则完整的系统为  
#include <reg51.h>   
  
#define MAX\_TASKS 2 //任务槽个数.必须和实际任务数一至   
#define MAX\_TASK\_DEP 12 //最大栈深.最低不得少于2个,保守值为12.   
unsigned char idata task\_stack[MAX\_TASKS][MAX\_TASK\_DEP];//任务堆栈.   
unsigned char task\_id; //当前活动任务号   
unsigned char idata task\_sp[MAX\_TASKS]；//堆指针的保存处  
  
//任务切换函数(任务调度器)   
void task\_switch(){   
task\_sp[task\_id] = SP;   
  
if(++task\_id == MAX\_TASKS)   
task\_id = 0;   
  
SP = task\_sp[task\_id];   
}   
  
//任务装入函数.将指定的函数(参数1)装入指定(参数2)的任务槽中.如果该槽中原来就有任务,则原任务丢失,但系统本身不会发生错误.   
void task\_load(unsigned int fn, unsigned char tid){   
task\_sp[tid] = task\_stack[tid] + 1;   
task\_stack[tid][0] = (unsigned int)fn & 0xff;   
task\_stack[tid][1] = (unsigned int)fn >> 8;   
}   
  
//从指定的任务开始运行任务调度.调用该宏后,将永不返回.   
#define os\_start(tid) {task\_id = tid,SP = task\_sp[tid];return;}

看看这个多任务系统的原理:  
  
这个多任务系统准确来说,叫作"协同式多任务".  
所谓"协同式",指的是当一个任务持续运行而不释放资源时,其它任务是没有任何机会和方式获得运行机会,除非该任务主动释放CPU.  
在本例里,释放CPU是靠task\_switch()来完成的.task\_switch()函数是一个很特殊的函数,我们可以称它为"任务切换器".  
要清楚任务是如何切换的,首先要回顾一下堆栈的相关知识.  
  
有个很简单的问题,因为它太简单了,所以相信大家都没留意过:  
我们知道,不论是CALL还是JMP,都是将当前的程序流打断,请问CALL和JMP的区别是什么?  
你会说:CALL可以RET,JMP不行.没错,但原因是啥呢?为啥CALL过去的就可以用RET跳回来,JMP过去的就不能用RET来跳回呢?  
  
很显然,CALL通过某种方法保存了打断前的某些信息,而在返回断点前执行的RET指令,就是用于取回这些信息.  
不用多说,大家都知道,"某些信息"就是PC指针,而"某种方法"就是压栈.  
很幸运,在51里,堆栈及堆栈指针都是可被任意修改的,只要你不怕死.那么假如在执行RET前将堆栈修改一下会如何?往下看:  
当程序执行CALL后,在子程序里将堆栈刚才压入的断点地址清除掉,并将一个函数的地址压入,那么执行完RET后,程序就跳到这个函数去了.  
事实上,只要我们在RET前将堆栈改掉,就能将程序跳到任务地方去,而不限于CALL里压入的地址.  
  
重点来了......  
首先我们得为每个任务单独开一块内存,这块内存专用于作为对应的任务的堆栈,想将CPU交给哪个任务,只需将栈指针指向谁内存块就行了.  
接下来我们构造一个这样的函数:  
  
当任务调用该函数时,将当前的堆栈指针保存一个变量里,并换上另一个任务的堆栈指针.这就是任务调度器了.  
  
OK了,现在我们只要正确的填充好这几个堆栈的原始内容,再调用这个函数,这个任务调度就能运行起来了.  
那么这几个堆栈里的原始内容是哪里来的呢?这就是"任务装载"函数要干的事了.  
  
在启动任务调度前将各个任务函数的入口地址放在上面所说的"任务专用的内存块"里就行了!对了,顺便说一下,这个"任务专用的内存块"叫作"私栈",私栈的意思就是说,每个任务的堆栈都是私有的,每个任务都有一个自已的堆栈.  
  
话都说到这份上了,相信大家也明白要怎么做了:  
  
1.分配若干个内存块,每个内存块为若干字节:  
这里所说的"若干个内存块"就是私栈,要想同时运行几少个任务就得分配多少块.而"每个子内存块若干字节"就是栈深.记住,每调一层子程序需要2字节.如果不考虑中断,4层调用深度,也就是8字节栈深应该差不多了.  
  
unsigned char idata task\_stack[MAX\_TASKS][MAX\_TASK\_DEP]  
  
当然,还有件事不能忘,就是堆指针的保存处.不然光有堆栈怎么知道应该从哪个地址取数据啊  
unsigned char idata task\_sp[MAX\_TASKS]  
  
上面两项用于装任务信息的区域,我们给它个概念叫"任务槽".有些人叫它"任务堆",我觉得还是"槽"比较直观  
  
对了,还有任务号.不然怎么知道当前运行的是哪个任务呢?  
unsigned char task\_id  
当前运行存放在1号槽的任务时,这个值就是1,运行2号槽的任务时,这个值就是2....  
  
2.构造任务调度函函数:  
void task\_switch(){  
task\_sp[task\_id] = SP;//保存当前任务的栈指针  
  
if(++task\_id == MAX\_TASKS)//任务号切换到下一个任务  
task\_id = 0;  
  
SP = task\_sp[task\_id];//将系统的栈指针指向下个任务的私栈.  
}  
  
  
3.装载任务:  
将各任务的函数地址的低字节和高字节分别入在  
task\_stack[任务号][0]和task\_stack[任务号][1]中:  
  
为了便于使用,写一个函数: task\_load(函数名, 任务号)  
  
void task\_load(unsigned int fn, unsigned char tid){  
task\_sp[tid] = task\_stack[tid] + 1;  
task\_stack[tid][0] = (unsigned int)fn & 0xff;  
task\_stack[tid][1] = (unsigned int)fn >> 8;  
}  
  
4.启动任务调度器:  
将栈指针指向任意一个任务的私栈,执行RET指令.注意,这可很有学问的哦,没玩过堆栈的人脑子有点转不弯:这一RET,RET到哪去了?嘿嘿,别忘了在RET前已经将堆栈指针指向一个函数的入口了.你别把RET看成RET,你把它看成是另一种类型的JMP就好理解了.  
  
SP = task\_sp[任务号];  
return;  
  
做完这4件事后,任务"并行"执行就开始了.你可以象写普通函数一个写任务函数,只需(目前可以这么说)注意在适当的时候(例如以前调延时的地方)调用一下task\_switch(),以让出CPU控制权给别的任务就行了.  
  
  
最后说下效率问题.  
这个多任务系统的开销是每次切换消耗20个机器周期(CALL和RET都算在内了),贵吗?不算贵,对于很多用状态机方式实现的多任务系统来说,其实效率还没这么高--- case switch和if()可不像你想像中那么便宜.  
  
关于内存的消耗我要说的是,当然不能否认这种多任务机制的确很占内存.但建议大家不要老盯着编译器下面的那行字"DATA = XXXbyte".那个值没意义,堆栈没算进去.关于比较省内存多任务机制,我将来会说到.  
概括来说,这个多任务系统适用于实时性要求较高而内存需求不大的应用场合,我在运行于36M主频的STC12C4052上实测了一把,切换一个任务不到3微秒.