教程：建立一个属于自己的 AVR的 RTOS

[日期： 2006-2-8] 来源： 21icbbs.com 作者：黄健昌 [字体：大 中 小]

序

自从 03 年以来，对单片机的 RTOS 的学习和应用的热潮可谓一浪高过一浪.03 年，在离开校园前的，非典的那几个月，在华师的后门那里买了本邵贝贝的《UCOSII》，通读了几次，没有实验器材，也不了了之。

在 21IC 上，大家都可以看到杨屹写的关于 UCOSII 在 51 上的移植，于是掀起了 51 上的 RTOS 的热潮。

再后来，陈明计先生推出的 small rots，展示了一个用在 51 上的微内核，足以在 52 上进行任务调度。

前段时间，在 ouravr 上面开有专门关于 AVR 的 Rtos 的专栏，并且不少的兄弟把自己的作品拿出来，着实开了不少眼界。这时，我重新回顾了使用单片机的经历，觉得很有必要，从根本上对单片机的 RTOS 的知识进行整理，于是，我开始了编写一个用在 AVR 单片机的 RTOS。

当时，我所有的知识和资源有：

Proteus6.7 可以用来模拟仿真 avr 系列的单片机
WinAVR v2.0.5.48 基于 GCC AVR 的编译环境，好处在于可以在 C 语言中插入 asm 的语句
mega8 1K 的 ram 有 8K 的 rom,是开发 8 位的 RTOS 的一个理想的器件，并且我对它也比较熟悉。

写 UCOS 的 Jean J.Labrosse 在他的书上有这样一句话，“渐渐地，我自然会想到，写个实时内核直有那么难吗？不就是不断地保存，恢复 CPU 的那些寄存器嘛。”

好了，当这一切准备好后，我们就可以开始我们的 Rtos for mega8 的实验之旅了。

本文列出的例子，全部完整可用。只需要一个文件就可以编译了。我相信，只要适当可用，最简单的就是最好的，这样可以排除一些不必要的干扰，让大家专注到每一个过程的学习。

第一篇：函数的运行

在一般的单片机系统中，是以前后台的方式(大循环+中断)来处理数据和作出反应的。

例子如下:

makefile 的设定:运行 WinAvr 中的 Mfile，设定如下

MCU Type: mega8

Optimization level: s

Debug format :AVR-COFF

C/C++ source file: 选译要编译的 C 文件

```
#include <avr/io.h>
void fun1(void)
{
  unsigned char i=0;
  while(1)
  {
    PORTB=i++;
    PORTC=0x01<<(i%8);
  }
}

int main(void)
{
  fun1();
}
```

首先，提出一个问题：如果要调用一个函数，真是只能以上面的方式进行吗？

相信学习过 C 语言的各位会回答，No!我们还有一种方式，就是“用函数指针变量调用函数”，如果大家都和我一样，当初的教科书是谭浩强先生的《C 程序设计》的话，请找回书的第 9.5 节。

例子：用函数指针变量调用函数

```
#include <avr/io.h>
void fun1(void)
{
  unsigned char i=0;
  while(1)
  {
    PORTB=i++;
    PORTC=0x01<<(i%8);
  }
}
void (*pfun)();  //指向函数的指针

int main(void)
{

  pfun=fun1;    //
  (*pfun)();    //运行指针所指向的函数
}
```

第二种，是“把指向函数的指针变量作函数参数”

```
#include <avr/io.h>
void fun1(void)
{
  unsigned char i=0;
  while(1)
  {
    PORTB=i++;
    PORTC=0x01<<(i%8);
  }
}
```

```
void RunFun(void (*pfun)())  //获得了要传递的函数的地址
{
  (*pfun)();                //在 RunFun 中，运行指针所指向的函数
}

int main(void)
{
   RunFun(fun1);            //将函数的指针作为变量传递

}
```

看到上面的两种方式，很多人可能会说，“这的确不错”，但是这样与我们想要的 RTOS，有什么关系呢？各位请细心向下看。

以下是 GCC 对上面的代码的编译的情况：

对 main()中的 RunFun(fun1); 的编译如下

```
  ldi r24,lo8(pm(fun1))
  ldi r25,hi8(pm(fun1))
  rcall RunFun
```

对 void RunFun(void (*pfun)())的编译如下

```
                /*void RunFun(void (*pfun)())*/
               /*(*pfun)();*/
.LM6:
  movw r30,r24
  icall
  ret
```

在调用 void RunFun(void (*pfun)())的时候，的确可以把 fun1 的地址通过 r24 和 r25 传递给 RunFun()。但是，RT

OS 如何才能有效地利用函数的地址呢？

第二篇： 人工堆栈

在单片机的指令集中，一类指令是专门与堆栈和 PC 指针打道的，它们是

rcall 相对调用子程序指令

icall 间接调用子程序指令

ret 子程序返回指令

reti 中断返回指令

对于 ret 和 reti，它们都可以将堆栈栈顶的两个字节被弹出来送入程序计数器 PC 中，一般用来从子程序或中断中退出。其中 reti 还可以在退出中断时，重开全局中断使能。

有了这个基础，就可以建立我们的人工堆栈了。

例：

```
#include <avr/io.h>
void fun1(void)
{
  unsigned char i=0;
  while(1)
  {
    PORTB=i++;
    PORTC=0x01<<(i%8);
  }
}

unsigned char Stack[100]; //建立一个 100 字节的人工堆栈

void RunFunInNewStack(void (*pfun)(),unsigned char *pStack)
{
  *pStack--=(unsigned int)pfun>>8;    //将函数的地址高位压入堆栈，
  *pStack--=(unsigned int)pfun;       //将函数的地址低位压入堆栈，
  SP=pStack;                          //将堆栈指针指向人工堆栈的栈顶
  __asm__ __volatile__("RET \n\t");    //返回并开中断,开始运行 fun1()
```

```
}

int main(void)
{
  RunFunInNewStack(fun1,&Stack[99]);
}
```

RunFunInNewStack(),将指向函数的指针的值保存到一个 unsigned char 的数组 Stack 中，作为人工堆栈。并且将栈顶的数值传递组堆栈指针 SP,因此当用"ret"返回时，从 SP 中恢复到 PC 中的值，就变为了指向 fun1()的地址，开始运行 fun1().

上面例子中在 RunFunInNewStack()的最后一句嵌入了汇编代码 "ret",实际上是可以去除的。因为在 RunFunInNewStack()返回时，编译器已经会加上"ret"。我特意写出来，是为了让大家看到用"ret"作为返回后运行 fun1()的过程。

第三篇：GCC 中对寄存器的分配与使用

在很多用于 AVR 的 RTOS 中，都会有任务调度时，插入以下的语句：

入栈：

```
__asm__ __volatile__("PUSH R0  \n\t");
__asm__ __volatile__("PUSH R1  \n\t");
......
__asm__ __volatile__("PUSH R31 \n\t");
```

出栈

```
__asm__ __volatile__("POP  R31 \n\t");
......
__asm__ __volatile__("POP  R1  \n\t");
__asm__ __volatile__("POP  R0  \n\t");
```

通常大家都会认为，在任务调度开始时，当然要将所有的通用寄存器都保存，并且还应该保存程序状态寄存器 SREG。然后再根据相反的次序，将新任务的寄存器的内容恢复。

但是，事实真的是这样吗？如果大家看过陈明计先生写的 small rots51,就会发现，它所保存的通用寄存器不过是 4 组通用寄存器中的 1 组。

在 Win AVR 中的帮助文件 avr-libc Manual 中的 Related Pages 中的 Frequently Asked Questions,其实有一个问题是"What registers are used by the C compiler?" 回答了编译器所需要占用的寄存器。一般情况下，编译器会先用到以下寄存器

1 Call-used registers (r18-r27, r30-r31): 调用函数时作为参数传递，也就是用得最多的寄存器。

2 Call-saved registers (r2-r17, r28-r29): 调用函数时作为结果传递,当中的 r28 和 r29 可能会被作为指向堆栈上的变量的指针。

3 Fixed registers (r0, r1): 固定作用。r0 用于存放临时数据，r1 用于存放 0。

还有另一个问题是"How to permanently bind a variable to a register?",是将变量绑定到通用寄存器的方法。而且我发现，如果将某个寄存器定义为变量，编译器就会不将该寄存器分配作其它用途。这对 RTOS 是很重要的。

在"Inline Asm"中的"C Names Used in Assembler Code"明确表示,如果将太多的通用寄存器定义为变量，刚在编译的过程中，被定义的变量依然可能被编译器占用。

大家可以比较以下两个例子，看看编译器产生的代码：(在*.lst 文件中)

第一个例子：没有定义通用寄存器为变量

```
#include <avr/io.h>

unsigned char add(unsigned char b,unsigned char c,unsigned char d)
{
  return b+c*d;
}
```

```
int main(void)
{
  unsigned char a=0;
  while(1)
  {
    a++;
    PORTB=add(a,a,a);
  }
}
```

在本例中，"add(a,a,a);"被编译如下:

```
mov r20,r28
mov r22,r28
mov r24,r28
rcall add
```

第二个例子：定义通用寄存器为变量

```
#include <avr/io.h>

unsigned char add(unsigned char b,unsigned char c,unsigned char d)
{
   return b+c*d;
}


register unsigned char a asm("r20");  //将 r20 定义为 变量 a

int main(void)
{

    while(1)
```

```
  {
    a++;
      PORTB=add(a,a,a);
  }
}
```

在本例中，"add(a,a,a);"被编译如下:

```
mov r22,r20
mov r24,r20
rcall add
```

当然，在上面两个例子中，有部份代码被编译器优化了。

通过反复测试，发现编译器一般使用如下寄存器:

第 1 类寄存器，第 2 类寄存器的 r28,r29,第 3 类寄存器

如在中断函数中有调用基它函数，刚会在进入中断后，固定地将第 1 类寄存器和第 3 类寄存器入栈，在退出中断又将它们出栈。

第四篇：只有延时服务的协作式的内核

Cooperative Multitasking

前后台系统，协作式内核系统，与占先式内核系统，有什么不同呢？

记得在 21IC 上看过这样的比喻,“你(小工)在用厕所，经理在外面排第一，老板在外面排第二。如果是前后台，不管是谁，都必须按排队的次序使用厕所；如果是协作式，那么可以等你用完厕所，老板就要比经理先进入；如果是占先式，只要有更高级的人在外面等，那么厕所里无论是谁，都要第一时间让出来，让最高级别的人先用。”

```
#include <avr/io.h>
```

```
#include <avr/Interrupt.h>
#include <avr/signal.h>
unsigned char Stack[200];

register unsigned char OSRdyTbl          asm("r2");    //任务运行就绪表
register unsigned char OSTaskRunningPrio asm("r3");    //正在运行的任务

#define OS_TASKS 3                    //设定运行任务的数量
struct TaskCtrBlock           //任务控制块
{
  unsigned int OSTaskStackTop;  //保存任务的堆栈顶
  unsigned int OSWaitTick;      //任务延时时钟
} TCB[OS_TASKS+1];

//防止被编译器占用
register unsigned char tempR4  asm("r4");
register unsigned char tempR5  asm("r5");
register unsigned char tempR6  asm("r6");
register unsigned char tempR7  asm("r7");
register unsigned char tempR8  asm("r8");
register unsigned char tempR9  asm("r9");
register unsigned char tempR10 asm("r10");
register unsigned char tempR11 asm("r11");
register unsigned char tempR12 asm("r12");
register unsigned char tempR13 asm("r13");
register unsigned char tempR14 asm("r14");
register unsigned char tempR15 asm("r15");
register unsigned char tempR16 asm("r16");
register unsigned char tempR16 asm("r17");



//建立任务
```

```
void OSTaskCreate(void (*Task)(void),unsigned char *Stack,unsigned char TaskID)
{
  unsigned char i;
  *Stack--=(unsigned int)Task>>8;    //将任务的地址高位压入堆栈，
  *Stack--=(unsigned int)Task;        //将任务的地址低位压入堆栈，

  *Stack--=0x00;                    //R1 __zero_reg__
  *Stack--=0x00;                    //R0 __tmp_reg__
  *Stack--=0x80;                                //SREG 在任务中，开启全局中断
  for(i=0;i<14;i++)    //在 avr-libc 中的 FAQ 中的 What registers are used by the C compiler?
    *Stack--=i;                    //描述了寄存器的作用
    TCB[TaskID].OSTaskStackTop=(unsigned int)Stack;    //将人工堆栈的栈顶，保存到堆栈的数组中
    OSRdyTbl|=0x01<<TaskID;      //任务就绪表已经准备好
}

//开始任务调度,从最低优先级的任务的开始
void OSStartTask()
{
  OSTaskRunningPrio=OS_TASKS;
  SP=TCB[OS_TASKS].OSTaskStackTop+17;
  __asm__ __volatile__(    "reti"       "\n\t"  );
}

//进行任务调度
void OSSched(void)
{
   //  根据中断时保存寄存器的次序入栈，模拟一次中断后，入栈的情况
  __asm__ __volatile__("PUSH __zero_reg__         \n\t");  //R1
  __asm__ __volatile__("PUSH __tmp_reg__          \n\t");  //R0
  __asm__ __volatile__("IN   __tmp_reg__,__SREG__ \n\t");  //保存状态寄存器 SREG
  __asm__ __volatile__("PUSH __tmp_reg__          \n\t");
  __asm__ __volatile__("CLR  __zero_reg__         \n\t");  //R0 重新清零
```

```
__asm__ __volatile__("PUSH R18                \n\t");
__asm__ __volatile__("PUSH R19                \n\t");
__asm__ __volatile__("PUSH R20                \n\t");
__asm__ __volatile__("PUSH R21                \n\t");
__asm__ __volatile__("PUSH R22                \n\t");
__asm__ __volatile__("PUSH R23                \n\t");
__asm__ __volatile__("PUSH R24                \n\t");
__asm__ __volatile__("PUSH R25                \n\t");
__asm__ __volatile__("PUSH R26                \n\t");
__asm__ __volatile__("PUSH R27                \n\t");
__asm__ __volatile__("PUSH R30                \n\t");
__asm__ __volatile__("PUSH R31                \n\t");
__asm__ __volatile__("PUSH R28                \n\t");  //R28 与 R29 用于建立在堆栈上的指针
__asm__ __volatile__("PUSH R29                \n\t");  //入栈完成

TCB[OSTaskRunningPrio].OSTaskStackTop=SP;          //将正在运行的任务的堆栈底保存



unsigned char OSNextTaskID;                          //在现有堆栈上开设新的空间
for (OSNextTaskID = 0;                             //进行任务调度
  OSNextTaskID < OS_TASKS && !(OSRdyTbl & (0x01<<OSNextTaskID));
  OSNextTaskID++);
  OSTaskRunningPrio = OSNextTaskID ;

cli();  //保护堆栈转换
SP=TCB[OSTaskRunningPrio].OSTaskStackTop;
sei();

  //根据中断时的出栈次序
__asm__ __volatile__("POP  R29                \n\t");
__asm__ __volatile__("POP  R28                \n\t");
__asm__ __volatile__("POP  R31                \n\t");
```

```
  __asm__ __volatile__("POP  R30                \n\t");
  __asm__ __volatile__("POP  R27                \n\t");
  __asm__ __volatile__("POP  R26                \n\t");
  __asm__ __volatile__("POP  R25                \n\t");
  __asm__ __volatile__("POP  R24                \n\t");
  __asm__ __volatile__("POP  R23                \n\t");
  __asm__ __volatile__("POP  R22                \n\t");
  __asm__ __volatile__("POP  R21                \n\t");
  __asm__ __volatile__("POP  R20                \n\t");
  __asm__ __volatile__("POP  R19                \n\t");
  __asm__ __volatile__("POP  R18                \n\t");
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //SERG 出栈并恢复
  __asm__ __volatile__("OUT  __SREG__,__tmp_reg__ \n\t");     //
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //R0 出栈
  __asm__ __volatile__("POP  __zero_reg__       \n\t");     //R1 出栈
  //中断时出栈完成
}

void OSTimeDly(unsigned int ticks)
{
  if(ticks)                          //当延时有效
  {
    OSRdyTbl &= ~(0x01<<OSTaskRunningPrio);
    TCB[OSTaskRunningPrio].OSWaitTick=ticks;
    OSSched();                       //从新调度
  }
}


void TCN0Init(void)    // 计时器 0
{
  TCCR0 = 0;
```

```
  TCCR0 |= (1<<CS02);  // 256 预分频
  TIMSK |= (1<<TOIE0); // T0 溢出中断允许
  TCNT0 = 100;         // 置计数起始值

}



SIGNAL(SIG_OVERFLOW0)
{
  unsigned char i;
  for(i=0;i<OS_TASKS;i++)       //任务时钟
  {
    if(TCB[i].OSWaitTick)
    {
      TCB[i].OSWaitTick--;
      if(TCB[i].OSWaitTick==0)     //当任务时钟到时,必须是由定时器减时的才行
      {
        OSRdyTbl |= (0x01<<i);     //使任务在就绪表中置位
      }
    }
  }
  TCNT0=100;
}

void Task0()
{
  unsigned int j=0;
  while(1)
  {
    PORTB=j++;
    OSTimeDly(2);
  }
```

```
}

void Task1()
{
  unsigned int j=0;
  while(1)
  {
    PORTC=j++;
    OSTimeDly(4);
  }
}

void Task2()
{
  unsigned int j=0;
  while(1)
  {
    PORTD=j++;
    OSTimeDly(8);
  }
}




void TaskScheduler()
{
  while(1)
  {
     OSSched();      //反复进行调度
  }
}
```

```
int main(void)
{
  TCN0Init();
  OSRdyTbl=0;
  OSTaskRunningPrio=0;
  OSTaskCreate(Task0,&Stack[49],0);
  OSTaskCreate(Task1,&Stack[99],1);
  OSTaskCreate(Task2,&Stack[149],2);
  OSTaskCreate(TaskScheduler,&Stack[199],OS_TASKS);
  OSStartTask();
}
```

在上面的例子中，一切变得很简单，三个正在运行的主任务，都通过延时服务，主动放弃对 CPU 的控制权。

在时间中断中，对各个任务的的延时进行计时，如果某个任务的延时结束，将任务重新在就绪表中置位。

最低级的系统任务 TaskScheduler()，在三个主任务在放弃对 CPU 的控制权后开始不断地进行调度。如果某个任务在就绪表中置位，通过调度，进入最高级别的任务中继续运行。

第五篇： 完善的协作式的内核

现在为上面的协作式内核添加一些 OS 中所必须的服务：

1 挂起和重新运行任务

2 信号量(在必要时候，可以扩展成邮箱和信息队列)

3 延时

```
#include <avr/io.h>
#include <avr/Interrupt.h>
#include <avr/signal.h>
unsigned char Stack[400];

register unsigned char OSRdyTbl          asm("r2");    //任务运行就绪表
```

```
register unsigned char OSTaskRunningPrio asm("r3");    //正在运行的任务

#define OS_TASKS 3                    //设定运行任务的数量
struct TaskCtrBlock
{
  unsigned int OSTaskStackTop;  //保存任务的堆栈顶
  unsigned int OSWaitTick;      //任务延时时钟
} TCB[OS_TASKS+1];

//防止被编译器占用
register unsigned char tempR4  asm("r4");
register unsigned char tempR5  asm("r5");
register unsigned char tempR6  asm("r6");
register unsigned char tempR7  asm("r7");
register unsigned char tempR8  asm("r8");
register unsigned char tempR9  asm("r9");
register unsigned char tempR10 asm("r10");
register unsigned char tempR11 asm("r11");
register unsigned char tempR12 asm("r12");
register unsigned char tempR13 asm("r13");
register unsigned char tempR14 asm("r14");
register unsigned char tempR15 asm("r15");
register unsigned char tempR16 asm("r16");
register unsigned char tempR16 asm("r17");


//建立任务
void OSTaskCreate(void (*Task)(void),unsigned char *Stack,unsigned char TaskID)
{
  unsigned char i;
  *Stack--=(unsigned int)Task>>8;    //将任务的地址高位压入堆栈，
  *Stack--=(unsigned int)Task;        //将任务的地址低位压入堆栈，
```

```
  *Stack--=0x00;                    //R1 __zero_reg__
  *Stack--=0x00;                    //R0 __tmp_reg__
  *Stack--=0x80;

//SREG 在任务中，开启全局中断
  for(i=0;i<14;i++)    //在 avr-libc 中的 FAQ 中的 What registers are used by the C compiler?
    *Stack--=i;                    //描述了寄存器的作用
  TCB[TaskID].OSTaskStackTop=(unsigned int)Stack;    //将人工堆栈的栈顶，保存到堆栈的数组中
  OSRdyTbl|=0x01<<TaskID;      //任务就绪表已经准备好
}

//开始任务调度,从最低优先级的任务的开始
void OSStartTask()
{
  OSTaskRunningPrio=OS_TASKS;
  SP=TCB[OS_TASKS].OSTaskStackTop+17;
  __asm__ __volatile__(    "reti"       "\n\t"  );
}

//进行任务调度
void OSSched(void)
{
   //  根据中断时保存寄存器的次序入栈，模拟一次中断后，入栈的情况
  __asm__ __volatile__("PUSH __zero_reg__         \n\t");  //R1
  __asm__ __volatile__("PUSH __tmp_reg__          \n\t");  //R0
  __asm__ __volatile__("IN   __tmp_reg__,__SREG__ \n\t");  //保存状态寄存器 SREG
  __asm__ __volatile__("PUSH __tmp_reg__          \n\t");
  __asm__ __volatile__("CLR  __zero_reg__         \n\t");  //R0 重新清零
  __asm__ __volatile__("PUSH R18                  \n\t");
  __asm__ __volatile__("PUSH R19                  \n\t");
  __asm__ __volatile__("PUSH R20                  \n\t");
```

```
__asm__ __volatile__("PUSH R21                \n\t");
__asm__ __volatile__("PUSH R22                \n\t");
__asm__ __volatile__("PUSH R23                \n\t");
__asm__ __volatile__("PUSH R24                \n\t");
__asm__ __volatile__("PUSH R25                \n\t");
__asm__ __volatile__("PUSH R26                \n\t");
__asm__ __volatile__("PUSH R27                \n\t");
__asm__ __volatile__("PUSH R30                \n\t");
__asm__ __volatile__("PUSH R31                \n\t");
__asm__ __volatile__("PUSH R28                \n\t");  //R28 与 R29 用于建立在堆栈上的指针
__asm__ __volatile__("PUSH R29                \n\t");  //入栈完成

TCB[OSTaskRunningPrio].OSTaskStackTop=SP;          //将正在运行的任务的堆栈底保存

unsigned char OSNextTaskID;                          //在现有堆栈上开设新的空间
for (OSNextTaskID = 0;                             //进行任务调度
  OSNextTaskID < OS_TASKS && !(OSRdyTbl & (0x01<<OSNextTaskID));
  OSNextTaskID++);
  OSTaskRunningPrio = OSNextTaskID ;

cli();  //保护堆栈转换
SP=TCB[OSTaskRunningPrio].OSTaskStackTop;
sei();

  //根据中断时的出栈次序
__asm__ __volatile__("POP  R29                \n\t");
__asm__ __volatile__("POP  R28                \n\t");
__asm__ __volatile__("POP  R31                \n\t");
__asm__ __volatile__("POP  R30                \n\t");
__asm__ __volatile__("POP  R27                \n\t");
__asm__ __volatile__("POP  R26                \n\t");
__asm__ __volatile__("POP  R25                \n\t");
```

```
  __asm__ __volatile__("POP  R24                \n\t");
  __asm__ __volatile__("POP  R23                \n\t");
  __asm__ __volatile__("POP  R22                \n\t");
  __asm__ __volatile__("POP  R21                \n\t");
  __asm__ __volatile__("POP  R20                \n\t");
  __asm__ __volatile__("POP  R19                \n\t");
  __asm__ __volatile__("POP  R18                \n\t");
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //SERG 出栈并恢复
  __asm__ __volatile__("OUT  __SREG__,__tmp_reg__ \n\t");     //
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //R0 出栈
  __asm__ __volatile__("POP  __zero_reg__       \n\t");     //R1 出栈
  //中断时出栈完成
}



//////////////////////////////////////////////任务处理
//挂起任务
void OSTaskSuspend(unsigned char prio)
{
  TCB[prio].OSWaitTick=0;
  OSRdyTbl &= ~(0x01<<prio); //从任务就绪表上去除标志位
  if(OSTaskRunningPrio==prio)  //当要挂起的任务为当前任务
    OSSched();              //从新调度
}

//恢复任务 可以让被 OSTaskSuspend 或 OSTimeDly 暂停的任务恢复
void OSTaskResume(unsigned char prio)
{
  OSRdyTbl |= 0x01<<prio;    //从任务就绪表上重置标志位
    TCB[prio].OSWaitTick=0;        //将时间计时设为 0,到时
  if(OSTaskRunningPrio>prio)   //当要当前任务的优先级低于重置位的任务的优先级
    OSSched();              //从新调度              //从新调度
```

```
}

// 任务延时
void OSTimeDly(unsigned int ticks)
{
  if(ticks)                          //当延时有效
  {
    OSRdyTbl &= ~(0x01<<OSTaskRunningPrio);
    TCB[OSTaskRunningPrio].OSWaitTick=ticks;
    OSSched();                       //从新调度
  }
}



//信号量
struct SemBlk
{
  unsigned char OSEventType;     //型号 0,信号量独占型；1 信号量共享型
  unsigned char OSEventState;    //状态 0,不可用;1,可用
  unsigned char OSTaskPendTbl;   //等待信号量的任务列表
} Sem[10];

//初始化信号量
void OSSemCreat(unsigned char Index,unsigned char Type)
{
  Sem[Index].OSEventType=Type;  //型号 0,信号量独占型；1 信号量共享型
  Sem[Index].OSTaskPendTbl=0;
  Sem[Index].OSEventState=0;
}

//任务等待信号量,挂起
unsigned char OSTaskSemPend(unsigned char Index,unsigned int Timeout)
```

```
{

  //unsigned char i=0;
  if(Sem[Index].OSEventState)              //信号量有效
  {
    if(Sem[Index].OSEventType==0)          //如果为独占型
    Sem[Index].OSEventState = 0x00;        //信号量被独占，不可用
  }
  else
  {                                        //加入信号的任务等待表
    Sem[Index].OSTaskPendTbl |= 0x01<<OSTaskRunningPrio;
    OSRdyTbl &= ~(0x01<<OSTaskRunningPrio);  //从任务就绪表中去除
    TCB[OSTaskRunningPrio].OSWaitTick=Timeout;   //如延时为 0，刚无限等待
    OSSched();  //从新调度
    if(TCB[OSTaskRunningPrio].OSWaitTick==0) return 0;
  }
  return 1;
}




//发送一个信号量，可以从任务或中断发送
void OSSemPost(unsigned char Index)
{
if(Sem[Index].OSEventType)               //当要求的信号量是共享型
  {
    Sem[Index].OSEventState=0x01;          //使信号量有效
    OSRdyTbl |=Sem [Index].OSTaskPendTbl;  //使在等待该信号的所有任务就绪
    Sem[Index].OSTaskPendTbl=0;            //清空所有等待该信号的等待任务
  }
  else                                     //当要求的信号量为独占型
  {
```

```
    unsigned char i;
    for (i = 0; i < OS_TASKS && !(Sem[Index].OSTaskPendTbl & (0x01<<i)); i++);
    if(i < OS_TASKS)                    //如果有任务需要
    {
      Sem[Index].OSTaskPendTbl &= ~(0x01<<i); //从等待表中去除
      OSRdyTbl |= 0x01<<i;                  //任务就绪
    }
    else
    {
      Sem[Index].OSEventState =1;       //使信号量有效
    }
  }
}

//从任务发送一个信号量，并进行调度
void OSTaskSemPost(unsigned char Index)
{
  OSSemPost(Index);
  OSSched();
}

//清除一个信号量,只对共享型的有用。
//对于独占型的信号量，在任务占用后，就交得不可以用了。

void OSSemClean(unsigned char Index)
{
  Sem[Index].OSEventState =0;         //要求的信号量无效
}



void TCN0Init(void)    // 计时器 0
{
```

```
  TCCR0 = 0;
  TCCR0 |= (1<<CS02);  // 256 预分频
  TIMSK |= (1<<TOIE0); // T0 溢出中断允许
  TCNT0 = 100;         // 置计数起始值

}



SIGNAL(SIG_OVERFLOW0)
{
  unsigned char i;
  for(i=0;i<OS_TASKS;i++)       //任务时钟
  {
    if(TCB[i].OSWaitTick)
    {
      TCB[i].OSWaitTick--;
      if(TCB[i].OSWaitTick==0)     //当任务时钟到时,必须是由定时器减时的才行
      {
        OSRdyTbl |= (0x01<<i);     //使任务在就绪表中置位
      }
    }
  }
  TCNT0=100;
}

void Task0()
{
  unsigned int j=0;
  while(1)
  {
    PORTB=j++;
    OSTaskSuspend(1);    //挂起任务 1
```

```
    OSTaskSemPost(0);
    OSTimeDly(50);
    OSTaskResume(1);     //恢复任务 1
    OSSemClean(0);
    OSTimeDly(50);
  }
}

void Task1()
{
  unsigned int j=0;
  while(1)
  {
    PORTC=j++;
    OSTimeDly(5);
  }
}

void Task2()
{
  unsigned int j=0;
  while(1)
  {
      OSTaskSemPend(0,10);
    PORTD=j++;
    OSTimeDly(5);
  }
}




void TaskScheduler()
```

```
{
  while(1)
  {
     OSSched();      //反复进行调度
  }
}



int main(void)
{
  TCN0Init();
  OSRdyTbl=0;
  OSSemCreat(0,1);  //将信号量设为共享型
  OSTaskCreate(Task0,&Stack[99],0);
  OSTaskCreate(Task1,&Stack[199],1);
  OSTaskCreate(Task2,&Stack[299],2);
  OSTaskCreate(TaskScheduler,&Stack[399],OS_TASKS);
  OSStartTask();
}
```

第六篇:时间片轮番调度法的内核

Round-Robin Sheduling

时间片轮调法是非常有趣的。本篇中的例子，建立了 3 个任务，任务没有优先级，在时间中断的调度下，每个任务都轮流运行相同的时间。如果在内核中没有加入其它服务，感觉上就好像是有三个大循环在同时运行。

本例只是提供了一个用时间中断进行调度的内核，大家可以根据自己的需要，添加相应的服务。

要注意到:

1,由于在时间中断内调用了任务切换函数，因为在进入中断时，已经将一系列的寄存器入栈。

2,在中断内进行调度，是直接通过"RJMP Int_OSSched"进入任务切换和调度的，这是 GCC AVR 的一个特点，为用 C 编写内核提供了极大的方便。

3,在阅读代码的同时，请对照阅读编译器产生的 *.lst 文件，会对你理解例子有很大的帮助。

```
#include <avr/io.h>
#include <avr/Interrupt.h>
#include <avr/signal.h>
unsigned char Stack[400];

register unsigned char OSRdyTbl          asm("r2");    //任务运行就绪表
register unsigned char OSTaskRunningPrio asm("r3");    //正在运行的任务

#define OS_TASKS 3                       //设定运行任务的数量
struct TaskCtrBlock
{
  unsigned int OSTaskStackTop;  //保存任务的堆栈顶
  unsigned int OSWaitTick;      //任务延时时钟
} TCB[OS_TASKS+1];

//防止被编译器占用
register unsigned char tempR4  asm("r4");
register unsigned char tempR5  asm("r5");
register unsigned char tempR6  asm("r6");
register unsigned char tempR7  asm("r7");
register unsigned char tempR8  asm("r8");
register unsigned char tempR9  asm("r9");
register unsigned char tempR10 asm("r10");
register unsigned char tempR11 asm("r11");
register unsigned char tempR12 asm("r12");
register unsigned char tempR13 asm("r13");
register unsigned char tempR14 asm("r14");
register unsigned char tempR15 asm("r15");
register unsigned char tempR16 asm("r16");
register unsigned char tempR16 asm("r17");
```

```
//建立任务
void OSTaskCreate(void (*Task)(void),unsigned char *Stack,unsigned char TaskID)
{
  unsigned char i;
  *Stack--=(unsigned int)Task>>8;    //将任务的地址高位压入堆栈，
  *Stack--=(unsigned int)Task;        //将任务的地址低位压入堆栈，

  *Stack--=0x00;                    //R1 __zero_reg__
  *Stack--=0x00;                    //R0 __tmp_reg__
  *Stack--=0x80;

//SREG 在任务中，开启全局中断
  for(i=0;i<14;i++)    //在 avr-libc 中的 FAQ 中的 What registers are used by the C compiler?
    *Stack--=i;                    //描述了寄存器的作用
  TCB[TaskID].OSTaskStackTop=(unsigned int)Stack;    //将人工堆栈的栈顶，保存到堆栈的数组中
  OSRdyTbl|=0x01<<TaskID;      //任务就绪表已经准备好
}

//开始任务调度,从最低优先级的任务的开始
void OSStartTask()
{
  OSTaskRunningPrio=OS_TASKS;
  SP=TCB[OS_TASKS].OSTaskStackTop+17;
  __asm__ __volatile__(    "reti"       "\n\t"  );
}

//进行任务调度
void OSSched(void)
{
   //  根据中断时保存寄存器的次序入栈，模拟一次中断后，入栈的情况
  __asm__ __volatile__("PUSH __zero_reg__         \n\t");  //R1
```

```
__asm__ __volatile__("PUSH __tmp_reg__          \n\t");  //R0
__asm__ __volatile__("IN   __tmp_reg__,__SREG__ \n\t");  //保存状态寄存器 SREG
__asm__ __volatile__("PUSH __tmp_reg__          \n\t");
__asm__ __volatile__("CLR  __zero_reg__         \n\t");  //R0 重新清零
__asm__ __volatile__("PUSH R18                  \n\t");
__asm__ __volatile__("PUSH R19                  \n\t");
__asm__ __volatile__("PUSH R20                  \n\t");
__asm__ __volatile__("PUSH R21                  \n\t");
__asm__ __volatile__("PUSH R22                  \n\t");
__asm__ __volatile__("PUSH R23                  \n\t");
__asm__ __volatile__("PUSH R24                  \n\t");
__asm__ __volatile__("PUSH R25                  \n\t");
__asm__ __volatile__("PUSH R26                  \n\t");
__asm__ __volatile__("PUSH R27                  \n\t");
__asm__ __volatile__("PUSH R30                  \n\t");
__asm__ __volatile__("PUSH R31                  \n\t");

__asm__ __volatile__("Int_OSSched:              \n\t");  //当中断要求调度，直接进入这里
__asm__ __volatile__("PUSH R28                  \n\t");  //R28 与 R29 用于建立在堆栈上的指针
__asm__ __volatile__("PUSH R29                  \n\t");  //入栈完成

TCB[OSTaskRunningPrio].OSTaskStackTop=SP;          //将正在运行的任务的堆栈底保存

if(++OSTaskRunningPrio>=OS_TASKS) //轮流运行各个任务，没有优先级
   OSTaskRunningPrio=0;

//cli();  //保护堆栈转换
SP=TCB[OSTaskRunningPrio].OSTaskStackTop;
//sei();

  //根据中断时的出栈次序
__asm__ __volatile__("POP  R29                  \n\t");
```

```
  __asm__ __volatile__("POP  R28                \n\t");
  __asm__ __volatile__("POP  R31                \n\t");
  __asm__ __volatile__("POP  R30                \n\t");
  __asm__ __volatile__("POP  R27                \n\t");
  __asm__ __volatile__("POP  R26                \n\t");
  __asm__ __volatile__("POP  R25                \n\t");
  __asm__ __volatile__("POP  R24                \n\t");
  __asm__ __volatile__("POP  R23                \n\t");
  __asm__ __volatile__("POP  R22                \n\t");
  __asm__ __volatile__("POP  R21                \n\t");
  __asm__ __volatile__("POP  R20                \n\t");
  __asm__ __volatile__("POP  R19                \n\t");
  __asm__ __volatile__("POP  R18                \n\t");
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //SERG 出栈并恢复
  __asm__ __volatile__("OUT  __SREG__,__tmp_reg__ \n\t");     //
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //R0 出栈
  __asm__ __volatile__("POP  __zero_reg__       \n\t");     //R1 出栈
  __asm__ __volatile__("RETI                    \n\t");     //返回并开中断
  //中断时出栈完成
}



void IntSwitch(void)
{
  __asm__ __volatile__("POP  R31                \n\t");  //去除因调用子程序而入栈的 PC
  __asm__ __volatile__("POP  R31                \n\t");
  __asm__ __volatile__("RJMP Int_OSSched        \n\t");  //重新调度
}
```

```
void TCN0Init(void)    // 计时器 0
{
  TCCR0 = 0;
  TCCR0 |= (1<<CS02);  // 256 预分频
  TIMSK |= (1<<TOIE0); // T0 溢出中断允许
  TCNT0 = 100;         // 置计数起始值
}



SIGNAL(SIG_OVERFLOW0)
{
  TCNT0=100;
  IntSwitch();        //任务调度
}

void Task0()
{
  unsigned int j=0;
  while(1)
  {
    PORTB=j++;
    //OSTimeDly(50);
  }
}

void Task1()
{
  unsigned int j=0;
  while(1)
  {
    PORTC=j++;
    //OSTimeDly(5);
```

```
  }
}

void Task2()
{
  unsigned int j=0;
  while(1)
  {
    PORTD=j++;
    //OSTimeDly(5);
  }
}




void TaskScheduler()
{
  while(1)
  {
     OSSched();      //反复进行调度
  }
}



int main(void)
{
  TCN0Init();
  OSRdyTbl=0;
  OSTaskCreate(Task0,&Stack[99],0);
  OSTaskCreate(Task1,&Stack[199],1);
  OSTaskCreate(Task2,&Stack[299],2);
  OSTaskCreate(TaskScheduler,&Stack[399],OS_TASKS);
```

```
  OSStartTask();
}
```

第七篇:占先式内核(只带延时服务)

Preemptive Multitasking

当大家理解时间片轮番调度法的任务调度方式后，占先式的内核的原理，已经伸手可及了。

先想想，占先式内核是在什么地方实现任务调度的呢？对了，它在可以在任务中进行调度，这个在协作式的内核中已经做到了；同时，它也可以在中断结束后进行调度，这个问题，已经在时间片轮番调度法中已经做到了。

由于中断是可以嵌套的，只有当各层嵌套中要求调度，并且中断嵌套返回到最初进入的中断的那一层时，才能进行任务调度。

```
#include <avr/io.h>
#include <avr/Interrupt.h>
#include <avr/signal.h>
unsigned char Stack[400];

register unsigned char OSRdyTbl          asm("r2");    //任务运行就绪表
register unsigned char OSTaskRunningPrio asm("r3");    //正在运行的任务
register unsigned char IntNum            asm("r4");  //中断嵌套计数器
//只有当中断嵌套数为 0，并且有中断要求时，才能在退出中断时，进行任务调度
register unsigned char OSCoreState       asm("r16"); // 系统核心标志位 ,R16 编译器没有使用
//只有大于 R15 的寄存器才能直接赋值 例 LDI R16,0x01
//0x01 正在任务 切换  0x02 有中断要求切换

#define OS_TASKS 3                    //设定运行任务的数量
struct TaskCtrBlock
{
  unsigned int OSTaskStackTop;  //保存任务的堆栈顶
  unsigned int OSWaitTick;      //任务延时时钟
} TCB[OS_TASKS+1];
```

```
//防止被编译器占用
//register unsigned char tempR4  asm("r4");
register unsigned char tempR5  asm("r5");
register unsigned char tempR6  asm("r6");
register unsigned char tempR7  asm("r7");
register unsigned char tempR8  asm("r8");
register unsigned char tempR9  asm("r9");
register unsigned char tempR10 asm("r10");
register unsigned char tempR11 asm("r11");
register unsigned char tempR12 asm("r12");
register unsigned char tempR13 asm("r13");
register unsigned char tempR14 asm("r14");
register unsigned char tempR15 asm("r15");
//register unsigned char tempR16 asm("r16");
register unsigned char tempR16 asm("r17");



//建立任务
void OSTaskCreate(void (*Task)(void),unsigned char *Stack,unsigned char TaskID)
{
  unsigned char i;
  *Stack--=(unsigned int)Task>>8;    //将任务的地址高位压入堆栈，
  *Stack--=(unsigned int)Task;        //将任务的地址低位压入堆栈，

  *Stack--=0x00;                    //R1 __zero_reg__
  *Stack--=0x00;                    //R0 __tmp_reg__
  *Stack--=0x80;

//SREG 在任务中，开启全局中断
  for(i=0;i<14;i++)    //在 avr-libc 中的 FAQ 中的 What registers are used by the C compiler?
    *Stack--=i;                    //描述了寄存器的作用
  TCB[TaskID].OSTaskStackTop=(unsigned int)Stack;    //将人工堆栈的栈顶，保存到堆栈的数组中
```

```
  OSRdyTbl|=0x01<<TaskID;      //任务就绪表已经准备好
}

//开始任务调度,从最低优先级的任务的开始
void OSStartTask()
{
  OSTaskRunningPrio=OS_TASKS;
  SP=TCB[OS_TASKS].OSTaskStackTop+17;
  __asm__ __volatile__(    "reti"       "\n\t"  );
}

//进行任务调度
void OSSched(void)
{

  __asm__ __volatile__("LDI  R16,0x01            \n\t");
  //清除中断要求任务切换的标志位,设置正在任务切换标志位
  __asm__ __volatile__("SEI                      \n\t");
  //开中断,因为如果因中断在任务调度中进行,要重新进行调度时，已经关中断
  //根据中断时保存寄存器的次序入栈，模拟一次中断后，入栈的情况
  __asm__ __volatile__("PUSH __zero_reg__        \n\t");  //R1
  __asm__ __volatile__("PUSH __tmp_reg__         \n\t");  //R0
  __asm__ __volatile__("IN   __tmp_reg__,__SREG__ \n\t");  //保存状态寄存器 SREG
  __asm__ __volatile__("PUSH __tmp_reg__         \n\t");
  __asm__ __volatile__("CLR  __zero_reg__        \n\t");  //R0 重新清零
  __asm__ __volatile__("PUSH R18                 \n\t");
  __asm__ __volatile__("PUSH R19                 \n\t");
  __asm__ __volatile__("PUSH R20                 \n\t");
  __asm__ __volatile__("PUSH R21                 \n\t");
  __asm__ __volatile__("PUSH R22                 \n\t");
  __asm__ __volatile__("PUSH R23                 \n\t");
  __asm__ __volatile__("PUSH R24                 \n\t");
```

```
  __asm__ __volatile__("PUSH R25                \n\t");
  __asm__ __volatile__("PUSH R26                \n\t");
  __asm__ __volatile__("PUSH R27                \n\t");
  __asm__ __volatile__("PUSH R30                \n\t");
  __asm__ __volatile__("PUSH R31                \n\t");

  __asm__ __volatile__("Int_OSSched:            \n\t");  //当中断要求调度，直接进入这里
  __asm__ __volatile__("SEI                     \n\t");
//开中断,因为如果因中断在任务调度中进行，已经关中断
  __asm__ __volatile__("PUSH R28                \n\t");  //R28 与 R29 用于建立在堆栈上的指针
  __asm__ __volatile__("PUSH R29                \n\t");  //入栈完成

  TCB[OSTaskRunningPrio].OSTaskStackTop=SP;          //将正在运行的任务的堆栈底保存

  unsigned char OSNextTaskPrio;                      //在现有堆栈上开设新的空间
  for (OSNextTaskPrio = 0;                           //进行任务调度
    OSNextTaskPrio < OS_TASKS && !(OSRdyTbl & (0x01<<OSNextTaskPrio));
    OSNextTaskPrio++);
    OSTaskRunningPrio = OSNextTaskPrio ;

  cli();  //保护堆栈转换
  SP=TCB[OSTaskRunningPrio].OSTaskStackTop;
  sei();

  //根据中断时的出栈次序
  __asm__ __volatile__("POP  R29                \n\t");
  __asm__ __volatile__("POP  R28                \n\t");
  __asm__ __volatile__("POP  R31                \n\t");
  __asm__ __volatile__("POP  R30                \n\t");
  __asm__ __volatile__("POP  R27                \n\t");
  __asm__ __volatile__("POP  R26                \n\t");
  __asm__ __volatile__("POP  R25                \n\t");
```

```
  __asm__ __volatile__("POP  R24                \n\t");
  __asm__ __volatile__("POP  R23                \n\t");
  __asm__ __volatile__("POP  R22                \n\t");
  __asm__ __volatile__("POP  R21                \n\t");
  __asm__ __volatile__("POP  R20                \n\t");
  __asm__ __volatile__("POP  R19                \n\t");
  __asm__ __volatile__("POP  R18                \n\t");
  __asm__ __volatile__("POP  __tmp_reg__          \n\t");     //SERG 出栈并恢复
  __asm__ __volatile__("OUT  __SREG__,__tmp_reg__ \n\t");     //
  __asm__ __volatile__("POP  __tmp_reg__          \n\t");     //R0 出栈
  __asm__ __volatile__("POP  __zero_reg__         \n\t");     //R1 出栈
  //中断时出栈完成
  __asm__ __volatile__("CLI                     \n\t");  //关中断
  __asm__ __volatile__("SBRC R16,1              \n\t");  //SBRC 当寄存器位为 0 刚跳过下一条指令
  //检查是在调度时，是否有中断要求任务调度 0x02 是中断要求调度的标志位
  __asm__ __volatile__("RJMP OSSched            \n\t");  //重新调度
  __asm__ __volatile__("LDI  R16,0x00           \n\t");
  //清除中断要求任务切换的标志位,清除正在任务切换标志位
  __asm__ __volatile__("RETI                    \n\t");     //返回并开中断
}



//从中断退出并进行调度
void IntSwitch(void)
{
  //当中断无嵌套，并且没有在切换任务的过程中，直接进行任务切换
  if(OSCoreState == 0x02 && IntNum==0)
  {
    //进入中断时，已经保存了 SREG 和 R0,R1,R18~R27,R30,R31
    __asm__ __volatile__("POP  R31                \n\t");  //去除因调用子程序而入栈的 PC
    __asm__ __volatile__("POP  R31                \n\t");
    __asm__ __volatile__("LDI  R16,0x01           \n\t");
```

```
    //清除中断要求任务切换的标志位,设置正在任务切换标志位
    __asm__ __volatile__("RJMP Int_OSSched          \n\t");  //重新调度
  }
}

// 任务延时
void OSTimeDly(unsigned int ticks)
{
  if(ticks)                          //当延时有效
  {
    OSRdyTbl &= ~(0x01<<OSTaskRunningPrio);
    TCB[OSTaskRunningPrio].OSWaitTick=ticks;
    OSSched();                       //从新调度
  }
}




void TCN0Init(void)    // 计时器 0
{
  TCCR0 = 0;
  TCCR0 |= (1<<CS02);  // 256 预分频
  TIMSK |= (1<<TOIE0); // T0 溢出中断允许
  TCNT0 = 100;         // 置计数起始值

}

SIGNAL(SIG_OVERFLOW0)
{
  IntNum++;     //中断嵌套+1
  sei();  //在中断中，重开中断
```

```
  unsigned char i,j=0;
  for(i=0;i<OS_TASKS;i++)        //任务时钟
  {
    if(TCB[i].OSWaitTick)
    {
      TCB[i].OSWaitTick--;
      if(TCB[i].OSWaitTick==0)        //当任务时钟到时,必须是由定时器减时的才行
      {
        OSRdyTbl |= (0x01<<i);        //使任务可以重新运行
        OSCoreState|=0x02;            //要求任务切换的标志位
      }
    }
  }
  TCNT0=100;
  cli();
  IntNum--;              //中断嵌套-1
  IntSwitch();        //进行任务调度
}

void Task0()
{
  unsigned int j=0;
  while(1)
  {
    PORTB=j++;
    OSTimeDly(50);
  }
}

void Task1()
{
  unsigned int j=0;
```

```
  while(1)
  {
    PORTC=j++;
    OSTimeDly(20);
  }
}

void Task2()
{
  unsigned int j=0;
  while(1)
  {
    PORTD=j++;
    OSTimeDly(5);
  }
}




void TaskScheduler()
{
  OSSched();
  while(1)
  {
     //OSSched();      //反复进行调度
  }
}



int main(void)
{
  TCN0Init();
```

```
  OSRdyTbl=0;
  IntNum=0;
  OSTaskCreate(Task0,&Stack[99],0);
  OSTaskCreate(Task1,&Stack[199],1);
  OSTaskCreate(Task2,&Stack[299],2);
  OSTaskCreate(TaskScheduler,&Stack[399],OS_TASKS);
  OSStartTask();
}
```

第八篇:占先式内核(完善的服务)

如果将前面所提到的占先式内核和协作式内核组合在一起，很容易就可以得到一个功能较为完善的占先式内核，它的功能有：

1,挂起和恢复任务

2,任务延时

3,信号量(包括共享型和独占型)

另外，在本例中，在各个任务中加入了从串口发送任务状态的功能。

```
#include <avr/io.h>
#include <avr/Interrupt.h>
#include <avr/signal.h>
unsigned char Stack[400];

register unsigned char OSRdyTbl          asm("r2");    //任务运行就绪表
register unsigned char OSTaskRunningPrio asm("r3");    //正在运行的任务
register unsigned char IntNum            asm("r4");     //中断嵌套计数器
//只有当中断嵌套数为 0，并且有中断要求时，才能在退出中断时，进行任务调度
register unsigned char OSCoreState       asm("r16"); // 系统核心标志位 ,R16 编译器没有使用
//只有大于 R15 的寄存器才能直接赋值 例 LDI R16,0x01
//0x01 正在任务 切换  0x02 有中断要求切换

#define OS_TASKS 3                    //设定运行任务的数量
```

```
struct TaskCtrBlock
{
  unsigned int OSTaskStackTop;  //保存任务的堆栈顶
  unsigned int OSWaitTick;      //任务延时时钟
} TCB[OS_TASKS+1];

//防止被编译器占用
//register unsigned char tempR4  asm("r4");
register unsigned char tempR5  asm("r5");
register unsigned char tempR6  asm("r6");
register unsigned char tempR7  asm("r7");
register unsigned char tempR8  asm("r8");
register unsigned char tempR9  asm("r9");
register unsigned char tempR10 asm("r10");
register unsigned char tempR11 asm("r11");
register unsigned char tempR12 asm("r12");
register unsigned char tempR13 asm("r13");
register unsigned char tempR14 asm("r14");
register unsigned char tempR15 asm("r15");
//register unsigned char tempR16 asm("r16");
register unsigned char tempR16 asm("r17");



//建立任务
void OSTaskCreate(void (*Task)(void),unsigned char *Stack,unsigned char TaskID)
{
  unsigned char i;
  *Stack--=(unsigned int)Task>>8;    //将任务的地址高位压入堆栈，
  *Stack--=(unsigned int)Task;        //将任务的地址低位压入堆栈，

  *Stack--=0x00;                    //R1 __zero_reg__
  *Stack--=0x00;                    //R0 __tmp_reg__
```

```
  *Stack--=0x80;

//SREG 在任务中，开启全局中断
  for(i=0;i<14;i++)    //在 avr-libc 中的 FAQ 中的 What registers are used by the C compiler?
    *Stack--=i;                    //描述了寄存器的作用
  TCB[TaskID].OSTaskStackTop=(unsigned int)Stack;    //将人工堆栈的栈顶，保存到堆栈的数组中
  OSRdyTbl|=0x01<<TaskID;      //任务就绪表已经准备好
}

//开始任务调度,从最低优先级的任务的开始
void OSStartTask()
{
  OSTaskRunningPrio=OS_TASKS;
  SP=TCB[OS_TASKS].OSTaskStackTop+17;
  __asm__ __volatile__(    "reti"       "\n\t"  );
}

//进行任务调度
void OSSched(void)
{

  __asm__ __volatile__("LDI  R16,0x01            \n\t");
  //清除中断要求任务切换的标志位,设置正在任务切换标志位
  __asm__ __volatile__("SEI                      \n\t");
  //开中断,因为如果因中断在任务调度中进行,要重新进行调度时，已经关中断
   //  根据中断时保存寄存器的次序入栈，模拟一次中断后，入栈的情况
  __asm__ __volatile__("PUSH __zero_reg__         \n\t");  //R1
  __asm__ __volatile__("PUSH __tmp_reg__          \n\t");  //R0
  __asm__ __volatile__("IN   __tmp_reg__,__SREG__ \n\t");  //保存状态寄存器 SREG
  __asm__ __volatile__("PUSH __tmp_reg__          \n\t");
  __asm__ __volatile__("CLR  __zero_reg__         \n\t");  //R0 重新清零
  __asm__ __volatile__("PUSH R18                  \n\t");
```

```
__asm__ __volatile__("PUSH R19                \n\t");
__asm__ __volatile__("PUSH R20                \n\t");
__asm__ __volatile__("PUSH R21                \n\t");
__asm__ __volatile__("PUSH R22                \n\t");
__asm__ __volatile__("PUSH R23                \n\t");
__asm__ __volatile__("PUSH R24                \n\t");
__asm__ __volatile__("PUSH R25                \n\t");
__asm__ __volatile__("PUSH R26                \n\t");
__asm__ __volatile__("PUSH R27                \n\t");
__asm__ __volatile__("PUSH R30                \n\t");
__asm__ __volatile__("PUSH R31                \n\t");

__asm__ __volatile__("Int_OSSched:            \n\t");  //当中断要求调度，直接进入这里
__asm__ __volatile__("SEI                     \n\t");
  //开中断,因为如果因中断在任务调度中进行，已经关中断
__asm__ __volatile__("PUSH R28                \n\t");  //R28 与 R29 用于建立在堆栈上的指针
__asm__ __volatile__("PUSH R29                \n\t");  //入栈完成

TCB[OSTaskRunningPrio].OSTaskStackTop=SP;          //将正在运行的任务的堆栈底保存

unsigned char OSNextTaskPrio;                      //在现有堆栈上开设新的空间
for (OSNextTaskPrio = 0;                           //进行任务调度
  OSNextTaskPrio < OS_TASKS && !(OSRdyTbl & (0x01<<OSNextTaskPrio));
  OSNextTaskPrio++);
  OSTaskRunningPrio = OSNextTaskPrio ;

cli();  //保护堆栈转换
SP=TCB[OSTaskRunningPrio].OSTaskStackTop;
sei();

//根据中断时的出栈次序
__asm__ __volatile__("POP  R29                \n\t");
```

```
  __asm__ __volatile__("POP  R28                \n\t");
  __asm__ __volatile__("POP  R31                \n\t");
  __asm__ __volatile__("POP  R30                \n\t");
  __asm__ __volatile__("POP  R27                \n\t");
  __asm__ __volatile__("POP  R26                \n\t");
  __asm__ __volatile__("POP  R25                \n\t");
  __asm__ __volatile__("POP  R24                \n\t");
  __asm__ __volatile__("POP  R23                \n\t");
  __asm__ __volatile__("POP  R22                \n\t");
  __asm__ __volatile__("POP  R21                \n\t");
  __asm__ __volatile__("POP  R20                \n\t");
  __asm__ __volatile__("POP  R19                \n\t");
  __asm__ __volatile__("POP  R18                \n\t");
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //SERG 出栈并恢复
  __asm__ __volatile__("OUT  __SREG__,__tmp_reg__ \n\t");     //
  __asm__ __volatile__("POP  __tmp_reg__        \n\t");     //R0 出栈
  __asm__ __volatile__("POP  __zero_reg__       \n\t");     //R1 出栈
  //中断时出栈完成
  __asm__ __volatile__("CLI                     \n\t");  //关中断
  __asm__ __volatile__("SBRC R16,1              \n\t");  //SBRC 当寄存器位为 0 刚跳过下一条指令
  //检查是在调度时，是否有中断要求任务调度 0x02 是中断要求调度的标志位
  __asm__ __volatile__("RJMP OSSched            \n\t");  //重新调度
  __asm__ __volatile__("LDI  R16,0x00           \n\t");
  //清除中断要求任务切换的标志位,清除正在任务切换标志位
  __asm__ __volatile__("RETI                    \n\t");     //返回并开中断
}



//从中断退出并进行调度
void IntSwitch(void)
{
  //当中断无嵌套，并且没有在切换任务的过程中，直接进行任务切换
```

```
  if(OSCoreState == 0x02 && IntNum==0)
  {
    //进入中断时，已经保存了 SREG 和 R0,R1,R18~R27,R30,R31
    __asm__ __volatile__("POP  R31                \n\t");  //去除因调用子程序而入栈的 PC
    __asm__ __volatile__("POP  R31                \n\t");
    __asm__ __volatile__("LDI  R16,0x01            \n\t");
    //清除中断要求任务切换的标志位,设置正在任务切换标志位
    __asm__ __volatile__("RJMP Int_OSSched         \n\t");  //重新调度
  }
}
//////////////////////////////////////////////任务处理
//挂起任务
void OSTaskSuspend(unsigned char prio)
{
  TCB[prio].OSWaitTick=0;
  OSRdyTbl &= ~(0x01<<prio); //从任务就绪表上去除标志位
  if(OSTaskRunningPrio==prio)  //当要挂起的任务为当前任务
    OSSched();              //从新调度
}

//恢复任务 可以让被 OSTaskSuspend 或 OSTimeDly 暂停的任务恢复
void OSTaskResume(unsigned char prio)
{
  OSRdyTbl |= 0x01<<prio;    //从任务就绪表上重置标志位
    TCB[prio].OSWaitTick=0;        //将时间计时设为 0,到时
  if(OSTaskRunningPrio>prio)   //当要当前任务的优先级低于重置位的任务的优先级
    OSSched();              //从新调度              //从新调度
}

// 任务延时
void OSTimeDly(unsigned int ticks)
{
```

```
  if(ticks)                          //当延时有效
  {
    OSRdyTbl &= ~(0x01<<OSTaskRunningPrio);
    TCB[OSTaskRunningPrio].OSWaitTick=ticks;
    OSSched();                       //从新调度
  }
}



//信号量
struct SemBlk
{
  unsigned char OSEventType;     //型号 0,信号量独占型；1 信号量共享型
  unsigned char OSEventState;    //状态 0,不可用;1,可用
  unsigned char OSTaskPendTbl;   //等待信号量的任务列表
} Sem[10];

//初始化信号量
void OSSemCreat(unsigned char Index,unsigned char Type)
{
  Sem[Index].OSEventType=Type;  //型号 0,信号量独占型；1 信号量共享型
  Sem[Index].OSTaskPendTbl=0;
  Sem[Index].OSEventState=0;
}

//任务等待信号量,挂起
//当 Timeout==0xffff 时，为无限延时
unsigned char OSTaskSemPend(unsigned char Index,unsigned int Timeout)
{

  //unsigned char i=0;
  if(Sem[Index].OSEventState)                    //信号量有效
```

```
 {
  if(Sem[Index].OSEventType==0)                //如果为独占型
  Sem[Index].OSEventState = 0x00;              //信号量被独占，不可用
 }
 else
 {                                    //加入信号的任务等待表
  Sem[Index].OSTaskPendTbl |= 0x01<<OSTaskRunningPrio;
  TCB[OSTaskRunningPrio].OSWaitTick=Timeout;   //如延时为 0，刚无限等待
  OSRdyTbl &= ~(0x01<<OSTaskRunningPrio);      //从任务就绪表中去除
  OSSched();  //从新调度
  if(TCB[OSTaskRunningPrio].OSWaitTick==0 )    //超时，未能拿到资源
       return 0;
 }
 return 1;
}



//发送一个信号量，可以从任务或中断发送
void OSSemPost(unsigned char Index)
{
if(Sem[Index].OSEventType)              //当要求的信号量是共享型
 {
  Sem[Index].OSEventState=0x01;          //使信号量有效
  OSRdyTbl |=Sem [Index].OSTaskPendTbl;  //使在等待该信号的所有任务就绪
  Sem[Index].OSTaskPendTbl=0;            //清空所有等待该信号的等待任务
 }
 else                                 //当要求的信号量为独占型
 {
  unsigned char i;
  for (i = 0; i < OS_TASKS && !(Sem[Index].OSTaskPendTbl & (0x01<<i)); i++);
  if(i < OS_TASKS)                     //如果有任务需要
```

```
    {
      Sem[Index].OSTaskPendTbl &= ~(0x01<<i); //从等待表中去除
      OSRdyTbl |= 0x01<<i;                    //任务就绪
    }
    else
    {
      Sem[Index].OSEventState =1;        //使信号量有效
    }
  }
}

//从任务发送一个信号量，并进行调度
void OSTaskSemPost(unsigned char Index)
{
  OSSemPost(Index);
  OSSched();
}

//清除一个信号量,只对共享型的有用。
//对于独占型的信号量，在任务占用后，就交得不可以用了。

void OSSemClean(unsigned char Index)
{
  Sem[Index].OSEventState =0;          //要求的信号量无效
}




void TCN0Init(void)    // 计时器 0
{
  TCCR0 = 0;
  TCCR0 |= (1<<CS02);  // 256 预分频
```

```
  TIMSK |= (1<<TOIE0); // T0 溢出中断允许
  TCNT0 = 100;        // 置计数起始值

}

SIGNAL(SIG_OVERFLOW0)
{
  IntNum++;     //中断嵌套+1
  sei();  //在中断中，重开中断

  unsigned char i;
  for(i=0;i<OS_TASKS;i++)        //任务时钟
  {
    if(TCB[i].OSWaitTick && TCB[i].OSWaitTick!=0xffff)
    {
      TCB[i].OSWaitTick--;
      if(TCB[i].OSWaitTick==0)        //当任务时钟到时,必须是由定时器减时的才行
      {
        OSRdyTbl |= (0x01<<i);        //使任务可以重新运行
        OSCoreState|=0x02;        //要求任务切换的标志位
      }
    }
  }
  TCNT0=100;
  cli();
  IntNum--;              //中断嵌套-1
  IntSwitch();        //进行任务调度
}


unsigned char __attribute__ ((progmem)) proStrA[]="Task                    ";
```

```
unsigned char strA[20];

SIGNAL(SIG_UART_RECV)        //串口接收中断
{
  strA[0]=UDR;
}


//////////////////////////////////////串口发送

unsigned char *pstr_UART_Send;
unsigned int  nUART_Sending=0;

void UART_Send(unsigned char *Res,unsigned int Len)    //发送字符串数组
{
  if(Len>0)
  {
    pstr_UART_Send=Res;    //发送字串的指针
    nUART_Sending=Len;    //发送字串的长度
    UCSRB=0xB8;                    //发送中断使能
  }
}


//SIGNAL 在中断期间，其它中断禁止

SIGNAL(SIG_UART_DATA)       //串口发送数据中断
{

  IntNum++;     //中断嵌套+1,不充许中断

  if(nUART_Sending)                    //如果未发完
```

```
  {
    UDR=*pstr_UART_Send;        //发送字节
    pstr_UART_Send++;               //发送字串的指针加 1
    nUART_Sending--;               //等待发送的字串数减 1
  }
  if(nUART_Sending==0)           //当已经发送完
  {
    OSSemPost(0);
    OSCoreState|=0x02;      //要求任务切换的标志位
    UCSRB=0x98;
  }
  cli();                          //关发送中断
  IntNum--;
  IntSwitch(); //进行任务调度
}



void UARTInit()    //初始化串口
{
#define fosc 8000000 //晶振 8  MHZ UBRRL=(fosc/16/(baud+1))%256;
#define baud 9600     //波特率
  OSCCAL=0x97;          //串口波特率校正值，从编程器中读出
  //UCSRB=(1<<RXEN)|(1<<TXEN);//允许发送和接收
  UCSRB=0x98;
  //UCSRB=0x08;
  UBRRL=(fosc/16/(baud+1))%256;
  UBRRH=(fosc/16/(baud+1))/256;
  UCSRC=(1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);//8 位数据+1 位 STOP 位
  UCSRB=0xB8;
  UDR=0;
}
```

```
//打印 unsigned int 到字符串中 00000
void strPUT_uInt(unsigned char *Des,unsigned int i)
{
  unsigned char j;
  Des=Des+4;
  for(j=0;j<5;j++)
  {
    *Des=i%10+'0';
    i=i/10;
    Des--;
  }
}

void strPUT_Star(unsigned char *Des,unsigned char i)
{
  unsigned char j;
  for(j=0;j<i;j++)
  {
    *Des++='*';
  }
  *Des++=13;
}

unsigned int strPUT_TaskState(unsigned char *Des,
                              unsigned char TaskID,
                    unsigned char Num)
{
  //unsigned int i=0;
  *(Des+4)='0'+TaskID;
  strPUT_uInt(Des+6,Num);
  strPUT_Star(Des+12,TaskID);
```

```
  return 12+TaskID+1;
}

void Task0()
{
  unsigned int j=0;
  while(1)
  {
    PORTB=j++;
    if(OSTaskSemPend(0,0xffff))
    {
      unsigned int m;
      m=strPUT_TaskState(strA,OSTaskRunningPrio,j);
      UART_Send(strA,m);
    }
    OSTimeDly(200);
  }
}

void Task1()
{
  unsigned int j=0;
  while(1)
  {
    PORTC=j++;
    if(OSTaskSemPend(0,0xffff))
    {
      unsigned int m;
      m=strPUT_TaskState(strA,OSTaskRunningPrio,j);
      UART_Send(strA,m);
    }
    OSTimeDly(100);
```

```
  }
}

void Task2()
{
  unsigned int j=0;
  while(1)
  {
    if(OSTaskSemPend(0,0xffff))
    {
      unsigned int m;
      m=strPUT_TaskState(strA,OSTaskRunningPrio,j);
      UART_Send(strA,m);
    }
    PORTD=j++;
    OSTimeDly(50);
  }
}




void TaskScheduler()
{
  OSSched();
  while(1)
  {
  }
}



int main(void)
{
```

```
  strlcpy_P(strA,proStrA,20);
  UARTInit();
  TCN0Init();

  OSRdyTbl=0;
  IntNum=0;
  OSTaskCreate(Task0,&Stack[99],0);
  OSTaskCreate(Task1,&Stack[199],1);
  OSTaskCreate(Task2,&Stack[299],2);
  OSTaskCreate(TaskScheduler,&Stack[399],OS_TASKS);
  OSStartTask();
}
```

结束语

本文中的例子，基本上用 WinAVR 和 Proteus 调试仿真成功，一定可能存在某些方面的缺陷,因为工作上时间的压力，就没有进一步查找。

但我相信，大家通过学习，会一步步了解一个内核的具体实现形式，慢慢完善，并且最终写出一个属于自己的内核。

当掌握一定的基本知识后，再回头看看 UCOSII 和 small rots51 等，可能会有更深的体会，对进一步了解嵌入式系统和操作系统，条理会更加明析。希望本文能帮助大家做到这一点。

希望大家能够提出自己宝贵的意见，我会进行阶段性的总结，并尽可能地不断改进。

牛顿曾说过，“我能够看得更远，是因为站在巨人的肩膀上。”

希望大家都能出一份力，推动我们的嵌入式的事业的进一步发展。

2006 年 1 月 14 日 希望同行多多交流。

作者 黄健昌 Alex
QQ :43679769
mobil: 13719033059
e-mail: magenta-hjc@tom.com-