**时间片轮调法**

假定单片机要执行的任务有task\_1()，task\_2()，task\_3()，…… task\_n()，各个任务对时间要求不同。

1. **系统基准时间片**

采用定时器中断来产生系统的基准时间片，也叫系统的基准节拍，例如每4ms中断一次。这个可以形象的比喻成脉搏心跳。

1. **任务(时间)的轮调**

每一次心跳，我们就给任务执行的时间标志计数。当标志计数到了，就执行该任务函数。

1. **事件的要求**
2. 每一个事件的执行时间不允许超过一个时间片
3. 事件中不使用较长的delay()函数，可以使用定时器延时等待，但必须永远遵循第一条要求。
4. 执行时间较长的任务，或者较为复杂的任务，可以分割到多个时间片内执行
5. **实时性任务要求**

对于实时性要求较高的任务，比如串口的收发事件，可以考虑在主循环调用，或者在定时中断中调用。

1. **参数传递要求**

各个任务函数之间参数传递，建议使用全局变量。任务中的内部函数，可以使用局部变量。

1. **程序架构1**

//主程序架构

void main()

{

Init\_All(); //初始化函数

while(1) //主循环函数

{

TimeEvent(); //事件驱动函数

RxProcess(); //串口接收数据处理

TxProcess(); //串口发送数据处理

}

} //主程序结束

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

系统基准节拍

Input:NONE

Output:Return KeyCode

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Timer0Interrupt(void)

{

TH0=0x0fc;

TL0=0x66;

if((++Time\_1ms&&0x03)==0)

TimeIntFlag=1;

}

//时间事件，时间轮询法调用函数

void TimeEvent(void)

{

if(TimeIntFlag)

{

TimeIntFlag=0;

ClearWatchDog(); //这里喂狗

Display(); //在4ms事件中，通过LED扫描显示，以及喂狗

if(++Time20ms==5)

{

Time20ms=0;

TimeEvent20ms(); //在20ms事件中，处理键盘扫描函数

}

if(++Time1s==10)

{

Time1s=0;

TimeEvent1s(); //在1s事件中，使工作指示灯闪烁

}

UARTTimeEvent(); //串口的数据接收事件在4ms事件中处理

}

}

架构分析，使用一个定时器产生系统时钟滴答，然后时钟滴答到了，就更新时间标志，然后统一用一个事件函数来根据时间标志分时的执行各个任务函数。但是任务执行完后，时间标志被重置，并重新计数。那么这个任务函数相当于被调度在了任务队列的末尾。当各个任务函数调用的时间不同，就造成了任务执行频率的不同。这也是时间片的大小商定，以及时间片分布的问题。需要从实际的任务考虑，并取得一个最佳的时间片，以及合理的安排各个任务函数的关系。

1. **程序架构2**

与程序架构1相比，原理大致相同，执行机制不同。

void main (void)

{

Init(void);

while(1)

{

if(sys.5ms>0)

{

sys.5ms=0;

task\_index++;

if(task\_index>n)

{

task\_index=0;

}

}

switch(task\_index)

{

case 0: Task\_1(); break;

case 1: Task\_2();break;

case n: Task\_N();break;

default:break;

}

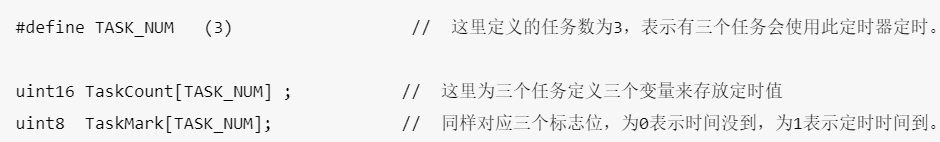
}

}

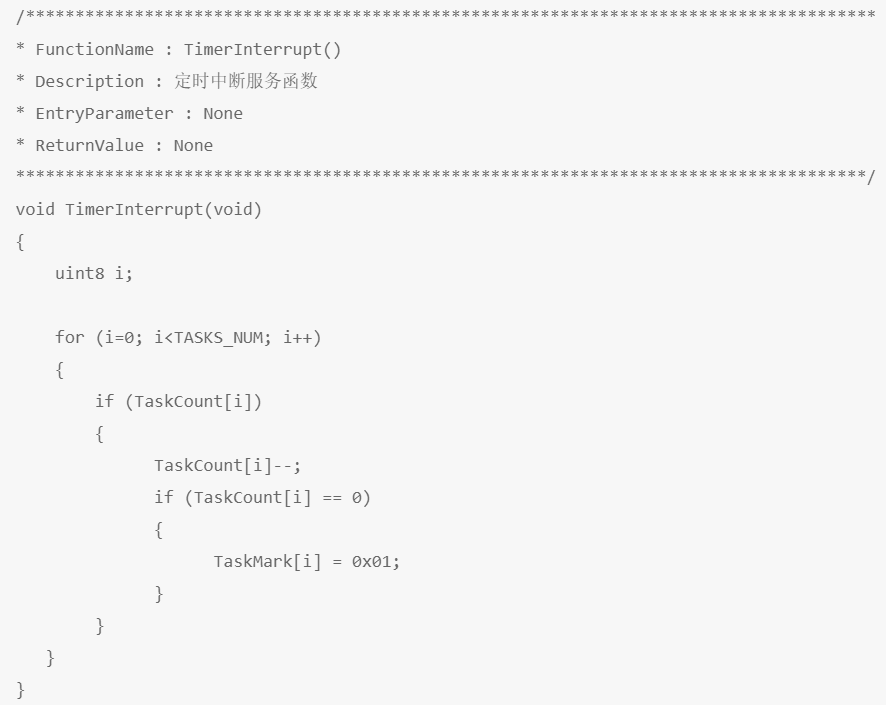
1. **具体实例**

使用1个定时器，可以是任意的定时器，这里不做特殊说明，下面假设有3个任务，那么我们应该做如下工作：

1. 初始化定时器，这里假设定时器的定时中断为1ms(当然你可以改成10ms，这个和操作系统一样，中断过于频繁效率就低，中断太长，实时性差)。
2. 定义一个数值：

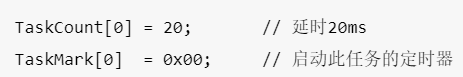


1. 在定时器中断服务函数中添加



**代码解释：**定时中断服务函数，在中断中逐个判断，如果定时值为0了，表示没有使用此定时器或此定时器已经完成定时，不着处理。否则定时器减一，知道为零时，相应标志位值1，表示此任务的定时值到了。

1. 在我们的应用程序中，在需要的应用定时的地方添加如下代码，下面就以任务1为例

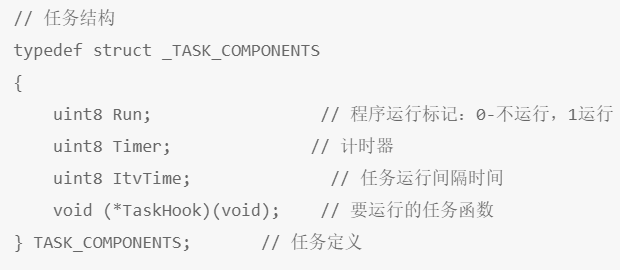


到此我们只需要在任务中判断TaskMark[0] 是否为0x01即可。其他任务添加相同，至此一个定时器的复用问题就实现了。通过上面对1个定时器的复用我们可以看出，在等待一个定时的到来的同时我们可以循环判断标志位，同时也可以去执行其他函数。

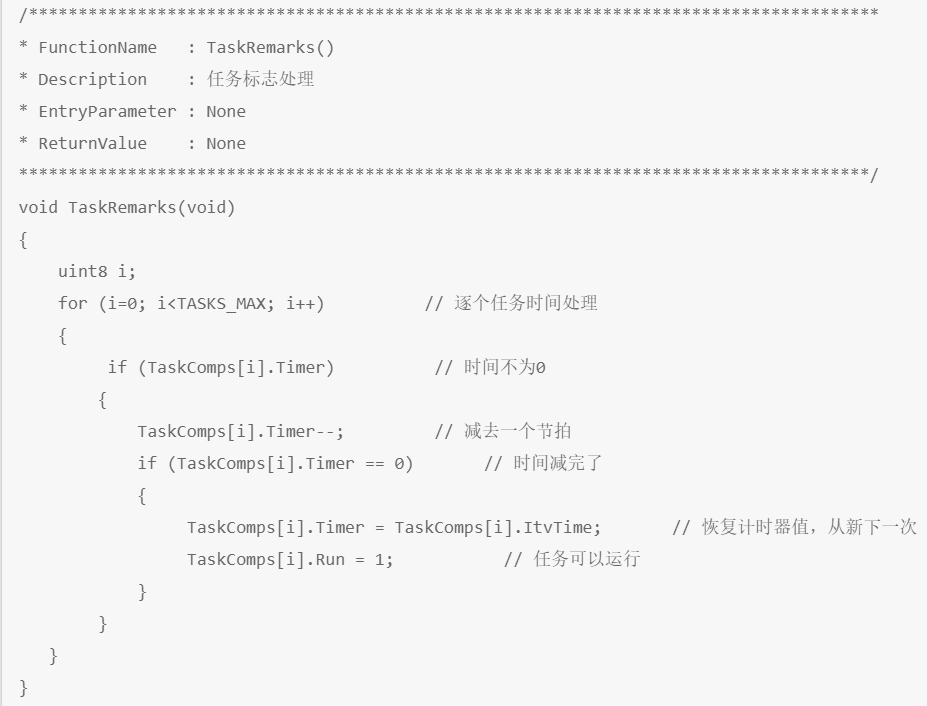
**循环判断标志位：**循环判断标志位，就是类似于顺序执行程序，一个大循环，只是这个延时比普通的for循环要精确一些，可以实现精确延时。

**执行其他函数：**在一个函数延时的时候去执行其他的函数，充分利用CPU时间。

1. **时间片轮询法架构**
2. 设计一个结构体

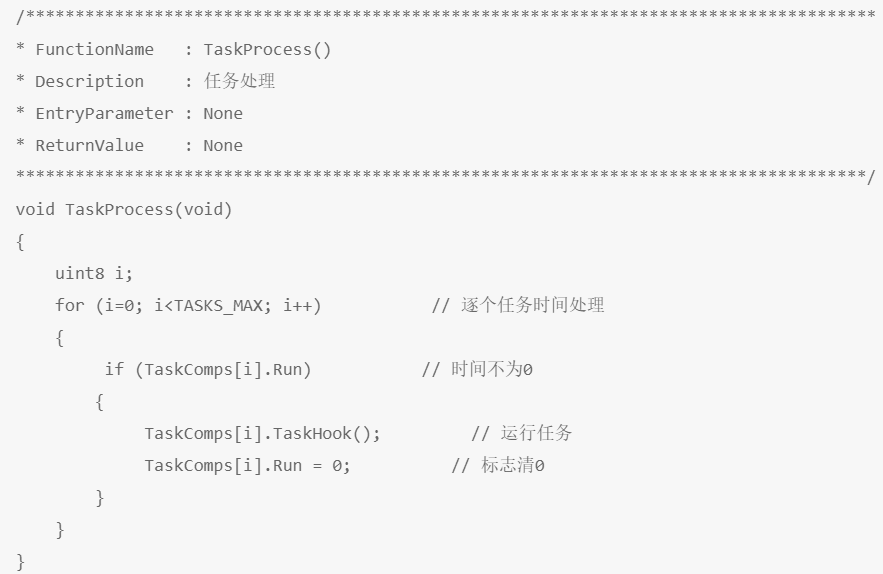


1. 任务运行标志出来，此函数就相当于中断服务函数，需要在定时器的中断服务函数中调用此函数，这里独立出来，并于移植和理解。



1. 任务处理

此函数就是判断什么时候该执行那一个任务了，实现任务的管理操作，应用者只需要在main()函数中调用此函数就可以了，并不需要去分别调用和处理任务函数。



1. **示例**

假设我们有三个任务：时钟显示，按键扫描，和工作状态显示。

1. 定义一个上面定义的那种结构体变量



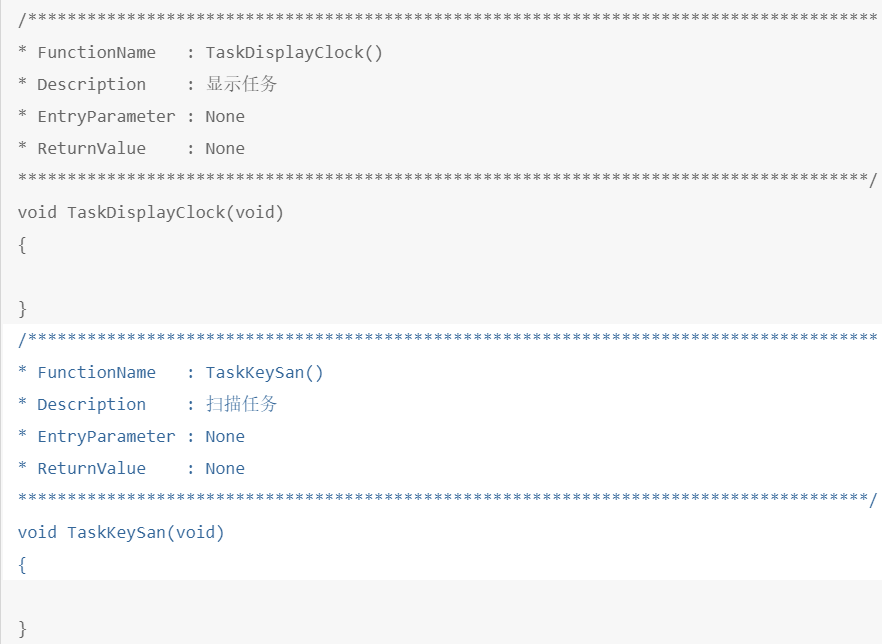
在定义变量时，我们已经初始化了值，这些值的初始化，非常重要，跟具体的执行时间优先级等都有关系，这个需要自己掌握。

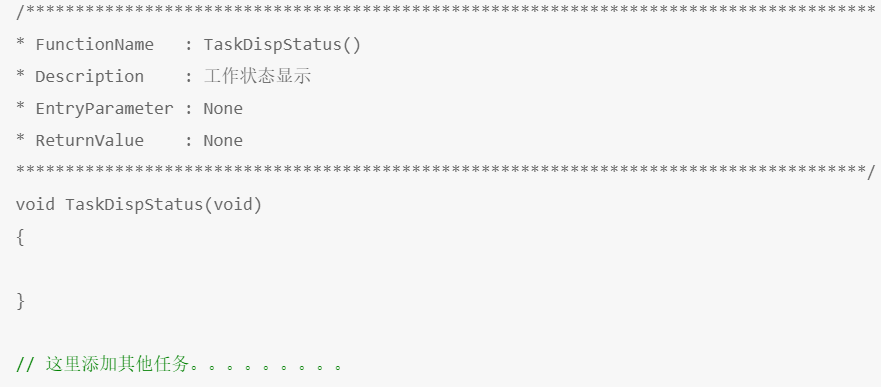
1. 大概意思是，我们有三个任务，没1s执行以下时钟显示，因为我们的时钟最小单位是1s，所以在秒变化后才显示一次就够了。
2. 由于按键在按下时会参数抖动，而我们知道一般按键的抖动大概是20ms，那么我们在顺序执行的函数中一般是延伸20ms，而这里我们每20ms扫描一次，是非常不错的出来，即达到了消抖的目的，也不会漏掉按键输入。
3. 为了能够显示按键后的其他提示和工作界面，我们这里设计每30ms显示一次，如果你觉得反应慢了，你可以让这些值小一点。后面的名称是对应的函数名，你必须在应用程序中编写这函数名称和这三个一样的任务。
4. 任务列表



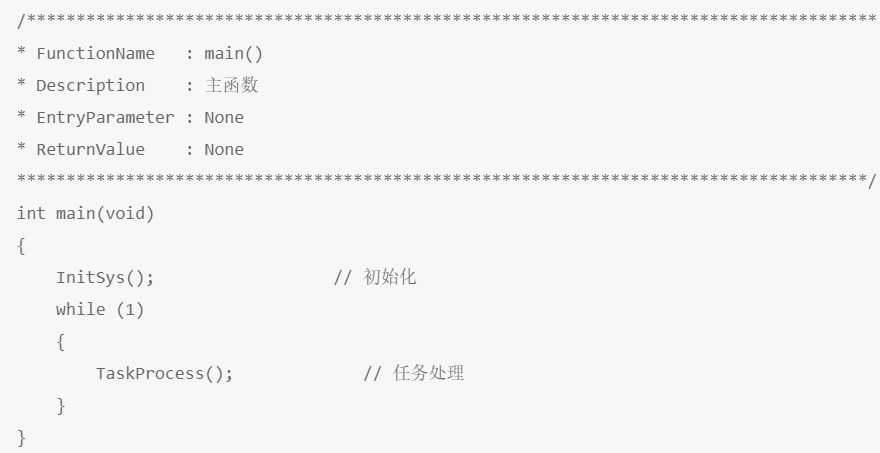
这里定义这个任务清单的目的其实就是参数TASKS\_MAX的值，其他值是没有具体的意义的，只是为了清晰的表面任务的关系而已。

1. 编写任务函数





1. 主函数



看看任务之间是不是相互并不干扰？并行运行呢？当然重要的是，还需要，注意任务之间进行数据传递时，需要采用全局变量，除此之外还需要注意划分任务以及任务的执行时间，在编写任务时，尽量让任务尽快执行完成。