

关于 RTX51 TINY 的分析与探讨

■ 兰州交通大学 冯桂平 鲁怀伟

1 概述

RTX51 TINY 是一种应用于 MCS51 系列单片机的小型多任务实时操作系统。它完全集成在 Keil C51 编译器中,具有运行速度快、对硬件要求不高、使用方便灵活等优点,因此越来越广泛地应用到单片机的软件开发中。它可以在单个 CPU 上管理几个作业(任务),同时可以在没有扩展外部存储器的单片机系统上运行。

RTX51 TINY 允许同时“准并行”地执行多个任务:各个任务并非持续运行,而是在预先设定的时间片(time slice)内执行。CPU 执行时间被划分为若干时间片,RTX51 TINY 为每个任务分配一个时间片,在一个时间片内允许执行某个任务,然后 RTX51 TINY 切换到另一个就绪的任务并允许它在其规定的时间片内执行。由于各个时间片非常短,通常只有几 ms,因此各个任务看起来似乎就是被同时执行了。

RTX51 TINY 利用单片机内部定时器 0 的中断功能实现定时,用周期性定时中断驱动 RTX51 TINY 的时钟。它最多可以定义 16 个任务,所有的任务可以同时被激活,允许循环任务切换,仅支持非抢占式的任务切换,操作系统为每一个任务分配一个独立的堆栈区,在任务切换的同时改变堆栈的指针,并保存和恢复寄存器的值。RTX51 TINY 没有专门的时间服务函数和任务挂起函数,而是通过 `os_wait()` 中的参数设定实现的。使用 RTX51 TINY 时用户程序中不需要包含 `main()` 函数,它会自动地从任务 0 开始运行。如果用户程序中包含有 `main()` 函数,则需要利用 `os_create_task` 函数来启动 RTX51 实时操作系统。

2 任务切换

2.1 RTX51 TINY 任务状态

RTX51 TINY 的用户任务具有以下几个状态:

① 运行(RUNNING)——任务正处于运行中。同一时刻只有一个任务可以处于“RUNNING”状态。

② 就绪(READY)——等待运行的任务处于“READY”状态。在当前运行的任务退出运行状态后,就绪队列中的任务根据调度策略被调度执行,进入到运行状态。

③ 阻塞(BLOCKED)——等待一个事件的任务处于“BLOCKED”状态。如果等待的事件发生,则此任务进入“READY”状态,等待被调度。

④ 休眠(SLEEPING)——被声明过但没有开始运行的任务处于休眠状态。运行过但已经被删除的任务也处在休眠状态中。

⑤ 超时(TIMEOUT)——任务由于时间片用完而处于“TIMEOUT”状态,并等待再次运行。该状态与“READY”状态相似,但由于是内部操作过程使一个循环任务被切换,因而单独算作一个状态。

处于“READY/TIMEOUT”、“RUNNING”和“BLOCKED”状态的任务被认为是激活的状态,三者之间可以进行切换。“SLEEPING”状态的任务是非激活的,不能被执行或认为已经终止。

2.2 RTX51 TINY 任务切换

任务切换是 RTX51 TINY 提供的基本服务。RTX51 TINY 是基于时间片调度算法的操作系统,它支持的是非抢占式的任务切换。所以在一个任务被执行时不能对其进行中断,除非该任务主动放弃 CPU 的资源,中断才可以打断当前的任务,中断完成后把 CPU 的控制权再交还该被中断的任务。任务切换有两种情况,一种是当前任务主动让出 CPU 资源;另一种情况是在当前任务的时间片已经用完的情况下,进行任务切换。CPU 执行时间被分成若干个时间片,RTX51 TINY 为每个任务分配一个时间片。时间片是通过变量 `TIMESHARING` 的设置来确定的,即用“`TIMESHARING EQU 5;`”设置多少个系统时钟周期为一个时间片。系统默认 5 个系统时钟为一个时间片,如果晶振频率为 11.059 2 MHz,则时间片为 $10.8507 \times 5 = 54.2535 \text{ ms}$ 。

RTX51 TINY 的任务切换共有 `TASKSWITCHING`

和 SWITCHINGNOW 两个入口,前者供定时器 T0 的中断服务程序调用,后者供系统函数 os_delete 和 os_wait 调用。相应地也有两个不同的出口,分别是恢复保护现场和清除状态标志位。系统首先将当前任务置为“TIMEOUT”状态,等待下一次时间片循环,然后找到下一个处于“READY”状态的任务,通过堆栈管理,将自由堆栈空间分配给该任务,使其成为当前任务。清除使该任务进入“READY”或“TIMEOUT”状态的相关位后,执行该任务。任务切换的流程如图 1 所示。

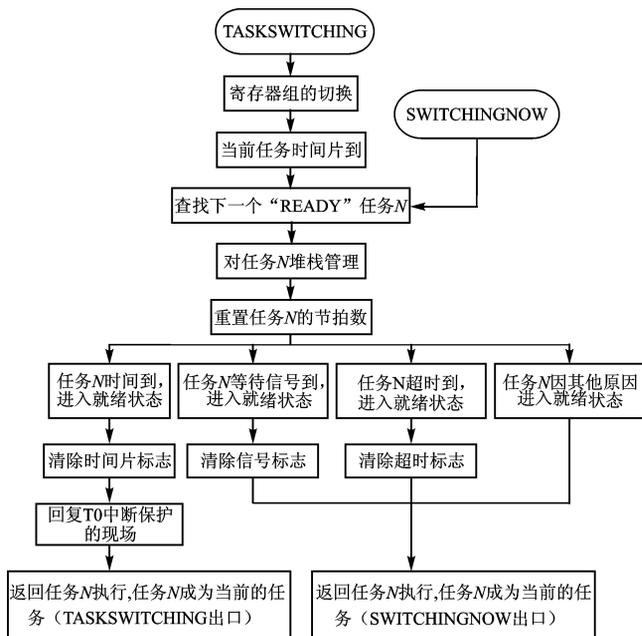


图 1 任务切换流程

3 共享资源实现^[1]

RTX51 TINY 由于是一个多任务的操作系统,那么就不免会有几个任务使用同一个资源,这些资源可能是一个变量,也可能是输入/输出设备。这就要求一个任务在使用共享资源时必须独占该资源,否则可能会造成数据被破坏。

在 RTX51 TINY 中实现共享资源独占的方法比较多。比如,可以通过 TIMESHARING 这个变量来禁止时间片轮转,使其值为 0,就可以实现禁止任务切换,从而当前任务就可以独占共享资源。还可以关闭中断来实现,使 EA=0,定时器 T0 的中断被关闭,不能再为时间片轮转提供基准,从而禁止了任务切换。但这两种方法都带有一定的局限性,前一种方法只能适用于实时性要求不高的场合,后一种方法由于 T0 中断关闭时间不能太长,只能适用于一些简单变量操作的场合。基于以上情况,下面通过另一种方法来实现共享资源的使用。

在 RTX51 full 中可以利用信号量很好地实现对共享资源的操作,也可以把这种思想应用到 RTX51 TINY 中;而在 RTX51 TINY 中不支持信号量,这就要求用户自己定义信号量及其操作过程。以下是部分代码:

```

struct signal { // 定义信号量结构体
    uchar count; // 该信号量的当前计数值
    uint list_tasks; // 等待该信号量任务表
} signal_list[ 3 ];
/* 初始化信号量 */
void init_signal(uchar task_id, uchar count) {
    signal_list[ task_id ].count= count;
    signal_list[ task_id ].list_tasks= 0;
}
/* 等待信号量 */
char wait_signal( uchar task_id ) {
    if( signal_list[ task_id ].count> 0 ) {
        signal_list[ task_id ].count -- ; // 获取信号量
        return( - 1 );
    }
    signal_list[ task_id ].list_tasks|= ( 1 << os_running_task_id
    ( ) ); // 标记为等待状态
    return( 0 );
}
void wait_sem(uchar task_id) {
    if( wait_signal( task_id ) == 0 )
        while( os_wait( K_TMO, 255, 0 ) != RDY_EVENT );
        // 等待,直到该任务就绪
}
/* 释放信号量 */
char release_signal(uchar task_id) {
    uchar i;
    uint temp= 1;
    if( ( signal_list[ task_id ].count> 0 ) || ( signal_list[ task_
    id ].list_tasks== 0 ) ) {
        signal_list[ task_id ].count+ + ; // 释放信号量
        return( - 1 );
    }
    for( i= 0; i< 16; i+ + ) {
        if( ( signal_list[ task_id ].list_tasks& ( temp ) ) != 0 ) {
            // 查找任务表
            signal_list[ task_id ].list_tasks&= ~ ( 1 < i );
            return( i ); // 返回等待信号量的任务号
        }
        temp<<= 1;
    }
}
void release_sem( uchar task_id ) {
    char task_temp;

```

```

task_temp= release_signal( task_id);
if( task_temp!= - 1) {
    os_set_ready(task_temp); //任务 task_id 进入就绪状态
    os_switch_task();
}
}

```

有了以上几个函数的定义和实现, 就可以应用等待信号量和释放信号量来完成对共享资源的独占。例如:

```

void job() _task_id {
    用户代码
    wait_sem(task_id); //等待任务 task_id 的信号量
    对共享资源使用代码
    release_sem( task_id); //释放任务 task_id 的信号量
    用户代码
}

```

应用信号量来实现共享资源的使用, 不用禁止时间片轮转和关闭 T0 中断, 可以有效地实现对共享资源的独占; 但增加了代码, 等待和释放信号量花费了一定的时间, 在具体应用中要视情况而定。

4 需要注意的问题

在应用 RTX51 TINY 时应注意以下几点:

① 尽可能不使用循环任务切换。使用循环任务切换时要求有 13 个字节的堆栈区来保存任务内容(工作寄存器等)。如果由 os_wait() 函数来进行任务触发, 则不需要保存任务内容。由于正处于等待运行的任务并不需要等待全部循环切换时间结束, 因此 os_wait() 函数可以产生一种改进的系统响应时间。

② 不要将时钟节拍中断速率设置得太高, 设定为一个较低的数值可以增加每秒的时钟节拍个数。每次时钟节拍中断大约需要 100~ 200 个 CPU 周期, 因此应将时钟节拍率设定得足够高, 以便使中断响应时间达到最小化。

③ 在 os_wait() 函数中有 3 个参数: K_TMO、K_IVL 和 K_SIG。其中对于 K_TMO 和 K_IVL 的使用要加以区别。在使用时, 两者似乎差别不是很大。其实不然, 两者

存在很大的区别: K_TMO 是指等待一个超时信号, 只有时间到了, 才会产生一个信号。它产生的信号是不会累计的, 产生信号后, 任务进入就绪状态。而 K_IVL 是指周期信号, 每隔一个指定的周期, 就会产生一次信号, 产生的信号是可以累计的。这样就使得在指定事件内没有响应的信号, 通过信号次数的叠加, 在以后信号处理时, 重新得以响应, 从而保证了信号不会被丢失。而通过 K_TMO 方式进行延时的任务, 由于某种原因信号没有得到及时的响应, 那么这样就可能会丢失一部分没有响应的信号。不过两者都是有效的任务切换方式, 在使用时要根据应用场合来确定对两者的使用。

结 语

RTX51 TINY 实时操作系统既能保证对外界的信息以足够快的速度进行相应处理, 又能并行运行多个任务, 具有实时性和并行性的特点, 因此能很好地完成对多个信息的实时测量、处理, 并进行相应的多个实时控制。任务切换是 RTX51 TINY 的一个基本服务。本文对任务切换做了详细的分析, 在实际应用中还要对任务切换时的堆栈管理有一定了解, 这样才能更好地掌握任务切换的机制。共享资源的使用在多任务操作系统中是不可避免的, RTX51 TINY 中没有专门的处理共享资源函数, 所以在实际应用中要视情况来应用文中提到的几种方法。

参考文献

- [1] 朱珍民, 隋雪青, 段斌. 嵌入式实时操作系统及其应用开发 [M]. 北京: 北京邮电大学出版社, 2006: 44 - 49.
- [2] Keil Software Inc. RTX51 Tiny User's Guide, 2004.
- [3] 徐爱钧, 彭秀华. 单片机高级语言 C51 Windows 环境编程与应用 [M]. 北京: 电子工业出版社, 2001.

冯桂平(硕士研究生), 主要研究方向为单片机开发与应用; 鲁怀伟(教授、博导), 主要研究方向为光纤无源器件。

(收稿日期: 2007-12-03)

NXP 推出 4 款全新 32 位 ARM9 微控制器

恩智浦半导体(NXP Semiconductors)(由飞利浦创建的独立半导体公司)推出全新的 LPC3200 系列微控制器, 进一步扩展了 ARM7 和 ARM9 微控制器产品线。恩智浦 LPC3200 系列基于广受欢迎的 ARM926EJ 处理器, 针对消费电子、工业、医疗和汽车电子应用, 为设计师提供一种高性能、高功耗效率的微控制器。

全新的恩智浦 LPC3200 系列采用 90 nm 工艺设计, 结合了一个 ARM926EJ 核、一个矢量浮点协处理器(VFP)、一个 LCD 控制器、一个以太网 MAC、On The Go USB、一个高效的总线阵列以及大量的标准外设, 使得嵌入式系统设计师能够在不损失任何性能的前提下减少片上器件数量, 并且最大程度地节省功耗。