嵌入式合作式调度器——eCS用户手册

（embedded Coroutine Scheduler，eCS）

作者：刘海成

email：liuhaicheng@126.com

目 录

一 eCS的移植

eCS（embedded Coroutine Scheduler，eCS）是基于协程（Coroutine）思想的合作式调度器，任何应用的执行都是靠事件来触发。触发事件可以是外部设备、内部定时器、其它任务发送的信号量或消息等事件。eCS的上下文切换不用人工设定处理堆栈，所有任务共享同一个堆栈空间。eCS的实现与风格近似于操作系统，适合于MCU和中低端MPU应用。

eCS支持3种协程线程。一是单纯的时间触发线程，eCS中称之为时间触发型线程；二是借鉴Protothreads思想的线程，eCS中称之为LOOP型线程；三是指针法FSM，eCS中称之为FSM型线程。LOOP型线程和FSM型任务也都支持时间触发；带有状态转换语句的时间触发型函数和线程函数都可以作为FSM型任务的一个状态；LOOP型线程也可以调用子FSM。显然，eCS支持复杂的HFSM，应用灵活多变，方便使用。三种任务都支持优先级和消息。eCS最多可支持255个线程，当然是不会用这么多线程的。

eCS采用全标准C语言实现，移植方便，只须一个具有自动重载功能的硬件定时器（嘀嗒定时器）为其提供调度节拍的“心跳”即可，以实现时间触发。

1 相关文件：eCS_cfg.h、eCS.h、eCS.c

鉴于嵌入式系统的有限资源，eCS的代码基于函数和宏来实现。借用C/C++的代码的预处理功能，以宏为主。

eCS只涉及3个文件，将这三个文件复制到自己的工程目录下，在工程中包含“eCS.h”，此头文件会自动包含“eCS_cfg.h”。

使用时，根据不同的处理器和应用需要简单配置“eCS_cfg.h”即可。

2 配置eCS_cfg.h文件

eCS_cfg.h内有多个地方可配置，如下：

1、定义CPU类型

```
#define CPU_CortexM            0
#define CPU_AVR8               1
#define CPU_51                 100   //具体的51 CPU要定义为大于100的值
#define CPU_51_MCS51TIMER2     101   //与经典型51的Timer2兼容的
                                     //51单片机，Timer2作为嘀嗒定时器：
                                     //AT89/STC89/MPC82G516/C8051F020
#define CPU_51_STC12C5A60S2    102   //PCA作为嘀嗒定时器
#define CPU_51_IAP15W4K58S4    103   //Timer4作为嘀嗒定时器
```

其他CPU自2依次向下排。

确定CPU的方法，以使用ARM Cortex-Mx为内核的处理器为例：

```
#define CPU_TYPE                CPU_CortexM
```

2、定义调度系统时钟节拍时间

```
#define eCS_SYSTICK_MS          1
```

调度系统时钟节拍eCS_SYSTICK_MS的时间单位是 ms，一般配置为 1~10ms 即可，对

于执行速度快的单片机选如Cortex-Mx、STM8s、AVR等配置 1ms， 传统的51 单片机速度慢的选5ms以上比较好。

3、资源与效率的平衡

采用双向链表实现eCS会有很高的调度效率，但会增加一个方向的链表指针的内存消耗。采用其他方式则调度效率上有了一点影响。

```
#define _eCS_SaveRAM                0
```

这个宏定义为0，则采用节约内存的方式，采用单向链表；设定为1则是追求高效调度模式，采用双向链表。

4、确认是否为GUN C编译器

```
#define _eCS_CompilerIsGUN_C99      1
```

若是GUN 或C99编译器，要优先将该宏设置为1，因为GUN 或C99编译器的阻塞跳转效率更高。

5、以下是根据不同的CPU需要修改的内容，各个CPU条件编译方式写好，则一劳永逸。添加新的CPU时，各个宏含义如下：

（1）使能和关闭总中断的API定义

```
#define  _eCS_EnableIRQ()           __enable_irq()
#define  _eCS_DisableIRQ()          __disable_irq()
```

不同的CPU要修改使能和关闭总中断的的方法。默认是Cortex-Mx的MDK函数。

（2）总中断是否使能的开机默认情况

```
#define  _eCS _GLOBAL_INTERRUPT_DEFAULT_ENABLE      1
```

开机总中断默认是否已经使能（开启）：1-开启 0-关闭

51、AVR等都是0，Cortex-Mx设置为1。

（3）嘀嗒定时器中断函数

```
#define  _eCS_SysTick_OverflowIRQ()  SysTick_Handler()
```

嘀嗒定时器中断函数宏定义，以自动适应不同单片机中断函数的写法。

该函数中不用处理总中断的使能问题，只要使能嘀嗒定时器中断即可。

（4）滴答定时器初始化

```
#define  _eCS_SysTickInit()                                  \
SysTick_Config(SystemCoreClock/1000* eCS_SYS_TICKS_MS)
```

调度器滴答定时器初始化宏定义，据不同CPU要修改或编写相应的初始化函数。

（5）嘀嗒定时器中断标志处理

```
#define  _eCS_SysTick_InterruptFlag_IsAutoClear    1
或
#define  _eCS_SysTick_InterruptFlag_IsAutoClear    0
#define  _eCS_SysTick_InterruptFlagClear()         具体的清除标志语句
```

定义作为嘀嗒定时器的定时器，执行ISR时是否自动清除中断标志，不自动清则必须给出清中断标志的语句。

（6）以下宏用于实现微妙延时

```
#if CPU_TYPE == CPU_CortexM         //Cortex-Mx的嘀嗒定时器作为系统嘀嗒
  typedef _eCS_UINT32  _eCS_SysTick_usReadType;//滴答定时器的统计计数器位宽
  typedef _eCS_UINT32  _eCS_SysTick_usAccType; //滴答定时器的统计计算数值位宽
  #define _eCS_SysTick_Value         (SysTick->VAL)
  #define _eCS_SysTick_ReloadValue  (SystemCoreClock/1000 * eCS_SYSTICK_MS)
  #define _eCS_SysTick_IsUpCounted   0  //1:加法计数器; 0:减法计数器
  #define _eCS_SysTick_CyclesPerUs   CyclesPerUs
#endif
```

以上是借助嘀嗒定时器，为实现微秒级的各类延时API进行设置或提供参数。

例如，对于Cortex-Mx内核的ARM，其嘀嗒定时器是24位减法计数器，所以计数器的重载值_eCS_SysTick_ReloadValue要定义为(SystemCoreClock/1000 * eCS_SYS_TICKS_MS)。配合与其一起运算和统计的变量则要32位，所以统计变量的位宽_eCS_SysTick_usReadType和_eCS_SysTick_usAccType定义为32位（eCS_UINT32）。

读取该嘀嗒定时器当前计数器值_eCS_SysTick_Value的寄存器为SysTick->VAL。

每个微秒的系统时钟个数_eCS_SysTick_CyclesPerUs为CyclesPerUs。

二 eCS的使用及API（包括原语）

1 定义线程调度器（Thread Scheduler，TS）

调度器管理的对象是线程调度器（本质是线程控制块TCB），所以要为每个线程定义自己的线程调度器。例如要建立一个线程vTestTask，则除了编写 vTestTask 线程函数外，还要定义它的线程调度器TS_vTestTask。定义格式为：

```
-API- eCS_TS  TS_vTestTask = {0};   //建议后面有初始化= {0}
```

或直接采用如下方法定义：

```
-API- def_eCS_TS(TS_vTestTask);
```

仿照此方法，我们可以增添和编写更多的线程vTestTaskN，并同时定义好它的线程调度器TS_vTestTaskN。

2 创建线程、启动调度器及线程优先级

当线程编写完并定义好它的线程调度器，就可以创建线程了，eCS所支持的三种线程创建的方法相同：

```
-API- eCS_TaskCreate(TS, Task, priority, Ticks);  //创建线程
```

例如：

```
eCS_TaskCreate(TS_vTestTask, vTestTask, 0, 2);  //创建线程 vTestTask
eCS_TaskCreate(TS_vTestTaskN, vTestTaskN, 1, 0);//创建线程vTestTaskN
```

4个参数分别为：对应线程的线程调度器、线程（函数）名、线程初始时间触发值和优先级。

eCS支持非抢占的插队式优先级。eCS采用在满足时间触发条件并获取到被调度机会后是否主动放弃被调度来体现优先级别，即采取被调度次数占比的多少来体现优先级高低。

eCS将每个线程设置为4级优先级，分别为0、1、2和3，其中0级是最高级。eCS按照所设定的优先级，每2^(优先级)次调度机会执行1次的方法来实现优先级越高被调度次数占比越多。这就可以将需要快速被调度查询事件的线程优先级设置为0，满足时间触发条件并获取到被调度机会后就会被立即调度执行，进而增加其被调度的机会，增强优先级较高的相关线程的实时响应能力。要进一步说明的是，抢占式的高实时的快速处理在eCS中只能在ISR中被直接调用，高实时简单处理后由eCS线程函数再接续处理。

初始化线程的不同时间触发值，可使得各个线程错开运行，从而提高效率。例如，若A和B两个线程都是间隔10个"eCS_SYSTICK_MS"运行一次，如果延迟初值相同，则会同时在第10个"eCS_SYSTICK_MS"时满足运行条件，则A、B会先后运行；但如果初始化时，A线程计数器初值为0，B线程计数器初值为1个"eCS_SYSTICK_MS"，则A、B线程会相差1个"eCS_SYSTICK_MS"满足运行条件，即尽可能在不同的TICK内错开运行，从而提高效率。

若是FSM型线程，创建线程时的函数名一般是其Idle状态函数名，如：

```
eCS_TaskCreate(TS_vTestIdleState, vTestIdleState, 1, 3);
```

创建的各个线程可以是分别为独立的线程，也可能是几个线程构成一个进程型线程，当然也可能所有线程形成的只是一个进程。

线程创建好后，要进行调度器启动，以实现嘀嗒定时器等的初始化，启动方法如下：

```
-API- eCS_Start();
```

3 main()函数及调度函数

main函数的while(1) 中放置调度函数：

```
-API- eCS_TaskSchedule();
```

main函数的while(1) 中还可以写其他合作式调度器的调度函数，比如Protothreads函数，从而可以直接将合作式的应用直接移植过来，无需修改。

该因此，对于一个完整的eCS调度系统，main 函数至少应该包含如下部分：

```
int main(void)
{
    sys_Init();                 //硬件系统初始化

    eCS_UserTaskCreate();    //创建线程
    eCS_Start();                 //调度器的嘀嗒定时器等初始化
    while(1)
    {
        eCS_TaskSchedule();

        //这里可以写其他合作式调度器的调度函数，比如Protothreads函数，
        //从而可以直接将合作式的应用直接抑制过来，无需修改
    }
}
```

4 时间触发型线程及定义

eCS支持时间触发型线程、Protothreads思想的LOOP型线程，以及FSM型线程。

时间触发型线程适合于满足时间触发条件就被执行一次的线程。时间触发型线程的函数定义：

```
eCS_TT(TaskName) //或eCS_Thread(TaskName)，或eCS_State(TaskName)
{
    用户变量定义，存在记录历史的变量需要加static
                :
    具体的线程语句                  //可选
    eCS_SleepMe(Ticks);             //下次时间触发设置语句，必选
                                    //可与条件语句连用
}
```

时间触发型线程，满足时间触发条件后被调度，线程完成后通过

```
-API- eCS_SleepMe(Ticks);
```

语句设置为再经过Ticks个嘀嗒后该线程再次被调度。

时间触发型线程没有函数内的断点问题，每次都是从头开始运行，所以，记录“历史”的变量需要加static关键字。这是时间触发型线程函数与其它eCS协程函数（线程函数和先状态等待的一般状态函数）的本质区别。

一个具有两种闪烁周期并自动切换的LED闪烁控制实例：

```
eCS_TT(LedToggle)
{
    static uint8_t sign_Counter = 0;

    //具体的线程语句
    LED2 = !LED2;

    //下次被调度的时间触发设置
    if(++sign_Counter < 40)         //周期式时间触发40次，闪烁20下
    {
        eCS_SleepMe(100);           //时间触发设置语句
    }
    else if(++sign_Counter < 60)    //周期式时间触发20次，闪烁10下
    {
        eCS_SleepMe(500);           //时间触发设置语句
    }
    else
    {
        sign_Counter = 0;           //变量恢复到初始值
        eCS_SleepMe(100);           //时间触发设置语句
    }
}
```

5 LOOP型线程及定义

eCS的LOOP型线程是基于Protothreads思想实现的，是对Protothreads的优化和重构。eCS的LOOP型线程函数的定义：

```
eCS_LOOP(TaskName)                    //或   eCS_Thread(TaskName)
{
    用户变量定义，存在断点问题的变量需要加static
    eCS_Begin();                                                    -API-
    static型的用户变量初始化
    Task_init();                      //可选
    while(1)
    {
        具体的线程语句                //可选
        协程语句(.., ..);             //必选，可以多个
        具体线程语句                  //可选
    }
    eCS_End();                                                      -API-
}
```

【注意】:

（1）LOOP型线程函数的主体部分以**eCS_Begin();**开始，以**eCS_End();**结束；

（2）LOOP型线程函数内必须是**无限循环**，不能使用return语句来结束或删除线程。线程的结束或删除有专门的API和相应的约束条件；

（3）线程中必须有延时、等待信号量、调用子线程或调用子FSM等**yield语句**，主动出让CPU使用权；

（4）线程函数中使用的非静态局部变量会在线程出让CPU后，变量消失，因此要想某个变量不消失，变量数据保持历史信息，则要使用静态变量（用static定义变量）；

（5）基于switch的协程，while循环中不要再出现switch结构。当然，若出现的switch结构中再没有任何协程语句是可以的，但是最好不用，容易混乱，很“危险”；

（6）基于switch的协程，while循环中，无论什么结构都不要出现break语句，以防止与switch的阻塞混为一个结构。当然，没有协程阻塞语句的循环语句中出现break是可以的，但也要慎用。但是可以使用continue语句。

LOOP型线程函数中可以随意调用子函数，但是子函数必须有足够的时间效率，否则在执行该子函数时失去了协程的本质。如果子函数输入时间冗长型，则该子函数也需要具有协程特性，该函数必须定义为线程子函数。

带协程阻塞语句的子线程函数的书写格式：

```
eCS_SubThread(SubThreadName)
{
    用户变量定义，协程变量需要加static
    eCS_SubBegin();          //与父线程不一样                       -API-
    static型用户变量初始化
```

```
    具体的线程语句                  //可选
    协程语句(..,..);                //至少1个，否则就是一般性子程序，而非子线程
                                    //函数，就没有必要按线程的方式编写子程序了

    具体线程语句                    //可选
    eCS_SubEnd();                   //与父线程不一样                    -API-
}
```

【注意】:

（1）LOOP型线程和子线程函数的开始与结束的写法不同。子线程函数的主体部分以**eCS_SubBegin();**开始，以**eCS_SubEnd();**结束。

（2）子线程函数不可以是无限循环，否则子线程后面程序无法执行，除非有退出死循环的语句；

（3）子线程函数不能通过形参传递参数。可以通过全局变量或信号量等进行参数传递；

（4）子线程函数中必须有延时、等待信号量、调用子线程或调用子FSM等yield语句来主动出让CPU使用权；

（5）子线程函数中使用的非静态局部变量会在线程出让CPU后，变量消失，因此要想线程函数中的某个变量不消失，则要使用静态变量；

（6）同样，基于switch的协程，子线程函数中不要再出现switch结构，也不要出现break语句。当然，没有协程语句的循环语句中出现break是可以的，也要慎用。但是含有循环结构时可以使用continue语句；

（7）LOOP型线程函数或子线程函数调用含有协程阻塞语句的子线程函数不能直接调用，必须采用如下调用方法为

```
-API- eCS_CallSub(SubThreadName);
```

（8）子线程函数中不能直接使用return语句，因为简单的返回没有上下文信息。若程序流程需要中途退出，只能通过如下语句返回到调用该函数的原线程函数的断点处：

```
-API- eCS_SubReturn();
```

（9）每个子线程型函数有且只有一个父线程线程，子线程函数只支持对应线程的重入。子线程函数数量根据实际应用而定；

（10）子线程函数中还可以调用子线程。

（10）子线程函数中还可以调用子FSM。

综上，LOOP型线程和子线程函数中必须要有yield语句。yield语句包括3种：阻塞或条件阻塞yield语句，调用子线程函数，调用子FSM。

图24为LOOP型线程的构成及执行。无论是线程函数，还是子线程函数，都可以调用子线程和子FSM。

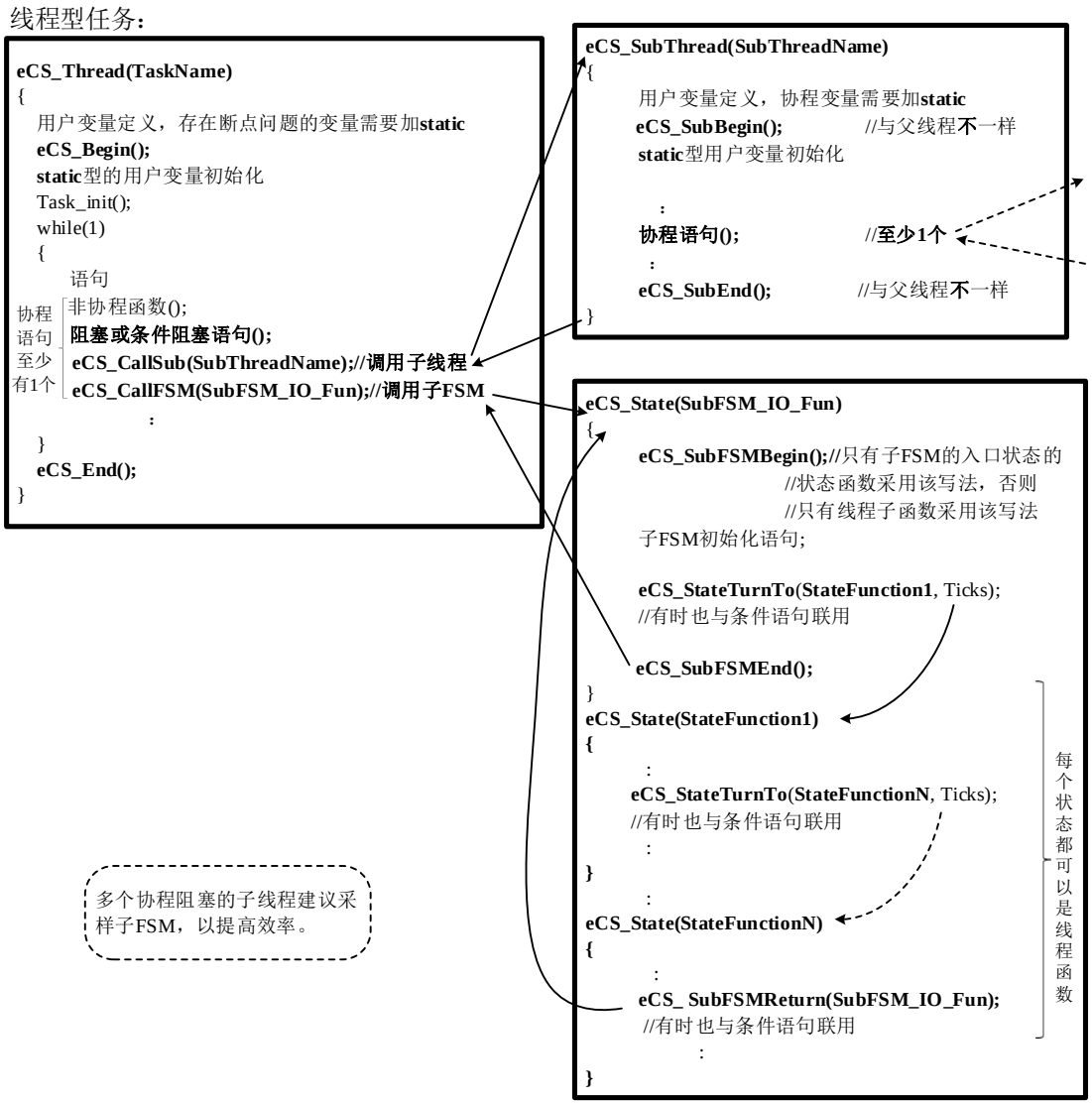

图24 LOOP型线程构成及执行

调用子线程函数前面已经讲述，调用FSM的方法将在后面说明。

6 LOOP型线程及子线程函数中的阻塞和条件阻塞语句

eCS的LOOP型线程及子线程函数中有10个用于yield让出CPU资源的阻塞或条件阻塞语句，直接阻塞返回，释放CPU，避免当前线程过度占用CPU资源。这10个API是LOOP型线程实现协程的关键。

1．时间触发yield设置。

```
-API- eCS_YieldWait(Ticks);
```

用于时间触发yield设置，当前线程延时Ticks个时间节拍后恢复。

当Ticks设为0（eCS_YieldWait(0);）时，始终满足时间触发条件，随后立刻又被调度。

该函数是时间触发思想下LOOP型线程的最常用API。

2．仅仅是yield，使得线程直接转入就绪状态，功能上同eCS_YieldWait(0)，但效率更高，这是因为省去了对时间触发进行赋值的语句。

```
-API- eCS_Yield();
```

该API主要用于分解线程的繁重程度和需要尽快就再次被调度的情况。

3. 条件为真则阻塞，并设定经Ticks时间节拍后继续检测，直至条件为假继续向下运行。若Ticks设置为0，则很快（因为已经处于就绪状态）就进行下一次阻塞判断。

```
-API- eCS_YieldWhile(condition, Ticks);
```

4. 条件为假则阻塞，并设定经Ticks时间节拍后继续检测，直至条件为真继续向下运行。若Ticks设置为0，则很快(因为已经处于就绪状态)就进行下一次阻塞判断。

```
-API- eCS_ YieldUtill(condition, Ticks);
```

基于以上两个宏和需要时对应线程设定的信号量标志可以实现消息机制，等待消息时就是根据该信号量Count实现阻塞，其他线程或中断有消息传递给该线程时则将Count设置为消息的数据数量（不能为0），线程则据情况，通过消息指针到相应的消息数组中获取消息并继续执行，并清除Count（令Count=0）。例如，变量Count，只有别的线程将其改成了非零，就是等到了消息才继续运行，则可写为

```
eCS_YieldUtill(Count, 5);   //没有消息(Count=0)就让出CPU
```

该句的含义是，若条件为假（没有等到消息，即消息不是1）则阻塞，并设定经5个时间节拍后继续检测，直至条件为真继续向下运行。

注意：以上两个API的Ticks参数，可以是常数，也可以是整形变量，但不能是自增等变量表达式。

5．该宏等效为Ticks为0时的eCS_YieldWhile(condition, Ticks)，但效率更高。

```
-API- eCS_YieldWhen(condition);
```

6．该宏等效为Ticks为0时的eCS_YieldUtill(condition, Ticks)，但效率更高。

```
-API- eCS_YieldUnless(condition);
```

7．条件为真则阻塞语句，并超时检测。

```
-API- eCS_YieldWhile_TimeoutControl(condition,Ticks,times,dealwith);
```

该语句主动阻塞Ticks，times（双字节）个Ticks时间（即times次检判）后，若条件仍为真，则说明条件超时，可能有问题，则运行dealwith超时处理方法。如：

```
eCS_YieldWhile_TimeoutControl(condition, 5, 100, fun_warning());
```

表示100次5个Ticks后若条件为真则说明超时，运行fun_warning()方法。

8．条件为假则则阻塞语句，并超时检测语句：

```
-API- eCS_YieldUtill_TimeoutControl(condition,Ticks,times, dealwith);
```

注意：以上两个API的Ticks、times参数，可以是常数，也可以是整形变量，但不能是自增等变量表达式。

9．条件为真则超时判断阻塞的语句，并超时检测。

该语句主动阻塞，但若Ticks时间内不停的被检判后，条件仍为真，则说明条件超时，可能有问题，则运行delwith超时处理方法。

```
-API- eCS_YieldWhen_TimeoutControl(condition, Ticks, dealwith);
```

如：

```
eCS_YieldWhen_TimeoutControl(condition, 5, fun_warning());
```

表示5个Ticks内条件一直为真则说明超时，运行fun_warning()方法。

该API与第7条表述的API的区别在于：持续对阻塞条件检测，时间触发时间就是超时时间。

10．条件为假则超时判断阻塞的语句，并超时检测。

该语句主动阻塞，但若Ticks时间内不停的被检判后，条件仍为假，则说明条件超时，可能有问题，则运行delwith超时处理方法。

```
-API- eCS_YieldUnless_TimeoutControl(condition, Ticks, dealwith);
```

该API与第8条表述的API的区别在于：持续对阻塞条件检测，时间触发时间就是超时时间。

另外，子线程调用语句“eCS_CallSub(SubTaskName);”和子状态机调用语句“eCS_CallFSM(subFSM_IDLE_Name);”都是无条件yield语句。还有6.12节讲述的微秒延时协程语句eCS_YieldWaitus(tus);。

综上，再次强调，eCS的yield语句包括3种：

（1）阻塞或条件阻塞yield语句来让出CPU资源（时间触发是条件阻塞的一种）；

（2）调用子线程函数；

（3）调用子FSM；

其中，条件阻塞事件包括三类：

（1）时间触发（最高优先级）；

（2）一般条件作为条件的条件阻塞型；

（3）信号量作为条件的条件阻塞型，实现支持线程间的消息传送和线程同步等。

7 FSM型线程的状态函数及状态转换

FSM型线程的状态函数：

```
eCS_State(StateFunction)
{


}
```

状态函数StateFunction中描述 StateFunction状态，包括其行为（即状态线程）及状态转换。

eCS的FSM型线程支持4种状态函数：

（1）LOOP型线程函数（LOOP型线程作为FSM的一个状态）；

（2）时间触发函数（时间触发型线程作为FSM的一个状态）；

（3）条件阻塞状态函数；

（4）无阻塞状态函数。

线程函数和时间触发函数作为状态时，关键字都可以写成**eCS_State**，也建议写成**eCS_State**。

线程型协程函数及其子线程函数可以作为FSM型线程的一个状态，LOOP型线的协程状

态函数可直接运用LOOP型线程的各类方法实现协程。

很显然，时间触发型线程和LOOP型线程都可看作只有一个状态的FSM型线程的特例。

每个状态函数中一定有状态转换函数，eCS采用指针指向下一个状态及状态转换函数入口方法实现，且提供了两个状态转换函数。函数如下：

（1）Ticks个滴答时钟后转入pNextStateFunction状态（函数）

```
-API- eCS_StateTurnTo(pNextStateFunction, Ticks);
```

（2）直接转入pNextStateFunction状态（函数）

```
-API- eCS_StateTransferTo(pNextStateFunction);
```

运行状态转换函数则直接协程，让出CPU资源，再次调度该FSM型线程时运行次态状态函数。

FSM型线程的状态函数书写格式如下：

```
eCS_State(FSMStateName)
{
        :
    //一定会有状态转换语句，且状态转换语句经常与条件判断一起使用
    eCS_StateTurnTo(pNextStateFunction,Ticks);或:
    eCS_StateTransferTo(pNextStateFunction);
}
```

时间触发型线程和LOOP型线程作为状态函数前面已经讲述，这里不再赘述。下面给出无阻塞状态函数和条件阻塞型状态函数的书写格式。

无阻塞状态函数的书写格式如下：

```
eCS_State(FSMStateName)
{
    状态转换前要处理的状态线程
    //一定会有状态转换语句，且状态转换语句经常与条件判断一起使用
    eCS_StateTurnTo(pNextStateFunction,Ticks);或:
    eCS_StateTransferTo(pNextStateFunction);
}
```

由于无阻塞状态函数内没有阻塞，所以进入无阻塞状态函数后必须先要完成状态线程，然后立即转入FSM的其他状态。

条件阻塞型状态函数的状态条件阻塞与LOOP型线程的阻塞方法不同，因为条件阻塞型状态函数没有构建Protothreads结构，eCS中为条件阻塞型状态函数的条件阻塞设计了8个API，前4个没有超时检测功能，后4个带有超时检测：

（1）条件为真则保持原状态及断点，并设定经Ticks时间节拍后继续检测，直至条件为真继续向下运行。若Ticks设置为0，则很快（因为已经处于就绪状态）就进行下一次状态条件判断：

```
-API- eCS_StateWaitWhile(condition, Ticks);
```

（2）条件为假则保持原状态及断点，并设定经Ticks时间节拍后继续检测，直至条件为真继续向下运行：

```
-API- eCS_StateWaitUtill(condition, Ticks);
```

（3）条件为真则保持原状态及断点，并持续判断，直至条件为真继续向下运行，比“eCS_StateWaitWhile(condition, 0);”效率更高：

```
-API- eCS_StateWaitWhen(condition);
```

（4）条件为假则保持原状态及断点，并持续判断，直至条件为真继续向下运行，比“eCS_StateWaitUtill(condition, 0);”效率更高：

```
-API- eCS_StateWaitUnless(condition);
```

下面4个API带有超时检测功能：

（5）条件为真则阻塞语句，并超时检测：

```
-API- eCS_StateWaitWhile_TimeoutControl(condition, Ticks,
                                        times, dealwith);
```

该语句在条件为真时主动阻塞Ticks，times个Ticks时间（即times次检判）后，若条件仍为真，则说明条件超时，可能有问题，则运行dealwith超时处理方法。dealwith可以是语句，也可以是函数。如：

```
eCS_StateWaitWhile_TimeoutControl(condition,5,100,fun_warning());
```

表示100次5个Ticks后若条件依然为真则说明超时，运行fun_warning()方法。

若Ticks设置为0，则很快（因为已经处于就绪状态）就进行下一次状态条件判断。这时，超时检测变成超次数检测。

（6）条件为假则阻塞语句，并超时检测：

```
-API- eCS_StateWaitUtill_TimeoutControl(condition, Ticks,
                                        times, dealwith);
```

该语句在条件为假时主动阻塞。工作过程及要点与前一个API一致。

注意：以上两个API的Ticks、times参数，可以是常数，也可以是整形变量，但不能是自增等变量表达式。

（7）条件为真则超时判断阻塞的语句，并超时检测，条件为假继续向下运行：

```
-API- eCS_StateWaitWhen_TimeoutControl(condition, Ticks, dealwith);
```

该语句主动阻塞，但若Ticks时间内不停的被检判后，条件仍为真，则说明条件超时，可能有问题，则运行dealwith()超时处理方法。同样，dealwith可以是语句，也可以是函数。如：

```
eCS_YieldWhen_TimeoutControl(condition, 5, fun_warning());
```

表示5个Ticks内条件一直为真则说明超时，运行fun_warning()方法。

该API与前面的API的区别在于：持续对阻塞条件检测，所设定的Ticks是多次阻塞的总超时时间。

（8）条件为假则判断阻塞的语句，并超时检测，条件为真继续向下运行：

```
-API- eCS_StateWaitUnless_TimeoutControl(condition, Ticks, dealwith);
```

该API与前一条API的工作过程及要点一致，只不过阻塞条件是条件为假。

注意：以上两个API的Ticks参数，可以是常数，也可以是整形变量，但不能是自增等变量表达式。

基于以上8个阻塞API的条件阻塞型状态函数语句分为两部分，前一部分是这8个宏中的某一条，且位于状态函数的开始位置，表示处于阻塞状态，后面紧跟着第二部分就是次态初始化（可选）和状态转移语句。

```
eCS_State(FSMStateName)       //条件阻塞型状态的状态函数
{
    //基于以下8个条件状态等待函数可以实现等待事件机制
    eCS_StateWaitWhile(condition, Ticks);
或:eCS_StateWaitUtill(condition, Ticks);
或:eCS_StateWaitWhen(condition);
或:eCS_StateWaitUnless(condition);
或:eCS_StateWaitWhile_TimeoutControl(condition, Ticks, times, dealwith);
或:eCS_StateWaitUtill_TimeoutControl(condition, Ticks, times, dealwith);
或:eCS_StateWaitWhen_TimeoutControl(condition, Ticks, dealwith);
或:eCS_StateWaitUnless_TimeoutControl (condition, Ticks, dealwith);

    状态转换前要处理的事情    //可选
        :
    //一定会有状态转换语句，且状态转换语句经常与条件判断一起使用
    eCS_StateTurnTo(pNextStateFunction,Ticks);或:
    eCS_StateTransferTo(pNextStateFunction);
}
```

8 FSM作为LOOP型线程的断点子线程

eCS中，线程型函数可以直接作为FSM的状态，FSM也可以作为LOOP型线程的子线程，该子线程称为子FSM。但是，若线程函数调用“子FSM”，即FSM作为线程函数的内部子状态机，这时，该子FSM需要专门的接口函数完成线程函数对其进行调用和返回，该接口函数也有固定的写法。

（1）线程函数的子FSM的接口函数书写格式

线程函数的子FSM的接口函数书写格式如下：

```
eCS_State(pSubFSM_IO_Fun)
{
    eCS_SubFSMBegin();  //协程线程的子FSM的接口函数开始定义。
                        //只有子FSM的入口状态的状态函数采用该写法。其他状态
                        //函数，要么是普通状态函数，要么就是线程函数作为一个
                        //状态，不会出现LOOP型线程在子线程函数作为一个状态
    子FSM初始化语句;     //可选，但不要有协程语句或等待返回语句
    eCS_StateTurnTo(pNextStateFunction, Ticks); //一定会有状态转换语句,
                        //有时也与条件语句联用，且pNextStateFunction
                        //多指向IDLE状态，或根据条件进入相应原态
    eCS_SubFSMEnd();    //协程线程的子FSM的接口函数结束定义
}
```

该函数的首要功能是用于进入子FSM。然后顺便进行子FSM的初始化，但不能作为子

FSM的一个状态再次进入。因为该函数的第二功能就是，若利用子FSM返回原线程函数eCS_SubFSMReturn(pSubFSM_IO_Fun)再次调用该函数，则恢复断点，返回到原线程函数。

也就是说pSubFSM_IO_Fun()函数只被调用两次，第一次调用进入子FSM，第二次调用则退出子FSM回到原线程函数断点处。

（2）线程函数调用其子FSM的调用方法

```
-API- eCS_CallFSM(pSubFSM_IO_Fun);
```

（3）由子FSM返回到原线程函数

```
-API- eCS_SubFSMReturn(pSubFSM_IO_Fun);
```

在子FSM的任一状态运行该语句，则由子FSM返回原线程函数。要说明的是，本质是通过再次返回到接口函数后自动返回到原线程函数，因此该语句带有参数：即接口函数的函数名。例如：

```
eCS_State(SubFSM_StateName)
{
    状态相关语句;                          //可选
    eCS_SubFSMReturn(pSubFSM_IO_Fun);  //多与条件语句联用

}
```

eCS有两种HFSM形式，一是，线程型函数作为FSM的一个状态时，其本身就是个子FSM，如图25所示，如果内部再调用子FSM，即LOOP型线程函数作为FSM的一个状态时，其子FSM就是HFSM的大状态机下的小状态机。子状态机的状态也可以再嵌套子状态机。

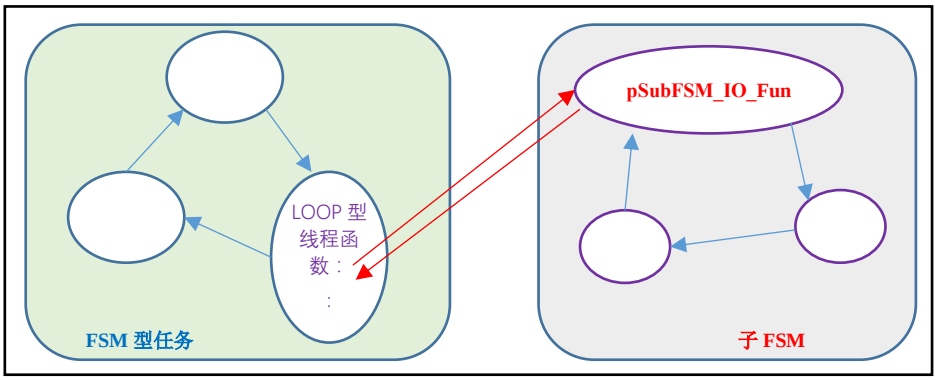

图25 三级HFSM

二是，一个FSM作为某父状态机的一个状态，如图26所示，当进入FSM这个大状态时，直接通过两个状态转换函数之一，由父状态的某状态直接状态转换到该FSM的IDLE状态即可，跳转回父状态机也直接通过状态转换函数实现。

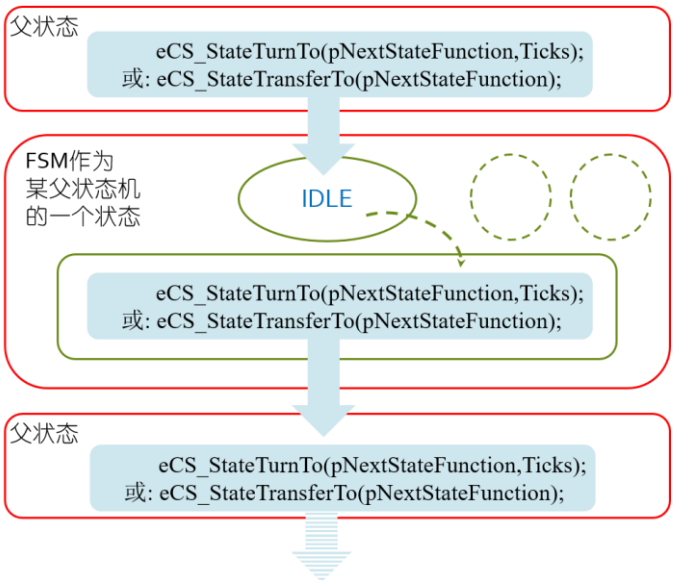

图26 FSM作为HFSM的一个完整的状态

9 信号量和消息的定义及API

eCS支持信号量和消息，eCS将消息队列和邮箱简化为全局消息数组。

当多个线程需要线程同步时需要定义一个或多个信号量。eCS定义的每个信号量包括一个16bit的计数整形变量、一个空指针，以及配合生产者消费者模型的两个16位变量。eCS的每个信号量的定义格式为：

```
-API- eCS_SM  SM_s1;
```

仿照此方法，我们可以增添更多的信号量SM_sN。

eCS还提供了个API用于信号量和消息操作：

（1）信号量自增

```
-API- eCS_SemSignal(&SM);
```

（2）信号量自减

```
-API- eCS_SemReduce(&SM);
```

（3）发送信号量（给信号量赋初值）

```
-API- eCS_SemSet(&SM, v);
```

如：

```
eCS_SemSet(&SM, 5);
```

表示设定信号量SM的数值为5，可以表示有5个信息，也可以表示其他特定含义，由用户自行灵活使用。

（4）线程清信号量(semaphore)

```
-API- eCS_SemClr(&SM);
```

（5）读取线程的信号量(或消息的数量)，只读

```
-API- eCS_Sem(&SM);
```

信号量可以表示消息的数量，也可以作为标志等。比如，读取信号量后，可据信号量的值判断是否有消息，有消息则完成x()功能：

```
if(eCS_Sem(&SM))
{
    x();
}
```

（6）读消息成员内容

```
eCS_Msg(&SM, index, elementType)
```

其中，index是消息成员的角标。elementType是消息成员的数据类型，直接书写类型名称即可，如int。

收到消息时，信号量就是消息的数量。一般是利用读eCS_Sem()的方法获取有消息或消息的数量（大于0），然后读取消息的各个成员。

```
int a[20], i;
for(i = 0; i < eCS_Sem(&SM); i ++)
{
    a[i] = eCS_Msg(&SM, i, int);
}
eCS_SemClr(&SM);
```

注意，读取完成之后还要清信号量。

（7）发出一个邮件给指定的线程，其中:

```
-API- eCS_MsgPost(pSM, pDat, Len);
```

其中，pDat为消息的实际数组首地址。len是消息的数据长度。

eCS支持生产者消费者信号量应用模型，且先生产的先被消费，后生产的后消费，以队列模型实现。以下3个eCS的API用于消费者生产者模型：

（1）初始化

```
-API- eCS_ProducerConsumer_init(pSM, pQueue);
```

该API用于将数组指针pQueue装载到信号量pSM的空指针中，并初始化各个变量，使得信号量作为消费者生产者应用系统使用。

（2）生产

```
-API- eCS_Production(pSM, pushData, elementType, QueueLength);
```

将pushData作为生产的产品压入elementType型数组队列中，数组的长度为QueueLength。elementType是消息成员的数据类型，直接书写类型名称即可，如int。

（3）消费

```
-API- eCS_Consumption(pSM, getData, elementType, QueueLength);
```

将elementType型数组队列中生产的产品取出一个给getData变量进行消费，数组的长度为QueueLength。elementType是消息成员的数据类型，直接书写类型名称即可，如int。

实例：

定义信号量、消息队列，并消费者生产者模型初始化

```
eCS_SM  Sound_sem;
#define  BufferSize  8
uint8_t  VoiceQueue[BufferSize];
eCS_ProducerConsumer_init(&Sound_sem, VoiceQueue);
```

生产：添加1个数据（cmd）

```
eCS_YieldWhen(eCS_Sem(&Sound_sem) == BufferSize);
eCS_Production(&Sound_sem, cmd, uint8_t, BufferSize);
```

消费：取出1个数据（給cmd变量）

```
eCS_YieldUnless(eCS_Sem(&Sound_sem));  //等待信号量
eCS_Consumption(&Sound_sem, &cmd, uint8_t, BufferSize ); //消费
```

10 重置所有线程为初始状态，即重启所有线程

方法为：

```
-API- eCS_SysRestart(eCS_UserTaskCreate);
```

其中，eCS_UserTaskCreate是初始创建各个线程的线程创建函数。

用于系统正在运行时重启所有线程。

11 操作非当前线程API

操作指定线程的API共8个：

（1）时间触发：将指定线程调度器承载的线程延时Ticks个时间节拍后恢复

```
-API- eCS_Sleep(&TS, Ticks);
```

一般不要运用该函数设定自己的时间触发。时间触发型线程，设定时间触发的同时还要有让出CPU资源的功能，API为：

```
-API- eCS_SleepMe(Ticks);
```

LOOP型线程或线程型状态的自身线程时间触发设定还需要带有yield阻塞功能的：

```
-API- eCS_YieldWait(Ticks);
```

（2）重启指定线程。Ticks滴答后重新自pTask函数开始重新调度TS承载的线程。

```
-API- eCS_Restart(&TS, pTask, Ticks);
```

可以运用该函数重启自己，但建议重启自己时才有专门的函数：

```
-API- eCS_RestartMe(pTask, Ticks);
```

不影响重启线程的挂起情况。

（3）关闭指定线程，让出对应线程的CPU使用权限

```
-API- eCS_Close(&TS);
```

至少有两个线程被调度才可以关闭1个线程调度器，因此关闭其它线程一定是没有问题的。

被关闭的线程如果需要再次被调度，需要在其他线程中通过eCS_TaskCreate重新创建，而不可以通过eCS_Resume重新加入调度链表。

不要用该函数关闭自己，因为关闭自己就不能再继续向下执行本线程，用该函数关闭自

己可能会出现致命错误。有专门关闭自己的语句：

```
-API- eCS_CloseMe();
```

（4）挂起（暂停）指定线程。

```
-API- eCS_Suspend(&TS);
```

说明：

①至少有两个线程被调度才可以挂起1个线程调度器，因此挂起其它线程一定是没有问题的；

②若挂起后线程调度器没有重装载其他线程，通过eCS_Resume()解挂后该线程时此线程将从原断点处继续运行；

③该该函数不可以挂起自己，因为挂起自己有上下文断点问题，挂起自己可能会出现致命错误。有专门挂起自己的语句：

```
-API- eCS_YieldAndSuspend();
-API- eCS_SuspendMeAtState(pNextStateFunction);
```

（5）指定线程由挂起到等待(Ticks≠0)或就绪(Ticks=0)状态。成功返回为真，但系统线程已经达到最大时，解挂会返回失败。

```
-API- eCS_Resume(&TS, Ticks);
```

不能解挂自己，因为自己在运行状态，而不是挂起状态。挂起自己会返回失败。

（6）使指定的线程进入就绪状态

```
-API- eCS_Ready(pTS);
```

如果线程在等待状态，使指定的线程满足时间触发条件，进入就绪状态；

如果线程在挂起或关闭状态，使指定的线程加入调度链表并进入就绪状态；

（7）线程调度器更换线程，且无论新的线程是否以前被调度过都自新线程开始处经Ticks时间后重新启动运行：

```
-API- eCS_Replace(&pTS, pOtherTask, Ticks, Priority);
```

如果事先定义了一个公共的线程调度器，并加入调度链表，那么通过这个功能就可以轮番的装载运行不同的不共生线程。

若线程调度器之前没有加入链表，运行该API相当于新更换的线程处于挂起状态。

可以运用该函数重装载自己的线程调度器，更建议采用自己的线程替换掉有门的函数：

```
-API- eCS_ReplaceMe(pOtherTask, Ticks, Priority);
```

不影响被替换线程的挂起情况。

（8）设置某线程的优先级

设置某线程的优先级为0,1,2,3。pri=0时优先级最高。注意，这里的优先级是“机会把握 ”式近似插队式合作式优先级。

```
-API- eCS_SetPriority(&TS, pri);
```

12 操作当前线程API

操作当前线程的API共7个：

（1）用于时间触发型函数内：延时Ticks个时间节拍后再次被调用，用于时间触发型线

程的后面，设置下次被调用的时间间隔。

```
-API- eCS_SleepMe(Ticks);
```

如：“eCS_SleepMe(100);”表示100个嘀嗒后，线程再次被调度。

应用的一般结构：

```
eCS_State(Task_or_State_Name) //时间触发型线程，时间触发型状态函数
{
    用户变量定义，涉及重入后前后文信息的变量需要加static

    具体的线程语句

    eCS_SleepMe(10); //设定下次调用并执行该线程或状态的周期
}
```

如果该线程就这一个状态，则该线程每10个Tick被调用1次。

（2）重启（重置）当前线程，Ticks滴答后重新自pTask函数开始重新调度，即实现自身线程重启

```
-API- eCS_RestartMe(pTask, Ticks);
```

如：

```
eCS_RestartMe(LED_init(), 10);
```

表示10个嘀嗒后，线程从LED_init()函数开始处重新运行。

不影响重启线程的挂起情况。

（3）关闭当前线程，释放CPU使用权。

```
-API- eCS_CloseMe();
```

关闭线程后，若要再次被调度，需要在其他线程中通过eCS_TaskCreate重新创建该线程。

（3）挂起当前线程。若没有解挂该线程将一直处于挂起状态。

①协程函数中挂起当前线程：

```
-API- eCS_YieldAndSuspend();
```

该函数既可以用在LOOP型线程中，也可以用在线程函数作为FSM的一个状态的线程函数中。

②状态函数（3种线程函数均可）中挂起当前FSM型线程，解挂后进入下一个状态或子状态，因此，运用该函数之前已经完成该状态所有事宜。

```
-API- eCS_SuspendMeAtState(pNextStateFunction);
```

当然，该函数也可以用在线程函数中，此时该线程函数为FSM的一个状态。

要注意，通信线程是不能轻易被挂起的，如I^2C、UART、SPI等。以I^2C为例，项目中要把I^2C通信要专门作为1个线程，其他线程通过信号量机制与I^2C线程配合完成相应的通信需求，不能所有线程都协程操作同一I^2C，否则发生错乱。更不能挂起、重置或被删除I^2C线程。

前述的“eCS_YieldAndSuspend();”、“eCS_SuspendMeAtState(pNextStateFunction);”和“eCS_CloseMe();”三个API对只有一个线程不能挂起和关闭的情况采取不操作的方式。如果有必要，需要使用者自己写相关代码，如

```
例1：
        eCS_YeildUnless(eCS_getRunTaskSum() > 1)
        eCS_YieldAndSuspend();
例2：
        eCS_StateTransferTo(Function_x_suspend)
            :
    }
    state(Function_x_suspend)                    //执着的等待挂起
    {
        if(eCS_getRunTaskSum() <= 1) return;
        eCS_SuspendMeAtState(State_xxxxxxxxx);
    }
例3：
        eCS_StateTransferTo(Function_x_Close)
            :
    }
    state(Function_x_Close)                    //执着的等待关闭
    {
        if(eCS_getTaskSum() <= 1) return;
        eCS_CloseMe();
    }
```

（5）当前“线程调度器”更换线程，且无论新的线程是否过去被调度过都自新线程开始处重新启动运行。

eCS_ReplaceMe (pOtherTask, Ticks, Priority);

如果事先定义了一个公共的线程调度器，那么通过这个功能就可以轮番的装载运行不同的不共生线程。比如，多界面的按键应用，即不同的界面按键的功能不一样，那就建立一个线程调度器，不同的界面装载不同的按键处理线程。

实现HFSM的一种方法就是利用该函数，通过该函数可以重装载线程来改变状态，将装载线程作为对应切换到的次状态（机）。

不影响被替换线程的挂起情况。

（6）设置当前线程的优先级

设置当前线程的优先级为0、1、2和3，设置pri为0时优先级最高。注意，这里的优先级是“机会把握 ”式类似插队式合作式优先级。

```
-API- eCS_SetMePriority(pri);
```

13 当前线程的微秒延时

利用滴答定时器实现了微秒级延时功能。包括设定和判断等待两个部分：

（1）微妙延时设定

微秒延时设定：

```
-API- eCS_WaitusSet(tus);
```

（2）条件阻塞状态函数的微秒延时阻塞判断等待

```
-API- eCS_StateWaitUnless_WaitusOK();
```

eCS_StateWaitUnless_WaitusOK ()与eCS_WaityusSet (tus)要配对使用，且在前后的两个状态中，前一个状态设定时间，后一个状态等待延时结束后进行后续动作。

（3）协程函数内的微秒延时

```
-API- eCS_YieldWaitus(tus);
```

这个API内部隐含了设定函数，不需要在前面进行微秒设定。

（4）占用CPU资源型微秒延时，协程函数和非协程函数中都可以使用，但要慎用。

```
-API- eCS_Waitus(tus);
```

这个API内部隐含了设定函数，不需要在前面进行微秒设定。

主要用于低速运行CPU。对于低速运行CPU，若协程等待延时，线程再次被调度时，可能已经过去过多的微秒延时。

14 获取当前线程总数

1. 获取当前已经创建线程的总数

```
-API- eCS_getCreateTaskSum()
```

只读。

2. 获取当前被调度的线程总数(不包括挂起的线程)

```
-API- eCS_getRunTaskSum()
```

只读。

15 时间触发下的时间表示

1、时间触发下的ms表示

```
-API- eCS_ms(n)
```

基于时基的毫秒表示，注意n要为eCS_SYSTICK_MS的倍数。

2、时间触发下的钟表时间表示

```
-API- eCS_second(s)
-API- eCS_minute(m)
-API- eCS_hour(h)
-API- eCS_date(d)
```

16 获取已经开机多少ms

```
-API- eCS_millis()
```

应用eCS_millis()（只读）可获取已经开机多少ms，返回值为unsigned long型，最长的记录时间为约49.71天，有1个Tick的误差（纪录少了），如果超出时间将从0开始。但调度器没有49天虫问题。

17 获取自最近一次系统复位（热启动）开始已经多少ms

```
-API- eCS_millis_HotReset()
```

应用eCS_millis_HotReset ()（只读）可获取自最近一次系统复位（热启动）开始已经多少ms，返回值也为unsigned long型，最长的记录时间为约49.71天，也有1个Tick的误差（纪录少了），如果超出时间将从0开始。