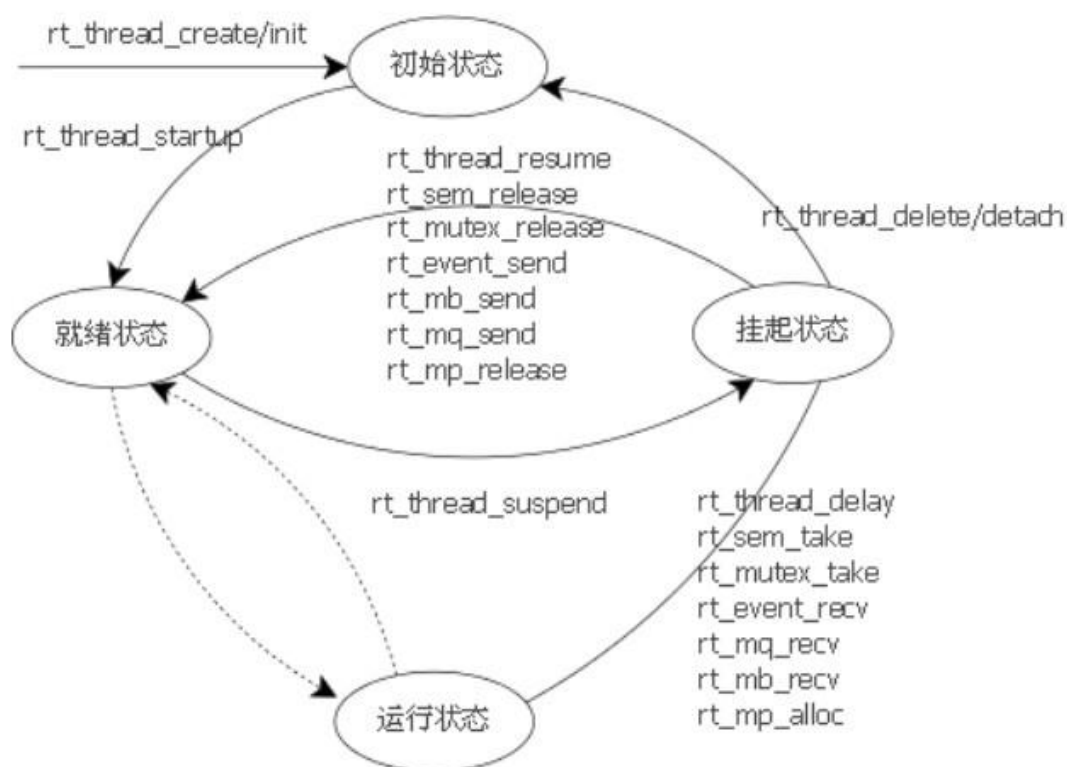
第五篇 RTT-线程调度、管理

日期：2013-04-10

◇ 线程状态

RT-Thread 中的线程有 4 种状态：初始态、就绪态、运行态、挂起态，RT-Thread 提供了一系列的 API 函数，使得各个线程可以在这 4 个状态之间来回迁变。

这几种状态间的转换关系如下图所示：



线程通过调用系统函数 `rt_thread_create/rt_thread_init` 进入初始状态，通过调用系统函数 `rt_thread_startup` 变为就绪状态，当这个线程调用 `rt_thread_delay`、`rt_sem_take`、`rt_mbrece` 等系统函数时，将主动挂起或由于获取不到资源进入到挂起状态。挂起状态的线程，如果它等待超时依然未获得资源或由于其他线程释放了资源，它将返回到就绪状态。当系统中有比当前运行态线程优先级还要高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占 CPU 进入运行态。

◇ 线程优先级、系统时钟

RT-Thread 共支持 256 个优先级(0-255，数值越小的优先级越高，0 为最高优先级，255 分配给空闲线程使用；线程总数不受限制，只和能提供给系统的 ram 有关)，我们一般通过在 `rt_config.h` 配置文件中将系统配置为 32 个优先级：

```
/* PRIORITY_MAX */
#define RT_THREAD_PRIORITY_MAX 32
```

每一个操作系统中都存在一个“系统心跳”时钟，这个时钟负责系统和时间相关的一些操作，另外系统的任务调度也会由它来驱动。系统心跳时钟一般是由硬件定时器的定时中断产生，这个定时间隔我们称之为“时间片”或叫系统滴答，系统滴答的频率我们根据 cpu 的处理能力来决定，一般我们设置为 10HZ-100HZ。系统滴答频率越快，系统的负荷就越大；频率越小，时间处理精度由不够，我们在 stm32 平台上一一般设置系统滴答为 100HZ，即每个时间片是 10ms，此值我们可以在 rt_config.h 中自行设置（[建议使用默认值](#)）：

```
/* Tick per Second */
#define RT_TICK_PER_SECOND 100//1 秒100 次即10ms 一次
```

RT-Thread 中配置系统时钟的代码如下：

```
SystemInit();
/* NVIC Configuration */
NVIC_Configuration();

/* 一上电就尽快配置系统时钟 */
SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );
```

✧ 空闲线程

空闲线程是系统线程中一个比较特殊的线程，它具备最低的优先级，当系统中无其他线程可运行时，调度器将调度到空闲线程。空闲线程通常是一个死循环，永远不被挂起。在 RT-Thread 实时操作系统中空闲线程提供了钩子函数，可以让系统在空闲的时候执行一定任务，例如系统运行指示灯闪烁，CPU 使用率统计等等。另外空闲线程还负责一些系统资源回收。

PS：我们可以将空闲线程看做是我们裸机编程时候的一个空循环，比如：

```
int main(void)
{
    /*伪代码*/
    system_init();
    /*下面部分可以理解为 RTOS 中的空闲线程*/
    /*不能直接运行到return 啊闲着也要空跑!*/
    while(1)
    {
        ;
    }
    return 0;
}
```


✧ 线程调度规则

RT-Thread 系统中，不同优先级的线程根据优先级顺序进行调度；相同优先级的线程根据时间片来调度。

RT-thread 的线程调度思想是：“近似地”每时每刻让优先级最高的就绪线程处于运行状态，它在系统或用户调用系统函数和执行中断服务程序结束时来进行系统调度，以确定应该运行的任务并运行它。当线程 A 和线程 B 的优先级相同时，系统在线程 A 运行完所分配的时间片后切换到线程 B 运行。

```
/* 堆栈大小 512 bytes , 优先级 20 , 时间片 5 个系统滴答 */
test_thread1 = rt_thread_create("thread1",
                                thread1_entry, RT_NULL,
                                512, 20, 5);

/* 堆栈大小 512 bytes , 优先级 18 , 时间片 5 个系统滴答 */
test_thread2 = rt_thread_create("thread2",
                                thread2_entry, RT_NULL,
                                512, 18, 5);
```

上面代码中的 20、18 既是线程的优先级，5 是为线程所分配的时间片。这里需要注意的是，当一个线程的优先级独一无二的时候，它的时间片这个参数将失去作用，我们不要认为上面的两个线程运行完 5 个系统 ticks 后就会主动交出 cpu 使用权，当运行完 5 个 ticks 后如果它不需等待任何资源，也不主动让出 cpu 使用权的话，它还会继续运行，时间片这个参数只在具有相同优先级的线程之间起作用，可是即便如此，这个参数也不能设为 0，因为你不知道后续是否还会创建线程。（PS：哎，真绕口，不知道我这么说，大家能否明白我在说啥？）

由于我们的线程一般都是一个无限循环，而 RT-Thread 又是一个抢占式的内核，所以为了使高优先级的线程不至于独占 CPU，可以给其他优先级较低的线程获得 CPU 使用权的机会，我们往往需要在线程中的合适位置调用系统函数 `rt_thread_delay()`，使当前线程的运行延时一段时间并进行一次任务调度，以让出 CPU 的使用权。示意代码如下：

```
void test_thread_entry(void* parameter)
{
    .
    while(1)
    {
        .
        rt_thread_delay( x ); /* 等待一会，让出 cpu 权限，让其他线程也跑一跑 */
        .
    }
}
```

举一个不太恰当的例子来说明一下系统的调度：

幼儿园里有很多小孩子（N 个线程），大家都想玩一个皮球（CPU），可是这些小孩家庭背景不太一样（优先级不同），幼儿园的老师是个势力的家伙（系统的调度机制确定），她先让家庭背景强硬（优先级最高）的小孩去玩这个皮球（优先获得 CPU 使用权），其他小孩只能眼巴巴的看着，哈喇子都流出来了也没用（线程处于挂起状态），当这个小孩玩腻了或累了或发了慈悲不玩了（线程等待资源或主动让出 CPU），其他的小孩才能有机会玩，当然这也要看剩下的孩子哄哪个孩子家庭背景 NB，当遇到两个小孩的家庭背景相当（优先级相同）的时候，那么他们两个就轮流着玩，你玩 10 分钟，我玩 10 分钟（按时间片调度）。

PS：貌似这个例子举得很失败。

第六篇实例解析 RT-Thread 线程调度

日期：2013-04-13

✧ 线程基本管理

在《篇 6-例程 1-线程的基本管理》中我们建立两个测试线程，分别打印其运行信息，如果这两个线程都挂起的话则系统自动运行空闲线程。

线程源码如下：

```
void test1_thread_entry(void* parameter)
{
    rt_uint32_t i;
    /* 无限循环*/
    while(1)
    {
        for(i =0; i<10; i++)
        {
            rt_kprintf(" %d \r\n", i);
            /* 等待 1s, 让出 cpu 权限, 切换到其他线程 */
            rt_thread_delay(100);
        }
    }
}

void test2_thread_entry(void* parameter)
{
    rt_uint32_t i=0;
    /* 无限循环*/
    while(1)
    {
        rt_kprintf(" test2 thread count:%d \r\n",++i);
        /* 等待 0.5s, 让出 cpu 权限, 切换到其他线程 */
        rt_thread_delay(50);
    }
}

void demo_thread_creat(void)
{
    rt_err_t result;

    /* 创建静态线程: 优先级 15 , 时间片 10 个系统滴答 */
    result = rt_thread_init(&thread_test1,
                            "test1",
```

```
        test1_thread_entry, RT_NULL,
        (rt_uint8_t*)&thread1_stack[0],
        sizeof(thread1_stack),15,10);

    if(result == RT_EOK)
    {
        rt_thread_startup(&thread_test1);
    }

/* 创建静态线程: 优先级 16 , 时间片 25 个系统滴答 */
    result = rt_thread_init(&thread_test2,
        "test2",
        test2_thread_entry, RT_NULL,
        (rt_uint8_t*)&thread2_stack[0],
        sizeof(thread2_stack),16,25);

    if(result == RT_EOK)
    {
        rt_thread_startup(&thread_test2);
    }
}
```

编译后烧录进开发板，设置好串口调试助手，然后运行，我们会看到如下运行信息：



程序运行分析：

- 1、首先系统调度 test1 线程投入运行，打印第 0 次运行的信息，然后通过延时函数将自己挂起 100 个时间片，系统将 test2 线程调度运行；
- 2、test2 线程打印第 0 次运行信息，然后通过延时函数将自己挂起 50 个时间片；
- 3、系统中无任务运行，系统将空闲线程调入运行；
- 4、50 个时间片后 test2 线程被唤醒，打印第 1 次运行的信息，再继续通过延时函数将自己挂起 50 个时间片；
- 5、系统中无任务运行，系统将空闲线程调入运行；
- 6、50 个时间片时间到，test1 线程被唤醒，打印第 1 次运行信息，继续挂起 100 个时间片；
- 7、test2 线程被唤醒，打印第 2 次运行的信息，再继续通过延时函数将自己挂起 50 个时间片；
- 8、系统中无任务运行，系统将空闲线程调入运行；

9、50 个时间片后 test2 线程被唤醒，打印第 3 次运行的信息，再继续通过延时函数将自己挂起 50 个时间片；

10、循环执行 5-9 的过程。

为了演示 test1 线程、test2 线程之间的切换，我们需在程序中屏蔽掉 finsh 组件（finsh 组件会建立 shell 线程）：

```
/* SECTION: finsh, a C-Express shell */
//#define RT_USING_FINSH
```

✧ 相同优先级线程的调度

我们在《篇 6-例程 2-相同优先级线程轮询调度》中建立两个相同优先级的线程用来演示一下相同优先级线程的时间片调度。

程序代码如下：

```
void test1_thread_entry(void* parameter)
{
    rt_uint8_t i;

    for(i = 0; i < 6; i++)
    {
        rt_kprintf("Thread1: \r\n");
        rt_kprintf("This is a demo for same priority !\r\n");
        rt_thread_delay(4);
    }
}

void test2_thread_entry(void* parameter)
{
    rt_uint8_t j;

    for(j = 0; j < 60; j++)
    {
        rt_kprintf("Thread2: \r\n");
        rt_kprintf("This is a demo for same priority !\r\n");
    }
}

void demo_thread_creat(void)
{
    rt_err_t result;
```

```

/* 创建静态线程：优先级 15，时间片 2 个系统滴答 */
result = rt_thread_init(&thread_test1,
                        "test1",
                        test1_thread_entry, RT_NULL,
                        (rt_uint8_t*)&thread1_stack[0],
                        sizeof(thread1_stack), 15, 2);

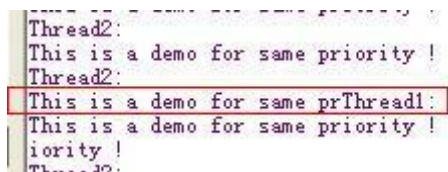
if(result == RT_EOK)
{
    rt_thread_startup(&thread_test1);
}

/* 创建静态线程：优先级 15，时间片 1 个系统滴答 */
result = rt_thread_init(&thread_test2,
                        "test2",
                        test2_thread_entry, RT_NULL,
                        (rt_uint8_t*)&thread2_stack[0],
                        sizeof(thread2_stack), 15, 1);

if(result == RT_EOK)
{
    rt_thread_startup(&thread_test2);
}
}

```

编译、下载到板子后运行，通过观察串口输出，我们会看到如下图中红色框中的信息，



```

Thread2: This is a demo for same priority !
Thread2: This is a demo for same prThread1:
This is a demo for same priority !
iority !
Thread2:

```

线程 test2 的打印信息输出不全，说明 test2 线程的执行被打断了，因为 test1 线程和 test2 线程的优先级都是 15，并不会发生抢占的情况，所以说 test2 线程是等到自己的执行时间片到达时，被系统剥夺了 CPU 使用权，而将使用权交给了 test1 线程，从而 test1 线程重新获得执行。由此可以看出当两个相同线程间，运行是以时间片为基准的，时间片到达，则交出 CPU 使用权，交给下一个就绪的同优先级线程执行。

PS：由于上面的两个线程都不是无限循环结构，在其正常退出后，其线程状态变为初始化状态，然后在空闲线程中将其从线程调度列表中删除。

本例程同样需要关闭 finsh 组件。

✧ 线程的让出

前面两个例子演示的线程调度是由系统“主动干预”的情况的线程切换，其实我们也可以根据实际情况，采用主动让出 CPU 使用权。

RT-Thread 中的系统函数：rt_thread_yield()，可以让调用它的线程暂时让出 CPU 的使用权，而使下一个最高优先级的线程得以运行，但这时调用 rt_thread_yield()的线程还保持的是就绪态。这和“孔融让梨”有点像：这个梨我不吃，下一个梨我可就不客气了。

下面我们用《篇 6-例程 3-线程的让出》这个例子来演示一下线程的让出，主要代码如下：

```
void test1_thread_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while(1)
    {
        /* 打印线程 1 的输出 */
        rt_kprintf("thread1: count = %d\n", count ++);

        /* 执行 yield 后应该切换到 test2 执行 */
        rt_thread_yield();
    }
}

void test2_thread_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while(1)
    {
        /* 执行 yield 后应该切换到 test1 执行 */
        rt_thread_yield();

        /* 打印线程 2 的输出 */
        rt_kprintf("thread2: count = %d\n", count ++);
    }
}

void demo_thread_creat(void)
{
    rt_err_t result;

    /* 创建静态线程：优先级 15，时间片 5 个系统滴答 */
```



```

result = rt_thread_init(&thread_test1,
                        "test1",
                        test1_thread_entry, RT_NULL,
                        (rt_uint8_t*)&thread1_stack[0],
                        sizeof(thread1_stack),15,5);

if(result == RT_EOK)
{
    rt_thread_startup(&thread_test1);
}

/* 创建静态线程：优先级 15，时间片 5 个系统滴答 */
result = rt_thread_init(&thread_test2,
                        "test2",
                        test2_thread_entry, RT_NULL,
                        (rt_uint8_t*)&thread2_stack[0],
                        sizeof(thread2_stack),15,5);

if(result == RT_EOK)
{
    rt_thread_startup(&thread_test2);
}
}

```

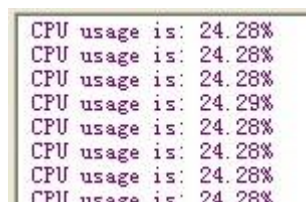
✧ 使用空闲线程统计 CPU 使用率

CPU 的使用率一般是我们比较关心的问题，在这里我们就用空闲线程的钩子函数去统计 CPU 的使用率，并通过串口打印出来。

首先我们在初始化线程中设置好钩子函数，并在 LED 线程中给系统人为的加入很多“事情”，让其占用率变高，方便统计，然后每隔 1 秒打印一次 CPU 使用率。

代码详见《篇 6-例程 4-使用空闲线程统计 CPU 使用率》。其中 CPU 的使用率我们是根据 CPU 的空闲率反推出来的。

编译、烧录、运行，我们会得到如下的运行信息：



```

CPU usage is: 24.28%
CPU usage is: 24.28%
CPU usage is: 24.28%
CPU usage is: 24.29%
CPU usage is: 24.28%
CPU usage is: 24.28%
CPU usage is: 24.28%
CPU usage is: 24.28%

```

前面说过，系统的心跳时钟过快，会增加 cpu 的负担，我们可以在这里来验证，读者可以将系统滴答时间改为 1ms，然后你将会发现 cpu 的使用率从 24% 升高到了 87% !!，如下图：

```

CPU usage is: 87.7%
CPU usage is: 87.7%
CPU usage is: 86.93%
CPU usage is: 87.7%
CPU usage is: 86.93%
CPU usage is: 86.93%
CPU usage is: 86.93%
CPU usage is: 87.7%
CPU usage is: 86.93%
CPU usage is: 86.93%

```

✧ 多线程导致的临界区问题

临界资源是指一次仅允许一个线程使用的共享资源。不论是硬件临界资源，还是软件临界资源，多个线程必须互斥地对它们进行访问。每个线程中访问临界资源的那段代码称为临界区（Critical Section），每次只准许一个线程进入临界区，进入后不允许其他线程进入。多线程程序的开发方式不同于裸机程序，多个线程在宏观上是并发运行的，因此使用一个共享资源是需要注意，否则就可能出现错误的运行结果。

下面我们通过一个简单全局变量来演示多线程中的临界区问题，在《篇 6 例程 5-多线程导致的临界区问题》中我们建立两个不同优先级的线程：

```

void test1_thread_entry(void* parameter)
{
    rt_uint32_t i;

    g_tmp = 0;
    rt_kprintf("g_tmp=%d \r\n", g_tmp);
    for(i=0; i<10000; i++)//100000
    {
        g_tmp++;
    }
    rt_kprintf("g_tmp=%d \r\n", g_tmp);
}

void test2_thread_entry(void* parameter)
{
    rt_thread_delay(100);// 1
    g_tmp++;
}

void demo_thread_creat(void)
{
    rt_err_t result;

    /* 创建静态线程：优先级 16，时间片 2 个系统滴答 */
    result = rt_thread_init(&thread_test1,
                           "test1",
                           test1_thread_entry, RT_NULL,

```

```

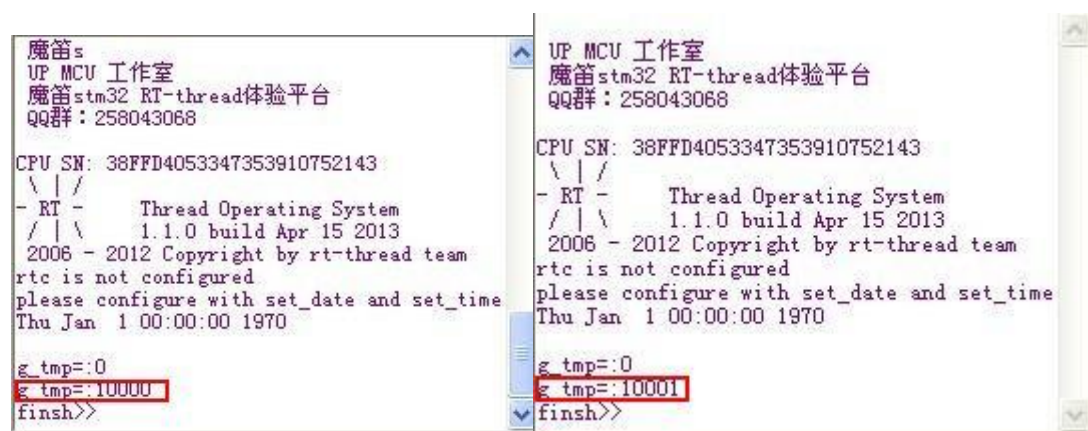
                                (rt_uint8_t*)&thread1_stack[0],
                                sizeof(thread1_stack),16,2);

if(result == RT_EOK)
{
    rt_thread_startup(&thread_test1);
}
/* 创建静态线程: 优先级 15, 时间片 1 个系统滴答 */
result = rt_thread_init(&thread_test2,
                        "test2",
                        test2_thread_entry, RT_NULL,
                        (rt_uint8_t*)&thread2_stack[0],
                        sizeof(thread2_stack),15,1);

if(result == RT_EOK)
{
    rt_thread_startup(&thread_test2);
}
}

```

下载后运行，我们会看到如左图的运行结果，当我们修改线程 test2 中的延时参数为 1 后再次编译、下载、运行，我们会看到如右图的运行结果。



我们心目中的期望结果是在：test1 线程最后输出 10000，但是第二次的结果为何输出了 10001，这是违背了我们的期望的结果，为什么呢？

结果分析：

在 test1 线程的 for 循环中我们对 i 做了 10000 次累加，如果没有其他线程的“干预”，那么全局变量 g_tmp 的值应该是 10000，现在的输出结果是 10001，这意味全局变量 g_tmp 的值被线程 2 修改过。

整个程序运行过程中各个线程的状态变化是这样的：rt_application_init 中创建两个线程之后，由于 test2 线程的优先级比 test1 线程的优先级高，因此 test2 线程先运行，其线程处

理函数第一句为 `rt_thread_delay(1)`，这会使得 `test2` 线程被挂起，挂起时间为 1 个时间片，在 `test2` 线程挂起的这段时间中，`test1` 线程是所有就绪态线程中优先级最高的线程，因此被内核调度运行。在 `test1` 线程执行了一部分代码后，1 个 tick 时间到，`test2` 线程被唤醒，从而成为所有就绪线程中优先级最高的线程，因此会被立刻调度运行，`test1` 线程被 `test2` 线程抢占，`test2` 线程中对全局变量 `g_tmp` 做累加操作，接下来 `test2` 线程执行完毕，`test1` 线程再次被调度运行，根据程序的运行结果可以看出，此时 `test1` 线程继续执行，但是我们并不知道此时 `test1` 线程大致是从什么地方在开始执行的，从最后的输出结果来看，只能得知此时 `test1` 线程还没有执行到第二条 `rt_kprintf` 输出语句。最后 `test1` 线程再次打印全局变量 `g_tmp` 的值，其值就应该是 10001。

当 `test2` 线程中的第一句为 `rt_thread_delay(100)` 的时候，在 `test2` 线程休眠的整个时间里，`test1` 线程都已经执行完毕，因此最后的输出结果为 10000。

从以上可以看到：当公共资源在多个线程中公用时，如果缺乏必要的保护错误，最后的输出结果可能与预期的结果完全不同。为了解决这种问题，需要引入线程间通信机制，这就是所谓的 IPC 机制（Inter-Process Communication）。

第七篇线程间同步和通信

日期: 2013-04-20

✧ 禁止系统调度

上一节《篇 6-例程 5-多线程导致的临界区问题》中由于 test1 线程被 test2 线程打断，才导致了我们没有得到预期的结果，我们一般可通过[关闭中断](#)和[调度器上锁](#)这两种简单的途径来禁止系统调度，防止线程被打断，从而保证临界区不被破坏。

1、关闭中断

线程中关闭中断保护临界区的结构如下：

```
void test1_thread_entry(void* parameter)
{
    rt_base_t level;

    while(1)
    {
        /* 关闭中断*/
        level = rt_hw_interrupt_disable();
        /* 以下是临界区*/
        . . . .
        /* 关闭中断*/
        rt_hw_interrupt_enable(level);
    }
}
```

所有线程的调度都是建立在中断的基础上的，拿 CM3 核来举例：

在 cm3 处理器上，所有的调度条件满足后(不管是在任务还是在中断中)系统会触发 pendsv 中断，在 pendsv 中断中去执行调度工作。

所以，当我们关闭中断后，系统将不能再进行调度，线程自身也自然不会被其他线程抢占了。

2、调度器上锁

锁住调度器以保护临界区的结构如下：

```
void test1_thread_entry(void* parameter)
{
    .
    .
    .
}
```

```

while(1)
{
    /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
    rt_enter_critical();
    /* 以下进入临界区 */
    . . .
    /* 调度器解锁 */
    rt_exit_critical();
}
}

```

把调度器锁住也能让当前运行的任务不被换出，直到调度器解锁。但和关闭中断有一点不相同的是，对调度器上锁，系统依然能响应外部中断，中断服务例程依然有可能被运行。所以在使用调度器上锁的方式来做任务同步时，需要考虑好，[任务访问的临界资源是否会被中断服务例程所修改](#)，如果可能会被修改，那么将不适合采用此种方式作为同步的方法。

PS 上面两种方法的的宗旨其实就是：在临界区内只允许一个线程运行！

我们用锁定调度器的方法来解决上一节中的问题，源码见《篇 7-例程 1-调度器上锁》。编译、烧录、运行，可看到串口输出如下，说明临界区受到了保护

```

CPU SN: 38FFD4053347353910752143
\ | /
- RT -   Thread Operating System
/ | \   1.1.0 build Apr 21 2013
2006 - 2012 Copyright by rt-thread team
g tmp=:0
g tmp=:10000
finsh>>

```

除了禁止调度器调度，我们还用线程间通信的方式来保证线程间的同步，下面我们来介绍 RT-Thread 中的 IPC 对象：信号量、互斥锁、事件、消息队列、邮箱。

✧ 信号量的基本操作

我们先以一个停车场的运作为例来描述一下信号量的概念，简单起见，假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆直接进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复。

在这个停车场系统中，车位是公共资源，每辆车好比一个线程，[看门人起的就是信号量的作用](#)。

抽象的来讲，信号量的特性如下：信号量是一个非负整数（车位数），所有通过它的线程/进程（车辆）都会将该整数减一（通过它当然是为了使用资源），当该整数值为 0 时，所有试图通过它的线程都将处于等待状态。在信号量上我们定义两种操作：take（获取）和

Release（释放）。当一个线程调用 take 操作时，它要么得到资源然后将信号量减一，要么一直等下去（指放入阻塞队列），直到信号量大于等于一时。Release（释放）实际上是在信号量上执行加操作，take（获取）实际上是在信号量上执行减操作。

RT-Thread 中的信号量有静态和动态之分（同静态线程、动态线程），和信号量有关的操作如下：

初始化—rt_sem_init()（对应静态信号量）；

建立—rt_sem_create()（对应动态信号量）；

获取—rt_sem_take()；

释放—rt_sem_release()；

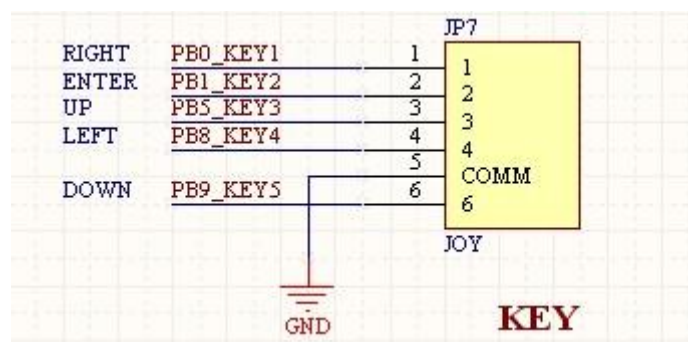
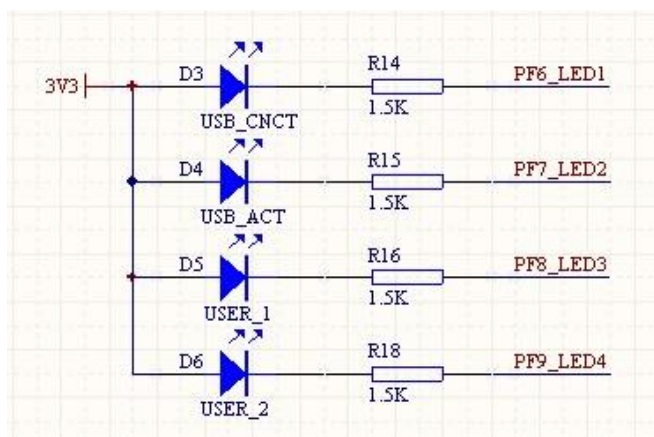
脱离—rt_sem_detach()（对应静态信号量）；

删除—rt_sem_delete()（对应动态信号量）；

在例程《篇 7-例程 2-信号量的基本使用》中演示了静态、动态信号量的各种操作。例程的运行输出如下：



✧ 信号量实际运用—按键点灯



上面是魔笛开发板上 LED 和按键的 IO 分布, 我们通过信号量的方法来同步按键线程和 LED 线程, 实现当 enter 键按下后, 点亮或关闭 LED 的动作。

在例程《篇 7-例程 3-信号量实际使用-按键点灯》中, 首先我们先初始化一个信号量:

```
/* 初始化静态信号量, 初始值是 0 */
result =rt_sem_init(&key_sem, "keysem", 0, RT_IPC_FLAG_FIFO);
if(result != RT_EOK)
{
    rt_kprintf("init keysem failed.\n");
    return-1;
}
```

然后, 按键检测线程中, 当检测到五向按键中的 enter 按下时, 去释放一次信号量:

```
case KEY_ALL_NSET &(~KEY_SET_PIN)://set
/* 释放一次信号量 */
rt_sem_release(&key_sem);
break;
```

最后, 当 led 线程得到信号量后, 会点亮或熄灭 led:

```
while(1)
{
    /* 以永久等待方式获取信号量*/
```



```

rt_sem_take(&key_sem, RT_WAITING_FOREVER);
/* 当得到信号量以后才有可能执行下面程序*/
led_state ^=1;
if(led_state!=0)
{
    GPIO_ResetBits(led1_gpio,led1_pin|led2_pin|led3_pin|led4_pin);
    rt_kprintf(" get semaphore ok, led all on \r\n");
}
else
{
    GPIO_SetBits(led1_gpio,led1_pin|led2_pin|led3_pin|led4_pin);
    rt_kprintf(" get semaphore ok, led all off \r\n");
}
}

```

在开发板上实际运行程序后，我们看到，信号量确实起到了按键线程和 led 线程之间的同步作用：只有当按键按下后，led 灯才会有动作。

✧ 互斥锁

互斥锁和信号量很相似，RT-Thread 中的互斥锁也有静态和动态之分，和互斥锁有关的操作如下：

初始化—rt_mutex_init()（对应静态互斥锁）；

建立—rt_mutex_create()（对应动态互斥锁）；

获取—rt_mutex_take()；

释放—rt_mutex_release()；

脱离—rt_mutex_detach()（对应静态信号量）；

删除—rt_mutex_delete()（对应动态信号量）；

例程《篇 7-例程 4-互斥锁》中演示了互斥锁的基本使用。

我们看到信号量和互斥锁如此形似，那么它们的区别在哪里？我以我的理解，区别一下这两个 IPC 对象：

1、信号量哪里都可以释放，但互斥锁只有获得了其控制权的线程才可以释放，即：只有“锁上”它的那个线程才有“钥匙”打开它，有“所有权”的概念。

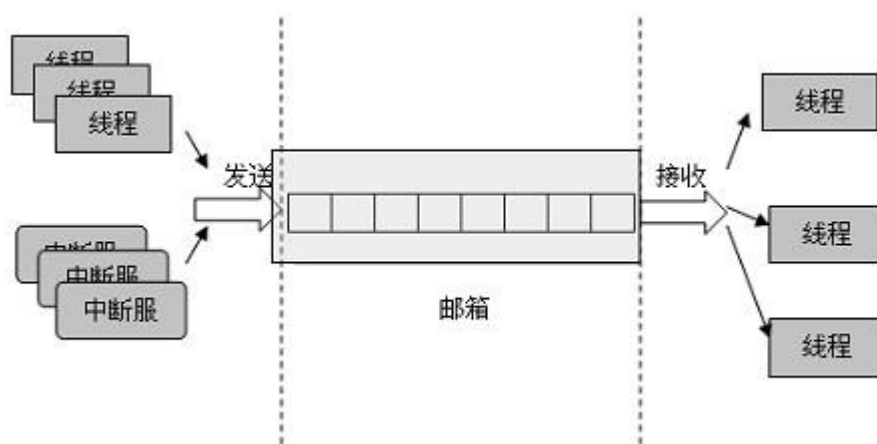
2、信号量可能导致线程优先级反转，而互斥锁可通过优先级继承的方法解决优先级反转问题（详见《RT-Thread 编程指南》）。

实际中我们常遇到这样的情况，比如一个总线上挂接着 N 个设备，这是我们必须“分时”的去操作各个设备，这时候互斥锁就派上用场了：我们在开始操作某个设备前先使用 `rt_mutex_take()` 锁住总线，然后开始对设备的具体操作，最后通过 `rt_mutex_release()`，解锁总线，让给其他设备去使用。

信号量用于同步的时候就像交通灯，任务只有在获得许可的时候才可以运行，强调的是运行步骤；信号量用于互斥的时候就像一把钥匙，它强调只有获得钥匙的任务才可以运行，强调的是许可和权限。这两者都不具备任何数据交换的功能，下面来介绍具有数据交换功能的 IPC 对象：邮箱和消息队列。

✧ 邮箱

邮箱服务是实时操作系统中一种典型的任务间通信方法，通常开销比较低，效率较高，每一封邮件只能容纳固定的 4 字节内容（针对 32 位处理系统，刚好能够容纳一个指针）。典型的邮箱也称作交换消息，如下图所示，线程或中断服务例程把一则 4 字节长度（典型的是一个指针）的邮件发送到邮箱中。而一个或多个线程可以从邮箱中接收这些邮件进行处理。



RT-Thread 采用的邮箱通信机制有点类似传统意义上的管道，用于线程间通讯。它是线程、中断服务、定时器向线程发送消息的有效手段。邮箱与线程对象等之间是相互独立的。线程，**中断服务和定时器都可以向邮箱发送消息，但是只有线程能够接收消息**（因为当邮箱为空时，线程将有可能被挂起）。RT-Thread 的邮箱中共可存放固定条数的邮件，邮箱容量在创建邮箱时设定，**每个邮件大小为 4 字节**，正好是一个指针的大小。当需要在线程间传递比较大的消息时，可以传递指向一个缓冲区的指针。当邮箱满时，线程等不再发送新邮件，返回-RT_EFULL。当邮箱空时，将可能挂起正在试图接收邮件的线程，使其等待，当邮箱中有新邮件时，再唤醒等待在邮箱上的线程，使其能够接收新邮件并继续后续的处理。

RT-Thread 中的邮箱依然是有静态和动态之分，和邮箱有关的操作如下：

初始化—`rt_mb_init()`（对应静态邮箱）；

建立—`rt_mb_create()`（对应动态邮箱）；

发送邮件—`rt_mb_send()`;

接收邮件—`rt_mb_recv()`;

脱离—`rt_mb_detach()`（对应静态邮箱）;

删除—`rt_mb_delete()`（对应动态邮箱）;

我们用《篇 7-例程 5-邮箱》来演示 RT-Thread 中邮箱的基本使用：建立按键线程和 led 线程，按键线程在不同的按键按下时向邮箱中发送不同的“邮件”，LED 线程在收到“邮件”后根据邮件的内容来决定点亮哪个灯。

首先，初始化一个邮箱，并建立 key 和 led 线程：

```
rt_err_t demo_thread_creat(void)
{
    rt_err_t result;

    /* 初始化一个静态邮箱 */
    result = rt_mb_init(&mb,
                       "mbt", /* 名称是 mbt */
                       &mb_pool[0], /* 邮箱用到的内存池是 mb_pool */
                       sizeof(mb_pool)/4, /* 邮箱中的邮件数目，因为一封邮件占 4 字节 */
                       RT_IPC_FLAG_FIFO); /* 采用 FIFO 方式进行线程等待 */

    if(result != RT_EOK)
    {
        rt_kprintf("init mailbox failed.\n");
        return -1;
    }

    /* 创建 led 线程：优先级 16，时间片 5 个系统滴答 */
    result = rt_thread_init(&led_thread,
                           "led",
                           led_thread_entry, RT_NULL,
                           (rt_uint8_t*)&led_stack[0],
                           sizeof(led_stack), 16, 5);

    if(result == RT_EOK)
    {
        rt_thread_startup(&led_thread);
    }

    /* 创建 key 线程：优先级 15，时间片 5 个系统滴答 */
    result = rt_thread_init(&key_thread,
                           "key",
```

```

        key_thread_entry, RT_NULL,
        (rt_uint8_t*)&key_stack[0],
        sizeof(key_stack),15,5);

if(result == RT_EOK)
{
    rt_thread_startup(&key_thread);
}
return 0;
}

```

在 key 线程中发送邮件:

```

void key_scan(void)
{
    static vu16 s_KeyCode; // 内部检查按键使用
    static vu8 s_key_debounce_count, s_key_long_count;
    vu16 t_key_code;

    t_key_code = GPIO_ReadInputData(KEY_PORT) & KEY_ALL_NSET;

    if((t_key_code == KEY_ALL_NSET) || (t_key_code != s_KeyCode))
    {
        s_key_debounce_count = 0; // 第一次
        s_key_long_count = 0;
    }
    else
    {
        if(++s_key_debounce_count == DEBOUNCE_SHORT_TIME)
        { // 短按键
            switch(s_KeyCode)
            {
                {
                    case KEY_ALL_NSET & (~KEY_UP_PIN): // up
                        key_info[4] = '1';
                        break;

                    case KEY_ALL_NSET & (~KEY_DOWN_PIN): // down
                        key_info[4] = '2';
                        break;

                    case KEY_ALL_NSET & (~KEY_LEFT_PIN): // left
                        key_info[4] = '3';
                        break;

                    case KEY_ALL_NSET & (~KEY_RIGHT_PIN): // right

```

```

        key_info[4]='4';
        break;

    case KEY_ALL_NSET &(~KEY_SET_PIN)://set
        key_info[4]='5';
        break;

    default://其他组合不做处理
        break;

}
/* 这里我们向邮箱中发一个指针*/
rt_mb_send(&mb,(rt_uint32_t )&key_info[0]);

```

led 线程中，接收邮件，并根据邮件内容，点亮 led:

```

void led_thread_entry(void* paramete)
{
    char* str;
    vu8 led_state =0;

    rt_hw_led_init();
    /* 无限循环*/
    while(1)
    {
        if(rt_mb_recv(&mb,(rt_uint32_t*)&str, RT_WAITING_FOREVER)== RT_EOK)
        {
            rt_kprintf("led: get a mail from mailbox, the content:%s\n", str);
            /*判断邮件内容*/
            if(str[4]=='1') led_state =0x01;
            if(str[4]=='2') led_state =0x02;
            if(str[4]=='3') led_state =0x03;
            if(str[4]=='4') led_state =0x04;

            if(led_state==1)
            {
                GPIO_ResetBits(led1_gpio,led1_pin);
            }
            else
            {
                GPIO_SetBits(led1_gpio,led1_pin);
            }
            if(led_state==2)
            {
                GPIO_ResetBits(led2_gpio,led2_pin);
            }

```

串口调试器 COMPort Debugger V2.00

初始化

端口号 COM4

波特率 115200

数据位 8

停止位 1

校验位 None

自动发送: 间隔 1000 ms 发送(S) 停止(T)

按16进制显示或发送 清空内容 读入文件

关闭串口(C) OK

计数

发送 0 清空计数

接收 4602

线路状态

☐ DTR ☐ CTS

☐ RTS ☐ DSR

☐ BREAK ☐ RING

☐ DCD

选项(O) 退出(X)

UP MCU 工作室
魔笛stm32 RT-thread体验平台
QQ群: 258043068

CPU SN: 38FFD4053347353910752143

\ | /
- RT - Thread Operating System
/ | \ 1.1.0 build Apr 21 2013
2006 - 2012 Copyright by rt-thread team
Sun Apr 28 12:23:24 2013

finsh>>led: get a mail from mailbox, the content:key 3 has been pressed!
led: get a mail from mailbox, the content:key 1 has been pressed!
led: get a mail from mailbox, the content:key 2 has been pressed!
led: get a mail from mailbox, the content:key 3 has been pressed!
led: get a mail from mailbox, the content:key 4 has been pressed!
led: get a mail from mailbox, the content:key 5 has been pressed!

按16进制显示 暂停显示 清空内容 保存为

当我们按按键时，亮灯的规律如下：

当按下魔笛开发板上上键时，第 1 个 led 会亮起；

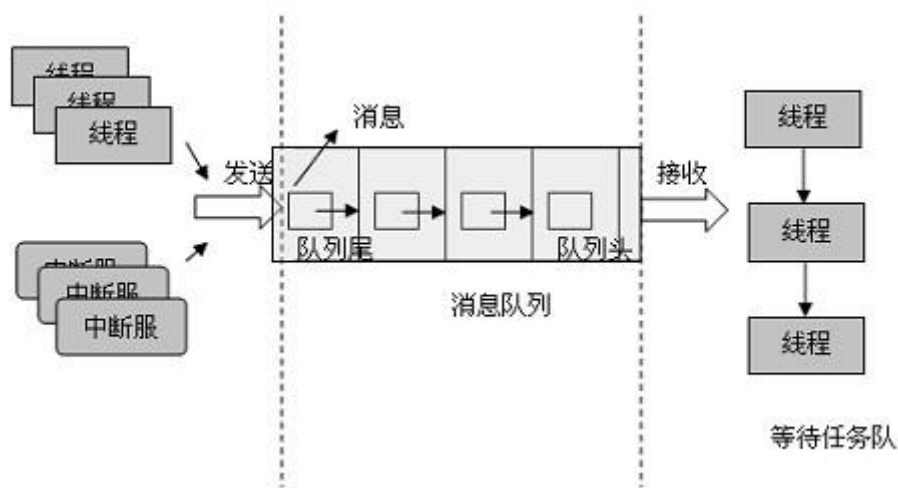
当按下魔笛开发板上下键时，第 2 个 led 会亮起；

当按下魔笛开发板上左键时，第 3 个 led 会亮起；

当按下魔笛开发板上右键时，第 4 个 led 会亮起。

✧ 消息队列

消息队列是另一种常用的线程间通讯方式，它能够接收来自线程的不固定长度的消息（而邮件只能容纳固定的 4 字节内容），并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。而当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。



消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。RT-Thread 的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：队列名，内存缓冲区，消息大小以及队列长度等。同时每个消息队列对象中包含着多个消息框，每个消息框可以存放一条消息。消息队列中的第一个和最后一个消息框被分别称为队首和队尾，对应于消息队列控制块中的 `msg_queue_head` 和 `msg_queue_tail`。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

和之前的 IPC 对象一样，消息队列也是有静态和动态之分，和消息队列有关的操作如下：

初始化—`rt_mq_init()`（对应静态消息队列）；

建立—`rt_mq_create()`（对应动态消息队列）；

发送消息—`rt_mq_send()`;

发送紧急消息—`rt_mq_urgent()`;

接收消息—`rt_mq_recv()`;

脱离—`rt_mq_detach()`（对应静态消息队列）;

删除—`rt_mq_delete()`（对应动态消息队列）;

可以看到，消息队列和邮箱的操作大体相同，不过消息队列提供发送紧急消息的操作，当发送紧急消息时，消息块不是挂到消息队列的队尾，而是挂到队首，这样，接收者能够优先接收到紧急消息，从而及时进行消息处理。

在《篇 7-例程 6-消息队列》里，我们用消息队列的方式来替代邮箱，完成按键点灯的操作，以此熟悉消息队列的应用，其程序运行效果和上一节中相同。

Ps:例子中，在消息队列初始化时我们定义消息总大小是 128，所以发送的消息长度不能超过这个值，若要发送更长的消息，可在初始化或创建时将消息总大小设大些。

✧ IPC 对象使用注意

上面介绍的这几种 IPC 对象，在其等待获取资源时，都会造成线程阻塞（除非以非等待的方式获取）所以，**不要在中断服务程序中去尝试去执行获取**信号量、互斥锁、邮箱、消息队列的操作。

✧ 事件机制

RT-Thread 中的信号量主要用于“一对一”的线程同步，当需要“一对多”、“多对一”、“多对多”的同步时，就需要事件机制来处理了。

RT-Thread 中的事件用一个 32 位无符号整型变量来表示，变量中的一位代表一个事件，线程通过“逻辑与”或“逻辑或”与一个或多个事件建立关联形成一个事件集。

事件的“逻辑或”也称为是独立型同步，指的是线程与任何事件之一发生同步，只要有一个事件发生，即满足条件；

事件的“逻辑与”也称为是关联型同步，指的是线程与若干事件都发生同步，只有这些事件全部发生，才满足条件。

事件的相关操作如下：

初始化—`rt_event_init()`（对应静态事件）;

建立—`rt_event_create()`（对应动态事件）;

发送事件—`rt_event_send()`;

接收事件—`rt_event_recv()`;

脱离—`rt_event_detach()`（对应静态事件）;

删除—`rt_event_delete()`（对应动态事件）;

《篇 7-例程 7-事件机制》中，我们用静态事件的方式来处理按键点灯的操作，顺便了解 RTT 中事件机制的用法。（老大，都腻了，咱能不点灯吗？呵呵，演示不同的东西，管用！）

在 `test.c` 中我们先初始化一个事件：

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

rt_err_t demo_thread_creat(void)
{
    rt_err_t result;

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

在按键扫描中，根据按键值发送不同的事件：

```
switch(s_KeyCode)
{
    case KEY_ALL_NSET & (~KEY_UP_PIN): //up
        rt_event_send(&event, (1 << 0));
        rt_kprintf("key: send event0\r\n");
        break;

    case KEY_ALL_NSET & (~KEY_DOWN_PIN): //down
        rt_event_send(&event, (1 << 1));
        rt_kprintf("key: send event1\r\n");
        break;

    case KEY_ALL_NSET & (~KEY_LEFT_PIN): //left
        rt_event_send(&event, (1 << 2));
```

```

        rt_kprintf("key: send event2\r\n");
        break;

    case KEY_ALL_NSET &(~KEY_RIGHT_PIN)://right
        rt_event_send(&event,(1 << 3));
        rt_kprintf("key: send event3\r\n");
        break;

    case KEY_ALL_NSET &(~KEY_SET_PIN)://set
        rt_event_send(&event,(1 << 4));
        rt_kprintf("key: send event4\r\n");
        break;

    default://其他组合不做处理
        break;
}

```

在 led 线程中，处理这些事件：其中事件 0、事件 1 以逻辑与的方式等待，等待到以后，点亮 led1，10 个时间片内没等到则继续往下运行；事件 2、事件 3 以逻辑或的方式等待，只要有一个事件发生则点亮 led2，如果 10 个时间片内没有任何一个事件发生，则继续往下运行；等待到事件 4 后，点亮 led3。

Led 线程代码如下：

```

void led_thread_entry(void* paramete)
{
    /* 保存事件结果 */
    rt_uint32_t evt;
    vu8 led_state =0;

    rt_hw_led_init();
    /* 无限循环*/
    while(1)
    {
        /* 以逻辑与的方式接收事件 */
        if(rt_event_rcv(&event,((1 << 0)|(1 << 1)), \
            RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,\
            10,&evt)== RT_EOK)
        {
            rt_kprintf("led: AND rcv event 0x%x\r\n", evt);
            /* 点亮第1个LED*/
            led_state =0x01;
            led_on_off(led_state);
        }
        /* 以逻辑或的方式接收事件 */
    }
}

```

```

if(rt_event_recv(&event,((1 << 2)|(1 << 3)), \
    RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,\
    10,&evt)== RT_EOK)
{
    rt_kprintf("led:OR recv event 0x%x\r\n", evt);
    /* 点亮第2个LED*/
    led_state =0x02;
    led_on_off(led_state);
}
/* 以逻辑与的方式接收事件 */
if(rt_event_recv(&event,(1 << 4), \
    RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,\
    10,&evt)== RT_EOK)
{
    rt_kprintf("led:AND recv event 0x%x\r\n", evt);
    /* 点亮第3个LED*/
    led_state =0x03;
    led_on_off(led_state);
}
}
}

```

实际结果，请自行下载验证，这里就不多说了。

从上面程序和结果验证中，我们可以总结出 RT-Thread 中的事件特点：

1. 事件只与线程相关，事件间相互独立；
2. 事件仅用于同步，不提供数据传输功能；
3. 事件无排队性，即多次向线程发送同一事件(如果线程还未来得及读走)，其效果等同于只发送一次。

由于信号量一般是一对一的同步，当需要一种“广播式”的同步时，我们可以选择用事件机制来处理。

✧ 使用全局变量进行线程间通信

除了 RT-Thread 为我们提供的这么多线程间通信机制之外，还有一种最直接的线程间通信方式——全局变量，对，你没看错，就是全局变量！尽管很多人对使用全局变量很忌讳，但不得不承认，使用全局变量是一种“性价比”很高的线程间通信方式。

我们可以在 RT-Thread 中，像裸机编程那样去使用全局变量，可以在某个线程中改写全局变量，而在另一个线程中根据这个全局变量的值来进行相关选择和处理。不过我们需要考虑由于并发操作而导致的竞态问题。

Ok，先来两个名词解释：

并发：并发是指多个单元同时、并行被执行，可理解为多线程运行；

竞态：假设有一个设备，线程 A 对其写入 5 个字符'a'，而另一个线程 B 对其写入 16 个'b'，第三个线程 C 读取此设备中的所有字符，如果线程 A 、B 对于设备的写入操作同时发生，此时就会造成竞态。

由于 4 字节对齐上的整型变量在读取和写入时都是原子操作，我们可以不用考虑竞态的问题；我们也可以使用关闭中断的方式来解决使用全局变量而带来的竞态问题。

第八篇 RT-Thread 的命令行—Finsh 组件

日期: 2013-05-03

finsh shell 是 RT-Thread 的用户命令行组件，用户能够通过串口设备或 telnet、甚至是其他方式使用 finsh shell。

finsh shell 在 RT-Thread 中被设计成一个独立的线程，它试图从外部设备中获得用户的输入，然后对用户命令进行解析执行。其命令的语法格式与 C 语言的单行表达式完全相同。

对于用户而言，finsh 主要有以下功能：

- (1) 获取系统运行时信息，如各种 RT-Thread 内核对象的动态信息。
- (2) 能够对任意寄存器和内存地址进行读写操作
- (3) 能够直接在 shell 中调用系统函数，访问系统变量

总之，finsh 将极大的方便我们的调试！

✧ Finsh 的输入设备

串口是 finsh 组件的常用输入设备：串口的接收作为 finsh 的输入；串口的发送作为终端打印，可显示 finsh 命令的运行结果。RT-Thread 的 bsp 包中为我们提供了 uart 设备驱动，并向系统注册了 uart1 设备供我们使用。

我们想要正确的使用 finsh，需要一个关联过程，在基本的 RT-Thread 系统中这个过程大概是这样的：

rt_hw_board_init()函数调用串口初始化函数 rt_hw_usart_init()，此函数初始化串口，并向系统注册“uart1”设备，接着系统调用 rt_console_set_device()函数设置“uart1”作为 console 输出。

为了使用 finsh，我们在 rththread_startup()函数中调用 finsh_system_init()初始化 finsh 组件，并通过调用 finsh_set_device("uart1")，将“uart1”和 finsh 关联了起来，这样 uart1 的输入即可被 finsh 读入并分析、执行。

当然这个过程还需要正确的配置：

首先在 rt_config.h 中选择使用 finsh 组件：#define RT_USING_FINSH

并为串口驱动 (usart.c) 指定串口号，这里我们使用串口 1：#define RT_USING_UART1

这两项配置如下：

```

/* USING DEVICE SYSTEM */
#define RT_USING_DEVICE
#define RT_USING_UART1

/* SECTION: Console options */
#define RT_USING_CONSOLE
/* the buffer size of console*/
#define RT_CONSOLEBUF_SIZE 128

/* SECTION: finsh, a C-Express shell */
#define RT_USING_FINSH
/* Define symbol table */

```

由于前面的串口驱动选择了 uart1，为了使 console 设备也和 uart1 关联，我们需要在 board.h 中对 console 终端设置：

```

// <o> Console on USART: <0=> no conso
// <i>Default: 1
#define STM32_CONSOLE_USART 1

// <o> Ethernet Interface: <0=> Microc:
#define STM32_ETH_IF 1

void rt_hw_board_led_on(int n);
void rt_hw_board_led_off(int n);
void rt_hw_board_init(void);

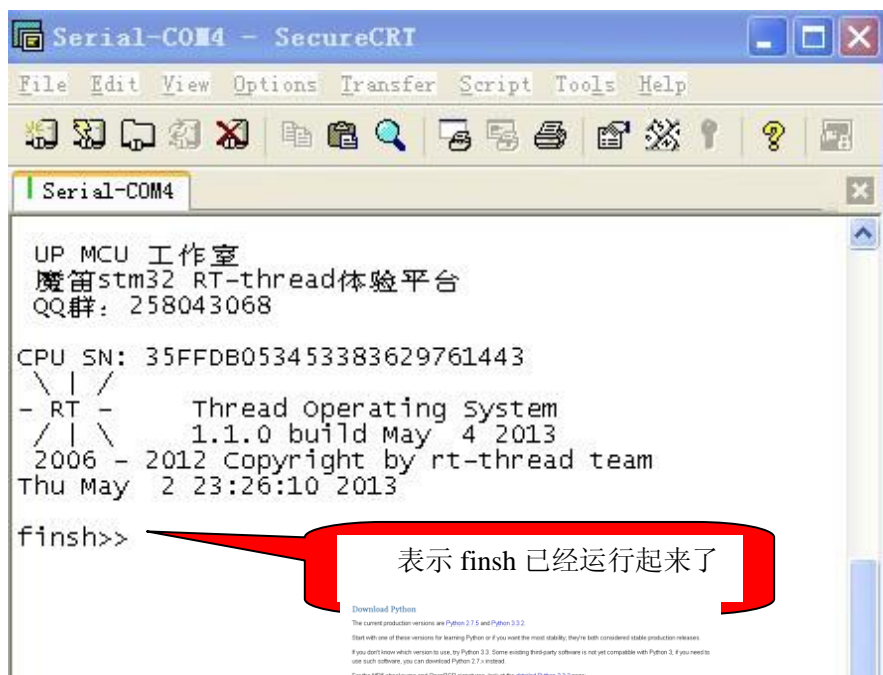
#if STM32_CONSOLE_USART == 0
#define CONSOLE_DEVICE "no"
#elif STM32_CONSOLE_USART == 1
#define CONSOLE_DEVICE "uart1"
#elif STM32_CONSOLE_USART == 2
#define CONSOLE_DEVICE "uart2"
#elif STM32_CONSOLE_USART == 3
#define CONSOLE_DEVICE "uart3"

```

（以上默认使用了 uart1，如果你的目标板是别的串口，可以按照实际情况设置）

◇ Finsh 亮相

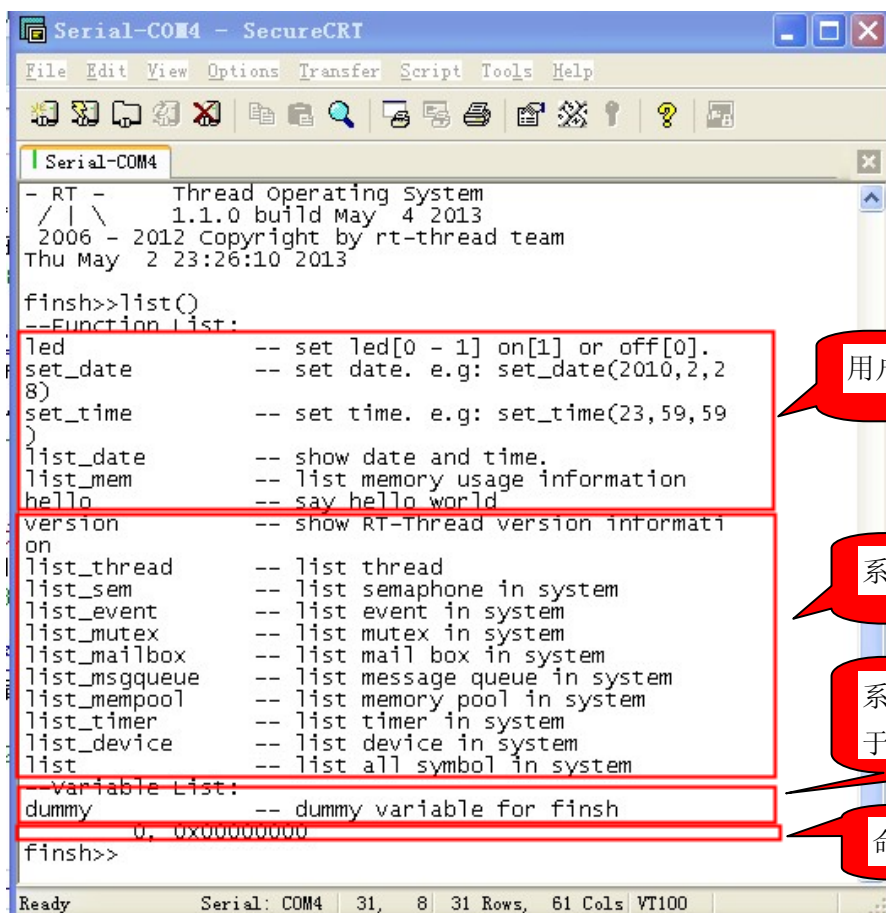
《篇 8-例程 1-finsh 组件》例程是默认配置好的带 finsh 组件工程，我们将开发板的串口连接上 secure.CRT(这个软件魔笛开发板配套资料中有，你也可以百度下载，这是一个功能较强大，且好用的终端调试软件)，运行这个程序，串口的输出如下，当我们看到 finsh>>的提示后，表明 finsh 已经运行起来了。



表示 finsh 已经运行起来了

和其他的命令行一样，在>>的提示符下，RT-Thread 最小系统支持哪些命令，这

我们先来看看一个例子：



用户自定义命令

系统命令

系统 demo 命令，用于查看变量信息

命令的运行返回值

list() 运行后列出了如上面图中的所有命令，除了系统命令外还有一些用户自定义的命令，每个命令后面有这个命令的简要说明，我们可以根据这个说明来了解这些命令的作用，我们再来试运行下几个命令：

```

finsh>>list_thread()
thread pri status sp stack size max used left tick error
-----
tidle 0x1f ready 0x000000058 0x000000100 0x000000060 0x000000009 000
tshell 0x14 ready 0x000000088 0x000000800 0x0000002f0 0x000000003 000
led 0x14 suspend 0x000000078 0x000000200 0x000000078 0x000000002 000
0, 0x000000000
finsh>>list_device()
device type
-----
rtc RTC
uart1 Character Device
0, 0x000000000
finsh>>list_mem()
total memory: 58472
used memory : 696
maximum allocated memory: 2896
0, 0x000000000
finsh>>set_time(10,20,34)
0, 0x000000000
finsh>>list_time()
Null node
finsh>>list_data()
Null node
finsh>>list_date()
Fri May 3 10:21:20 2013
0, 0x000000000

```

列出系统中存在的线程，及其信息

列出系统注册的设备

列出系统内存情况（堆空间）

设置系统时间

不存在或命令错误将不会被执行

显示系统时间

运行几个命令后，是不是觉得 finsh 组件的功能相当惊艳！呵呵，觉得惊艳那就对了，RT-Thread 的其他组件也都有相应的 finsh 命令供我们使用，我们可以在后续的使用中慢慢摸索。

前面看到，finsh 已经有了一些用户自定义的命令，我们要想自己定义一个命令，该如何做？

✧ finsh 中自定义命令，运行函数、查看变量

向 Finsh 中添加自定义命令的简单方法是使用宏方式输出，即使用下面的类似“对外声明”的方式：（我们还需要在 rt_config.h 中使能：#define FINSH_USING_SYMTAB）

```
FINSH_FUNCTION_EXPORT ( )
```

```
FINSH_FUNCTION_EXPORT_ALIAS ( )
```

```
FINSH_VAR_EXPORT ( )
```

我们大可以参照 RT-Thread 源码中 hello() 命令和 dummy() 命令的具体方法，依葫芦画瓢添加我们自己的命令，这里先列出 RT-Thread 源码中 hello() 和 dummy() 的实现：


```

long hello()
{
    rt_kprintf("Hello RT-Thread!\n");
    return 0;
}

FINSH_FUNCTION_EXPORT(hello, say hello world)

static int dummy = 0;
FINSH_VAR_EXPORT(dummy, finsh_type_int, dummy variable for finsh)

```

参照以上，我在《篇 8-例程 1-finsh 组件》例程中 led.c 中分别实现了几个简单的自定义命令：

```

int min_select(int a,int b)
{
    return((a)>(b)?(b):(a));
}

int max_select(int a,int b)
{
    return((a)>(b)?(a):(b));
}

/*          函数名命令说明          */
FINSH_FUNCTION_EXPORT(min_select,return the min one.)

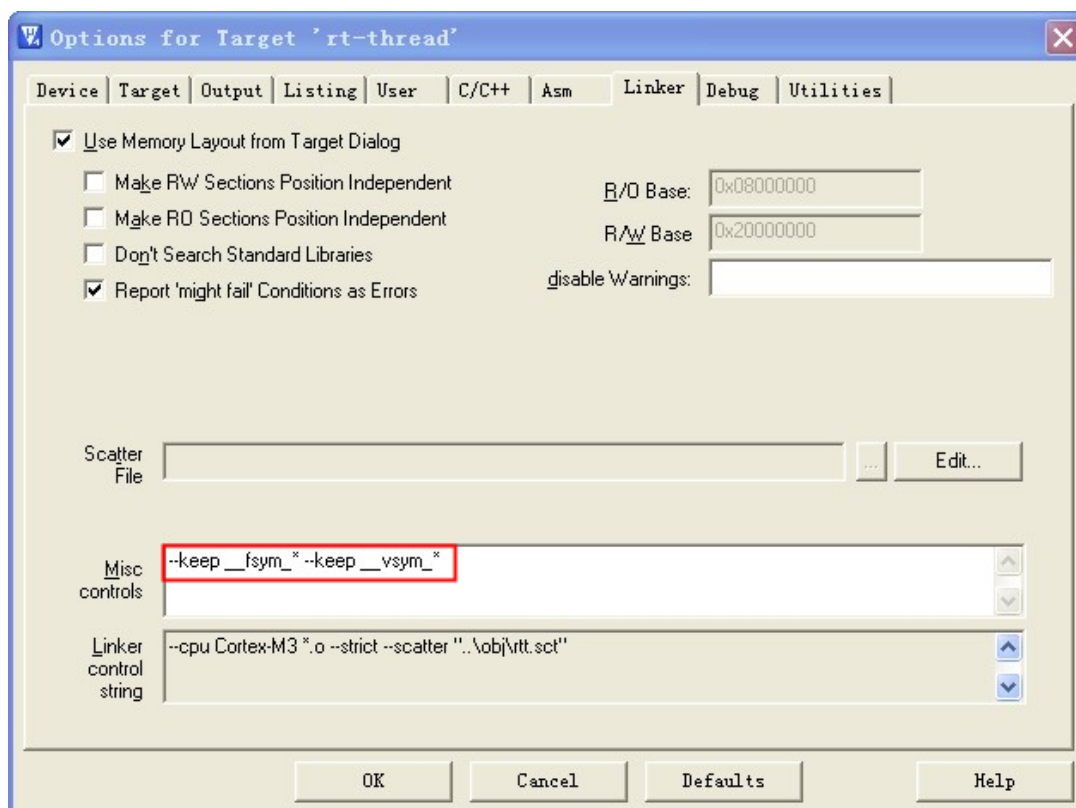
/*          函数名别名命令说明          */
FINSH_FUNCTION_EXPORT_ALIAS(max_select, max,return the max one.)

static int count;
/*          变量名变量类型命令说明          */
FINSH_VAR_EXPORT(count, finsh_type_int, count variable for finsh)

```

其中 FINSH_FUNCTION_EXPORT_ALIAS(max_select, max, return the max one.)的意思是使用 max 这个命令来代替 max_select 函数，我们在原函数名太长或很难记的时候可以使用这种别名的方式来替换。

最后提醒下，另外为了防止 MDK 编译时连接器移除我们自定义的命令，还需要在 mdk 的设置中加入如下红色框中的控制字符（scons 自动生成工程时，会自动帮我们加上的）：



编译后下载运行，可以看到，我们的自定义命令出现在了命令列表中：

```
--FUNCTION LIST:
led          -- set led[0 - 1] on[1] or off[0].
min_select   -- return the min one.
max          -- return the max one.
set_date     -- set date. e.g: set_date(2010,2,28)
set_time     -- set time. e.g: set_time(23,59,59)
list_date    -- show date and time.
list_mem     -- list memory usage information
hello        -- say hello world
version      -- show RT-Thread version information
list_thread  -- list thread
list_sem     -- list semaphore in system
list_event   -- list event in system
list_mutex   -- list mutex in system
list_mailbox -- list mail box in system
list_msgqueue -- list message queue in system
list_mempool  -- list memory pool in system
list_timer   -- list timer in system
list_device  -- list device in system
list         -- list all symbol in system
--Variable List:
count        -- count variable for finsh
dummy        -- dummy variable for finsh
...          0, 0x00000000
```

第九篇软件定时器

日期：2013-05-18

◇ 定时器介绍

RT-Thread 中除了使用芯片的自身硬件定时器产生系统滴答时钟，来处理和时间有关的操作外，还提供了软件定时器供我们使用。

软件定时器构建在硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。软件定时器的精度取决于它使用的硬件定时器精度，例如硬件定时器精度是 100ms，那么它能够提供 100ms，200ms，800ms 等 100ms 整数倍的定时器，而不能提供 80ms 的定时器。

在 RT-Thread 中，软件定时器模块是以一个系统滴答为时间单位的，比如我们设置 RT_TICK_PER_SECOND 为 100，那么软件定时器的周期就是 10ms。另外 RT-Thread 中的软定时器提供两类定时机制：

- 1、单次触发式定时器，这类定时器只会触发一次定时器事件，然后定时器自动停止。
- 2、周期式触发定时器，这类定时器会周期性的触发定时器时间。

◇ 定时器基本操作

和信号量、邮箱等一样，RT-Thread 中的定时器也有静态和动态之分，和定时器有关的操作如下：

初始化定时器—rt_timer_init()（对应静态定时器）；

建立定时器—rt_timer_create()（对应动态定时器）；

控制定时器—rt_timer_control()；

停止定时器—rt_timer_stop()；

脱离定时器—rt_timer_detach()（对应静态定时器）；

删除定时器—rt_timer_delete()（对应动态定时器）；

我们使用了《篇 9-例程 1-软件定时器》例子来演示定时器的基本操作。

首先，建立一个动态定时器，并启动它：

```
/*创建动态定时器*/
timer_test = rt_timer_create("timer1",/* 定时器名字是 timer1 */
                             timeout_callbak,/* 超时回调的处理函数 */
                             RT_NULL,/* 超时函数的入口参数 */
```

```

        timeout_value, /* 定时长度, 以 OS Tick 为单位, 即 timeout_value 个 OS Tick */
        RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */

/* 启动定时器 */
if(timer_test != RT_NULL)
    rt_timer_start(timer_test);

```

上面的参数指定定时器为周期性定时器, 超时时间是 `timeout_value` 个系统滴答, 超时后将执行 `timeout_callbak()` 函数, 这个函数会进行 led 灯的反转:

```

/* 超时回调的处理函数 */
void timeout_callbak(void* parameter)
{
    flag ^= 1;

    if(flag)
        rt_hw_led_off(0);
    else
        rt_hw_led_on(0);

    rt_kprintf("timer time out !\n");
}

```

这样, 在程序运行起来后我们会看到, led 灯快速闪烁 (初始的超时时间是 10 个系统滴答, 所以闪烁较快), 然后我们在检测到开发板上“确认”键按下后调用 `timer_conrol()` 函数, 对定时器进行控制, 控制细节如下:

```

/* 控制定时器 */
void timer_conrol(void)
{
    timeout_value += 10;
    /* 更改定时器超时时间 */
    rt_timer_control(timer_test, RT_TIMER_CTRL_SET_TIME,
                     (void*)&timeout_value);
    rt_kprintf("timer timeout time set to %d !\n", timeout_value);
    if(timeout_value == 500)
    {
        rt_timer_stop(timer_test); /* 停止定时器 */
        rt_kprintf("timer stoped !\n");
    }

    if(timeout_value >= 510)
    {
        /* 再次启动定时器 */
        rt_timer_start(timer_test);
        timeout_value = 10;
    }
}

```

```
    rt_timer_control(timer_test, RT_TIMER_CTRL_SET_TIME,  
                      (void*)&timeout_value);  
}  
}
```

每一次进入控制时，将定时器超时时间增加 10 个系统滴答时间（即 100ms），我们会看到 led 灯的闪烁频率变慢了，当超时时间到达 500 个系统滴答时，我们使定时器停止，led 灯将不再闪烁。定时器停止后如果想再开启，可以通过调用 `rt_timer_start()`。

好了，定时器的控制函数还有其他几个参数：

```
#define RT_TIMER_CTRL_SET_TIME      0x0  /**< set timer control command */  
#define RT_TIMER_CTRL_GET_TIME      0x1  /**< get timer control command */  
#define RT_TIMER_CTRL_SET_ONESHOT   0x2  /**< change timer to one shot */  
#define RT_TIMER_CTRL_SET_PERIODIC  0x3  /**< change timer to periodic */
```

读者，可以自己去试一试。

第十篇 RT-Thread 相关开发工具安装配置

日期：2013-06-17

前面的讲解，我们都使用 Keil MDK 作为开发平台，这主要是为了降低初学者学习和使用 RT-Thread 的难度。但随着学习的深入，我们最终还是需要学习如何使用官方推荐的 Scons 去编译和构建工程，这是一个逃不过的坎。其实只要我们认真按照下面的说明认真走一遍这个过程，会发现使用 Scons 其实也很简单！

这一篇我们来讲一讲用 Scons 来编译和构建 RT-Thread 工程所需要用到的软件的安装和使用，相关内容 RT-Thread 的官方网站的 wiki 和论坛中也有资料介绍，笔者这里还是再梳理一下。

✧ 所需的软件及其作用

■ MsysGit 、TortoiseGit

RT-Thread 的相关的代码都已经移到了 github 上托管，TortoiseGit 是用户下载同步 RT-thread 各部分最新源码的工具，同时开发者也使用它来提交代码，为了在我们的电脑上能够安装 TortoiseGit，我们需要先安装 msysgit。

■ Scons

RT-Thread 使用 SCons 作为默认的编译和构建工具。

■ Python

由于 SCons 基于 Python 开发，因此我们使用 Scons 时必须安装 Python。

■ Keil MDK 、GCC

为 Scons 提供 armcc、gcc 工具链，同时我们也可以在 Keil 的图形界面下直接编辑和编译工程。目前模块（module）编译只支持 gcc 工具链。

注意：建议在这些工具软件的安装路径中尽量不要包含中文路径名

✧ TortoiseGit 工具的安装及如何从 github 端下载源码

■ 安装 TortoiseGit

前面讲到，要安装 TortoiseGit 必须先安装 msysgit，msysgit 的安装包下载地址是：

<http://code.google.com/p/msysgit/downloads/list>，打开此网址后，选择最新版本下载，如下图所示：



下载完毕后，双击安装包进行安装，除了选择安装路径外，其他均选择默认，一路 Next 即可。

TortoiseGit 的下载地址是:<http://code.google.com/p/tortoisegit/wiki/Download?tm=2> ,点击进入后，我们根据自己的系统选择相应的安装包进行下载，笔者这里下载了 32-bitOS 版本：

Download

The current version is: 1.8.3.0

For detailed info on what's new, read the [ReleaseNotes](#).

[System prerequisites and installation howto](#)

Please make sure that you choose the right installer for your PC, otherwise the setup will fail

for 32-bit OS	for 64-bit OS
Download TortoiseGit 1.8.3.0 - 32-bit	Download TortoiseGit 1.8.3.0 - 64-bit

TortoiseGit 的安装也很简单，我们还是按照默认，一路点击 Next 即可。

完成后，我们在桌面点击鼠标右键，可看到如下多出了如下图红色框中的菜单：

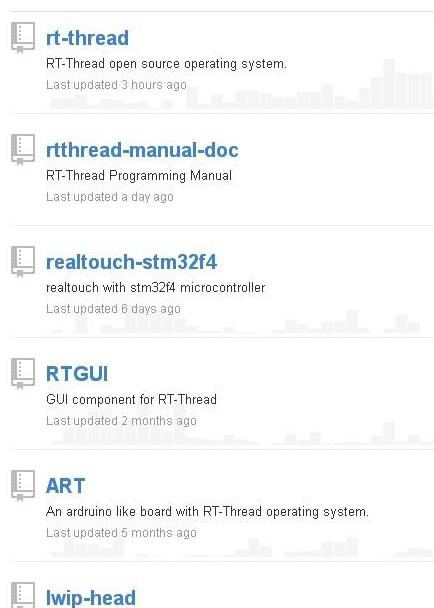


■ 从 github 端获得 RT-Thread 源码

文档的开头讲到，我们可以从 RT-Thread 官网：<http://www.rt-thread.org/>获得 RT-Thread 源码的稳定发行版，另外我们还有两种直接获得 RT-Thread 最新版本源码的方法：

1. 网页直接下载压缩包

登陆：<https://github.com/RT-Thread>，可看到和 RT-Thread 相关的项目代码：



点击我们要下载的项目，比如我们下载主程序，点击“RT-Thread”，在跳转后的页面点击如下图所示的红色框，即可下载到源码的压缩包

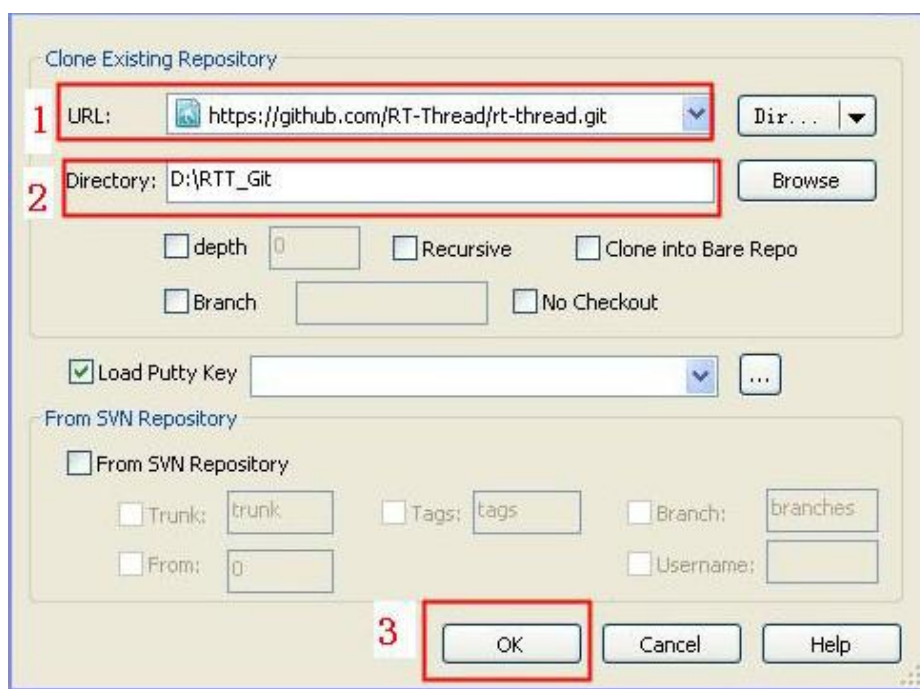


2. 使用 Git 工具进行源代码克隆

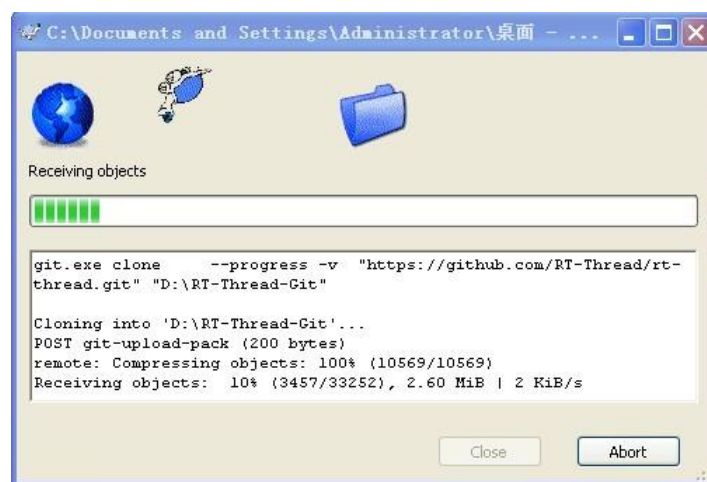
在上图的界面，除了左侧的“ZIP”下载链接，右侧还有如下图所示的 Git 代码托管链接：



我们将这个链接复制到剪贴板，然后在桌面点击右键，选择“Git Clone”，然后弹出如下的对话框



上图中的 1 处，是刚才我们复制的 RT-Thread 源码托管链接，2 处我们选择源码的存放路径，然后点击 3 处的 OK，开启远端代码下载，如下图：



✧ Python 和 Scons 的安装

由于 Scons 基于 Python 编写，所以我们需要优先安装 Python。

■ 安装 Python

Python 的安装包下载地址：<http://www.python.org/getit>，打开此网页，界面如下：

Download Python

The current production versions are [Python 2.7.5](#) and [Python 3.3.2](#).

Start with one of these versions for learning Python or if you want the most stability, they're both considered stable production releases.

If you don't know which version to use, try Python 3.3. Some existing third-party software is not yet compatible with Python 3; if you need to use such software, you can download Python 2.7.x instead.

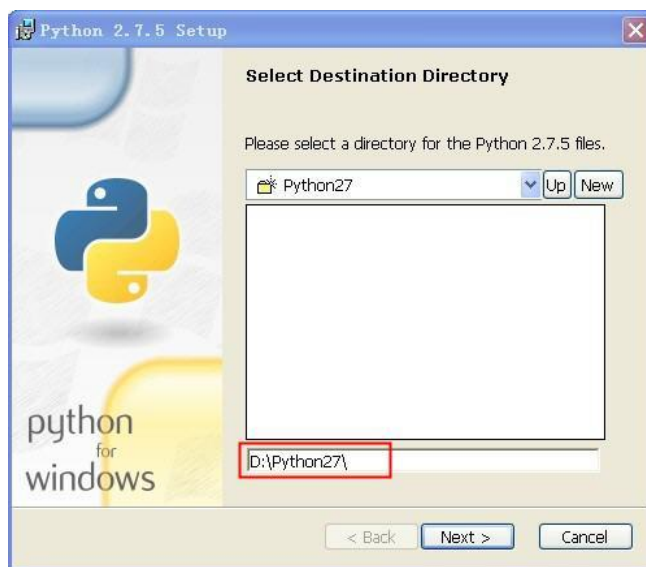
For the MD5 checksums and OpenPGP signatures, look at the [detailed Python 3.3.2](#) page:

- [Python 3.3.2 Windows x86 MSI Installer](#) (Windows binary -- does not include source)
- [Python 3.3.2 Windows X86-64 MSI Installer](#) (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- [Python 3.3.2 Mac OS X 64-bit/32-bit x86-64/i386 Installer](#) (for Mac OS X 10.6 and later [2])
- [Python 3.3.2 Mac OS X 32-bit i386/PPC Installer](#) (for Mac OS X 10.5 and later [2])
- [Python 3.3.2 bzipipped source tarball](#) (for Linux, Unix or Mac OS X)
- [Python 3.3.2 xzipped source tarball](#) (for Linux, Unix or Mac OS X, better compression)

For the MD5 checksums and OpenPGP signatures, look at the [detailed Python 2.7.5](#) page:

- [Python 2.7.5 Windows Installer](#) (Windows binary -- does not include source)
- [Python 2.7.5 Windows X86-64 Installer](#) (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- [Python 2.7.5 Mac OS X 64-bit/32-bit x86-64/i386 Installer](#) (for Mac OS X 10.6 and later [2])
- [Python 2.7.5 Mac OS X 32-bit i386/PPC Installer](#) (for Mac OS X 10.3 and later [2])
- [Python 2.7.5 compressed source tarball](#) (for Linux, Unix or Mac OS X)
- [Python 2.7.5 bzipipped source tarball](#) (for Linux, Unix or Mac OS X, more compressed)

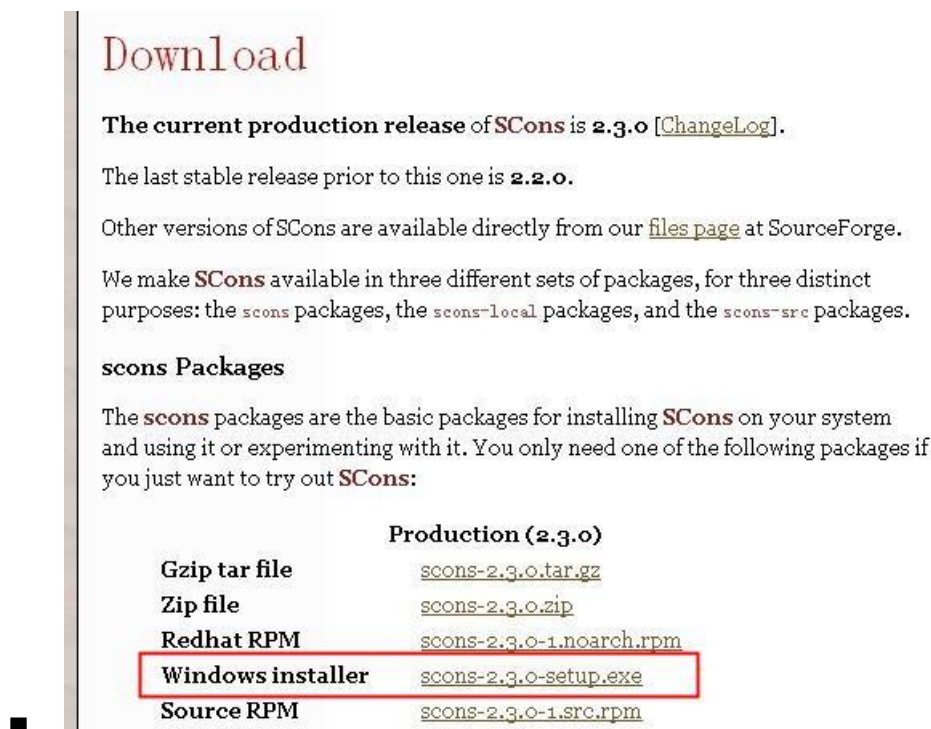
目前 scons 还不支持 python3.0 以上版本所以我们下载时选择 2.7 版的，我们需要根据自己电脑的实际情况选择性的下载所需安装包，笔者的电脑是 32 位的 Windows XP，所以选择上图中红色框的 python2.7.5。安装包下载完毕后，双击安装，在下图界面中设置安装路径后，一路点击 Next 完成安装。



笔者的安装路径是：D:\Python27

■ 安装 Scons

Scons 的安装包下载地址：<http://www.scons.org/download.php>，打开此网页，界面如下：



我们选择上图中红色框中的 Windows 安装版进行安装，安装时 Scons 会自动搜索 Python 的安装路径，所以我们一路点击 Next 即可。

笔者的 Scons 被安装到了 D:\Python27\Scripts

■ 设置 Python、Scons 环境变量

接下来我们需要把 Python 和 SCons 安装目录添加到系统的环境变量中。我们右击“我的电脑”选择“属性”，在弹出的对话框中选择“高级”选项卡（Win7 电脑可能界面不太一样），如下图：



双击红色框中的“环境变量”，将弹出下图：



我们在“系统变量”中找到“Path”变量，双击 path，根据我们的实际安装路径将 Python 和 Scons 的安装目录加入到 path 中，两个路径分别用分号隔开，然后点击确定。



如上图，笔者将 D:\Python27 和 D:\Python27\Scripts 加入到了其中。

保险起见，我们来看一看我们刚才设置的环境变量是否起效了。打开 cmd 命令行，运行 set path 命令，当看到输出结果中有我们刚才设置的那两个路径的话，即说明我们的设置正确起效，否则需要重新设置。

```
C:\Documents and Settings\Administrator>set path
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;D:\Python27;D:\Python27\Scripts;
```

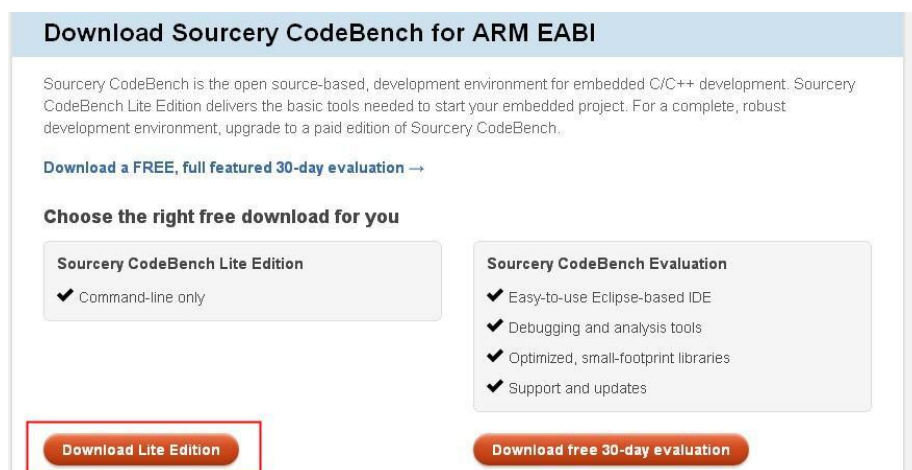
✧ Keil MDK 和 GCC 编译器的安装

■ Keil MDK 安装

Keil MDK 的安装使用，想必大家都非常熟悉了，这里就不多讲了，Keil MDK 最新版下载地址是：<http://www.keil.com/download/product/>

■ GCC 编译器的安装

打开 <https://sourcery.mentor.com/sgpp/lite/arm/portal/subscriptions3053>，界面如下：



我们选择下载 Lite 版本，点击“Download Lite Edition”，在下面界面中填写自己的注册信息，然后点击“Get Lite”。

GNU debugger

First Name

Last Name

Email

Country

A valid email address is required.

Please Provide Your City

Get Lite!

我们按照网页的提示在自己的注册邮箱中去获取下载地址，打开下载地址连接，如下：

Available Releases

This table lists all releases for download.

Release	Status	Date
Sourcery CodeBench Lite 2013.05-23	Release	2013-05-07
Sourcery CodeBench Lite 2012.09-63	Release	2012-11-13
Sourcery CodeBench Lite 2012.03-56	Release	2012-06-11
Sourcery CodeBench Lite 2011.09-69	Release	2011-12-19
Sourcery G++ Lite 2011.03-42	Release	2011-05-02
Sourcery G++ Lite 2010.09-51	Release	2010-11-10
Sourcery G++ Lite 2010q1-188	Release	2010-04-23

如上图我们选择最新版本，点击上图中红色框中超链接，再选择 Windows 下的安装包进行下载：

Download	MD5 Checksum
Recommended Packages	
IA32 GNU/Linux Installer	25c5dce1fd93410cde1c76f282ca1ed0
IA32 Windows Installer	a01ef7d89fb1a4301f1d5dfcf5318f2d
Advanced Packages	
IA32 GNU/Linux TAR	9d206de1c74f9454e468ddcdd72c9c53
IA32 Windows TAR	10fe6bc1cedda2ad04f5019e6784a88a
Source TAR	4c791ddb3d4bcfee29a26eb4db5a244

下载完毕后双击安装包进行安装。安装过程中，除了更改安装路径外，为了使 GCC 安装目录被自动加入到系统环境变量中，请一路选择默认安装。笔者的安装路径是：**D:\Program Files\CodeSourcery\Sourcery_CodeBench_Lite_for_ARM_EABI**

✧ 设置 RTT_ROOT 环境变量

使用 Scons，需要设置 RT-Thread 源码的存放路径到系统环境变量中。右键点击“我的电脑”选择“属性”，然后选择“高级”选项卡，打开环境变量设置，点击下图红色框中的“新建”



变量名处输入：RTT_ROOT，变量值处输入我们的 RT-Thread 源码所在目录，如下图：



这里，笔者的 RT-Thread 代码目录是：D:\RTT_Git\RT-Thread 1.1.0，设置完毕之后点击确定（请保证目录不包含中文文件名和空格）。

我们可使用 `set RTT_ROOT` 命令来查看环境变量是否被正确设置了：

```
D:\RTT_Git\RT-Thread 1.1.0\bsp\stm32f10x>set RTT_ROOT
RTT_ROOT=D:\RTT_Git\RT-Thread 1.1.0
```

注意：如果之前开着 cmd 命令行窗口，在设置完 RTT_ROOT 环境变量后，需要关闭 cmd 命令行窗口后重新打开才能生效。

✧ 使用 Scons 生成、编译工程

到这里整个 scons 编译环境算是搭建完毕了，我们来测试一下这个环境

■ Scons 下使用 Keil 编译工程

打开某个 bsp 目录，这里我们选择 stm32f10x 目录，找到 rtconfig.py 文件并将其用写字板打开，修改一下 keil 的安装路径，具体见下面的黄色字体部分：

```
import os
# toolchains options
ARCH='arm'
CPU='cortex-m3'
CROSS_TOOL='keil'
# 此处多余部分代码暂时省略
if CROSS_TOOL == 'gcc':
    PLATFORM = 'gcc'
    EXEC_PATH = r'D:\Program
Files\CodeSourcery\Sourcery_CodeBench_Lite_for_ARM_EABI\bin'
elif CROSS_TOOL == 'keil':
    PLATFORM = 'armcc'
    EXEC_PATH = r'D:\Keil'
elif CROSS_TOOL == 'iar':
    PLATFORM = 'iar'
    IAR_PATH = 'C:/Program Files/IAR Systems/Embedded Workbench 6.0
Evaluation'
```

上面指定了编译工具使用 keil，并设置了 keil 安装路径，读者需要按照自己电脑的实际路径设置，笔者的 keil 安装路径是 D:\Keil，我们可以将路径设置为：r'D:\Keil'或者：r'D:/Keil'，如果安装路径中含有空格，则一定要加上“r”，如果不含空格，则“r”可以不要。

设置好 keil 路径后，打开 cmd 命令行窗口，将工作目录切换到 bsp/stm32f10x 目录，输入 `scons -j4` 命令，即可开始编译工程，正确的编译结果如下：

```
fromelf --bin rtthread-stm32.axf --output rtthread.bin
fromelf -z rtthread-stm32.axf

=====

** Object/Image Component Sizes

      Code (inc. data)   RO Data   RW Data   ZI Data   Debug   Object Name
      186638      17560      291710      2808      7112      1997371   rtthread-stm
32.axf (uncompressed)
      186638      17560      291710      1348      7112      1997371   rtthread-stm
32.axf (compressed)
      186638      17560      291710      1348           0           0   ROM Totals f
or rtthread-stm32.axf
scons: done building targets.
```


这时我们看到 stm32f10x 目录下产生了目标文件：rtthread.bin，若要清除编译结果，运行 `scons -c` 命令即可。

注意：scons -jx 中的 x，是指开启 x 个线程，同时进行编译，x 是电脑 cpu 核心数的 2 被，笔者电脑是双核所以用 `scons -j4`，如果读者电脑是 4 核，可使用 `scons -j8`。

如果在编译时看到如下的输出信息，说明 RTT_ROOT 环境变量设置不正确，需要重新设置

```
D:\RTT_Git\RT-Thread1.1.0\bsp\stm32f10x>scons -j4
scons: Reading SConscript files ...
ImportError: No module named building:
  File "D:\RTT_Git\RT-Thread1.1.0\bsp\stm32f10x\SConstruct", line 11:
    from building import *
```

■ Scons 下使用 gcc 编译工程

和使用 keil 编译类似，我们先指定编译工具为 gcc，然后在设置其安装路径，如下面的黄色部分，其中 gcc 的安装路径请按照自己电脑的实际路径来设置，并注意是否要在路径前加 'r'。

```
import os
# toolchains options
ARCH='arm'
CPU='cortex-m3'
CROSS_TOOL='gcc'
# 此处多余部分代码暂时省略
if CROSS_TOOL == 'gcc':
    PLATFORM = 'gcc'
    EXEC_PATH = r'D:\Program
Files\CodeSourcery\SourceCodeBench_Lite_for_ARM_EABI\bin'
elif CROSS_TOOL == 'keil':
    PLATFORM = 'armcc'
    EXEC_PATH = r'D:\Keil'
elif CROSS_TOOL == 'iar':
    PLATFORM = 'iar'
    IAR_PATH = 'C:/Program Files/IAR Systems/Embedded Workbench 6.0
Evaluation'
```

设置好以后同样运行 `scons -j4` 命令来编译，正确的输出结果如下：

```
arm-none-eabi-objcopy -O binary rtthread-stm32.axf rtthread.bin
arm-none-eabi-size rtthread-stm32.axf
  text    data    bss     dec     hex filename
 61828    248    5360    67436   1076c rtthread-stm32.axf
scons: done building targets.
```

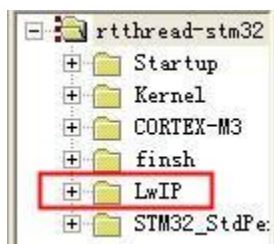
■ 使用 Scons 生成 MDK、IAR 工程

当我们需要对 RT-Thread 工程进行裁剪时，如果手动添加或删除对应.c 文件，将会很麻烦而且容易出错，RT-Thread 为我们提供了自动生成 MDK 工程的方法，这里举例演示一下。

假如我们需要将 lwip 组件加入到工程，首先我们要在 rtconfig.h 中打开宏定义：

```
#define RT_USING_LWIP
```

然后打开 cmd 命令行，切换到对应 bsp 目录，运行: `scons --target=mdk4 -s` 命令，完成后，我们打开工程 `project.uvproj`，即可看到 lwip 已经被加入：



注意：

- 1、如果你的电脑使用的是 MDK3.x 版，对应的命令则是 `scons -target=mdk -s`
- 2、自动生成工程依赖对应 bsp 目录下的 `template.uvproj` 和 `template.Uv2`，请勿误删这两个文件。

使用 scons 自动生成 IAR 工程的命令是：`scons --target=iar -s`，这个也需要 bsp 目录下 `template.ewp` 文件的支持。

✧ CMD 命令行使用的快捷途径

前面我们使用 cmd 命令行时，你是否觉得每次启动 cmd 切换行路径是件很麻烦的事？

这里提供一个快捷途径：

■ Windows xp 电脑用户

在对应的 bsp 目录（如 `bsp/stm32f10x`）中建立一个 txt 文件，比如起名为 `startcmd.txt` 然后使用文本编辑器打开，向其中写入 `start cmd.exe` 后保存，然后再将 `startcmd.txt` 重命名为 `startcmd.bat`，鼠标双击 `startcmd.bat`，看看是不是此时打开的 cmd 命令行窗口路径已经是 `bsp\stm32f103` 呢？

■ Windows 7 电脑用户

对于 win7 用户，按住 `shit`，右键单击文件夹（如 `bsp/stm32f10x`），在弹出的右键菜单选择 `Command Prompt Here`，即可在当前目录打开 cmd 命令行窗口；

另外，我们也可以在工程文件所在的目录下创建一个 creatproject.txt 文档，将运行的命令放到这里，比如 `scons -target=mdk4 -s` 保存后将文件的扩展名更改为 .bat，双击即可运行命令行更新工程。

✧ 结束语

是不是发现，使用 `scons` 其实很简单也很方便呢？好吧，enjoy it！

另外，如果在用 `Scons` 来进行工程编译时有问题（各种脚本问题），也不要紧，因为我们用 `scons` 的目的主要是为了能在裁剪配置 RT-Thread 后的重新生成 `mdk` 或 `iar` 工程，所以只要能产生工程，那么 `scons` 下能否编译就不重要了，我们可以在 `mdk` 或 `iar` 下用 `IDE` 来编译、调试。

第十一篇 RT_Thread 和 RTGUI 版本匹配问题

日期：2013-06-22

截至 2013-6-21, RT-Thread 的最新稳定版是 1.1.0, RTGUI 的最新发布版是 0.6.2。由于 RTGUI 不断更新, 其中一些处理机制也有变化, 所以我们在使用 RTGUI 时需要选择合适的版本, 否则会遇到很多编译错误提示, 这将会使一些初次接触 RTGUI 的用户不知所措。

笔者在这里做一个大概的说明:

✧ 添加 RTGUI 代码

RT_Thread 从 1.1.0 版开始不再包含 RTGUI 部分的代码, 而是将 RTGUI 拉出来单独开发、单独发布 (RTGUI 详情见: <https://github.com/RT-Thread/>), 所以我们在使用 scons 去生成包含 RTGUI 组件的工程时需要做一个代码拷贝工作:

- 1、将 RTGUI 相关代码 (RTGUI 源码的 components 下的 rtgui 目录) 拷贝到 RT-Thread 源码的 components 文件夹下;
- 2、将 RTGUI 的 examples 示例代码 (RTGUI 源码的 demo 下的 examples 目录下全部文件) 拷贝到 RT-Thread 源码的 examples 文件夹下的 gui 文件夹 (如果没有 gui 文件夹, 需要新建一个)。

✧ 生成包含 RTGUI 的工程的注意事项

RTGUI 从 0.6.1 版开始将触摸校准 calibration.c 收录到了 rtgui 的 common 文件夹中, 这样我们不必为每个 bsp 单独添加触摸校准程序了, 同时 Github 端的最新 RT-Thread 代码 bsp 目录中的 SConscript 文件也取消了构建工程时对 calibration.c 文件的添加。所以当我们使用最新 RT-Thread 代码生成包含 RTGUI 组件的工程时没有任何问题, 但要是使用 1.1.0 版 RT-Thread 代码则需要将对应的 bsp 目录下 SConscript 文件做一个修改, 去掉对 calibration.c 文件的添加, 即把

```
if GetDepend('RT_USING_RTGUI'):

    src_drv += ['touch.c', 'calibration.c']
```

改为:

```
if GetDepend('RT_USING_RTGUI'):

    src_drv += ['touch.c']
```

同时最好也将 bsp 目录下的 calibration.c 文件删除, 以绝后患。

以上做完后, 打开对应 bsp 目录 rtconfig.h 中#define RT_USING_RTGUI 宏定义, 即可用 scons --target=mdk4 -s 正确生成包含 RTGUI 的工程了。

第十二篇给 RTGUI 添加字库

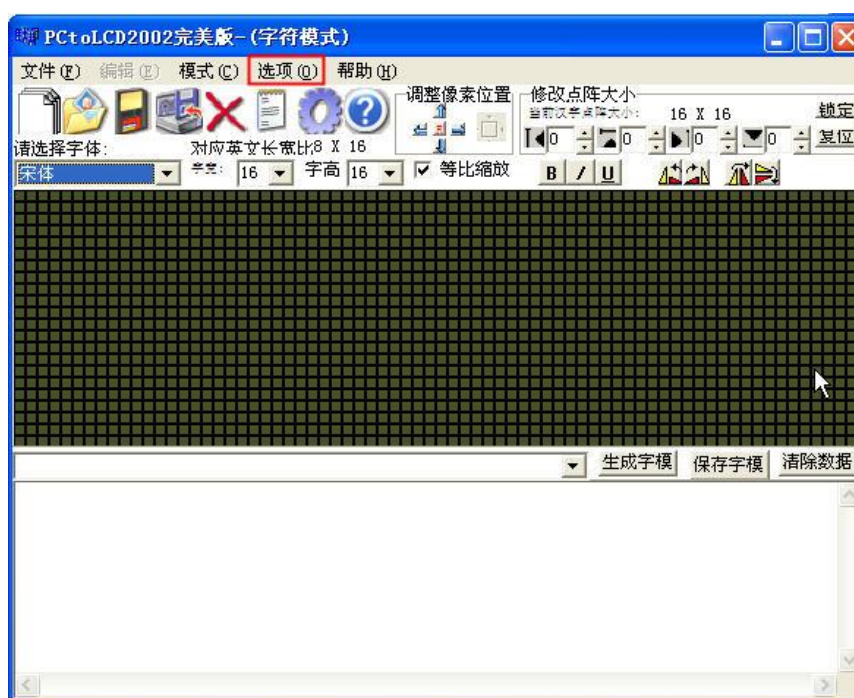
日期：2013-06-23

RTGUI 默认给出了 8*12、8*16 的英文字库和 12*12、16*16 的中文字库，其中英文字库被编译在芯片内部 flash 中，中文字库因为体积较大，我们一般采用外部加载的方式来使用（resource 文件夹中的 hzk12.fnt、hzk16.fnt）。

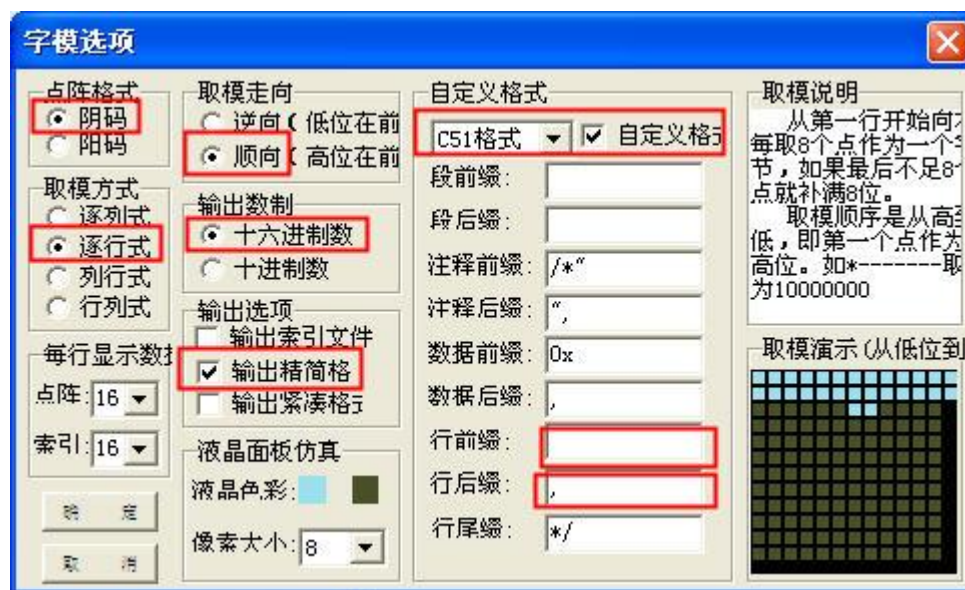
这里来介绍如何自己制作系统外字库，我们使用的字库工具是 PCtoLCD2002，这个工具网上即可下载到：<http://www.cr173.com/soft/37775.html>

✧ 添加英文字库

运行 PCtoLCD2002，界面如下：



点击上图红色框的“选项”，设置字模的取模形式：



上图中红色框住的部分要注意与上图保持一致，否则产生的字模会有问题，设置好后点击左下角的确定。

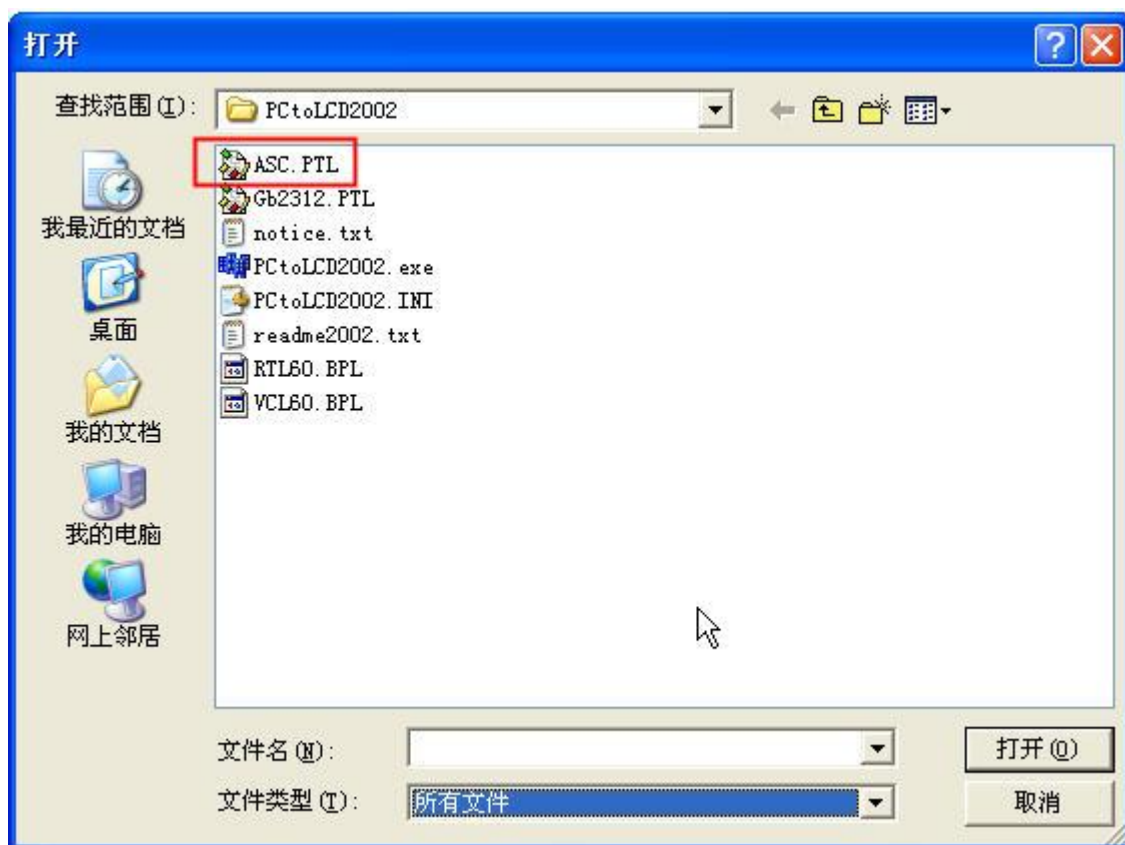
然后按下图红色框部分设置要生成的字库大小，这里我们制作 12*24 的字库：



接下来我们点击下图中红色框，从文本中生成字库



由于我们要生成英文字库，这里就选择“ASC.PTL”



选择好以后，点击下图中的“开始生产”，设置好保存路径，即可得到字库文件



这里得到的字库文件是 txt 文件，还不能直接使用，需要按照 RTGUI 的既定格式做一个“改装”，具体格式我们参考 RTGUI 中的 asc16font.c，“改装”步骤如下：

- 1、新建 asc24font.c 文件，在其中定义一个数组：

```
const unsigned char asc24_font[] =
{
}
```

将刚才生成的 txt 字库文件中的所有内容复制后，粘贴到上述的数组中。

- 2、组建字库信息，将 asc16font.c 文件中最后那两个结构体复制到 asc24font.c 中，并修改如下的黄色部分

```
struct rtgui_font_bitmap asc24 =
{
    (const rt_uint8_t *)asc24_font, /* bmp */
    RT_NULL, /* each character width, NULL for fixed font */
    RT_NULL, /* offset for each character */
    12, /* width */
    24, /* height */
    0, /* first char */
    255, /* last char */
}
```



```
};

struct rtgui_font rtgui_font_asc24=
{
    "asc", /* family */
    24, /* height */
    1, /* refer count */
    &bmp_font_engine, /* font engine */
    &asc24, /* font private data */
};
```

3、保存 asc24font.c 文件，并将其加入到需要此字体的工程中去。

这样一个 12*24 的英文字库就做好了，当需要用到此字库时，使用下面语句将其加入到 RTGUI 字体系统中就好了：

```
{
    extern struct rtgui_font rtgui_font_asc24;
    rtgui_font_system_add_font(&rtgui_font_asc24);
}
```

✧ 添加中文字库

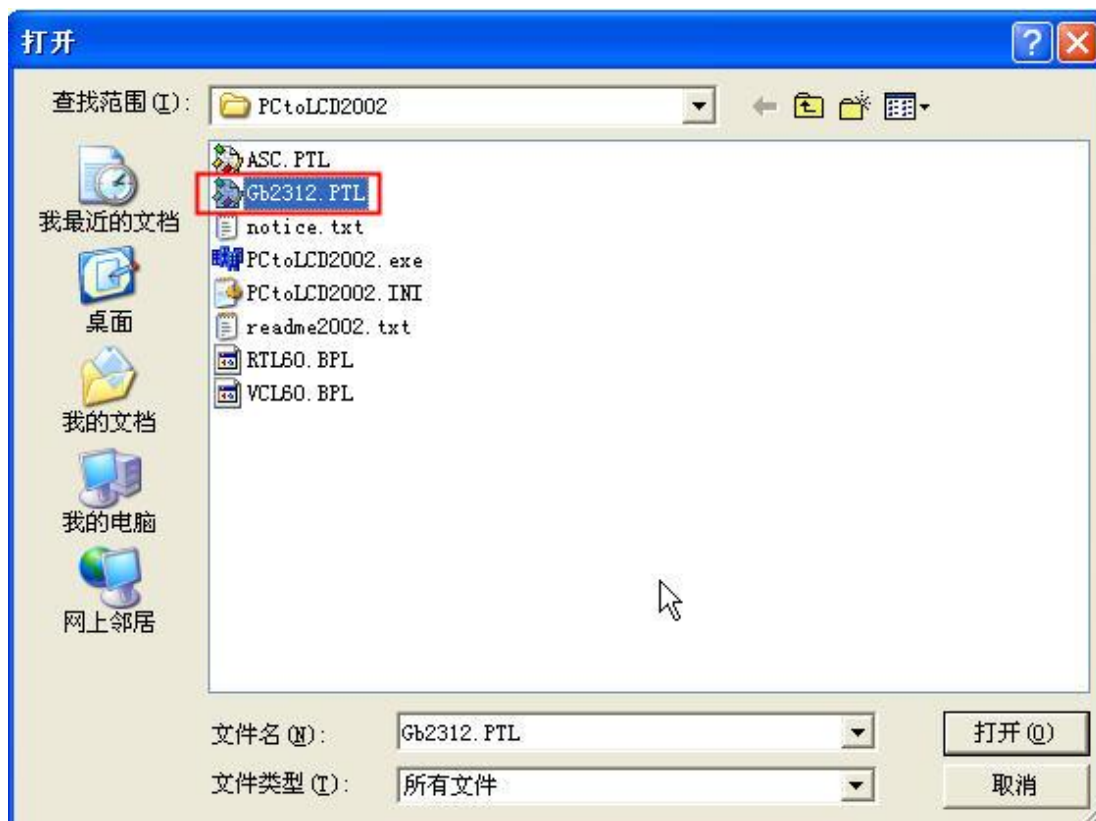
在 RTGUI 中，对于中文字体有两种处理方式：

一种是把点阵数据转换成 C 语言数组，然后在代码中直接使用，比较适合于把字体与程序都烧写在 flash 中的方式；

另外一种是把点阵数据保存为一个二进制文件，在使用时，动态的从文件系统中载入进来。

■ C 数组格式的中文字库生成

这种方式字库的取模方式和英文字库的取模方式一样，唯一的不同是要选择不同的字体文件，如下图：



我们按照生成英文字库的步骤设置好取模方式和所要生成的字体、字体大小（笔者这里选择宋体 24*24）后选择“Gb2312.PTL”，然后生成 txt 格式字库文件

这里我们还要参考 RTGUI 中的 hz16font.c 中文字库格式对这个 txt 格式的中文字库做一个“改装”，步骤如下：

- 1、新建 hz24font.c 文件，在其中定义一个数组：

```
const unsigned char hz24_font[] =
{
}

```

将刚才生成的 txt 字库文件中的所有内容复制后，粘贴到上述的数组中。

- 2、组建字库信息，将 hz16font.c 文件中 16741 行到 16760 行的两个结构体复制到 hz24font.c 中，并修改如下的黄色部分：

```
const struct rtgui_font_bitmap hz24 =
{
    hz24_font, /* bmp */
    RT_NULL, /* each character width, NULL for fixed font */
    RT_NULL, /* offset for each character */
    24, /* width */
    24, /* height */
    0, /* first char */
}

```

```

    255/* last char */
};

extern struct rtgui_font_engine hz_bmp_font_engine;
struct rtgui_font rtgui_font_hz24=
{
    "hz",/* family */
    24,/* height */
    1,/* refer count */
    &hz_bmp_font_engine,/* font engine */
    (void*)&hz24,/* font private data */
};

```

3、保存 hz24font.c 文件，并将其加入到需要此字体的工程中去。

这样一个 24*24 的中字库就做好了，当需要用到此字库时，使用下面语句将其加入到 RTGUI 字体系统中就好了：

```

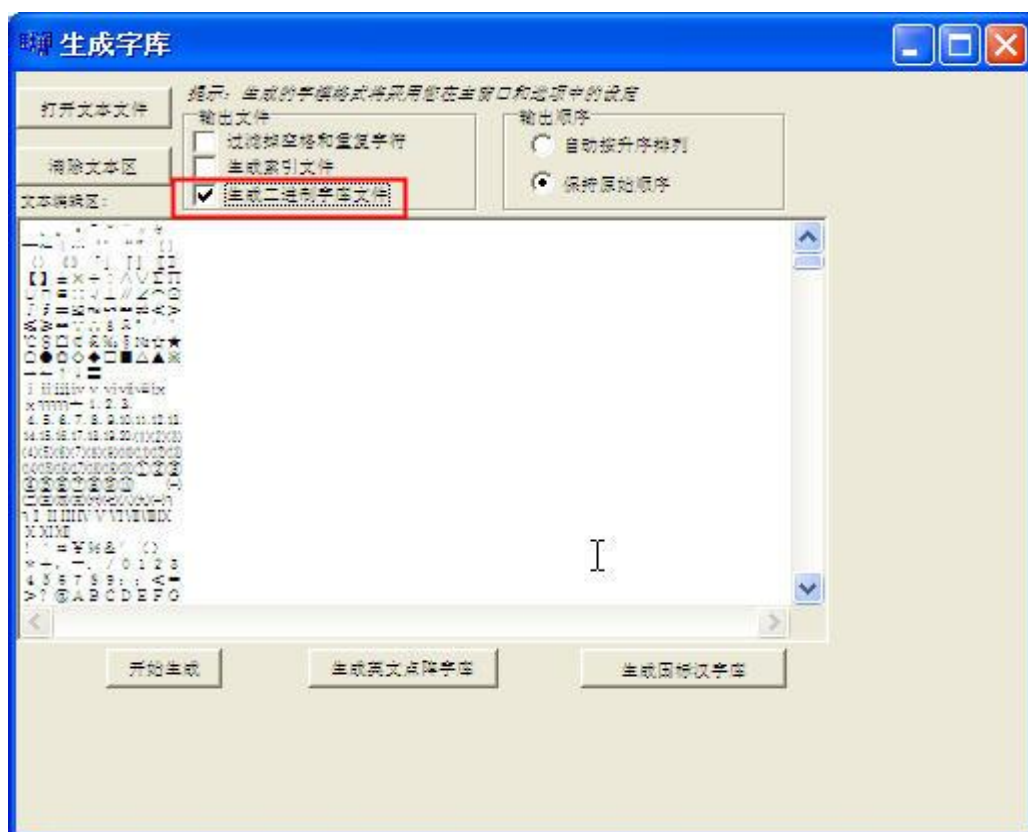
{
    extern struct rtgui_font rtgui_font_hz24;
    rtgui_font_system_add_font(&rtgui_font_hz24);
}

```

■ 外部加载的二进制中文字库生成

使用 PCtoLCD2002 同样可以生成用于外部加载的二进制中文字库，我们只需按照前面的方法设置好取模方式和字体、字体大小后按照下图提示选择“生成二进制字库文件”然后点击“开始生成”，即可生成.FON 格式的二进制字库。

我们将此.FON 文件重命名为.fnt，然后将字库文件放入到外部文件系统（比如放在 SD 卡的 resource 文件夹）



为了使用刚才的字库，我们需要在程序的合适地方（比如 `hz16font.c` 文件的最后面）参考 `hz16font.c` 文件的格式构建出这个字库的相关信息，请注意下面的黄色字体部分：

```
struct rtgui_hz_file_font hz24=
{
    {RT_NULL},/* cache root      */
    0,/* cache size      */
    24,/* font size      */
    24*24/8,/* font data size  */
    -1,/* fd            */
    "/resource/hzk24.fnt"/* font_fn      */
};

struct rtgui_font rtgui_font_hz24=
{
    "hz",/* family */
    24,/* height */
    1,/* refer count */
    &rtgui_hz_file_font_engine,/* font engine */
    (void*)&hz24,/* font private data */
};
```

当需要用到此字库时，使用下面语句将其加入到 RTGUI 字体系统中就好了：

```
{
    extern struct rtgui_font rtgui_font_hz24;
    rtgui_font_system_add_font(&rtgui_font_hz24);
}
```

✧ 添加英文字库到外部文件系统

如果一个系统中需要用到多套较大字体的英文字库，那么将其放入到 flash 中一起编译的话明显不是一个好方法，RT-Thread 的一个网友【郁海难填】提供了一种方法可以将英文字库像中文字库那样放到外部文件系统来访问，我在这里介绍一下这个方法。

此方法是对现有 RTGUI 代码加一个“补丁”文件，使其支持从外部文件系统得到英文字库，此补丁见连载教程例程包的第 12 篇。使用时我们将此补丁加入到工程中，并在 font.h 中添加如下定义：

```
struct asc_cache
{
    SPLAY_ENTRY(asc_cache) asc_node;

    rt_uint8_t asc_id;
};

struct rtgui_asc_file_font
{
    struct asc_cache_tree cache_root;
    rt_uint16_t cache_size;

    /* font size */
    rt_uint16_t font_size;
    rt_uint16_t font_data_size;

    /* file descriptor */
    int fd;

    /* font file name */
    const char* font_fn;
};

extern const struct rtgui_font_engine rtgui_asc_file_font_engine;
```

这样一来，RTGUI 就支持从外部文件系统加载英文字库了，外部二进制英文字库的制作方法和中文字库的方法一样，字库信息构建方式也一样，比如 16*24 的二进制英文字库信息构建如下：

```
struct rtgui_asc_file_font ascii24=
{
    {RT_NULL}, /* cache root */
    /*
```

```

0,/* cache size      */
24,/* font size      */
16*24/8,/* font data size */
-1,/* fd             */
"/resource/ascii24.fnt"/* font_fn      */
};

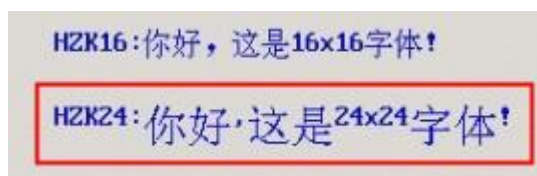
struct rtgui_font rtgui_font_ascii24=
{
    "asc",/* family */
    24,/* height */
    1,/* refer count */
    &rtgui_asc_file_font_engine,/* font engine */
    (void*)&ascii24,/* font private data */
};

```

✧ RTGUI 字库使用注意事项

■ 中英文字库要成双成对

如果使用某种大小的中文字库，那么系统中也需要有同样大小的英文字库，否则英文字体将按照默认的大小显示，情况和下图类似：



■ 外部模块加载时要注意符号导出

如果使用外部模块加载的方式运行 gui 相关程序，且用到了系统外自定义的字库，需要将一些符号 RTM_EXPORT 导出，否则将不能正常显示。

比如，当我们看到串口输出的信息中有类似下面的报错信息时，则需要进行先关符号的导出，如：RTM_EXPORT(rtgui_font_system_add_font)

```

rt_module_load: fontdemo ,can't find rtgui_font_system_add_font in kernel symbol
table
can't find rtgui_font_refer in kernel symbol table
can't find rtgui_hz_file_font_engine in kernel symbol table

```

第十三篇 RT-Thread 新的组件初始化方式

日期：2013-11-08

从 RT-Thread1.2.0bata 版开始，内核使用了全新的组件初始化方式，关于实现这个初始化方式的原因和原理，熊老师在这里有一个说明：

<http://www.rt-thread.org/phpBB3/viewtopic.php?f=3&t=2867>

我们先来认真看一看上面的帖子……

✧ 为什么提出全新的初始化方式

看过上面的帖子之后之后，对于这个新的初始化方式我的理解是这样的：

1、避免过多、过杂的宏定义，使用户代码尽量清晰；

按照新的初始化方式，我们的 init 初始化线程可以简化成如下的一个很简短的函数，

```
void rt_init_thread_entry(void* parameter)
{
    {
        extern void rt_platform_init(void);
        rt_platform_init();
    }

    /* initialization RT-Thread Components */
    rt_components_init();
}
```

比起原来那种“宏套宏”的累赘方式，我们的 init 线程代码完成了“极度瘦身”。

2、各个被用到的组件会自动加入到初始化列表，而不用手动去调用；

```
void rt_hw_board_init(void)
{
    /* NVIC Configuration */
    NVIC_Configuration();

    /* Configure the SysTick */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );

#ifdef STM32_EXT_SRAM
    EXT_SRAM_Configuration();
#endif
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#else
    rt_hw_usart_init();
#endif
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
}
```

```
}
```

如上面代码所示，使用新的初始化方式时，我们不必再手动调用 `rt_hw_usart_init()` 等这类函数，`rt_components_board_init()` 函数帮我们自动完成了类似这样的调用。

✧ 新初始化方式的原理

当用户需要使用新的初始化方式来自动调用自己实现的初始化代码时，需要将其使用 `INIT_XXX_EXPORT` 宏来导出，这些宏有如下 6 个等级：

```
#define INIT_BOARD_EXPORT(fn)      INIT_EXPORT(fn, "1")
#define INIT_CPU_EXPORT(fn)       INIT_EXPORT(fn, "2")
#define INIT_DEVICE_EXPORT(fn)    INIT_EXPORT(fn, "3")
#define INIT_COMPONENT_EXPORT(fn) INIT_EXPORT(fn, "4")
#define INIT_FS_EXPORT(fn)        INIT_EXPORT(fn, "5")
#define INIT_APP_EXPORT(fn)       INIT_EXPORT(fn, "6")
```

被以上 6 个宏导出的函数，初始化的顺序按照从上到下的顺序，即被 `INIT_BOARD_EXPORT` 导出的函数，最先被初始化，被 `INIT_APP_EXPORT` 导出的函数，最后被初始化。

另外使用同一个宏导出的 N 的函数，它们的初始化顺序由编译器决定，如：

```
INIT_DEVICE_EXPORT(FUNC1)
INIT_DEVICE_EXPORT(FUNC2)
INIT_DEVICE_EXPORT(FUNC3)
```

`FUNC1`、`FUNC2`、`FUNC3` 这三个函数的初始化调用顺序完全由编译器决定，所以，我们需要保证这三个函数之间无相互依赖关系。

✧ 如何使用

1、我们如果要使用这种全新的初始化方式，首先要在 `rtconfig.h` 中开启对应的宏：

```
#define RT_USING_COMPONENTS_INIT
```

2、需要被 `INIT_xxxx_EXPORT` 宏导出的函数，其原型需要改为这样的形式：

```
int xxxx_init(void)
```

比如：`void rt_hw_usart_init(void)` 我们要改为 `int rt_hw_usart_init(void)`；

`void rt_hw_rtc_init(void)` 我们要改为 `int rt_hw_rtc_init(void)`

另外 `int xxxx_init(void)` 的函数原型告诉我们，如果是带有初始化形参的函数，是不能使用这种初始化方式的，比如在 `spi` 总线架构下的 `spi flash` 的初始化：

```
w25qxx_init("flash0", "spi10");
```


如果不想使用新的初始化方式，我们还可以按照之前的方式，自己完成各个组件的相关初始化函数调用。

这一篇里，我给出在魔笛 F1（stm32 radio）和魔笛 F4（realtouch）上使用这种全新初始化方式的含有 `finsh` 组件、文件系统、RTC 的完整 MDK 工程，这些工程使用 `scons` 生成。

请大家仔细对比 `rtc` 初始化、串口初始化、`spi` 总线初始化和原来有何不同，对比 `components.c` 中的实现和原来的 `components.c` 又有什么不同，来理解和体会这种全新的方式。

第十四篇文件系统操作一网打尽

日期: 2013-11-08

第十五篇 RTGUI 之 LCD 驱动篇

日期: 2013-11-11

✧ 简单介绍

介绍一下版本情况, 目前 RTGUI 最新版本是 0.80, 这个版本的 GUI 至少要配合 RT-Thread1.2.0RC 版才行。

我们在使用 RTGUI 时, 只需要在配置文件 `rtconfig.h` 中开启宏

```
#define RT_USING_RTGUI
```

然后使用 `scons` 重新生成工程, 这样 `gui` 相关需要的文件即加入到了工程 (请先复习 [第十一篇-代码添加章节](#)), 有了 `gui` 源码还不够, 我们还需要根据自己所使用的屏幕写好 LCD 的驱动, 这才是整套 `gui` 最终要作用的地方。

✧ LCD 接口方式

说到 LCD 驱动, 这里先介绍一下最常用的两种 LCD 的接口方式: RGB 接口、MCU 接口。

MCU 接口

主要针对单片机领域在使用, 因此得名。MCU 接口的标准术语是 Intel 提出的 8080 总线标准, 因此在很多文档中用 I80 来指 MCU 接口屏。它可以分为 8080 模式和 6800 模式 (6800 是摩托罗拉提出的, 目前基本绝迹), 这两者之间主要是时序的区别。

MCU 接口屏的传输线和优缺点如下:

传输数据位: 8 位、9 位、16 位、18 位、24 位。

连线: CS、RS(寄存器选择) RD、WR、数据线。

优点: 控制简单方便, 无需时钟和同步信号。

缺点: 要耗费 GRAM, 所以难以做到大屏 (3.8 以上)。

对于 MCU 接口的 LCM (模块), 其内部的芯片就叫 LCD 驱动器。主要功能是对主机发过的数据/命令, 进行解析, 变成每个像素的 RGB 数据, 使之在屏上显示出来。这个过程不需要点、行、帧时钟, 我们常用的 LCD 驱动器有 ILI93xx、SSD1289、SSD1963 等等 (相信大家对这些控制器不陌生)。

RGB 接口

这是大屏采用较多的接口模式, RGB 接口屏的传输线和优缺点如下:

<http://shop73275611.taobao.com83> / 157

传输数据位：8 位、16 位、18 位、24 位。

连线：VSYNC、HSYNC、DOTCLK、CS、RESET、RS（有时需要）、数据线。

它的优缺点正好和 MCU 接口模式相反。

其他差异

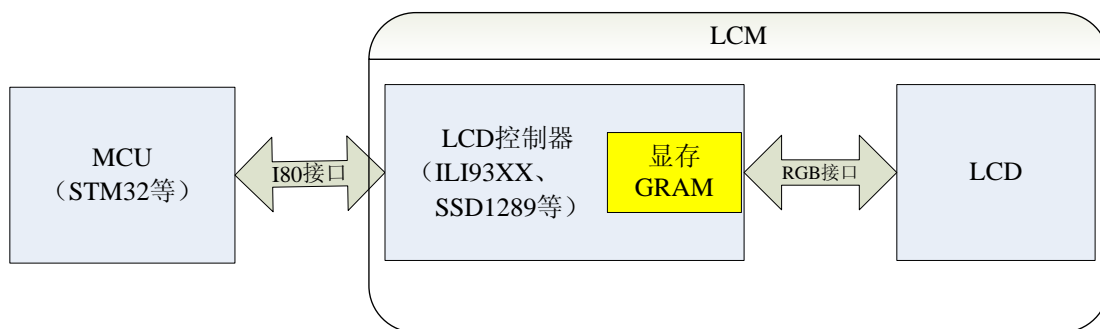
MCU 接口屏和 RGB 接口屏主要区别还在于显存的位置：

RGB 接口屏的显存是由系统内存充当的，因此其大小只受限于系统的内存大小，因此 RGB 接口屏可以做出较大尺寸；而 MCU 接口屏的设计之初只要考虑单片机的内存较小，因此都是把显存内置在 LCD 模块内部（即 LCD 控制器中的 GRAM）。然后软件通过专门显示命令来更新显存，因此 MCU 接口屏往往不能做得很大。

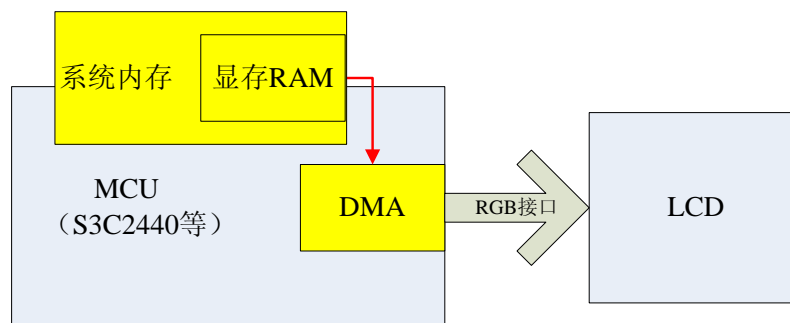
另外 MCU 接口屏和 RGB 接口屏的显示数据传输模式也有差别的：

RGB 接口屏只需显存组织好数据，启动显示后，MCU 的 DMA 控制器会自动把显存中的数据通过 RGB 接口送到 LCM；而 MCU 接口屏则需要发送画点的命令来修改 LCD 控制器内部的 GRAM。所以 RGB 接口屏显示速度明显比 MCU 接口屏快。

两种接口方式的系统框图



MCU 接口系统框图



RGB 接口系统框图

✧ LCD 驱动编写

以上的两种 LCD 接口方式，RTGUI 都支持，其中 RGB 接口，RTGUI 中对应的是 framebuffer 设备，本篇中我们主要介绍 MCU 接口方式的屏幕驱动。

LCD 驱动的编写，要遵循按照 RT-Thread 设备驱动架构。具体实现，主要是完成 `rt_device` 结构体的创建，并完成 LCD 的硬件初始化动作，最后向系统注册一个“图形设备”。

整个工作我们分三步来完成（怎么和把大象放进冰箱一样，呵呵）：

1、LCD 硬件初始化

完成相关 IO 口配置，并通过写控制寄存器的方式，初始化 LCD 相关寄存器（比如 LCD 显示方向、水平分辨率、垂直分辨率等等），下面给出 SSD1963 的初始化代码作为参考：

```
void lcd_initialization(void)
{
    //IO 口初始化，包括 FSMC 配置
    lcd_port_init();
    delays(100);
    GPIO_ResetBits(GPIOE, GPIO_Pin_5);
    delays(1000);
    GPIO_ResetBits(GPIOC, GPIO_Pin_6); /* RESET LCD */
    delays(1000);
    GPIO_SetBits(GPIOC, GPIO_Pin_6); /* release LCD */
    delays(1000);
    write_reg(0x00E2); //PLL multiplier, set PLL clock to 120M
    write_data(0x0023); //N=0x36 for 6.5M, 0x23 for 10M crystal
    write_data(0x0002);
    write_data(0x0004);
    write_reg(0x00E0); // PLL enable
    write_data(0x0001);
    delays(5);
    write_reg(0x00E0);
    write_data(0x0003);
    delays(5);
    write_reg(0x0001); // software reset
    delays(50);
    write_reg(0x00E6); //PLL setting for PCLK, depends on resolution
    write_data(0x0003);
    write_data(0x00ff);
    write_data(0x00ff);
    write_reg(0x00B0); //LCD SPECIFICATION
    write_data(0x0027);
```

```
write_data(0x0000);
write_data((HDP>>8)&0X00FF);//Set HDP
write_data(HDP&0X00FF);
write_data((VDP>>8)&0X00FF);//Set VDP
write_data(VDP&0X00FF);
write_data(0x0000);
write_reg(0x00B4);//HSYNC
write_data((HT>>8)&0X00FF);//Set HT
write_data(HT&0X00FF);
write_data((HPS>>8)&0X00FF);//Set HPS
write_data(HPS&0X00FF);
write_data(HPW);//Set HPW
write_data((LPS>>8)&0X00FF);//Set LPS
write_data(LPS&0X00FF);
write_data(0x0000);
write_reg(0x00B6);//VSYNC
write_data((VT>>8)&0X00FF);//Set VT
write_data(VT&0X00FF);
write_data((VPS>>8)&0X00FF);//Set VPS
write_data(VPS&0X00FF);
write_data(VPW);//Set VPW
write_data((FPS>>8)&0X00FF);//Set FPS
write_data(FPS&0X00FF);
write_reg(0x00BA);
write_data(0x000F);//GPIO[3:0] out 1
write_reg(0x00B8);
write_data(0x0007);//GPIO3=input, GPIO[2:0]=output
write_data(0x0001);//GPIO0 normal
write_reg(0x0036);//rotation
write_data(0x0000);
write_reg(0x00F0);//pixel data interface
write_data(0x0003);
delays(5);
write_reg(0x0029);//display on
write_reg(0x00d0);
write_data(0x000d);
delays(5);
// 数据总线测试, 用于测试硬件连接是否正常.
lcd_data_bus_test();
// 清屏
lcd_clear( Blue );
}
```

2、定义 LCD 设备，并完成其相关成员的实现

先来定义：

```
struct rt_device _lcd_device;
```

接下来看看这个结构体的细节，了解下都有哪些成员是需要我们来实现的

```
struct rt_device
{
    struct rt_object      parent; /**< inherit from rt_object */
    enum rt_device_class_type type; /**< device type */
    rt_uint16_t          flag; /**< device flag */
    rt_uint16_t          open_flag; /**< device open flag */
    rt_uint8_t           device_id; /**< 0 - 255 */

    /* device call back */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void*buffer);
    /* common device interface */
    rt_err_t (*init)(rt_device_t dev);
    rt_err_t (*open)(rt_device_t dev, rt_uint16_t oflag);
    rt_err_t (*close)(rt_device_t dev);
    rt_size_t (*read)(rt_device_t dev, rt_off_t pos, void*buffer, rt_size_t
size);
    rt_size_t (*write)(rt_device_t dev, rt_off_t pos, const void*buffer,
                    rt_size_t size);
    rt_err_t (*control)(rt_device_t dev, rt_uint8_t cmd, void*args);
    void*user_data; /**< device private data */
};
```

init、open、close、control 等函数和 user_data 私有域是我们来完成，不过 init、open、close 函数我们只需要做个空架子就好，最重要的是 control 函数和 user_data 私有域

LCD 作为图形设备，control 操作支持的命令有下面几个：

```
#define RTGRAPHIC_CTRL_RECT_UPDATE    0
#define RTGRAPHIC_CTRL_POWERON        1
#define RTGRAPHIC_CTRL_POWEROFF       2
#define RTGRAPHIC_CTRL_GET_INFO       3
#define RTGRAPHIC_CTRL_SET_MODE       4
#define RTGRAPHIC_CTRL_GET_EXT        5
```

我们只需简单的给 RTGRAPHIC_CTRL_GET_INFO 命令做好反馈，即：像素宽度、RGB 顺序还是 BGR 顺序（RTGRAPHIC_PIXEL_FORMAT_RGB565P 和 RTGRAPHIC_PIXEL_FORMAT_RGB565 的区别）、屏的水平分辨率、屏的垂直分辨率，具体参见下面代码：

```

static rt_err_t lcd_control(rt_device_t dev, rt_uint8_t cmd, void*args)
{
    switch(cmd)
    {
        case RTGRAPHIC_CTRL_GET_INFO:
        {
            struct rt_device_graphic_info *info;

            info =(struct rt_device_graphic_info*) args;
            RT_ASSERT(info != RT_NULL);
            info->bits_per_pixel =16;
            info->pixel_format =RTGRAPHIC_PIXEL_FORMAT_RGB565P;
            info->framebuffer =RT_NULL;//RGB 接口屏使用此接口函数
            info->width =LCD_WIDTH;
            info->height =LCD_HEIGHT;
        }
        break;

        case RTGRAPHIC_CTRL_RECT_UPDATE:
            /* nothing to be done */
            break;

        default:
            break;
    }
    return RT_EOK;
}

```

接下来说说 `user_data`, `rt_device` 作为一个通用的设备模型, 不可能满足所有的外设所需要的相关操作, `user_data` 私有域则给了我们一个将某个设备“差异化”途径, 在图形设备中, 我们使用下面的结构体来填充此私有域:

```

struct rt_device_graphic_ops lcd_ili_ops =
{
    rt_hw_lcd_set_pixel, // 在某个坐标上画像素点
    rt_hw_lcd_get_pixel, // 读取某个坐标上像素点, 按照 RGB 格式返回
    rt_hw_lcd_draw_hline, // 画水平线
    rt_hw_lcd_draw_vline, // 画垂直线
    rt_hw_lcd_draw_blit_line // 将某个数组中的数据 copy 至 lcd 的 gram 中
};

```

以上 5 个函数, 相信大家在裸机下都会写, 这里就不列出代码了。

3、注册 lcd 设备

经过前两步的相关函数准备，这里我们只需要参照下面代码，调用 lcd 底层初始化函数，然后填充图形设备结构体相关成员，最后调用 rt_device_register 函数向系统注册“lcd”设备就万事大吉了。

```
void ssd1963_init(void)
{
    lcd_Initializtion();           //硬件初始化
    /* register lcd device */      //成员函数填充
    _lcd_device.type = RT_Device_Class_Graphic;
    _lcd_device.init = lcd_init;
    _lcd_device.open = lcd_open;
    _lcd_device.close = lcd_close;
    _lcd_device.control = lcd_control;
    _lcd_device.read = RT_NULL;
    _lcd_device.write = RT_NULL;

    _lcd_device.user_data = &lcd_ili_ops;

    /* register graphic device driver */ //注册设备
    rt_device_register(&_amp;_lcd_device, "lcd",
        RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_STANDALONE);
}
```

◇ 驱动测试

本篇的例程是在上一篇的例程基础上加入了 RTGUI 代码后的 MDK 工程，例子中提供了多种 LCD 控制器的驱动模板（SSD1289、SSD1963、RA8875、ILI93xx 等），我们编译烧录后运行，在 finsh 中运行 list_device 命令，来查看我们刚才注册的图形设备：

```
finsh />list_device()
device      type
-----
rtc         RTC
lcd         Graphic Device
sdo         Block Device
flash0      Block Device
uart3       Character Device
spi22       SPI Device
spi21       SPI Device
spi20       SPI Device
spi2        SPI Bus
            0, 0x00000000
finsh />
```

上面我们只看到图形设备注册成功，要来验证一下驱动怎么办？OK，使用 RTGUI 自带的 example 例子是最直接的方式：

像对应例程中的 SConstruct 文件那样，添加如下代码，使 scons 生成工程时能将 GUI example 相关代码加入到其中：

```
if GetDepend('RT_USING_RTGUI'):
    objs = objs + SConscript(RTT_ROOT + '/examples/gui/SConscript',
variant_dir='build/examples/gui', duplicate=0)
```

添加完 GUI example 代码后，在初始化线程中调用以下函数，使 demo 得以运行：

```
int ui_init(void)
{
    rt_device_t device;

    device = rt_device_find("lcd");
    if(device == RT_NULL)
    {
        rt_kprintf("no graphic device in the system.\n");
        return-1;
    }

    /* re-set graphic device */
    rtgui_graphic_set_device(device);

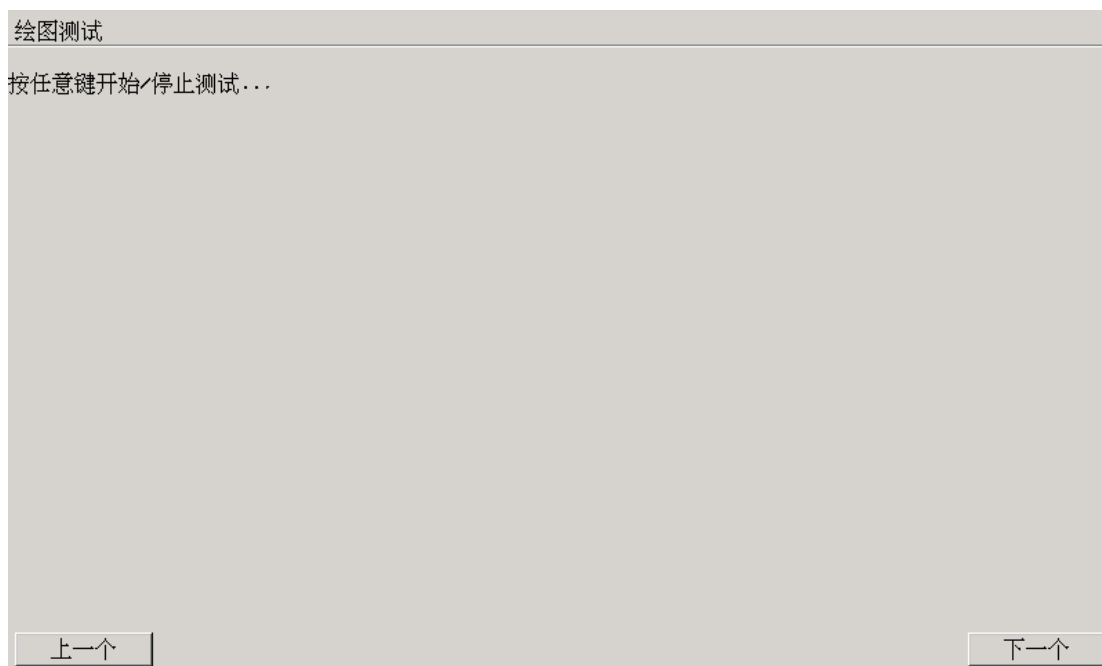
    {
        externvoid application_init();
        application_init();
    }
    return0;
}
```

```
void rt_init_thread_entry(void*parameter)
{
    rt_platform_init();

    /* initialization RT-Thread Components */
    rt_components_init();

#ifdef RT_USING_RTGUI
    {
        int ui_init(void);
        ui_init();
    }
#endif /* RT_USING_RTGUI */
}
```

如果驱动编写正确，我们运行示例代码将会看到如下界面：



Realtouch 上运行例程代码的截图

有童鞋可能会问，这个截图怎么来的，在 `rtconfig.h` 配置文件中打开如下宏，使能 bmp 图片支持

```
#define RTGUI_IMAGE_BMP
```

这样在生成的工程中将会有有一个名为 `screenshot` 的 `finsh` 命令，我们在 `finsh` 端运行截图命令就可以了：`finsh->screenshot("/SD/xx.bmp")` 哈哈，`finsh` 就是如此惊艳！

对于 GUI 来说，单单有个 LCD 驱动，并不够，因为用户还需要使用输入设备和系统进行交互。接下来的篇章中，我们继续来讲解常用的两种输入设备：`keyboard` 和 `touch_panel` 的驱动编写。

第十六篇 RTGUI 之 keyboard 驱动篇

日期：2013-11-12

✧ keyboard 结构体

RTGUI 中的按键事件处理机制，是按照一套完整的 PC 键盘来设计的，整个按键对象用如下的结构体来表示：

```
struct rtgui_event_kbd
{
    _RTGUI_EVENT_WIN_ELEMENTS

    rt_uint16_t type; /* key down or up */
    rt_uint16_t key; /* current key */
    rt_uint16_t mod; /* current key modifiers */
    rt_uint16_t unicode; /* translated character */
};
```

RTGUI_EVENT_WIN_ELEMENTS 成员表示事件所对应的窗体元素

type 成员表示按键事件是按下还是抬起，可取的值有两个：

```
RTGUI_KEYDOWN, /* Keys pressed */
RTGUI_KEYUP, /* Keys released */
```

key 成员表示当前的按键值，可取的值有很多：包括数字键值、字母键值、功能键值、方向键值等，比如（只是一部分取值）：

```
RTGUIK_UP
RTGUIK_DOWN
RTGUIK_RIGHT
RTGUIK_LEFT
RTGUIK_INSERT
RTGUIK_HOME
RTGUIK_END
RTGUIK_PAGEUP
RTGUIK_PAGEDOWN
```

```
RTGUIK_LEFTBRACKET
RTGUIK_BACKSLASH
RTGUIK_RIGHTBRACKET
RTGUIK_CARET
RTGUIK_UNDERSCORE
RTGUIK_BACKQUOTE
RTGUIK_a
RTGUIK_b
```

```
RTGUIK_c
RTGUIK_d
RTGUIK_e
```

`mod` 成员表示按键是组合键还是正常的单按键，可取值如下：

```
RTGUI_KMOD_NONE
RTGUI_KMOD_LSHIFT
RTGUI_KMOD_RSHIFT
RTGUI_KMOD_LCTRL
RTGUI_KMOD_RCTRL
RTGUI_KMOD_LALT
RTGUI_KMOD_RALT
RTGUI_KMOD_LMETA
RTGUI_KMOD_RMETA
RTGUI_KMOD_NUM
RTGUI_KMOD_CAPS
RTGUI_KMOD_MODE
RTGUI_KMOD_RESERVED
```

一般我们正常使用按键时，将 `mod` 设为 `RTGUI_KMOD_NONE` 就行了。

`unicode` 成员用于将键值转换为字符，这在输入法中能派上用场，一般应用不用关心。

✧ keyboard 驱动编写

要让按键和 GUI 产生交互，需要在按键动作产生后，向 GUI 服务端上报按键动作，这个上报按键的动作是比较简单的，这里我们举例来说明。

首先我们要先检测到按键，本篇例子中我们使用定时扫描的方法来进行按键检测。我们创建一个软件定时器，每隔 20ms 定时调用 `time_out` 函数进行按键的扫描，并进行短按、长按、连接的相关判断，最后把需要的键值进行上报。

```
int rt_hw_key_init(void)
{
    GPIO_Configuration();

    key =(struct rtgui_key*)rt_malloc (sizeof(struct rtgui_key));
    if(key == RT_NULL)
        return-1; /* no memory yet */

    /* init keyboard event */
    RTGUI_EVENT_KBD_INIT(&(key->kbd_event));
    key->kbd_event.wid = RT_NULL;
    key->kbd_event.mod  = RTGUI_KMOD_NONE; /* 一般单按键处理*/
```

```

    key->kbd_event.unicode = 0;

    key->key_last =0;
    key->key_current =0;
    key->key_get =0;
    key->key_debounce_count =0;
    key->key_long_count =0;
    key->key_special_count =0;
    key->key_relese_count =0;
    key->key_flag =0;

    /* create 1/50=20ms timer */
    key->poll_timer = rt_timer_create("key", key_timeout, RT_NULL,
                                     RT_TICK_PER_SECOND/50, RT_TIMER_FLAG_PERIODIC);

    /* 启动定时器 */
    if(key->poll_timer != RT_NULL)
        rt_timer_start(key->poll_timer);
    return 0;
}

```

```

/* 检测判断部分略去*/
.....
/* 检测到按键后向系统上报键值 */
key->kbd_event.key = RTGUIK_UNKNOWN;

if(key->key_get)
{
    //rt_kprintf("key = %x \n",key->key_get);
    if(((key->key_get)==C_UP_KEY)&&((key->key_flag)& C_FLAG_SHORT))
        key->kbd_event.key = RTGUIK_UP;

    if(((key->key_get)==C_DOWN_KEY)&&((key->key_flag)& C_FLAG_SHORT))
        key->kbd_event.key = RTGUIK_DOWN;

    if(((key->key_get)==C_LEFT_KEY)&&((key->key_flag)& C_FLAG_SHORT))
        key->kbd_event.key = RTGUIK_LEFT;

    if(((key->key_get)==C_RIGHT_KEY)&&((key->key_flag)& C_FLAG_SHORT))
        key->kbd_event.key = RTGUIK_RIGHT;

    //if (((key->key_get)==C_STOP_KEY) &&((key->key_flag) & C_FLAG_SHORT))
    //    key->kbd_event.key = RTGUIK_UP;
}

```

```
//if (((key->key_get)==C_MENU_KEY) &&((key->key_flag) & C_FLAG_SHORT))
// key->kbd_event.key = RTGUIK_UP;
if((key->key_get)==C_ENTER_KEY)
    key->kbd_event.key = RTGUIK_RETURN;

if((key->key_get)==C_HOME_KEY)
    key->kbd_event.key = RTGUIK_HOME;
}

key->kbd_event.type = RTGUI_KEYDOWN;

if(key->kbd_event.key != RTGUIK_UNKNOWN)
{
    /* 先上报按键按下*/
    rtgui_server_post_event(&(amp;key->kbd_event.parent),
                           sizeof(key->kbd_event));

    /* delay to post up event */
    rt_thread_delay(2);
    /* 再上报按键松开, 完成一个从按下到松开的组合*/
    key->kbd_event.type = RTGUI_KEYUP;
    rtgui_server_post_event(&(amp;key->kbd_event.parent),
                           sizeof(key->kbd_event));
}
```

有了按键驱动, 我们就可以使用左右键来进行 GUI example 例子的切换浏览了.

操作一下, 看是否如我们所想呢。

第十七篇 RTGUI 之 touch panel 驱动篇

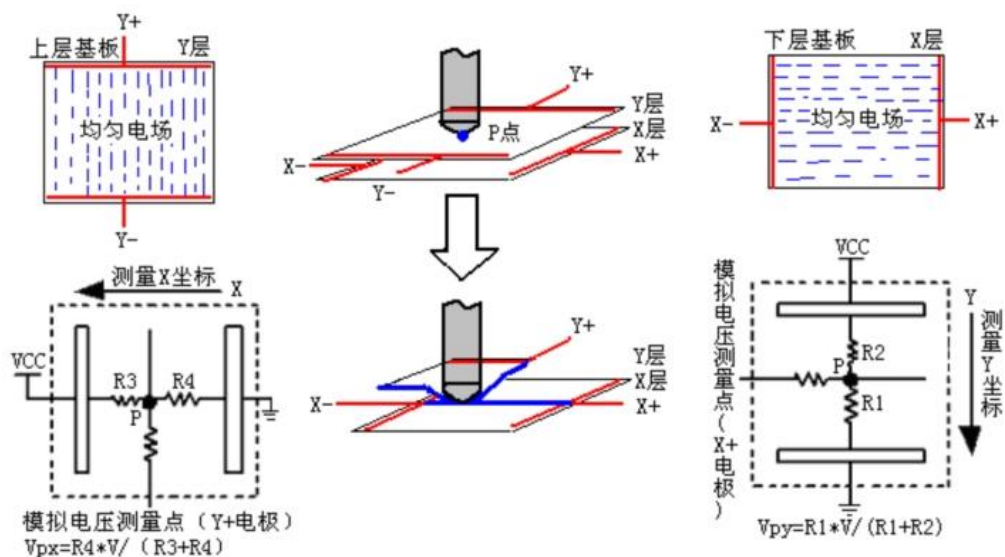
日期: 2013-11-21

触摸屏分 2 大类, 电阻式和电容式, 我们的实验板上用的是四线电阻式触摸屏, 这里我们讲的也是针对电阻屏。

✧ 触摸屏工作原理

PS 原理性的讲解大家选择性浏览

四线电阻式触摸屏, 主要由两层镀有 ITO 镀层的薄膜组成。其中一层在屏幕的左右边缘各有一条垂直总线, 另一层在屏幕的底部和顶部各有一条水平总线, 如果在一层薄膜的两条总线上施加电压, 在 ITO 镀层上就会形成均匀电场。当使用者触击触摸屏时, 触击点处两层薄膜就会接触, 在另一层薄膜上就可以测量到接触点的电压值, 如下图:



为了在 X 轴方向进行测量, 将左侧总线偏置为 0V, 右侧总线偏置为 VCC。将顶部或底部总线连接到 ADC, 当顶层和底层相接触时即可作一次测量。

为了在 Y 轴方向进行测量, 将顶部总线偏置为 VCC, 底部总线偏置为 0V。将 ADC 输入端接左侧总线或右侧总线, 当顶层与底层相接触时即可对电压进行测量。

如上图, 测量出来的电压值与接触点的位置线性相关, 即可以由 VPX 和 VPY 分别计算出接触点 P 的 X 和 Y 坐标。

在实际测量中, 控制电路会交替在 X 和 Y 电极组上施加 VCC 电压, 进行电压测量和计算接触点的坐标。举例说明测量流程:

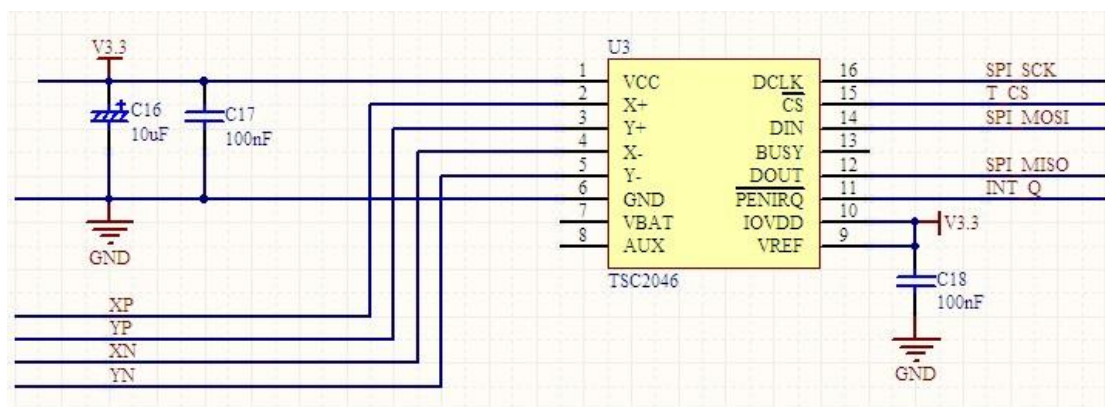
第一步, 在 X+ 上施加 VCC, X- 上施加 0V 电压, 测量 Y+ (或 Y-) 电极上的电压值 VPX, 计算出接触点 P 的 X 坐标;

第二步，在 Y+上施加 VCC，Y-上施加 0V 电压，测量 X+(或 X-)电极上的电压值 VPY，计算出接触点 P 的 Y 坐标；

以上两步组成一个测量周期，可以得到一组(X,Y)坐标。

✧ 实际触摸数据采集

我们魔笛 F1 实验板上采用的是集成了 4 线电阻式触摸屏和触摸采集芯片的 LCD 模块，其中触摸芯片采用的是 XPT2046，这也是一款很常用的触摸芯片。触摸芯片一边连接触摸屏上的 4 根采集线 XP、XN、YP、YN，一边通过 SPI 方式（T_CS、SPI_SCK、SPI_MISO、SPI_MOSI）与 MCU 通信。图中的 INT_Q 信号是触摸芯片给出的触摸中断信号（此引脚平时为高电平，当我们触摸屏幕时变为低电平，也成为“笔中断”），我们可以使用中断方式来处理触摸，提高 CPU 使用效率。



整个触摸的处理，涉及到以下几个部分：

touch 设备注册；

中断处理；

SPI 接口通信；

触摸 X、Y 值上报；

触摸校准。

我们可以用一个结构体将需要用到的东西进行打包：

```
ruct rtgui_touch_device
{
    struct rt_device parent; /* 用于注册设备 */

    rt_uint16_t x, y; /* 记录读取到的位置值 */

    rt_bool_t calibrating; /* 触摸校准标志 */
}
```

```

rt_touch_calibration_func_t calibration_func; /* 触摸函数函数指针 */

rt_uint16_t min_x, max_x; /* 校准后 X 方向最小最大值 */
rt_uint16_t min_y, max_y; /* 校准后 Y 方向最小最大值 */

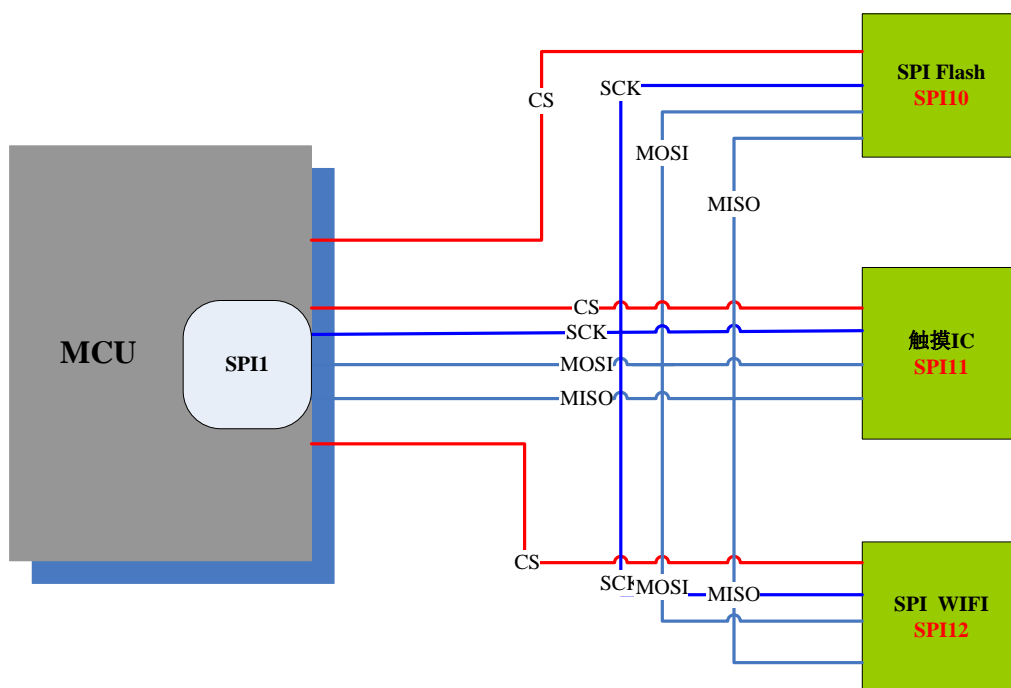
struct rt_spi_device * spi_device; /* SPI 设备用于通信 */
struct rt_event event; /* 事件同步, 用于“笔中断” */
};
static struct rtgui_touch_device *touch = RT_NULL;

```

下面以魔笛 F1 中的触摸驱动 (touch.c) 为原型, 对各部分来一个“剖析”

✧ SPI 接口通信

魔笛 F1 实验板上, MCU 通过 SPI1 和触摸 IC 通信, 除此之外 SPI1 还与其他 SPI 外设进行通信, 系统框图如下所示:



对于 SPI 操作, 我们使用 SPI 驱动框架, 即: 将 MCU 的 SPI1 注册为一个“SPI Bus”, 将与其连接的从机设备都“挂接”到这个“Bus”上, 从设备之间的操作互斥则由系统去处理。

总线的注册和设备的挂接代码实现是这样的:

```

void rt_hw_spi1_init(void)
{
    static struct stm32_spi_bus stm32_spi;
    GPIO_InitTypeDef GPIO_InitStructure;

```

```

        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |
                                RCC_APB2Periph_AFIO ,ENABLE);
#ifdef SPI_USE_DMA
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
#endif
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA,&GPIO_InitStructure);
    /* SPI1BUS 注册*/
    stm32_spi_register(SPI1, &stm32_spi, "spi1");
}
{ //spi flash cs
    static struct rt_spi_device spi_device;
    static struct stm32_spi_cs spi_cs;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE ,ENABLE);
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    spi_cs.GPIOx = GPIOE;
    spi_cs.GPIO_Pin = GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Pin = spi_cs.GPIO_Pin;

    GPIO_Init(spi_cs.GPIOx,&GPIO_InitStructure);
    GPIO_SetBits(spi_cs.GPIOx, spi_cs.GPIO_Pin);
    /* SPI10 设备挂接*/
    rt_spi_bus_attach_device(&spi_device, "spi10", "spi1",
                             (void*)&spi_cs);
}
/* attach cs */
{ //touch cs
    static struct rt_spi_device spi_device;
    static struct stm32_spi_cs spi_cs;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC ,ENABLE);
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    spi_cs.GPIOx = GPIOC;
    spi_cs.GPIO_Pin = GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Pin = spi_cs.GPIO_Pin;
    GPIO_Init(spi_cs.GPIOx,&GPIO_InitStructure);

```

```

    GPIO_SetBits(spi_cs.GPIOx, spi_cs.GPIO_Pin);
    /* SPI11 设备挂载*/
    rt_spi_bus_attach_device(&spi_device, "spi11", "spi1",
                             (void*)&spi_cs);
}

/* attach cs */
{ //wifi cs
    static struct rt_spi_device spi_device;
    static struct stm32_spi_cs spi_cs;
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    spi_cs.GPIOx = GPIOE;
    spi_cs.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Pin = spi_cs.GPIO_Pin;
    GPIO_Init(spi_cs.GPIOx, &GPIO_InitStructure);
    GPIO_SetBits(spi_cs.GPIOx, spi_cs.GPIO_Pin);
    /* SPI12 设备挂载*/
    rt_spi_bus_attach_device(&spi_device, "spi12", "spi1",
                             (void*)&spi_cs);
}
}

```

对于不同的 SPI 设备，其操作时具体配置也许会有差别，比如：数据宽度、时钟速率、模式（由 CPHA、CPOL 决定）这些方面的差异。

阅读 XPT2046 的数据手册，我们可以得到对于它的操作配置：

数据宽度：8 位

时钟速率：最快可达 4M，这里我们取 500K，以保证质量

模式：mode0，MSB First（CPHA=0、CPOL=0 上升沿读取数据，下降沿所存数据，始终空闲时保持低电平，高位优先传输）

于是，我们在具体操作前，先将 SPI11 进行初始化配置：

```

    spi_device = (struct rt_spi_device *)rt_device_find(spi_device_name);
    if(spi_device == RT_NULL)
    {
        rt_kprintf("spi device %s not found!\r\n", spi_device_name);
        return -RT_ENOSYS;
    }

```

```

/* config spi */
{
    struct rt_spi_configuration cfg;
    cfg.data_width = 8;
    cfg.mode = RT_SPI_MODE_0 | RT_SPI_MSB; /* SPI Compatible Modes 0 */
    cfg.max_hz = 500 * 1000; /* 500K */
    rt_spi_configure(spi_device,&cfg);
}

```

在需要读取触摸值时，我们分别对 X、Y 方向的数据进行读取：

```

uint8_t i, j, k, min;
uint16_t temp;
rt_uint16_t tmpxy[2][SAMP_CNT];
uint8_t send_buffer[1];
uint8_t recv_buffer[2];
for(i=0; i<SAMP_CNT; i++)
{
    send_buffer[0] = TOUCH_MSR_X;
    rt_spi_send_then_recv(touch->spi_device,
                          send_buffer,1, recv_buffer,2);
    #if defined(_ILI_HORIZONTAL_DIRECTION_)
        tmpxy[1][i]=(recv_buffer[0]&0x7F)<<4;
        tmpxy[1][i]|=(recv_buffer[1]>>3)&0x0F;
    #else
        tmpxy[0][i]=(recv_buffer[0]&0x7F)<<4;
        tmpxy[0][i]|=(recv_buffer[1]>>3)&0x0F;
    #endif
    send_buffer[0] = TOUCH_MSR_Y;
    rt_spi_send_then_recv(touch->spi_device,
                          send_buffer,1, recv_buffer,2);
    #if defined(_ILI_HORIZONTAL_DIRECTION_)
        tmpxy[0][i]=(recv_buffer[0]&0x7F)<<4;
        tmpxy[0][i]|=(recv_buffer[1]>>3)&0x0F;
    #else
        tmpxy[1][i]=(recv_buffer[0]&0x7F)<<4;
        tmpxy[1][i]|=(recv_buffer[1]>>3)&0x0F;
    #endif
}
/*再次打开触摸中断*/
send_buffer[0]=1<<7;
rt_spi_send(touch->spi_device, send_buffer,1);

```

```

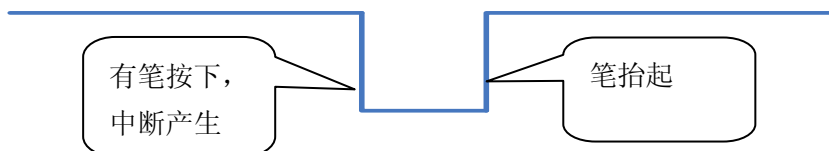
/* 多次采样计算平均 */
{
    rt_uint32_t total_x =0;
    rt_uint32_t total_y =0;
    for(k=0; k<2; k++)
    {
        // 冒泡排序
        for(i=0; i<SAMP_CNT-1; i++)
        {
            min=i;
            for(j=i+1; j<SAMP_CNT; j++)
            {
                if(tmpxy[k][min]> tmpxy[k][j])
                    min=j;
            }
            temp = tmpxy[k][i];
            tmpxy[k][i]= tmpxy[k][min];
            tmpxy[k][min]= temp;
        }
        //check value for Valve value
        if((tmpxy[k][SAMP_CNT_DIV2+1]-tmpxy[k][SAMP_CNT_DIV2-2])> SH)
        {
            return;
        }
    }
    // 抽取认为可用值进行平均
    total_x=tmpxy[0][SAMP_CNT_DIV2-2]+tmpxy[0][SAMP_CNT_DIV2-1]+
        tmpxy[0][SAMP_CNT_DIV2]+tmpxy[0][SAMP_CNT_DIV2+1];

    total_y=tmpxy[1][SAMP_CNT_DIV2-2]+tmpxy[1][SAMP_CNT_DIV2-1]+tmpxy[1][SAMP_C
    NT_DIV2]+tmpxy[1][SAMP_CNT_DIV2+1];
    //calculate average value
    touch->x=total_x>>2;
    touch->y=total_y>>2;
    printf("touch->x:%d touch->y:%d\r\n", touch->x, touch->y);
}/* calculate average */
}/* read touch */

```

✧ 中断处理

我们利用 XPT2046 提供的“笔中断”，来启动数据采集



将“笔中断”IO口（INT_Q）配置为下降沿触发的中断方式：

```
rt_inline void touch_int_cmd(FunctionalState NewState)
{
    EXTI_InitTypeDef EXTI_InitStructure;

    /* Configure EXTI */
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger =EXTI_Trigger_Falling;

    EXTI_InitStructure.EXTI_LineCmd = NewState;

    EXTI_ClearITPendingBit(EXTI_Line0);
    EXTI_Init(&EXTI_InitStructure);
}
```

在对应的中断处理函数中，向数据采集线程发送一个“事件”：

```
void EXTI0_IRQHandler(void)
{
    /*暂时关闭触摸中断响应，待处理完相关数据后再开启*/
    touch_int_cmd(DISABLE);

    rt_event_send(&touch->event, 1);

    EXTI_ClearITPendingBit(EXTI_Line0);
}
```

然后数据采集线程以阻塞（死等）的方式去等待此“事件”，当等到后才进行相关处理

```
while(1)
{
    /* 接收到触摸中断事件 */
    if(rt_event_rcv(&touch->event,1,
                    RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                    RT_WAITING_FOREVER,
                    &event_value)== RT_EOK)
    {
        /*具体处理数据采集数据上报等*/
        .....
    }
}
```

✧ Touch 设备注册

```

/* init device structure */
touch->parent.type = RT_Device_Class_Unknown;
touch->parent.init = rtgui_touch_init;
touch->parent.control = rtgui_touch_control;
touch->parent.user_data = RT_NULL;

/* register touch device to RT-Thread */
rt_device_register(&(touch->parent), "touch", RT_DEVICE_FLAG_RDWR);

/* 创建触摸数据采集线程 */
touch_thread = rt_thread_create("touch",
                                touch_thread_entry, RT_NULL,
                                1024, RTGUI_SVR_THREAD_PRIORITY-1, 1);
if(touch_thread != RT_NULL) rt_thread_startup(touch_thread);

```

设备初始化函数，主要是进行一些中断配置、并将 XPT2046 使能：

```

/* RT-Thread Device Interface */
static rt_err_t rtgui_touch_init (rt_device_t dev)
{
    uint8_t send;
    struct rtgui_touch_device * touch_device =(struct rtgui_touch_device
*)dev;
    NVIC_Configuration();
    EXTI_Configuration();

    send = START | DIFFERENTIAL | POWER_MODE0;
    rt_spi_send(touch_device->spi_device, &send, 1); return RT_EOK;
}

```

control 函数主要为校准时提供接口：

```

static rt_err_t rtgui_touch_control (rt_device_t dev, rt_uint8_t
cmd, void*args)
{
    switch(cmd)
    {
        Case RT_TOUCH_CALIBRATION: /* 开始进行触摸校准 */
            touch->calibrating = RT_TRUE;
            touch->calibration_func =(rt_touch_calibration_func_t)args;
            break;

        case RT_TOUCH_NORMAL: /* 正常模式 */
            touch->calibrating = RT_FALSE;
            break;
    }
}

```

```

    case RT_TOUCH_CALIBRATION_DATA: /* 校准完成, 进行校准数据更新 */
    {
        struct calibration_data* data;
        data = (struct calibration_data*) args;
        //update
        touch->min_x = data->min_x;
        touch->max_x = data->max_x;
        touch->min_y = data->min_y;
        touch->max_y = data->max_y;
    }
    break;
}
return RT_EOK;
}

```

✧ 触摸 X Y 坐标值上报

上一篇中讲到的 key 驱动是将按键值依照 PC 键盘的键值向系统上报；我们的触摸屏是将 X、Y 方向的坐标依照 PC 鼠标的形式向系统上报。RT-Thread 中采用 `rtgui_event_mouse` 来描述一个鼠标事件：

```

struct rtgui_event_mouse
{
    _RTGUI_EVENT_WIN_ELEMENTS

    rt_uint16_t x, y;
    rt_uint16_t button;
};

```

其中 `button` 成员可以是如下宏：

```

#define RTGUI_MOUSE_BUTTON_LEFT      0x01 /* 左击 */
#define RTGUI_MOUSE_BUTTON_RIGHT     0x02 /* 右击 */
#define RTGUI_MOUSE_BUTTON_MIDDLE    0x03 /* 滚轮中键 */
#define RTGUI_MOUSE_BUTTON_WHEELUP   0x04 /* 滚轮上滚 */
#define RTGUI_MOUSE_BUTTON_WHEELDOWN 0x08 /* 滚轮下滚 */

```

和如下宏：

```

#define RTGUI_MOUSE_BUTTON_DOWN      0x10 /* 按下 */
#define RTGUI_MOUSE_BUTTON_UP        0x20 /* 抬起 */

```

的组合。

当然对于触摸屏来说，我们应该知道，只能有这么两种组合：

`RTGUI_MOUSE_BUTTON_LEFT | RTGUI_MOUSE_BUTTON_UP`

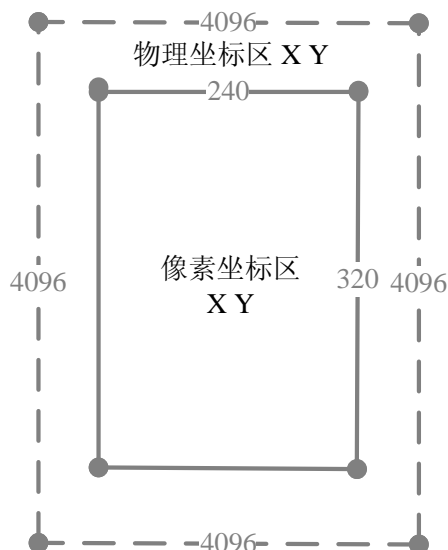
RTGUI_MOUSE_BUTTON_LEFT|RTGUI_MOUSE_BUTTON_DOWN

这就是一个模拟鼠标单击的过程。

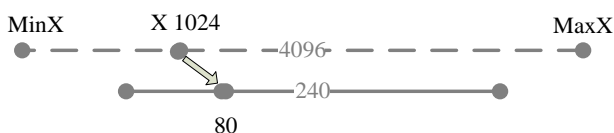
我们直接读取到的 X、Y 坐标值是“物理坐标”，这样的坐标是不能直接向系统上报的，需要转换成“像素坐标”后才可以进行上报。

触摸屏和 LCD 显示屏是两个不同的器件，触摸检测 IC XPT2046 返回的 X、Y 坐标值是 12 位的一个整数，范围是 0-4095，对应触摸笔实际接触到的物理点，我们将这个直接读到的物理点的 X、Y 坐标称之为“物理坐标”；

我们使用的 LCD 显示屏，本身有一个分辨率，魔笛 F1 配套的 3.2 寸屏分辨率是 240*320，即水平方向 240 个像素点，垂直方向 320 个像素点，LCD 上的各种 UI 控件的位置都是按照像素点的位置来描述的。我们将以像素来表示的坐标称为“像素坐标”，比如屏幕左上角的像素坐标就是（0，0）。



从“物理坐标”进行比例缩放就可以得到“像素坐标”，这里以 X 方向为例来说明



如上图，“物理坐标”上的一个点 X ‘=1024，通过下面公式计算即可得到其在像素坐标中的位置。

$$X = (X' - \text{MinX}) / (\text{MaxX} - \text{MinX}) * 240$$

之前我们采集到的 X、Y 坐标，进行平均处理后，就是根据以上公式，进行了坐标变换：

```

    if(touch->max_x > touch->min_x)
    {
        touch->x = (touch->x-touch->min_x) * X_WIDTH/(touch->max_x-touch->min_x);
    }
    else
    {
        /*默认思维是左边算作MinX，右边算作MaxX，实际上有可能正好相反，此时，公式同理调整 */
        touch->x=(touch->min_x-touch->x)* X_WIDTH/(touch->min_x-touch->max_x);
    }

    if(touch->max_y > touch->min_y)
    {
        touch->y = (touch->y - touch->min_y) * Y_WIDTH
        /(touch->max_y-touch->min_y);
    }
    else
    {
        /*默认思维是上边算作MinY，下边算作MaxY，实际上有可能正好相反，此时，公式同理调整 */
        touch->y=(touch->min_y-touch->y)*Y_WIDTH /(touch->min_y-touch->max_y);
    }

```

得到“像素坐标”后，我们就可以在触摸笔按下和抬起时分别向系统上报坐标信息了：

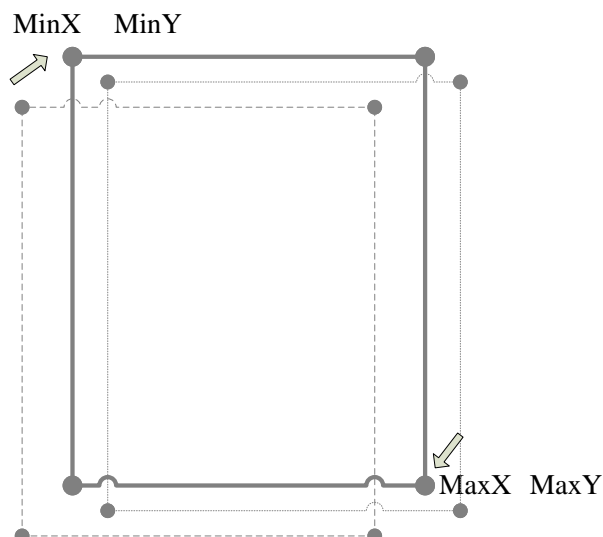
```

staticvoid touch_thread_entry(void*parameter)
{
    .....
    RTGUI_EVENT_MOUSE_BUTTON_INIT(&emouse);
    emouse.wid = RT_NULL;
    while(1)
    { /* 接收到触摸中断事件 */
        if(rt_event_rcv(&touch->event, 1,
                        RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                        RT_WAITING_FOREVER,&event_value)== RT_EOK)
        {
            while(1)
            {
                if (IS_TOUCH_UP())
                { /* 触摸笔抬起 */
                    /* touch up */
                    emouse.button = (RTGUI_MOUSE_BUTTON_LEFT |
                                    RTGUI_MOUSE_BUTTON_UP);
                    .....
                    { /* 向ui 发送触摸坐标 */
                        rtgui_server_post_event(&emouse.parent,
                                                sizeof(struct rtgui_event_mouse));
                    }
                }
            }
        }
    }
}

```

上面再讲坐标系转换时，提到了 **MinX**（X 方向最小值）、**MaxX**（X 方向最大值）、**MinY**（Y 方向最小值）、**MaxY**（Y 方向最大值）。这些值从何而来呢？

<http://shop73275611.taobao.com>108 / 157

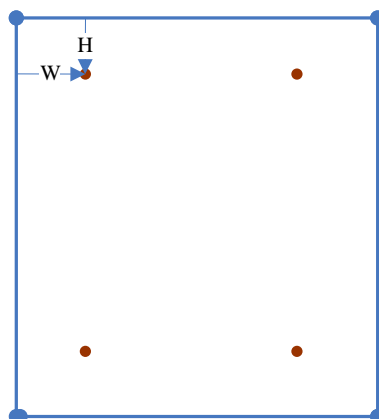


程序中的下面宏定义就是指的这个意思

```
#if defined(_ILI_HORIZONTAL_DIRECTION_)
#define MIN_X_DEFAULT 141
#define MAX_X_DEFAULT 1912
#define MIN_Y_DEFAULT 1808
#define MAX_Y_DEFAULT 140
#else
#define MIN_X_DEFAULT 0x7bd
#define MAX_X_DEFAULT 0x20
#define MIN_Y_DEFAULT 0x53
#define MAX_Y_DEFAULT 0x79b
#endif
```

但是对于一批屏幕来说，在安装时，不可避免的会存在一定的偏差，如：旋转、平移。（如上图中的虚线框所示），另外，触摸屏本身的材质也会有差异，并且随着时间的推移，其参数也会变化。所以，不可能每一个屏得到的这 4 个特征值都完全相同，因此上面的宏定义也失去了其通用性，所以我们需要引入一个“标定”机制，即：校准。

触摸校准方法比较多，RT-GUI 源码中为我们提供了一种基本的触摸校准方法（calibration.c），具体就是采集屏幕上的 4 个特征“像素坐标”上的“物理坐标”数据（如下图中的红色点位置），然后通过简单运算得到 MinX、MaxX、MinY、MaxY。



上图中的 4 个红色坐标是我们要采集的“物理坐标”，我们称之为 $PP(X_x, Y_x)$ ，当然它们所对应的“像素坐标” $PL(X_x, Y_x)$ 我们是事先设定好的，假设屏幕的宽度是 `width`，高度为 `height`，四个“像素坐标”距离屏幕边缘的 X 方向距离是 `CALIBRATION_WIDTH`，距离屏幕边缘的 Y 方向距离是 `CALIBRATION_HEIGHT`，这样我们的四个“像素坐标”分别是：

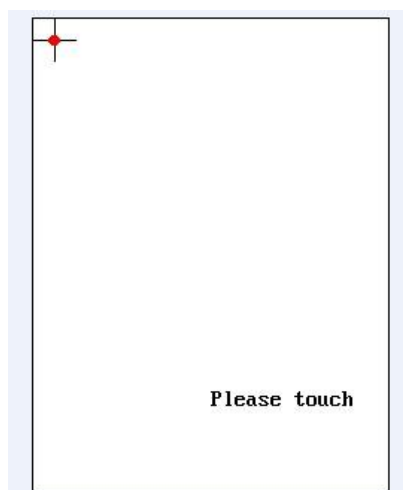
`PL(CALIBRATION_WIDTH, CALIBRATION_HEIGHT)`

`PL(width-CALIBRATION_WIDTH, CALIBRATION_HEIGHT)`

`PL(CALIBRATION_WIDTH, height-CALIBRATION_HEIGHT)`

`PL(width-CALIBRATION_WIDTH, height-CALIBRATION_HEIGHT)`

其对应的“物理坐标”在每次触摸校准时采集：



```
if((touch->calibrating == RT_TRUE)&&(touch->calibration_func != RT_NULL))
{
    /* 调用校准函数 */
    touch->calibration_func(emouse.x, emouse.y);
}
```



```
}
```

`calibration_func` 这个函数指针所指的函数原型在 `calibration.c` 中定义，此函数会根据我们在屏幕上点触的 4 个点的“物理坐标”运算出 `MinX`、`MaxX`、`MinY`、`MaxY`：

```
static void calibration_data_post(rt_uint16_t x, rt_uint16_t y)
{
    if(calibration_ptr == RT_NULL)
        return;
    switch(calibration_ptr->step)
    {
        case CALIBRATION_STEP_LEFTTOP:
            calibration_ptr->data.min_x = x;
            calibration_ptr->data.min_y = y;
            break;
        case CALIBRATION_STEP_RIGHTTOP:
            calibration_ptr->data.max_x = x;
            calibration_ptr->data.min_y =
                (calibration_ptr->data.min_y + y) / 2;
            break;
        case CALIBRATION_STEP_LEFTBOTTOM:
            calibration_ptr->data.min_x =
                (calibration_ptr->data.min_x + x) / 2;
            calibration_ptr->data.max_y = y;
            break;
        case CALIBRATION_STEP_RIGHTBOTTOM:
            calibration_ptr->data.max_x =
                (calibration_ptr->data.max_x + x) / 2;
            calibration_ptr->data.max_y =
                (calibration_ptr->data.max_y + y) / 2;
            break;
        case CALIBRATION_STEP_CENTER:
            /* calibration done */
            {
                rt_uint16_t w, h;

                struct rtgui_event_command ecmd;
                RTGUI_EVENT_COMMAND_INIT(&ecmd);
                ecmd.wid = calibration_ptr->win;
                ecmd.command_id = TOUCH_WIN_CLOSE;
                /* calculate calibrated data */
                if(calibration_ptr->data.max_x > calibration_ptr->data.min_x)
                    w = calibration_ptr->data.max_x - calibration_ptr->data.min_x;
                else
                    w = calibration_ptr->data.min_x - calibration_ptr->data.max_x;
```

```

w =(w /(calibration_ptr->width -2* CALIBRATION_WIDTH))*
    CALIBRATION_WIDTH;

if(calibration_ptr->data.max_y > calibration_ptr->data.min_y)
    h = calibration_ptr->data.max_y - calibration_ptr->data.min_y;
else
    h = calibration_ptr->data.min_y - calibration_ptr->data.max_y;
    h =(h /(calibration_ptr->height -2* CALIBRATION_HEIGHT))*
        CALIBRATION_HEIGHT;

if(calibration_ptr->data.max_x > calibration_ptr->data.min_x)
{
    calibration_ptr->data.min_x -= w;
    calibration_ptr->data.max_x += w;
}
else
{
    calibration_ptr->data.min_x += w;
    calibration_ptr->data.max_x -= w;
}
if(calibration_ptr->data.max_y > calibration_ptr->data.min_y)
{
    calibration_ptr->data.min_y -= h;
    calibration_ptr->data.max_y += h;
}
else
{
    calibration_ptr->data.min_y += h;
    calibration_ptr->data.max_y -= h;
}
    rtgui_send(calibration_ptr->app,&ecmd.parent,sizeof(struct
        rtgui_event_command));
}
calibration_ptr->step =0;
return;
}
.....
}

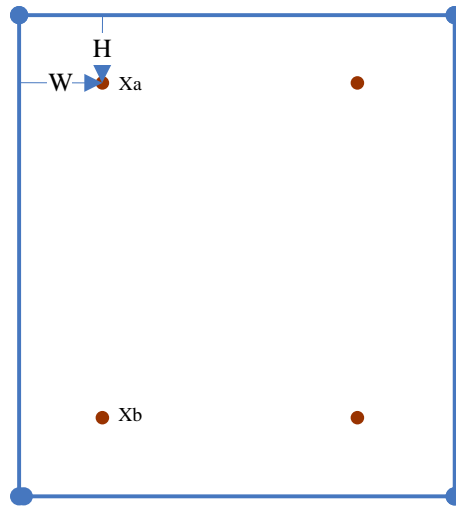
```

拿 MinX 来举例，我们在“像素坐标”（左边的两个点）

PL(CALIBRATION_WIDTH, CALIBRATION_HEIGHT)和

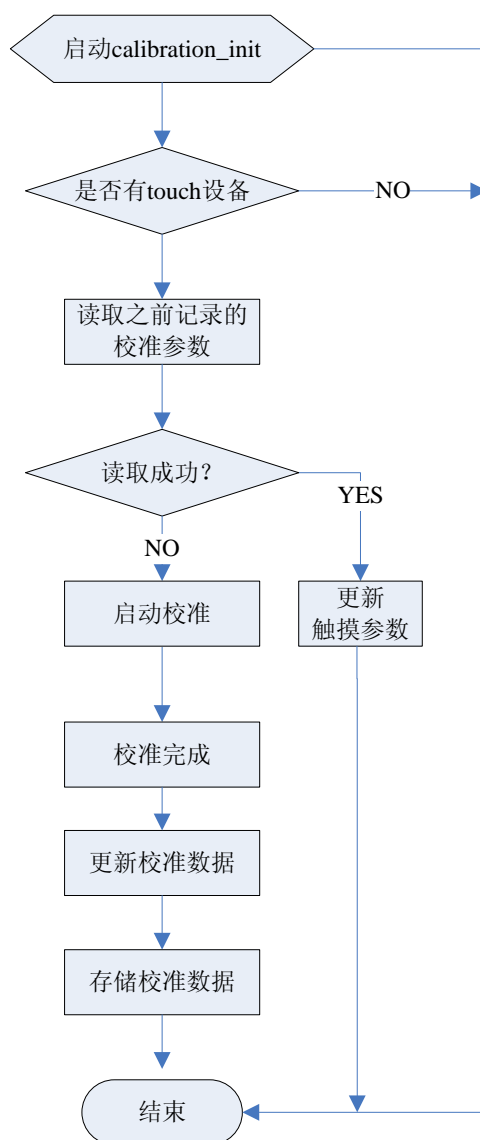
PL(CALIBRATION_WIDTH, height-CALIBRATION_HEIGHT)

上得到的两个物理 x 坐标 (X_a , X_b)，相加求平均后即得到距离屏幕最边缘 w 宽度的“物理坐标” X_c ，此 X_c 减去 w ($=$ 缩放系数*`CALIBRATION_WIDTH`) 就是 MinX 。



其他 3 个点： MaxX 、 MinY 、 MaxY 计算方法相同。到这里整个触摸校准结束，我们可以将触摸校准参数存储起来，以便系统下一次上电时使用，我们的例程中将触摸参数存储在了文件系统根目录下的 `setup.ini` 文件中。

搞明白了触摸校准的原理后，我们再来纵览一下 RTGUI 的 calibration 流程：



说到触摸数据的读取和存储，RTGUI 对用户提供了两个 API:

```

/* 用于读取之前已经存储的校准数据 */
void calibration_set_restore(rt_bool_t (*calibration_restore)(void))
/* 用于将校准数据进行存储 */
void calibration_set_after(void(*calibration_after)(struct calibration_data
*data))

```

我们需要在运行 calibration_init()前调用这两个 API，设定好入口处的函数指针

```

calibration_set_restore(cali_setup);
calibration_set_after(cali_store);

```

我们例子中的是这么做的:

```

rt_bool_t cali_setup(void)
{
    struct setup_items setup;

```

```
if(setup_load(&setup)== RT_EOK)
{
    struct calibration_data data;
    rt_device_t device;
    /* 读取到后更新触摸参数 */
    data.min_x = setup.touch_min_x;
    data.max_x = setup.touch_max_x;
    data.min_y = setup.touch_min_y;
    data.max_y = setup.touch_max_y;

    device = rt_device_find("touch");
    if(device != RT_NULL)
        rt_device_control(device, RT_TOUCH_CALIBRATION_DATA,&data);
    return RT_TRUE;
}
/* 没读取到返回失败, 让其启动校准流程 */
return RT_FALSE;
}

void cali_store(struct calibration_data *data)
{
    struct setup_items setup;
    setup.touch_min_x = data->min_x;
    setup.touch_max_x = data->max_x;
    setup.touch_min_y = data->min_y;
    setup.touch_max_y = data->max_y;
    setup_save(&setup);
}
```

第十八篇网络套接字编程基础

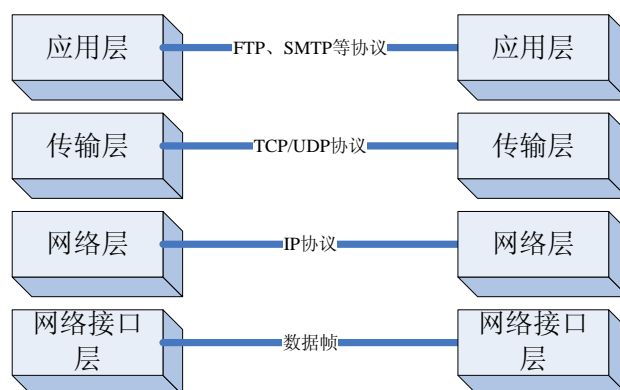
日期：2013-12-15

套接字（Socket）是网络应用程序通信的基石，是支持 TCP/IP 协议的网络通信应用的基本操作单元。可以将套接字看作是不同主机间的进程进行双向通信的端点：网络中两台通信的主机各自在自己机器上建立通信的端点——套接字，然后使用套接字进行数据通信。

RT-Thread 中的 Lwip 提供完整的 BSD Socket 相关 API。（PS：Lwip 还提供 raw API，raw API 主要在无操作系统时使用）

✧ TCP/UDP 协议

在介绍套接字编程前，我们先对 TCP/UDP 协议做一个介绍。TCP/UDP 协议工作在 TCP/IP 协议栈的传输层，如下图所示：



TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的协议，使用该协议时，可以保证客户端和服务端的连接是可靠和安全的。使用 TCP 协议进行通信之前，通信双方必须先建立连接，然后再进行数据传输，通信结束后终止连接。

UDP（User Datagram Protocol 用于数据报协议）是一种非面向连接的协议，它不能保证网络连接的可靠性。当接收数据时它不向发送方提供确认信息，如果出现丢失包或重份包的情况，也不会向发送方发出差错报文。

✧ 创建套接字

使用 socket 通信之前，通信双方都需要各自建立一个 socket。我们通过调用 socket 函数来创建一个 socket 套接字：

```
int socket(int domain,int type,int protocol)
```

从上面可以看到，套接字的特性由三个属性确定，它们是：域（domain），类型（type）和协议（protocol）。

参数 `domain` 指定套接字通信中使用的网络介质，POSIX.1 中指定的各个通信域如下：

domain	描述
AF_INET	Ipv4 因特网域
AF_INET6	Ipv6 因特网域
AF_UNIX	UNIX 域
AF_UNSPEC	未指定

其中 AF_INET 是我们最常用的。

参数 `type` 确定套接字的类型，进一步确定通信特性，POSIX.1 中定义的套接字类型如下：

Type	描述
SOCK_DGRAM	长度固定的、无连接的不可靠的报文传递（UDP）
SOCK_RAW	IP 协议的数据报接口
SOCK_STREAM	有序、可靠、双向的面向连接字节流（TCP）

SOCK_STREAM（数据流套接字）提供面向连接的数据传输，保证在传输过程中数据包不会被丢失、破坏、或重复，按照一定顺序到达目的端。它是最常用的套接字类型，由 TCP 协议所支持。

SOCK_DGRAM（数据报套接字）提供非面向连接的数据传输，不保证数据传输的可靠性和顺序性，它有 UDP 协议锁支持。从资源的角度来看，相对来说它的开销比较小，因为不需要维持网络连接，而且因为无需花费时间来建立连接，它们的速度也很快。

参数 `protocol` 通常是 0，表示按给定的 `domain` 和 `type` 选择默认协议。当对同一 `domain` 和 `type` 支持多个协议时，可以使用 `protocol` 参数选择一个特定协议。在 AF_INET 通信域中套接字类型 SOCK_STREAM 的默认协议是 TCP。在 AF_INET 通信域中套接字类型 SOCK_DGRAM 的默认协议是 UDP。

✧ 绑定套接字

`bind` 函数用来将套接字与计算机上的一个端口号相绑定，进而在该端口监听服务请求，该函数的一般形式如下：

```
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)
```

参数 `sockfd`，为要绑定的 Socket 描述符；

参数 `my_addr` 为一个指向含有本机 IP 地址和端口号等信息的 `sockaddr` 结构的指针；

参数 `addrlen` 通常设为 `sockaddr` 结构的长度。

`sockaddr` 结构体定义如下：

```
struct sockaddr {
```

```

u8_t sa_len;
u8_t sa_family;
char sa_data[14];
};

```

在 IPv4 因特网域 (AF_INET) 中, 我们使用 `sockaddr_in` 结构体来代替 `sockaddr` 结构体:

```

struct sockaddr_in {
    u8_t sin_len;
    u8_t sin_family;
    u16_t sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};

```

其中, `sin_family` 一般固定写 `AF_INET`; `sin_port` 为套接字的端口号; `sin_addr` 为套接字的 IP 地址, `sin_zero` 通常全为 0, 主要功能是为了与 `sockaddr` 结构在长度上保持一致。这样指向 `sockaddr_in` 的指针和指向 `sockaddr` 的指针可以互相转换。

一般情况下, 可以将 `sin_port` 设为 0, 这样系统会随机选择一个未被占用的端口号。同样, `sin_addr` 设为 `INADDR_ANY`, 系统会自动填入本机的 IP 地址。如果自行设置时, 需要进行字节顺序的转换。

在网络中都采用大端字节序, 但是不同的嵌入式系统, 其字节序不一定都是大端格式, 相反小端字节序到是很常见, 比如 STM32。我们在设置 IP 和端口号时, 要根据自己的平台特点进行必要的字节序转换。

下面给出套接字字节转换函数的列表:

`htons()` —— “Host to Network Short” 主机字节顺序转换为网络字节顺序

`htonl()` —— “Host to Network Long” 主机字节顺序转换为网络字节顺序

`ntohs()` —— “Network to Host Short” 网络字节顺序转换为主机字节顺序

`ntohl()` —— “Network to Host Long” 网络字节顺序转换为主机字节顺序

对于一个 “192.168.2.1” 这种字符串形式的 IP 地址, 我们如何将其正确的转换为网络字节序呢? 可以使用 `inet_addr` (“192.168.2.1”), 结果直接就是网络字节序了; 我们也可以使用 `inet_ntoa()` (“ntoa” 代表 “Network to ASCII”) 函数将一个长整形的 IP 地址转换为一个字符串。

注意: 当调用 `bind` 函数时, 不要将端口号设为小于 1024 的值, 因为 1-1024 为系统的保留端口号, 我们可以选择大于 1024 的任何一个未被占用的端口号。

✧ 监听端口

`listen` 函数用来将套接字设为监听模式, 并在套接字指定的端口上开始监听, 以便对到达的服务请求进行处理。`listen` 函数的一般形式如下:

```

int listen(int sockfd, int backlog)

```


参数 `sockfd` 为进行绑定后 `socket` 描述符;

参数 `backlog` 是未经过处理的连接请求队列可以容纳的最大数目。`backlog` 具体一些是什么意思呢? 每一个连入请求都要进入一个连入请求队列, 等待 `listen` 的程序调用 `accept()` (`accept()` 函数下面有介绍) 函数来接受这个连接。当系统还没有调用 `accept()` 函数的时候, 如果有很多连接, 那么本地能够等待的最大数目就是 `backlog` 的数值。你可以将其设成 5 到 10 之间的数值 (推荐)。

✧ 接受连接请求

`accept` 函数用来从完全建立的连接的队列中接受一个连接, 它的一般形式如下:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
```

参数 `sockfd` 为被监听的 `socket` 描述符;

参数 `addr` 指向一个 `sockaddr_in` 结构的指针, 用来存放提出连接请求服务的主机(客户端)IP 地址和端口号等信息;

参数 `addrlen` 设为一个指向 `socklen_t` 的指针, 用来存放 `sockaddr_in` 结构的长度。如果我们不关心客户端的 IP 地址和端口号, 也可以将以上两个参数设为 `NULL`。

服务端接受连接后, `accept` 函数会返回一个新的 `socket` 描述符, 线程可以使用这个新的描述符通客户端传输数据。

✧ 建立连接

`connect` 函数用来与服务器建立一个 TCP 连接, 它的一般形式如下:

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
```

参数 `sockfd` 为 `socket` 描述符;

参数 `serv_addr` 为指向 `sockaddr` 结构的指针, 存放要连接的服务器的 IP 地址和端口号等信息;

参数 `addr_len` 设为 `sockaddr` 结构体的长度;

✧ 面向连接的数据传输 (TCP)

`send` 和 `recv` 函数用来在面向连接的数据流 `socket` 模式下进行数据传输, `send` 函数的一般形式如下:

```
int send(int sockfd, const void *msg, size_t len, int flags)
```

参数 `sockfd` 为准备发送数据的 `socket` 描述符;

参数 `msg` 为指向所要发送的数据区的指针;

参数 `len` 为所要发送的字节数；

参数 `flags` 为控制选项，通常为 0。

`Send` 函数返回实际发送的字节数。如果你给 `send()` 的参数中包含的数据的长度远远大于 `send()` 所能一次发送的数据，则 `send()` 函数只发送它所能发送的最大数据长度，然后它相信你会把剩下的数据再次调用它来进行第二次发送。所以，记住如果 `send()` 函数的返回值小于 `len` 的话，则你需要再次发送剩下的数据。幸运的是，如果包足够小（小于 1K），那么 `send()` 一般都会一次发送光的。

`recv` 函数的一般形式如下：

```
int recv(int socket, void*buf, size_t len, int flags)
```

参数 `sockfd` 为准备接收数据的 `socket` 描述符；

参数 `buf` 为指向你能存储数据的内存缓存区域；

参数 `len` 为缓冲区的长度；

参数 `flags` 为控制选项，通常为 0。

`recv()` 返回它所真正收到的数据的长度。（也就是存到 `buf` 中数据的长度）。

✧ 无连接的数据传输（UDP）

`sendto` 和 `recvfrom` 函数用来在无连接的数据报 `socket` 模式下进行数据传输，`sendto` 函数的一般形式如下：

```
int sendto(int socket, const void*msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen)
```

该函数比 `send` 函数多了两个参数，参数 `to` 为指向 `sockaddr` 结构体的指针，存放目的主机的 IP 和端口号，参数 `tolen` 设为 `sockaddr` 结构体的长度。由于在数据报 `socket` 模式下，本地套接字并没有与目的主机建立连接，所以在发送数据时要指明目的地址。

和 `send()` 一样，`sendto()` 返回它所真正发送的字节数（当然也和 `send()` 一样，它所真正发送的字节数可能小于你所给它的数据的字节数）。

`recvfrom` 函数的一般形式如下：

```
int recvfrom(int s, void*buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen)
```

该函数比 `recv` 函数多了两个参数，参数 `from` 为指向 `sockaddr` 结构体的指针，存放源主机的 IP 和端口号，参数 `fromlen` 设为 `sockaddr` 结构体的长度。

`recvfrom` 函数返回实际接收到的字节数。

✧ 关闭套接字

程序进行网络传输完毕后，我们需要关闭这个套接字描述符所表示的连接。实现这个非常简单，只需要使用 `closesocket()`。

执行 `closesocket()` 之后，套接字将不会在允许进行读操作和写操作。任何有关对套接字描述符进行读和写的操作都会接收到一个错误。

如果我们想对网络套接字的关闭进行进一步的操作的话，则可以使用函数 `shutdown()`。它允许你进行单向的关闭操作，或是全部禁止掉。

`shutdown` 函数的一般形式为：

```
int shutdown(int sockfd, int how)
```

参数 `sockfd` 为要关闭的 `socket` 描述符；

参数 `how` 为控制选项，取值如下：

SHUT_RD：关闭接收信道，进程不能继续从接收缓冲区中接收数据，缓冲区中未读取的数据将被丢弃，但进程仍然可以向发送缓冲区中写入数据。

SHUT_WR：关闭发送信道，进程不能继续向发送缓冲区中写入数据，但缓冲区中未发送的数据会继续发送，进程仍然可以从接收缓冲区中接收数据。

SHUT_RDWR：将发送和接收信道全部关闭。

到此基础知识介绍完毕，下一篇中我们就是用上面的相关函数来进行网络通信的实际操作。

✧ 域名系统

由于 IP 地址难以记忆和识别，人们更习惯于通过通过域名来访问主机（比如我们访问百度时是通过输入 www.baidu.com，而不是输入其 IP <http://115.239.210.26/>），这就需要使

用域名服务器（DNS）来进行域名和 IP 地址之间的转换。这里我们介绍几个常用函数。

1、gethostbyname

此函数可以通过域名来获取主机的 IP 地址等信息，它的一般形式如下：

```
struct hostent* gethostbyname(const char* name)
```

参数 `name` 为主机域名，可以是具体域名，如：“www.baidu.com”，也可以是 IP 地址，如：“192.168.2.56”。

函数返回一个 `hostent` 结构体指针，这个结构体定义如下：

```
struct hostent {
```

```

char*h_name; /* 主机正式域名 */
char**h_aliases; /* 主机的别名数组 */
int h_addrtype; /* 协议类型, 对于TCP/IP 为AF_INET */
int h_length; /* 协议的字节长度, 对于IPv4 为4 个字节 */
char**h_addr_list; /* 地址的列表 */
}

```

这篇中 DNS 测试的例子代码如下:

```

void dns_test(constchar* url)
{
    struct hostent *h;

    /* 取得主机信息 */
    h=(struct hostent *) gethostbyname(url);
    if(h ==NULL)
    {
        /* gethostbyname 失败 */
        rt_kprintf("Socket error\n");
        return;
    }
    /* 打印程序取得的信息 */
    rt_kprintf("Host name : %s\n", h->h_name);
    rt_kprintf("IP Address : %s\n", inet_ntoa (*((struct in_addr
                                                    *)h->h_addr)));

    return;
}

```

上面例子我们需要在 RT-Thread 中开启如下宏:

```
#define RT_LWIP_DNS
```

Ok, 我们来测试一下。在 finsh 中运行 dns_test 命令, 测试几个主流网站:

```

finsh>>dns_test("www.rt-thread.org")
Host name : www.rt-thread.org
IP Address : 210.76.114.59
           0, 0x00000000
finsh>>dns_test("www.taobao.com")
Host name : www.taobao.com
IP Address : 121.14.13.51
           0, 0x00000000
finsh>>dns_test("www.baidu.com")
Host name : www.baidu.com
IP Address : 115.239.210.26
           0, 0x00000000

```

在得到以上网站的具体 IP 后, 你可以用这些 IP 去访问, 看是否和输入域名一样

2、getsockname

此函数可以获取本地主机的信息, 它的一般形式如下:

```

int getsockname(int socket, struct sockaddr *name, socklen_t *namelen)
http://shop73275611.taobao.com122/157

```

参数 `socket` 为已经生成的套接字描述符；

参数 `name` 为 `sockaddr` 结构体指针，用来存储得到的主机信息；

参数 `namelen` 设为 `sockaddr` 结构体的长度；

3、`getpeername`

此函数可以得到与本地主机连接的远程主机的信息，它的一般形式如下：

```
int getpeername(int socket, struct sockaddr *name, socklen_t *namelen)
```

参数 `socket` 为已经生成的套接字描述符；

参数 `name` 为 `sockaddr` 结构体指针，用来存储得到的主机信息；

参数 `namelen` 设为 `sockaddr` 结构体的长度；

第十九篇网络套接字编程实战演练

日期：2013-12-16

套接字编程一般采用客户端-服务器模式，即由客户进程向服务器进程发出请求，服务器进程执行请求的任务并将执行结果返回给客户进程。

这里，我们先将网络套接字编程的流程列出来，然后根据流程来给出具体实例。

✧ 基于 TCP 的 Socket 编程流程

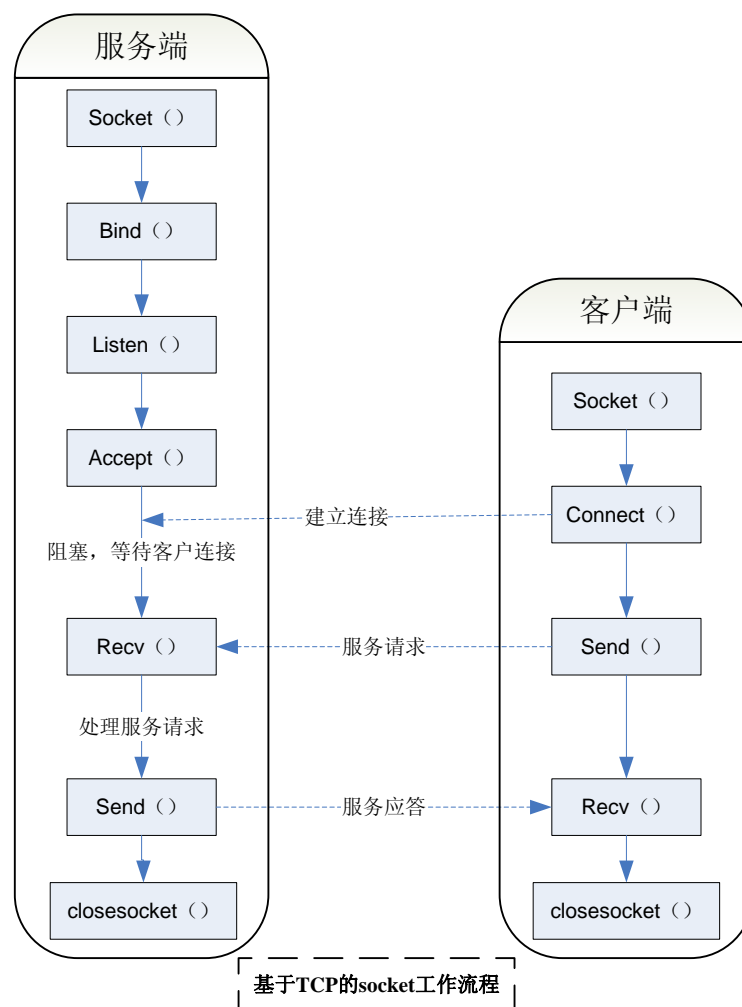
服务器端

- 1、调用 `socket` 函数创建一个套接字；
- 2、将创建的套接字一个 IP 地址和端口号上，通过调用 `bind` 函数来实现；
- 3、调用 `listen` 函数将套接字设为监听模式，以等待连接请求；
- 4、请求到来后，接受连接请求，并返回一个与该连接对应的套接字，通过调用 `accept` 函数来实现；
- 5、与客户端进行通信；
- 6、终止连接，关闭套接字。

客户端

- 1、调用 `socket` 函数创建一个套接字；
- 2、向服务器发出连接请求，通过调用 `connect` 函数来实现；
- 3、与服务器进行通信；
- 4、终止连接，关闭套接字。

以上步骤的流程图如下：



✧ 基于 TCP 的 Socket 编程实例

清楚流程后，我们来做一个应用实例

服务器端

本篇例程中 `tcpsever.c` 是一个 TCP 服务器的示例：

```
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用 BSD Socket 接口必须包含 sockets.h 这个头文件 */

#define SERV_PORT  2345    // 端口 2345, 不能小于 1024, 1-1024 为系统保留端口
#define BACKLOG    5       // 请求队列的长度
#define BUF_SIZE   1024    // 接收缓冲区长度

static const char send_data[] = "This is TCP Server from RT-Thread."; /* 发送用到的数据 */

void tcpsever(void)
```

```

{
    char*recv_data; /* 用于接收的指针，后面会做一次动态分配以请求可用内存 */
    rt_uint32_t sin_size;
    int sockfd, clientsfd; /* 定义监听 socket 描述符和数据传输 socket 描述符 */
    int bytes_received;
    struct sockaddr_in server_addr, client_addr; /* 本机 IP 和端口号信息客户端 IP
和端口号信息 */
    rt_bool_t stop = RT_FALSE; /* 停止标志 */

    recv_data = rt_malloc(BUF_SIZE); /* 分配接收用的数据缓冲 */
    if(recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return;
    }

    /* 一个 socket 在使用前，需要预先创建出来，指定 SOCK_STREAM 为 TCP 的 socket */
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        /* 创建失败的错误处理 */
        rt_kprintf("Socket error\n");

        /* 释放已分配的接收缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 初始化服务端地址 */
    server_addr.sin_family = AF_INET; /* Ipv4 因特网域 */
    server_addr.sin_port = htons(SERV_PORT); /* 服务端工作的端口这里进行网络字节
序转换 */
    server_addr.sin_addr.s_addr = INADDR_ANY; /* 本机 IP 地址 */
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero)); /* 清
0 */

    /* 绑定 socket 到服务端地址 */
    if(bind(sockfd, (struct sockaddr *)&server_addr,
            sizeof(struct sockaddr)) == -1)
    {
        /* 绑定失败 */
        rt_kprintf("Unable to bind\n");

        /* 释放已分配的接收缓冲 */
        rt_free(recv_data);
    }
}

```



```
    return;
}

/* 在 sockfd 上进行监听 */
if(listen(sockfd, BACKLOG)==-1)
{
    rt_kprintf("Listen error\n");

    /* 释放已分配的接收缓冲 */
    rt_free(recv_data);
    return;
}

rt_sprintf(recv_data,"%4d", SERV_PORT);
rt_kprintf("\nTCPServer Waiting for client on port %s...\n",recv_data);

while(stop != RT_TRUE)
{
    sin_size =sizeof(struct sockaddr_in);

    /* 接受一个客户端连接 socket 的请求, 这个函数调用是阻塞式的 */
    clientsfd = accept(sockfd,(struct sockaddr *)&client_addr,&sin_size);
    /* 返回的是连接成功的 socket 描述符, 后续将使用这个描述符来进行通信 */

    /* 打印客户端的 IP 和端口号 */
    rt_kprintf("I got a connection from (IP: %s ,PORT: %d)\n",
        inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port));

    /* 客户端连接的处理 */
    while(1)
    {
        /* 发送数据到 connected socket */
        send(clientsfd, send_data, strlen(send_data),0);

        /* 从 connected socket 中接收数据, 接收 buffer 是 1024 大小, 但并不一定能够收到 1024 大小的数据 */
        bytes_received =recv(clientsfd, recv_data, BUF_SIZE,0);
        if(bytes_received <=0)
        {
            /* 接收失败, 关闭这个 connected socket */
            closesocket(clientsfd);
            break;
        }
        /* 有接收到数据, 把末端清零即加入字符串结束符*/
    }
}
```

```

recv_data[bytes_received]='\0';
if(strcmp(recv_data,"q")==0|| strcmp(recv_data,"Q")==0)
{
    /* 如果首字母是 q 或 Q, 关闭这个连接 */
    closesocket(clientsfd);
    break;
}
elseif(strcmp(recv_data,"exit")==0)
{
    /* 如果接收的是 exit, 则关闭整个服务端 */
    closesocket(clientsfd);
    stop = RT_TRUE;
    break;
}
else
{
    /* 在控制终端显示收到的数据 */
    rt_kprintf("RECIEVED DATA = %s \n", recv_data);
}
}
}

/* 关闭 socket 退出服务 */
closesocket(sockfd);
/* 释放接收缓冲 */
rt_free(recv_data);
return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出 tcpserv 函数到 finsh shell 中 */
FINSH_FUNCTION_EXPORT(tcpserv, startup tcp server);
#endif

```

RT-Thread 的配置文件中需打开如下宏：

```

#define RT_USING_LWIP
#define RT_LWIP_TCP
#define RT_LWIP_DHCP

```

开发板接上网线（即接上接路由器，路由器许开启 DHCP 功能）运行上述程序。我们先运行 `list_if()` 命令，确定开发板获得的 IP 地址，如下：

```
list_if()
network interface: e0 (default)
MTU: 1500
MAC: 00 60 6e 11 22 33
FLAGS: UP LINK UP DHCP ETHARP
ip address: 192.168.2.6
gw address: 192.168.2.1
net mask : 255.255.255.0

dns server #0: 202.96.134.33
dns server #1: 202.96.128.86
             0, 0x00000000
finsh>>
```

然后运行 tcp_sever 命令:

```
finsh>>tcpserv()
TCPserver waiting for client on port 2345...
```

这样，开发板作为 TCP 服务器端已经在运行了，我们接着在 PC 端模拟一个 TCP 客户端来和其通信。打开网络调试助手，按下图设置，然后开启 TCP 客户端（我们可以在 finsh 中运行 list_if() 命令来获得开发板的 IP）：



最后，我们可以进行通信了，可以通过网络调试助手来发送数据到开发板，开发板收到数据后会打印出收到的数据然后回发固定的字符串："This is TCP Server from RT-Thread."，我们也可以根据客户端发来的数据进行分析，然后有选择性的回发，下面即是开发后收到数据后打印的东西：

```
I got a connection from (IP: 192.168.2.4 ,PORT: 4684)
RECIEVED DATA = hello, i am jiezhi320
```

客户端

本篇例程中 tcpclient.c 是一个 TCP 客户端的示例：

```
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含 netdb.h 头文件 */
#include <lwip/sockets.h> /* 使用 BSD socket，需要包含 sockets.h 头文件 */
```

```
#define BUF_SIZE 1024

Static const char send_data[]="This is TCP Client from RT-Thread."; /* 发送用到的数据 */

void tcpclient(constchar*url,int port)
{
    char*recv_data; /*接收缓冲区指针*/
    struct hostent *host; /*用于通过DNS 解析服务器端信息*/
    int sockfd; /* socket 描述符*/
    int bytes_received;
    struct sockaddr_in server_addr; /* 存储服务端IP 和端口号*/

    /* 通过函数入口参数url 获得host 地址（如果是域名，会做域名解析） */
    host =gethostbyname(url);

    /* 分配用于存放接收数据的缓冲 */
    recv_data = rt_malloc(BUF_SIZE);
    if(recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return;
    }

    /* 创建一个socket，类型是SOCKET_STREAM，TCP 类型 */
    if((sockfd =socket(AF_INET, SOCK_STREAM,0))==-1)
    {
        /* 创建socket 失败 */
        rt_kprintf("Socket error\n");

        /* 释放接收缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 初始化预连接的服务端地址 htons*/
    server_addr.sin_family = AF_INET; /* IPv4 因特网域*/
    server_addr.sin_port = htons(port); /* 服务器端的端口这里进行字节序转换*/
    server_addr.sin_addr =*((struct in_addr *)host->h_addr); /* 主机IP 信息*/
    /*
        注意上面不用inet_addr（“192.168.2.1”）这种形式来做主机IP 的网络字节序转换
        是由于笔者用那种方式没调通，暂时来不知道为何
    */
    rt_memset(&(server_addr.sin_zero),0,sizeof(server_addr.sin_zero));
```

```
/* 连接到服务端 */
if(connect(sockfd,(struct sockaddr *)&server_addr,sizeof(struct
sockaddr))== -1)
{
    /* 连接失败 */
    rt_kprintf("Connect error\n");

    /* 释放接收缓冲 */
    rt_free(recv_data);
    return;
}

while(1)
{
    /* 从 sock 连接中接收最大 BUF_SIZE - 1 字节数据 */
    bytes_received =recv(sockfd, recv_data, BUF_SIZE -1,0);
    if(bytes_received <=0)
    {
        /* 接收失败, 关闭这个连接 */
        closesocket(sockfd);

        /* 释放接收缓冲 */
        rt_free(recv_data);
        break;
    }

    /* 有接收到数据, 把末端清零即加入字符串结束符*/
    recv_data[bytes_received]='\0';

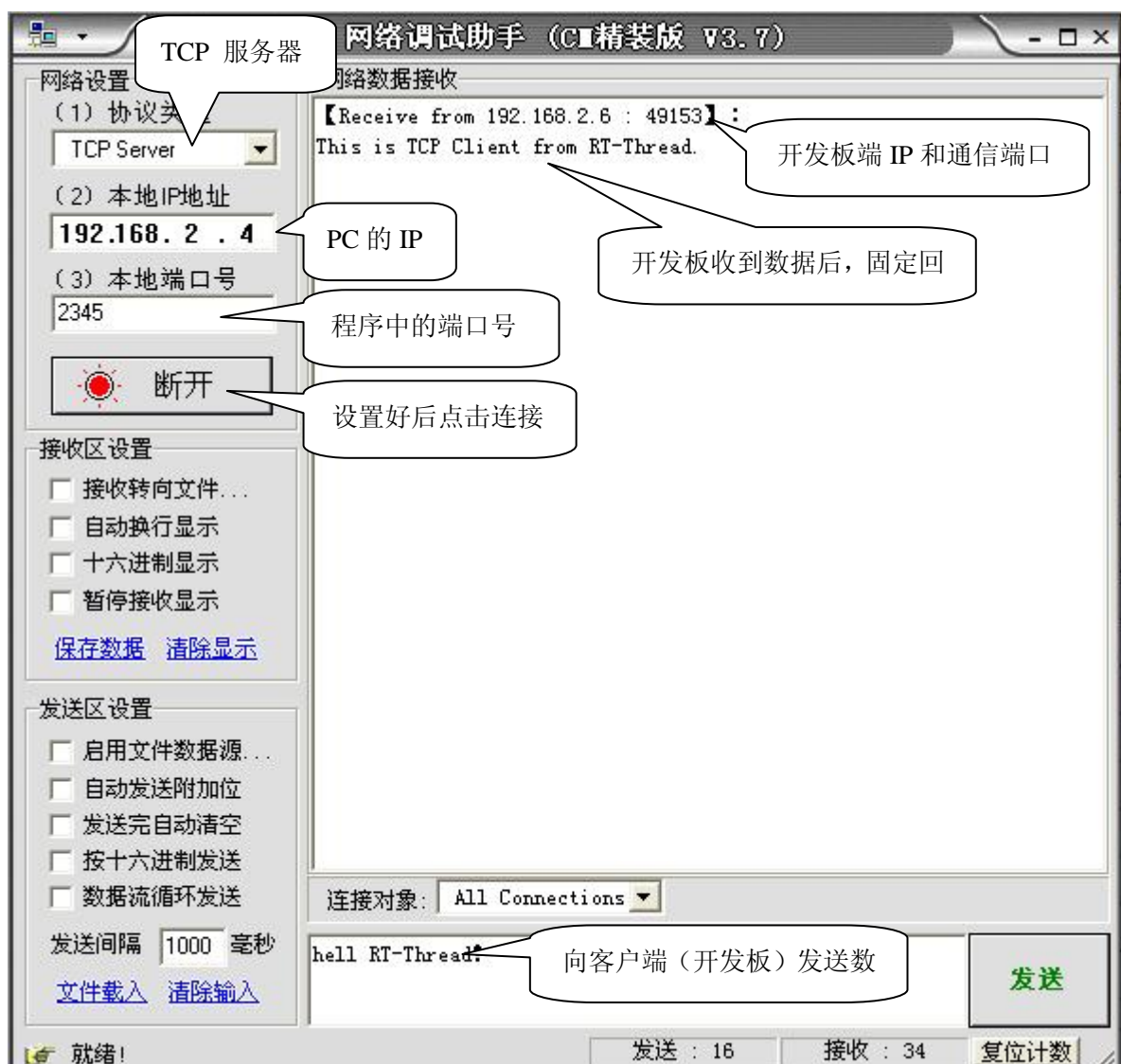
    if(strcmp(recv_data ,"q")==0|| strcmp(recv_data ,"Q")==0)
    {
        /* 如果首字母是 q 或 Q, 关闭这个连接 */
        closesocket(sockfd);

        /* 释放接收缓冲 */
        rt_free(recv_data);
        break;
    }
    else
    {
        /* 在控制终端显示收到的数据 */
        rt_kprintf("\nRecieved data = %s ", recv_data);
    }
}
```

```
    /* 发送数据到 sock 连接 */
    send(sockfd, send_data, strlen(send_data), 0);
}
return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出 tcpclient 函数到 finsh shell 中
   //tcpclient("192.168.2.4", 2345);
   //tcpclient("baidu.com", 80);
*/
FINSH_FUNCTION_EXPORT(tcpclient, startup tcp client);
#endif
```

为了验证以上程序，我们先在 PC 端用网络调试助手搭建一个 TCP 服务器，服务器的 IP 即是 PC 的 IP，端口号我们按照程序中设定的端口号来设置：



然后开发板联网后上电，运行连接命令：

```
finsh>>tcpclient("192.168.2.4",2345)
```

连接建立起来后，可以在 PC 端向开发板发送数据，开发板收到数据后会将收到的数据打印出来并回传固定数据，下面就是开发板收到数据后打印的数据：

```
Recieved data = hell RT-Thread!
```

✧ 基于 UDP 的 Socket 编程流程

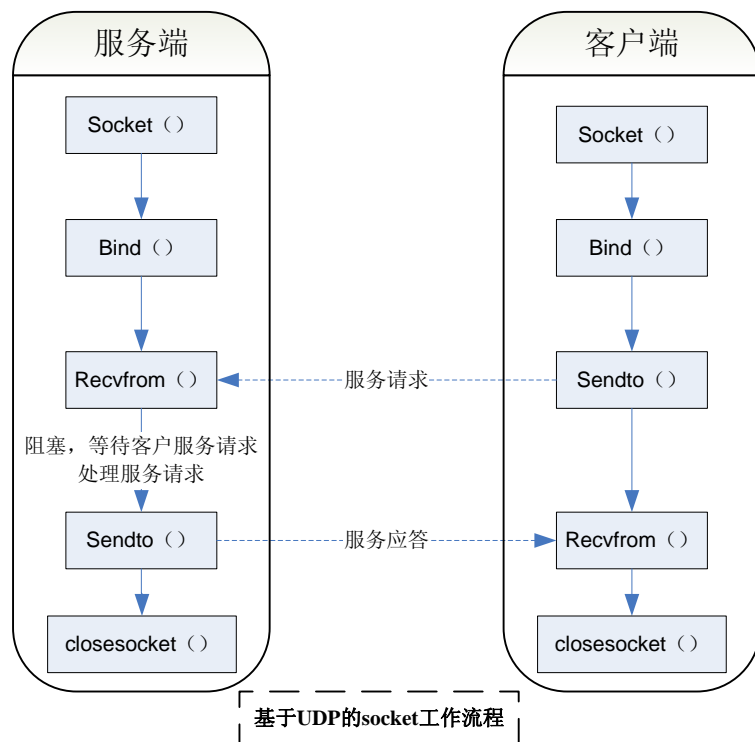
服务器端

- 1、调用 socket 函数创建一个套接字；
- 2、将创建的套接字一个 IP 地址和端口号上，通过调用 bind 函数来实现；
- 3、等待接收数据报，处理完成后将结果返回到客户端；
- 4、终止连接，关闭套接字。

客户端

- 1、调用 `socket` 函数创建一个套接字；
- 2、向服务器发送数据报
- 3、终止连接，关闭套接字。

以上步骤的流程图如下：



✧ 基于 UDP 的 Socket 编程实例

服务器端

本篇例程中 `udpserver.c` 是一个 UDP 服务器的示例：

```
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用 BSD socket，需要包含 sockets.h 头文件 */

#define SERV_PORT 2369
#define BUF_SIZE 1024

void udpserv(void)
{
    int sockfd; /* socket 描述符*/
    struct sockaddr_in server_addr; /* 主机 IP 地址和端口号 */
    struct sockaddr_in client_addr; /* 客户端 IP 地址和端口号 */
    int bytes_read;
```

```
char*recv_data; /* 接收缓冲区*/
rt_uint32_t addr_len;

/* 分配接收缓冲区 */
recv_data = rt_malloc(BUF_SIZE);
if(recv_data == RT_NULL)
{
    /* 分配内存失败, 返回 */
    rt_kprintf("No memory\n");
    return;
}

/* 创建一个 socket, 类型是 SOCK_DGRAM, UDP 类型 */
if((sockfd =socket(AF_INET, SOCK_DGRAM,0))==-1)
{
    rt_kprintf("Socket error\n");

    /* 释放接收用的数据缓冲 */
    rt_free(recv_data);
    return;
}

/* 初始化服务端地址 */
server_addr.sin_family = AF_INET; /*IPv4 因特网域*/
server_addr.sin_port = htons(SERV_PORT); /*端口号, 这里进行网络字节序的转换*/
server_addr.sin_addr.s_addr = INADDR_ANY; /* 本机 IP 地址*/
rt_memset(&(server_addr.sin_zero),0,sizeof(server_addr.sin_zero));

/* 绑定 socket 到服务端地址 */
if(bind(sockfd,(struct sockaddr *)&server_addr,
sizeof(struct sockaddr))==-1)
{
    /* 绑定地址失败 */
    rt_kprintf("Bind error\n");

    /* 释放接收用的数据缓冲 */
    rt_free(recv_data);
    return;
}

rt_sprintf(recv_data,"%4d", SERV_PORT);
rt_kprintf("UDPServer Waiting for client on port %s...\n", recv_data);
```

```

addr_len = sizeof(struct sockaddr);
/* 循环接收UDP 数据 */
while(1)
{
    /* 从 sock 中收取最大 BUF_SIZE - 1 字节数据 */
    bytes_read = recvfrom(sockfd, recv_data, BUF_SIZE - 1, 0,
                          (struct sockaddr *)&client_addr, &addr_len);
    /* UDP 不同于 TCP，它基本不会出现收取的数据失败的情况，除非设置了超时等待 */

    recv_data[bytes_read] = '\0'; /* 把末端清零即字符串结束符 */

    /* 输出接收的数据 */
    rt_kprintf("\n(%s , %d) said : ", inet_ntoa(client_addr.sin_addr),
               ntohs(client_addr.sin_port));
    rt_kprintf("%s", recv_data);

    /* 如果接收数据是 exit，退出 */
    if(strcmp(recv_data, "exit") == 0)
    {
        closesocket(sockfd);

        /* 释放接收用的数据缓冲 */
        rt_free(recv_data);
        break;
    }
}
return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出 udp_serv 函数到 finsh shell 中 */
FINSH_FUNCTION_EXPORT(udp_serv, startup udp server);
#endif

```

RT-Thread 的配置文件中需打开如下宏：

```

#define RT_USING_LWIP
#define RT_LWIP_UDP
#define RT_LWIP_DHCP

```

开发板接上网线（即接上接路由器，路由器许开启 DHCP 功能）运行上述程序。我们先运行 `list_if()` 命令，确定开发板获得的 IP 地址，如下：

```

T1st_if()
network interface: e0 (default)
MTU: 1500
MAC: 00 60 6e 11 22 33
FLAGS: UP LINK UP DHCP ETHARP
ip address: 192.168.2.6
gw address: 192.168.2.1
net mask : 255.255.255.0

dns server #0: 202.96.134.33
dns server #1: 202.96.128.86
             0, 0x00000000
finsh>>

```

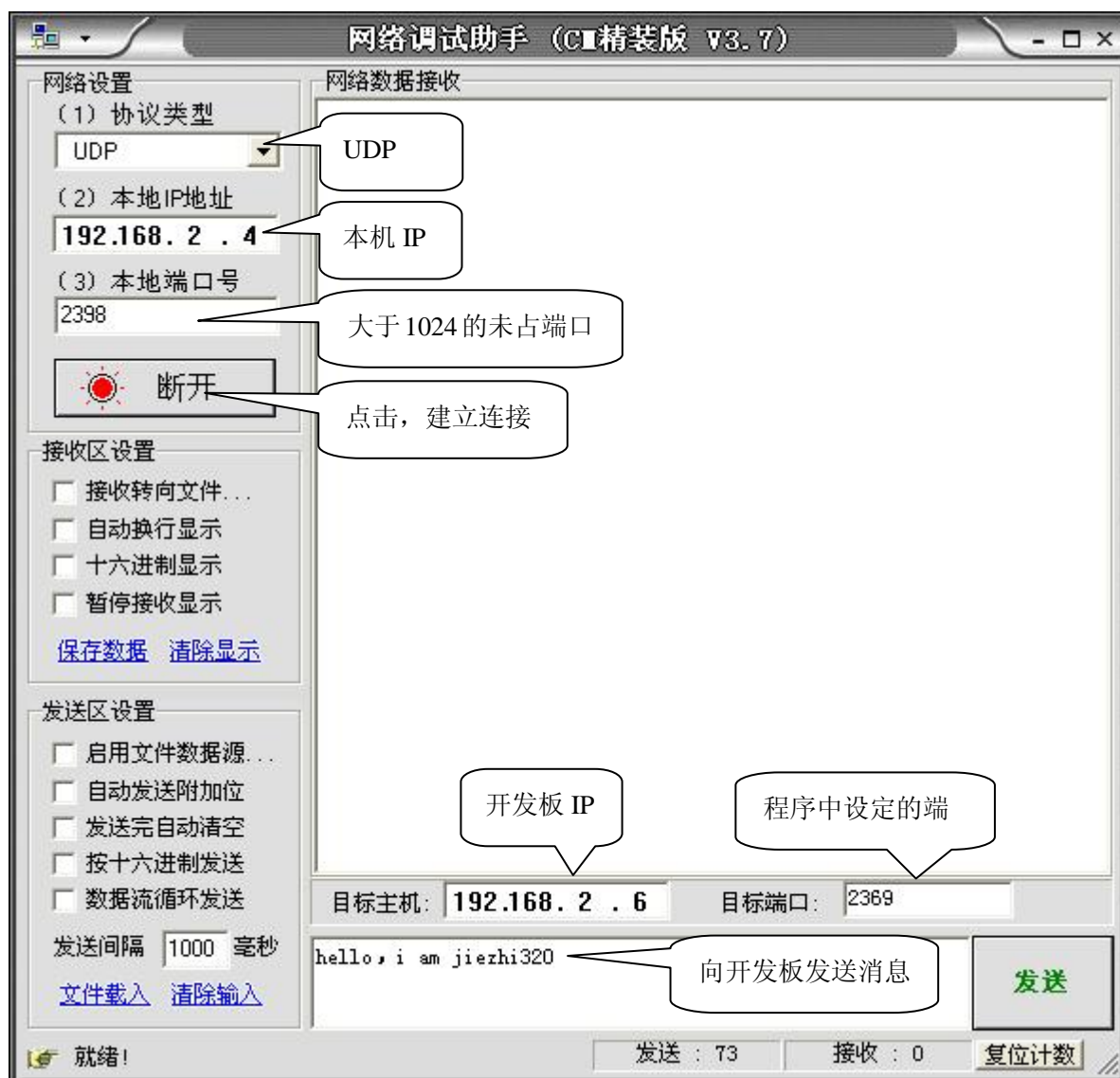
然后运行 `udp_sever` 命令：

```

finsh>>udpserv()
UDPServer waiting for client on port 2369...

```

这样，开发板作为 UDP 服务器端已经在运行了，我们接着在 PC 端模拟一个 UDP 连接来和其通信。打开网络调试助手，按下图设置，然后建立起 UDP 连接：



当我通过网络调试助手向开发板发送消息时，开发板将会把收到的消息打印出来，并且打印出消息的来源：

```
(192.168.2.4 , 2398) said : http://www.cmsoft.cn
(192.168.2.4 , 2398) said : ty
(192.168.2.4 , 2398) said : ty
(192.168.2.4 , 2398) said : hello, i am jiezhi320
```

客户端

本篇例程中 udpcient.c 是一个 UDP 客户端的示例：

```
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含 netdb.h 头文件 */
#include <lwip/sockets.h> /* 使用 BSD socket，需要包含 sockets.h 头文件 */

constchar send_data[]="This is UDP Client from RT-Thread.\n"; /* 发送用到的数据 */
void udpcient(constchar* url,int port,int count)
{
    int sockfd;
    struct hostent *host;
    struct sockaddr_in server_addr;

    /* 通过函数入口参数url 获得host 地址（如果是域名，会做域名解析） */
    host=(struct hostent *)gethostbyname(url);

    /* 创建一个socket，类型是SOCK_DGRAM，UDP 类型 */
    if((sockfd =socket(AF_INET, SOCK_DGRAM,0))==-1)
    {
        rt_kprintf("Socket error\n");
        return;
    }

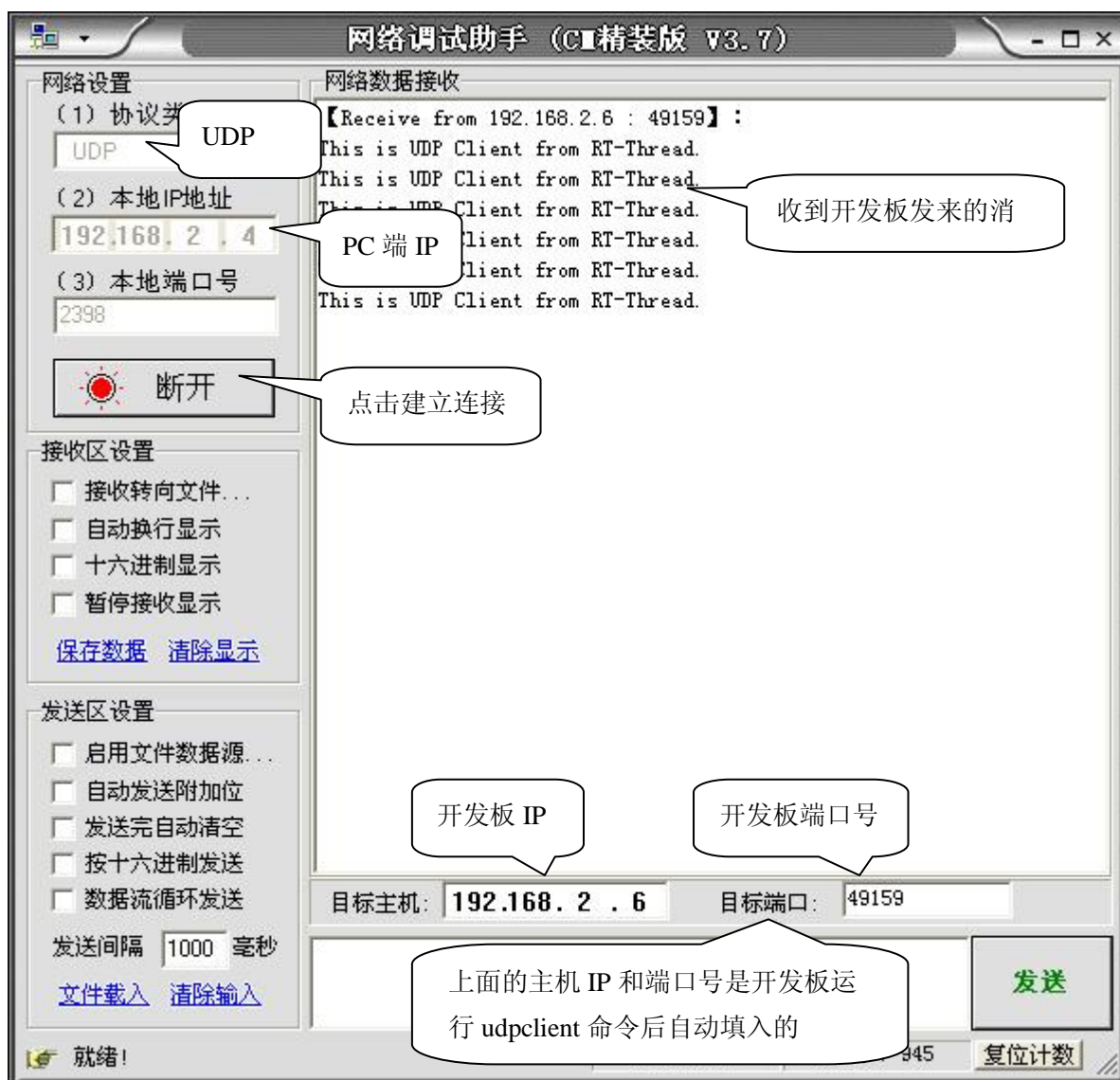
    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr =*((struct in_addr *)host->h_addr);
    rt_memset(&(server_addr.sin_zero),0,sizeof(server_addr.sin_zero));
    /* 总计发送count 次数据 */
    while(count)
    {
        /* 发送数据到服务远端 */
        sendto(sockfd, send_data, strlen(send_data),0,
            (struct sockaddr *)&server_addr,sizeof(struct sockaddr));
        /* 线程休眠一段时间 */
    }
}
```

```

    rt_thread_delay(50);
    /* 计数值减一 */
    count --;
}
/* 关闭这个 socket */
closesocket(sockfd);
}
#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出 udpclient 函数到 finsh shell 中 */
FINSH_FUNCTION_EXPORT(udpclient, startup udp client);
#endif

```

我们先在 PC 端用网络调试助手建立起一个 UDP 连接作为服务器：



上面的连接建立起来后，我们在开发板上运行命令：

finsh>>udpclient("192.168.2.4",2398,6), (其中"192.168.2.4"为 PC 端 IP, 2398 为我们在网络
<http://shop73275611.taobao.com>140 / 157

调试助手中设置的端口号，6 表示开发板发送 6 次数据后断开连接），然后网络调试助手就会收到开发板发过去的 6 条信息。

✧ 广播与多播

以上的讲解，都是单播连接（点对点通信），对于 UDP 传输来说，还有一个广播和多播的概念（因为 TCP 是面向连接的传输协议，对于多播/广播这样没有指明目标的行为，TCP 是无法完成任务的）。广播和多播对于需将报文同时传往多个接收者的应用来说十分重要。

广播（broadcast）就是是一个主机要向网上的所有其他主机发送数据帧。

多播(multicast)处于单播和广播之间，帧仅传送给属于多播组的多个主机。

网络中的 IP 地址，有单播地址、多播地址、广播地址之分。本篇中例子中出现的那些 IP 地址都是单播地址，如 192.168.2.4。下面我们来说说广播地址和多播地址（广播和多播其实就是向某个特定的 IP 地址发送数据而已）。

广播地址是目标 IP 地址的主机部分全为 1 的 IP 地址，例如：

C 类网络 192.168.1.0 的默认子网掩码为 255.255.255.0，其广播地址为 192.168.1.255；

B 类网络 172.16.0.0 的默认子网掩码为 255.255.0.0，其广播地址为 172.16.255.255；

A 类网络 10.0.0.0 的默认子网掩码为 255.0.0.0，其广播地址为 10.255.255.255。

多播地址存在于 D 类 IP 地址中，多播组地址包括为 1110 的最高 4 bit 和多播组号，范围从 224.0.0.0 到 239.255.255.255。例如，224.0.0.1 代表“该子网内的所有系统组”，224.0.0.2 代表“该子网内的所有路由器组”。多播地址 224.0.1.1 用作网络时间协议 NTP。

广播和多播实际上就是将源主机发送数据的任务交给了路由器或网络中对的主机什么的来进行，网络内的主机和路由器会自动帮我们完成转发任务。

针对广播，我们再说一点。UDP 客户端除了向上面说的广播地址发送消息以完成广播任务外，还可以通过 `setsockopt` 函数的设置来完成广播功能。

函数 `setsockopt` 的一般形式如下：

```
int setsockopt(int sockfd,int level,int optname,const void*optval, socklen_t optlen)
```

参数 `sockfd` 必须是一个已经创建完成的套接字描述符；

参数 `level` 一般设成 `SOL_SOCKET`，表示网络层；

参数 `optname` 为设置的选型，如果取值 `SO_BROADCAST`，则表示广播方式；

参数 `optval` 为设置的值；

参数 `optlen` 为 `optval` 的长度。

在上面 UDP 客户端例子中，如果要使用广播方式，在套接字创建完成后，添加下面代码：

```
opt = 1;
if(setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, &opt) == -1)
{
    rt_kprintf("setsockopt error\n");
    return;
}
```

✧ 开发板与 PC 互传图片

在以上的实例代码中，我们都是通过发送字符串来演示网络通信的。当然通信所承载的内容，除了字符串，也可以是图片或视频数据等，这些字符串、图片、视频等，我们都一概称之为“数据”，套接字编程所搭建起来的“通信线路”可以传输任何格式的数据。

我们的示例代码中有一个通过 TCP 通信来传输图片数据的例子，其中 PC 端用 C# 做了一个 TCP 服务器，开发板做客户端，这样 PC 与开发板就可以互传图片（网友钟童鞋提供 PC 端代码，再次感谢！）。

首先打开例程目录，运行网络传输 PC 端\网络传输 demo\bin\Debug\网络传输 demo.exe 启动 tcp 服务器端，并开始监听：



开发板联网上电，finsh 下运行 `tcpclient("192.168.2.4", 5000)`，以上 IP 地址和端口号根据 PC 端服务器的情况来设定。稍等片刻，PC 端软件会弹出图片传输界面，我们可以从 PC 端发 bmp 图片到开发板并通过 lcd 显示，也可以发送固化在开发板内的图片到 PC 然后显示：



注：bmp图片请选择240*320的24bit格式图片；PC端软件需要.net环境支持，可用vs2008以上版本来自行编辑修改；代码中bmp图片的数组是原图片用winhex打开得到的。

第二十篇 TCP 并发服务器模型

日期: 2014-01-7

第 19 篇中的 TCP 服务器的例子是这样一个结构:

```
socket(...);
bind(...);
listen(...);
while(1)
{
    accept(...);
    Process(...);
    Closesocket(...);
}
```

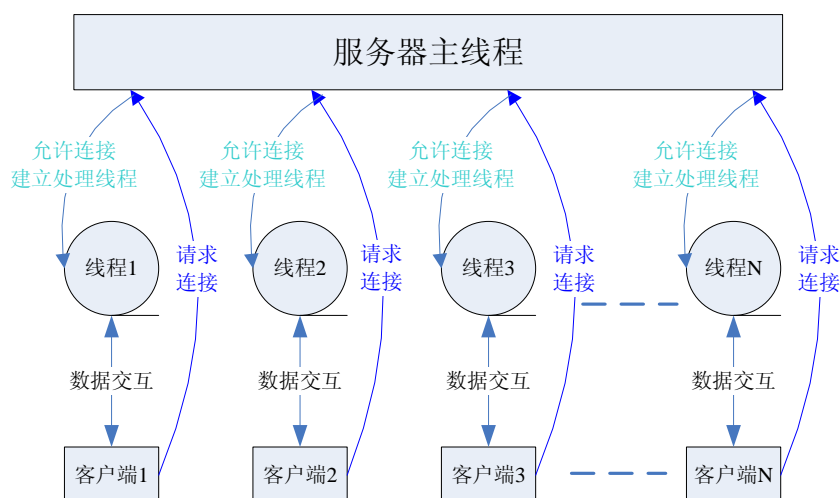
这样的服务器结构模型, 我们称之为**循环服务器模型**。因为 TCP 是面向连接的, 因此 TCP 循环服务器一次只能处理一个客户端的请求 (当一个客户端与服务器连接之后, 其他客户不能访问服务器); UDP 是面向无连接的, 因此 UDP 服务器可以同时服务多个客户端。

实际应用中, 一个 TCP 服务器往往要响应多个客户端的请求, 为解决这个问题, 我们引入 **TCP 并发服务器模型**。在 linux 下 TCP 并发服务器有多种具体实现方法, 在 RT-Thread 下我们可以使用多线程和 IO 多路复用这两种方法来实现。

✧ 多线程并发服务器

服务器每允许一个客户端的连接请求后, 立即创建一个新的线程去做处理, 服务器“腾出手来”去处理别的客户端的连接请求。

这种方法图示如下:



接下来我们给出一个实际编程的例子，这个例子中的 TCP 服务器可以同时响应多个客户端，当其收到某个客户端发来的消息后，将其打印出来并向客户端发回固定的数据。

需要说明的是，在这里我们创建新的线程时，使用 RT-Thread 下的 POSIX thread 接口 `pthread_create` 来创建线程，为此需要在配置文件中开启如下宏：

```
#define RT_USING_PTHREADS
```

```
/*
此 demo 用于演示 TCP 多线程服务器
*/
#define LWIP_TIMEVAL_PRIVATE 0

#include <rtthread.h>
#include <lwip/sockets.h> /* 使用 BSD Socket 接口必须包含 sockets.h 这个头文件 */
#include <pthread.h> /* 使用多线程必须包含这个头文件 */

#define SERV_PORT    2345    // 端口 2345, 不能小于 1024, 1-1024 为系统保留端口
#define BACKLOG      5      // 最多允许的客户端连接数
#define BUF_SIZE     1024    // 接收缓冲区长度

static const char send_data[] = "This is TCP Server from RT-Thread."; /* 发送用到的数据 */

void process_cli(int connectfd, struct sockaddr_in client);
/* 新线程中的处理函数 */
void* start_routine(void* arg);

typedef struct _ARG {
    int connfd;
    struct sockaddr_in client;
} ARG;

void tcpserv(void* parameter)
{
    char tmp[10];
    rt_uint32_t sin_size;
    int listenfd, connectfd; /* 定义监听 socket 描述符和数据传输 socket 描述符 */
    struct sockaddr_in server_addr, client_addr; /* 本机 IP 和端口号信息客户端 IP 和端口号信息 */
    pthread_t p_tid;
```

```

ARG *arg;
int opt;

/* 一个socket 在使用前, 需要预先创建出来, 指定SOCK_STREAM 为TCP 的socket */
if((listenfd = socket(AF_INET, SOCK_STREAM,0))==-1)
{
    /* 创建失败的错误处理 */
    rt_kprintf("Socket error\n");
    return;
}

opt=1;

setsockopt( listenfd, IPPROTO_TCP, TCP_NODELAY,&opt,sizeof(opt));//最小化
报文传输的延时
setsockopt( listenfd, SOL_SOCKET, SO_REUSEADDR,&opt,sizeof(opt));//允许地
址重用
setsockopt( listenfd, SOL_SOCKET, SO_REUSEPORT,&opt,sizeof(opt));//允许端
口重用

/* 初始化服务端地址 */
server_addr.sin_family = AF_INET; /* Ipv4 因特网域 */
server_addr.sin_port = htons(SERV_PORT); /* 服务端工作的端口这里进行网络字节
序转换*/
server_addr.sin_addr.s_addr = INADDR_ANY; /* 本机IP 地址 */
rt_memset(&(server_addr.sin_zero),0,sizeof(server_addr.sin_zero)); /* 清
0 */

/* 绑定socket 到服务端地址 */
if(bind(listenfd,(struct sockaddr *)&server_addr,sizeof(struct
sockaddr))==-1)
{
    /* 绑定失败 */
    rt_kprintf("Unable to bind\n");
    return;
}

/* 在sockfd 上进行监听 */
if(listen(listenfd, BACKLOG)==-1)
{
    rt_kprintf("Listen error\n");
    return;
}

```

```

rt_sprintf(tmp,"%4d", SERV_PORT);
rt_kprintf("\nTCPServer Waiting for client on port %s...\n",tmp);

sin_size =sizeof(struct sockaddr_in);

while(1)
{
    /* 接受一个客户端连接 socket 的请求, 这个函数调用是阻塞式的 */
    if((connectfd = accept(listenfd,(struct sockaddr
*)&client_addr,&sin_size))!=-1)
    {
        /* 返回的是连接成功的 socket 描述符, 后续将使用这个描述符来进行通信 */
        rt_kprintf("accept error \n");
        break;
    }

    /* 客户端连接上以后新建立一个线程处理*/
    arg =(ARG*)rt_malloc(sizeof(ARG));
    if(arg == RT_NULL)
    {
        rt_kprintf("No memory\n");
        break;
    }
    arg->connfd = connectfd;
    memcpy(&arg->client,&client_addr,sizeof(client_addr));

    if(pthread_create(&p_tid,NULL, start_routine,(void*)arg))
    {
        /* handle exception */
        rt_kprintf("Pthread_create error");
        rt_free(arg);
        break;
    }
}
/* 关闭 socket 退出服务 */
closesocket(listenfd);
return;
}

void process_cli(int connectfd,struct sockaddr_in client)
{
    int bytes_received;
    char*recv_data; /* 用于接收的指针, 后面会做一次动态分配以请求可用内存 */

    recv_data = rt_malloc(BUF_SIZE); /* 分配接收用的数据缓冲 */

```

```
if(recv_data == RT_NULL)
{
    rt_kprintf("No memory for recv_data\n");
    return;
}

/* 打印客户端的IP 和端口号 */
rt_kprintf("I got a connection from (IP: %s ,PORT: %d)\n",
    inet_ntoa(client.sin_addr), ntohs(client.sin_port));

/* 发送数据到connected socket */
send(connectfd, send_data, strlen(send_data),0);

bytes_received = recv(connectfd, recv_data, BUF_SIZE,0);
if(bytes_received<=0)
{
    closesocket(connectfd);
    rt_kprintf("Client disconnected.\n");
    rt_free(recv_data);
    return;
}
while(bytes_received)
{
    /*每次收到客户端的消息后则打印出收到的消息并向客户端发回固定的消息*/
    /* 有接收到数据, 把末端清零即加入字符串结束符*/
    recv_data[bytes_received]=0;
    rt_kprintf("RECIEVED DATA = %s from IP: %s ,PORT: %d)\n",recv_data,
        inet_ntoa(client.sin_addr), ntohs(client.sin_port));

    /* 发送数据到connected socket */
    send(connectfd, send_data, strlen(send_data),0);

    bytes_received = recv(connectfd, recv_data, BUF_SIZE,0);
    if(bytes_received<=0)
    {
        rt_kprintf("Client disconnected.\n");
        break;
    }
}

closesocket(connectfd);/* close connectfd */
rt_free(recv_data);
}

void* start_routine(void* arg)
```

```

{
    ARG *info;

    info =(ARG *)arg;
/* 处理客户端请求 */
    process_cli(info->connfd, info->client);

    rt_free(info);
    //pthread_exit(NULL);

    return 0;
}

int tcp_demo_init(void)
{
    rt_thread_t demo_thread;

    demo_thread = rt_thread_create("tcp_demo",
                                    tcpserv, RT_NULL, 2048, 20, 20);

    if(demo_thread != RT_NULL)
        rt_thread_startup(demo_thread);

    return 0;
}

#ifdef RT_USING_FINSH
#include <finsh.h>

FINSH_FUNCTION_EXPORT(tcp_demo_init, startup tcp server);

#endif

```

第 19 篇中我们讲解了 TCP 服务器例子的验证方法，对于这个并发服务器例子的验证，与之类似，只不过这里我们要在 PC 端同时运行 N 个 TCP 客户端去连接服务器。

开发板端的运行输出信息如下：

```

finsh>>tcp_demo_init()
0, 0x00000000
finsh>>
TCPserver waiting for client on port 2345...
new connection
I got a connection from (IP: 192.168.2.4 ,PORT: 4319)
new connection
I got a connection from (IP: 192.168.2.4 ,PORT: 4322)
new connection
I got a connection from (IP: 192.168.2.4 ,PORT: 4323)
RECIEVED DATA = i am client1 from IP:192.168.2.4 ,PORT: 4323)
RECIEVED DATA = i am client2 from IP:192.168.2.4 ,PORT: 4323)
RECIEVED DATA = i am client3 from IP:192.168.2.4 ,PORT: 4323)

```

✧ 单线程 IO 多路复用并发服务器

基础知识准备

IO 多路复用是指内核一旦发现线程指定的一个或者多个 IO 条件准备读取，它就通知该线程。和多线程相比，I/O 多路复用技术的最大优势是系统开销小，系统不必创建线程，也不必维护这些线程，从而大大减小了系统的开销。

在 RT-Thread 中，实现 IO 多路复用我们用到 select 这个系统 API。

select 让我们的程序可以监视多个文件句柄(文件描述符或套接字描述符)的状态变化，程序会在 select 这里等待，直到被监视的文件句柄有某一个或多个发生了状态改变。

select 函数形式如下：

```
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
          struct timeval *timeout)
```

select 函数返回值为就绪文件描述符的个数，如果超时返回 0，出错返回-1。

参数 maxfdp1 指定待监测的文件描述符个数，它的值是待测试的最大描述字加 1。

中间的三个参数 readset、writeset 和 exceptset 指定我们要让内核测试读、写和异常条件的文件描述符。如果对某一个的条件不感兴趣，就可以把它设为空指针 NULL。

struct fd_set 可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：

```
void FD_ZERO(fd_set *fdset); // 清空集合
void FD_SET(int fd, fd_set *fdset); // 将一个给定的文件描述符加入集合之中
void FD_CLR(int fd, fd_set *fdset); // 将一个给定的文件描述符从集合中删除
int FD_ISSET(int fd, fd_set *fdset); // 检查集合中指定的文件描述符是否可以读写
```

参数 timeout 告知内核等待所指定描述符中的任何一个就绪的超时时间。其 timeval 结构用于指定这段时间的秒数和微秒数。

```
struct timeval{
    long tv_sec; //seconds
    long tv_usec; //microseconds
};
```

这个参数有三种可能：

1、永远等待下去，仅在有一个描述字准备好 I/O 时才返回。为此，把该参数设置为空指针 NULL。

2、等待一段固定时间，在有一个描述字准备好 I/O 时返回，但是不超过由该参数所指向的 timeval 结构中指定的秒数和微秒数。

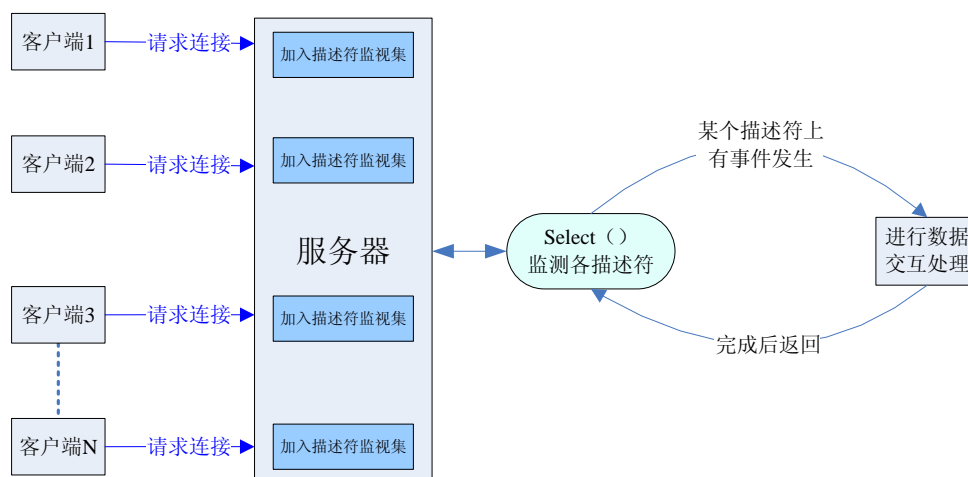
3、根本不等待，检查描述符后立即返回，这称为轮询。为此，该参数必须指向一个 `timeval` 结构，而且其中的定时器值必须为 0。

采用 `select` 函数实现 IO 多路复用的基本步骤如下：

- 1、清空描述符集合；`FD_ZERO`
- 2、建立需要监视的描述符与描述符集合的联系；`FD_SET`
- 3、调用 `select（）` 函数；
- 4、检查所有需要监视的描述符，利用 `FD_ISSET` 宏判断某个描述符是否已经准备好；
- 5、对已经准备好的描述符进行 IO 操作。

实际应用

用 `select` 来实现 TCP 并发服务器的原理如下图所示：



下面是我们给出的一个实际例子，例子中，TCP 服务器同时服务多个客户端，当收到客户端发来的消息后将其打印出来，并将消息原封不动的再发回给客户端。

```
/*
此 demo 用于演示 IO 多路复用 TCP 服务器
*/

#include <rtthread.h>
#include <lwip/sockets.h> /* 使用 BSD Socket 接口必须包含 sockets.h 这个头文件 */

#define SERV_PORT 2345 // 端口 2345, 不能小于 1024, 1-1024 为系统保留端口
#define BACKLOG 5 // 最多允许的客户端连接数
#define BUF_SIZE 1024 // 接收缓冲区长度
```

```

typedef struct _CLIENT{
    int connectfd;
    struct sockaddr_in addr; /* client's address information */
} CLIENT;

void tcpserv(void* parameter)
{
    int maxfd, i, nbyte, res;
    rt_uint32_t sin_size;
    char buf[BUF_SIZE];
    int listenfd, connfd; /* 定义监听 socket 描述符和数据传输 socket 描述符 */
    struct sockaddr_in server_addr, client_addr; /* 本机 IP 和端口号信息 */
    fd_set global_rdfs, current_rdfs;
    int opt;
    CLIENT client[BACKLOG];

    /* 一个 socket 在使用前, 需要预先创建出来, 指定 SOCK_STREAM 为 TCP 的 socket */
    if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        /* 创建失败的错误处理 */
        rt_kprintf("Socket error\n");
        return;
    }

    opt=1;
    setsockopt( listenfd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)); // 最小化
    /* 报文传输的延时 */
    setsockopt( listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)); // 允许地
    /* 址重用 */
    setsockopt( listenfd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt)); // 允许端
    /* 口重用 */

    /* 初始化服务端地址 */
    server_addr.sin_family = AF_INET; /* Ipv4 因特网域 */
    server_addr.sin_port = htons(SERV_PORT); /* 服务端工作的端口这里进行网络字节
    /* 序转换 */
    server_addr.sin_addr.s_addr = INADDR_ANY; /* 本机 IP 地址 */
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero)); /* 清
    0 */

    /* 绑定 socket 到服务端地址 */
    if(bind(listenfd, (struct sockaddr *)&server_addr, sizeof(struct
sockaddr)) == -1)
    {

```

```
/* 绑定失败 */
rt_kprintf("Unable to bind\n");
return;
}

/* 在 sockfd 上进行监听 */
if(listen(listenfd, BACKLOG)==-1)
{
    rt_kprintf("Listen error\n");
    return;
}

rt_sprintf(buf,"%4d", SERV_PORT);
rt_kprintf("\nTCPServer Waiting for client on port %s...\n",buf);

FD_ZERO(&global_rdfs); //清除监视描述符集
FD_SET(listenfd,&global_rdfs); //将listenfd 加入监视描述符集
maxfd = listenfd;
while(1)
{
    current_rdfs = global_rdfs;

    //检测看描述符集上有哪个有事
    res =select(maxfd+1,&current_rdfs,NULL,NULL,NULL);
    if( res <0)
    {
        rt_kprintf("fail to select");
        break;
    }
    elseif(res==0)
    {
        continue;
    }
    else
    { //有描述符上有事情需要处理了
        for(i=0; i<=maxfd; i++)
        { //看看到底是哪个描述符上有事情发生
            if(FD_ISSET(i,&current_rdfs))
            {
                if(i == listenfd)
                { //是监听描述符则进行客户端的连接
                    rt_kprintf("new connection\n");
                    sin_size =sizeof(struct sockaddr_in);
                    if((connfd = accept(i,(struct sockaddr *)&client_addr,
```

```
/* 关闭socket 退出服务 */
```

<http://shop73275611.taobao.com>154 / 157

```
    closesocket(listenfd);
    return;
}

int tcp_demo_init(void)
{
    rt_thread_t demo_thread;

    demo_thread = rt_thread_create("tcp_demo",
                                    tcpserv, RT_NULL, 2048, 20, 20);

    if(demo_thread != RT_NULL)
        rt_thread_startup(demo_thread);

    return 0;
}

#ifdef RT_USING_FINSH
#include <finsh.h>

FINSH_FUNCTION_EXPORT(tcp_demo_init, startup tcp server);

#endif
```

例程列表及下载地址:

✧ 下载地址:

例程源码放在百度网盘:

<http://pan.baidu.com/share/link?shareid=1502327077&uk=506725102>

✧ 例程目录:

篇 2-例程 1-第一次运行 RTT
篇 4-例程 1-自己创建静态、动态线程
篇 6-例程 1-线程的基本管理
篇 6-例程 2-相同优先级线程轮询调度
篇 6-例程 3-线程的让出
篇 6-例程 4-使用空闲线程统计 CPU 使用率
篇 6-例程 5-多线程导致的临界区问题
篇 7-例程 1-调度器上锁
篇 7-例程 2-信号量基本操作
篇 7-例程 3-信号量实际使用-按键点灯
篇 7-例程 4-互斥锁
篇 7-例程 5-邮箱
篇 7-例程 6-消息队列
篇 7-例程 7-事件机制
篇 8-例程 1-finish 组件
篇 9-例程 1-软件定时器
篇 12-字库补丁
第 13 篇-例程 1-RT-Thread 中新的初始化机制（包括魔笛 F1 魔笛 F4）
第 15 篇-例程 1-RTGUI 之 LCD 驱动（包括魔笛 F1 魔笛 F4）
第 16 篇-例程 1-RTGUI 之 Keyboard 驱动（包括魔笛 F1 魔笛 F4）
第 17 篇-例程 1-RTGUI 之 touch panel 驱动（包括魔笛 F1 魔笛 F4）
第 18 篇-例程 1-DNS
第 19 篇-例程 1-网络套接字编程实战演练-TCP 服务器端
第 19 篇-例程 2-网络套接字编程实战演练-TCP 客户端
第 19 篇-例程 3-网络套接字编程实战演练-UDP 服务器端
第 19 篇-例程 4-网络套接字编程实战演练-UDP 客户端
第 19 篇-例程 5-网络套接字编程实战演练-开发板与 PC 图片传输
第 20 篇-例程 1-TCP 并发服务器模型-多线程 TCP 服务器
第 20 篇-例程 2-TCP 并发服务器模型-select IO 多路复用 TCP 服务器

RT-Thread 系统 API 速查手册:

RT-Thread 系统函数 API 请访问: http://www.rt-thread.org/rt-thread/rtdoc_0_4_0/