《嵌入式实时操作系统》课程实验指导书

I、操作系统实验部分

实验 1　进程管理

实验 1.1　进程创建

- 内容：编写一段程序，使用系统调用 fork( )创建两个子进程，再用系统调用 signal( )让父进程捕捉键盘上来的中断信号（即按 ctrl+c 键），当捕捉到中断信号后，父进程用系统调用 kill( )向两个子进程发出信号，子进程捕捉到信号后，分别输出下列信息后终止：

  Child process 1 is killed by parent!

  Child process 2 is killed by parent!

  父进程等待两个子进程终止后，输出以下信息后终止：

  Parent process is killed!
- 实验要求：

  (1) 运行程序并分析结果。

  (2) 在程序中哪些地方分别用了系统调用 fork（），wait()， kill（）和 exit(), 有什么作用？

实验 1.2 模拟进程调度算法

内容：实现教材中所描述的先来先服务算法，短进程优先调度算法和时间片轮转调度算法。

要求：

编写一模拟程序，实现几种常见的进程调度算法，通过对几组进程分别使用不同的调度算法，计算进程的平均周转时间，比较算法的优劣。

实验内容：

编程实现本实验的程序，要求：

【a】编程实现进程的进程控制块，定义一个进程控制块的数据结构（PCB）应包括：

- 进程名称（ID）
- 进程需要执行时间
- 进入就绪队列时间
- 进程执行开始时间，进程执行结束时间
- 进程的优先级

【b】进程调度实现 **先来先服务算法、短进程优先调度算法、时间片轮转调度算法**。

【c】进程及相关信息从文件中读入。（读入信息包括：进入就绪时间，需要执行时间，优先级）；模拟 5 个进程序列，每个序列包括 10 个进程

【d】时间片与时间流逝的模拟。（运行一个时间片，表示为已占用 CPU 时间加 1，需要执行时间减 1。）

【e】一组进程序列执行完毕，打印出结果信息: 按照进程的 ID 输出其执行序列,计算出每个进程的开始时间、 结束时间、周转时间，并为整个进程序列计算**平均周转时间**。

程序的计算结果按一定的格式显示在计算机屏幕上或输出到文件中。

实验 1.3 多进程（线程）实现快速排序

- 内容：编多进程及多线程程序实现对 10000 个随机数的快速排序算法。
- 要求：

产生 10000 个随机数，编写一个多进程（线程）实现快速排序算法; 要求说明你的程序运行的系统资源配置，给出测试结果并对测试程序和结果做出说明. 回答用多进程实现和多线程实现体现了什么差异，产生的原因是什么？

- 实现提示:
  每次数据分割后产生两个新的进程（线程）处理分割后的数据;
  每个进程（线程）处理的数据小于 500 以后不再分割

实验 2 存储管理

内容：

编程序模拟请求页面调度中的页面置换算法，基于同样的页面序列对比它们的命中率，观察当物理内存容量变化时算法的结果有什么不同。

要求：

设页面大小为 1K，用户虚存容量 32K，物理内存大小从 4K 到 32K 分别进行算法测试，每次测试请给出 **FIFO, LRU, OPT** 的命中率。分析实验结果，对比三种算法性能，以及当物理内存大小变化时命中率的变化。

方法：

1. 页面序列生成：
   a) 生成 320 条指令组成的序列，其地址顺序假定为：1）当前指令的下一条指令是顺序执行的，概率 0.5；2)下一条指令均匀分布在前地址部分，概率 0.25; 3）下一条指令均匀分布在后地址部分，概率 0.25;

   执行下面算法直到序列长度达到 320：

   随机选取 p∈【0，319】;
   iaddress[0]=← p;
   for (i=1;i<320;i++)
   {
   p ← iaddress[i-1];
   生成随机数 r，0<r<1.0；
   if (r<0.5) iaddress[i] ←p;
   else if (r<0.75) iaddress [i] ←随机选取 p'∈【0，p-1】;
   else iaddress [i] ←随机选取 p'∈【p+1,319】;
   }

   b) 将指令序列转换成页面序列：每个页面包含 10 个指令。

   for (i=0;i<320;i++)
   page[i]= iaddress[i] / 10;

   随机数生成函数 rand(); 初始化 srand(int seed);

2. 测试不同物理内存帧数（4--32）情况下 FIFO, LRU, OPT 的命中率 (= 1− 缺页次数/页面序列长度)。

```
for(frame=4; rame<=32;frame++)
{
FIFO(frame);//计算并输出 FIFO 算法命中率
LRU(frame);//计算并输出 LRU 算法命中率
OPT(frame); //计算并输出 OPT 算法命中率
}
```

实验 3　文件系统

目的：

通过一个简化的多用户文件系统模型的设计，进一步理解 unix 文件系统的结构及其功能的实现。

内容：

在 LINUX 环境下设计一个简单的二级文件系统，给出程序运行过程和结果。要求做到以下几点：

1、可以实现下列几条命令（至少 4 条）(不必写命令行解释程序，直接在 main 函数中调用命令的实现函数。参考后面的源码。)

Login	用户登录
Dir	列文件目录
Create	创建文件
Delete	删除文件
Open	打开文件
Close	关闭文件
Read	读文件
Write	写文件

2、列目录时要列出文件名、物理地址、保护码和文件长度

3、源文件可以进行读写保护

准备

1、首先应确定文件系统的数据结构：主目录、子目录及活动文件等。主目录和子目录都以文件的形式存放于磁盘，这样便于查找和修改；

2、用户创建的文件，可以编号存储于磁盘上。如 file0,file1,file2……并以编号作为物理地址，在目录中进行登记。

实验指导

一、文件管理

要将文件存储在磁盘（带）上，必须为之分配相应的存储空间，这就涉及到对文件存储空间的管理；采取何种方式存储，又涉及到文件的物理结构；为了简化对文件的访问和共享，还应设置相应的用户文件描述表及文件表。

1、文件存储空间的管理

（1）文件卷的组织

UNIX 中，把每个磁盘（带）看作是一个文件卷，每个文件卷上可存放一个具有独立目录结构的文件系统。一个文件卷包含许多物理块，并按块号排列如下图：

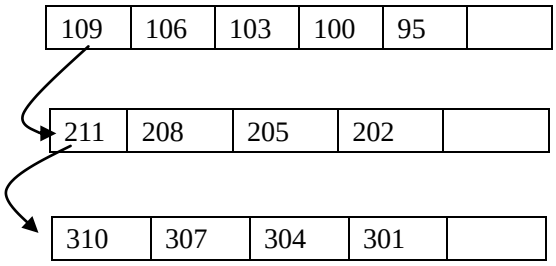
0#	1#	2#	3# ……K#	K+1#	……N#

其中，0#块用于系统引导或空闲，1#为超级块(superblock)，存放文件卷的资源管理信息，如整个文件卷的盘块数、磁盘索引结点的盘块数、空闲盘块号栈及指针等。2#~K#存放磁盘索引结点。每个索引结点 64B，第 K+1#~N#存放文件数据。

（2）空闲盘块的组织

UNIX 采用成组链接法组织空闲盘块。它将若干个空闲盘块划归一个组，将每组中所有盘块号存放在其前一组的第一个空闲盘块中，而第一组中所有空闲盘块号放入超级块的空闲盘块号栈中。

例：　　超级块表



（3）空闲盘块的分配与回收

内核要从文件系统中分配一盘块时，先检查超级块空闲盘块号栈是否已上锁。是则调用 sleep 睡眠，否则将超级块中空闲盘块栈栈顶盘块号分配出去。

回收时，若空闲盘块号栈未满，直接将回收盘块编号记入空闲盘块号栈中。若回收时栈已满，须先将栈中的所有空闲盘块号复制到新回收的盘块中，再将新回收盘块的编号作为新栈的栈底块号进栈。

2、文件的物理结构

UNIX 未采用传统的三种文件结构形式，而是将文件所占用盘块的盘块号，直接或间接地存放在该文件索引结点的地址项中。查找文件时，只需找到该文件的索引结点，便可直接或间接的寻址方式获得指定文件的盘块。

过程 bmap 可将逻辑文件的字节偏移量转换为文件的物理块号。先将字节偏移量转换为文件逻辑块号及块内偏移量，再把逻辑块号转换为文件的物理块号。

3、用户文件描述符表和文件表的管理

每个进程的 U 区中设置一张用户文件描述符表。只在首次打开文件时才需给出路径名。内核在该进程的用户文件描述符表中，分配一空项，取其在该表中的位移量作为文件描述符 fd(file discriptor)返还给用户。当用户再次访问该文件时，只需提供 fd，系统根据 fd 便可找到相应文件的内存索引结点。fd 表项的分配由 ufalloc 完成。

为了方便用户对文件进行读/写及共享，系统中设置了一张文件表。每个用户在打开文件时，都要在文件表中获得一表项，其中包含下述内容：

f.flag：文件标志，指示该文件打开是为了读或写；

f.inode：指向打开文件的内存索引结点指针；

f.offset：文件读写指针偏移值；

f.count：文件引用计数。

二、目录管理

UNIX 中，为了加速对文件目录的查找，将文件名和文件说明分开，由文件说明形成一个称为索引结点的数据结构，而相应的文件目录项则只由文件符号名和指向索引结点的指针构成。

对目录的管理应包括的功能有：

1、对索引结点的管理

每个文件都有一唯一的磁盘索引结点(di_node)。文件被打开后，还有一个内存索引结点(i_node)。创建一新文件时，就为之建立一个磁盘索引结点，以将文件的有关信息记入其中，并将用户提供的文件名和磁盘索引结点号一并组成一个新目录项，记入其父目录文件中。文件被撤消时，系统要回收该文件的磁盘索引结点，从其父目录中删除该目录项。随着文件的打开与关闭，系统还要为之分配和回收内存索引结点。

（1）磁盘索引结点

磁盘索引结点中，包含有关文件的下述一系列信息：

文件模式 di_mode。可以是正规文件、目录文件、字符特别文件、块特别文件和管道文件等几种；

文件所有者用户标识符 di_uid。指拥有该文件的用户标识符；

同组用户标识符 di_gid。与拥有该文件的用户在同一小组的用户标识符；

文件长度 di_size。以字节计数的文件大小；

文件的联接计数 di_nlink。表明在本文件系统中所有指向该文件的文件名计数；

文件的物理地址 di_addr。di_addr 地址项共有 13 项，即 di_addr(0) 到 di_addr(12)，每个地址项占 3 字节；

文件的访问时间 di_atime。指文件最近被进程访问的时间；

文件的修改时间 di_mtime。指文件和索引结点最近被进程修改的时间；

文件的建立时间 di_citime。

（2）内存索引结点

文件被打开后，系统为它在内存索引结点表区中建一内存索引结点，以方便用户和系统对文件的访问。其中，一部分信息是直接从磁盘索引结点拷贝过来的，如 i_mode、i_uid、i_gid、i_size、i_addr、i_nlink 等，并又增加了如下各信息：

索引结点编号 i_number。作为内存索引结点的标识符；

状态 i_flag。指示内存索引结点是否已上锁、是否有进程等待此 i 结点解锁、i 结点是否被修改、是否有最近被访问等标志；

引用计数 i_count。记录当前有几个进程正在访问此 i 结点，每当有进程访问此 i 结点时，对 i_count+1，退出-1；

设备号 i_dev。文件所属文 S 件系统的逻辑设备号；

前向指针 i_forw。Hash 队列的前向指针；

后向指针 i_back。Hash 队列的后向指针；

（3）磁盘索引结点的分配与回收

分配过程 ialloc：当内核创建一新文件时，要为之分配一空闲磁盘 i 结点。如分配成功，便再分配一内存 i 结点。其过程如下：

检查超级块上锁否。由于超级块是临界资源，诸进程必须互斥地访问它，故在进入 ialloc 后，要先检查它是否已上锁，若是则睡眠等待；

检查 i 结点栈空否。若 i 结点栈中已无空闲结点编号，则应从盘中再调入一批 i 结点号进栈。若盘中已无空闲 i 结点，则出错处理，返回；

从空闲 i 结点编号栈中分配一 i 结点，并对它初始化、填写有关文件的属性；

分配内存 i 结点；

将磁盘 i 结点总数-1，置超级块修改标志，返回。

回收过程 ifree：

当删除文件时，应回收其所占用的盘块及相应的磁盘 i 结点。具体有：

检查超级块上锁否。若是，直接返回，即不把本次回收的 i 结点号记入空闲 i 结点编号

栈中；

检查 i 结点编号栈满否。若已满，无法再装入新回收的 i 结点号，立即返回，若未满，便将回收的 i 结点编号进栈，并使当前空闲结点数+1；

置超级块修改标志，返回。

（4）内存索引结点的分配与回收

分配过程 iget：

虽然 iget 用在打开文件时为之分配 i 结点，但由于允许文件被共享，因此，如果一文件已被其他用户打开并有了内存 i 结点，则此时只需将 i 结点中的引用计数+1。如果文件尚未被任何用户（进程）打开，则由 iget 过程为该文件分配一内存 i 结点，并调用 bread 过程将其磁盘 i 结点的内容拷贝到内存 i 结点中并进行初始化。

回收过程 iput：

进程要关闭某文件时，须调用 iput 过程，先对该文件内存 i 结点中的引用计数-1。若结果为 0，便回收该内存 i 结点，再对该文件的磁盘 i 结点中的连接计数减 1，若其结果也为 0，便删除此文件，并回收分配给该文件的盘块和磁盘 i 结点。

2、构造目录 make_node

文件系统的一个基本功能是实现按名存取，它通过文件目录来实现。为此须使每一个文件都在文件目录中有一个目录项，通过查找文件目录可找到该文件的目录项和它的索引结点，进而找到文件的物理位置。对于可供多个用户共享的文件，则可能有多个目录项。如果要将文件删除，其目录项也应删除。

构造目录先调用 ialloc 为新建文件分配一磁盘 i 结点及内存 i 结点。若分配失败则返回，分配成功时须先设置内存 i 结点的初值（含拷贝），调用写目录过程 wdir，将用户提供的文件名与分配给该文件的磁盘 i 结点号一起，构成一新目录项，再将它记入其父目录文件中。

3、检索目录 namei

用户在第一次访问某文件时，需要使用文件的路径名，系统按路径名去检索文件目录，得到该文件的磁盘索引结点，且返回给用户一个 fd。以后用户便可利用该 fd 来访问文件，这时系统不需再去检索文件目录。

namei 根据用户给出的路径名，从高层到低层顺序地查找各级目录，寻找指定文件的索引结点号。检索时，对以'/'开头的路径名，须从根目录开始检索，否则，从当前目录开始，并把与之对应的 i 结点作为工作索引结点，然后用文件路径名中的第一分量名与根或与当前目录文件中的各目录项的文件名，逐一进行比较。由于一个目录文件可能占用多个盘块，在检索完一个盘块中所有目录项而未找到匹配的文件分量名时，须调用 bmap 和 bread 过程，将下一个盘块中的所有目录项读出后，再逐一检索。若检索完该目录文件的所有盘块而仍未找到，才认为无此文件分量名。

检索方式采用 Hash 方法。

三、主要文件操作的处理过程

1、打开文件 open

检索目录。内核调用 namei 从根目录或从当前目录，沿目录树查找指定的索引结点。若未找到或该文件不允许存取，则出错处理返回 NULL，否则转入下一步；

分配内存索引结点。如果该文件已被其它用户打开，只需对上一步中所找到的 i 结点引用计数+1，否则应为被打开文件分配一内存 i 结点，并调用磁盘读过程将磁盘 i 结点的内容拷贝到内存 i 结点中，并设置 i.count=1；

分配文件表项。为已打开的文件分配一文件表项，使表项中的 f.inode 指向内存索引结点；

分配用户文件描述表项。

2、创建文件 creat

核心调用 namei，从根目录或当前目录开始，逐级向下查找指定的索引结点。此时有以下二种情况：

重写文件。namei 找到了指定 i 结点，调用 free 释放原有文件的磁盘块。此时内核忽略用户指定的许可权方式和所有者，而保持原有文件的存取权限方式和文件主。最后打开。

新建。namei 未找到。调用 ialloc，为新创建的文件分配一磁盘索引结点，并将新文件名及所分配到的 i 结点编号，写入其父目录中，建立一新目录项。利用与 open 相同的方式，把新文件打开。

3、关闭文件 close

根据用户文件描述符 fd，从相应的用户文件描述符表项中，获得指向文件表项的指针 fp，再对该文件表项中的 f.count-1。

四、主要数据结构

1、i 节点

```
 struct   inode
 {   struct   inode   *i_forw;
     struct   inode   *i_back;
     char   i_flag;
     unsigned   int   i_ino;                    /*磁盘 i 节点标号*/
   unsigned   int   i_count;                  /*引用计数*/
     unsigned   short   di_number;            /*关联文件数，当为 0 时，则删除该文件*/
     unsigned   short   di_mode;              /*存取权限*/
     unsigned   short   di_uid;               /*磁盘 i 节点用户 id*/
     unsigned   short   di_gid;               /*磁盘 i 节点组 id*/
     unsigned   int   di_addr[NADDR];       /*物理块号*/
```

2、磁盘 i 节点

```
     struct dinode
     {    unsigned   short   di_number;          /*关联文件数*/
unsigned   short   di_mode;             /*存取权限*/
unsigned   short   di_uid
unsigned   short   di_gid;
unsigned   long    di_size;             /*文件大小*/
unsigned   int     di_addr[NADDR];   /*物理块号*/
}
```

3、目录项结构

```
struct direct
{    char   d_name[DIRSIZ];             /*目录名*/
unsigned   int   d_ino;                /*目录号*/
}
```

4、超级块

```
struct filsys
{    unsigned short   s_isize;                  /*i 节点块块数*/
unsigned long   s_fsize;                   /*数据块块数*/
```

```
unsigned int    s_nfree;                    /*空闲块块数*/
unsigned short    s_pfree;                  /*空闲块指针*/
unsigned int    s_free[NICFREE];            /*空闲块堆栈*/
unsigned int    s_ninode;                   /*空闲 i 节点数*/
unsigned short    s_pinode;                 /*空闲 i 节点指针*/
unsigned int    s_inode[NICINOD];           /*空闲 i 节点数组*/
unsigned int    s_rinode;                   /*铭记 i 节点*/
char    s_fmod;                             /*超级块修改标记*/
}
```

5、用户密码

```
struct    pwd
{     unsigned    short    p_uid;
unsigned    short    p_gid;
char    password[PWOSIZ];
};
```

6、目录

```
struct dir
{     struct    direct    direct [DIRNUM];
int    size;
};
```

7、查找内存 i 节点的 hash 表

```
struct hinode
{     struct    inode    *i_forw;
};
```

8、系统打开表

```
struct file
{     char    f_flag;                       /*文件操作标志*/
unsigned    int    f_count;                 /*引用计数*/
struct    inode    *f_inode;                /*指向内存 i 节点*/
unsigned    long    f_off;                  /*读/写指针*/
};
```

9、用户打开表

```
struct user
{         unsigned short u_default_mode;
unsigned short u_uid;                       /*用户标志*/
unsigned short u_gid;                       /*用户组标志*/
unsigned short u_ofile[NOFILE];             /*用户打开表*/
};
```

三、主要函数

1、i 节点内容获取函数 iget( )

2、i 节点内容释放函数 iput( )

3、目录创建函数 mkdir( )

4、目录搜索函数 namei( )

5、磁盘块分配函数 balloc( )

6、磁盘块释放函数 bfree( )
7、分配 i 节点区函数 ialloc( )
8、释放 i 节点区函数 ifree( )
9、搜索当前目录下文件的函数 iname( )
10、访问控制函数 access( )
11、显示目录和文件用函数_dir( )
12、改变当前目录用函数 chdir( )
13、打开文件函数 open( )
14、创建文件函数 create( )
15、读文件用函数 read( )
16、写文件用函数 write( )
17、用户登录函数 login( )
18、用户退出函数 logout( )
19、文件系统格式化函数 format( )
20、进入文件系统函数 install( )
21、关闭文件系统函数 close( )
22、退出文件系统函数 halt( )
23、文件删除函数 delete( )
四、主程序说明
begin
step1 对磁盘进行格式化
step2 调用 install( ),进入文件系统
step3 调用_dir( ),显示当前目录
step4 调用 login( ),用户注册
step5 调用 mkdir( )和 chdir( )创建目录
step6 调用 creat( )，创建文件 0
step7 分配缓冲区
step8 写文件 0
step9 关闭文件 0 和释放缓冲
step10 调用 mkdir( )和 chdir( )创建子目录
step11 调用 creat( ),创建文件 1
step12 分配缓冲区
step13 写文件 1
step14 关闭文件 1 和释放缓冲
step15 调用 chdir 将当前目录移到上一级
step16 调用 creat( ) ，创建文件 2
step17 分配缓冲区
step18 调用 write( ),写文件 2
step19 关闭文件 2 和释放缓冲
step20 调用 delete( ) ,删除文件 0
step21 调用 creat( ) ，创建文件 3
step22 为文件 3 分配缓冲区
step23 调用 write( ),写文件 3
step24 关闭文件 3 和释放缓冲

step25　　调用 open( ),打开文件 2
step26　　为文件 2 分配缓冲区
step27　　调用 open( ),打开文件 2
step28　　释放缓冲
step29　　用户退出(logout)
step30　　关闭(halt)
end

由上述描述过程可知，该文件系统实际是为用户提供一个解释执行相关命令的环境。主程序中的大部分语句都被用来执行相应的命令。

下面，我们给出每个过程的相关 C 语言程序。读者也可以使用这些子过程，编写出一个用 shell 控制的文件系统界面。

II、 μc/os II 实验

实验一 熟悉软硬件环境及μC/OS-II 移植

1 实验目的

1）掌握ADS1.2软件及DNW工具的使用
2）掌握移植的概念、方法和步骤
3）掌握ARM 体系结构
4）掌握μ C/OS-II 软件和硬件体系结构
5）学习ARM 指令集和ARM 处理器硬件特征
6）学习μ C/OS-II 移植条件和在 ARM 上的移植方法

2 实验要求

1）建立移植概念。
2）学习μ C/OS-II 在其他处理器上的移植代码和ARM 指令。
3）理解移植步骤内容。
4）阅读μ C/OS-II 源代码和移植文档，阅读ARM Architecture Reference Manual 和开发板硬件手册。
5）将μ C/OS-II 移植到ARM 处理器上。并且编写一个简单的测试代码检验移植效果。

3 准备事项

1）用ARM ADS1.2 集成开发环境，编写和调试程序的基本过程。
2）ARM 指令编程。
3）μ C/OS-II 和处理器相关的代码的运行流程。
4）硬件：ARM 嵌入式开发板、PC 机Pentumn100 以上、串口线
5）软件：PC 机操作系统 win98 、Win2000 或 WinXP 、ARM ADS1.2 集成开发环境、仿真器驱动程序、串口通讯程序、实验原理及说明

4 实验原理

4.1 ARM 体系结构

ARM（Advanced RISC Machines）是目前在嵌入式领域里应用最广泛的RISC 微

处理器结构，以其低成本、低功耗、高性能的特点占据了嵌入式系统应用领域的领先地位。ARM系列的处理器当前有ARM7、ARM9、ARM9E、ARM10等多个产品，此外ARM公司合作伙伴，例如Intel 也提供基于XScale 微体系结构的相关处理器产品。所有的ARM 处理器都共享ARM通用的基础体系结构，所以开发者在不同的ARM处理器上做操作系统移植时，可以将节省相当多的工作量，这无疑将大大降低软件开发成本。ARM 完整的体系结构可参考ARM Architecture Reference Manual。下面是几个和移植工作有关的概念。

1． 处理器模式： **(cpu mode)**

ARM 的处理器有 7 种工作模式，如图 1-1 所示：

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks (ARM architecture version 4 and above)

图 1-1 ARM processor modes

这里除 usr 模式以外的其他模式都叫做特权模式，除 usr 和 sys 外的其他5 种模式叫做异常模式。在 usr 模式下对系统资源的访问是受限制的，也无法主动地改变处理器模式。异常模式通常都是和硬件相关的，例如中断或者是试图执行未定义指令等。这里需要强调的是和移植相关的两种处理器模式：svc 态和irq 态，分别指操作系统的保护模式和通用中断处理模式。这两种模式之间的转换可以通过硬件的方式，也可以通过软件的方式。uC/OS-II 内核在执行过程中，大部分时间都是工作在 svc 态，当有硬件中断，例如时钟中断到来时，cpu 硬件上会自动完成从svc 态进入 irq态，在中断处理程序的结束处，则需要通过编程的方法使得 cpu 从irq 态恢复到 svc 态。

. 2 ARM 寄存器：(register)

ARM 处理器一共有37 个寄存器，其中31 个是通用寄存器，包括一个程序计数器 PC。另外6个就是上面提到的程序状态寄存器。

1．通用寄存器：

(1) R0－R7：与所有处理器模式无关的寄存器，可以用作任何用途。

(2) R8－R14：与处理器模式有关的寄存器，在不同的模式下，对应到不同的物理寄存器。其中 R13又叫做 sp，一般用于堆栈指针。R14又叫做lr，一般用于保存返回地址。这两个寄存器在每种异常模式下都对应到不同的物理寄存器上，例如lr_irq、lr_svc、lr_fiq 等。

(3) R15：又叫做程序计数器，即pc，所有的模式下都使用同一个 pc。

2．状态寄存器：

(1) CPSR：当前程序状态寄存器，所有的模式下都使用同一个 CPSR。

(2) SPSR：保存的程序状态寄存器，每种异常模式下都有自己的SPSR，一共有5种SPSR，即 SPSR_irq、SPSR_fiq、SPSR_svc、SPSR_abt、SPSR_und。usr 和sys 态下没有 SPSR 。

所有的ARM寄存器的命名和含义，可参照图1-2 来说明，其中相同命名的都是同一个物理寄存器，不同命名的寄存器都对应不同的物理寄存器。

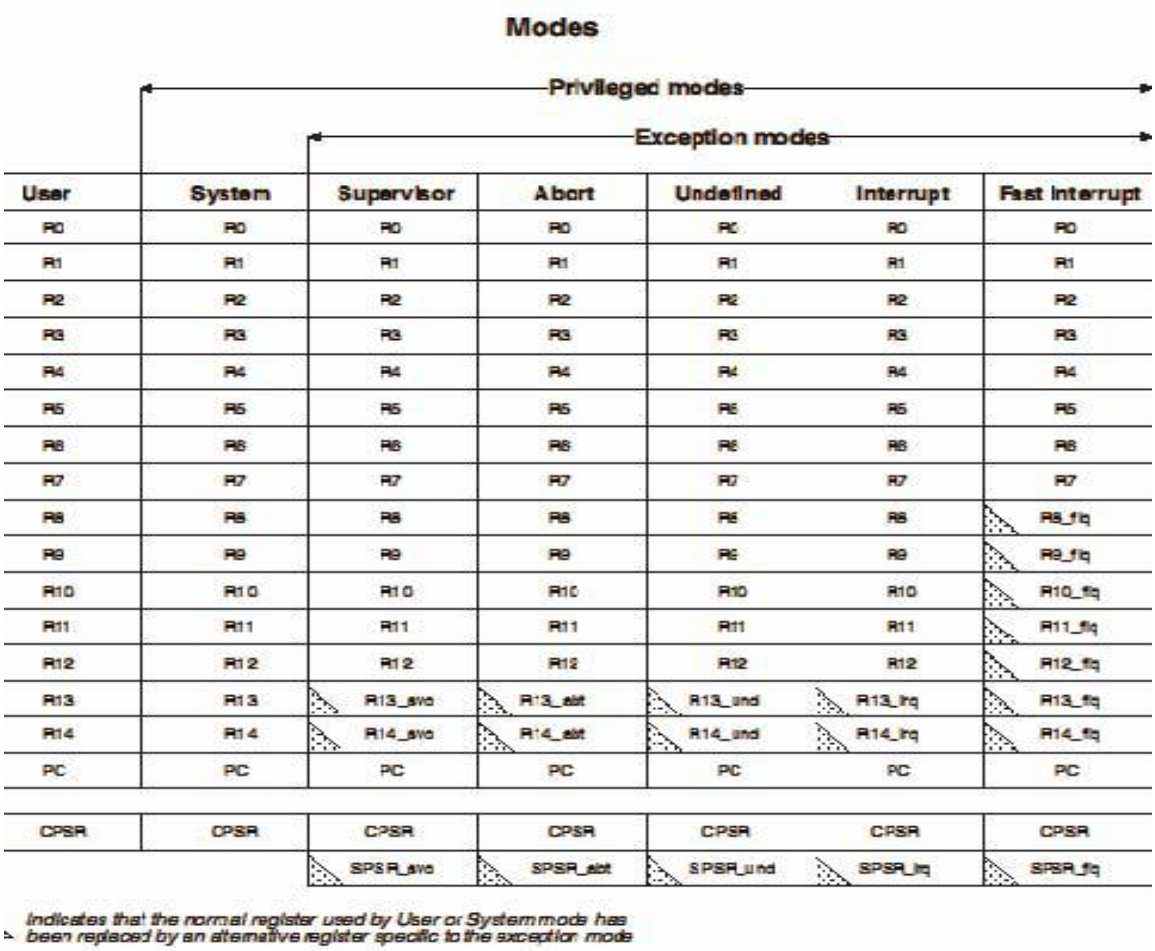


图 1-2 ARM 寄存器命名和含义

4.2 μC/OS-II 的移植条件

要使μC/OS-II 正常工作，处理器必须满足如下要求：

（1） 处理器的C编译器能产生可重入代码。

（2） 在程序中可以打开或者关闭中断

（3） 处理器支持中断，并能产生定时中断（通常为10－100Hz）

（4） 处理器支持能够容纳一定量数据的硬件堆栈

（5） 处理器有将堆栈指针和其他CPU寄存器存储和读出到堆栈（或者内存）的指令。

μC/OS-II进行任务调度的时候，会把当前任务的CPU寄存器存放到此任务的堆栈中，然后再从另一个任务的堆栈中恢复原来的工作寄存器，继续运行这个任务。所以，寄存器的入栈和出栈是μC/OS-II多任务调度的基础。

ARM7TDMI 处理器完全满足上述要求。

4.3 μC/OS-II 软件和硬件体系结构图

图 1-3 示意了μ C/OS-II 的体系结构以及它与系统硬件之间的关系。

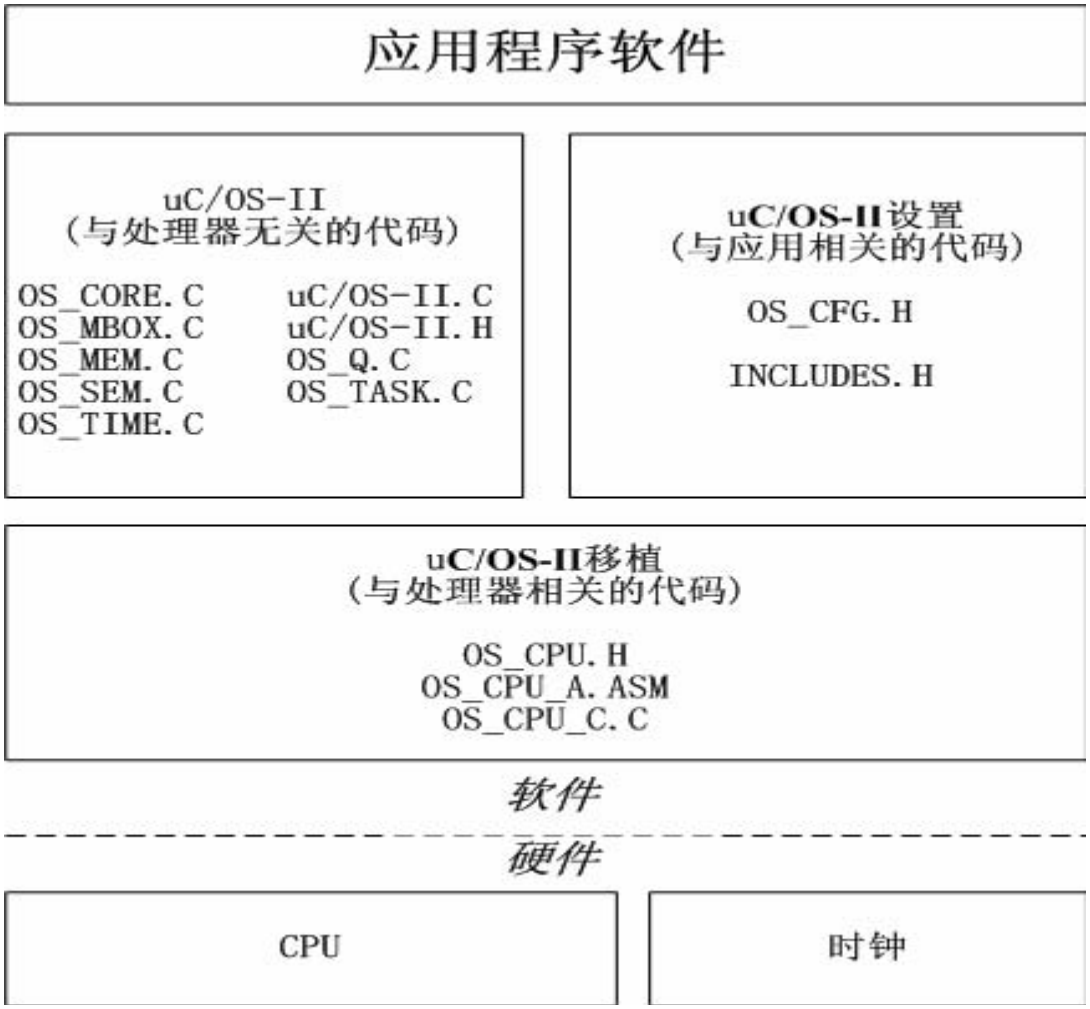


图 1-3 μ C/OS-II 硬件/软件体系结构

4.5 移植原理

移植工作的绝大部分都集中在多任务切换的实现上，因为这部分代码主要是用来保存和恢复处理器现场（即相关寄存器），因此不能用C语言，只能使用特定的处理器汇编语言完成。

uC/OS-II 的全部源代码量大约是6000－7000 行，一共有15 个文件。将uC/OS-II移植到ARM处理器上，需要修改三个和ARM体系结构相关的文件，代码量大约是500行。 OS_CPU.H 文件

1．与编译器相关的数据类型

不同的编译器有不同的字长。比如int，同样在x86平台上，如果用GNU的gcc 编译器，则编译为4 bytes，而使用MS VC＋＋则编译为2 bytes。

uC/OS-II 的移植需要定义一系列的类型定义以确保其可移植性。尤其是 uC/OS-II 代码不使用 C 的 short、int 和 long 等数据类型。相关的数据类型定义如图 1.4 所示：

```
typedef unsigned char  BOOLEAN;

typedef unsigned char  INT8U;                    /* Unsigned  8 bit quantity
typedef signed   char  INT8S;                    /* Signed    8 bit quantity

typedef unsigned short   INT16U;                   /* Unsigned 16 bit quantity
typedef signed   short   INT16S;                   /* Signed   16 bit quantity

typedef unsigned long  INT32U;                   /* Unsigned 32 bit quantity
typedef signed   long  INT32S;                   /* Signed   32 bit quantity
typedef float          FP32;                     /* Single precision floating point
typedef double         FP64;                     /* Double precision floating point
```

图 1-4 数据类型定义

2．堆栈单位

处理器现场的寄存器在任务切换时都将会保存在当前运行任务的堆栈中，所以OS_STK数据类型应该是和处理器的寄存器长度一致的。

```
typedef unsigned int   OS_STK;                   /* Each stack entry is 32-bit wide
```

3．堆栈增长方向

堆栈由高地址向低地址增长，这个也是和编译器有关的，当进行函数调用时，入口参数和返回地址一般都会保存在当前任务的堆栈中，编译器的编译选项和由此生成的堆栈指令就会决定堆栈的增长方向。

```
/* stack stuff */
#define OS_STK_GROWTH    1
```

4．宏定义

包括开关中断的宏定义，以及进行任务切换的宏定义。

```
#define OS_ENTER_CRITICAL()      ARMDisableInt()
#define OS_EXIT_CRITICAL()       ARMEnableInt()
#define OS_TASK_SW()             OSCtxSw()
```

OS_CPU_C.C 文件

(1) 任务堆栈初始化在ARM体系结构下，任务堆栈空间由高至低依次将保存着pc、lr、r12、r11、r10、… r1、r0、CPSR、SPSR。（图1-5）

当前任务堆栈初始化完成后，OSTaskStkInit 返回新的堆栈指针stk，在OSTaskCreate（）执行时将会调用OSTaskStkInit 的初始化过程，然后通过OSTCBInit（）函数调用将返回的sp指针保存到该任务的TCB块中。

初始化后的堆栈模拟了一次中断发生后的堆栈结构。任务被创建后并不是直接就获得执行的，而是通过 OSSched（）函数进行调度分配，满足执行条件后才能获得执行的。为了使这个调度简单一致，就预先将该任务的 pc 指针和返回地址 lr 都指向函数入口，以便被调度时从堆栈中恢复刚开始运行时的处理器现场。

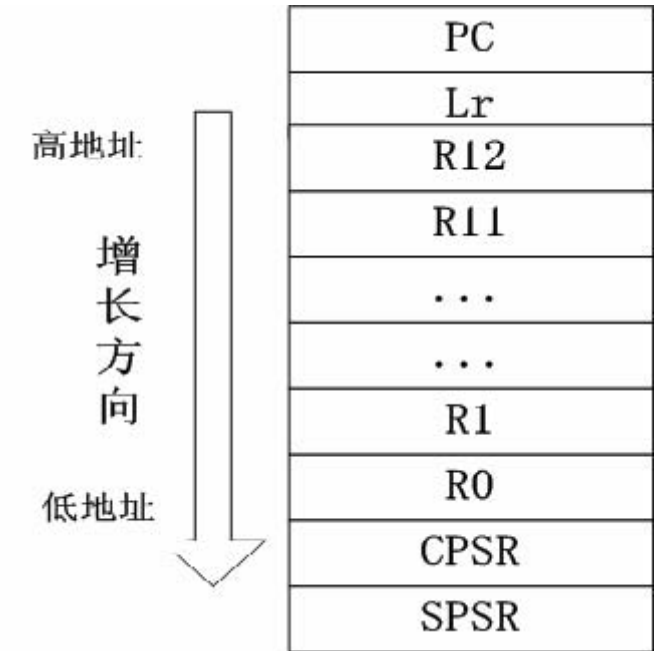


图 1-5 任务堆栈空间

(2) 系统hook 函数需要实现几个操作系统规定的hook函数，如下：

OSSTaskCreateHook( )OSTaskDelHook( )OSTaskSwHook( )

OSTaskStatHook( )

OSTimeTickHook( ) 可以将其设为空函数。

OS_CPU_A.S 文件

(1) OSStartHighRdy（）

此函数是在 OSStart（）多任务启动之后，负责从最高优先级任务的 TCB 控制块中获得该任务的堆栈指针 sp，通过 sp 依次将 cpu 现场恢复，这时系统就将控制权交给用户创建的该任务进程，直到该任务被阻塞或者被其他更高优先级的任务抢占 cpu。该函数仅仅在多任务启动时被执行一次，用来启动第一个，也就是最高优先级的任务执行，之后多任务的调度和切换就是由下面的函数来实现。

```
53 ;*********************************************************************/
54 OSStartHighRdy
55         BL      OSTaskSwHook              ; Call user-defined hook function
56
57         LDR     r4, =OSRunning             ; Indicate that multitasking has started
58         MOV     r5, #1
59         STRB    r5, [r4]                  ; OSRunning = true
60
61         LDR     r4, =OSTCBHighRdy         ; Get highest priority task TCB address
62         LDR     r4, [r4]                  ; get stack pointer
63         LDR     sp, [r4]                  ; switch to the new stack
64
65         LDMFD   sp!, {r4}                 ; pop new task s spsr
66         MSR     spsr_c, r4
67         LDMFD   sp!, {r4}                 ; pop new task s psr
68         MSR     cpsr_c, r4
69         LDMFD   sp!, {r0-r12,lr,pc}       ; pop new task s r0-r12,lr & pc
70
```

(2) OSCtxSw（）

任务级的上下文切换，它是当任务因为被阻塞而主动请求 cpu 调度时被执行，由于此时的任务切换都是在非异常模式下进行的，因此区别于中断级别的任务切换。它的工作是先将当前任务的 cpu 现场保存到该任务堆栈中，然后获得最高优先级任务的堆栈指针，从该堆栈中恢复此任务的 cpu 现场，使之继续执行。这样就完成了一次任务切换。

```
97 ;*********************************************************************/
98 OSCtxSw
99          STMFD   sp!, {lr}                ; push pc (lr is actually be pushed in place of PC)
100         STMFD   sp!, {r0-r12,lr}         ; push lr & register file
101         MRS     r4, cpsr                 ; copy CPSR to R4
102         STMFD   sp!, {r4}                ; push current psr
103         MRS     r4, spsr                 ; copy SPSR to R4
104         STMFD   sp!, {r4}                ; push current spsr
105
106 OSCtxSw
107         LDR     r4, =OSPrioCur           ; OSPrioCur = OSPrioHighRdy
108         LDR     r5, =OSPrioHighRdy
109         LDRB    r6, [r5]
110         STRB    r6, [r4]
111
112         LDR     r4, =OSTCBCur            ; Get current task TCB address
113         LDR     r5, [r4]
114         STR     sp, [r5]                 ; store sp in preempted tasks's TCB
115
116         BL      OSTaskSwHook             ; call Task Switch Hook
117
118         LDR     r6, =OSTCBHighRdy        ; Get highest priority task TCB address
119         LDR     r6, [r6]
120         LDR     sp, [r6]                 ; get new task s stack pointer
121
122         STR     r6, [r4]                 ; set new current task TCB address
123
124         LDMFD   sp!, {r4}                ; pop new task spsr
125         MSR     spsr_c, r4
126         LDMFD   sp!, {r4}                ; pop new task cpsr
127         MSR     cpsr_c, r4
128         LDMFD   sp!, {r0-r12,lr,pc}      ; pop new task r0-r12,lr & pc
129
```

(3) OSIntCtxSw（）

中断级的任务切换，它是在时钟中断 ISR（中断服务例程）中发现有高优先级任务等待的时钟信号到来，则需要在中断退出后并不返回被中断任务，而是直接调度就绪的高优先级任务执行。这样做的目的主要是能够尽快地让高优先级的任务得到响应，保证系统的实时性能。它的原理基本上与任务级的切换相同，但是由于进入中断时已经保存过了被中断任务的 cpu 现场，因此这里就不用再进行类似的操作，只需要对堆栈指针做相应的调整，原因是函数的嵌套。

```
153 ;
154 ;*********************************************************************/
155 OSIntCtxSw
156         LDR     r0, =OSIntCtxSwFlag      ; OSIntCtxSwFlag = true
157         MOV     r1, #1
158         STR     r1, [r0]
159         MOV     pc, lr                   ;This is only change flag,return to OSIntExit
160
161 ;/*********************************************************************
162 ;
```

(4) OSTickISR（）

时钟中断处理函数，它的主要任务是负责处理时钟中断，调用系统实现的OSTimeTick函数，如果有等待时钟信号的高优先级任务，则需要在中断级别上调度其执行。其他相关的两个函数是OSIntEnter（）和OSIntExit（），都需要在ISR中执行。（函数略）

(5) ARMEnableInt（）& ARMDisableInt（）

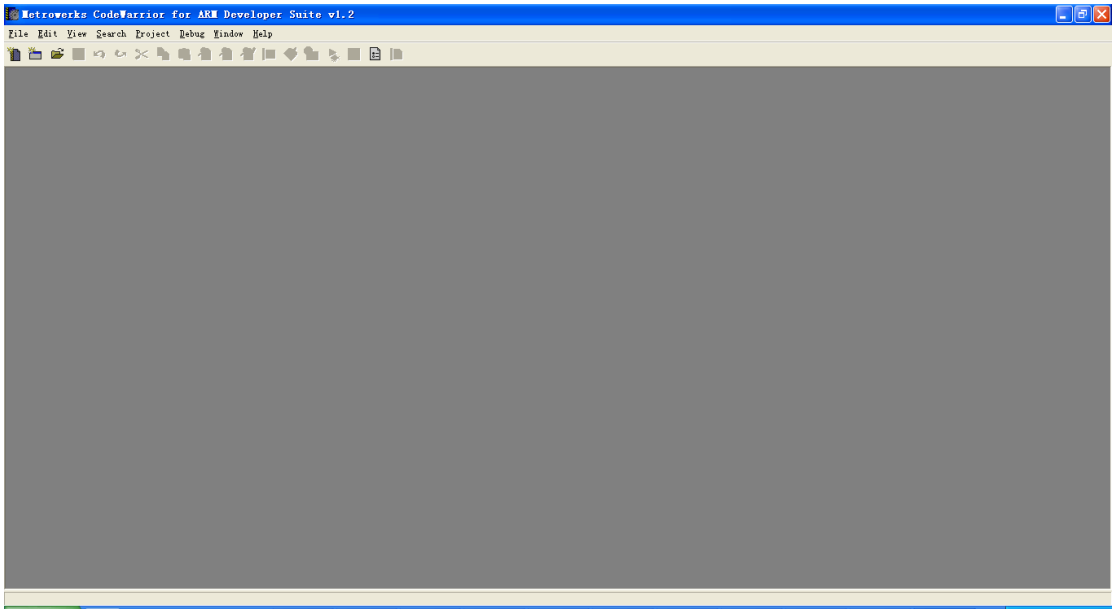
分别是退出临界区和进入临界区的宏指令实现。主要用于在进入临界区之前关闭中断，在退出临界区的时候恢复原来的中断状态。它的实现比较简单，可以采用 a）直接开关中断来实现，也可以采用 b）通过保存关闭/恢复中断屏蔽位来实现。

```
268 ;*****************************************************************;
269 ARMDisableInt
270                 MRS     r0, cpsr
271                 STMFD   sp , {r0}               ; push current PSR
272                 ORR     r0, r0, #0x80
273                 MSR     cpsr_c, r0              ; disable IRQ Int s
274
275                 MOV     pc, lr
276
277
278     ;-----------------------------------------------------------------
279 ARMEnableInt
280                 LDMFD   sp , {r0}               ; pop current PSR
281                 MSR     cpsr_c, r0              ; restore original cpsr
282
283                 MOV     pc, lr
284     ;-----------------------------------------------------------------
```

5 实验内容及步骤

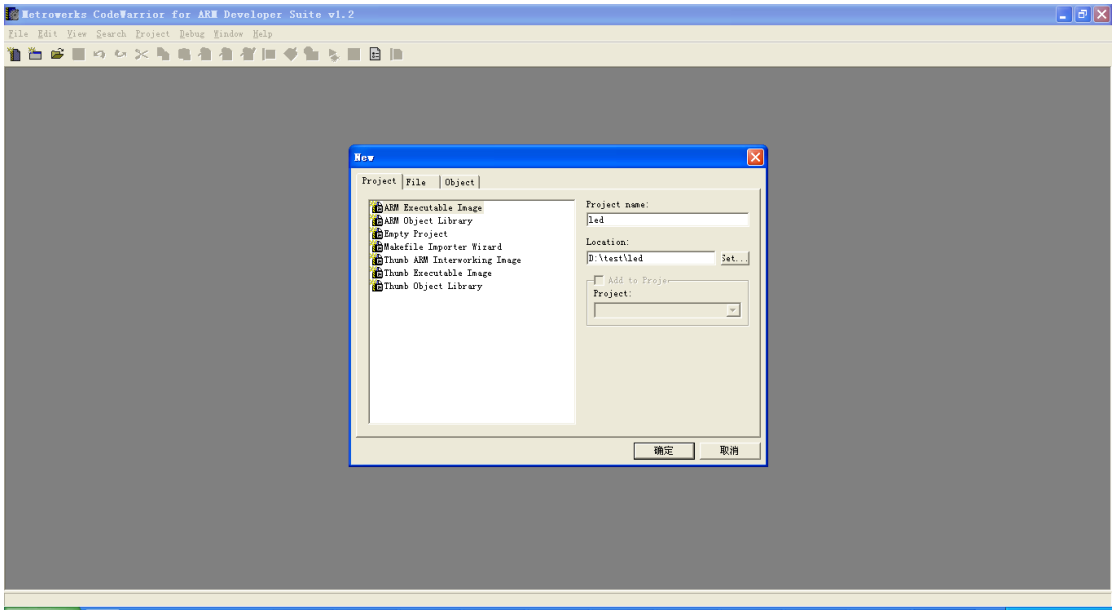
ADS1.2初步使用

1 选择“开始—〉所有程序—〉ARM Developer Suite v1.2” 下的“CodeWarrior for ARM Developer Suite”打开集成开发环境，如图所示。

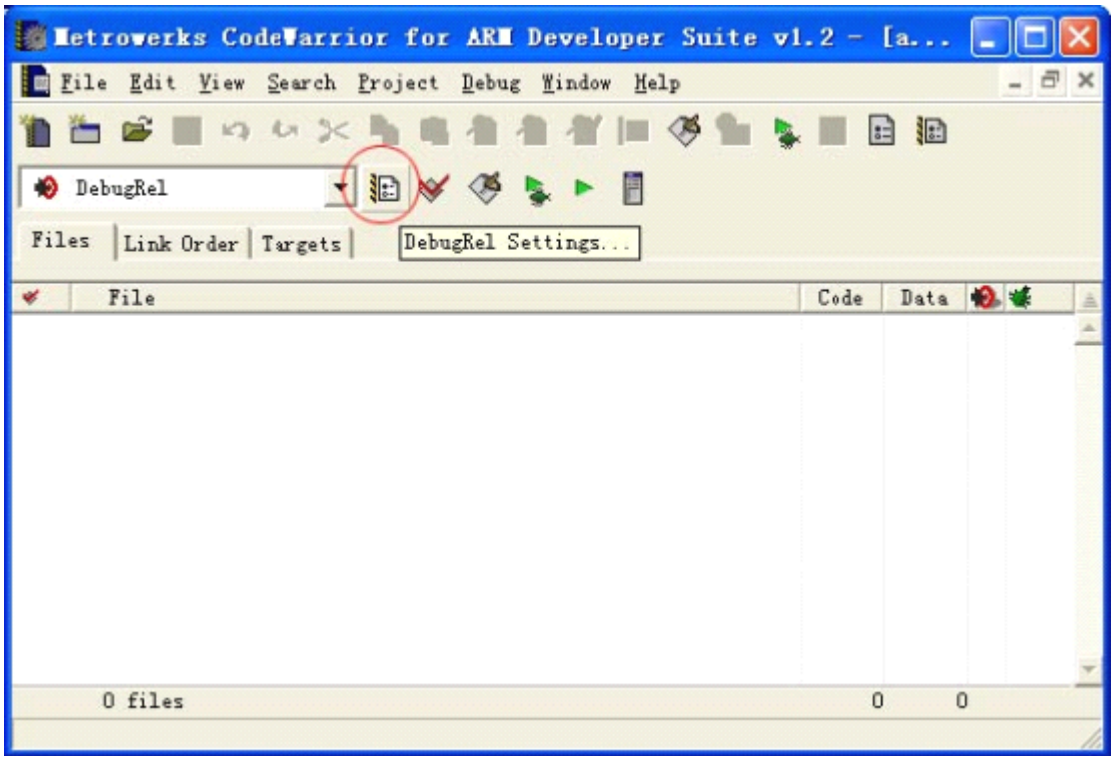


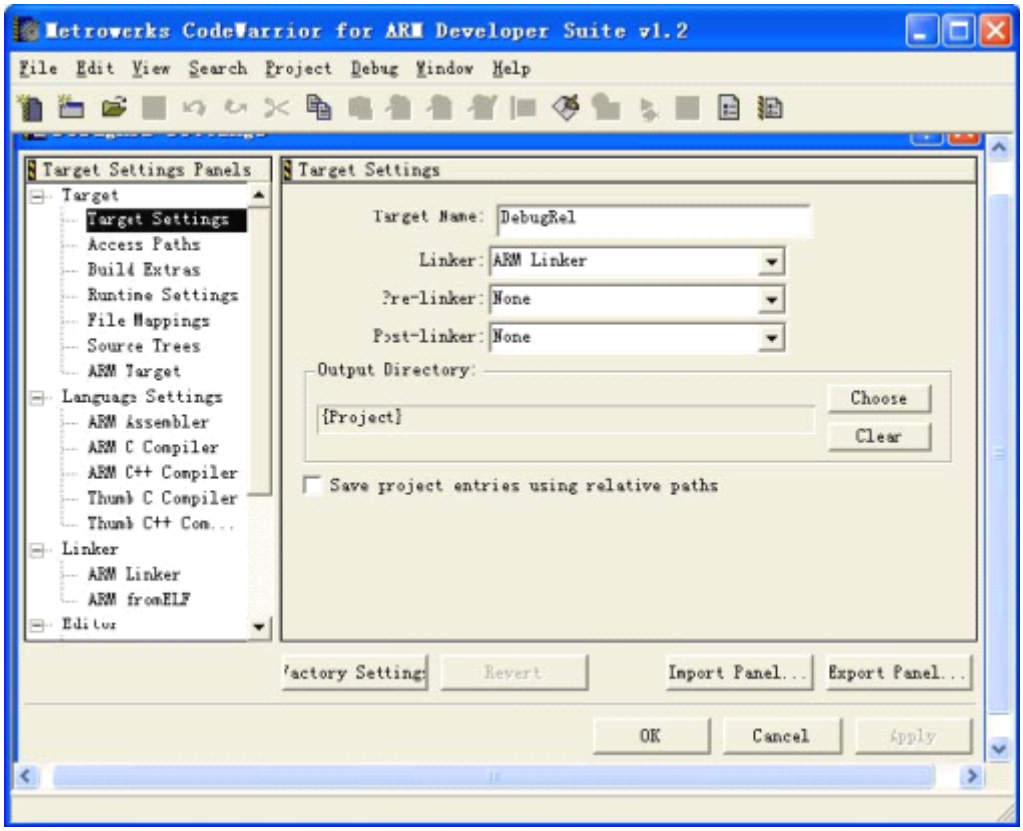
2 新建工程及编译、链接选项设置

（1）单击File 菜单，选择New 菜单项即弹出New 对话框，选择工程模板为ARM 可执行映像（ARM Executable Image），然后在Location 项选择工程存放路径，并在Project name 项输入工程名称，单击“确定”按钮即可建立相应工程，工程文件名后缀为.mcp。

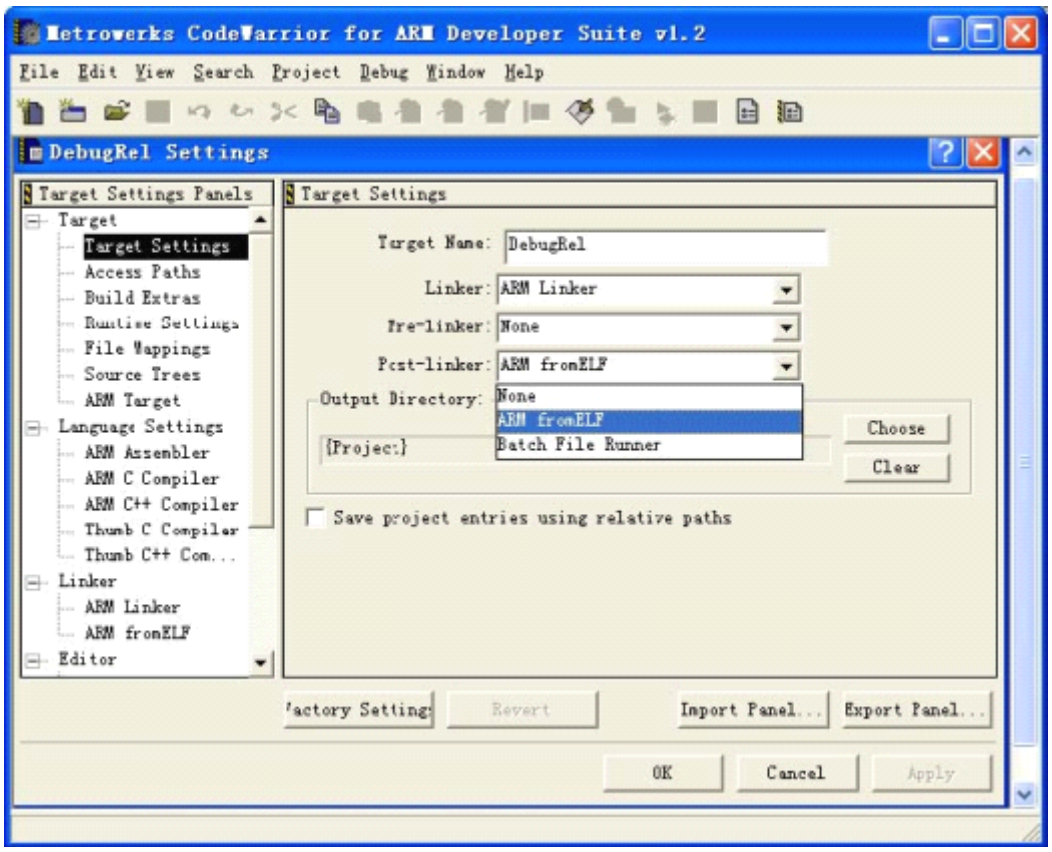


（2） 编译选项设置。点击确定按钮后出现工程窗口，在工程窗口中选择DebugRel Setting 进入编译选项设置对话框。

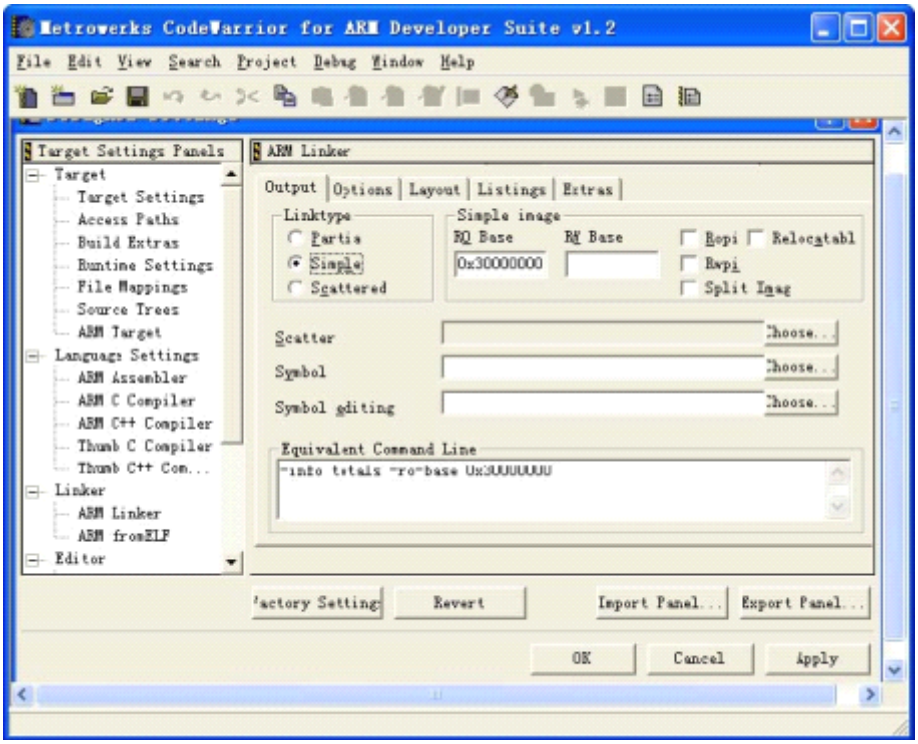




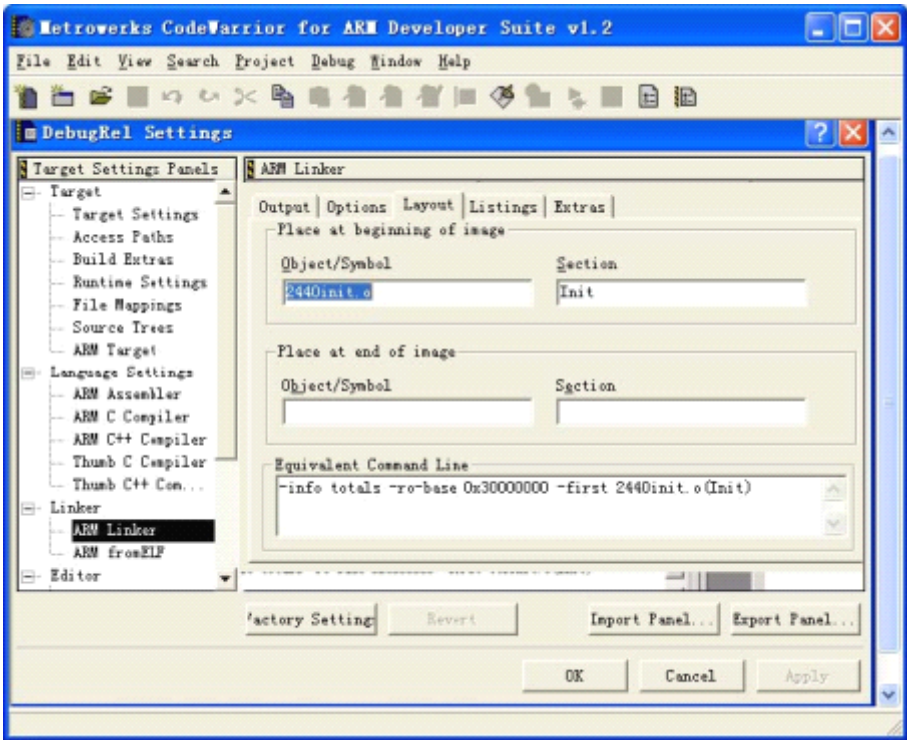
（3）在Debug Settings 对话框中选择Target Settings 选项，在Post-linker 列表框中选择ARM fromELF，单击右下角的Apply 按钮使其有效。



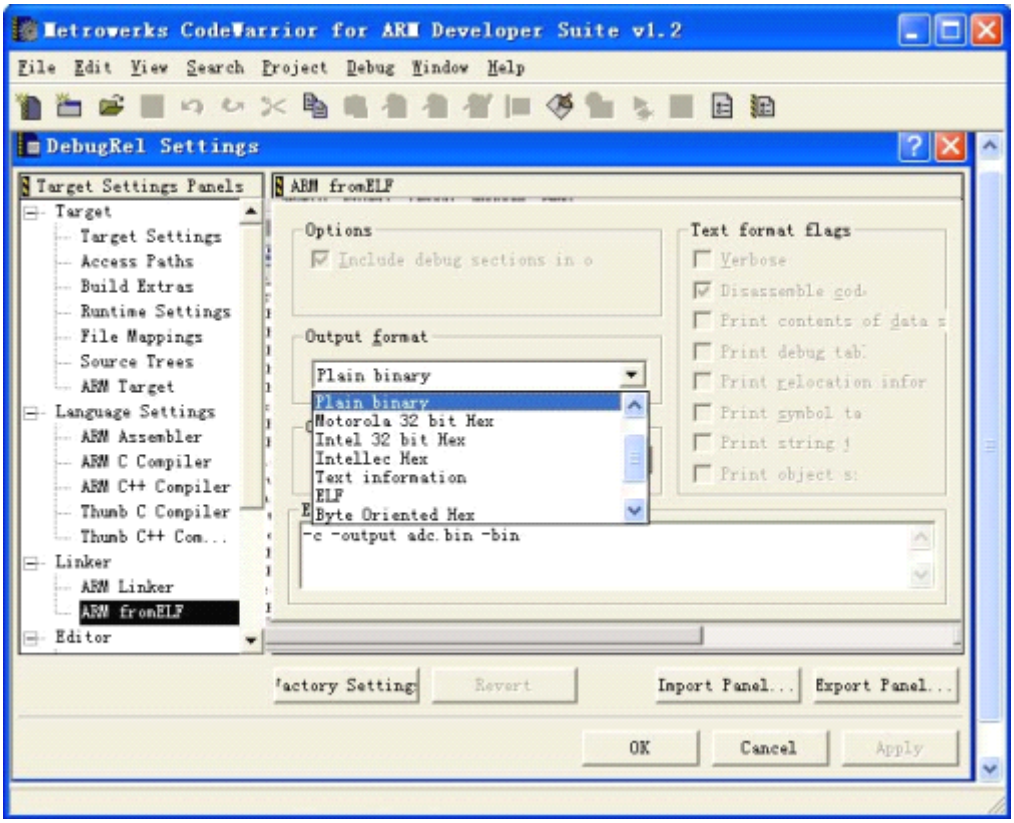
（4）设置Target Settings 在Debug Settings 对话框中选择ARM Linker 选项，选中Simple 单选按钮，在Simple image 选项组中设置连接的Read Only（只读）和Read-Write（读写）地址。地址0x30000000 是开发板上SDRAM 的真实地址，是由系统的硬件决定的。本实验中对系统可读写的内存地址并没有分配，系统将自动分配地址。



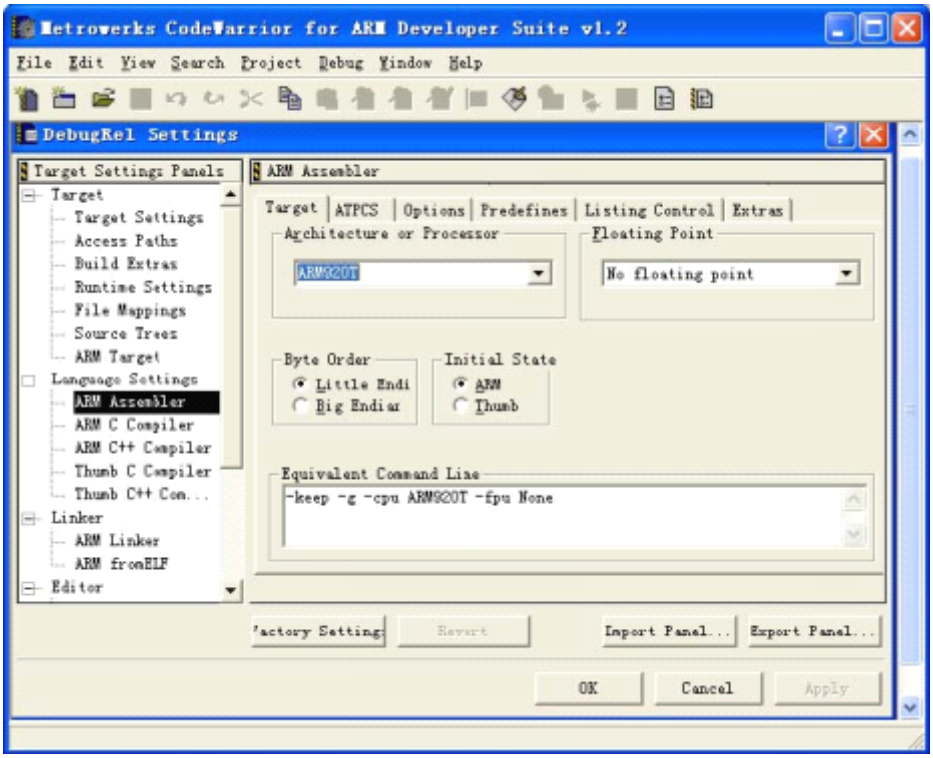
（5）点击Layout 选项页，在该选项页中的Place at beginning of image 选项组中设置程序的入口模块。指定在生成的代码中，程序是从2440init.s 开始运行的。Object/Symbol/ 项设为2440init.o，Section 项设为Init。



（6）在Debug Settings 对话框中选择ARM fromELF 选项，设置Output format 为Plain binary。也可在此对话框中设置生成的BIN 文件名。



（7） 最后在Language Settings 选项中的5 个子项中将“Architecture or Processor” 栏都选择为ARM920T。图中只列出了ARM Assembler 子项的设置情况。单击OK 按钮退出设置对话框。

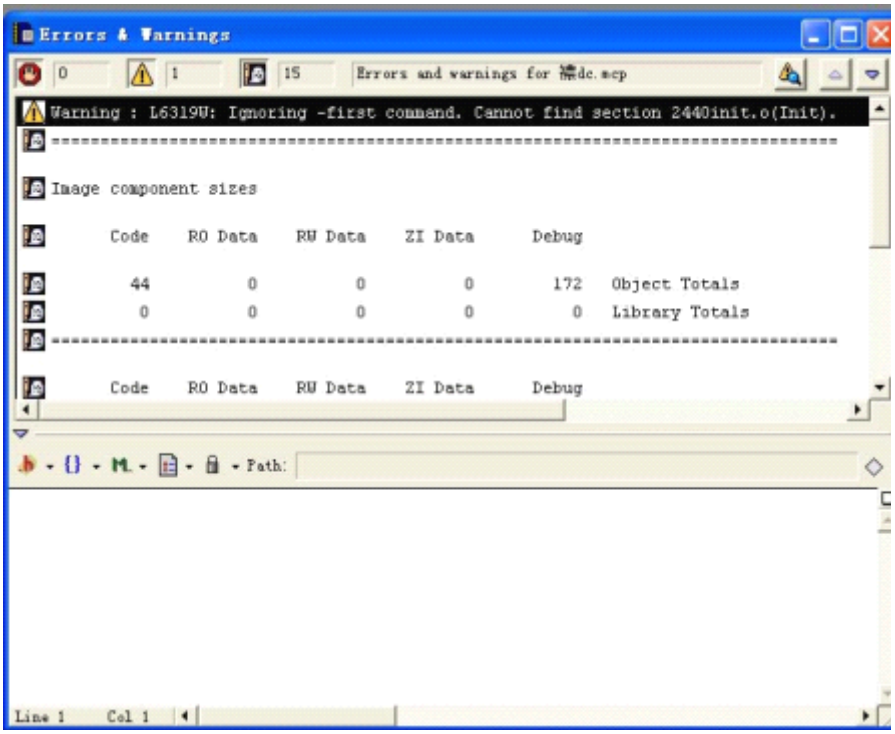


3 编辑源文件建立新的源文件或者添加已有的源文件皆可

选择File 菜单下的New，打开New 对话框。在该对话框中选择File 选项页，输入文件名称，文件存放路径并把它加入到刚才所建的工程中。

4 编译工程

在工程窗口中按“Make”按钮，或者直接按F7 快捷键，编译工程。在出现的错误/警告窗口中选择某错误/警告信息，ADS会自动打开相应的源文件并用箭头指向出错的文本行。错误/警告窗口。编译成功后在工程目录下的DebugRel 里会生成xx.bin文件。该文件可以直接下载到实验板上运行。



DNW软件安装和使用

嵌入式软件开发完成后，最终通过交叉编译，在目标系统上运行。运行的方式一般包括ROM 运行和RAM 运行两种。这就需要使用方便的工具来使用这些功能。在SinoSys-EA2440a中，已经通过JTAG 将一个功能比较完备的Boot Loader 烧写到Nor-Flash 中。在这段代码里驱动了SinoSys-EA2440a 的串口和USB 口，并实现了USB 读写内存及Flash 的烧写功能。这就需要我们有一个方便的工具实现串口及USB 口的操作，而DNW 工具软件正符合我们的要求。

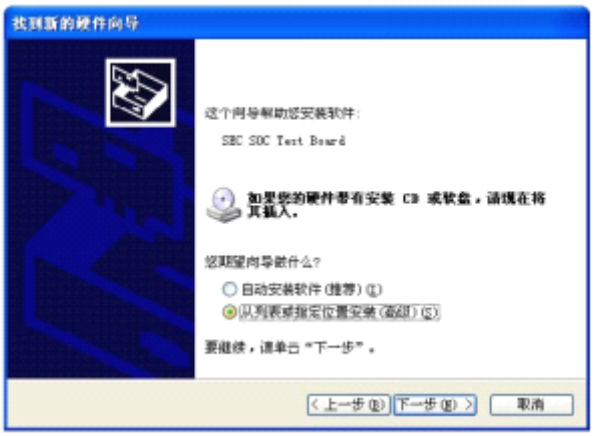
在PC 上安装DNW 工具需要先安装驱动程序，本实验主要介绍DNW 工具驱动的安装及如何使用DNW 工具配合Boot Loader 来实现内存读写和Flash 的烧写功能。总体来说，DNW 就是一个串口加USB 的终端工具。

打开试验箱包装，取出电源线将SinoSys-EA2440a 实验板与电源相连。取出USB 线将SinoSys-EA2440a 实验板和PC 机USB 口相连，取出串口线将SinoSys-EA2440a 和PC 机的串口相连。

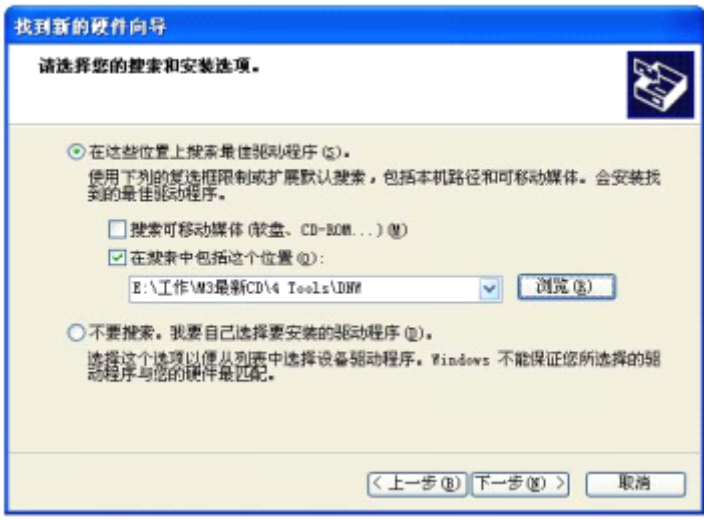
将tools 目录DNW 驱动程序的内容拷贝到用户PC 机上，然后去除拷贝好的全部文件的只读属性。记住这点很重要，否则DNW 不能正常工作。

将SinoSys-EA2440a 实验板设置为从Nor-Flash 启动，打开EA2440a 电源开关。如果是

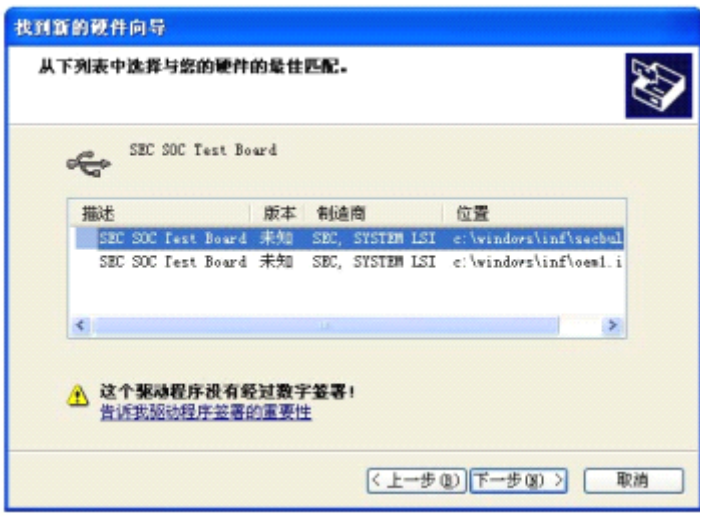
第一次使用DNW 工具的话，将会发现PC 机会有一个USB 设备被发现，下面开始添加驱动。选择从列表安装，点击下一步。如下图。



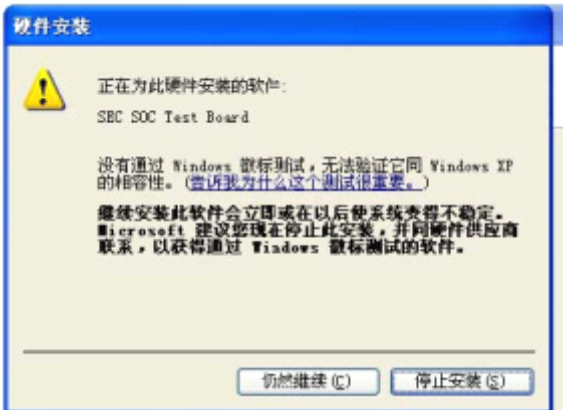
选择搜索路径，找到DNW 驱动程序所在路径。如下图。



点击下一步，找到匹配的驱动程序，如下图。

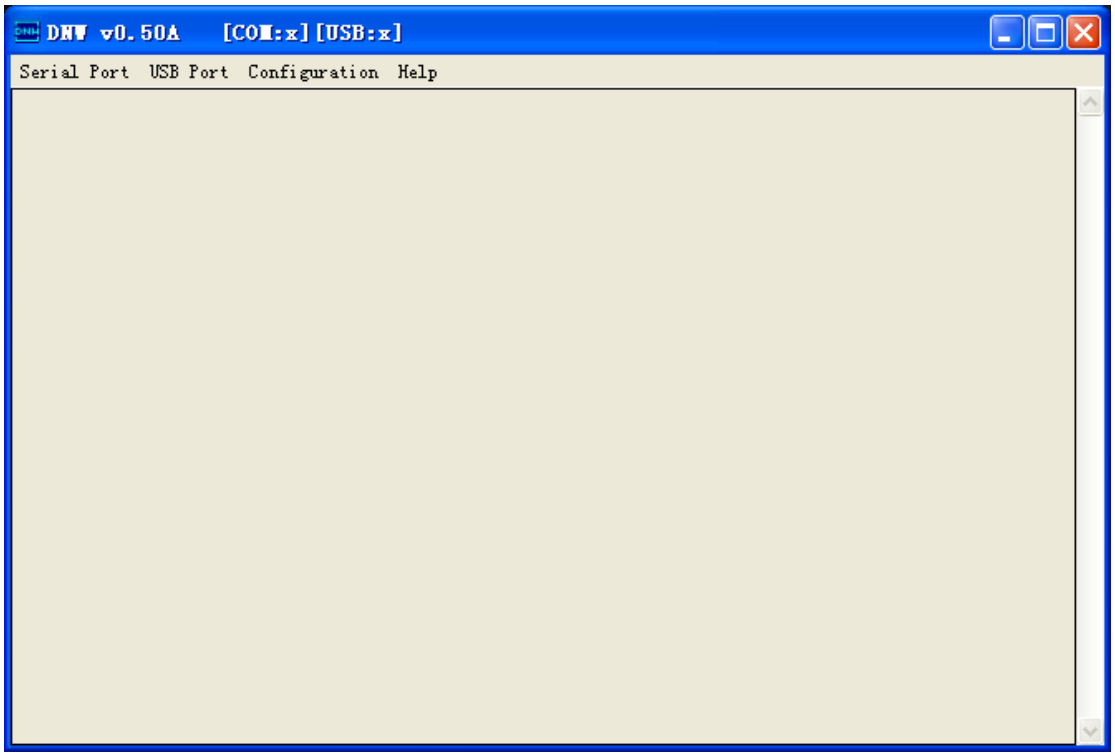


点击下一步，出现如下图对话框，选择“仍然继续”。



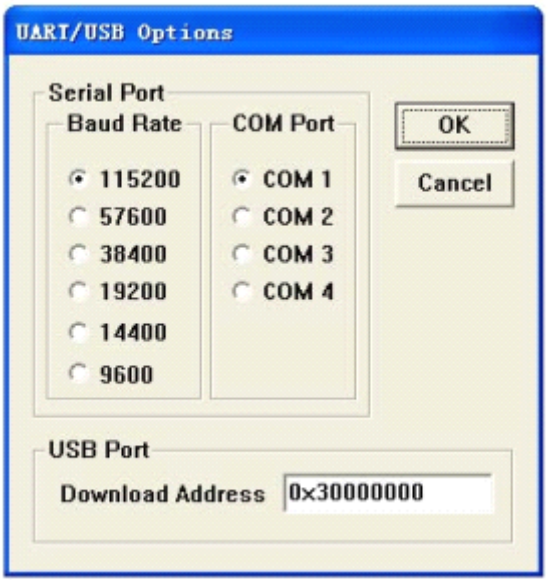
出现如下对话框，说明DNW 的驱动已经安装成功，接下来就可以使用DNW 工具了。

打开DNW 应用程序，界面如下图。



连接串口，选择“Serial Port”菜单下的“Connect”。现在就可以看到在上图所示的界面中对话框的标题栏中COM 串口和USB 口都已经连接好。

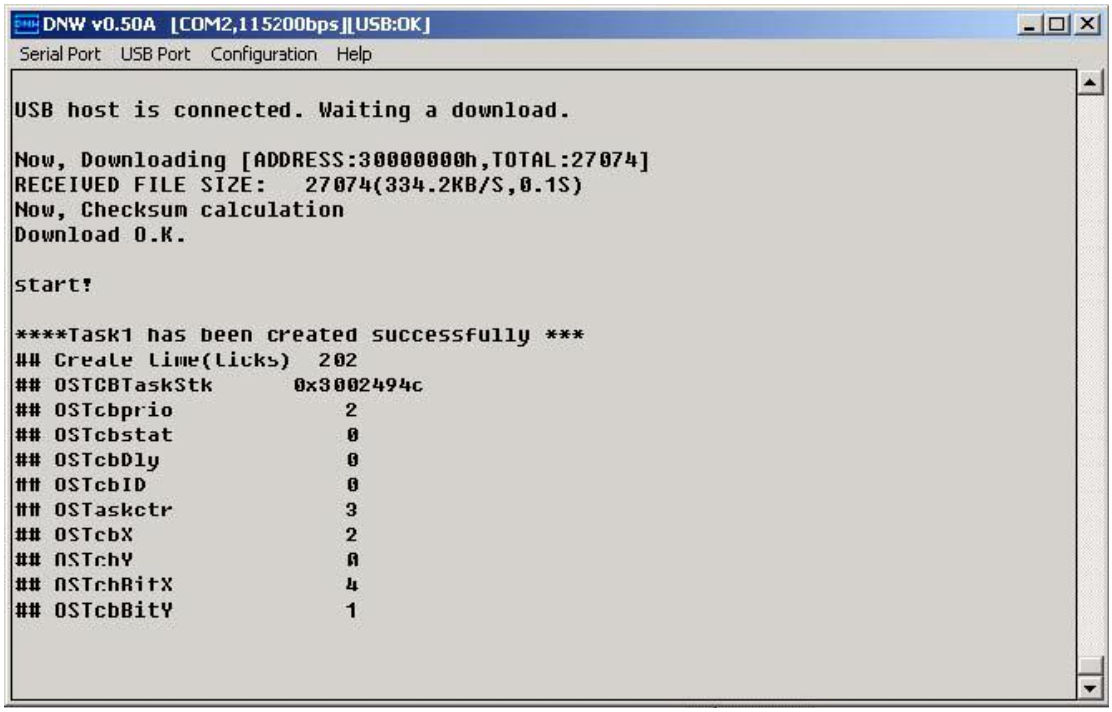
首先我们掌握使用DNW 工具下载可执行文件到内存中的方法。由于在SinoSys-EA2440a中RAM 的映射地址是0x30000000 ， 因此首先设定下载地址为0x30000000。选择Configuration”菜单下的“Options”，弹出如下对话框。



在这个对话框里我们可以设定COM 端口，串口比特率和USB 的下载地址。我们可以按照上图所示进行设置。单击OK 按钮退出。选择“USB Port”菜单下的“Transmit”，弹出打开文件按钮对话框，在对话框里选择要下载的可执行文件，单击打开按钮就可以把所选择的文件下载到地址为0x30000000 的内存中去了。

测试

(1). 打开\SourceCode\uCOS-II\proj\OS_taskinit\OS_taskinit.mcp 工程。
(2) 编译成功后生成文件OS_taskinit.bin，该文件已经是可执行的二进制文件了。使用DNW工具将该程序下载到RAM中运行。



6 实验思考

1、什么是可重入代码？如何使一个函数具有可重入性？
2、移植uC/OS-II操作系统对处理器有哪些要求？
3、怎样提高代码的可移植性？

实验二 任务管理实验

1 实验目的

1）掌握μ C/OS-II 基于优先级多任务原理。
2）学习编程实现μ C/OS-II 下多任务切换。
3）了解实时操作系统多任务管理的特性。

2 实验要求

1）创建四个任务实现流水灯的功能。
2）DNW中显示当前任务的编号。

参考工程： ..\SourceCode\uCOS-II\proj\OS_MULTITASK\OS_MULTITASK.mcp

3 准备事项

1）用ARM ADS1.2 集成开发环境，编写和调试程序的基本过程。
2）ARM 应用程序的框架结构。
3）μ C/OS-II 应用程序的框架结构。
4）硬件：ARM 嵌入式开发板、PC 机Pentumn100 以上、串口线
5）软件：PC 机操作系统win98 、Win2000 或WinXP 、ARM ADS1.2 集成开发环境、仿真器驱动程序、串口通讯程序、实验原理及说明

4 实验原理

4.1 任务

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为CPU完全只属于该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套CPU寄存器和自己的栈空间(如图2-1所示)。

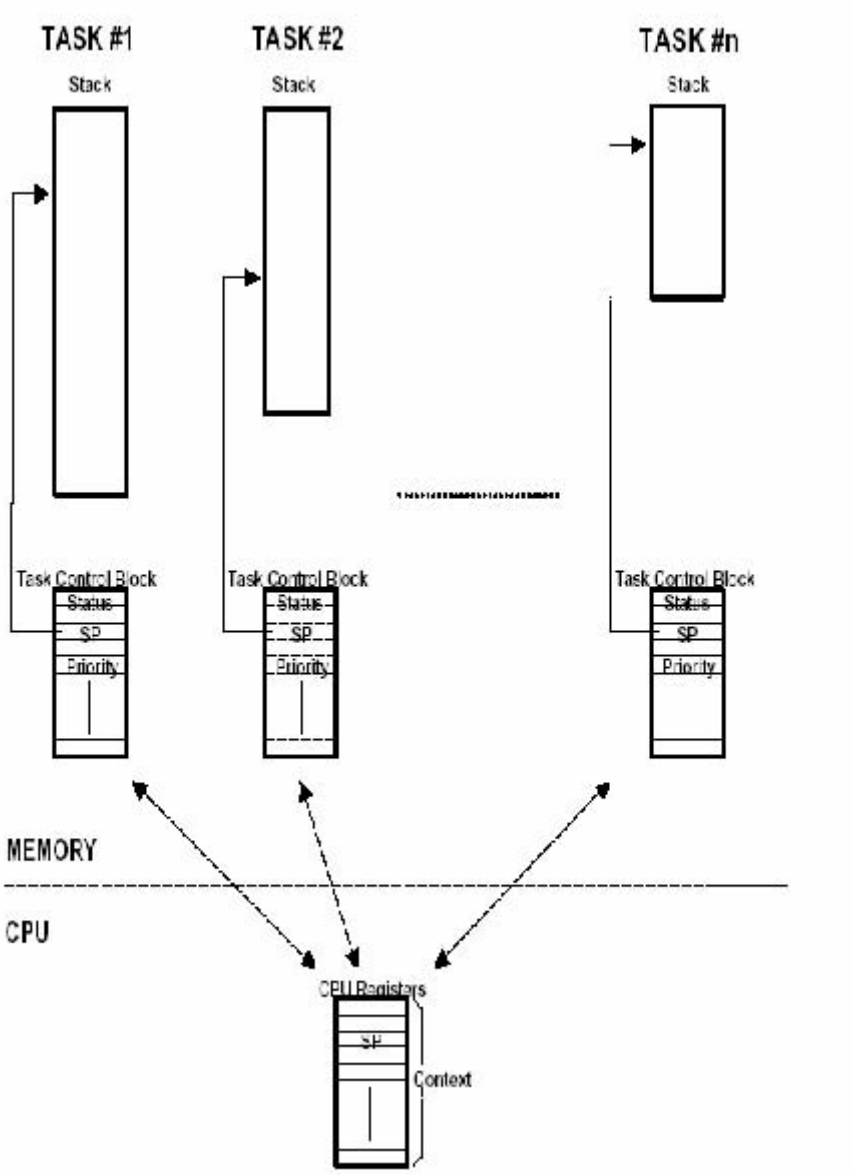


图2-1 多任务

4.2 任务状态

每个任务都是一个无限的循环。每个任务可以在五种状态下切换：休眠态，就绪态、运行态、挂起态(等待某一事件发生)和被中断态(参见图2-2）休眠态下，该任务驻留在内存中，但不被多任务内核所调度。就绪态时，该任务已经具备运行的条件，但由于其优先级比当前运行任务的优先级低，因此暂时不能运行。运行态的任务获得了CPU的控制权，正在运行中。挂起也叫等待事件态，指该任务在等待某一事件的发生，（如等待某外设的I/O操作、某共享资源变成可用状态、定时脉冲到来或等待超时信号以结束当前等待）。被中断态下，CPU 提供相应的中断服务，原来正在运行任务暂时不能运行，就进入了被中断状态。

下图表示μ C/OS-Ⅱ中一些函数提供的服务，这些函数可以使任务从一种状态变到另一种状态。

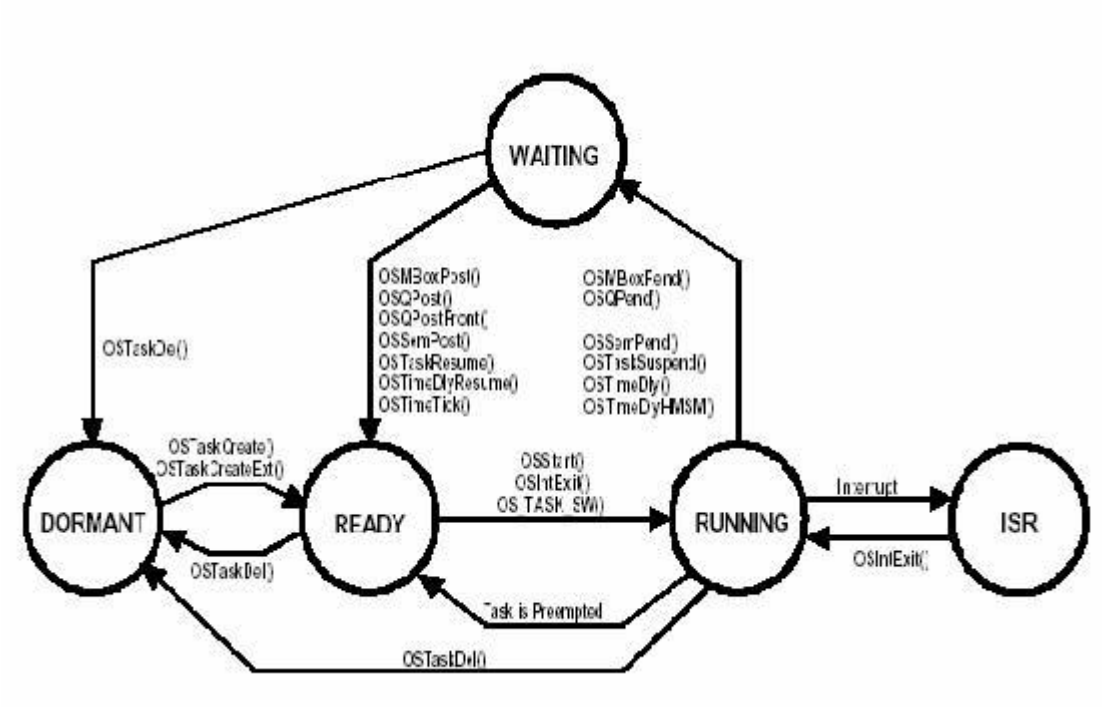


图2-2

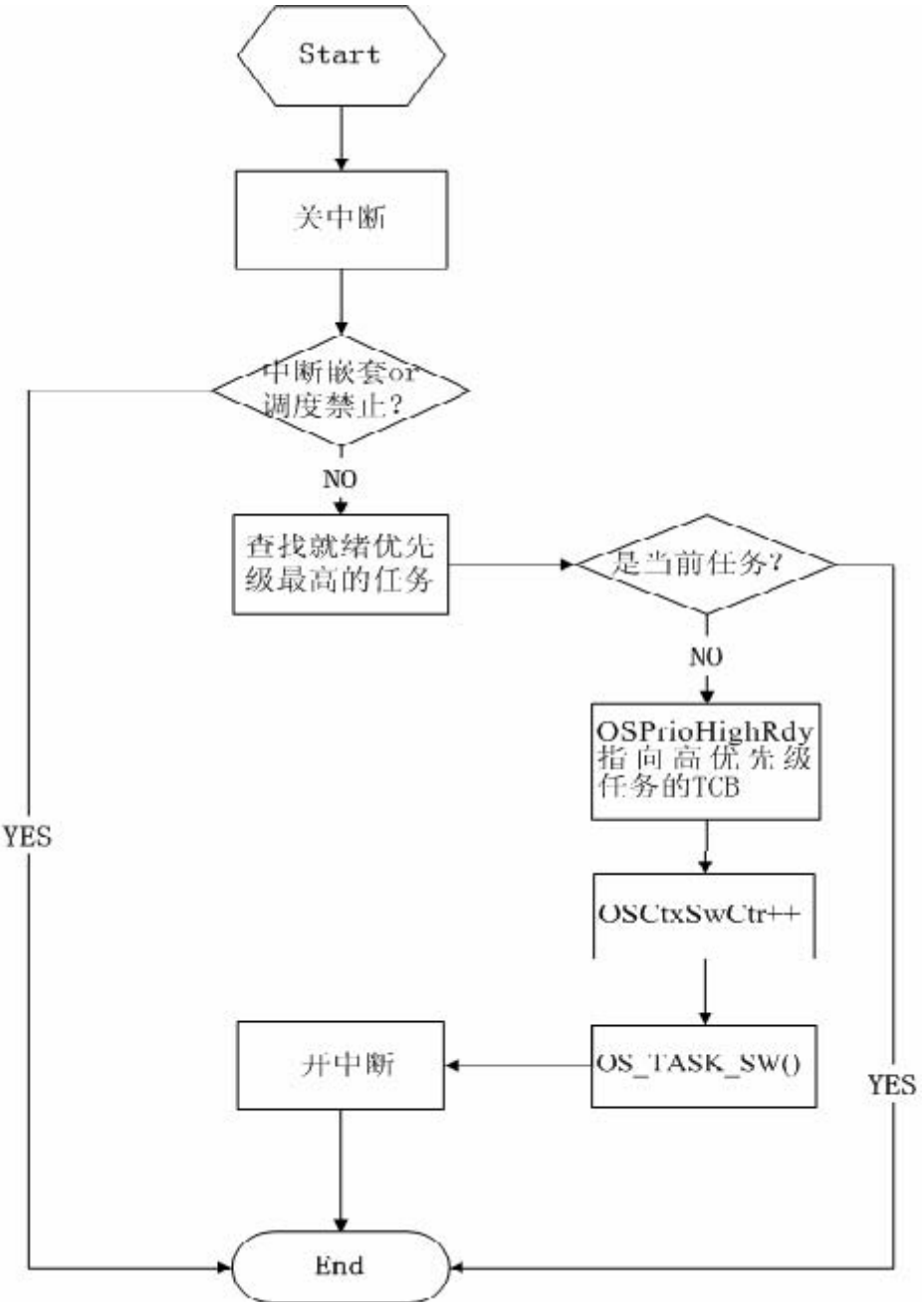
4.3 任务切换

Context Switch 即上下文切换，也叫任务切换，或CPU 寄存器内容切换。当多任务内核决定运行另外的任务时，它会保存正在运行任务的当前状态（Context），即CPU寄存器中的全部内容。这些内容被保存在其任务堆栈的当前状态存储区（Task's ContextStorage area） （见图2-1）。入栈完毕后，下一个将要运行的任务的当前状态被装入CPU的寄存器，并开始运行,这个过程叫做任务切换。任务切换过程增加了应用程序的额外负荷。CPU的内部寄存器越多，额外负荷就越重。任务切换所需的时间取决于CPU有多少寄
存器要入栈。实时内核的性能不应该以每秒钟能做多少次任务切换来评价。

4.4 任务调度

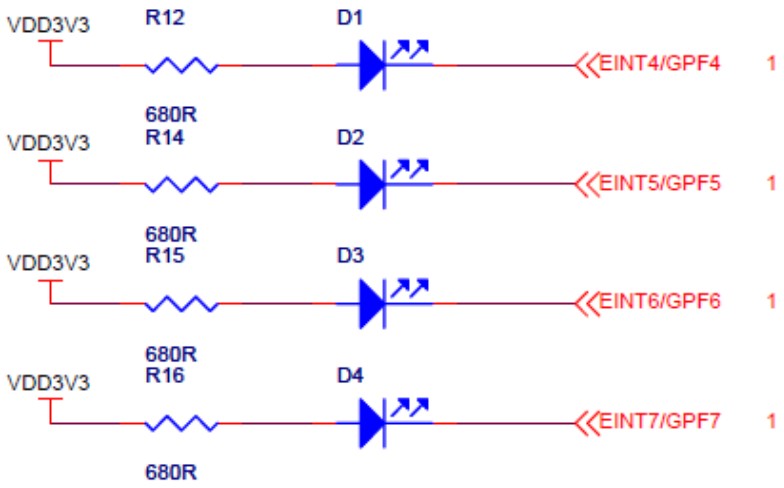
μC/OS-Ⅱ总是运行就绪态任务中优先级最高的那一个。确定哪个任务优先级最高，接下来该哪个任务运行了的工作由调度器（Scheduler）完成。任务级的调度是由函数OSSched()完成；中断级的调度是由函数OSIntExt()完成的。OSSched() 流程如图2-3所示。
由宏调用OS_TASK_SW()实现的任务切换分两步完成。首先将被挂起任务的微处理器寄存器推入堆栈，然后将较高优先级的任务的寄存器值从栈中恢复到寄存器中。在μC/OSⅡ中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有微处理器的寄存器都保存在栈中。μC/OS-Ⅱ进行任务调度所要做的一切，只是恢复所有的CPU寄存器并运行中断返回指令。为了做任务切换，运行OS_TASK_SW(),人为模仿了一次中断。中断服务子程序或陷阱处理（Trap hardler），也称作事故处理（exception handler），必须提供中断向量给汇编语言函数OSCtxSw()。OSCtxSw()除了需要OS_TCBHighRdy指向即将被挂起的任务，还需要让当前任务控制块OSTCBCur指向即将被挂起的任务。OSSched()的所有代码都属临界段代码。在寻找进入就绪态的优先级最高的任务过程中，为防止中断服务子程序把一个或几个任务的就绪位置位，

中断是被关掉的。为缩短切换时间，OSSched()全部代码都可以用汇编语言写。为增加可读性、可移植性和将汇编语言代码最少化，OSSched()是用C写的。



4.4 S3C2440 LED流水灯设计

下图对应S3C2440开发板LED灯的原理图及对应端口，实现对端口的控制首先要配置相应的寄存器，可参见S3C2440A手册。可调用2440lib.c中的void Led_Display(int data) 函数实现流水灯的控制。


VDD3V3
R12
680R
D1
EINT4/GPF4
1
VDD3V3
R14
680R
D2
EINT5/GPF5
1
VDD3V3
R15
680R
D3
EINT6/GPF6
1
VDD3V3
R16
680R
D4
EINT7/GPF7
1

实验三 同步和进程通信实验

1 实验目的

1）掌握μ C/OS-II内部进程通信同步机制。
2）学习编程实现μ C/OS-II的进程间的同步模拟。
3）学习μ C/OS-II基于信号量、邮箱和消息队列实现任务之间的通信机制。

2 实验要求

按下开发板上的按键、令ARM芯片启动A/D采样；采样结束后，再将采样结果通过串口线发送到电脑上并在DNW上显示

3 准备事项

1）用ARM ADS1.2集成开发环境，编写和调试程序的基本过程。
2）ARM应用程序的框架结构。
3）μ C/OS-II应用程序的框架结构。
4）硬件：ARM嵌入式开发板、PC机Pentumn100以上、串口线
5）软件：PC机操作系统win98、Win2000或WinXP、ARM ADS 1.2集成开发环境、仿真
器驱动程序、串口通讯程序、实验原理及说明

4实验原理

4.1信号量

信号量实际上是一种约定机制，在多任务内核中普遍使用.信号量用于：
1. 控制共享资源的使用权(满足互斥条件)
2. 标志某事件的发生
3. 使两个任务的行为同步

一般地说，对信号量只能实施三种操作：初始化(INITIALIZE)，也可称作建立(CREATE)；等信号(WAIT)也可称作挂起(PEND)；给信号(SIGNAL)或发信号(POST)。信号量初始化时要给信号量赋初值，等待信号量的任务表(Waiting list)应清空。

4.2进程同步

可以利用信号量使任务与中断服务同步(或者是与另一个任务同步，这两个任务间没
有数据交换)。如图 3-1 所示。注意，图中用一面旗帜，或称作一个标志表示信号量。这
个标志表示某一事件的发生(不再是一把用来保证互斥条件的钥匙)。用来实现同步机制的

信号量初始化成 0，信号量用于这种类型同步的称作单向同步(unilateral rendezvous)。
一个任务做 I/O 操作时因等待某一信号而进入就绪态。当中断服务程序(或另外一个任务)
发出信号时，该任务得以回到运行态继续往下执行。

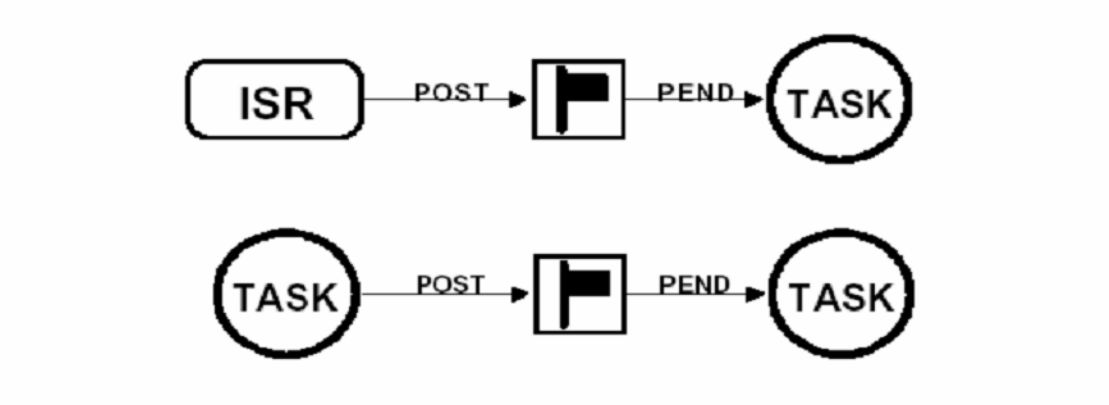


图 3-1 用信号量使任务与中断服务程序（或另一任务）同步
两个任务可以用两个信号量同步它们的行为。如图 3-2 所示，称之为双向同步(bilateral rendezvous)。双向同步与单向同步类似，只是两个任务要相互同步。

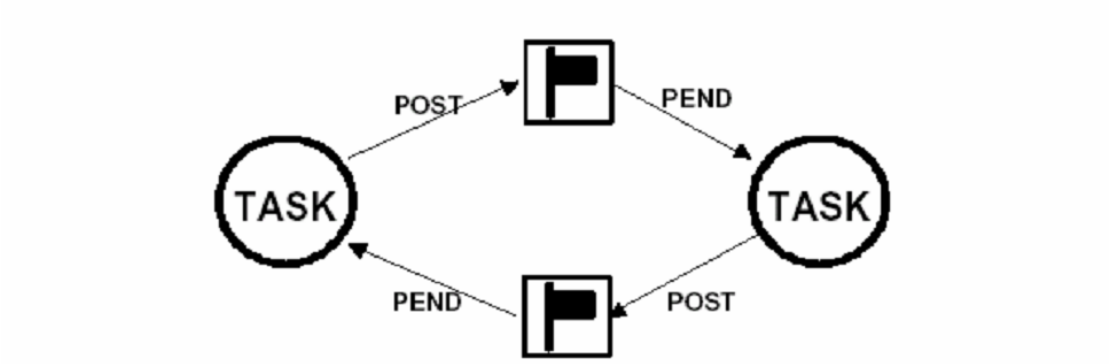


图3-2两个任务用信号量同步彼此的行为

4.3 任务通信（Intertask communication）

有时在任务间或中断服务与任务间的通讯是很必要的，这种信息传递称为任务间的通讯。任务间信息的传递有两个途径：通过全局变量或发消息给另一个任务 用全局变量时，必须保证每个任务或中断服务程序独享该变量。中断服务中保证独享的唯一办法是关中断。如果两个任务共享某变量，各个任务实现独享该变量的办法可以是关中断再开中断，或使用信号量(如前面提到的那样)。请注意，任务只能通过全程变量与中断服务程序通讯，而任务并不知道什么时候全局变量被中断服务程序修改了，除非中断程序以信号量方式向任务发送信号或者是该任务以查询方式不断周期性地查询变量的值。
要避免这种情况，用户可以考虑使用邮箱或消息队列。

4.4 消息邮箱（Message Mail Box）

通过内核服务可以给任务发送消息。典型的消息邮箱也称作交换消息，是用一个指针型变量，通过内核服务，一个任务或一个中断服务程序可以把一则消息

(即一个指针)放到邮箱里去。同样，一个或多个任务可以通过内核服务接收这则消息。发送消息的任务和接收消息的任务约定传递的消息实际上是传递的指针指向的内容。 每个邮箱均有相应的正在等待消息的任务列表，要得到消息的任务会因为邮箱是空的而被挂起，且被记录到等待消息的任务表中，直到收到消息。一般地说，内核允许用户定义等待超时，等待消息的时间超过了预定值仍然没有收到该消息则这任务进入就绪态，并返回出错信息报告等待超时错误。消息放入邮箱后，或者是把消息传给等待消息的任务表中优先级最高的那个任务(基于优先级)，或者是将消息传给最先开始等待消息的任务(基于先进先出)。图3-3所示，一个任务把消息放入邮箱，与另一个任务通信。

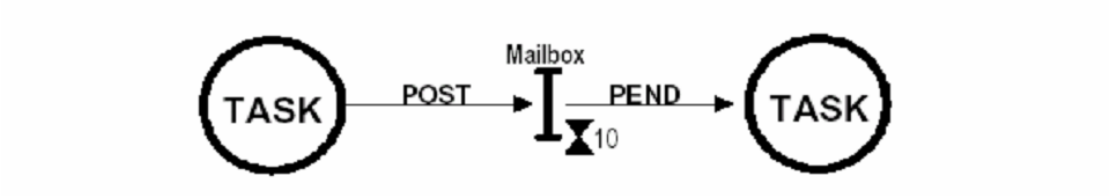


图3-3消息邮箱

内核一般提供以下邮箱服务：

1. 邮箱内消息的内容初始化，邮箱里最初可以有，也可以没有消息
2. 将消息放入邮箱(POST)
3. 等待有消息进入邮箱(PEND)
4. 如果邮箱内有消息，就接受这则消息。如果邮箱里没有消息任务也不被挂起(ACCEPT)而是用返回代码表示调用结果，是收到了消息还是没有收到消息。

4.5消息队列(Message Queue）

消息队列实际上是邮箱阵列。通过内核提供的服务，任务或中断服务程序可以将一条
消息(该消息的指针)放入消息队列。同样，一个或多个任务可以通过内核服务从消息队列
中得到消息。发送和接收消息的任务约定，传递的消息实际上是传递的指针指向的内容。
通常，先进入消息队列的消息先传给任务，即任务先得到的是最先进入消息队列的消息，
即先进先出原则(FIFO)。μ C/OS-Ⅱ也允许使用后进先出方式(LIFO)。
内核提供的消息队列服务如下：

1. 消息队列初始化。队列初始化时总是清为空。
2. 放一则消息到队列中去(Post)
3. 等待一则消息的到来(Pend)
4. 如果队列中有消息则任务可以得到消息，但如果此时队列为空，内核并不将该任务
挂起(Accept)。如果有消息，则消息从队列中取走。没有消息则用特别的返回代码
通知调用者，队列中没有消息。

4.6 按键与AD

下图对应S3C2440开发板按键端口及AD，实现对端口的控制首先要配置相应的寄存器，寄存器配置及端口操作参见手册S3C2440A。

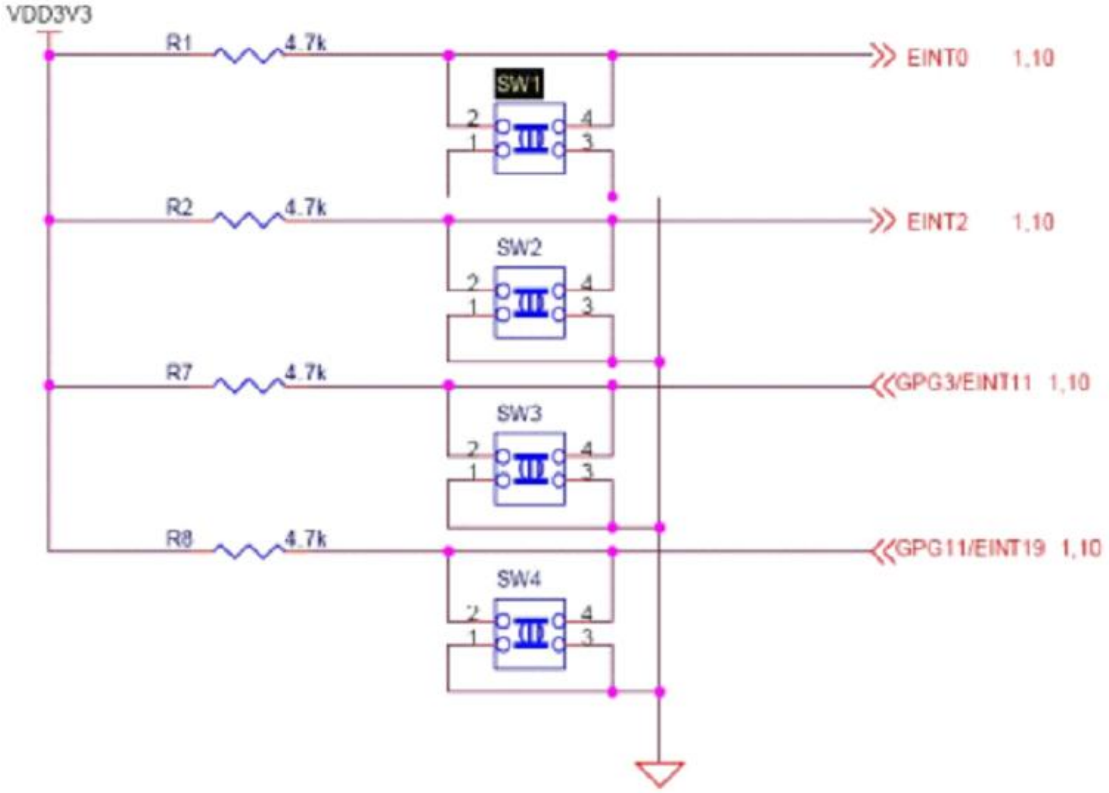


图 3-4 S3C2440A 按键对应端口及原理图

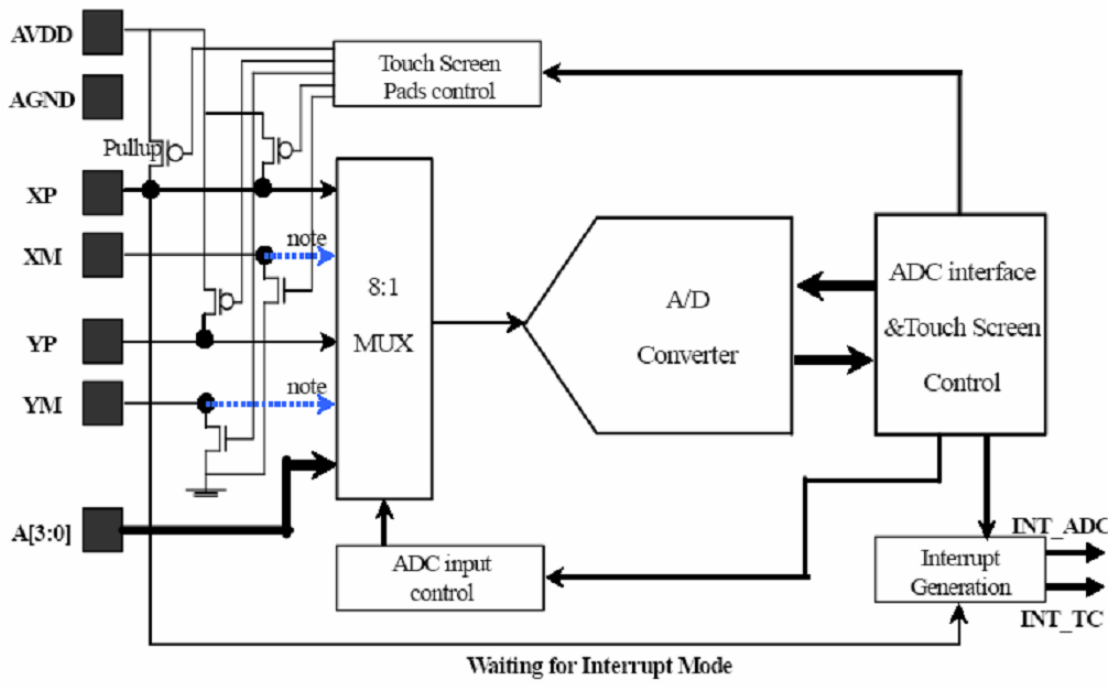


图 3-5 S3C2440A 的 ADC 内部结构图

实验五 实时中断实验

1 实验目的

1）掌握μ C/OS-II中断处理机制。
2）掌握实时操作系统的中断响应和处理过程。
3）学习编程实现μ C/OS-II中一个简单中断服务程序。
4）编程体会实时系统对中断服务例程的要求。

2 实验要求

开始时蜂鸣器一直在间歇鸣叫；用户按下板上的按键、触发外部中断；外部中断再触发“读取ADC然后将采样结果发送至电脑DNW显示”的操作。

3 准备事项

1）用ARM ADS1.2集成开发环境，编写和调试程序的基本过程。
2）ARM应用程序的框架结构。
3）μ C/OS-II应用程序的框架结构。
4）实时系统中中断及任务之间的通信机制。
5）中断和中断服务程序过程和特点。
6）硬件：ARM嵌入式开发板、PC机Pentumn100以上、串口线
7）软件：PC机操作系统win98、Win2000或WinXP、ARM ADS1.2集成开发环境、仿真器驱动程序、串口通讯程序、实验原理及说明

4 实验原理

4.1中断

中断是一种硬件机制。用于通知 CPU“有异步事件发生”。中断一旦被识别，CPU 将保存部分（或全部）现场。即部分或者全部寄存器的值，跳转到专门的子程序（中断服务子程序 ISR）。中断服务子程序做出事件响应处理。完毕后，程序回到：

1. 前 /后台系统中，程序回到后台程序。
2. 对不可剥夺型内核而言，程序回到被中断的任务
3. 对可剥夺型内核而言，让进入就绪态的任务级最高的任务开始运行。

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询是否有事件发生。在实时系统中，关中断的时间应该尽量短。关中断影响中断延迟时间。关中断时间越长，可能会引起中断丢失。微处理器一般允许中断嵌套，即在中断服务期间，处理器可以识别另一个更重要的中断，并服务于那个更重要

的中断。

4.2 ISR处理时间

中断服务的处理时间应该尽可能的短，但是对处理时间并没有绝对的限制。不能说中断服务必须全部小于 100μ S，500μ S 或 1mS。如果中断服务是在任何给定的时间开始，且中断服务程序代码是应用程序中最重要的代码，则中断服务需要多长时间就应该给它多长时间。

在大多数情况下，中断服务子程序应识别中断来源，从请求中断的设备取得数据或状态，并通知真正做该事件处理的那个任务。当然应该考虑到，通知一个任务去做事件处理所花的时间是否比处理这个事件所花的时间还多。在中断服务中，通知一个任务做时间处理(通过信号量、邮箱或消息队列)是需要一定时间的，如果事件处理需花的时间短于给一个任务发通知的时间，就应该考虑在中断服务子程序中做事件处理，并在中断服务子程序中开中断，以允许优先级更高的中断打入并优先得到服务。

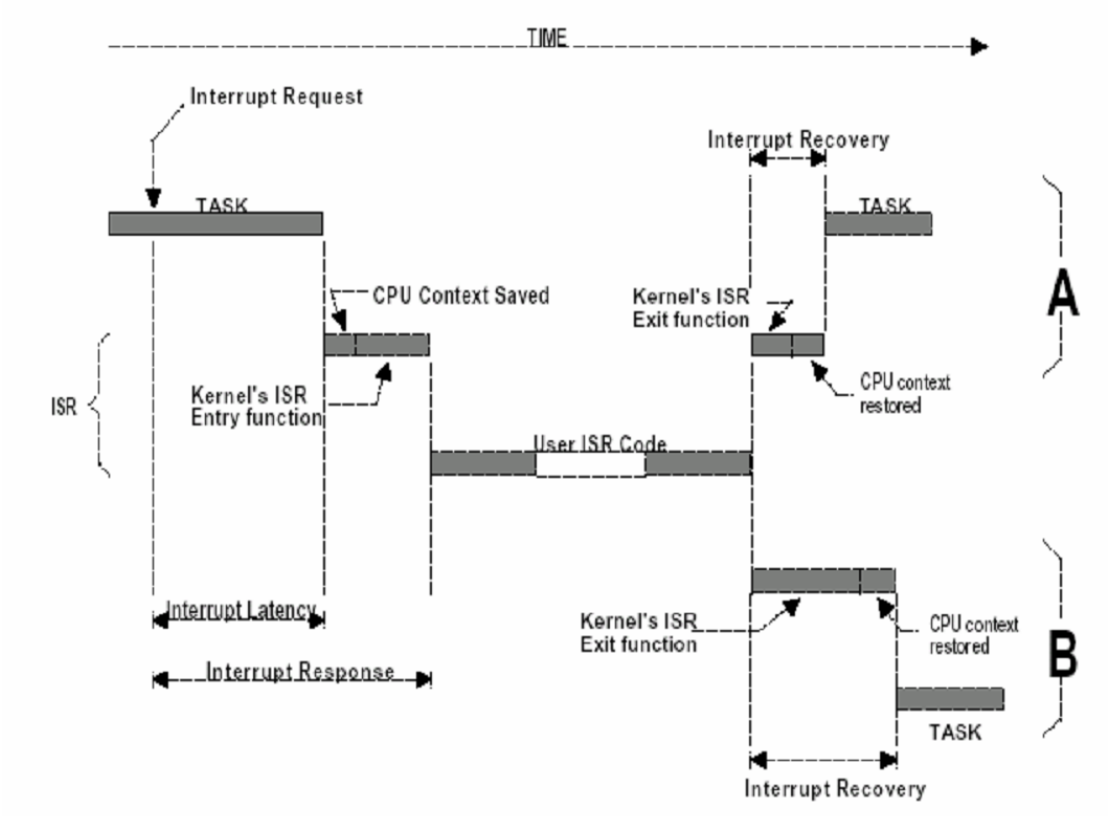


图4-1中断延迟、响应和恢复(可剥夺型内核)

4.3 μC/OS中的中断处理

μ C/OS 中，中断服务子程序要用汇编语言来写。如果 C 编译器支持在线汇编，则用户可以直接将中断服务子程序代码放在 C 语言的程序文件中。中断服务子程序的流程如图 4-2 所示

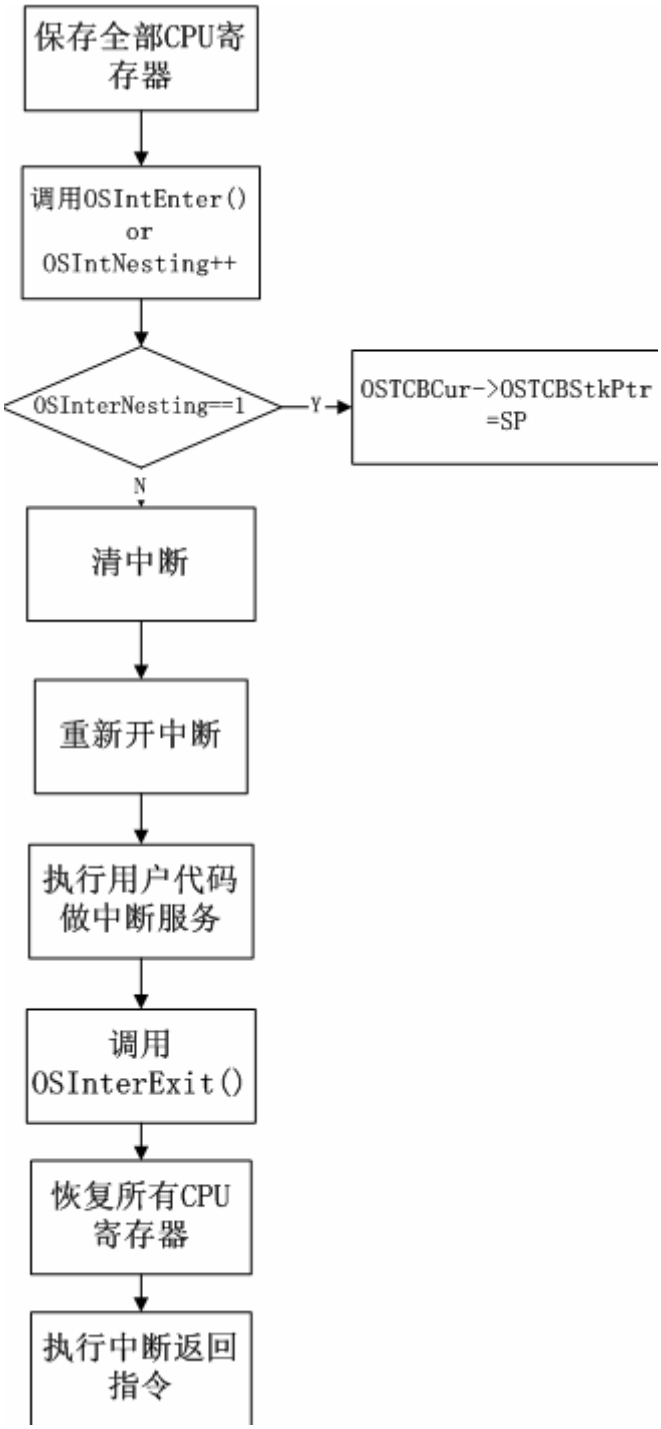


图 4-2 中断处理过程

注意：1）用户代码应将全部CPU寄存器推入当前任务栈

2）用户应该调用OSIntEnter()或者将全局变量OSIntNesting加1，使μC/OS-Ⅱ知道用户在做中断服务。

3）用户代码响应中断，进入ISR运行。为允许中断嵌套，在多数情况下，用户应在开中断之前先清中断源。

4）ISR 结束后，调用OSIntExit()表示终结，OSIntExit()将中断嵌套层数计数器减一。当所有中断（包括嵌套的中断）都完成的时候，μC/OS-

Ⅱ需要判定有没有优先级搞的任务被ISR（或任一嵌套的中断）唤醒。

5）优先级高的任务处于就绪态，μC/OS-Ⅱ返回该任务，并且恢复所保存的寄存器的值。执行中断返回。

6）若调度被禁止，则μC/OS-Ⅱ返回到被中断了的任务继续执行。

图 4-3 为中断处理时序图。中断来到了[(1)]但还不能被被 CPU 识别，也许是因为中断被μC/OS-Ⅱ或用户应用程序关了，或者是因为 CPU 还没执行完当前指令。一旦 CPU 响应了这个中断[（2）]，CPU 的中断向量（至少大多数微处理器是如此)跳转到中断服务子程序[(3)]。如上所述，中断服务子程序保存 CPU 寄存器（也叫做 CPU context）[（4）]，一旦做完，用户中断服务子程序通知μC/OS-Ⅱ进入中断服务子程序了，办法是调用OSIntEnter()或者给OSIntNesting 直接加 1[（5）]。然后用户中断服务代码开始执行[（6）]。用户中断服务中做的事要尽可能地少，要把大部分工作留给任务去做。中断服务子程序通知某任务去做事的手段是调用以下函数之一：OSMboxPost()，OSQPost()，OSQPostFront()，OSSemPost()。中断发生并由上述函数发出消息时，接收消息的任务可能是，也可能不是挂起在邮箱、队列或信号量上的任务。用户中断服务完成以后，要调用OSIntExit()[（7）]。

从时序图上可以看出，对被中断了的任务说来，如果没有高优先级的任务被中断服务子程序激活而进入就绪态，OSIntExit()只占用很短的运行时间。进而，在这种情况下，CPU 寄存器只是简单地恢复[（8）]并执行中断返回指令[（9）]。如果中断服务子程序使一个高优先级的任务进入了就绪态，则 OSIntExit()将占用较长的运行时间，因为这时要做任务切换[（10）]。新任务的寄存器内容要恢复并执行中断返回指令[(12)]。

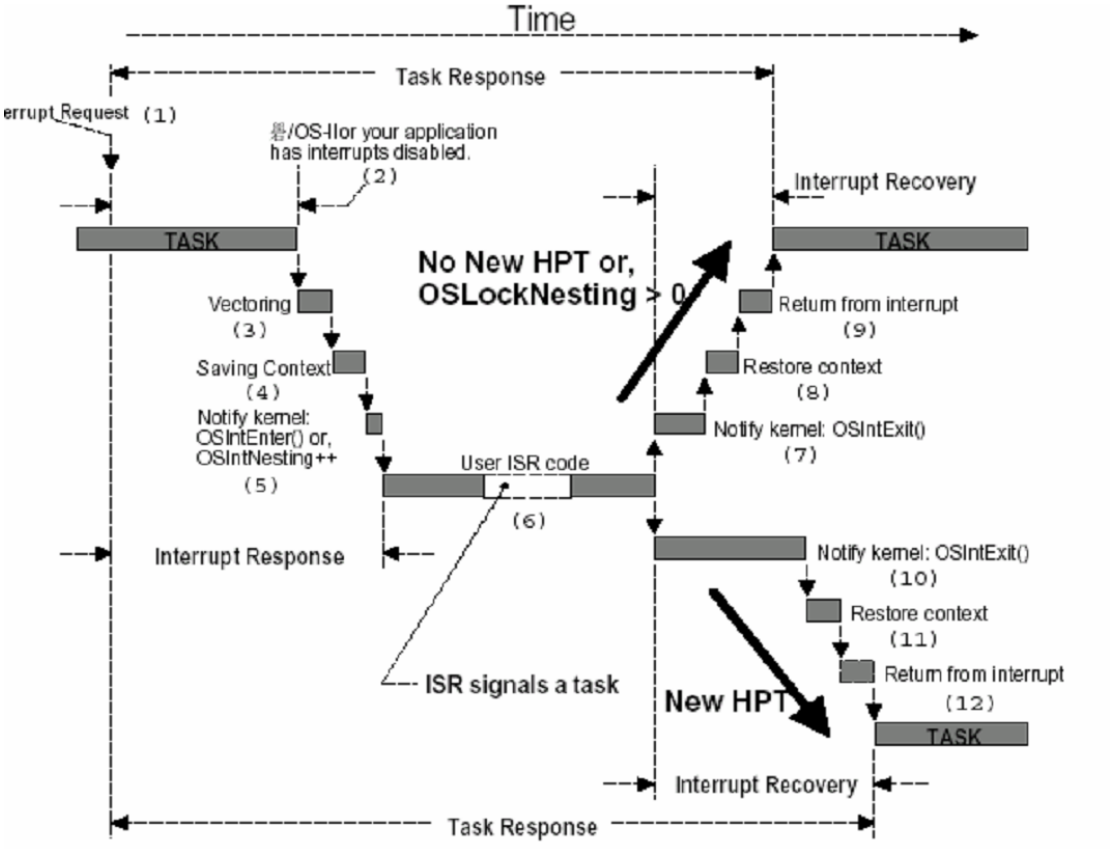
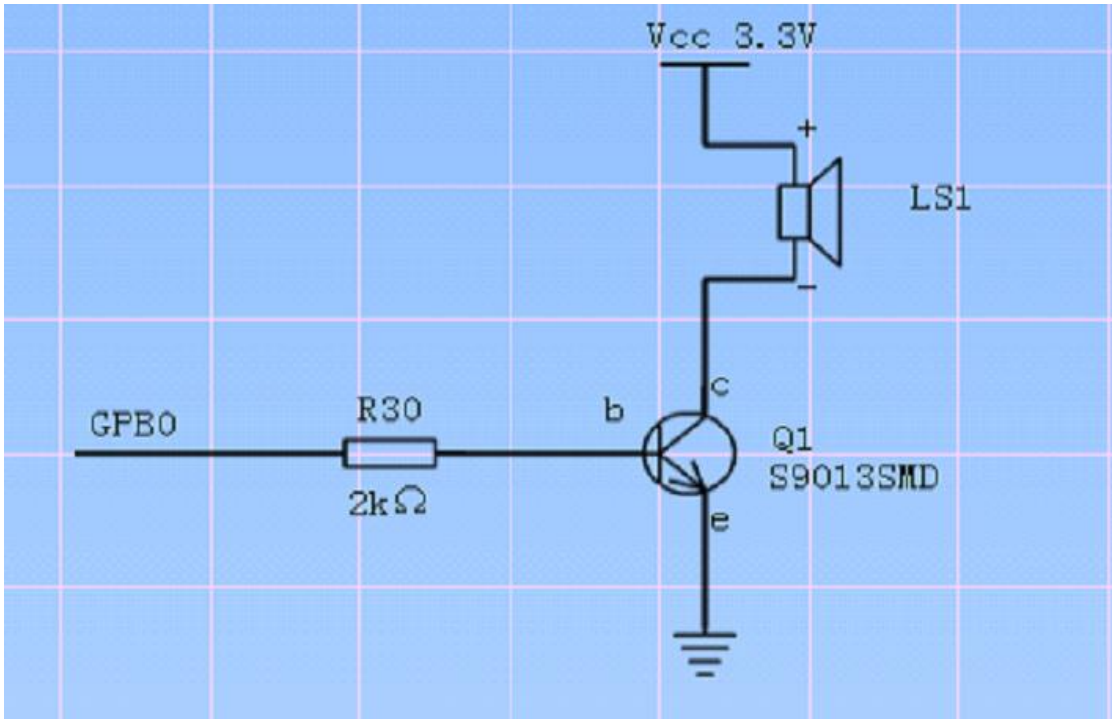


图 4-3 中断服务

4.4 蜂鸣器接口电路

系统的报警器电路就是由平台将 S3C2440 的定时器 0 的脉冲输出端口（TOUT0）GPB0与报警器的脉冲输入端口相连。在系统初始化时，就要进行 I/O 端口初始化，设置端口控制寄存器（将再下面讲到），将 GPB0 设置为工作方式 1，并设置为输出状态。 控制电路如图所示



课程设计

选择一：

题目：万年历的设计

要求：1、基于uc/os II 操作系统在三星S3C2440开发板进行设计

2、实时显示：不断显示当前的年月日、时间信息；

3、时间设置：可设置万年历的时钟和年月日信息，完成设置后，可更新显示。

4、闹钟等定时提醒功能。

选择二：

题目及设计自拟，要求基于uc/os II 操作系统在三星S3C2440开发板进行设计。