
小小调度器（作者“smset”）

下文为“Gthgth”自己的整理说明

一、前言

（一）、调度器说明

- 1) 超级可以移植性，与 CPU 无关，几乎任何支持 C 语言编程的 CPU 都可以用！（本文仅仅以 51 单片机为例而已，但实际上可以任意移植）
- 2) 小之又小，原理很简单，一看就懂。
- 3) 省之又省，可以说对 RAM 和 ROM 省到极致。
- 4) 取 protothread 之精华，将定时器与状态机和伪线程语法融合到一个框架，任务函数可以有两种写法。
- 5) 基于定时器触发，调度效率高，最大化减少无效的代码运行时间。

小小调度器任务函数的写法主要注意的，主要有三点：

- 1) 任务函数内部变量，建议都用静态局部变量来定义。
 - 2) 任务函数内不能用 switch 语句。
 - 3) 任务函数内，不能用 return 语句。因为 return 已经被赋予任务延时的特定意义。（这是返回型任务函数版本的一个强制要求）
- 这三点，并不会明显造成写程序的不方便。

从裸奔到使用 OS 操作系统或调度系统的代价主要有：

硬件资源代价（对 RAM 和 ROM 的消耗），学习代价（学会其原理，并掌握其用法），移植代价（往不同 cpu 上移植的工作量），效率代价（使用调度系统后带来的额外 cpu 负担），商业代价（版权费用），稳定性代价（是否引入潜在不稳定因素，或者增大 bug 跟踪调试工作量）。

从这几方面来讲，应用小小调度器的代价，都是非常小的。

- 1) 硬件资源代价：前面的优化版本已经说明问题。keil 下，本例程 ram 消耗：22 字节，rom 消耗 126 字节。
- 2) 学习代价：小小调度器总共只有十多行代码，如果我们做一个简单的解释说明，理解起来其实是很快的。我相信学习时间比其他调度系统要短。

-
- 3) 移植代价： 几乎没有什么移植工作量，对于各种 cpu, 几乎是通吃。
 - 4) 效率代价： 我们一直在努力优化，相信调度效率已经不低了。比如任务切换时间，应该是可以做到 uS 级别，甚至亚 uS 级别。
 - 5) 商业代价： 小小本调度器为免费使用，无需支付任何费用。
 - 6) 稳定性代价：小小调度器本质上仅仅是几个宏而已，未涉及任何对内部寄存器或堆栈的操作，避免了引入不稳定风险因素，所有操作都在可预见，可把控的前提下进行。
-

（二）、调度器原理解读

ProtoThread 多任务核心原理解读：

- 1、#define _SS static unsigned char _lc; switch(_lc){default:
- 2、#define _EE ;}; _lc=0; return 255;
- 3、#define WaitX(tickets) do { _lc=__LINE__+((__LINE__%256)==0); return tickets ;} while(0); case __LINE__+((__LINE__%256)==0):
- 4、#define RunTask(TaskName,TaskID) do { if (timers[TaskID]==0) timers[TaskID]=TaskName(); } while(0);

- 1. 首先 ProtoThread 实现的多任务是“协作式”多任务，也即多个任务间通过“友好合作”的方法共享 CPU 资源，当一个任务不需要 CPU 时，主动让出 CPU。
 - 2. ProtoThread 任务调用 WaitX 主动让出 CPU 时，作了两件事：记录断点（代码当前行记录到_lc 变量里），让出 CPU（return 实现）
 - 3. 任务的恢复运行，当一个任务等待的条件发生时，恢复运行，这时应该要恢复到原来的断点处，这通过_SS 宏里的 switch(_lc)和 WaitX 宏里的 case __LINE__+((__LINE__%256)==0): 实现。
 - 2. 所以，恢复运行的本质是通过 switch 找到原来有 WaitX 留下来的断点。
 - 4. 至于 RunTask 宏是加了定时服务，当任务定时时间到后，去运行任务函数，也即充当了调度器的角色。
-

理解 ProtoThread 结合 timer 延时查询调度的功能，就明白为什么能实现多任务了。

首先，小小调度器是 C 语言通用的，只要支持 C 语言的编译器都可以用。

然后又基于延时查询的方式来调度的，调度器不断查询每个任务的延时，对延时时间到的任务，再继续往下执行。

所以，如果要取一个名字的话应该是：通用 C 延时 查询 调度器，翻译成英文是 General C Delay Querying Scheduler

作者 “smset” 建的群：小小调度器 QQ 群：371719283

<http://amobbs.com>

<http://bbs.yleee.com.cn/thread-35353-1-1.html>

http://bbs.elecfans.com/jishu_427231_1_1.html

二、小小调度器原形示例

小小调度器如下：

```
#include <stc89c51.h>
```

```
/*小调度器开始*/
```

```
#define MAXTASKS 3
```

```
volatile unsigned char timers[MAXTASKS];
```

```
#define _SS static unsigned char _lc=0; switch(_lc){default:
```

```
#define _EE }; _lc=0; return 255;
```

```
#define WaitX(tickets) do {_lc=(_lc%255)+1; return tickets; } while(0); case (_lc%255)+1:
```

```
#define RunTask(TaskName,TaskID) do { if (timers[TaskID]==0) timers[TaskID]=TaskName(); } while(0);

#define RunTaskA(TaskName,TaskID) { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} } //前面的任务优先保证执行

#define CallSub(SubTaskName) do {unsigned char currdt; _lc=(__LINE__%255)+1; return 0; case (__LINE__%255)+1: currdt=SubTaskName();
if(currdt!=255) return currdt;} while(0); //和上面是同一行，因为一行显示不下了，里面有 return 0，在执行子函数前释放了一下 CPU

#define InitTasks() {unsigned char i; for(i=MAXTASKS;i>0 ;i--) timers[i-1]=0; }

#define UpdateTimers() {unsigned char i; for(i=MAXTASKS;i>0 ;i--){if((timers[i-1]!=0)&&(timers[i-1]!=255)) timers[i-1]--;}}

#define SEM unsigned int

//初始化信号量

#define InitSem(sem) sem=0;
```

//等待信号量

#define WaitSem(sem) do{ sem=1; WaitX(0); if (sem>0) return 1;} while(0); //里面有 WaitX(0)，执行的时候释放了一下 CPU

//等待信号量或定时器溢出， 定时器 tickets 最大为 0xFFFF

#define WaitSemX(sem,tickets) do { sem=tickets+1; WaitX(0); if(sem>1){ sem--; return 1;} } while(0); //里面有 WaitX(0)，执行的时候释放了一下 CPU

//发送信号量

#define SendSem(sem) do {sem=0;} while(0);

/*****小小调度器结束*****/

/*****下面是示例程序*****/

sbit LED1 = P2^1;

```
sbit LED2 = P2^2;
```

```
sbit LED0 = P2^5;
```

```
unsigned char task0(){
```

```
    _SS
```

```
    while(1){
```

```
        WaitX(50);
```

```
        LED0=!LED0;
```

```
    }
```

```
    _EE
```

```
}
```

```
unsigned char  task1(){
```

```
  _SS
```

```
    while(1){
```

```
        WaitX(100);
```

```
        LED1=!LED1;
```

```
    }
```

```
  _EE
```

```
}
```

```
unsigned char  task2(){
```

```
  _SS
```

```
    while(1){
```

```
        WaitX(100);
```

```
    LED2=!LED2;
```

```
}
```

```
_EE
```

```
}
```

```
void InitT0()
```

```
{
```

```
    TMOD = 0x21;
```

```
    IE |= 0x82; // 12t
```

```
    TL0=0Xff;
```

```
    TH0=0XDB;
```

```
    TR0 = 1;
```

```
}
```

```
void INTT0(void) interrupt 1 using 1
```

```
{
```

```
    TL0=0Xff;    //10ms 重装
```

```
    TH0=0XDB;//b7;
```

```
    UpdateTimers();
```

```
    RunTask(task0,0);//任务 0 具有精确按时获得执行的权限，要求：task0 每次执行消耗时间<0.5 个 ticket
```

```
}
```

```
//小小调度器 QQ 群： 371719283
```

```
void main()

{

    InitT0();

    InitTasks(); //初始化任务，实际上是给 timers 清零

    while(1){

        // RunTask(task0,0);

        RunTaskA(task1,1);//任务 1 具有比任务 2 高的运行权限

        RunTaskA(task2,2);//任务 2 具有低的运行权限

    }

}
```

三、调度器宏展开分析

注：本文有关所有小小调度器文字，介绍，程序等，都是作者“smset”和各位大虾的对话，我仅仅整理了一下。本文主要摘自 amobbs 网站；由于某些原因，有些话具体是哪位大虾说的，我也搞不清楚，所以就没有指出大虾的具体名字；

（一）、主函数里面的循环

```
while(1){  
  
    // RunTask(task0,0);  
  
    RunTaskA(task1,1);//任务 1 具有比任务 2 高的运行权限  
  
    RunTaskA(task2,2);//任务 2 具有低的运行权限  
  
}
```

把宏替换掉 #define RunTaskA(TaskName,TaskID) { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} }

```
while(1){  
  
    // RunTask(task0,0);  
  
    // RunTaskA(task1,1);//任务 1 具有比任务 2 高的运行权限  
  
    // RunTaskA(task2,2);//任务 2 具有低的运行权限  
  
  
    { if (timers[1]==0) {timers[1]=task1(); continue;} }
```

```
    { if (timers[2]==0) {timers[2]=task2(); continue;} }  
  
}
```

在主函数中 CPU 反复的扫描各个 RunTaskA 函数，如果满足了（定时器减到 0 了）就执行 if 判断语句后面花括号里面的语句。

timers[1]=task1()（timers[TaskID]=TaskName()），这句话的意思是执行函数 task1 花括号里面的语句，并把返回值赋值给 timers[1]。

函数名就是指针，代表函数的首地址。函数名后面有括号代表执行函数，最后返回的是一个值。

在主函数里面的 while(1)，先执行任务函数 RunTaskA(task1,1);，因为在调度器初始化的时候就把所有 timers 清零；判断条件满足，返回值放到 timers[1]（timers[1]=100）里面后，遇到 continue 语句，结束本次循环，然后重新执行循环体。（从循环体的开始执行，while(1)后面花括号里面的是循环体）

第二次执行任务函数 RunTaskA(task1,1); 因为函数里面(timers[TaskID]==0)不成立（第一次执行的时候把 100 返回给 timers[1]），这时定时器延时没有减到 0；转而执行任务函数 RunTaskA(task2,2)，（因为在调度器初始化的时候就把所有 timers 清零，任务函数 2 里判断条件满足）同样执行完任务函数 2 后，遇到 continue；结束本次循环。重新执行循环体，就转到任务函数 RunTaskA(task1,1)里面执行了，判断延时减到 0 了没有，如果 (timers[TaskID]==0)成立就跳进去执行函数，如果条件不成立转去 RunTaskA(task2,2)函数；判断函数里面的定时延时是否为零来决定是否执行函数里面的语句。这样在主函数中不断的循环扫描，条件成立就执行任务函数里面的语句，条件不成立就继续循环扫描；等待定时器中断减减，直到把每个值减到 0，条件成熟然后执行。

因为每执行完一个任务函数都会执行 `continue`，这样 CPU 就会结束本次循环，重新跳转到循环体的开始来执行。这样就能解释写在主函数前面的语句运行权限高，会优先扫描执行的。但是仅仅有优先运行权，不是真正意义的优先级，因为“高优先级”代码不能抢占排在后面的低优先级代码。

如果主函数里面用的是 `RunTaskA` 函数，展开后有 `continue`。这样就导致了一个问题，任务函数里面不能有 `WaitX(0)`，因为有 `continue` 的关系 `WaitX(0)` 就不能释放 CPU 资源了，或者说释放 CPU 资源执行结果不是我们预期的那样。`RunTaskA` 用在每个任务函数相对独立的情况，写在主函数循环体前面的任务函数运行权限相对高，退出后每次优先扫描执行。

如果主函数里面用的是 `RunTask` 来调度函数，任务函数里面就可以用 `WaitX(0)`。详细区别后面会讲到。如果主函数里面用的是 `RunTask` 调度任务函数，每个任务函数优先级都是平等的，CPU 轮流扫描，条件满足就去调用；用到 `WaitX(0)` 返回到主函数中也是继续扫描本任务函数后面的任务函数，然后扫描执行前面的，这样循环一个周期。如果条件满足就执行。用 `RunTask` 调动任务函数，因为任务函数里面能用到 `WaitX(0)`，这样编程就更灵活。

说明：无特殊说明像 `unsigned char task0()` 函数，我们叫做任务函数。如任务函数 1，任务函数 2 等等

（二）、定时中断里面的函数

```
void INTT0(void) interrupt 1 using 1
```

```
{
```

```
    TL0=0Xff;    //10ms 重装
```

```
    TH0=0XDB;//b7;
```

```
UpdateTimers();
```

```
// RunTask(task0,0);//任务 0 具有精确按时获得执行的权限，要求：task0 每次执行消耗时间<0.5 个 ticket  
}
```

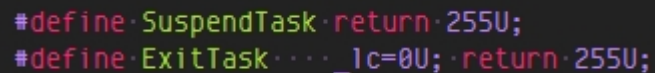
定时器中断，每隔 10MS 运行一次（ticket 可以根据具体情况修改）。

我们把 UpdateTimers();这个宏替换一下

```
{unsigned char i; for(i=MAXTASKS;i>0;i--){if((timers[i-1]!=0)&&(timers[i-1]!=255)) timers[i-1]--;}}
```

每个任务里面 timers 的值只要是不等于 0 和 255，定时中断后每个任务对应的延时就会就减一次。如果 timers[i]==0 后，在主函数扫描到的时候就会执行这个任务函数了。

如果 timers[i]==255 后，定时器每次中断都不会把里面的值减掉，这样在主函数扫描的时候虽然扫描到任务函数，但是条件不满足(timers[i]!=0)，这样任务函数就永远不会被执行的。对应的这个任务函数就会被挂起。



```
#define SuspendTask return 255U;  
#define ExitTask ... _lc=0U; return 255U;
```

上面的截图是一个网友发的

第一句宏定义 就是把任务挂起。第二句宏定义 就是把任务函数退出，因为下次再执行这个任务函数的时候就可以从头开始。

如果某个任务函数被挂起了或者退出了，下次想让它运行，把对应的 `timers[x]` 赋值为 0 就可以了。

示例中在定时中断里面注释掉 `RunTask(task0,0);`，后面的注释说的很清楚，因为写在中断函数里面的，每次定时中断都会执行，所以就要求任务函数每次执行消耗的时间要短，越短越好。因为它每次都会占用 CPU 的资源。如果任务函数 0 每次执行消耗 10ms，这样其他任务就没有机会执行了。作者要求 `task0` 每次执行消耗时间 < 0.5 个 `ticket`，是考虑到，其他函数里面基本上没有延时等待，或者长的延时等待，具体执行函数语句是非常快的。并不是每个 `ticket` 到来后所有任务函数都会满足执行，这样对全局也是没有大的影响的。当然了，你认为这样不好，那就不要用，这里只不过是给大家提供一种方法，告诉你在特殊的时候，你对全局把握较好，可以这么用。这样从某种意义上就可以比较准时的执行 `RunTask(task0,0);` 任务函数了。当然了，如果其他中断优先级比这个定时中断高，占着 CPU 不放，这样就会导致时基中断不准确。(STM32 的滴答定时器可以避免这种情况？只能保证时基不变。)

（三）、主函数和定时器中断调度

把主函数循环和中断服务函数里面的宏替换后如下：

```
while(1){  
  
    { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} }  
  
    { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} }  
  
}
```

```
void INTT0(void) interrupt 1 using 1
```

```
{
```

```
    TL0=0Xff;    //10ms 重装
```

```
    TH0=0XDB;//b7;
```

```
    {unsigned char i; for(i=MAXTASKS;i>0 ;i--){if((timers[i-1]!=0)&&(timers[i-1]!=255)) timers[i-1]--;}}
```

```
}
```

看一下主循环，和定时中断函数就能明白，是基于延时查询的方式来调度的，调度器不断查询每个任务的延时，对延时时间到的任务，再继续往下执行。

上面任务 1 里面返回的是 100，这样要减到 0 就相当于你延时 $100 \times 10\text{ms} = 1\text{s}$ ，也就是任务 1，每隔 1S 执行一次。

`volatile unsigned char timers[MAXTASKS];`这句语句用到了 `volatile`，是因为定时器和主函数都对它定义的值进行了改写，防止编译器对其进行优化修

改，具体用法请找度娘。`unsigned char timers[MAXTASKS]` 定义数组类型可知最多有 256 个任务函数，也就是说 main 主循环里面最多有 256 个任务

函数（不包括子任务函数的个数），`WaitX(tickets)`中的 `tickets` 取值范围是 0—255。0 对应的任务函数在主函数循环扫描到时时立即运行，255 对应

任务函数的挂起。`WaitX(tickets)`可以连用多个来增加延时运行的时间，

当然了在主函数中也可用 RunTask。RunTask(TaskName,TaskID) 和 RunTaskA(TaskName,TaskID)的区别就是函数后面的 continue,

在书中查了一下 continue 的说明：用在循环中，它使程序跳到循环的开头。

（四）、任务函数的分解

1、任务函数

```
RunTaskA(task1,1);
```

宏定义

```
#define RunTaskA(TaskName,TaskID) { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} } //前面的任务优先保证执行
```

注意在分析的时候看清是 RunTaskA，还是 RunTask。

2、任务函数分析

这段，作者和各位大侠都分析的很清楚了，感觉也是精华所在。这里把 SWITCH 语句的用法从书上拷贝一份如下：

```
Switch(表达式){
```

```
case 常量 1:
```

```
语句
```

```
...
```

break;

case 常量 2:

语句

...

break;

case 常量 n:

语句

...

break;

default:

语句

}

说明

Switch 语句计算表达式的值，然后转入 **case** 标号之一。标号是不允许重复的，所以只能选中一个 **case** 标号。表达式的结果必须是整数，字符或枚举量。

Case 的标号顺序不限，但必须是常量。**Default** 标号可以放在 **switch** 语句中的任何地方，任何两个 **case** 标号都不能有相同的值。当 **c** 遇到 **switch** 语句

时，它先求表达式的值，然后查找匹配的 **case** 标号。如果没有找到，就寻找 **default** 标号。如果没有找到 **default**，该语句就什么都不做。具体 **switch** 语句用法请查看书籍或问度娘。

接下来我们看一下程序。

函数原型

```
unsigned char task1(){
```

```
_SS
```

```
while(1){
```

```
    WaitX(100);
```

```
    LED1=!LED1;
```

```
}
```

```
_EE
```

```
}
```

把任务函数 1 里面的宏都替换过来；

```
unsigned char task1(){
```

```
static unsigned char _lc=0; switch(_lc){default:
```

```
while(1){  
  
do {_lc=(__LINE__%255)+1; return 100 ;} while(0); case (__LINE__%255)+1::  
  
LED1=!LED1;  
  
}  
  
;}; _lc=0; return 255;}
```

说明:

因为第一次运行, 静态变量 `_lc==0`,

`switch(0){default:`

因为 `case 0` 的状态是没有的; 后面行号最小是 1, 因为 `_lc=(__LINE__%255)+1`; 这句语句一执行, `_lc==1`。程序找不到相同的标号, 只能执行 `default:`

后面的函数了。执行 `do {_lc=(__LINE__%255)+1; return 100 ;} while(0) ; case (__LINE__%255)+1::`

运行到 `return`, 这样就把 100 返回去了, 同时结束函数。后面的 `case (__LINE__%255)+1`: 在这个时候就是一个确定的值了, 这行函数执行结果就是先把

当前行号放到 `_lc` 里面, 同时语句后半句 `case` 的标号就是这个值 (这里需要解释一下 `__LINE__`, `__LINE__` 是 C/C++ 预定义宏, 它是根据行号改变的常数,

是当前源代码行号。)。执行完 `default:` 后面的函数其实做了 2 件事情, 第一件就是 返回 100 个时间单位, 第二件就是把行号赋值给 `_lc`; 同时这个行号

也是 **case** 后面的值。下次程序运行就通过 **switch(_lc)** 寻找到 **case _lc**，执行后面的函数了。接下来把主函数 **RunTaskA(task1,1)** 宏替换一下。{ **if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} }** 里面的函数第一句就是判断延时减到 0 了没有，第一次运行因为初始化 **timers** 为 0，所以，执行函数；把任务函数返回值赋值给 **timers**，这时候 **timers[1]==100**；下次再运行 **RunTaskA** 的时候因为 **timers[TaskID]** 不等于 0 了，条件不符合，就转去执行其他 **RunTaskA** 函数。当定时器把 **timers[1]** 的数值减为零后，**cpu** 循环执行到任务函数 1 的时候，因为条件满足就去执行 **if** 后面花括号里面的语句，这个时候因为上次执行的时候把 **_lc=(_LINE_%255)+1**，所以再一次执行任务函数 1 里面的 **static unsigned char _lc=0; switch(_lc)** 语句的时候因为 **_lc** 是静态变量的原因（**static** 为静态局部变量赋初值是在编译时进行的，即只赋初值一次，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而为自动变量赋初值，不是在编译时进行的，而是在函数调用时进行，每调用函数重新给一次值，相当于执行一次赋值语句。）。**switch(_lc)** 就转去执行 **case (_LINE_%255)+1;** 后面的函数，执行 **LED1=!LED1;**。如下：

```
while(1){  
  
    do {_lc=(_LINE_%255)+1; return 100 ;} while(0); case (_LINE_%255)+1;  
  
    LED1=!LED1;  
  
}
```

任务函数里面的语句是在 **switch(_lc) {}** 作用范围之内，也就是说，执行的时候先跳转到 **case** 标号 执行后面的语句，在执行语句的时候遇到 **while(1) {}**,

因为有 `while(1)` 的存在，继续把行号放在 `_lc` 里面，继续 `return 100`，结束函数。主程序里面继续等待定时器减到 0，继续执行任务函数 1，如此循环下去。。。。。。，在外面看到就是 LED 每 100 个时间单位就翻转一次，如果把 LED 翻转改为其他执行的程序，就是每隔 100 个时间单位程序执行一次。

作者说过，如果你想让任务函数一直运行，在这里就用 `while(1)`；如果想运行次数可控就用 `for` 语句，如果想运行一次就停止，就不用循环语句（把任务函数里面的 `while(1)` 去掉）。

因为 `Switch(表达式)`，表达式没有规定是什么，但是楼主定义 `unsigned char _lc=0` 的最大值是 255，这样如果一个任务函数程序编写超过 256 行，编译器编译的时候这样就存在两个行号相同的情况，但无论多少行，`csae` 后面的值都不等于 0，因为作者 `_lc=(__LINE__%255)+1` 这么处理了。

每个任务函数里面可以写任意多行代码，代码行数不受限制。

每个任务函数里面最多可以用 255 个 `WaitX()`。当然了每个 `WaitX()`，单独一行。

每个任务函数里面前 255 行，行号不会重复，所以 `WaitX()` 可以连续用也不会有问题。

如果超过 255 行后，用 `WaitX()`，会有行号重复的概率；出现后，只需在 `WaitX()` 上面加一个回车就可以了。

因为无论 `__LINE__` 是什么类型，；一定是正数，为无符号数。`unsigned char _lc=0` 也是无符号正数，

`_lc=(__LINE__%255)+1`，无论行号是多少，对应的 `_lc` 都是在 1---255 之间变化，所以每个任务函数不容许超过 255 个 `WaitX()`。

最起码子函数内可以编写到 255 行以内不会出现行号重复的问题，（行号为 0-254 之间+1，_lc 为 1--255。）不会出现两个 case 后面的标号一样的情况，如果一个任务函数里面编写的行数很多，出现编译编译的时候出现错误，我们只需在错误处加一个回车符就可以了。一个独立的函数一般不会超出这么多行，因为这个不是 BUG，作者也就没有说什么。具体的请参考作者的原话，大体就是这意思。

所以有人说把_lc 定义为 uint16，作者说可以。

其实对于任务函数的分析，我感觉没有网上大家讨论的清楚，如果有机会大家看看网上作者和各位大侠的对话。

（五）、信号量的应用；重新定义了两个任务函数

使用信号量，先要用 SEM 定义。

信号量是任务函数之间的通信方式之一。//看清楚是任务函数之间的通讯。

#define SEM unsigned int; 信号量的类型是 int，可以定义 65536 个，是个全局变量。

先定义一个信号量

SEM sm1;

任务函数 1, 和 2

```
void task1(){
```

```
_SS
```

```
    InitSem(sm1);
```

```
    while(1){
```

```
        WaiSem(sm1);
```

```
        LED1=!LED1;
```

```
    }
```

```
_EE
```

```
}
```

```
void task2(){
```

```
_SS
```

```
    while(1){
```

```
    WaitX(100);

    SendSem(sm1);

}

_EE

}
```

这个例子执行的结果是：**task1** 每次收到 **task2** 的信号时，才进行一次 **led** 翻转。这里强调的是进行[一次](#)翻转。

分析如下：

我们把任务函数 **1** 里面的宏替换出来。

```
void task1(){

//_SS

static unsigned char _lc=0; switch(_lc){default:

// InitSem(sm1);

sem=0;
```

```
while(1){  
  
    // WaiSem(sm1);  
  
    //do{ sm1=1; WaitX(0); if (sm1>0) return 1;} while(0);  
  
    do{ sm1=1; do {_lc=(__LINE__%255)+1; return 0 ;} while(0); case (__LINE__%255)+1: ; if (sm1>0) return 1;} while(0);  
  
    LED1=!LED1;  
  
}  
  
//_EE  
  
;}; _lc=0; return 255;  
  
}
```

第一次执行的时候

因为初始化的时候给每个 **timers** 里面的值都赋为 **0**；在执行主函数的时候先从任务函数 **1** 执行；执行到 **do{ sm1=1; do {_lc=(__LINE__%255)+1; return 0 ;}**的时候，把行号给了局部静态变量 **_lc**；执行 **return 0** 的效果是结束函数，并把 **0** 返回给 **timers[1]**。结合主函数分析。（这个时候因为用到了 **WaitX(0)**，所以调用函数用 **RunTask**，没有 **continue** 的那个）

```
while(1){  
  
    { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); } }  
  
    { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); } }  
  
}
```

可以知道，执行完 **return 0**，就跳到主函数任务 1 函数中；接着执行 **RunTask(task2,2)**，也是因为第一次运行时初始化的时候清零了条件满足，执行

RunTask(task2,2)。

原函数如下：

```
void task2(){
```

```
_SS
```

```
while(1){
```

```
    WaitX(100);
```

```
    SendSem(sm1);
```

```
}
```

```
_EE
```

}

在程序执行到 `WaitX(100);` 的时候把行号放到 `_lc` 中，`return 100`，返回到主函数中；主函数执循环扫描到 `RunTask(task1,1)`；因为上次执行 `RunTask(task1,1)` 的时候 `timers[1]` 返回值为 0，这次执行判断的时候条件满足 (`timers[1]==0`)，执行 `timers[1]= task1`；根据上次 `_lc` 里面的值，执行 `switch(_lc)`，跳转到 `case (__LINE__%255)+1` 后面执行代码，查看原函数，并展开后面的函数为：

```
do{ sm1=1; do {_lc=__LINE__%255)+1; return 0 ;} while(0); case (__LINE__%255)+1: ; if (sm1>0) return 1;} while(0);
```

这时候因为 `sm1` 为 1（第一次执行任务函数 1 的时候初始化为 0 接着，执行 `WaiSem` 的时候给 `sm1` 赋值为 1），`if` 条件满足执行 `return 1`；这时 `timers[1]=1`；返回到主函数中；主函数执循环扫描到 `RunTask(task2,2)`；`if` 条件不满足（这个时候 `timers[2]=100`）；又循环扫描执行 `RunTask(task1,1)`，`if` 条件不满足。。。。。。这样反复循环扫描，等待定时器把 `timers` 减到 0。10MS 定时到 `RunTask(task1,1)` 里面 `if` 条件满足，执行 `case` 后面的函数，`case (__LINE__%255)+1: ; if (sm1>0) return 1;} while(0);` 函数返回，主程序继续循环扫描。。。。。。

100ms 定时到，扫描到 `RunTask(task2,2)`，`if` 条件满足执行，跳转到 `case` 后面的语句执行 `SendSem(sm1)`；我们把宏展开 `do {sem=0;} while(0);` 程序执行把 `sem` 赋值为 0，因为任务函数 2 里面 `while(1)` 的原因继续执行花括号里面的语句。执行到 `WaitX(100)`；记录行号，返回 100。在主循环中循环扫描 `RunTask(task1,1)`，这个时候分两种情况：

情况一：（感觉遇到的情况很少，也是正常情况）

如果前一个时刻任务函数返回的是 `return 0`；这时 `RunTask(task1,1)` 里面的 `if` 条件满足执行 `case` 后面的语句。

```
while(1){
```

```
    // WaiSem(sm1);
```

```
    //do{ sm1=1; WaitX(0); if (sm1>0) return 1;} while(0);
```

```
    do{ sm1=1; do {_lc=(__LINE__%255)+1; return 0 ;} while(0); case (__LINE__%255)+1: ; if (sm1>0) return 1;} while(0);
```

```
    LED1=!LED1;
```

```
}
```

因为 sm1 被赋值为 0，if 条件不满足，不执行 return 0。执行 LED1=!LED1;由于任务函数 1 里面 while(1)的原因继续循环执行花括号里面的循环体。

```
do{ sm1=1; do {_lc=(__LINE__%255)+1; return 0 ;} while(0); case (__LINE__%255)+1:; if (sm1>0) return 1;} while(0);
```

把 sm1 赋值为 1，记录行号，返回 0；主循环循环扫描执行。。。。。

情况二：（正常情况）

如果前一个时刻任务函数返回的是 return 1，这时 RunTask(task1,1)里面的 if 条件不满足，循环扫描 RunTask(task2,2)，if 条件不满足，循环扫描

RunTask(task1,1)和 RunTask(task2,2)，等到定时器把 timers[1]减到 0。扫描到 RunTask(task1,1)，里面 if 条件满足。执行 case 后面的语句。

因为 sm1 被赋值为 0，if 条件不满足，不执行 return 0。执行 LED1=!LED1;由于任务函数 1 里面 while(1)的原因继续循环执行循环体里面的内容。

`do{ sm1=1; do {_lc=(_LINE_%255)+1; return 0 ;} while(0); case (_LINE_%255)+1;; if (sm1>0) return 1;} while(0);`把 `sm1` 赋值为 1，记录行号，返回 0，主循环循环扫描。。。。。

情况三：（估计也没有人这么用。）

如果任务 1 里面发送信号量，且任务 1 上电只运行一次，是马上运行。

任务 2 里面初始化信号量和等待信号量

这个时候任务 2 因为初始化信号量，从而导致发送信号量丢失。

估计也没有人这么用。除非写程序的时候没注意写的。

当然了，如果任务函数 1 调用任务函数 2，情况大同小异，分析略。

正常情况下发送完信号量后，下一个时基开始响应信号量。

作者还写了一种等待信号量或定时器溢出的宏如下：

```
#define WaitSemX(sem,tickets) do { sem=tickets+1; WaitX(0); if(sem>1){ sem--; return 1;} } while(0);
```

分析略，原理大同小异。作者在注释中也说了：等待信号量或定时器溢出，定时器 `tickets` 最大为 `0xFFFFE`；这里需要注意的是等待信号量里面的 `tickets`

和 `WaitX(tickets)`里面的 `tickets`，是不一样的，一个最大值为 `0xFFFFE`，另一个最大值为 `0xFE`。（等待信号量里面的是不是修改一下？？如第一个字母

大写??)

如果任务函数 1 里面有条语句是 `do{ waitx(0);}while(TIMEOUT)`，本来是想在等待溢出的时候释放一下 CPU 的资源，让程序扫描执行 `WaitX(tickets)` 等待时间到的程序。但是由于不小心在主循环里面用 `RunTaskA` 来调用任务函数（有 `continue` 的）。就跟就没有释放 CPU 的资源，一直在任务 1 里面执行。我们分析如下：

把宏定义：`#define WaitX(tickets)` 展开如 `do {_lc=(__LINE__%255)+1; return tickets ;} while(0); case (__LINE__%255)+1:`

加入在执行任务函数 1 的时候，执行到 `waitx(0)` 了，我们展开分析一下，`do {_lc=(__LINE__%255)+1; return 0 ;} while(0);` 上半句是把行号放到 `_lc` 中，并返回 0，结束任务函数 1；CPU 跳转到主函数中执行，我们把主函数中运行任务函数 1 的宏替换一下

```
{ if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} }
```

把返回值 0 赋值给 `timers[1]`；运行 `continue` 语句，结束本次循环，从循环体的开始执行下一次循环；这样 CPU 还是运行任务函数 1，在任务函数 1 里面根据标号跳转到 `case (__LINE__%255)+1:` 执行后面语句；因为后面是花括号，程序继续执行 `waitx(0)`；这样在等待响应信号期间 `cup` 一直在任务 1 里面循环 这样就没有释放 CPU 的资源。

在信号量的应用中把几个宏展开后里面都有 `WaitX(0)`；

如：

```
#define WaitSem(sem) do{ sem=1; WaitX(0); if (sem>0) return 1;} while(0);
```

```
#define WaitSemX(sem,tickets) do { sem=tickets+1; WaitX(0); if(sem>1){ sem--; return 1;} } while(0);
```

如果用到信号量，建议主函数中用 `RunTask(TaskName,TaskID)` 调用函数。

下面是一种错误的用法，就是在主函数中用 `RunTaskA` 调用任务函数，且任务函数中用到 `WaitX(0)`，这是种错误的用法，是由于 `continue`，的语句造成的，为什么不行分析如下；还是以信号量的应用为例，只不过主循环里面用的 `RunTaskA`，其余不变。其实不用看，这样会导致我们编写程序的时候不小心出现错误。当然有时候还是没有问题的。下面这种情况中，感觉没有什么问题，但我们在实际中不会这样分析，不要在 `RunTaskA` 里面用 `WaitX(0)`。

第一次执行的时候

因为初始化的时候给每个 `timers` 里面的值都赋为 0；在执行主函数的时候先从 `RunTaskA(task1,1)` 执行；执行到 `do{ sm1=1; do {_lc=(_LINE_%255)+1; return 0 ;}`的时候，把行号给了局部静态变量 `_lc`；执行 `return 0` 的效果是结束函数，并把 0 返回给 `timers[1]`。结合主函数分析。（注意这时候是有 `continue;`的，这个将导致 `WaitX(0)`释放不了 CPU，或者虽然释放了 CPU 但是没有达到我们的预期效果。）

```
while(1){  
  
    { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} }  
  
    { if (timers[TaskID]==0) {timers[TaskID]=TaskName();continue;} }  
  
}
```

可以知道，执行完 `return 0`，就跳到主函数 `RunTaskA(task1,1)` 函数中，执行 `continue`; 结束本次循环，由于 `continue`; 作用在循环体的效力，使 `cpu` 从循环体的开始执行下一次循环，也就第二次执行 `RunTaskA(task1,1)`。

第二次执行的时候：

因为 `RunTaskA(task1,1)` 写在循环体的开始，执行 `RunTaskA(task1,1)`，前一次执行时 `_lc=(__LINE__%255)+1`，所以第二次执行的时候根据 `switch(_lc)` 查找标号，跳转到 `case (__LINE__%255)+1:` 后面的函数了。这里多了一个分号，里面什么也没有，相当于空语句，继续往下执行。执行到 `return 1;` 这样就把返回值赋值给 `timers[1]=1`，并退出函数。

第二次执行的时候执行到 `return 1`，也就是把 `timers[1]` 赋值为 `1`。下面我们查看一下宏

```
#define RunTaskA(TaskName,TaskID) { if (timers[TaskID]==0) {timers[TaskID]=TaskName(); continue;} } //前面的任务优先保证执行
```

把返回值赋值给 `timers` 后，执行 `continue`; 命令，退出本次循环，然后从主函数 `while(1)` 循环体的开始重新执行（准备第三次）。

任务函数 `return 1` 后面的语句就执行不了了。

第三次扫描主函数里面的任务函数

```
while(1){  
  
    // RunTask(task0,0);  
  
    RunTaskA(task1,1);//任务 1 具有比任务 2 高的运行权限  
  
    RunTaskA(task2,2);//任务 2 具有低的运行权限  
  
}
```

执行到 `RunTaskA(task1,1);`，因为 `if (timers[TaskID]==0)` 条件不符合，所以，执行 `RunTaskA(task2,2);`。

第一个任务函数执行了两次，但都是判断语句，里面没有实质的动作。

当 10MS 过去后，再次调用 `RunTaskA(task1,1);` 的时候，`if` 条件满足，执行任务函数，由于任务函数里面 `while(1)` 的原因，程序还在花括号里面运行。

重复上面的循环。

任务函数 2 展开：

```
void task2(){  
  
_SS  
  
while(1){
```

```
WaitX(100);
```

```
SendSem(sm1);
```

```
}
```

```
_EE
```

```
}
```

把宏替换一下

```
void task2(){
```

```
//_SS
```

```
static unsigned char _lc=0; switch(_lc){default:
```

```
while(1){
```

```
// WaitX(100);
```

```
do {_lc=(__LINE__%255)+1; return 100 ;} while(0); case (__LINE__%255)+1:
```

```
//SendSem(sm1);
```

```
do {sm1=0;} while(0);

}

//_EE

;}; _lc=0; return 255;

}
```

这是第一次运行任务函数 2

当运行到 `return 100;` 的时候返回；等到定时器把 100 个时间单位减到 0 的时候运行任务函数 2，并跳转到 `case (__LINE__%255)+1:` 执行函数，`do {sm1=0;} while(0);`如果还有其他指令也一起执行了，这样就把 `sm1` 赋值为 0；因为这个时候是在任务函数 2 里面的 `while(1){}`里面执行，CPU 循环执行 `do {_lc=(__LINE__%255)+1; return 100 ;} while(0); case (__LINE__%255)+1:`，碰到 `return 100`，结束任务函数 2；转去执行主函数 `RunTaskA(task2,2)` 里面的 `continue`，结束任务 2，转到循环体的开始进行执行。

先运行任务函数 1，判断条件是否满足；如果满足执行任务函数 1，不满足执行任务函数 2；接着判断条件 2 是否满足，满足执行，不满足转到任务 1，如此反复循环；直到定时器中断里把时间单位减到 0，条件满足执行任务函数。

其实在下一个时间单位到来时，任务函数 1 就满足了，执行并跳转到任务函数 1 里面的 `if (sm1>0) return 1; while(0); LED1=!LED1; }`

因为 `sm1==0`，就不执行返回函数了，而执行 `LED1=!LED1;`

执行完毕后，因为在任务函数 `while(1)` 的循环体里继续执行，碰到 `return`，返回主函数循环体内；继续执行任务 1，遇到 `continue`，退出，从头开始。。。。。。

（六）、子函数的调用。

在 `task1` 中调用 `task2`，系统的机理是先挂起 `task1`（而不是终止）；在 `task1` 的下一个时间片到来时再运行 `task2`（`WaitX(tickets)` 写在前面，子函数写在 `waitx` 的后面），然后等 `task2` 执行完毕后，再继续执行 `task1`。子函数不需要写在主函数中，在任务函数调用就可以了；可以调用 1 次，多次，和每次调用，根据具体情况编写。

```
unsigned char task11(){
```

```
    static int i;
```

```
    _SS
```

```
    for(i=0;i<11;i++){
```

```
        WaitX(10U);
```

```
        LED2=!LED2;
```

```
    }
```

```
    _EE
```

```
}
```

```
unsigned char task1(){
```

```
_SS
```

```
while(1){
```

```
    WaitX(100U);
```

```
    LED0=!LED0;
```

```
    CallSub(task11);
```

```
}
```

```
_EE
```

```
}
```

```
unsigned char task2(){
```

```
_SS
```

```
while(1){
```

```
    WaitX(10U);
```

```

    LED1=!LED1;

}

_EE

}

*****

while(1){

    // RunTask(task0,0);

    RunTask(task1,1);//任务 1

    RunTask(task2,2);//任务 2

}

```

子函数宏定义展开如下：

```
#define CallSub(SubTaskName)
```

```
do {unsigned char currdt; _lc=(__LINE__%255)+1; return 0; case (__LINE__%255)+1: currdt=SubTaskName(); if(currdt!=255) return currdt;}
```

```
while(0);
```

WaitX(tickets)宏定义展开如下

```
#define WaitX(tickets)
```

```
do {_lc=(__LINE__%255)+1; return tickets ;} while(0); case (__LINE__%255)+1:
```

这里我们把 WaitX(100U); 和 CallSub(task11);宏定义展开

```
unsigned char task1(){
```

```
_SS
```

```
while(1){
```

```
    //WaitX(100U);
```

```
    do {_lc=(__LINE__%255)+1; return 100 ;} while(0); case (__LINE__%255)+1:
```

```
    LED0=!LED0;
```

```
    //CallSub(task11);
```

```
    do {unsigned char currdt; _lc=(__LINE__%255)+1; return 0; case (__LINE__%255)+1:    currdt=task11(); if(currdt!=255) return currdt;}
```

```
while(0);
```

```
}
```

```
_EE
```

```
}
```

第一次执行到 `WaitX(100U);` 的时候返回 100 个时基。100 个时基到了，执行 `LED0=!LED0;` 和 `CallSub(task11);` `cpu` 定义一个局部变量 `currdt`，把行号放入 `_lc`，返回 0，`cup` 在主函数里面循环扫描执行，因为返回值为 0([这里释放了一下 CPU](#))，所以继续执行 `RunTask(task1,1)`，这次执行 `switch (_lc)` 时跳转到 `case (__LINE__%255)+1:` 后面执行 `currdt=task11(); if(currdt!=255) return currdt;` `while(0);` 语句。其中 `task11()` 子函数宏定义展开如下：

```
unsigned char task11(){
```

```
    static int i;
```

```
//_SS
```

```
static unsigned char _lc=0; switch(_lc){default:
```

```
    for(i=0;i<11;i++){
```

```
        //WaitX(10U);
```

```
do {_lc=(_LINE_%255)+1; return 10 ;} while(0); case (_LINE_%255)+1:

    LED2=!LED2;

}

//_EE

;; _lc=0; return 255;

}
```

运行 `currdt=task11()`；把子函数 `task11()` 里面的运行结果放在 `currdt` 中，返回值是等待的时基数；子函数把 `return 10` 返回，这时 `currdt` 为 10。

执行 `if(currdt!=255) return currdt;}` `while(0)`；把 10 返回到主函数中；10 个时基过去后，重新调用 `RunTask(task1,1)`；执行任务函数里面 `case (_LINE_%255)+1` 的程序 `currdt=task11()`；`if(currdt!=255) return currdt;}` `while(0)`；再次执行把 `task11()` 子函数的值返回到 `currdt` 中，这个时候是调用 `task11()` 函数里面的 `case` 后面的程序：`LED2=!LED2`；同时把地址放入子函数 `_lc` 中（`_lc` 是局部静态变量），如此反复 11 次后，最后一次执行子函数 `task11()` 里面的 `;; _lc=0; return 255; (_EE)` ——因为在任务函数 1 里面有判断指令 `if(currdt!=255) return currdt;}` `while(0)`；继续在任务函数里面执行，准备下一次重新调用子函数。。。在子函数和任务函数中行号保存在各自的局部变量中，相互没影响；子函数中返回的只有延时基数。

还有 1 个简单的程序是一位网友问作者的，我感觉也比较经典，拷贝过来，方便大家参考。

问：做个最简单的任务；一个按键,LED 输出。

功能:

1.上电后,LED 关闭。2.当按键按下,LED 常亮。

3.当松开按键后,LED 按以下顺序输出:

A.关 0.1 秒,开 0.1 秒.

B.关 0.2 秒,开 0.2 秒.

C.关 0.3 秒,开 0.3 秒.

D.关 0.4 秒,开 0.4 秒.

E.关 0.5 秒,开 0.5 秒.

E.关 1 秒,开 1 秒.

F.关闭 LED.

4.在按键松开后,可重新触发.同时要求按键 5 次. 5ms/次去抖.

这怎么做?

作者:

示意代码(统一按 20ms 去抖的):

- 1.
2. unsigned char SubTask1
3. {
4. _SS
- 5.
6. LED=0; //关闭 LED
7. WaitX(10);
8. LED=1; //打开 LED
9. WaitX(10);
- 10.

11. LED=0; //关闭 LED

12. WaitX(20);

13. LED=1; //打开 LED

14. WaitX(20);

15.

16. LED=0; //关闭 LED

17. WaitX(30);

18. LED=1; //打开 LED

19. WaitX(30);

20.

21. LED=0; //关闭 LED

22. WaitX(40);

23. LED=1; //打开 LED

```
24. WaitX(40);  
25.  
26. LED=0; //关闭 LED  
27. WaitX(50);  
28. LED=1; //打开 LED  
29. WaitX(50);  
30.  
31. LED=0; //关闭 LED  
32. WaitX(100);  
33. LED=1; //打开 LED  
34. WaitX(100);  
35.  
36. LED=0;
```

37.

38._EE

39.}

40.

41.unsigned char Task1

42.{

43.static unsigned char i;

44._SS

45. while(1){

46. LED=0; //关闭 LED

47. while(KEY) WaitX(0);

48.

49. LED=1; //打开 LED

50. WaitX(2);//20ms 按键去抖

51.

52. while(!KEY) WaitX(0);

53.

54. CallSub(SubTask1);

55.

56. for (i=0;i<5;i++){

57. while(KEY) WaitX(0);

58. WaitX(2);//20ms 按键去抖

59. while(!KEY) WaitX(0);

60. WaitX(2);//20ms 按键去抖

61. }

62. }

63._EE

64.

65.}

对于这个等待 `while(KEY) WaitX(0);` 有的网友认为用 `do{ waitx(0);}while (KEY)` 这样更合理。

四、其他宏定义

五、小小调度器使用注意事项。

在主函数里面，如果用了 `RunTaskA`，就不要在任务里面有 `WaitX(0)`；如果要用 `WaitX(0)`，主函数里面就因该用 `RunTask`，而不是 `RunTaskA`。

`unsigned char timers[MAXTASKS]` 定义数组类型可知最多有 256 个任务函数，

也就是说 `main` 主循环里面最多有 256 个任务函数

WaitX(1)和 WaitX(0)的区别

waitx(0)释放 CPU 资源，立即继续执行

waitx(0)不带来调度级别的等待。

只是释放 cpu 一下，立即回来继续运行，释放 CPU 资源，以便其他任务得到执行机会。否则一个任务阻塞占用的时间就太多了。实际上在任务的阻塞处，

插入 waitx(0)，会变得让任务不那么阻塞。如果每个任务都处理得不怎么阻塞了，整个并行效果就好了。

六、小技巧

1.比如某个任务，要阻塞延时 2 毫秒,（小于最小时基，如果大于或者等于就能用 WaitX(tickets);）。显然如果 delay_us(2000); 会导致 2 毫秒阻塞。

但是用

```
for(i=0;i<20;i++){
```

```
    delay_us(100);
```

```
    waitx(0);
```

```
}
```

可以不那么堵塞。当然了，这么用一定要注意，因为这里面有 delay_us(100); 这就要消耗时基里面的时间。

2.用在等待外部或者内部信号上，

```
do
{
    waitx(0);
}while( TIMEOUT )
```

问：如果用到串口通讯呢，如何处理

答：接收和发送都是用中断资源，任务里面只是简单的处理接收和要发送的数据，同时控制发送开启功能，效率会很高，基本上不会对其他任务造成影响。

思路：

```
void UartInter(void) interrupt 5 /* 串口中断服务函数 不记得 51 的串口是不是 5 了 */
{
    if(接收到一个字节数据)
    {
        将数据存放到定义好的缓冲区里
        清除接收标志位
    }
    else //发送数据中断启动了 一般启动发送中断后 只要发送缓冲器 SBUF 为空就会产生中断申请
    {
        SBUF = 发送的数据;
        if(i == 发送的个数)
        {
            i = 0;
            关闭发送中断功能; //在任务里开启发送功能
        }
    }
}
```

```
    }  
}  
} //完成
```

```
void task3(void)  
{  
    处理接收到的数据 或 把需要发送的数据写入自己定义的缓冲区 并且 确定个数  $i = ?$ ;  
    需要发送数据则开启发送中断  
} //完成
```

当然了各种小技巧的应用的前提是对程序的整体有把握，否则会导致一些问题的。

作者问答：

1、任务指令语句长度是不是不能大于 255;waitx()里面小于 256 是不是和这个有关？

答：任务指令语句的行数是可以大于 255 的，在小于 255 行时，是不可能出现行号重复的，在大于等于 255 行后有出现重复的行的概率，导致编译失败，可通过加空行来解决。

waitx()里面小于 255 和这个无关。

2、任务内函数调用 waitx()有什么限制么？

答：WaitX()里面的值，要小于 255。

3、任务间数据的互相访问怎么处理比较好？比如结构体或数组

答：使用全局变量，或自己实现 FIFO。

4、我如果想一个任务只运行一次就挂起自身，有什么办法？可以被其它任务唤醒并立即执行，有什么办法？

答：使用信号量。

1. Ticket 能不能选 1ms，选 1ms 会带来什么问题，需要注意哪些

答. Ticket 可以选 1ms，选 1ms 最大的区别是:好处，延迟更精确，中断级别的任务实时性更好。 坏处：导致 cpu 中断次数变多了，普通任务获得的 cpu 时间变少了，WaitX() 的最大延时变为 254 毫秒。

2. 为什么要求任务局部变量要求定义为静态变量？ 任务内调用的函数是不是局部变量也必须是静态的，任务内调用的子函数内用 switch 没问题吧

答. 因为任务没有自己的堆栈，释放 CPU 后再次继续往下运行时，还需要用上次的变量，所以要求定义为静态变量，以便于任务释放 CPU 后，变量不会消失。 但是任务内调用的函数（子任务函数除外），是用动态局部变量的，也可以用 switch。

3. #define WaitX(tickets) _lc=(unsigned char)(__LINE__)%255U+1U; if(_lc){return (tickets);} break; case ((unsigned char)(__LINE__)%255U)+1U: 这个宏定义中为什么去判断 if(_lc)，感觉_lc 不会等于 0 啊

答.这里的 if(_lc) 纯粹是为了通过 Misra C 规范检查。如果你的编译器没有选中这个 misra c 规则检查，也可以去掉这个 if 判断。

4 ...

main 进入 while(1)循环后，永远不会再有机会去发给 flg 赋值了。所以需要把赋值部分放到其他地方，最好是放在一个任务里面。

另外，建议 main 函数里面就是单纯的调用各个 RunTask 函数，不再判断，要判断可以在任务函数内部去判断。

只要任务函数内部的代码行小于 255 行的， 可以任意连续多行 waitx（当然，也是要小于 255 行），不会出现重复行了。