

微内核结构嵌入式实时操作系统的研究与设计

邓 昀, 程小辉, 王新政

(桂林理工大学 信息科学与工程学院, 广西 桂林 541004)

摘 要: 分析了微内核结构在设计嵌入式实时操作系统时的优点, 设计了一个微内核结构的嵌入式实时操作系统, 整个系统内核由任务管理、任务通信、时钟管理、中断管理、内存管理等模块组成. 内核采用抢占式调度和分时调度相结合的调度策略, 以及基于查表法的优先级队列管理, 使得系统在进行任务调度时能快速的找到所要调度的任务, 满足实时性的要求. 内核提供多种通信方式, 并对控制临界资源访问的信号量进行了特别的设计, 便于任务在必要的时候进行优先级继承, 很好地解决了抢占式嵌入式操作系统中普遍存在的优先级反转问题. 最后, 在基于 ARM 处理器的 EL-ARM-830 硬件平台, 给出了系统的实现.

关键词: 微内核结构; 嵌入式实时操作系统; 任务管理; 任务通信; 优先级继承

中图分类号: TP391. 41

文献标识码: A

文章编号: 1000-7180(2012)10-0133-07

Study and Design of Embedded Real-time Microkernel Operating System

DENG Yun, CHENG Xiao-hui, WANG Xin-zheng

(Institute of Information Science & Engineering, Guilin University of Technology, Guilin 541004, China)

Abstract: The thesis analyzes the advantage of the structure of micro-kernel when designing the embedded real-time operating system. An embedded real-time operating system for micro-kernel structure is designed. The kernel of the system is made up of the task manager, task communication, clock management, interrupt management, memory management and other modules. The core uses the combination of preemptive scheduling and time-sharing scheduling strategy, and priority-based look-up table queue management. It makes the system find the tasks quickly when dispatching them which meets the real-time requirements. The kernel provides a variety of communication, and designs a special control access to critical resources which makes it easy to take priority over other tasks when necessary. It proposes a good solution to the common problem of inversion of embedded operating systems preemptive priority. Finally, the thesis gives the system implementation on the ARM-based processor EL-ARM-830 hardware platform.

Key words: micro-kernel structure; embedded real-time operating system; task management; task communication; priority inheritance

1 引言

目前绝大多数的嵌入式系统采用高性能的微内核设计, 因为微内核具有功能单一、体积小, 具有很高的效率, 系统响应快等特点, 可用于实时性要求的场合, 且微内核采用客户-服务器的工作方式, 使

得系统更易于扩展, 符合嵌入式系统的要求.

本文描述了一个微内核结构的嵌入式实时操作系统设计, 整个系统内核由任务管理、任务通信、时钟管理、中断管理、内存管理等模块组成. 内核采用抢占式调度和分时调度相结合的调度策略, 以及基于查表法的优先级队列管理, 使得系统在进行任务

调度时能快速找到所要调度的任务,满足实时性的要求. 内核提供多种通信方式,并对控制临界资源访问的信号量进行了特别的设计,便于任务在必要的时候进行优先级继承,很好解决了抢占式嵌入式操作系统中普遍存在的优先级反转问题.

2 微内核整体结构

系统采用模块化结构设计,最多可支持 64 个优先级队列^[1],256 个任务,同一优先级队列上有多个任务,对于不同优先级队列的任务采用抢占式的调度策略,对于同一优先级队列上的任务采用时间片轮转的分时调度策略,整个系统中总是就绪的优先级最高的那个队列的任务优先得到执行. 系统内核分为任务调度、任务通信、中断管理、临界资源管理、时钟管理、内存管理等几个模块,整体结构如图 1 所示.

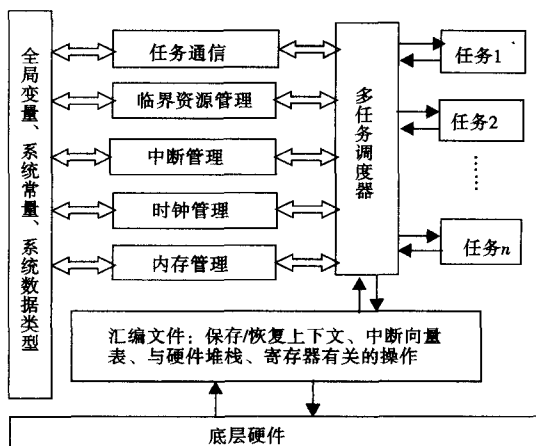


图 1 系统内核结构

为了便于系统在不同硬件平台上移植,不直接采用 int、long 等数据类型,因为这些数据类型与处理器和编译器有关,而是采用更直观的数据类型表示方式,如 8 位无符号整数就表示成 UINT8,当系统移植到具体的硬件平台时,可根据具体的处理器和编译器来进行设定,如可以用 typedef unsigned char UINT8 来定义 UINT8.

3 任务管理模块的设计

系统中每个任务定义了一个任务控制块 TCB (Task Control Block),包含与任务相关的所有信息,如任务执行时 CPU 的所有寄存器中的值、任务的状态以及任务堆栈的内容等. 本系统在处理寄存器和堆栈等上下文内容时,没有像 LINUX 等操作系统那样专门定义一个数据结构,而只是在任务控

制块里用了一个指针 task_stk_ptr 指向这部分内容的存储空间,具体的大小用户可以自己指定,以便于移植. 任务控制块 TCB 的结构如下:

```
typedef struct task_tcb{
    TASK_STACK * task_stk_ptr; /* 指向保存任务上下文堆栈的指针 */
    UINT8 task_id; /* 任务号,必须唯一, */
    UINT8 task_priority; /* 任务的优先级,未提升时与 primitive_priority 同 */
    UINT8 primitive_priority; /* 任务的原始优先级,正常 */
    UINT8 task_name[15]; /* 保存任务名的字符数组 */
    struct task_tcb * task_previous; /* 指向任务链表中的前一个任务控制块 */
    struct task_tcb * task_next; /* 指向任务链表中的后一个任务控制块 */
    UINT16 total_time; /* 时间片轮转调度时分配给任务的时间数 */
    UINT16 remain_time; /* 当被高优先级的任务剥夺时,如果时间片未用完则保存在此,任务刚开始被调度时与 total_time 相同 */
    UINT8 task_state; /* 任务状态 */
    void * task_wait; /* 指向任务等待的信号量、事件、或消息的指针 */
} TASK_TCB;
```

3.1 优先级队列结点的设计

在优先级队列表里,每个元素控制着这一优先级队列的任务,系统支持的最大优先级数为 64 级^[2]. 在进行任务调度时首先通过 task_queue_grp 和 task_queue_tbl[] 两个变量的值来确定当前就绪的任务中优先级最高的任务所在的队列,然后让最高就绪优先级队列指针 queue_high_ready_pt 指向该队列,如果队列的优先级比当前优先级队列指针 queue_current_pt 所指向的优先级要高,则保存正在执行的任务的上下文,把当前任务队列指针指向最高就绪优先级队列指针所指的队列,实现运行任务队列的切换,然后从这个队列的队列头取一个任务来运行. 在运行中系统总是对 queue_current_pt 指针所指向的队列采用时间片轮转的调度算法,当任务时间片用完后,重新设置时间片,并挂到队列尾,再从队列头取一个任务来运行. 如果队列上只有一个任务时,系统重新设置任务时间片后,继续运行该任务. 当任务在运行过程中有新的任务就

绪则需要通过优先级队列表重新定位优先级最高的任务队列,如果需要切换任务队列,但当前任务的时间片还没用完,则它并不挂到队列末尾,而是放在队列头,直到在下次调度中用完了剩余的时间片,才重新设置时间片的值,然后把任务挂到队列尾^[3]. 任务队列结点 QUEUE 的数据结构如下:

```
typedef struct queue{
    UINT8 queue_priority; /* 队列优先级 */
    UINT8 task_counter; /* 连在本队列上的任务数 */
    TASK_TCB * queue_head; /* 队列头指针 */
    TASK_TCB * queue_end; /* 队列尾指针 */
} QUEUE;
```

就绪任务和任务队列的关系如图 2 所示. 优先级表 Queue_Table[] 中各元素存放的队列的优先级即为其下标的值,如元素 Queue_Table[1] 存放的即是优先级为 1 的任务队列.

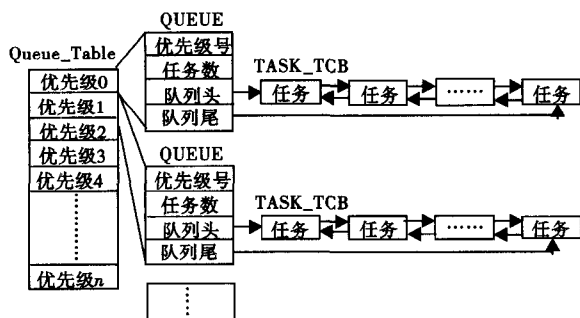


图 2 就绪任务和任务队列关系图

3.2 任务的创建

创建任务可以通过调用 `tsk_create(void (* task)(void), TASK_STACK * ptos, UINT8 prio, UINT8 id, UINT8 * pname)` 函数来完成,通过参数把任务的各种信息,如任务代码的起始地址、优先级、任务 id 号、堆栈地址等填入到任务控制块 TCB 中,如果任务创建成功则返回一个指向任务的指针,以后对任务的所有操作,如任务的调度、删除、挂起等都通过这个指针所指向的任务 TCB 来进行. 要创建一个新的任务,首先要看为任务指定的任务 ID 号是否已经被占用,因为任务的 ID 号在系统里必须是唯一的,如果已经被占用,则返回任务 ID 号被占用的错误信息. 如果 ID 号未被占用,就看系统中是否还有空闲的任务控制块,如果空闲任务控制块已被用完,则返回任务控制块被用完的出错信息,否则就从空闲任务控制块队列中取出一个任务控制块,然

后把任务的各种信息都填入任务控制块,再把此任务挂到相应优先级的就绪任务队列上. 任务可以在系统启动多任务调度前被创建,也可以在系统启动多任务后由任务或中断服务程序来创建. 如果系统已经处在多任务运行状态,即系统多任务运行标志 `task_running` 为 `TRUE`,则在任务创建结束或中断退出时调用 `tsk_schedule()` 函数来重新调度任务,看是否有新的更高优先级的任务就绪. 如果 `task_running` 为 `FALSE` 则任务创建结束后直接返回.

3.3 任务的调度

本系统结合了基于优先级抢占式调度^[4]和基于时间片轮转^[5]的分时调度两种方式,系统的任务优先级队列通过一张优先级队列就绪表来进行管理,最多可管理 64 个优先级队列,256 个任务,每个优先级队列上可有多个任务. 在进行应用系统开发时,如果所有的任务都定义在一个优先级上,则系统就完全变成了基于时间片轮转的系统;如果每个优先级上最多只设置了一个任务则系统就是完全的抢占式系统. 当一个任务运行结束或挂起,系统在运行过程中有新的任务被创建,有资源被释放使得新的任务就绪或任务提出资源申请得不到满足等都可能使调度器重新调度任务.

在系统里通过对优先级队列就绪表进行查表能快速的找到优先级最高的就绪队列,而且与系统中当前的就绪任务数无关,比遍历任务队列的方法性能要优越. 当查找到下一个将要运行的优先级队列后,队列头的那个任务即为将要调度运行的任务,就绪任务和其对应的队列关系如前面的图 2 所示.

在定位优先级队列时要用到两个量:优先级表 `task_queue_tbl[]` 和优先级组变量 `task_queue_grp`. 之所以把任务优先级队列设计成 64 个,是为了在提高系统性能的同时,能节省系统的存储空间,因为查找表随优先级数的增加成倍的扩大,且 64 个优先级已能满足绝大部分的应用要求. 在设计时把 64 个优先级分成了 8 组,每组 8 个,查找表时就不会过于庞大. 优先级表是一个数组 `task_queue_tbl[8]`,有 8 个元素,每个元素都是一个 8 位的无符号数,每一位表示一个优先级,如果该优先级队列上有就绪任务则该位置 1,否则清 0,同理,如果某组上有任务则 `task_queue_grp` 里与该组相关的位置 1,否则清 0. `task_queue_grp` 与 `task_queue_tbl[]` 的关系如图 3 所示.

在需要重新定位任务队列时,系统根据 `task_queue_grp` 的数值和 `task_queue_tbl[]` 数组的对应

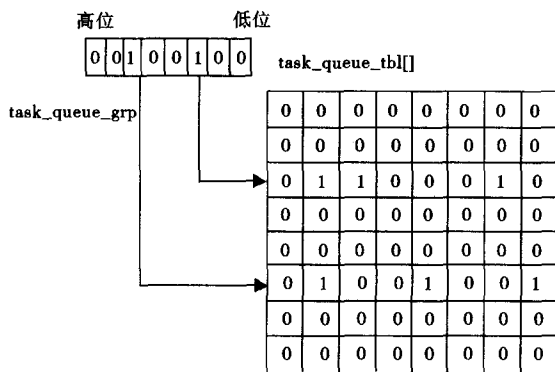


图3 优先级组与优先级表关系

数值查 search_table[] 表得到最高就绪任务队列的优先级. search_table[] 是一个事先设计好的常量数组, 如图 4 所示. 数组有 256 个元素, 每个元素为一个 8 位的无符号整数, 其内容通过计算来填好. 如 task_queue_grp 里的数值为 00100100, 即 0x24, 用 task_y 来表示任务所在的组, 通过查 search_table[], 有 task_y = search_table[0x24] = 2, 可知就绪任务中优先级最高的任务必定在第 2 组里, 即 task_queue_tbl[2] 里 (task_queue_tbl[0] ~ task_queue_tbl[7] 共 8 组), 又通过对第 2 组里的数据查表即可得是第 2 组里的第几个元素, 在这里, 第 2 组的数据 task_queue_tbl[2] = 0x62, 用 task_x 表示指定组里的第几个元素, 则有 task_x = search_table[0x62] = 1, 即系统中优先级最高的任务是第 2 组的第 1 个元素 (每组有 0 ~ 7 共 8 个元素). 用 highest_pri_rdy_num 来表示就绪表中优先级最高的优先级号, 则有 highest_pri_rdy_num = 2 * 8 + 1 = 17, 即就绪表中, 优先级为 17 的任务队列优先级最高.

当被调度队列的优先级确定后, 就能很容易找到将被切换进的任务了, 然后看任务队列上是否有多个任务, 如果有多个任务则把系统模式 sys_mode 置为 SYS_MODE_ROLL, 系统在没有更高优先级任务产生时就轮流调度此队列上的任务, 如果只有一个任务则把 sys_mode 置为 SYS_MODE_PREM, 然后进行任务切换.

4 任务间通信和同步机制的设计

微内核结构的嵌入式操作系统是建立在多任务间通信的概念上的, 任务间存在着各种复杂的逻辑关系, 必须通过通信在任务之间进行信息交换, 通过通信使得这些任务协调的工作. 内核提供了信号量、邮箱和队列多种通信方式.

search_table[]	0	0	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0
1	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
2	5	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
3	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
4	6	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
5	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	1	0
6	5	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
7	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
8	7	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
9	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
10	5	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
11	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
12	6	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
13	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
14	5	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0
15	4	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	0

图4 优先级查找表

系统中设计了一个信号量控制块 SCB (Semaphore Control Block), 通过信号量控制块数组下标或信号量句柄都能找到所需要的信号量, 查询到信号量的状态信息、等待该信号量的任务列表等. 除在系统启动时把信号量控制块数组初始化成空闲控制块链表的函数 scb_init() 外, 系统里还设计了 6 个对信号量进行操作的函数:

/* 创建信号量, 返回指向被创建信号量的指针作为该信号量的句柄 */

SCB * sem_create (UINT8 semtype, UINT8 * pname, UINT8 priority, UINT8 maxcount);

void sem_wait(SCB * pscb); /* 等待信号量 */

void sem_signal(SCB * pscb); /* 释放信号量 */

void sem_delete(SCB * pscb); /* 删除信号量 */

void sem_waits(SCB * pscb, ...); /* 等待一组信号量 */

void sem_signals(SCB * pscb, ...); /* 释放一组信号量 */

本系统设计的邮箱只能存放一条消息, 每次向邮箱写入消息时, 如果邮箱为空则把消息放到邮箱里, 如果邮箱满就把任务挂在写等待队列上, 然后重新调度, 选择一个新的任务投入运行, 当邮箱的消息被取走后再唤醒该任务; 当任务从邮箱读取消息时, 邮箱有消息则直接读取消息, 如果邮箱为空则把任务挂在邮箱的读等待队列上, 然后重新调度, 选择一

个新的任务投入运行,有消息到来时再唤醒任务,把消息读取。邮箱的结构如下:

```
typedef struct mail_box {
    UINT8 mtype; /* 类型 */
    struct mailbox * pnext_mbox /* 指向下一个邮箱 */
    UINT8 mboxid; /* 邮箱内部 ID 号,其值在初始化邮箱结点数组时赋值为数组的下标。 */
    UINT8 mcnt; /* 消息数 */
    TASK_TCB * pwrite; /* 邮箱满时,发送消息的任务挂在此指针指向的队列上等待 */
    TASK_TCB * pread; /* 邮箱空时,取消息的任务挂在此指针指向的队列上等待 */
    void * pmsg; /* 指向邮箱消息 */
}MAIL_BOX;
```

对消息队列的操作与邮箱相似,消息队列可以存放多条消息,就像一个存放消息的缓冲池,只要队列不空,任务就可以从队列中取消息,同样,只要队列不满,任务就可以向队列发消息。每条消息就是一个记录,消息所代表的含义由通信双方进行约定,通信的任务可以对消息队列进行读取和写入,消息队列的大小在建立时根据需要进行设定。采用消息队列的通信机制,发送消息的任务不必等待接收任务收取消息就可以继续发送(消息队列未滿),接收消息的任务,只要消息队列不空就可以从队列中取走消息,这样能在通信的任务间进行异步的信息传递,当需要交换的数据量很大时,采用这种方式能提高效率。对消息的存取采用 FIFO(先入先出)的方式,取数据的任务从队列头读取数据,发送消息的任务把数据存放在消息队列尾,整个消息队列是一个循环队列。消息队列结点的结构如下所示:

```
typedef struct msg_queue {
    UINT8 msgtype; /* 类型 */
    struct msgqueue * pnext_msgqueue; /* 下一个消息队列 */
    UINT8 msgqid; /* 邮箱队列的内部 ID 号 */
    UINT8 queuesize; /* 消息队列最大能存放的消息数 */
    UINT8 msgcnt; /* 队列中的消息数 */
    TASK_TCB * pwrite; /* 队列满时,写消息的任务挂在此指针指向的队列上等待 */
    TASK_TCB * pread; /* 队列空时,读消息的任务挂在此指针指向的队列上进行等待 */
    void * * phead; /* 指向消息队列头的指针
```

```
*/
    void * * pout; /* 指向下一条将要被读取的消息 */
    void * * pin; /* 指向下一个存放消息的空间 */
    void * * pend; /* 指向消息队列尾 */
}MSG_QUEUE;
```

5 任务优先级继承的设计

系统中采用优先级继承的方法解决优先级反转问题。优先级反转问题的产生归根到底是因为多个不同优先级的任务需要对同一共享临界资源进行访问而引起的,因此,要从对共享临界资源的访问控制上来解决优先级反转问题。在对需要互斥访问的临界资源进行保护时可以采用关中断/开中断或信号量的方式来处理,开/关中断的方式不会引起优先级反转问题,但是,如果对临界资源的访问需要的时间很长则不宜采用关中断的方式,因为关中断的时候会阻止系统对外部事件的响应,从而降低了系统的实时性,因此只能采用互斥信号量的方式来进行保护。

信号量控制块(SCB)有 ptskwait 和 ptskget 指针分别指向等待该信号量的任务队列和正在使用该信号量的任务。在使用信号量前,首先用 sem_create()创建一个二值信号量,如果信号量创建成功则返回指向被创建信号量的指针作为该信号量的句柄。当访问临界资源时先对该句柄执行 sem_wait()操作,sem_wait()操作判断信号量是否可用,如果信号量可用,即 count 减 1 后是否大于等于 0, count 可能为 1、0 和负数三种情况,1 表示任务可以获得信号量,其他表示任务不能马上获得信号量,需挂起等待,如果大于等于 0 则让 ptskget 指针指向此任务,否则说明信号量不可用,把任务挂在该信号量的等待队列上,同时把当前任务的优先级与已获得信号量的任务即 ptskget 所指向的任务的优先级进行比较,如果 ptskget 所指向的任务的优先级比当前任务的优先级低,则把它提高到当前任务的优先级,重新调用 tsk_schedule()进行任务调度。在完成对共享临界资源的使用后,用 sem_signal()来释放信号量,如果任务被提升过优先级,就恢复到原来的优先级。在信号量的等待队列上如果没有任务在等待,就简单的把信号量的计数值 count 加 1,否则,把信号量给等待队列中优先级最高的那个任务,当多个任务的优先级相同时,则给先等待的那个任务。

6 内存管理的设计

系统采用了二元伙伴算法来对内存进行管理,把除系统内存和堆栈之外的内存空间划分为 N 个区,每个区中内存块的大小为 $2n$ kB,并且把分区内的内存块连成一个双向链表^[6-7].分区和内存块的关系结构如图 5 所示.

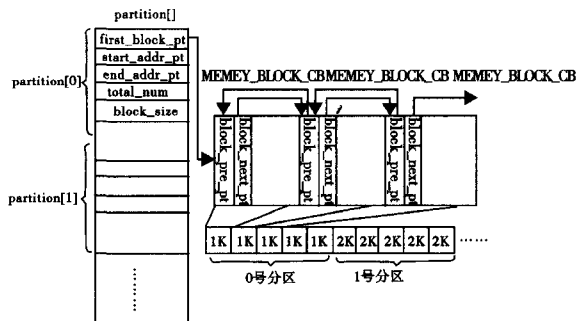


图 5 分区和内存块的关系图

伙伴算法的思想就是在分配内存时,系统根据申请内存块的大小,查找最小且有能满足要求的空闲内存块的分区链表.假设任务需要的内存空间为 1 kB,系统首先从 0 号分区中查找,如果有空闲内存块,就从链表头取一内存块分配出去,否则在 1 号分区中查找,如果找到空闲块,则把空闲块分成相等的两部分,前面的 1K 挂到 0 号分区的链表头部,后面 1K 分配给任务.如果 1 号分区中仍然没有空闲块,则从 2 号分区中进行查找,如果找到,则把内存块进行划分,先把 4K 的空间分成两个 2K,前面的 2K 挂到 1 号分区上,后面的 2K 再进行划分,前 1K 挂到 0 号分区上,后 1K 分配给任务,如果 2 号分区也没有空闲块,则重复前面步骤,直到所有分区都没有符合要求的空闲块,则把任务挂起.

释放内存块时,先查看内存块当前所属链表上是否有伙伴存在,如果没有伙伴存在就直接把要释放的内存块挂在链表头;如果有伙伴内存块存在,则把伙伴块从链表中取下来,合并成一个更大的内存块,然后用合并后的内存块在下一个分区中找伙伴块,如果找到就继续合并,直到没有伙伴块可合并为止.两个内存块要成为伙伴块,必须满足三个条件:(1)两个内存块的大小相同;(2)两个内存块的地址连续;(3)两个内存块是从同一分区中分离出来的.

7 中断管理模块的设计

系统在设计时为了提高对有高实时性要求的中断处理的响应,中断服务程序(ISR)只是负责在最

小的时间内通知系统中断的产生,除非是异常紧急的一些中断请求才放在中断服务程序中处理,否则,将中断引起的不要求实时处理功能放在相应的任务中实现,这使得系统对中断的反应速度能得到及大的提高.系统通知中断处理的任务是通过在中断服务程序里向中断处理的任务发送消息来实现的,中断处理任务在收到消息后,将在适当的时候完成对中断的处理.

8 系统在 ARM 平台上的实现

操作系统在基于 ARM 处理器的 EL-ARM-830 硬件平台进行了实现.

8.1 系统初始化代码的设计

系统的初始化分为两部分,一是对硬件平台的初始化,二是对操作系统的初始化.在系统启动文件里通过以下几步来对硬件进行初始化^[8-9]:

- (1) 设置中断向量表.系统在启动或复位后都是从 0X0 地址开始取址执行程序代码的,并且从 0X0 开始到 0X1C 的 32 个字节是系统的中断向量表空间,要在这里存放中断服务程序的入口地址.
- (2) 关闭看门狗,禁止所有的中断,设置时钟模块.
- (3) 初始化存储器.
- (4) 初始化各工作模式的堆栈.
- (5) 初始化数据存储器,并切换到用户模式.

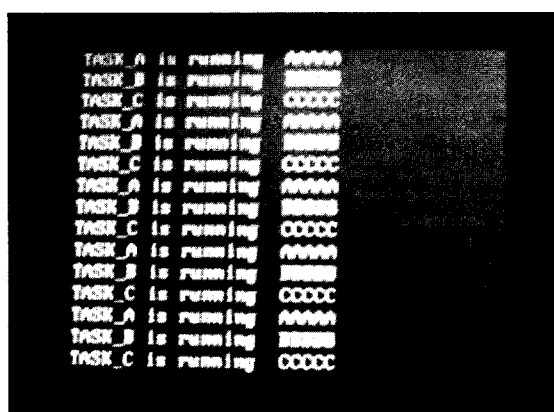
在完成以上部分的硬件初始化后,进入操作系统前还需要对各外设部件进行初始化,如 LCD 显示屏、键盘、串口等.各外设的初始化程序都在对应外设的驱动程序里,通过在 target_init() 函数里调用各外设初始化函数来完成对外部设备的初始化.

操作系统的初始化主要完成对一些系统公共变量赋初值,如任务数、优先级表、各任务队列等;把一些控制块连成链表,如空闲任务控制块 TASK-TCB、信号量控制块 SCB 等;在操作系统初始化的最后创建一个空闲任务,其优先级设置为 MAX-TASK-LEVEL-1, ID 号设置为 MAX-TASK-NUM-1,这样能保证系统在进行多任务调度时至少有一个任务可调度,这是一个优先级最低, ID 号最大的任务,在系统运行的整个过程中不能被删除.操作系统初始化的所有工作通过调用 ts_k_system_init() 函数来完成.

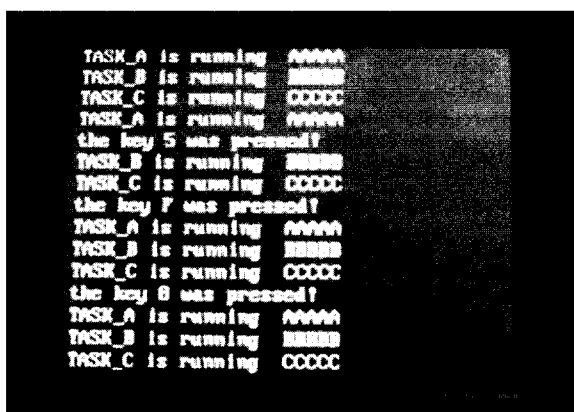
8.2 多任务调度测试

设计了同优先级任务调度和抢占式调度测试程序.同优先级任务调度程序设置了 3 个任务

TASK_A、TASK_B 和 TASK_C,给这三个任务的优先级都为 10,任务 id 号分别为 10、11 和 12. 抢占式调度程序中增加了键盘处理任务 TASK_KEY,把键盘处理任务的优先级设置为 9,任务 id 号为 9,由于其优先级比 TASK_A、TASK_B 和 TASK_C 都要高,所以只要 TASK_KEY 就绪,就会马上得到执行. 平台上键盘采用的是中断方式,当有键按下时,系统会产生一个中断,在这里,中断服务程序并不对按键做太多的处理,只是向保存按键的邮箱发送消息,等待消息的键盘处理任务 TASK_KEY 在收到消息后就会被置为就绪,然后对按键进行处理. 测试结果如图 6 所示.



(a) 时间片分时调度



(b) 高优先级任务抢占低优先级任务

图 6 测试结果

于优先级和时间片相结合的调度算法,根据任务对时间要求的严格程度不同赋予不同的优先级,可很好地满足嵌入式应用中的实时性要求. 在进行任务调度时,通过预先设计好的查找表,能快速的找到所要调度的任务,速度快、效率高. 对控制临界资源访问的信号量进行了特别的设计,能够克服任务调度过程中出现的优先级反转问题,使系统更加稳定. 该系统可以被定制为面向嵌入应用的特定操作系统,体现了微内核结构在嵌入式操作系统开发中的优越性.

参考文献:

- [1] 季虹,付少锋,等. 周利华. 实时嵌入式操作系统 $\mu C/OS-II$ 内核的分析与改进[J]. 计算机工程,2007,33(16):246-247.
- [2] 王新政,程小辉,周华茂. 实时操作系统任务调度策略的研究与设计[J]. 微计算机信息,2007,23(4-2):57-59.
- [3] 周昕,傅鹏,黄海伦. 面向实时嵌入式操作系统的进程机制[J]. 计算机工程,2010,36(15):51-54.
- [4] 霍华,崔欢欢. 一种新的嵌入式 Linux 实时调度算法[J]. 微电子学与计算机,2011,28(6):138-141.
- [5] 薛安琪,屈玉贵,赵宝华. 一种新的实时系统内核调度算法[J]. 计算机工程,2011,37(6):286-288.
- [6] 赵跃华,蔡贵贤,黄卫菊. 一种嵌入式安全内存管理的设计与实现[J]. 计算机工程与设计,2006(8):3092-3096.
- [7] 李平勇,游磊. 嵌入式操作系统 $\mu C/OS-II$ 的一种内存管理算法[J]. 微电子学与计算机,2011,28(11):98-101.
- [8] 蒋建春,汪同庆. 一种异构多核处理器嵌入式实时操作系统构架设计[J]. 计算机科学,2011,38(6):298-302.
- [9] 张荫芾,应忍冬,周玲玲. 支持多核架构的微内核操作系统设计[J]. 计算机工程,2009,35(23):249-251.

作者简介:



邓昀男,(1980—),硕士,讲师. 研究方向为嵌入式技术、物联网技术.

9 结束语

本文论述了微内核结构的嵌入式实时操作系统的设计,实现了多任务的调度、任务间的通信、时钟管理、内存管理等功能. 系统在任务调度方面采用基