一个比较靠谱的单片机裸奔通用程序框架

关键词：单片机裸奔、通用程序框架、状态机编程、事件/消息驱动编程

Alicedodo

QQ:379493557

一、对单片机系统提纲性的理解

在讨论程序框架之前，我先用 1 小节的篇幅来谈一谈自己对单片机系统的理解。码这些字绝不是为了凑字数，因为后面要谈到的程序框架是基于这种理解构建起来的，有了这些铺垫，就会发现这种程序框架的合理性，对理解这个程序框架也是有帮助的。

1、单片机系统是一个反应式系统

本节的核心内容：任何单片机系统都可看作是一个反应式系统。

什么是反应式系统？我自己给这个“术语”下了个定义：可接受外界输入，并能根据既定规则和既定状态做出输出响应的系统就是反应式系统。

从这个定义来看，单片机系统是可以称作反应式系统的。系统上电并初始化完毕之后，单片机程序开始进入死循环，无限等待外界输入。一次击键操作、一个数据包、一次 I/O 口电平变化都可以看作是外界对系统的输入，程序捕捉到并识别出输入信号以后，根据系统当前所处的状态和程序员预先设定的逻辑规则做出相应的动作，这些动作即是输出响应。

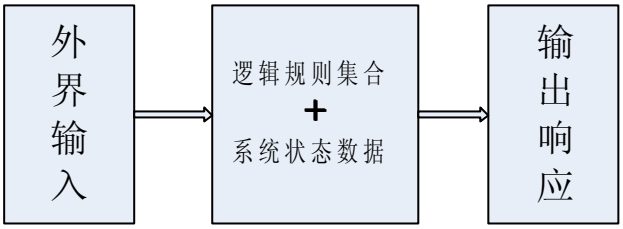

图 1 反应式系统示意图

输出响应可以是点亮一个 LED，也可以是接收发来的数据，甚至什么也不做也可以看作是输出响应。程序执行完输出响应后，对此次外界输入就算仁至义尽了，程序迅速回到等待状态，继续等待新的外界输入，如此周而复始，直至世界末日。

说到这里，具有哲学家特质的童鞋就要抬杠了，且看下面这一段流水灯小程序。

程序清单 List1：

```
void main(void)
{
    sys_init();

    while(1)
    {
        led_on();
        delay_ms(500);
        led_off();
        delay_ms(500);
    }
}
```

这是不是单片机程序 ？！

哪来的外界输入 ？！
这还是不是“反应式系统” ？！

对于这个反例，我想说的是：这个流水灯程序仍然是反应式系统。为了说明这个问题，我们需要把“外界输入”、“系统状态”、“输出响应”这些概念的涵义拓展一下。

2、反应式系统各个要素在单片机系统中的详述

(1) 外界输入

外界输入，更准确的说应该叫“系统输入”。用户的按键输入、串口上的数据请求、管脚上的电平变化等，这些来自单片机系统外部的输入很明显就是系统输入，除此之外，单片机系统内部发生的某些变化也可以看作是系统输入。

板级外设或者片上外设触发的中断请求对单片机程序而言就是系统输入；由程序员设定的、固化在程序代码中的初始数据也叫系统输入；说得再无耻一点，甚至程序在上一次死循环里得出的计算结果，只要能对下一次程序循环造成影响，我也把这些计算结果看作是系统输入！

(2) 系统状态

想必大家对数显式的电子表都很熟悉，这种表一般只有 MODE 和 SET 两个按键。按 MODE 键在年月日时分秒之间切换，选中的作闪烁状，按 SET 键修改数值，或加或减。现在只以 MODE 键为例，同样的按键操作，同样的程序，为什么每次按键表盘的输出内容都不一样呢？这就是系统状态所起到的作用了。

按一次 MODE 键，程序会根据当前的工作状态选择不同的输出响应，然后修改系统的工作状态，下一次处理按键操作时，程序就会根据新的工作状态选择新的输出响应了，当然后面同样要修改系统工作状态。所以说，单片机系统的输出响应是逻辑规则集合与系统状态协作的结果。

在程序中，系统状态有两种记录方式。

第一种方法是在内存中开辟一块内存单元，用这块内存所能存储的数值集合的子集或全集来表示所有可能的系统状态，每个数值代表一种状态。例如用一个字节的内存来存储状态，那么该系统最多有 256 个状态可用，0、1、……、255 中的任意数值都可代表一种状态。程序引用系统状态时只需要读取该内存中的数据即可，这是单片机程序记录系统状态最普遍最直观的方法，后面谈到的“状态机”就是采用的这种方式。

第二种方法是利用程序的执行位置来记录系统状态。看一下下面的一小段伪代码。

程序清单 List2：

```
……
wait_usr_click();
do_plan_first();
wait_usr_click();
do_plan_second();
wait_usr_click();
```

```
do_plan_third();
    ……
```

从上面的代码可以看到，3 次相同的用户输入，会得到 3 种不同的输出响应。显然这里面包含着系统状态，但是这里没有使用内存单元记录系统状态，仔细观察代码就会发现，其实每一个 wait_usr_click()函数所处的位置就代表着一个状态，不用判断，也不用读取内存，程序只管往下走就是了，因为所有的状态都已经分散地固化到代码里面了。

再回头看一看上面的那个流水灯程序(代码清单 List1), 和这个是不是有些相像呢？

(3) 输出响应

和系统输入类似，对输出响应也应该从一个更广更深的角度来理解。点亮 LED、奏响蜂鸣器、从串口向外发送数据，这些看得见摸得着的程序动作显然是正儿八经的输出响应，除此之外，那些隐藏在程序内部、外界难以察觉的动作也应视为单片机系统的输出响应。

设想这样的一个小例子：单片机接受用户按键，连击 10 次后让蜂鸣器响一声，如此周而复始。前 9 次按键的时候，从外面看程序好像什么也没干，但是我们可以推断出程序肯定在每次按键之后在内存中记录了按键的次数，我们可以把这种记录按键次数的操作看作是系统的输出响应。

隐蔽的输出响应主要就是程序对内存的操作，在系统输入的作用下，程序修改内存中的变量的值，这些变量可能是系统状态变量，也可能是某个功能模块的数据结构的一部分。

另外，还有一种最特殊的输出响应，那就是没响应！所谓空即是色，色即是空。

需要注意的是，系统的一次输出响应可以只有显示的输出，也可以只有隐式的输出，还可以是两种输出的组合，如上面的小例子，第 10 次按键之后，程序即要让蜂鸣器响一声，又要清零按键计数变量，为下一次循环做准备。

好了，基本概念到此已经解释完了，回过头来看一看那个流水灯程序(代码清单 List1)，反应式系统里的那些要素这个小程序也是有的：

逻辑规则：亮则灭之，灭则亮之
系统状态：参考系统状态的第二种表示方法
输出响应：亮灯，灭灯
系统输入：既然程序根据当前 LED 的状态选择动作，为什么不能把状态当做输入呢？

至于你们信不信，反正我是信的！

3、小结

单片机程序本质上讲就是一个状态机，它包含了一个数量有限的工作状态集合，在系统输入的驱动下有规律地在各个工作状态之间切换，这种切换也常常伴随着一些额外的动作，从功能的角度来看，这些动作就是输出响应。

把单片机程序看作是状态机，就可以用状态机的思想来编程。

把单片机程序看作是状态机，就会发现单片机系统实际上是一个被动系统，只有在系统输入(广义上的)的驱动下才会活动(状态转换、输出响应)。

系统有输入，说明发生了会对系统工作状态有影响的事件，程序要想处理系统输入，就要识别这些事件。在一个功能复杂，接口众多的单片机系统中，发生的事件可能是多种多样的，模拟输入、数字输入、各种中断、各种通信请求。为了降低编程的复杂度，更好的管理这些事件，应该对事件进行加工，把它们变成标准统一的数据格式提交程序(状态机化了的程序)处理。

把事件变成消息，用消息驱动单片机程序就是这种思想。

二、关于状态机编程

说起状态机编程，网上的资料一大把，但是考虑到这份资料的完整性，也为了免除大家再到处搜集文章的痛苦，我还是想系统性地陈述一下。

1、状态机概念及编程示例

什么是状态机？

状态机是一个这样的东东：状态机(state machine)有 5 个要素，分别是状态(state)、迁移(transition)、事件(event)、动作(action)、条件(guard)。

状态： 一个系统在某一时刻所存在的稳定的工作情况，系统在整个工作周期中可能有多个状态。例如一部电动机共有正转、反转、停转这 3 种状态。

一个状态机需要在状态集合中选取一个状态作为初始状态。

迁移：系统从一个状态转移到另一个状态的过程称作迁移，迁移不是自动发生的，需要外界对系统施加影响。停转的电动机自己不会转起来，让它转起来必须上电。

事件：某一时刻发生的对系统有意义的事情，状态机之所以发生状态迁移，就是因为出现了事件。对电动机来讲，加正电压、加负电压、断电就是事件。

动作：在状态机的迁移过程中，状态机会做出一些其它的行为，这些行为就是动作，动作是状态机对事件的响应。给停转的电动机加正电压，电动机由停转状态迁移到正转状态，同时会启动电机，这个启动过程可以看做是动作，也就是对上电事件的响应。

条件：状态机对事件并不是有求必应的，有了事件，状态机还要满足一定的条件才能发生状态迁移。还是以停转状态的电动机为例，虽然合闸上电了，但是如果供电线路有问题的话，电动机还是不能转起来。

只谈概念太空洞了，上一个小例子：一单片机、一按键、俩 LED 灯(记为 L1 和 L2)、一人，足矣！

规则描述：

1) L1L2 状态转换顺序 OFF/OFF--->ON/OFF--->ON/ON--->OFF/ON--->OFF/OFF
2) 通过按键控制 L1L2 的状态,每次状态转换需连续按键 5 次
3) L1L2 的初始状态 OFF/OFF

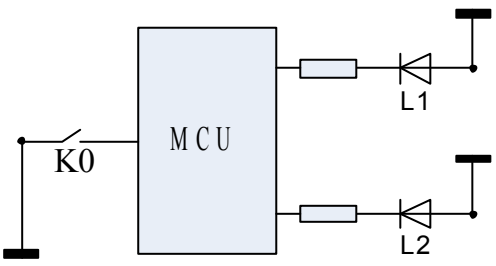

图 2 按键控制流水灯电路简图

下面这段程序是根据功能要求写成的代码。

程序清单 List3：

```
#define LS_OFFOFF     1
#define LS_ONOFF      2
#define LS_ONON       4
#define LS_OFFON      8

struct fsm_led
{
    INT8U u8LedStat;              /*状态机变量*/
    INT8U u8KeyCnt;               /*击键计数器*/

} g_stFSM ;        /*状态机结构体*/

void fsm_active(void);

/*****************************************
*FuncName    : main
*Description :   主函数
*Arguments   : void
*Return      : void
*History     : [2011-08-13]
*                   创建；
*****************************************/
void main(void)
{
    sys_init();

    led_off(LED1);              /*状态机初始化*/
    led_off(LED2);
    g_stFSM.u8LedStat = LS_OFFOFF;
    g_stFSM.u8KeyCnt = 0;

    while(1)
    {
        if(test_key()==TRUE)
        {
            fsm_active();
        }
        else
        {
            ; /*idle code*/
        }
    }
}
```

```
/**************************************
*FuncName    : fsm_active
*Description: 状态机管理函数
*Arguments    : void
*Return        : void
*History       : [2011-08-13]
*                    创建；
**************************************/
void fsm_active(void)
{
    if(g_stFSM.u8KeyCnt > 3)        /*击键是否满 5 次*/
    {
        switch(g_stFSM.u8LedStat)
        {
            case LS_OFFOFF:
                led_on(LED1);                          /*输出动作*/
                g_stFSM.u8KeyCnt = 0;
                g_stFSM.u8LedStat = LS_ONOFF; /*状态迁移*/
                break;

            case LS_ONOFF:
                led_on(LED2);                          /*输出动作*/
                g_stFSM.u8KeyCnt = 0;
                g_stFSM.u8LedStat = LS_ONON; /*状态迁移*/
                break;

            case LS_ONON:
                led_off(LED1);                         /*输出动作*/
                g_stFSM.u8KeyCnt = 0;
                g_stFSM.u8LedStat = LS_OFFON; /*状态迁移*/
                break;

            case LS_OFFON:
                led_off(LED2);                         /*输出动作*/
                g_stFSM.u8KeyCnt = 0;
                g_stFSM.u8LedStat = LS_OFFOFF; /*状态迁移*/
                break;

            default:                                   /*非法状态*/
                led_off(LED1);
                led_off(LED2);
                g_stFSM.u8KeyCnt = 0;
                g_stFSM.u8LedStat = LS_OFFOFF; /*恢复初始状态*/
                break;
        }
    }
```

```
    else
    {
        g_stFSM.u8KeyCnt++;                    /*状态不迁移，仅记录击键次数*/
    }
}
```

图 3 所示为程序清单 List3 对应的状态转换图。

实际上在状态机编程中，正确的顺序应该是先有状态转换图，后有程序，程序应该是根据设计好的状态图写出来的。不过考虑到有些童鞋会觉得代码要比转换图来得亲切，我就先把程序放在前头了。

这张状态转换图是用 UML(统一建模语言)的语法元素画出来的，语法不是很标准，但拿来解释问题足够了。

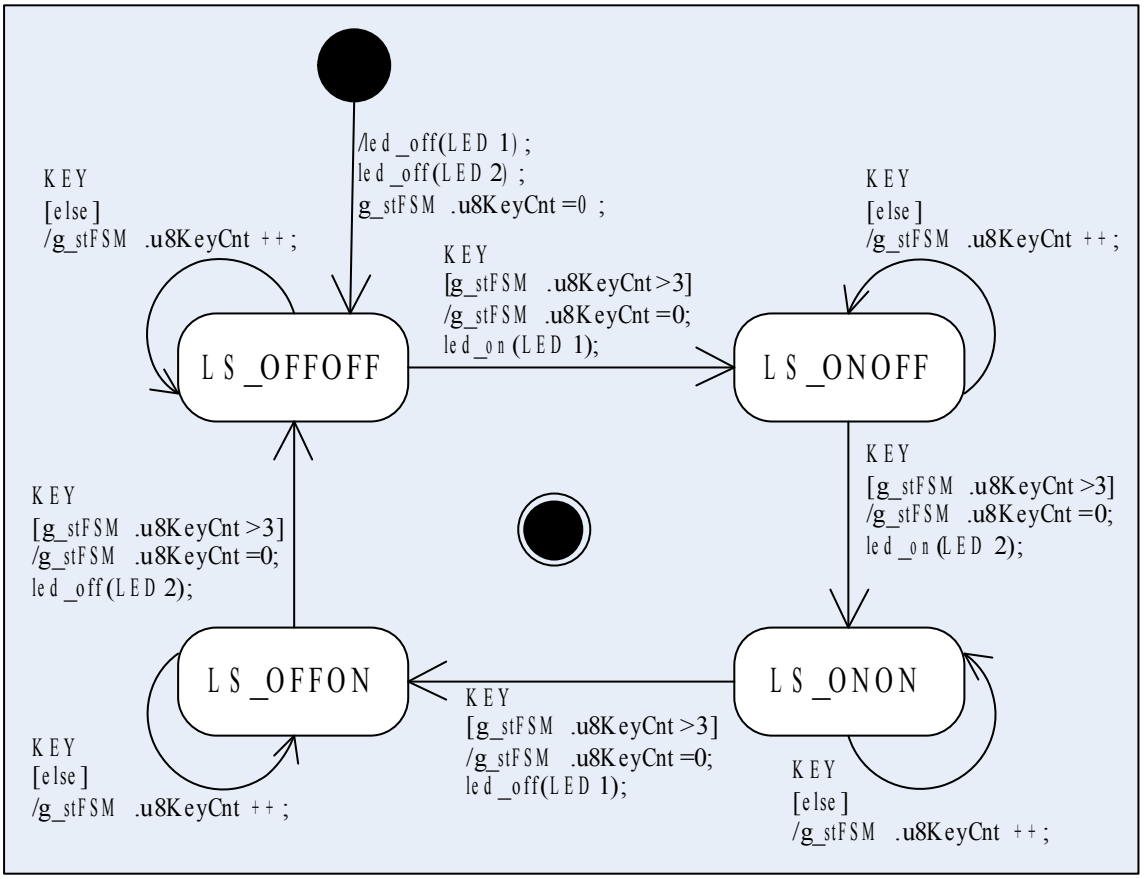

图 3 按键控制流水灯状态转换图

圆角矩形代表状态机的各个状态，里面标注着状态的名称。

带箭头的直线或弧线代表状态迁移，起于初态，止于次态。

图中的文字内容是对迁移的说明，格式是：事件[条件]/动作列表(后两项可选)。

“事件[条件]/动作列表”要说明的意思是：如果在某个状态下发生了“事件”，并且状态机

满足“[条件]”，那么就要执行此次状态转移，同时要产生一系列“动作”，以响应事件。在这个例子里，我用“KEY”表示击键事件。

图中有一个黑色实心圆点，表示状态机在工作之前所处的一种不可知的状态，在运行之前状态机必须强制地由这个状态迁移到初始状态，这个迁移可以有动作列表(如图 3 所示)，但不需要事件触发。

图中还有一个包含黑色实心圆点的圆圈，表示状态机生命周期的结束，这个例子中的状态机生生不息，所以没有状态指向该圆圈。

关于这个状态转换图就不多说了，相信大家结合着上面的代码能很容易看明白。

现在我们再聊一聊程序清单 List3。

先看一下 fsm_active()这个函数，g_stFSM.u8KeyCnt = 0;这个语句在 switch—case 里共出现了 5 次，前 4 次是作为各个状态迁移的动作出现的。从代码简化提高效率的角度来看，我们完全可以把这 5 次合并为 1 次放在 switch—case 语句之前，两者的效果是完全一样的，代码里之所以这样啰嗦，是为了清晰地表明每次状态迁移中所有的动作细节，这种方式和图 3 的状态转换图所要表达的意图是完全一致的。

再看一下 g_stFSM 这个状态机结构体变量，它有两个成员：u8LedStat 和 u8KeyCnt。用这个结构体来做状态机好像有点儿啰嗦，我们能不能只用一个像 u8LedStat 这样的整型变量来做状态机呢？

当然可以！我们把图 3 中的这 4 个状态各自拆分成 5 个小状态，这样用 20 个状态同样能实现这个状态机，而且只需要一个 unsigned char 型的变量就足够了，每次击键都会引发状态迁移，每迁移 5 次就能改变一次 LED 灯的状态，从外面看两种方法的效果完全一样。

为什么没有这样做呢？

假设我把功能要求改一下，把连续击键 5 次改变 L1L2 的状态改为连续击键 100 次才能改变 L1L2 的状态。这样的话第二种方法需要 4X100=400 个状态！而且函数 fsm_active()中的 switch—case 语句里要有 400 个 case，这样的程序还有法儿写么？！

同样的功能改动，如果用 g_stFSM 这个结构体来实现状态机的话，函数 fsm_active()只需要将 if(g_stFSM.u8KeyCnt > 3)改为 if(g_stFSM.u8KeyCnt > 98)就可以了！

g_stFSM 结构体的两个成员中，u8LedStat 可以看作是质变因子，相当于主变量；u8KeyCnt 可以看作是量变因子，相当于辅助变量。量变因子的逐步积累会引发质变因子的变化。

像 g_stFSM 这样的状态机被称作 Extended State Machine，我不知道业内正规的中文术语怎么讲，只好把英文词组搬过来了。

2、状态机编程的优点

说了这么多，大家大概明白状态机到底是个什么东西了，也知道状态机化的程序大体怎么

写了，那么单片机的程序用状态机的方法来写有什么好处呢？

(1) 提高 CPU 使用效率

话说我只要见到满篇都是 delay_ms()的程序就会蛋疼，动辄十几个 ms 几十个 ms 的软件延时是对 CPU 资源的巨大浪费，宝贵的 CPU 机时都浪费在了 NOP 指令上。那种为了等待一个管脚电平跳变或者一个串口数据而岿然不动的程序也让我非常纠结，如果事件一直不发生，你要等到世界末日么？

把程序状态机化，这种情况就会明显改观，程序只需要用全局变量记录下工作状态，就可以转头去干别的工作了，当然忙完那些活儿之后要再看看工作状态有没有变化。只要目标事件(定时未到、电平没跳变、串口数据没收完)还没发生，工作状态就不会改变，程序就一直重复着“查询—干别的—查询—干别的”这样的循环，这样 CPU 就闲不下来了。在程序清单 List3 中，if{}else{}语句里 else 下的内容(代码中没有添加，只是加了一条/*idle code*/的注释示意)就是上文所说的“别的工作” 。

这种处理方法的实质就是在程序等待事件的过程中间隔性地插入一些有意义的工作，好让 CPU 不是一直无谓地等待。

(2) 逻辑完备性

我觉得逻辑完备性是状态机编程最大的优点。

不知道大家有没有用 C 语言写过计算器的小程序，我很早以前写过，写出来一测试，那个惨不忍睹啊！当我规规矩矩的输入算式的时候，程序可以得到正确的计算结果，但要是故意输入数字和运算符号的随意组合，程序总是得出莫名其妙的结果。

后来我试着思维模拟一下程序的工作过程，正确的算式思路清晰，流程顺畅，可要碰上了不规矩的式子，走着走着我就晕菜了，那么多的标志位，那么多的变量，变来变去，最后直接分析不下去了。

很久之后我认识了状态机，才恍然明白，当时的程序是有逻辑漏洞的。如果把这个计算器程序当做是一个反应式系统，那么一个数字或者运算符就可以看做一个事件，一个算式就是一组事件组合。对于一个逻辑完备的反应式系统，不管什么样的事件组合，系统都能正确处理事件，而且系统自身的工作状态也一直处在可知可控的状态中。反过来，如果一个系统的逻辑功能不完备，在某些特定事件组合的驱动下，系统就会进入一个不可知不可控的状态，与设计者的意图相悖。

状态机就能解决逻辑完备性的问题。

状态机是一种以系统状态为中心，以事件为变量的设计方法，它专注于各个状态的特点以及状态之间相互转换的关系。状态的转换恰恰是事件引起的，那么在研究某个具体状态的时候，我们自然而然地会考虑任何一个事件对这个状态有什么样的影响。这样，每一个状态中发生的每一个事件都会在我们的考虑之中，也就不会留下逻辑漏洞。

这样说也许大家会觉得太空洞，实践出真知，某天如果你真的要设计一个逻辑复杂的程序，

我保证你会说：哇！状态机真的很好用哎！

(3) 程序结构清晰

用状态机写出来的程序的结构是非常清晰的。

程序员最痛苦的事儿莫过于读别人写的代码。如果代码不是很规范，而且手里还没有流程图，读代码会让人晕了又晕，只有顺着程序一遍又一遍的看，很多遍之后才能隐约地明白程序大体的工作过程。有流程图会好一点，但是如果程序比较大，流程图也不会画得多详细，很多细节上的过程还是要从代码中理解。

相比之下，用状态机写的程序要好很多，拿一张标准的 UML 状态转换图，再配上一些简明的文字说明，程序中的各个要素一览无余。程序中有哪些状态，会发生哪些事件，状态机如何响应，响应之后跳转到哪个状态，这些都十分明朗，甚至许多动作细节都能从状态转换图中找到。可以毫不夸张的说，有了 UML 状态转换图，程序流程图写都不用写。

套用一句广告词：谁用谁知道！

3、用 C 语言实现状态机

接下来再说一说如何用 C 语言来实现状态机。

状态机的实现无非就是 3 个要素：状态、事件、响应。转换成具体的行为就 3 句话。

发生了什么事？
现在系统处在什么状态？
在这样的状态下发生了这样的事，系统要干什么？

用 C 语言实现状态机主要有 3 种方法：switch—case 法、表格驱动法、函数指针法。

(1) switch—case 法

将状态用 switch—case 组织起来，将事件也用 switch—case 组织起来，然后让其中一个 switch—case 整体插入到另一个 switch—case 的每一个 case 项中。

程序清单 List4：

```
switch(StateVal)
{
    case S0:
        switch(EvntID)
        {
            case E1:
                action_S0_E1();              /*S0 状态下 E1 事件的响应*/
                StateVal = new state value;/*状态迁移，不迁移则没有此行*/
                break;
            case E2:
```

```
                action_S0_E2();              /*S0 状态下 E2 事件的响应*/
                StateVal = new state value;
                break;
                ......
            case Em:
                action_S0_Em();              /*S0 状态下 Em 事件的响应*/
                StateVal = new state value;
                break;
            default:
                break;
        }
        break;
    case S1:
        ......
        break;
        ......
    case Sn:
        ......
        break;
    default:
        break;
}
```

上面的伪代码示例只是通用的情况，实际应用远没有这么复杂。虽然一个系统中事件可能有很多种，但在实际应用中，许多事件可能对某个状态是没有意义的。例如在程序清单 List4 中，如果 E2、•••••• Em 对处在 S0 状态下的系统没有意义，那么在 S0 的 case 下有关事件 E2、•••••• Em 的代码根本没有必要写,状态 S0 只需要考虑事件 E1 的处理就行了。

既然是两个 switch—case 之间的嵌套，那么就有一个谁嵌套谁的问题，所以说 switch—case 法有两种写法：状态嵌套事件和事件嵌套状态。这两种写法都可以，各有利弊，至于到底选用哪种方式就留给设计人员根据具体情况自行决断吧。

关于 switch—case 法还有最后一点要说明，因为 switch—case 的原理是从上到下挨个比较，越靠后，查找耗费的时间就越长，所以要注意状态和事件在各自的 switch 语句中的安排顺序，不推荐程序清单 List4 那样按顺序号排布的方式。出现频率高或者实时性要求高的状态和事件的位置应该尽量靠前。

(2) 表格驱动法

如果说 switch—case 法是线性的，那么表格驱动法则是平面的。

表格驱动法的实质就是将状态和事件之间的关系固化到一张二维表格里，把事件当做纵轴，把状态当做横轴，交点[Sn , Em]则是系统在 Sn 状态下对事件 Em 的响应。

E2	Node_S0E2	Node_S1E2	Node_S2E2
E1	Node_S0E1	Node_S1E1	Node_S2E1
E0	Node_S0E0	Node_S1E0	Node_S2E0
	S0	**S1**	**S2**

图 4 表格驱动法示例图

如图 4，我把表格中的 Node_SnEm 叫做状态机节点，状态机节点 Node_SnEm 是系统在 Sn 状态下对事件 Em 的响应。这里所说的响应包含两个方面：输出动作和状态迁移。

状态机节点一般是一个类似程序清单 List5 中的结构体变量。

程序清单 List5：

```
struct fsm_node
{
    void (*fpAction)(void* pEvnt);
    INT8U u8NxtStat;
};
```

程序清单 List5 中的这个结构体有两个成员：fpAction 和 u8NxtStat。

fpAction 是一个函数指针，指向一个形式为 void func(void* pEvnt)的函数，func 这个函数是对状态转移中动作序列的标准化封装。也就是说，状态机在状态迁移的时候，不管输出多少个动作、操作多少个变量、调用多少个函数，这些行为统统放到函数 func 中去做。

把动作封装好了之后，再把封装函数 func 的地址交给函数指针 fpAction，这样，想要输出动作，只需要调用函数指针 fpAction 就行了。

再看看上面的 func 函数，会发现函数有一个形参 pEvnt，这是一个类型为 void*的指针，在程序实际运行时指向一个能存储事件的变量，通过这个指针我们就能获知关于事件的全部信息，这个形参是很有必要的。

事件一般包括两个属性：事件的类型和事件的内容。例如一次按键事件，我们不仅要知道这是一个按键事件，还要知道按下的到底是哪个键。

事件的类型和状态机当前的状态可以让我们在图 4 的表格中迅速定位，确定该调用哪个动作封装函数，但是动作封装函数要正确响应事件还需要知道事件的内容是什么，这也就是形参 pEvnt 的意义。由于事件的多样性，存储事件内容的数据格式不一定一样，所以就把 pEvnt 定义成了 void*型，以增加灵活性。

有关 fpAction 的最后一个问题：如果事件 Em 对状态 Sn 没有意义，那么状态机节点 Node_SnEm 中的 fpAction 该怎么办？我的答案是：那就让它指向一个空函数呗！前面不是说过么，什么也不干也叫响应。

u8NxtStat 存储的是状态机的一个状态值。我们知道，状态机响应事件要输出动作，也就是调用函数指针 fpAction 所指向的那个封装函数，函数调用完毕后程序返回主调函数，状态机对事件的响应就算结束了，下一步就要考虑状态迁移的问题了。可能要保持本状态不变，也可能要迁移到一个新的状态，该如何抉择呢？u8NxtStat 存储的状态就是状态机想要的答案！

图 4 的这张表格反映在 C 语言代码里就是一个二维数组，第 1 维就是状态机的状态，第 2 维就是统一分类的事件，而数组的元素则是程序清单 List5 中的结构体常量。

如果程序中使用表格驱动法，还需要注意一些特别的事项。

要将状态当做表格的横轴，那么就要求状态值集合必须满足以下条件：

(1) 该集合是一个递增的等差整数数列
(2) 该数列初值为 0
(3) 该数列等差值为 1

“事件”作为纵轴，其特点和要求与用来做横轴的“状态”完全一致。

在 C 语言提供的数据类型中，没有比枚举更符合以上要求的可选项了，极力推荐将状态集合和事件类型集合做成枚举常量。

表格驱动法的优点：调用接口统一 ，定位快速。

表格驱动法屏蔽了不同状态下处理各个事件的差异性，因此可以将处理过程中的共性部分提炼出来，做成标准统一的框架式代码，形成统一的调用接口。根据程序清单 List5 中的状态机节点结构体，做成的框架代码如程序清单 List6 所示。

表格驱动法查找目标实际上就是一次二维数组的寻址操作，所以它的平均效率要远高于 switch—case 法。

程序清单 List6：

```
extern struct fsm_node g_arFsmDrvTbl[][];      /*状态机驱动表格*/

INT8U u8CurStat = 0;                           /*状态暂存*/
INT8U u8EvntTyp = 0;                           /*事件类型暂存*/
void* pEvnt = NULL;                            /*事件变量地址暂存*/
struct fsm_node stNodeTmp = {NULL, 0};         /*状态机节点暂存*/

u8CurStat = get_cur_state();                   /*读取当前状态*/
u8EvntTyp = get_cur_evnt_typ();                /*读取当前触发事件类型*/
pEvnt = (void*)get_cur_evnt_ptr();             /*读取事件变量地址*/
stNodeTmp =   g_arFsmDrvTbl[u8CurStat ][u8EvntTyp ];/*定位状态机节点*/

stNodeTmp.fpAction(pEvnt );                    /*动作响应*/
set_cur_state(stNodeTmp.u8NxtStat);            /*状态迁移*/
.....
```

表格驱动法好则好矣，但用它写出来的程序还有点儿小问题，我们先来看看按照表格驱动

法写出来的程序有什么特点。

前面说过，表格驱动法可以把状态机调度的部分做成标准统一的框架代码，这个框架适用性极强，不管用状态机来实现什么样的应用，框架代码都不需要做改动，我们只需要根据实际应用场合规划好状态转换图，然后将图中的各个要素(状态、事件、动作、迁移，有关“条件”要素一会儿再说)用代码实现就行了，我把这部分代码称作应用代码。

在应用代码的.c 文件中，你会看到一个声明为 const 的二维数组，也就是图 4 所示的状态驱动表格，还会看到许多彼此之间毫无关联的函数，也就是前面提到的动作封装函数。这样的一份代码，如果手头上没有一张状态转换图，让谁看了也会一头雾水，这样的格式直接带来了代码可读性差的问题。

如果我们想给状态机再添加一个状态，反映到代码上就是给驱动表格再加一列内容，同时也要新添加若干个动作封装函数。如果驱动表格很大，做这些工作是很费事儿的，而且容易出错。如果不小心在数组中填错了位置，那么程序跑起来就和设计者的意图南辕北辙了，远没有在 switch—case 法中改动来得方便、安全。

上面说的只是小瑕疵，其实最让我不爽的是表格驱动法不能使用程序清单 List3 中 g_stFSM 那样的 Extended State Machine(对这个词组还有印象吧？)！

Extended State Machine 的最大特点就是状态机响应事件之前先判断条件，根据判定结果选择执行哪些动作，转向哪个状态。也就是说，系统在状态 Sn 下发生了事件 Em 后，转向的状态不一定是唯一的，这种灵活性是 Extended State Machine 的最有价值的优点。

回过头来看看程序清单 List5 中给出的状态机节点结构体，如果系统在状态 Sn 下发生了事件 Em，状态机执行完 fpAction 所给出的动作响应之后，必须转到 u8NxtStat 指定的状态。表格驱动法的这个特性直接杜绝了 Extended State Machine 在表格驱动法中应用的可能性，所以表格驱动法的代码实现中不存在“条件”这个状态机要素。

ESM，你是如此的优秀，我怎么舍得抛弃你 ？！

再看图 4 所示的表格驱动法示例图，如果我们把表格中的代表事件的纵轴去掉，只留下代表状态的横轴，将一列合并成一格，前文提到的问题是不是能得到解决呢？

不错！这就是失传江湖多年的《葵花宝典》——阉割版表格驱动法 !!

阉割版表格驱动法，又名压缩表格驱动法，一维状态表格与事件 switch—case 的合体。

压缩表格驱动法使用了一维数组作为驱动表格，数组的下标即是状态机的各个状态。表格中的元素叫做压缩状态机节点，节点的主要内容还是一个指向动作封装函数的函数指针，只不过这个动作封装函数不是为某个特定事件准备的，而是对所有的事件都有效的。节点中不再强制指定状态机输出动作完毕后所转向的状态，而是让动作封装函数返回一个状态，并把这个状态作为状态机新的状态。

压缩表格驱动法的这个特点，完美的解决了 Extended State Machine 不能在表格驱动法中使用的问题。

程序清单 List7 中的示例代码包含了压缩状态机节点结构体和状态机调用的框架代码。

程序清单 List7：

```
struct fsm_node                          /*压缩状态机节点结构体*/
{
    INT8U (*fpAction)(void* pEvnt);      /*事件处理函数指针*/
    INT8U u8StatChk;                     /*状态校验*/
};

    ......

u8CurStat = get_cur_state();                    /*读取当前状态*/

    ......

if(stNodeTmp.u8StatChk == u8CurStat )
{
    u8CurStat = stNodeTmp.fpAction(pEvnt );    /*事件处理*/
    set_cur_state(u8CurStat );                  /*状态迁移*/
}
else
{
    state_crash(u8CurStat );                    /*非法状态处理*/
}
    ......
```

对照程序清单 List5，就会发现程序清单 List7 中 struct fsm_node 结构体的改动之处。

首先，fpAction 所指向函数的函数形式变了，动作封装函数 func 的模样成了这样的了：

INT8U func(void * pEvnt)

现在的动作封装函数 func 是要返回类型为 INT8U 的返回值的，这个返回值就是状态机要转向的状态，也就是说，压缩表格驱动法中的状态机节点不负责状态机新状态的确定，而把这项任务交给了动作封装函数 func，func 返回哪个状态，状态机就转向哪个状态。新状态由原来的常量变成了现在的变量，自然要灵活许多。

上面说到现在的动作封装函数 func 要对当前发生的所有的事件都要负责，那么 func 怎么会知道到底是哪个事件触发了它呢？

看一下 func 的形参 void * pEvnt 。在程序清单 List5 中我们提到过，这个形参是用来向动作封装函数传递事件内容的，但是从前文的叙述中我们知道，pEvnt 所指向的内存包含了事件的所有信息，包括事件类型和事件内容 ，所以通过形参 pEvnt ，动作封装函数 func 照样可以知道事件的类型。

程序清单 List7 中 struct fsm_node 结构体还有一个成员 u8StatChk ，这里面存储的是状态机

的一个状态，干什么用的呢？

玩 C 语言数组的人都知道，要严防数组寻址越界，内存泄露可不是闹着玩儿的。要知道，压缩表格驱动法的驱动表格是一个以状态值为下标的一维数组，数组元素里面最重要的部分就是一个个动作封装函数的地址。函数地址在单片机看来无非就是一段二进制数据，和内存中其它的二进制数据没什么两样，不管程序往单片机 PC 寄存器里塞什么值，单片机都没意见。

假设程序由于某种意外而改动了存储状态机当前状态的变量，使变量值变成了一个非法状态。再发生事件时，程序就会用这个非法的状态值在驱动表格中寻址，这时候就会发生内存泄露，程序拿泄露内存中的未知数据当函数地址跳转，不跑飞才怪！

为了防止这种现象的发生，压缩状态机节点结构体中又添加了成员 u8StatChk 。u8StatChk 中存储的是压缩状态机节点在一维驱动表格的位置，例如某节点是表格中的第 7 个元素，那么这个节点的成员 u8StatChk 值就是 6。

看一下程序清单 List7 中的框架代码示例，程序在引用函数指针 fpAction 之前，先检查当前状态和当前节点成员 u8CurStat 的值是否一致，一致则认为状态合法，事件正常响应，如果不一致，则认为当前状态非法，转至意外处理，最大限度保证程序运行的安全。

当然，如果泄露内存中的数据恰好和 u8CurStat 一致，那么这种方法真的就回天乏力了。

还有一个方法也可以防止状态机跑飞，如果状态变量是枚举，那么框架代码就可以获知状态值的最大值，在调用动作封装函数之前判断一下当前状态值是否在合法的范围之内，同样能保证状态机的安全运行。

压缩表格驱动法中动作封装函数的定义形式我们已经知道了，函数里面到底是什么样子的呢？程序清单 List8 是一个标准的示例。

程序清单 List8：

```
INT8U action_S0(void* pEvnt)
{
    INT8U u8NxtStat = 0;
    INT8U u8EvntTyp = get_evnt_typ(pEvnt);

    switch(u8EvntTyp )
    {
        case E1:
            action_S0_E1();                 /*事件 E1 的动作响应*/
            u8NxtStat = new state value; /*状态迁移，不迁移也必须有本行*/
            break;
            .....
        case Em:
            action_S0_Em();                 /*事件 Em 的动作响应*/
            u8NxtStat = new state value; /*状态迁移，不迁移也必须有本行*/
            break;
```

```
            default:
                ; /*不相关事件处理*/
                break;
        }

        return u8NxtStat ;              /*返回新状态*/
    }
```

从程序清单 List8 可以看出，动作封装函数其实就是事件 switch—case 的具体实现。函数根据形参 pEvnt 获知事件类型，并根据事件类型选择动作响应，确定状态机迁移状态，最后将新的状态作为执行结果返回给框架代码。

有了这样的动作封装函数，Extended State Machine 的应用就可以完全不受限制了！

到此，有关压缩表格驱动法的介绍就结束了。个人认为压缩表格驱动法是相当优秀的，它既有表格驱动法的简洁、高效、标准，又有 switch—case 法的直白、灵活、多变，相互取长补短，相得益彰。

姜文：师爷，夫妻之间最重要的是什么？
葛优：恩爱！

(3) 函数指针法

上面说过，用 C 语言实现状态机主要有 3 种方法(switch—case 法、表格驱动法、函数指针法)，其中函数指针法是最难理解的，它的实质就是把动作封装函数的函数地址作为状态来看待。

不过，有了之前压缩表格驱动法的铺垫，函数指针法就变得好理解了，因为两者本质上是相同的。

压缩表格驱动法的实质就是一个整数值(状态机的一个状态)到一个函数地址(动作封装函数)的一对一映射，压缩表格驱动法的驱动表格就是全部映射关系的直接载体。在驱动表格中通过状态值就能找到函数地址，通过函数地址同样能反向找到状态值。

我们用一个全局的整型变量来记录状态值，然后再查驱动表格找函数地址，那干脆直接用一个全局的函数指针来记录状态得了，还费那劳什子劲干吗？！

这就是函数指针法的前世今生。

用函数指针法写出来的动作封装函数和程序清单 List8 的示例函数是很相近的，只不过函数的返回值不再是整型的状态值，而是下一个动作封装函数的函数地址，函数返回后，框架代码再把这个函数地址存储到全局函数指针变量中。

相比压缩表格驱动法，在函数指针法中状态机的安全运行是个大问题，我们很难找出一种机制来检查全局函数指针变量中的函数地址是不是合法值。如果放任不管，一旦函数指针变量中的数据被篡改，程序跑飞几乎就不可避免了。

4、小结

有关状态机的东西说了那么多，相信大家都已经感受到了这种工具的优越性，状态机真的是太好用了！其实我们至始至终讲的都是有限状态机(Finite State Machine 现在知道为什么前面的代码中老是有 fsm 这个缩写了吧！)，还有一种比有限状态机更 NB 更复杂的状态机，那就是层次状态机(Hierarchical State Machine 一般简写为 HSM)。

通俗的说，系统中只存在一个状态机的叫做有限状态机，同时存在多个状态机的叫做层次状态机(其实这样解释层次状态机有些不严谨，并行状态机也有多个状态机，但层次状态机各个状态机之间是上下级关系，而并行状态机各个状态机之间是平级关系)。

层次状态机是一种父状态机包含子状态机的多状态机结构，里面包含了许多与面向对象相似的思想，所以它的功能也要比有限状态机更加强大，当一个问题用有限状态机解决起来有些吃力的时候，就需要层次状态机出马了。层次状态机理论我理解得也不透彻，就不在这里班门弄斧了，大家可以找一些有关状态机理论的专业书籍来读一读。

要掌握状态机编程，理解状态机(主要指有限状态机)只是第一步，也是最简单的一步，更重要的技能是能用状态机这个工具去分析解剖实际问题：划分状态、提取事件、确定转换关系、规定动作等等，形成一张完整的状态转换图，最后还要对转换图进行优化，达到最佳。

把实际问题变成了状态转换图，工作的一大半就算完成了，这个是具有架构师气质的任务，剩下的问题就是按照状态图编程写代码了，这个是具有代码工特色的工作。

三、关于事件/消息驱动编程

状态机是一种思想，事件驱动也是一种思想。

1、事件驱动的概念

什么是事件驱动思想？

生活中有很多事件驱动的例子，上自习瞒着老师偷睡觉就是很生动的一个。

我们都是从高中时代走过来的，高中的学生苦啊，为了考高分整天玩儿命地学习，起得比鸡早，睡得比狗晚，白天要上课，晚上还要上自习，有木有啊！那时候累啊，每天睡眠严重不足，觉得睡觉是世界上最奢侈的东西，有时候站着都能睡着啊！老师看的严，上课睡觉不允许啊，要挨批啊！有木有！

相比而言，晚自习是比较宽松的，老师只是不定时来巡视，还是有机会偷偷睡一会儿的。现在的问题是，怎么睡才能既睡得好又不会让老师发现呢？

我们从系统的角度分析一下这个例子的特点。

毫无疑问，睡觉的这个人是整个睡觉体制的中心，因为构建这个体制的目的就是为了让这个人更好的睡觉。老师的巡视则是体制中对睡觉有影响的事件，老师的巡视是不定期的，可能马上就来，也可能整晚上都不来，老师巡视这件事是不能不管的，被抓是要挨批的，这非常符合事件的特点：事件是不可预知的，事件是必须响应的。

我们现在有三种睡觉方案：

方案 A：倒头就睡，管你三七二十一，睡够了再说，要知道有时候老师可能一整晚上都不来的。
方案 B：间歇着睡，先定上闹钟，5 分钟响一次，响了就醒，看看老师来没来，没来的话定上闹钟再睡，如此往复。
方案 C：睡之前让同桌给放哨，然后自己睡觉，什么也不用管，什么时候老师来了，就让同桌戳醒你。

不管你们选择的是哪种方案，我高中那会儿用的可是方案 C，安全又舒服！

方案 C 是很有特点的：本来自习课偷睡觉是你自己的事儿，有没有被老师抓着也是你自己的事儿，这些和同桌是毫无利害关系的，但是同桌这个环节对方案 C 的重要性是不言而喻的，他肩负着监控老师巡视和叫醒睡觉者两项重要任务，是事件驱动机制实现的重要组成部分。

上面这个例子是对事件驱动机制最朴素的解释，现实生活中类似的例子还有很多，网上书店也是一个典型的例子。没有订单的时候，网站的工作人员除了维护网站几乎是没有事情可做的，一旦网友下了订单，这份订单就启动了一系列的动作，打单、出货、配送、签收、付款，到此订单交易就算完成了。在这个例子里，网站电子交易系统的角色就类似于上个例子中那位放哨的同学。

在事件驱动机制中，对象对于外部事件总是处于“休眠”状态的，而把对外部事件的检测和监控交给了第三方组件。一旦第三方检测到外部事件发生，它就会启动某种机制，将对象从“休眠”状态中唤醒，并将事件告知对象。对象接到通知后，做出一系列动作，完成对本次事件响应，然后再次进入“休眠”状态，如此周而复始。

有没有发现，事件驱动机制和单片机的中断原理上很相似？

2、事件驱动与单片机编程

现在我们再回到单片机系统中来，看看事件驱动思想在单片机程序设计中的应用。

当我还是一个单片机菜鸟的时候(当然，我至今也没有成为单片机高手)，网络上的大虾们就谆谆教导：一个好的单片机程序是要分层的。曾经很长一段时间，我对分层这个概念完全没有感觉。

什么是程序分层？
程序为什么要分层？
应该怎么给程序分层？

随着手里的代码量越来越多，实现的功能也越来越多，软件分层这个概念在我脑子里逐渐地清晰起来，我越来越佩服大虾们的高瞻远瞩。

单片机的软件确实要分层的，最起码要分两层：驱动层和应用层。

应用是单片机系统的核心，与应用相关的代码担负着系统关键的逻辑和运算功能，是单片机程序的灵魂。硬件是程序感知外界并与外界打交道的物质基础，硬件的种类是多种多样的，各类硬件的操作方式也各不相同，这些操作要求严格、精确、琐细、繁杂。

与硬件打交道的代码只钟情于时序和寄存器，我们可以称之为驱动相关代码；与应用相关的代码却只专注于逻辑和运算，我们可称之为应用相关代码。这种客观存在的情况是单片机软件分层最直接的依据，所以说，将软件划分为驱动层和应用层是程序功能分工的结果。

那么驱动层和应用层之间是如何衔接的呢？

在单片机系统中，信息的流动是双向的，由内向外是应用层代码主动发起的，实现信息向外流动很简单，应用层代码只需要调用驱动层代码提供的 API 接口函数即可，而由外向内则是外界主动发起的，这时候应用层代码对于外界输入需要被动的接收，这里就涉及到一个接收机制的问题，事件驱动机制足可胜任这个接收机制。

外界输入可以理解为发生了事件，在单片机内部直接的表现就是硬件生成了新的数据，这些数据包含了事件的全部信息，事件驱动机制的任务就是将这些数据初步处理(也可能不处理)，然后告知应用层代码，应用代码接到通知后把这些数据取走，做最终的处理，这样一次事件的响应就完成了。

说到这里，可能很多人突然会发现，这种处理方法自己编程的时候早就用过了，只不过没

有使用“事件驱动”这个文绉绉的名词罢了。其实事件驱动机制本来就不神秘，生活中数不胜数的例子足以说明它应用的普遍性。

下面的这个小例子是事件驱动机制在单片机程序中最常见的实现方法，假设某单片机系统用到了以下资源：

一个串口外设 Uart0，用来接收串口数据；
一个定时器外设 Tmr0，用来提供周期性定时中断；
一个外部中断管脚 Exi0，用来检测某种外部突发事件；
一个 I/O 端口 Port0，连接独立式键盘，管理方式为定时扫描法，挂载到 Tmr0 的 ISR；

这样，系统中可以提取出 4 类事件，分别是 UART、TMR、EXI、KEY ，其中 UART 和 KEY 事件发生后必须开辟缓存存储事件相关的数据。所有事件的检测都在各自的 ISR 中完成，然后 ISR 再通过事件驱动机制通知主函数处理。

为了实现 ISR 和主函数通信，我们定义一个数据类型为 INT8U 的全局变量 g_u8EvntFlgGrp，称为事件标志组，里面的每一个 bit 位代表一类事件，如果该 bit 值为 0，表示此类事件没有发生，如果该 bit 值为 1，则表示发生了此类事件，主函数必须及时地处理该事件。图 5 所示为 g_u8EvntFlgGrp 各个 bit 位的作用。

0	1	2	3	4	5	6	7
UART	TMR	EXI	KEY	N/A	N/A	N/A	N/A

图 5 事件标志组 g_u8EvntFlgGrp 各位功能分配

程序清单 List9 所示就是按上面的规划写成的示例性代码。

程序清单 List9：

```
#define FLG_UART    0x01
#define FLG_TMR     0x02
#define FLG_EXI     0x04
#define FLG_KEY     0x08

volatile INT8U g_u8EvntFlgGrp = 0;                  /*事件标志组*/

INT8U read_envt_flg_grp(void);
/*****************************************
*FuncName    : main
*Description :   主函数
*Arguments   : void
*Return      : void
*History     : [2011-08-21]
*                  创建；
*****************************************/
void main(void)
```

```
{
    INT8U u8FlgTmp = 0;

    sys_init();

    while(1)
    {
        u8FlgTmp = read_envt_flg_grp();     /*读取事件标志组*/
        if(u8FlgTmp )                       /*是否有事件发生？ */
        {
            if(u8FlgTmp & FLG_UART)
            {
                action_uart();              /*处理串口事件*/
            }

            if(u8FlgTmp & FLG_TMR)
            {
                action_tmr();               /*处理定时中断事件*/
            }

            if(u8FlgTmp & FLG_EXI)
            {
                action_exi();               /*处理外部中断事件*/
            }

            if(u8FlgTmp & FLG_KEY)
            {
                action_key();               /*处理击键事件*/
            }
        }
        else
        {
            ;/*idle code*/
        }
    }
}
/*********************************************
*FuncName    : read_envt_flg_grp
*Description :  读取事件标志组  g_u8EvntFlgGrp ，
*               读取完毕后将其清零。
*Arguments   : void
*Return      : void
*History     : [2011-08-21]
*                 创建；
*********************************************/
INT8U read_envt_flg_grp(void)
```

```
{
    INT8U u8FlgTmp = 0;

    gbl_int_disable();
    u8FlgTmp = g_u8EvntFlgGrp;    /*读取标志组*/
    g_u8EvntFlgGrp = 0;           /*清零标志组*/
    gbl_int_enable();

    return u8FlgTmp;
}
/*********************************************
*FuncName    : uart0_isr
*Description : uart0 中断服务函数
*Arguments   : void
*Return      : void
*History     : [2011-08-21]
*                 创建；
*********************************************/
void uart0_isr(void)
{
    ......
    push_uart_rcv_buf(new_rcvd_byte); /*新接收的字节存入缓冲区*/

    gbl_int_disable();
    g_u8EvntFlgGrp |= FLG_UART;     /*设置 UART 事件标志*/
    gbl_int_enable();
    ......
}
/*********************************************
*FuncName    : tmr0_isr
*Description : timer0 中断服务函数
*Arguments   : void
*Return      : void
*History     : [2011-08-21]
*                 创建；
*********************************************/
void tmr0_isr(void)
{
    INT8U u8KeyCode = 0;
    ......
    gbl_int_disable();
    g_u8EvntFlgGrp |= FLG_TMR;          /*设置 TMR 事件标志*/
    gbl_int_enable();
    ......
    u8KeyCode = read_key();             /*读键盘*/
```

```
    if(u8KeyCode)                           /*有击键操作？*/
    {
        push_key_buf(u8KeyCode);            /*新键值存入缓冲区*/

        gbl_int_disable();
        g_u8EvntFlgGrp |= FLG_KEY;      /*设置 TMR 事件标志*/
        gbl_int_enable();
    }
    ......
}
/*********************************************
*FuncName    : exit0_isr
*Description : exit0 中断服务函数
*Arguments    : void
*Return       : void
*History      : [2011-08-21]
*                   创建；
*********************************************/
void exit0_isr(void)
{
    ......
    gbl_int_disable();
    g_u8EvntFlgGrp |= FLG_EXI;          /*设置 EXI 事件标志*/
    gbl_int_enable();
    ......
}
```

看一下程序清单 List9 这样的程序结构，是不是和自己写过的某些程序相似？

对于事件驱动机制的这种实现方式，我们还可以做得更绝一些，形成一个标准的代码模板，做一个包含位段和函数指针数组的结构体，位段里的每一个元素作为图 5 那样的事件标志位，然后在函数指针数组中放置各个事件处理函数的函数地址，每个处理函数对应位段里的每个标志位。这样，main()函数中的事件处理代码就可以做成标准的框架代码。

应该说，这样的实现方式是很好的，足以轻松地应对实际应用中绝大多数的情况。但是，事件驱动机制用这样的方式实现真的是完美的么？

在我看来，这种实现方式至少存在两个问题：

不同事件集中爆发时，无法记录事件发生的前后顺序。
同一事件集中爆发时，容易遗漏后面发生的那次事件。

图 6 所示为某一时段单片机程序的执行情况，某些特殊情况下，会出现上面提到的两个问题。

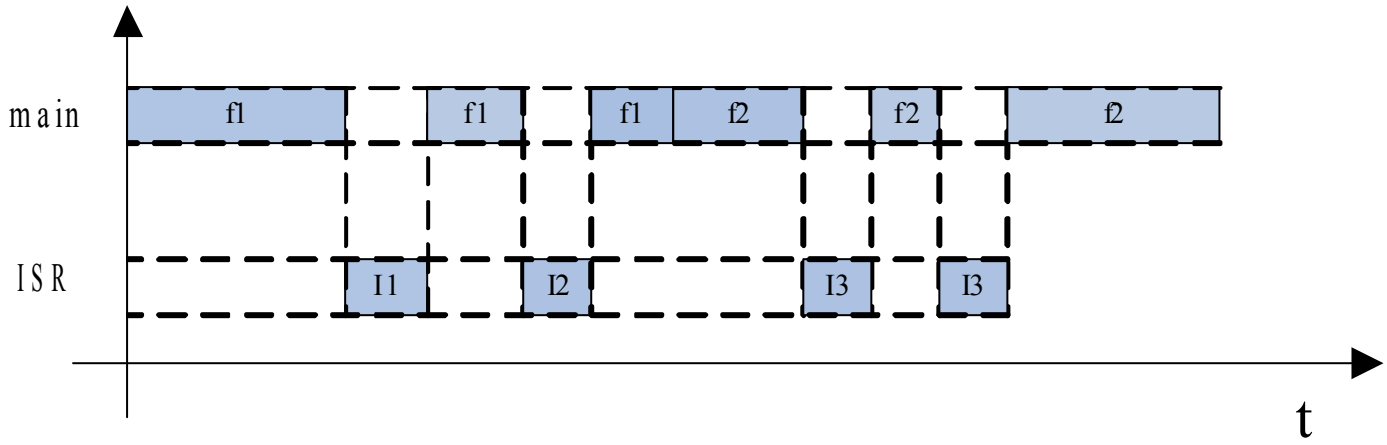

图 6 单片机程序事件驱动运行模拟图

图中，f1 为某事件的处理函数，f2 为另一事件的处理函数，I1、I2、I3 为 3 个不同事件触发的 ISR，假定 I1、I2、I3 分别对应事件 E1、E2、E3。

从图中可以看出，主函数在调用事件处理函数 f1 的时候，发生了 2 次事件，主函数被 I1 和 I2 中断了 2 次，I1 和 I2 执行的时候各自置位了相应的事件标志位。函数 f1 返回后，主函数又调用了另一个事件处理函数 f2，f2 执行期间先后发生了 2 次同样的事件，f2 被 I3 中断了 2 次，对应的事件标志位被连续置位了 2 次。

在图 6 中我们当然可以看出 I1 先于 I2 执行，即事件 E1 发生在事件 E2 之前，但是主函数再次读取事件标志组 g_u8EvntFlgGrp 的时候，看到的是两个“同时”被置位的标志位，无法判断出事件 E1 和 E2 发生的先后顺序，也就是说有关事件发生先后顺序的信息丢失了，这就是前面说的第 1 个问题：不同事件集中爆发时，无法记录事件发生的前后顺序。

在程序清单 List9 中，主函数在处理事件时，按照程序预先设定好的顺序，一个一个地处理发生的事件，如果不同事件某时段集中爆发，可能会出现事件的发生顺序和事件的处理顺序不一致的情况。倘若系统功能对事件的发生顺序敏感，那么程序清单 List9 中的程序就不能满足要求了。

同样的道理，如果 I3 对应的事件 E3 是程序清单 List9 中 EXI 那样的事件(这种事件没有缓冲机制)，事件 E3 第 2 次的发生就被遗漏了，这就是前面所说的第 2 个问题：同一事件集中爆发时，容易遗漏后后面发生的事件。

如果系统功能对事件 E3 的发生次数敏感，程序清单 List9 中的程序也是不能满足要求的。

既然事件驱动机制这样的实现方式存在缺陷，那么有没有一种更好的实现方式呢？当然有！把事件转换成消息存入消息队列就能完美解决这个问题，只不过大家不要对我这种自导自演的行文方式产生反感就好。

3、事件驱动与消息

什么是消息？

消息这个词在现实生活中是很普遍的，根据生活经验，我们对“消息”这个概念都有自己

的理解，但是我们现在谈的是在计算机的世界里的“消息”，我觉得这个“消息”和现实生活中的“消息”还是有点儿不一样的，希望大家在看下文的时候保持这个前提。

消息是数据信息的一种存储形式。从程序的角度看，消息就是一段存储着特定数据的内存块，数据的存储格式是设计者预先约定好的，只要按照约定的格式读取这段内存，就能获得消息所承载的有用信息。

消息是有时效性的。任何一个消息实体都是有生命周期的，它从诞生到消亡先后经历了生成、存储、派发、消费共 4 个阶段：消息实体由生产者生成，由管理者负责存储和派发，最后由消费者消费。被消费者消费之后，这个消息就算消亡了，虽然存储消息实体的内存中可能还残留着原来的数据，但是这些数据对于系统来讲已经没有任何意义了，这也就是消息的时效性。

说到这里，大家有没有发现，这里的“消息”和前面一直在说的“事件”是不是很相似？把“消息”的这些特点套用在“事件”身上是非常合适的，在我看来，消息无非是事件的一个马甲而已。

我喜欢把事件和消息分别比作历史和史书。历史事件和历史人物是时间上的一种客观存在，稍纵即逝，是不能保存的，而史书则是对这些人和事的记录，是可以长久保存的，后人通过史书的记载，就能还原历史。同样，程序通过分析消息就能获知事件。

我们在设计单片机程序的时候，都怀着一个梦想，即让程序对事件的响应尽可能的快，理想的情况下，程序对事件要立即响应，不能有任何延迟。这当然是不可能的，当事件发生时，程序总会因为这样那样的原因不能立即响应事件。

为了不至于丢失事件，我们可以先在事件相关的 ISR 中把事件加工成消息并把它存储在消息缓冲区里，ISR 做完这些后立即退出。主程序忙完了别的事情之后，去查看消息缓冲区，把刚才 ISR 存储的消息读出来，分析出事件的有关信息，再转去执行相应的响应代码，最终完成对本次事件的响应。只要整个过程的时间延迟在系统功能容许的范围之内，这样处理就没有问题。

将事件转化为消息，体现了以空间换时间的思想。

再插一句，虽然事件发生后对应的 ISR 立即被触发，但是这不是严格意义上的“响应”，顶多算是对事件的“记录”，“记录”和“响应”是不一样的。

事件是一种客观的存在，而消息则是对这种客观存在的记录。对于系统输入而言，事件是其在时间维度上的描述；消息是其在空间维度上的描述，所以，在描述系统输入这一功能上，事件和消息是等价的。

对比一下程序清单 List9 中的实现方式，那里是用全局标志位的方式记录事件，对于某些特殊的事件还配备了专门的缓冲区，用来存储事件的额外信息，而这些额外信息单靠全局标志位是无法记录的。

现在我们用消息+消息缓冲区的方式来记录事件，消息缓冲区就成了所有事件共用的缓冲区，无论发生的事件有没有额外的信息，一律以消息的形式存入缓冲区。

为了记录事件发生的先后顺序，消息缓冲区应该做成以“先入先出”的方式管理的环形缓冲队列。事件生成的消息总是从队尾入队，管理程序读取消息的时候总是从队头读取，这样，消息在缓冲区中存储的顺序就是事件在时间上发生的顺序，先发生的事件总是能先得到响应。一条消息被读取之后，管理程序回收存储这个消息的内存，将其作为空闲节点再插入缓冲队列的队尾，用以存储将来新生成的消息。

图 7 所示为使用了消息功能的事件驱动机制示意图。不知道有没有人对图中的“消费者”有疑问，这个“消费者”在程序中指的是什么呢？既然这个事件/消息驱动机制是为系统应用服务的，消费者当然就是指应用层的代码了，更明确一点儿的话，消费者就是应用代码中的状态机！

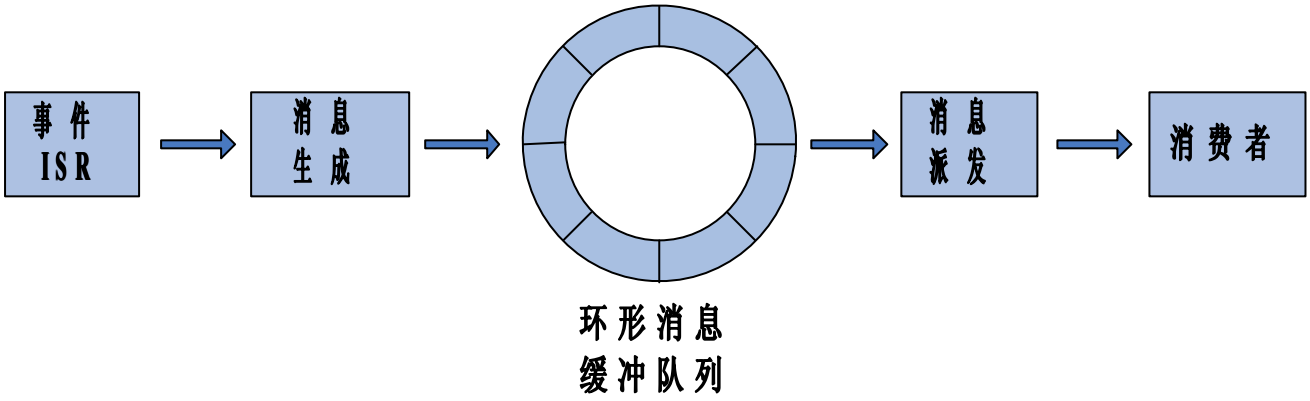

图 7 事件/消息驱动机制

用消息的方法来实现事件驱动机制完全解决了前面提到的那两个问题，即

不同事件集中爆发时，无法记录事件发生的前后顺序。
同一事件集中爆发时，容易遗漏后面发生的那次事件。

对于第一种情况，消息(事件)在缓冲队列中是以“先入先出”的方式存储的，存储顺序就代表了事件发生的先后顺序。对于第二种情况，任何被 ISR 捕捉到的事件都会以一个独立的消息实体存入缓冲队列，即使前后两个是同一个事件，只要 ISR 反应够快就不会遗漏事件。实际上，ISR 的主要工作就是填写消息实体，然后将其存入缓冲队列，做这些工作只占用 CPU 很短的时间。

接下来再说一说这个消息机制在程序中如何实现。

在程序中，消息机制可以看做是一个独立的功能模块，一个功能模块的实现无非就是数据结构+算法。先来看消息机制的数据结构。这里的数据结构是指和消息机制有关的数据组织形式，包含 2 个部分：

消息节点自身的数据组织形式
消息缓冲区的数据组织形式

程序清单 List10 所示就是消息机制的数据结构。

程序清单 List10：

```
typedef union msg_arg              /*消息参数共用体*/
{
    INT8U u8Arg;                   /*成员：8 位无符号*/
    INT8U s8Arg;                   /*成员：8 位有符号*/
#if CFG_MSG_ARG_INT16_EN>0
    INT16U u16Arg;                 /*可选成员：16 位无符号*/
    INT16S s16Arg;                 /*可选成员：16 位有符号*/
#endif
#if CFG_MSG_ARG_INT32_EN>0
    INT32U u32Arg;                 /*可选成员：32 位无符号*/
    INT32S s32Arg;                 /*可选成员：32 位有符号*/
#endif
#if CFG_MSG_ARG_FP32_EN>0
    FP32 f32Arg;                   /*可选成员：32 位单精度浮点*/
#endif
#if CFG_MSG_ARG_PTR_EN>0
    void* pArg;                    /*可选成员：void 指针*/
#endif
}MSG_ARG;

typedef struct _msg            /*消息结构体*/
{
    INT8U       u8MsgID;       /*消息 ID*/
#if CFG_MSG_USR_SUM > 1
    INT8U       u8UsrID;       /*消费者 ID*/
#endif
    MSG_ARG    uMsgArg;        /*应用消息参数*/

} MSG;

typedef struct msg_box     /*消息缓冲区结构体*/
{
    INT8U    u8MBLock;                                  /*队列上锁标识*/

    INT8U    u8MsgSum;                                  /*队列长度*/

    INT8U    u8MQHead;                                  /*队列头结点位置*/

    INT8U    u8MQTail;                                  /*队列尾节点位置*/

    MSG      arMsgBox[CFG_MSG_SUM_MAX];     /*存放队列的数组*/

} MB;

static MB g_stMsgUnit;                          /*消息管理单元全局变量*/
```

消息的数据结构包含 2 部分：消息头和消息参数，在消息结构体 MSG 中，u8MsgID 和 u8UsrID 就是消息头，共用体 MSG_ARG 就是消息参数。

u8MsgID 是消息的类型标志，也就是生成此消息的事件的事件类型标志，程序根据这个成员选择对应的事件处理函数；u8UsrID 是消息的消费者代号，如果应用代码中只有一个消费者，则成员 u8UsrID 可以忽略。

MSG_ARG 就是消息附带的参数，也就是事件的内容信息。系统中的事件是多种多样的，有的事件只需要类型标志即可，有的事件可能还需要整型变量存储事件内容，还有的事件可能需要大块的内存来存储一些附带的数据。为了将各种类型的事件生成的消息纳入统一管理，要求 MSG_ARG 必须能存储各种类型的数据，因此 MSG_ARG 被定义成了共用体。

从程序清单 List10 中可以看出，MSG_ARG 既可以存储 8 位~32 位有符号无符号整型数据，又可以存储单精度浮点，还可以存储 void*型的指针变量，而 void*的指针又可以强制转换成任意类型的指针，所以 MSG_ARG 可以存储指向任意类型的指针。

对于 MSG_ARG 中的某些成员，还配备了预编译常量 CFG_MSG_ARG_XXX_EN 加以控制，如果实际应用中不需要这些耗费内存较大的数据类型，可以设置 CFG_MSG_ARG_XXX_EN 去掉它们。全开的情况下，每个消息节点占用 6 个字节的内存，最精简的情况下，每个消息节点只占用 2 个字节。

全局结构体变量 g_stMsgUnit 是消息缓冲区的数据结构。

消息缓冲区是一个环形缓冲队列，这里将环形队列放在了一个一维数组中，也就是 g_stMsgUnit 的成员 arMsgBox[]，数组元素的数据类型就是消息结构体 MSG ，数组的大小由预编译常量 CFG_MSG_SUM_MAX 控制，该常量是环形缓冲队列的最大容量。

理论上，CFG_MSG_SUM_MAX 值的选取越大越好，但考虑到单片机的 RAM 资源有限，CFG_MSG_SUM_MAX 值的选取要在资源消耗和实际最大需求之间折中，只要能保证在最坏情况下环形缓冲队列仍有裕量即可。

用数组实现环形队列还需要一些辅助变量，也就是 g_stMsgUnit 剩余的成员。

u8MBLock 是队列的控制变量，u8MBLock>0 表示队列处于锁定/保护状态，不能写也不能读，u8MBLock=0 表示队列处于正常状态，可写可读；

u8MsgSum 是队列长度计数器，记录着当前队列中存有多少条消息，存入一条消息 u8MsgSum++，读出一条消息 u8MsgSum--；

u8MQHead 记录着当前队头消息节点在数组 arMsgBox[]中的位置，其值就是数组元素的下标，消息读取的时候所读出的就是 u8MQHead 所指向的节点，读完之后 u8MQHead 向队尾方向移动一个位置，指向新的队头节点；

u8MQTail 记录着当前队尾消息节点在数组 arMsgBox[]中的位置，其值是数组元素的下标，新消息写入之前，u8MQTail 向队尾方向后移一个位置，然后新写入的消息存入 u8MQTail 所指向的空闲节点；

图 8 所示为消息缓冲区结构体变量 g_stMsgUnit 的示意图。

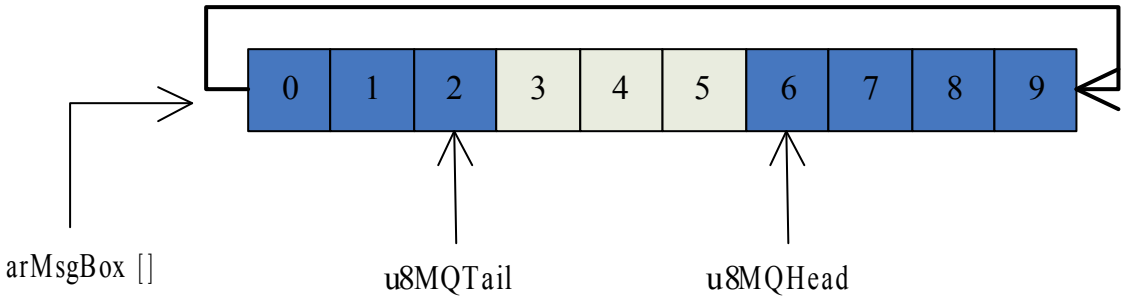

图 8 数组实现的环形消息队列

有了数据结构，还要有对应的算法实现，消息机制的数据主体就是一个数组化了的环形队列，环形队列的算法就是我们所要的算法。

消息机制是一个独立的功能模块，应该对外屏蔽其内部实现细节，而仅对外界开放一定数量的接口函数，外界通过调用这些接口来使用消息功能，这也就是我在声明 g_stMsgUnit 变量的时候使用了 static 关键词的原因。

消息模块的接口函数一共有 9 个：

void mq_init(void)
消息队列初始化，负责初始化 g_stMsgUnit 。

void mq_clear(void)
清空消息队列，效果同 mq_init()，可有可无。

void mq_lock(void)
消息队列锁定，锁定的消息队列不可读不可写。

void mq_unlock(void)
消息队列解锁，解锁后消息队列恢复正常功能。

BOOL mq_is_empty(void)
消息队列判空，返回 TRUE 表示消息队列当前为空，返回 FALSE 表示有消息存储。

INT8U mq_get_msg_cur_sum(void)
查询消息队列中当前存储的消息总数，函数返回值为查询结果。

INT8U mq_get_msg_sum_max(void)
查询消息队列的最大容量，函数返回值为查询结果。

INT8U mq_msg_post_fifo(MSG* pMsg)
向消息队列中寄送消息，方式为先入先出，形参 pMsg 指向消息的备份内存，函数返回操作结果。该函数多被 ISR 调用，所以必须为可重入函数。

INT8U mq_msg_req_fifo(MSG* pMsg)
从消息队列中读取消息，方式为先入先出，函数将读出的消息存入形参 pMsg 指向的内存，函数返回操作结果。该函数被主程序调用，可以不是可重入函数，但要对共享数据进行临界保护。

4、小结

事件/消息驱动机制是一个标准的通用的框架，配合 ISR，对任何系统输入都能应对自如。事件/消息驱动机制屏蔽了应用层程序获取各种系统输入的工作细节，将系统输入抽象整合，以一种标准统一的格式提交应用代码处理，极大地减轻了应用层代码获取系统输入的负担，应用层只需要专注于高级功能的实现就可以了。

从软件分层的角度来看，事件/消息驱动机制相当于驱动层和应用层之间的中间层，这样的层次结构如图 9 。

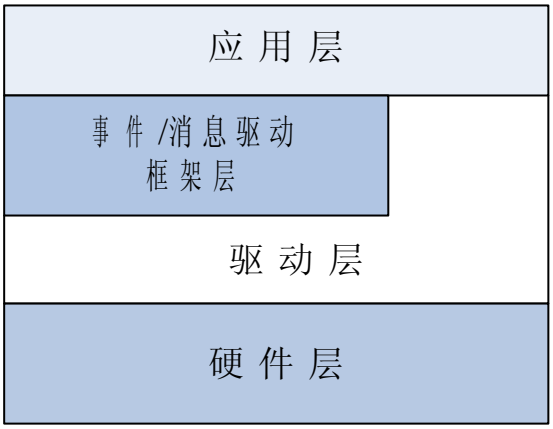

图 9 事件/消息驱动结构软件分层示意图

图 9 中之所以驱动层和应用层之间还有接触，是因为系统输出响应的时候，应用层可能还需要直接调用驱动层提供的函数接口。

如果一个单片机的软件是图 9 这样的结构，并且应用层的程序使用状态机来实现，在消息的驱动下使应用层的状态机运转起来，那么这个软件的设计思想就是整篇文章的主题：基于事件/消息驱动+状态机结构的裸奔通用框架。

四、裸奔程序框架 GF1.0：状态机+事件/消息驱动

事件/消息驱动和状态机是天生的搭档，这对黄金组合是分析问题解决问题的利器。

1、牛刀小试

上一个小例子，把图 2 中那个按键控制流水灯的例子改一下功能要求。

规则描述：

1) L1L2 状态转换顺序 OFF/OFF--->ON/OFF--->ON/ON--->OFF/ON--->OFF/OFF
2) 通过按键控制 L1L2 的状态,每次状态转换只需按键 1 次
3) 从上一次按键的时刻开始计时，如果 10 秒钟之内没有按键事件，则不管当前 L1L2 状态如何，一律恢复至初始状态。
4) L1L2 的初始状态 OFF/OFF

现在我们用状态机+事件/消息驱动的思想来分析问题。

系统中可提取出两个事件：按键事件和超时事件，分别用事件标志 KEY 和 TOUT 代替。

L1L2 的状态及转换关系可做成一个状态机，称为主状态机，共 4 个状态：LS_OFFOFF、LS_ONOFF、LS_ONON、LS_OFFON 。主状态机会在事件 KEY 或 TOUT 的驱动下发生状态迁移，各个状态之间的转换关系比较简单，在此略过。

事件/消息驱动机制的任务就是检测监控事件 KEY 和 TOUT，并提交给主状态机处理。检测按键需要加入消抖处理，消抖时间定为 20ms，10S 超时检测需要一个定时器进行计时。

这里将按键检测程序部分也做成一个状态机，共有 3 个状态：

WAIT_DOWN ：空闲状态，等待按键按下
SHAKE ：初次检测到按键按下，延时消抖
WAIT_UP ：消抖结束，确认按键已按下，等待按键弹起

按键状态机的转换关系可在图 10 中找到。

按键检测和超时检测共用一个定时周期为 20ms 的定时中断，这样就可以把按键检测和超时检测的代码全部放在这个定时中断的 ISR 中。我把这个中断事件用 TICK 标记，按键状态机在 TICK 的驱动下运行，按键按下且消抖完毕后触发 KEY 事件，而超时检测则对 TICK 进行软时钟计数，记满 500 个 TICK 则超时 10S，触发 TOUT 事件。

有了上面的分析，实现这个功能的程序的结构就十分清晰了， 图 10 是这个程序的结构示意图，这张图表述问题足够清晰了，具体的代码就不写了。

仔细瞅瞅，是不是有点儿那个意思了？

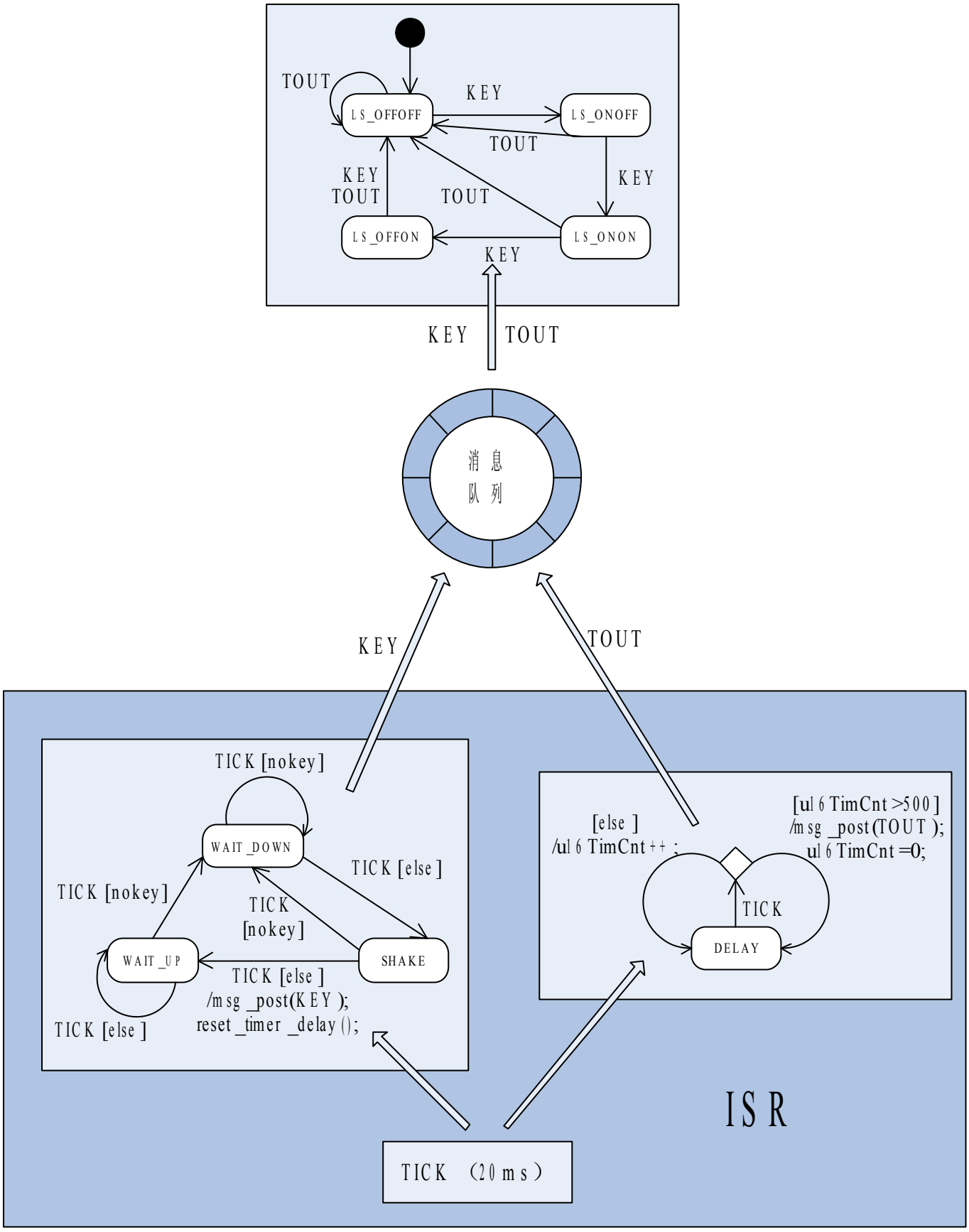

图 10 牛刀小试的程序结构示意图

如果忽略定时中断 ISR 中的细节，图 10 中的整个程序结构就是事件/消息驱动+主状态机的结构，ISR 是消息的生产者，与消息缓冲、派发相关的程序部分是管理者，而主状态机则是消息的消费者，应用层代码中只有这一个状态机，是消息的唯一消费者。

这个结构就是通用框架 GF1.0 的标准结构：多个 ISR + 一个消息缓冲区 + 一个应用层主状态机。

ISR 生成的消息(事件)全部提交主状态机处理，在消息的驱动下主状态机不断地迁移。如果把应用层主状态机看做是一台发动机，那么 ISR 生成的消息则是燃料，事件不断的发生，消息不断的生成，有了燃料(消息)的供给，发动机(主状态机)就能永不停息地运转。

接下来关注一下图 10 中的 ISR，这个 ISR 里面的内容是很丰富的，里面还套着 2 个小状态机：按键状态机和计时状态机。按键状态机自不必说，这个计时部分也可以看做是一个状态机，不过这个状态机比较特殊，只有一个状态 DELAY。

既然是状态机，想要跑起来就需要有事件来驱动，在这个 ISR 里，定时器的中断事件 TICK 就是按键状态机和计时状态机的驱动，只不过这两个事件驱动+状态机结构没有消息缓冲，当然也不需要消息缓冲，因为状态机在 ISR 中，对事件是立即响应的。

从宏观上看，图 10 中是事件/消息驱动+状态机，从微观上看，图 10 中的 ISR 也是事件驱动+状态机。ISR 中的状态机在迁移过程中生成消息(事件)，而这些消息(事件)对于主状态机来讲又是它自己的驱动事件。事件的级别越高，事件自身也就越抽象，描述的内容也就越接近人的思维方式。我觉得这种你中有我我中有你的特点正是事件驱动+状态机的精髓所在。

2、裸奔通用框架 GF1.0

前面说过，状态机总是被动地接受事件，而 ISR 也只是负责将消息(事件)送入消息缓冲区，这些消息仅仅是数据，自己肯定不会主动地跑去找状态机。那么存储在缓冲区中的消息(事件)是怎么被发送到目标状态机呢？

把消息从缓冲区中取出并送到对应的状态机处理，这是状态机调度程序的任务，我把这部分程序称作状态机引擎(State Machine Engine ， 简写作 SME)。图 11 是 SME 的大致流程图。

从图 11 可以看出，SME 的主要工作就是不断地查询消息缓冲队列，如果队列中有消息，则将消息按先入先出的方式取出，然后送入状态机处理。SME 每次只处理一条消息，反复循环，直到消息队列中的消息全部处理完毕。

当消息队列中没有消息时，CPU 处于空闲状态，SME 转去执行“空闲任务”。空闲任务指的是一些对单片机系统关键功能的实现无关紧要的工作，比如喂看门狗、算一算 CPU 使用率之类的工作，如果想降低功耗，甚至可以让 CPU 在空闲任务中进入休眠状态，只要事件一发生，CPU 就会被 ISR 唤醒，转去执行消息处理代码。

实际上，程序运行的时候 CPU 大部分时间是很“闲”的，所以消息队列查询和空闲任务这两部分代码是程序中执行得最频繁的部分，也就是图 11 的流程图中用粗体框和粗体线标出的部分。

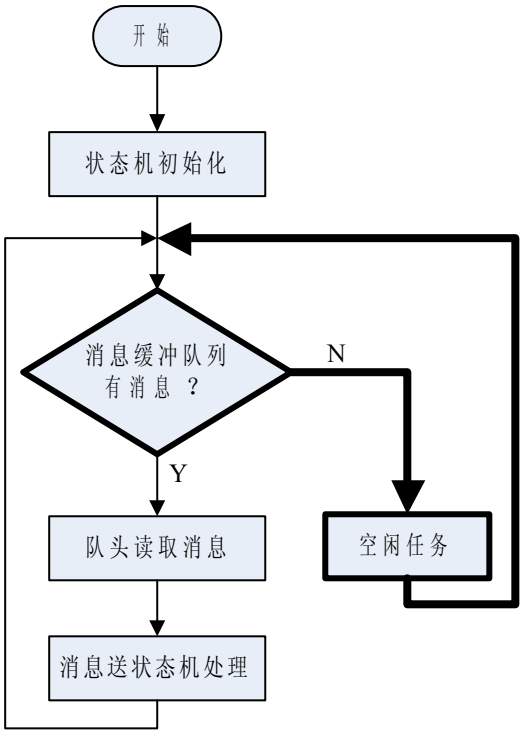

图 11 GF1.0 状态机引擎流程图

如果应用层的主状态机用压缩表格驱动法实现，结合第三章第 3 节给出的消息模块，则 GF1.0 的状态机引擎代码如程序清单 List11 所示。

程序清单 List11：

```
void sme_kernel(void);
/*****************************************
*FuncName    : main
*Description :   主函数
*Arguments   : void
*Return      : void
*History     : [2011-08-28]
*                  创建；
*****************************************/
void main(void)
{
    sys_init();

    sme_kernel();          /*GF1.0 状态机引擎*/
}
/*****************************************
*FuncName    : sme_kernel
*Description :   裸奔框架 GF1.0 的状态机引擎函数
*Arguments   : void
*Return      : void
*History     : [2011-08-28]
```

```
*                   创建；
**************************************/
void sme_kernel(void)
{
    extern struct fsm_node g_arFsmDrvTbl[];       /*状态机压缩驱动表格*/

    INT8U u8Err       = 0;                          /**/
    INT8U u8CurStat = 0;                            /*状态暂存*/
    MSG stMsgTmp;                                   /*消息暂存*/
    struct fsm_node stNodeTmp = {NULL, 0};        /*状态机节点暂存*/
    memset((void*)(&stMsgTmp), 0, sizeof(MSG));     /*变量初始化*/

    gbl_int_disable();              /*关全局中断*/
    mq_lock();                      /*消息队列锁定*/
    mq_init();                      /*消息队列初始化*/
    mq_unlock();                    /*消息队列解锁*/
    fsm_init();                     /*状态机初始化*/
    gbl_int_enable();               /*开全局中断*/

    while(1)
    {
        if(mq_is_empty() == FALSE)
        {
            u8Err = mq_msg_req_fifo(&stMsgTmp);     /*读取消息*/

            if(u8Err == MREQ_NOERR)
            {
                u8CurStat = get_cur_state();                    /*读取当前状态*/
                stNodeTmp = g_arFsmDrvTbl[u8CurStat];       /*定位状态机节点*/
                if(stNodeTmp.u8StatChk == u8CurStat)
                {
                    u8CurStat = stNodeTmp.fpAction(&stMsgTmp);   /*消息处理*/
                    set_cur_state(u8CurStat );                      /*状态迁移*/
                }
                else
                {
                    state_crash(u8CurStat );                        /*非法状态处理*/
                }
            }
        }
        else
        {
            idle_task();                                            /*空闲任务*/
        }
    }
}
```

3、状态机与 ISR 在驱动程序中的应用

在驱动层的程序中使用状态机和 ISR 能使程序的效率大幅提升。这种优势在通信接口中最为明显，以串口程序为例。

单片机和外界使用串口通信时大多以数据帧的形式进行数据交换，一帧完整的数据往往包含帧头、接收节点地址、帧长、数据正文、校验和帧尾等内容，图 12 所示为这种数据帧的常见结构。

帧头	接收节点地址	帧长	数据正文	校验和	帧尾
FRM_HEAD	FRM_USR	FRM_LEN	DATA(N bytes)	FRM_CHKSUM	FRM_END

图 12 串口通信数据帧结构

图 12 表明的结构只是数据帧的一般通用结构，使用时可根据实际情况适当简化，例如如果是点对点通信，那么接收节点地址 FRM_USR 可省略；如果通信线路没有干扰，可确保数据正确传输，那么校验和 FRM_CHKSUM 也可省略。

假定一帧数据最长不超过 256 个字节且串口网络中通信节点数量少于 256 个，那么帧头、接收节点地址、帧长、帧尾都可以用 1 个字节的长度来表示。虽然数据的校验方式可能不同，但校验和使用 1~4 个字节的长度来表示足以满足要求。

先说串口接收，在裸奔框架 GF1.0 的结构里，串口接收可以有 2 种实现方式：ISR+消息 or ISR+缓冲区+消息。

ISR+消息比较简单，ISR 收到一个字节数据，就把该字节以消息的形式发给应用层程序，由应用层的代码进行后续处理。这种处理方式使得串口接收 ISR 结构很简单，负担也很轻，但是存在 2 个问题。

数据的接收控制是一个很底层的功能，按照软件分层结构，应用代码不应该负责这些工作，混淆职责会使得软件的结构变差；用消息方式传递单个的字节效率太低，占用了太多的消息缓冲资源，如果串口波特率很高并且消息缓冲区开的不够大，会直接导致消息缓冲区溢出。

相比之下，ISR+缓冲区+消息的处理方式就好多了，ISR 收到一个字节数据之后，将数据先放入接收缓冲区，等一帧数据全部接收完毕后(假设缓冲区足够大)，再以消息的形式发给应用层，应用层就可以去缓冲区读取数据。

对于应用层来讲，整帧数据只有数据正文才是它想要的内容，数据帧的其余部分仅仅是数据正文的封皮，没有意义。从功能划分的角度来看，确保数据正确接收是 ISR 的职责，所以这部分事情应该放在 ISR 中做，给串口接收 ISR 配一个状态机，就能很容易的解决问题。图 13 为串口接收 ISR 状态转换图。

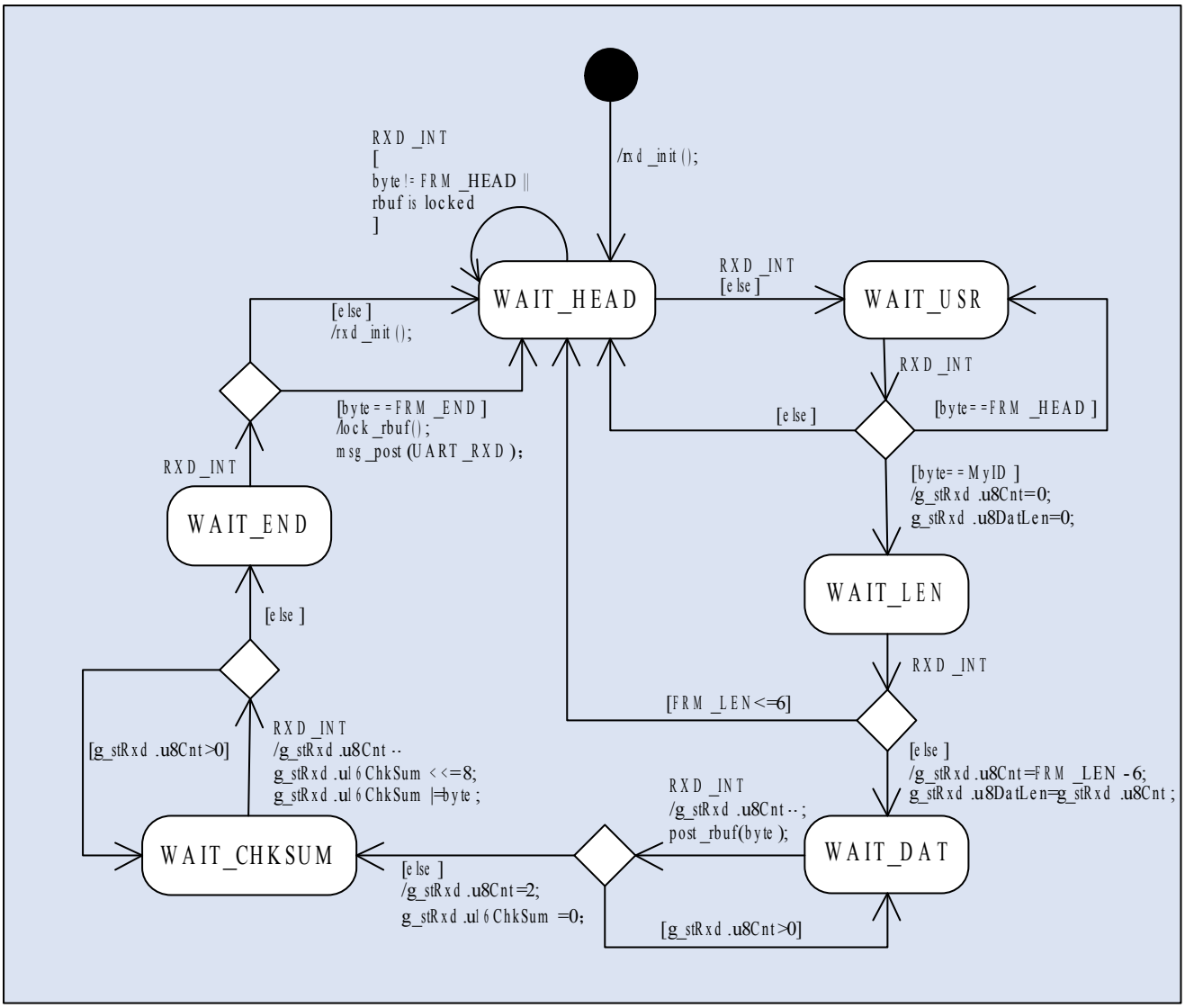

图 13 串口接收 ISR 中的状态转换图

图 13 中的数据帧使用 16 位校验和，发送顺序高字节在前，低字节在后。

接收缓冲区属于 ISR 和主程序的共享资源，必须实现互斥访问，所以 ISR 收完一帧数据之后对缓冲区上锁，后面再发生的 ISR 发现缓冲区上锁之后，不接收新的数据，也不修改缓冲区中的数据。应用层程序收到消息，读取缓冲区中的数据之后再对缓冲区解锁，使能 ISR 接收串口数据和对缓冲区的写入。

数据接收完毕后，应该校验数据，只有校验结果和收到的校验和相符，才能确信数据正确接收。数据校验比较耗时，不适合在 ISR 中进行，所以应该放在应用代码中处理。

这样实现的串口接收 ISR 比较复杂，代码规模比较大，看似和 ISR 代码尽量简短，执行尽量迅速的原则相悖，但是由于 ISR 里面是一个状态机，每次中断的时候 ISR 仅执行全部代码的一小部分，之后立刻退出，所以执行时间是很短的，不会比“ISR+消息”的方式慢多少。

串口发送比串口接收要简单的多，为提高效率也是用 ISR+缓冲区+消息的方式来实现。程

序发送数据时调用串口模块提供的接口函数，接口函数通过形参获取要发送的数据，将数据打包后送入发送缓冲区，然后启动发送过程，剩下的工作就在硬件和串口发送 ISR 的配合下自动完成，数据全部发送完毕后，ISR 向应用层发送消息，如果有需要，应用层可以由此获知数据发送完毕的时刻。图 14 为串口发送 ISR 的状态转换图。

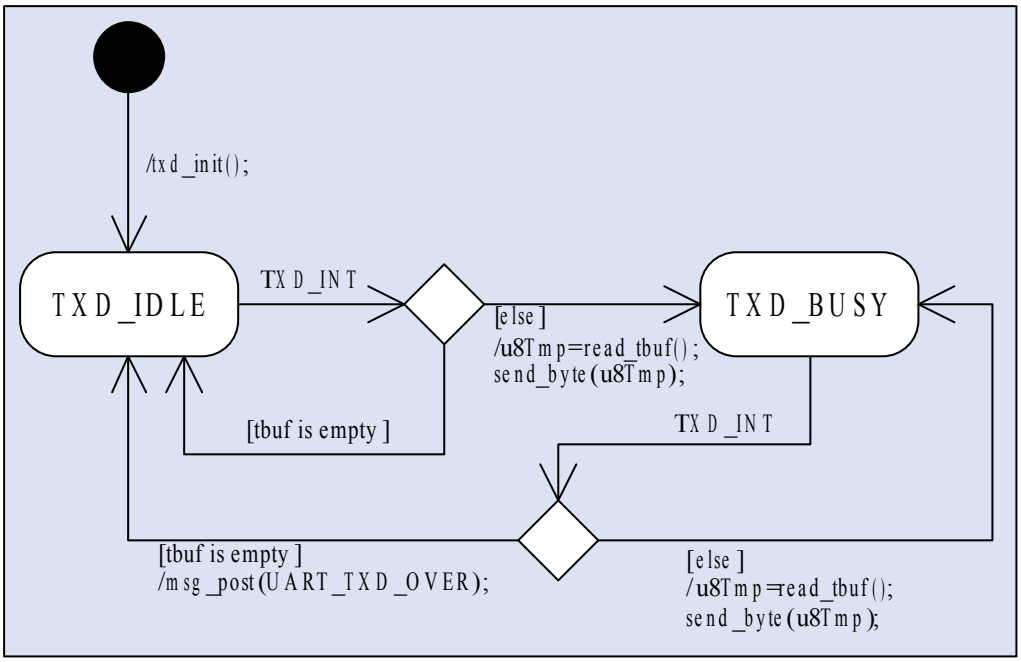

图 14 串口发送 ISR 中的状态转换图

上面只是讨论了串口设备的管理方法，其实这种状态机+ISR 的处理方式可以应用到很多的硬件设备中，一些适用的场合：

1) 标准的或自制的单总线协议 (状态机+定时中断+消息)
2) 用 I/O 模拟 I2C 时序并且通信速率要求不高 (状态机+定时中断+消息)
3) 数码管动态扫描 (状态机+定时中断)
4) 键盘动态扫描 (状态机+定时中断)

4、小结

裸奔框架 GF1.0 处处体现着事件驱动+状态机的思想，大到程序整体的组织结构，小到某个 ISR 的具体实现，都有这对黄金组合的身影。

从宏观上看，裸奔框架 GF1.0 是一个 ISR+消息管理+主状态机的结构，如图 15 所示。不管主状态机使用的是 FSM(有限状态机)还是 HSM(层次状态机)，GF1.0 中有且只有 1 个主状态机。主状态机位于软件的应用层，是整个系统绝对的核心，承担着逻辑和运算功能，外界和单片机系统的交互其实就是外界和主状态机之间的交互，单片机程序的其他部分都是给主状态机打杂的。

从微观上看，裸奔框架 GF1.0 中的每一个 ISR 也是事件驱动+状态机的结构。ISR 的主要任务是减轻主状态机获取外界输入的负担，ISR 负责处理获取输入时硬件上繁杂琐细的操作，将各种输入抽象化，以一种标准统一的数据格式(消息)提交给主状态机，好让主状态机能专注于高级功能的实现而不必关注具体的细节。

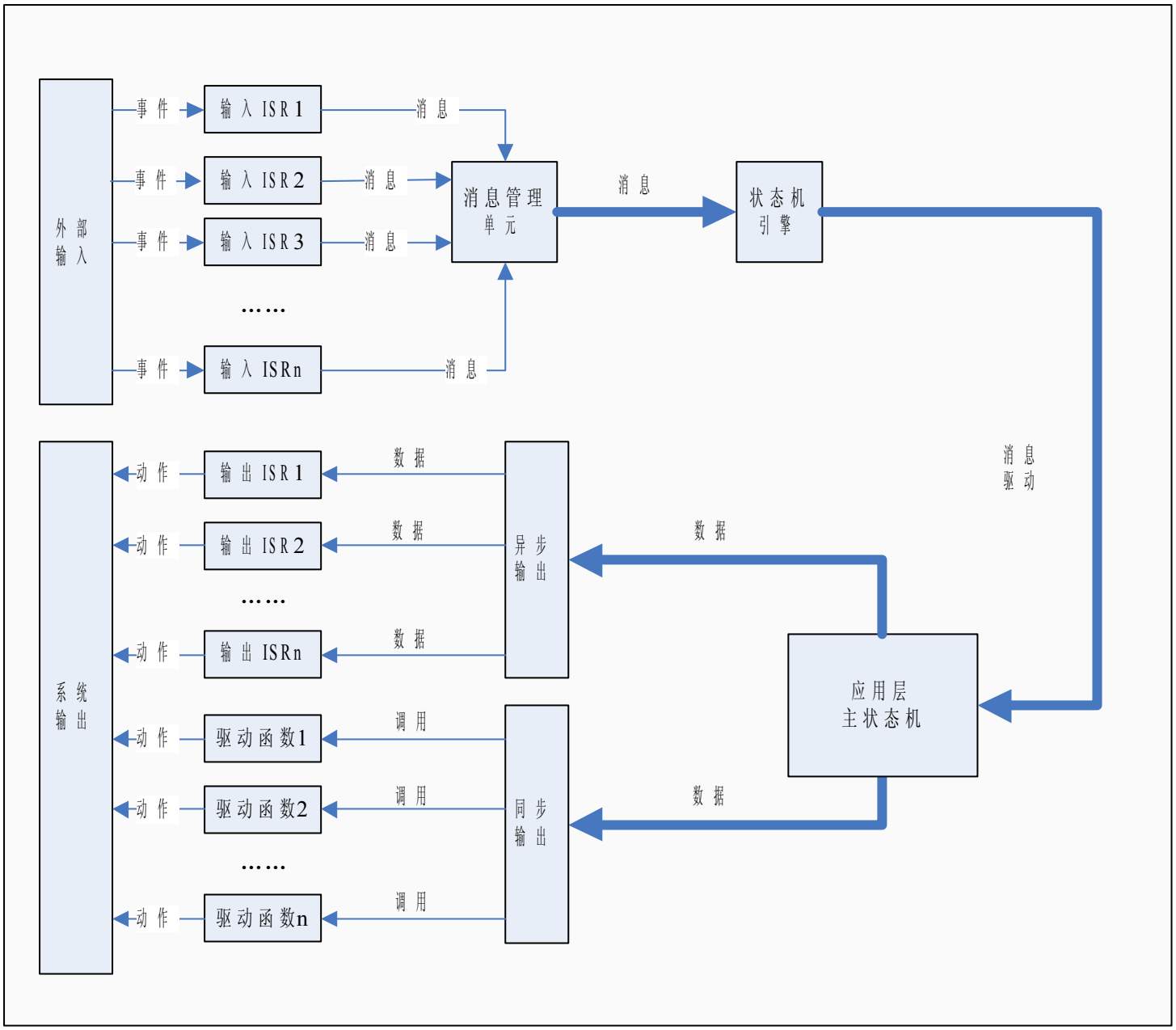

图 15 裸奔框架 GF1.0 整体结构图

裸奔框架 GF1.0 应用的难点在于主状态机的具体实现，对于一个实际的应用，不管功能多复杂，都必须将这些功能整合到一个主状态机中来实现。这既要求设计者对系统的目标功能有足够详细的了解，还要求设计者对状态机理论有足够深的掌握程度，如果设计出的状态机不合理，程序的其他部分设计得再好，也不能很好的实现系统的要求。

将实际问题状态机化，最重要的是要合理地划分状态，其次是要正确地提取系统事件，既不能遗漏，也不能重复。有了状态和事件，状态转换图的骨架就形成了，接下来就是根据事件确定状态之间的转换关系，自顶向下，逐步细化，最终实现整个功能。

五、一个更靠谱的裸奔程序框架 GF2.0：并行状态机+双消息驱动

本章节在裸奔通用框架 GF1.0 的基础上提出了一个改进版本 GF2.0，在 GF2.0 的结构中，应用层使用了并行状态机，实现了应用功能的多任务化，又根据并行状态机的特点，将消息机制进行了改进，使用了驱动消息+应用消息这样的双重消息模式。

改进框架 GF2.0 是事件/消息驱动+状态机编程框架的最终版，所以本章是全文的重点，涉及到的内容很多，为了限制篇幅，和 GF1.0 重复的内容在本章中不再涉及，而且叙述新的内容的时候也多以框图加文字说明的方式，较少直接贴出代码。有了前面几个章节的铺垫，我想这种方式不会对读者造成很大的困难。

1、GF1.0 的不足

(1) GF1.0 应用层主状态机的不足

GF1.0 中应用层中有且只有一个状态机(主状态机)，不管应用层要负担多少功能，所有的功能全部要整合到这个状态机中去，如果某些功能之间是相互独立的，这种处理方式直接带来了强耦合的问题，举个例子。

假设系统中存在 4 个事件，分别能生成消息 msg1、msg2、msg3、msg4，系统中有 2 个功能模块，分别为功能 A、功能 B。msg1 和 msg2 只对功能 A 有效，使功能 A 在 2 个工作状态之间切换，msg3 和 msg4 只对功能 B 有效,使功能 B 在 2 个工作状态之间切换，功能 A 功能 B 之间互不影响。

如果我们把功能 A 和功能 B 分开处理，会得到如图 16 所示的结构。

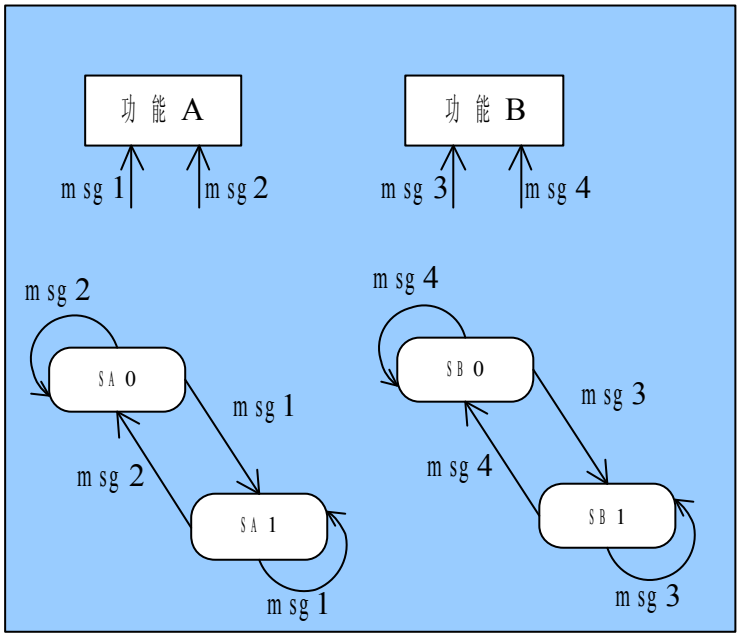

图 16 功能 A 和功能 B 并行处理示意图

如果我们使用 GF1.0 的裸奔框架来实现功能 A 和功能 B，就需要将这 2 个功能整合到一个

状态机中，图 17 所示为使用了 GF1.0 框架之后的结构。

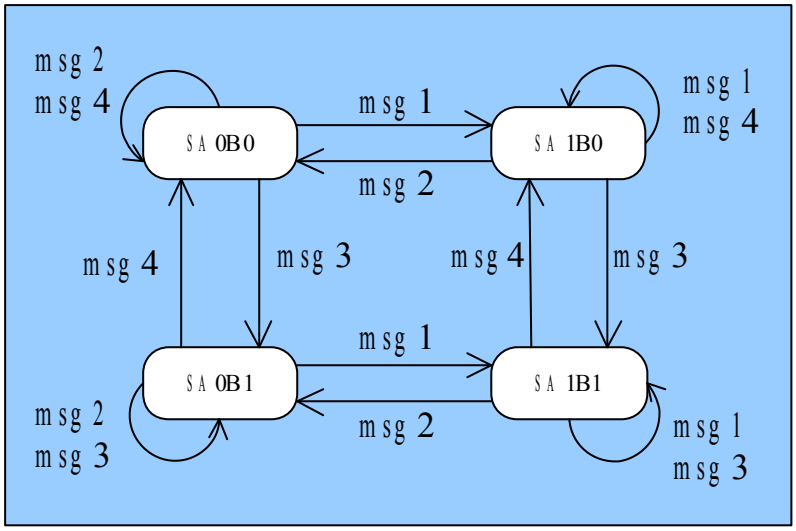

图 17 功能 A 和功能 B 合并处理示意图

从图 17 中可以看到，功能整合之后，我们必须使用 2X2=4 个状态才能完整描述系统所有的工作状态，而且在每个状态下我们需要同时考虑 msg1、msg2、msg3、msg4 这 4 个消息对当前状态的影响。

如果功能 A 有 m 个工作状态，i 个驱动消息，功能 B 有 n 个工作状态，j 个驱动消息，那么图 17 中就会有(m * n)个状态，每个状态都必须考虑(i + j)个消息对系统状态的影响，这给系统分析和代码实现带来了极大的困难。分析其中的原因不难发现，图 17 中的方法人为的把两个完全独立的功能整合到了一起，使系统功能之间的耦合度增加，造成了系统复杂度显著上升。

对比图 17，图 16 中的处理方式就要合理得多了，把两个功能 A 和功能 B 分开处理，互不关联，这样功能 A 在实现的时候就完全不用理会功能 B 对自己的影响，对于 msg3 和 msg4 这 2 个对自己没有意义的消息也完全不用考虑，只需要关注 msg1 和 msg2 就可以了，对功能 B 来将道理也是一样的。

图 16 这种并行处理的方式使无关联的各个功能之间完全独立开来，每个功能实现的时候只关注对自身有影响的那一小部分内容，所有功能实现以后再把它们拼接起来，就完成了整个系统的实现，这给系统设计带来了极大的简化，代码实现上也容易得多。这种处理方式体现了高内聚低耦合的设计思想，是裸奔框架中将单一状态机改进为并行状态机的思想基础。

(2) GF1.0 消息驱动机制的不足

再来看看 GF1.0 的消息驱动机制，GF1.0 的消息传递的过程是 ISR —> 消息缓冲队列 —> 状态机。ISR 位于单片机程序的最底层，直接面对的是硬件寄存器，接收的输入全部是最原始的信息，例如一个 I/O 管脚的电平变化、串口外设刚接收到的单字节数据、一次定时中断等等；接收消息的状态机位于应用层，负责一些高级的逻辑运算或算数运算，如行为决策等，工作方式比较接近人类的理解方式，所以状态机接收的输入趋向于抽象化的信息。

ISR 所提供的原始输入信息要想达到应用层程序对输入信息抽象化的要求，中间必须有一个对原始输入信息加工提炼的过程。在 GF1.0 的结构中，ISR 生成的消息就是输入信息的载体，由于 ISR 生成的消息是直接发给位于应用层的状态机的，所以原始输入的抽象化需要由 ISR 和应用层状态机中的一方独立承担或者双方共同承担。

以串口通信举一个例子，假设两个单片机系统通过 485 总线进行点对点数据通信，传送数据打包成如图 12 所示的数据帧，通信协议还包含数据校验、校验出错控制、帧超时控制等。根据第 4 章第 3 节有关串口 ISR 的分析，在 GF1.0 中实现这个完整的通信协议需要串口 ISR 和应用层状态机共同完成。由于数据校验比较耗费时间，串口接收 ISR 只负责数据帧的接收过程，可以设计成如图 13 所示的状态机，后续的工作如数据校验、校验出错控制、帧超时控制等只能交给应用层状态机来完成。

在这个例子中，串口 ISR 收到的一个个字节数据就是原始输入信息，而没有错误的数据帧中包含的数据则是应用层状态机真正想要的抽象化的输入信息。

可以肯定的是，数据帧接收完毕以后的处理过程是比较繁琐的，校验出错控制、帧超时控制等过程会给应用层状态机引入多个只与串口通信过程控制相关的状态。我们需要注意的是，在一个完整的单片机系统中，数据通信仅仅是系统功能的一部分，很多情况下甚至仅仅是很小的一部分。根据上一小节可知，GF1.0 想要实现串口通信控制，必须将串口通信相关的状态和其他功能相关的状态耦合在一起，这必然会使应用层状态机变得更加复杂。

从功能划分上分析，数据通信协议的实现只是一个很底层的功能，不应该让应用层的程序来负责。通信过程控制一方面太耗费时间，ISR 不能管，另一方面它又是一个底层功能，应用层不应该管，GF1.0 把它交给应用层只是不得已而为之，这是 GF1.0 面对此类问题暴露出来的一个缺陷。

GF1.0 出现这样的问题，症结就在于它的消息驱动机制，它使应用层状态机直接面对 ISR，ISR 提供的原始输入信息和应用层状态机抽象化的输入需求是“不对口”的。解决这种矛盾的方法就是在 ISR 和应用层之间再加入一个中间环节，这个中间环节负责接收 ISR 提供的原始输入信息，对其加工提炼，生成符合应用层程序要求的抽象化输入信息，再提交给应用层状态机。

对于上面串口通信的例子，我们可以设计一个通信协议控制器，加入控制器之后的功能结构如图 18 所示。

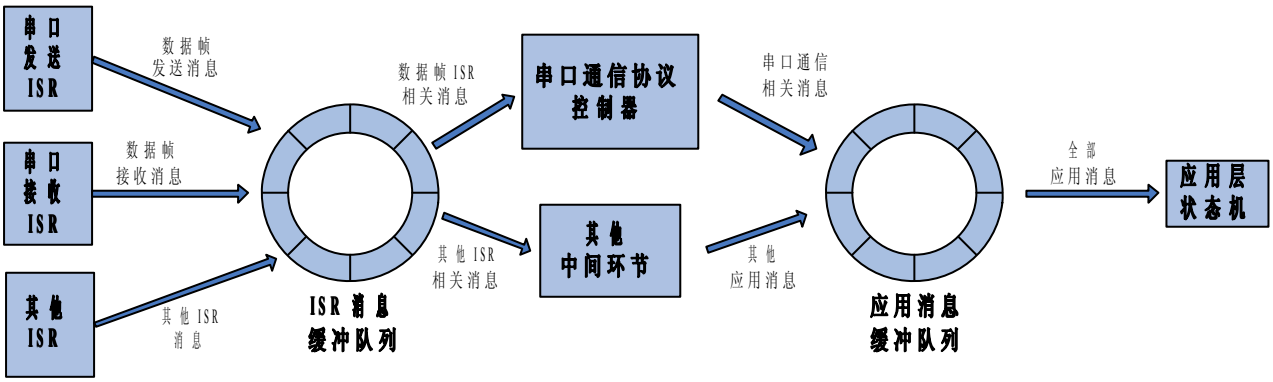

图 18 添加通信协议控制器之后的功能结构图

有了通信协议控制器，我们就可以将通信过程控制从应用层状态机中剥离出来，交给控制器来完成，控制器接收 ISR 发来的数据帧消息，完成后续的工作如数据校验、校验出错控制、帧超时控制。控制器只会将正确接收的数据帧提交应用层状态机，而且提交之前还会剥离无用的信息，出错的数据帧直接在控制器这一环节就被拦截了，不会影响到应用层状态机。

图 18 中使用了两个消息缓冲队列：中断消息缓冲队列和应用消息缓冲队列。中断消息缓冲队列负责收集各个 ISR 生成的消息，同时也是各个中间环节的消息源，应用消息缓冲队列负责收集由各个中间环节生成的消息，同时也是应用层状态机的消息源。我把这样的消息传递方式称作双消息驱动机制。

双消息驱动机制体现的是一种程序分层的思想，加入了中间环节之后，程序的各部分层次分明，各司其职，程序的结构更加趋于完美。相比于单消息驱动机制，双消息驱动机制还能更好的支持并行状态机，并行状态机加上双消息驱动，构成了全篇的核心——升级版的裸奔通用程序框架 GF2.0。

2、裸奔通用框架的升级版 GF2.0

经过上一小节的讨论，我们可以发现 GF2.0 的 2 个基本特征：并行状态机和双消息驱动机制。

应用层程序使用并行状态机是为了拆分系统功能，让每一个状态机去负责一项功能，各状态机之间相互独立，必要时通过 GF2.0 提供的消息功能彼此通信。相比 GF1.0 中“一机多能”的主状态机，并行状态机“专机专用”，极大的简化了应用层的设计，使程序结构更容易理解，也使程序的性能更加稳定可靠。

双消息驱动机制的引入最主要的目的是更好的调理外界对单片机系统的输入，在检测输入的底层 ISR 和应用层状态机之间加上一个中间环节，这个中间环节就相当于应用层状态机的专属厨师，将初级食材(ISR 生成的消息，代表系统输入的原始形态)烹制成符合应用层状态机口味的佳肴(抽象化之后的系统输入)。双消息驱动机制的另一个好处是使程序分层更加细致合理，各层职责更加明确，还在一定程度上提高了程序的实时性，这一点在后续的章节中会有说明。

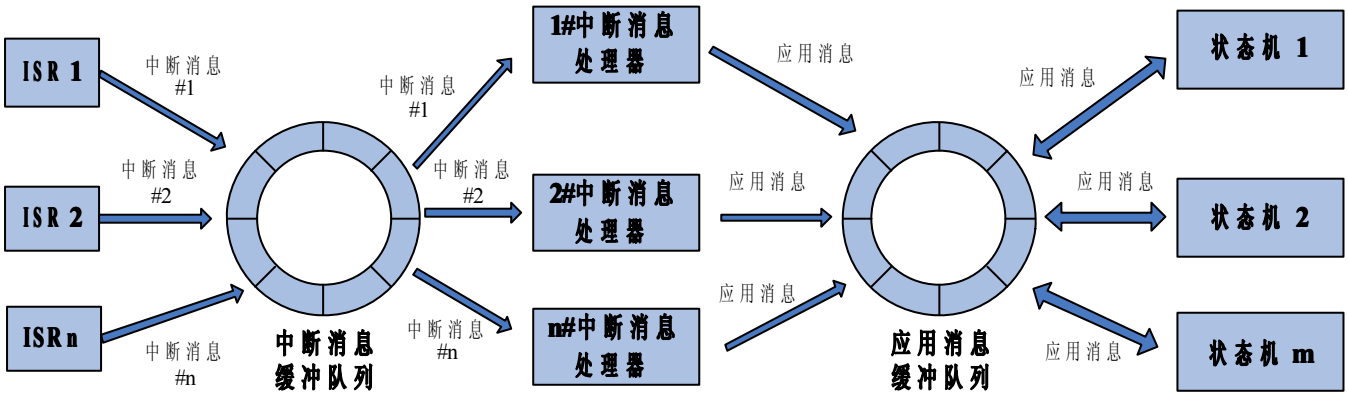

图 19 GF2.0 原理示意图

图 19 所示就是 GF2.0 的原理示意图，GF2.0 的各个关键部件在图中一目了然，，通过前面

小节的叙述，这个原理示意图还是很容易理解的。关于这张图，我还有 2 点需要说明：

1) 消息缓冲队列的复用

从图中可以看出，各个 ISR 都把生成的中断消息发送到同一个中断消息缓冲队列，也就是说中断消息缓冲队列是共用的。

理想情况下，每个生成消息的 ISR 都应该有独立的缓冲队列，这个缓冲队列只保存该 ISR 生成的消息，和这些中断消息相关的消息处理器都挂载在这个缓冲队列上。

缓冲队列的使用是要耗费 RAM 资源的，而 RAM 恰恰是单片机中比较珍贵稀缺的资源，如果每个 ISR 都要配备一个独立的缓冲队列，这样在 ISR 较多的情况下会占用大量的 RAM，这是很多单片机系统所不能接受的。另一方面，在中断事件不是很集中的时候，中断缓冲队列几乎是空闲的，这样闲置大量的 RAM 在单片机系统中确实是不明智的，而让所有的 ISR 共用一个中断消息缓冲队列就能极大的节省 RAM 的使用，同时还使中断消息的管理变得更加简单。

也正是因为同样的原因，应用消息缓冲队列也是由应用层的各个状态机共用的。

2) 应用消息缓冲队列和应用层状态机之间的双向箭头

从图中可以发现，消息的传递在 GF2.0 中很多环节都是单向的，唯独应用消息缓冲队列和每个个状态机之间的传递是双向的，这说明应用层状态机不但可以从应用消息缓冲队列中读取消息，还可以向应用消息缓冲队列发送消息。

在有些情况下，可能需要多个状态机共同合作来完成一项工作，这样的话就需要这些状态机之间可以相互通信，但是状态机之间是没办法通过诸如共享全局变量这样的方式直接通信的，能唤醒状态机的只有那些能驱动它工作的消息，所以两个状态机之间想要通信，只能将数据以消息的形式通过应用消息缓冲队列来中转。

了解了 GF2.0 的结构，再来看看这样的结构在程序中是怎么工作的，图 20 就是 GF2.0 在程序中的主循环流程图，包括 3 个循环：中断消息处理循环、应用消息处理循环以及空闲循环。

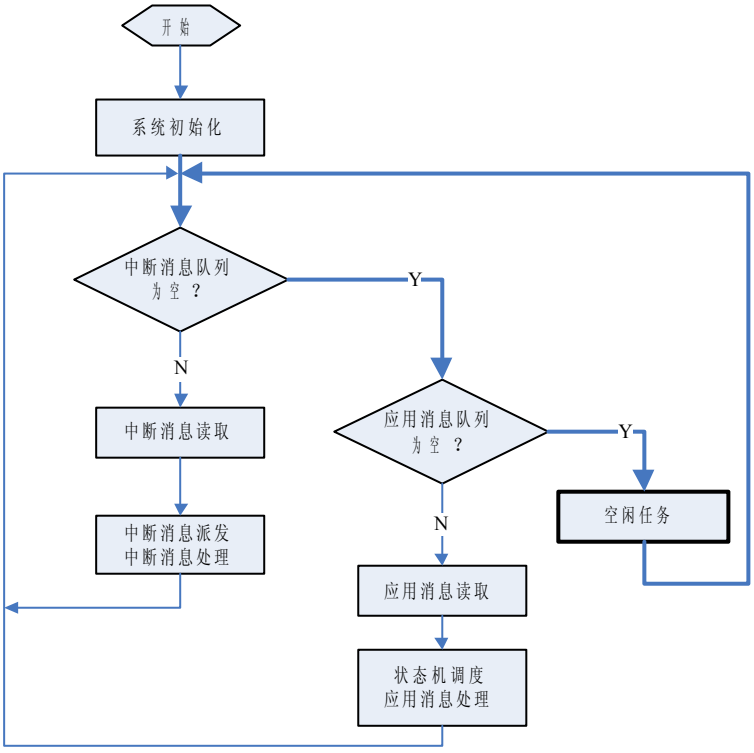

图 20　GF2.0 的主循环流程图

中断消息处理循环负责处理中断消息缓冲队列中积压的中断消息。中断发生后，如果 ISR 需要通知主程序进行中断事件的后续处理，则会生成一条中断消息并将其以先入先出的方式存入中断消息缓冲队列。得知中断消息缓冲队列中有消息，程序就会从队列对头摘取一条中断消息并根据消息类型派发给对应的中断消息处理器处理。在中断消息处理期间，中断消息处理器可能(确实是可能)会生成一条或多条应用消息，这些消息会以先入先出的方式存入应用消息缓冲队列。

如果当前的中断消息全部处理完了，中断消息处理循环中止，继而执行应用消息处理循环。如果应用消息队列不为空，程序从应用消息缓冲队列对头摘取一条应用消息，派发给消息指明的目的状态机，驱动状态机进行动作响应、状态迁移，这一过程叫做状态机的调度，也是应用消息的处理过程。

如果当前既没有中断消息也没有应用消息，程序会去执行空闲任务，空闲循环在图 20 中用粗体线标出。和 GF1.0 的主循环流程图(图 11)一样，空闲任务是流程图中最繁忙的部分，关于空闲循环前文已有说明，在此不多言。

仔细观察流程图会发现，中断消息处理循环是套在应用消息处理循环里面的，每次应用消息处理循环处理完一条应用消息都会重新检查中断消息缓冲队列是否为空，如果不为空就会再次进入中断消息处理循环。这也就是说，中断消息的优先级比应用消息要高，程序总是优先处理中断消息。应用消息是从中断消息中衍生出来的，从时间紧迫程度上来说中断消息当然要比应用消息更紧迫，所以这样的循环嵌套是合理的。需要说明的是，虽然中断消息比应用消息要优先执行，但两者的处理过程都位于主程序，是“串行”的，在某条应用消息处理的过程中，即使出现了新的中断消息，此次应用消息处理过程也不会被抢占，只有这条应用消息处理完毕，程序才会去理会新出现的中断消息。

3、双消息驱动机制在 GF2.0 中的实现

(1) 中断消息的结构和管理

关于 GF2.0 的中断消息，包含 4 个方面的内容：中断消息的结构、中断消息缓冲队列的数据结构、中断消息的管理函数、中断消息的处理。

在本文的第三章第 3 节(事件驱动与消息)的内容中，我们讨论过通用消息的实现方法，那里有关消息的结构、缓冲队列数据结构以及管理函数的实现方法几乎可以原封不动地照搬过来。但是由于中断消息的特殊性，在这里还要做几点说明：

a) 中断消息的结构

程序清单 List10 是通用消息的数据结构，GF2.0 的中断消息可以效仿，但是消息结构体 MSG 中的成员 u8UsrID(消费者 ID)可以去掉，u8UsrID 代表的是接收消息的应用层状态机标号，但是 GF2.0 的中断消息不需要交给应用层的状态机处理，所以这个成员在中断消息中可以去掉。

b) 中断消息的形参

在程序清单 List10 中，联合体 MSG_ARG 是消息 MSG 的参数部分，它使消息可以传递多种类型的消息参数。成员 void* pArg 可以传递指向一段内存的指针，而这段内存中是消息真正

想要传递的数据，这可以实现类似于 RTOS 中消息邮箱的功能。但是不同于 RTOS 中的消息邮箱，GF2.0 的中断消息不能传递指向自动局部变量的指针，也就是说成员 void* pArg 所指向的变量必须是静态局部变量或者全局变量。

在 C 语言中，一个函数调用返回以后，函数调用时使用的自动局部变量便消失了，系统会将临时分配给自动局部变量的内存回收。假设某个 ISR 要发送中断消息传递一个数据类型为 INT32U(32 位无符号整型数据)的数据且该数据保存在 ISR 的自动局部变量中，如果使用联合体 MSG_ARG 的成员 u32Arg 传递，中断消息会以副本的形式将数据保存在消息实体中，这样即使保存原始数据的那个自动局部变量消失，也不会对数据的传递产生影响。如果用指针的形式传递数据，即将自动局部变量的地址保存在 pArg 中，ISR 返回后自动局部变量的内存就被回收了，这样当程序执行到处理这条中断消息的代码时，pArg 指向的那段内存极有可能此前被分配给其他函数里的自动局部变量使用了，虽然 pArg 指向的内存地址没有变，但是这段内存中的内容却被改过了。

中断消息能直接传递的数据只能是基本数据类型的数据，想要传递结构体或者内存块只能使用传递指针的方法间接实现。根据上段的分析，如果要保证数据的正确传递，这些变量必须被声明为全局变量或者静态局部变量。

c) 中断消息缓冲队列的接口函数

第三章第 3 节(事件驱动与消息)列举了消息缓冲队列的接口函数，共有 9 个，对于 GF2.0 的中断消息是完全适用的，但中断消息对某些接口函数的实现还是有特殊要求的。

INT8U mq_msg_post_fifo(MSG* pMsg)

函数 mq_msg_post_fifo()负责向中断消息队列中寄送中断消息，在 GF2.0 中只供 ISR 调用。单片机系统中一般会有多个中断，为减少中断响应延迟往往还会允许中断嵌套。当低级中断 ISR 调用 mq_msg_post_fifo()时发生高级中断事件，高级中断 ISR 就会抢占低级中断 ISR，为保护低级 ISR 的断点不被高级 ISR 破坏，函数 mq_msg_post_fifo()必须是可重入的。另外，当函数 mq_msg_post_fifo()访问与中断消息缓冲队列相关的全局变量的时候，访问必须是互斥的，所以要对访问代码进行临界保护，即访问前关中断访问后开中断。

INT8U mq_msg_req_fifo(MSG* pMsg)

函数 mq_msg_req_fifo()在 GF2.0 中只由主程序调用，负责从中断消息缓冲队列中读取中断消息。由于 ISR 不会调用 mq_msg_req_fifo()，所以 mq_msg_req_fifo()可以不是可重入函数，但是函数 mq_msg_req_fifo()会访问与中断消息缓冲队列相关的全局变量，所以也要对函数中的访问代码进行临界保护。

d) 中断消息的处理

在系统中一般会存在多种中断消息，每个中断消息代表着不同的中断事件，程序根据中断消息结构体中标记消息类型的成员(参照程序清单 List10 中 MSG.u8MsgID)来识别中断消息。在第五章第 2 节我们谈到过双消息驱动机制的中间环节，这个中间环节作为中断消息和应用消息的衔接，主要的任务就是处理中断消息，我把这个中间环节叫做中断消息处理器。

中断消息处理器其实就是一个带参数的消息处理函数，形式如下：

void int_msg_processor(INT_MSG* pIntMsg)

形参指针 pIntMsg 指向存储待处理中断消息的内存，函数 int_msg_processor()根据形参指针 pIntMsg 获取消息实体的具体内容。不同的中断消息对应着不同的响应方式，即对应着不同的消息处理函数，所以中断消息的处理分为 2 个步骤：根据中断消息类型定位中断消息处理器、调用中断消息处理器响应中断消息。

定位消息处理函数最简单的方法就是使用 switch—case 语句，将所有的中断消息处理函数用 case 语句组织起来，定位函数的时候，将中断消息按消息类型和预先定义好的消息类型逐个比较即可。这种遍历方式虽然容易实现，但是程序执行效率不高，中断消息种类越多，查找耗时越多，加大了中断消息的响应延迟，而且也不能提炼出标准框架代码，在 GF2.0 中不推荐使用。

定位处理函数效率最高的方法就是查表法，把中断消息的消息类型声明成以 0 为起始值的枚举变量，然后将所有中断消息处理函数的函数地址按消息类型值递增的顺序放在一个函数指针数组中，定位处理函数时，只需要以中断消息的消息类型为数组下标，执行一次查表操作就能找到对应的中断消息处理函数。当然，为避免指针调用函数的时候程序跑飞，必须防止数组寻址越界，查表前需要对消息类型进行合法性检查。

图 21 所示为查表法的数据结构示意图，向量表中总共有 5 种中断消息的消息类型。

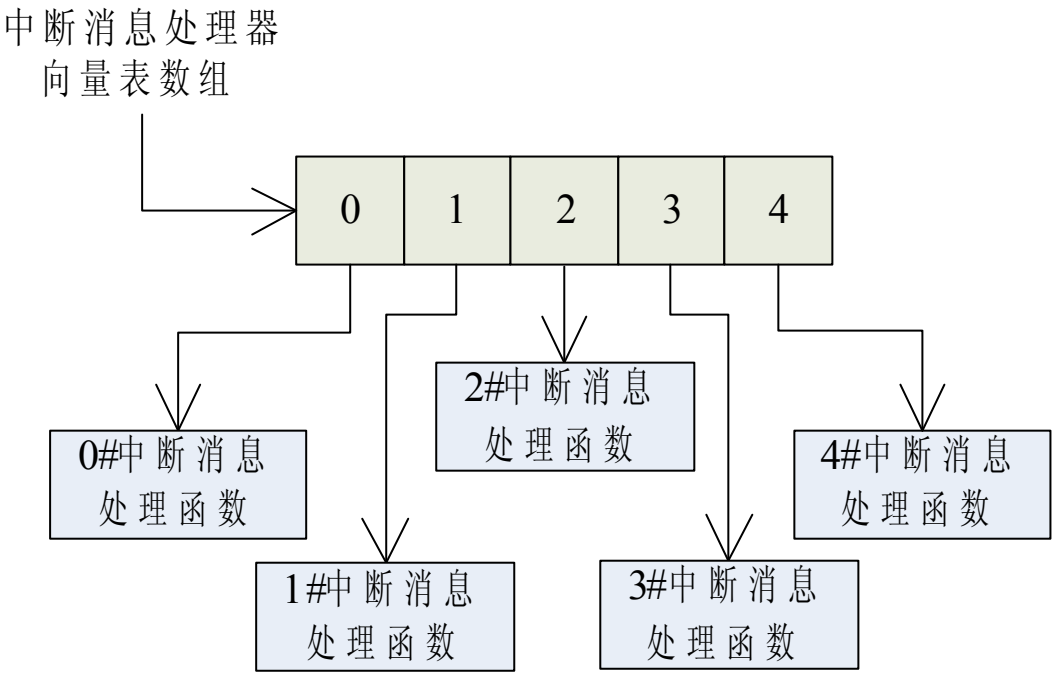

图 21 查表法定位中断消息处理函数

图 22 所示为中断消息从被读取到被处理大致过程的流程图。

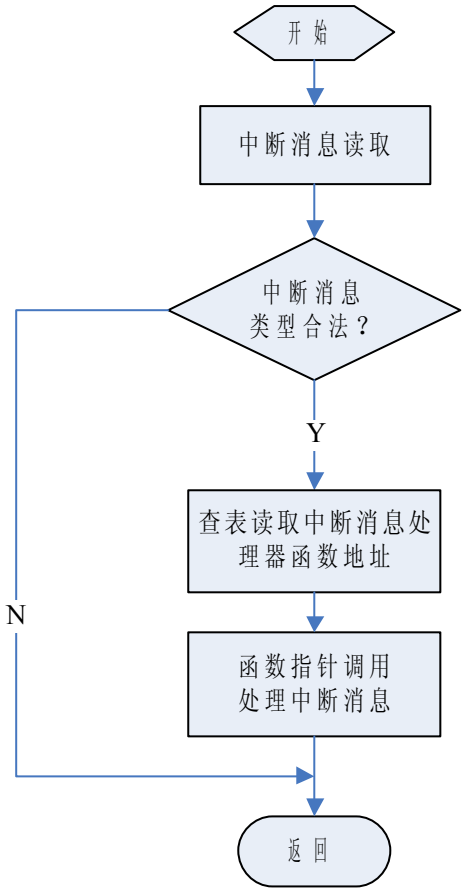

图 22 GF2.0 中断消息处理流程图

图 22 可以写成标准的框架代码，如程序清单 List12 所示。

程序清单 List12：

```
typedef void (*IMP)(INT_MSG* pIntMsg); /*指向中断消息处理器的指针类型*/
extern IMP g_arIMPTbl[];                /*中断消息处理函数向量表*/

/*****************************************
*FuncName    : gf20_int_msg_engine
*Description :GF2.0 中断消息处理示例函数
*Arguments   : void
*Return      : void
*History     : [2011-10-13]
*                 创建；
*****************************************/
void gf20_int_msg_engine(void)
{
    IMP       pfAction = NULL;          /*消息处理器函数指针*/
    INT_MSG stIntMsg;                   /*中断消息暂存*/

    memset(                             /*变量初始化*/
```

```
        (void*)(&stIntMsg),
                0                       ,
                sizeof(INT_MSG)    );

        mq_msg_req_fifo(&stIntMsg);     /*从缓冲队列读取消息*/

        if(stIntMsg.u8MsgID < INT_MSG_ID_MAX)     /*消息类型合法？*/
        {
            pfAction = g_arIMPTbl[stIntMsg.u8MsgID]; /*定位中断消息处理函数*/
            pfAction(&stIntMsg);                     /*函数调用，消息处理*/
        }
    }
```

关于中断消息处理函数向量表 g_arIMPTbl[]，多说两句。在 GF2.0 正式开始运行前，必须将 g_arIMPTbl[]提前配置好，如果程序启动后中断消息对应的处理函数不需要调整，可以将这个向量表放置在 ROM 中以节省 RAM，但是向量表的填写必须在源代码编译之前完成。如果希望向量表内容可以在程序运行的时候动态调整，g_arIMPTbl[]应该放置在 RAM 中，此外 GF2.0 还应该实现一个可以修改 g_arIMPTbl[]数组元素的接口函数

INT8U gf20_imp_resign(IMP pNewAction , INT_MSG_TYP eIntMsgId)

函数 gf20_imp_resign()的功能是重置某中断消息的消息处理函数，该中断消息的消息类型由形参 eIntMsgId 指明(INT_MSG_TYP 为中断消息的枚举数据类型)，新的消息处理函数函数地址由形参 pNewAction 指明，函数返回操作结果。函数的主要工作如下

g_arIMPTbl[eIntMsgId] = pNewAction ;

下面再聊一下中断消息处理器——中断消息处理函数。

中断消息处理器的出现相当于在底层 ISR 和应用层并行状态机之间加了一个隔离层，使程序层次性变得更好，如果应用层的结构需要变更，重新规划了状态机，则只需要在中断消息处理函数里做相应的改动即可，底层 ISR 不需要做任何改动，反之亦然。

在事件/消息驱动+状态机这种程序结构中，中断消息处理器的出现还使程序对中断事件的响应实时性变得更好，如果中断事件的响应动作不适合放在 ISR 中但响应延迟又不宜太长，可以把中断事件的响应放在中断消息处理器中实现，不必“惊动”应用层的状态机（分层的程序结构中越往上反应越迟钝）。

根据中断事件的不同特点，中断消息处理器分为 2 种：直通型和状态机型。

直通型：　中断消息处理器将中断消息中的数据稍微加工一下，直接生成应用消息发给应用层状态机。在这种消息处理器中，一条中断消息能生成一条或多条应用消息，这种消息处理器函数实现比较简单，实际应用中也比较多，例如 A/D 采样事件、系统时钟节拍事件、定时中断事件、外部中断事件等。

状态机型：状态机型的中断消息处理器实质上是一个状态机，包含若干个工作状态，状态机在中断消息的驱动下发生状态迁移和动作响应，但并不是每次状态迁移都会

生成应用消息，只有中断消息触发了某些特殊的状态迁移，中断消息处理器才会生成应用消息。状态机型中断消息处理器要比直通型功能强大，能实现比较复杂的消息处理。在第五章第 1 节举过一个串口通信的例子，图 18 中的那个串口通信协议控制器就是一个状态机型的中断消息处理器。

对一次中断事件的响应，是 ISR、中断消息处理器以及应用层状态机 3 方通力合作的结果，它们在事件响应中的职责也并不是界限分明的，各自承担工作比例的多少主要根据事件响应在时间维度上的需求进行权衡，同时还要考虑此类事件所包含的外界输入的信息量的多少。可能此次响应不需要惊动应用层，只需要 ISR 和中断消息处理器(甚至连它也不必操心)就能搞定了，也可能用户想使用按键操作启动单片机系统某项复杂的功能，虽然按键事件比较简单，但这次恐怕不惊动应用层状态机是不行了。

(2) 应用消息的结构和管理

和中断消息类似，GF2.0 的应用消息同样包含 4 个方面的内容：应用消息的结构、应用消息缓冲队列的数据结构、应用消息的管理函数、应用消息的处理。除应用消息的处理外，其他部分内容要比中断消息简略许多。

a）应用消息的结构

程序清单 List10 所给出的通用消息的数据结构对 GF2.0 的应用消息完全适用，不需要做任何改动。

b) 应用消息的形参

需要注意的问题可完全参考中断消息。

c) 应用消息缓冲队列的接口函数

第三章第 3 节(事件驱动与消息)列举的通用消息缓冲队列的 9 个接口函数，对于 GF2.0 的应用消息也是完全适用的，对其中 2 个接口函数在此做一下说明。

```
INT8U mq_msg_post_fifo(MSG* pMsg)
INT8U mq_msg_req_fifo(MSG* pMsg)
```

应用消息的接口函数 mq_msg_post_fifo()和 mq_msg_req_fifo()在 GF2.0 中只供主程序调用。所以这两个函数可以不是可重入的，访问应用消息缓冲队列时也不用对代码进行临界保护。

d) 应用消息的处理

GF2.0 中的应用消息消费者是应用层并行状态机，所以应用消息是由这些并行状态机来处理的，这部分内容涉及并行状态机的实现和并行状态机的调度，有关应用消息的处理在本章第 4 节中详述。

4、并行状态机在 GF2.0 中的实现

在程序设计的时候，一般来说会存在一个叫做正交化设计的设计原则。

先介绍一个概念——正交域，系统中相互独立且没有交集的部分称作正交域。假设现在有一个系统如图 23 所示，系统的整体功能可以细分出 4 个子功能：A、B、C、D 。A 和 B 存在一定的关联，功能上相互制约，可能彼此需要协调合作才能正常工作，C 和 D 之间是没有什么关联的，即使 C 出现了故障也不会对 D 造成任何影响，反之亦然。由正交域的概念可知，图 23 中存在 3 个正交域：AB、C、D。在设计程序实现图 23 所示的功能时，就可以按照正交化设计的原则将这 3 个正交的模块分别独立实现。

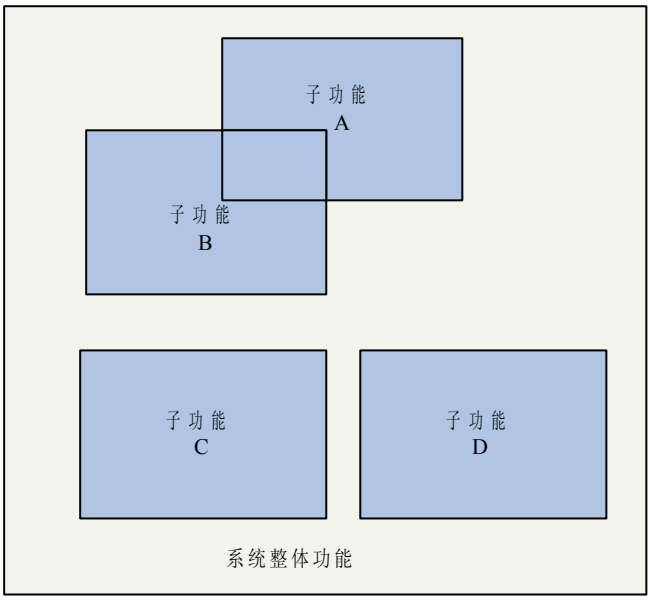

图 23 正交域和非正交域示意图

在第五章第 1 节(GF1.0 的不足)也举过一个正交域的例子，图 16 中 A 和 B 就是两个正交域。正交化是简化软件设计的重要手段，对于状态机编程同样适用。如果一个系统模型可划分为若干个正交域，每个正交域均可用一个状态机来实现，这些状态机各自包含的状态数目分别是 k、l、m、n…… ，正交化之后我们可以使用（k + l + m + n + ……）个状态来描述系统，但是如果系统只使用一个状态机来描述，那么这个状态机则要包含（k × l × m × n × ……）个状态才能完整描述整个系统。

既然正交域之间是相互独立的，那么所有的正交域在时间维度上就可以齐头并进，把正交域状态机化，这些状态机就可以并发执行，这就是我把这些状态机称作并行状态机的原因。

(1) 并行状态机的多任务属性

在状态机理论中有这样一个重要的约定：任何一个状态机对任何事件的响应都是串行的。这也就是说，状态机只有在对一个事件的响应动作全部完成以后，才能去处理下一个事件(如果不加特殊说明，对事件和消息不加区分，事件就是消息，消息就是事件)，状态机理论把状态机的这种特性称作 run-to-completion，简称 RTC 原则。

没有事件要处理的时候，状态机处于“空闲状态”，总是无限期的等待属于它自己的事件发生，等到新事件出现，状态机识别事件后进入“忙碌状态”，开始处理事件，动作响应，状态迁移，完成对该事件的响应，响应完毕后状态机又迅速回到“空闲状态”，期待下一次事件的发生。一个完整的、时间上连续的“忙碌状态”称作 RTC-step，根据 RTC 原则，RTC-step 是状态机的响应盲区，此时状态机无法响应新的事件，这时只能把新事件暂存在缓冲队列中，等到状态机从 RTC-step 中退出后再将缓存事件读出，新事件得到响应，整个过程如图 24 所示。

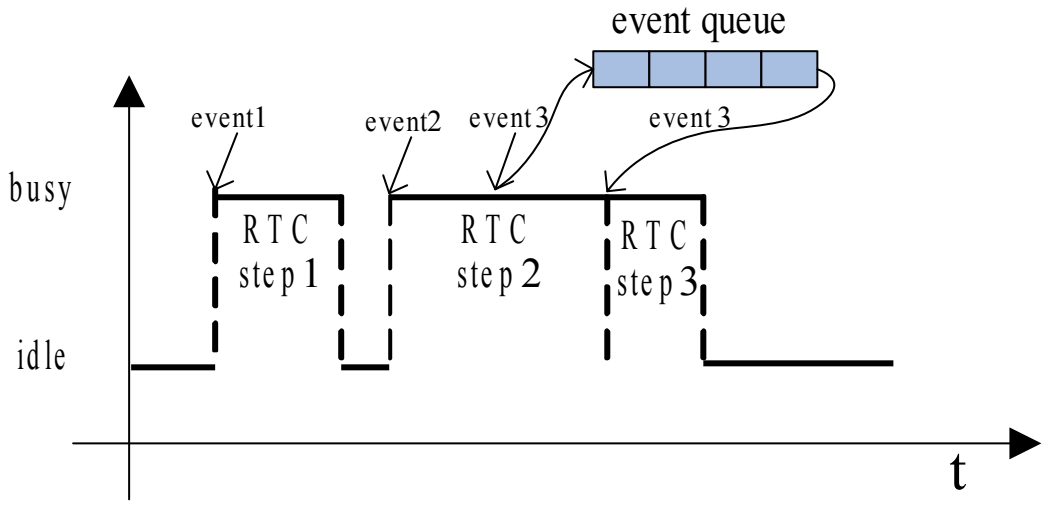

图 24 RTC 原则示意图

每个 RTC-step 都会耗费 CPU 时间，所以状态机对事件的响应有可能存在延迟，最坏情况下，响应延迟的总时间

$$T_{wait\max} = T_{step\max} * (N+1)$$

上式中，$T_{step\max}$ 是所有的状态机中最长 RTC-step 所耗费的时间，N 是事件缓冲队列的最大容量。如果最坏的响应延迟过长，以至于超过了事件所容许的最长等待时间，说明状态机状态划分不合理，需要对最长 RTC-step 所对应的状态再进行拆分，直至满足要求。

在 GF2.0 中每个并行状态机都独立承担一项系统功能，在属于自己的应用消息的驱动下工作，如果这些状态机的应用消息在时间上交错分布，那么这些状态机就会交替运行，微观上看这些状态机是顺序执行，但宏观上看它们是并发执行的，呈现出 RTOS 中多任务的特性，如图 25 所示(为突出主题，图中把 ISR、中断消息处理等过程均省略)，所以我又把并行状态机称作状态机任务。需要说明的是，把并行状态机称作“任务”只是一种形象化的比喻，状态机任务和 RTOS 中的任务是有很大差别的，我们做一下大体的比较。

a) 任务的结构

在 RTOS 中，任务主要由 3 部分组成：任务代码、任务堆栈、任务控制块。在 GF2.0 中状态机任务主要由 2 部分组成：状态机代码、状态存储全局变量。

在资源消耗方面，任务代码和状态机代码都位于 ROM 中，没有多少差别，但任务堆栈和

任务控制块要占用大量的 RAM 资源，而状态存储全局变量只占用几个字节的 RAM，差别明显。但是正是因为任务堆栈和任务控制块的存在，RTOS 可以对任务实现精细和复杂的控制，相比之下仅凭一个状态存储全局变量，GF2.0 对状态机只能实现比较粗糙的控制，所以性能上 RTOS 任务要明显好于并行状态机。

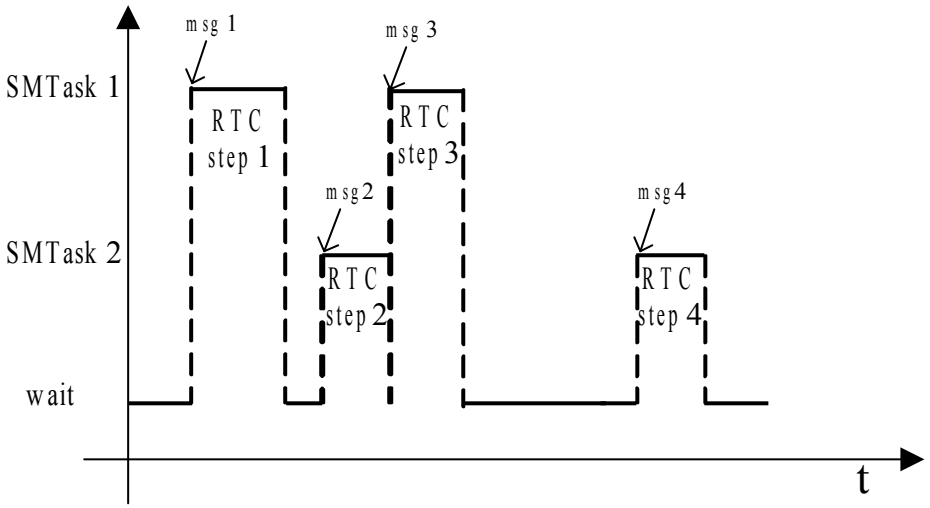

图 25 并行状态机并发执行示意图

b) 任务的切换方式

在 RTOS 中，如果发生了中断事件或者任务代码调用了系统服务，就可能会引发任务切换。任务切换是 RTOS 内核按照调度规则强制完成的，内核是主宰者，而任务自身处于被动的位置，不管是任务切入还是切出，CPU 的使用权是被动给予被动收回的，所以对任务来说任务切换是不可知不可控的。

相比之下，状态机任务切换时状态机自身的主动性要大得多。没有事件的时候，状态机处于休眠状态，等待事件，事件发生后，对应的状态机不会立即进入活动状态，只能由管理程序将事件分配给状态机然后再调用状态机代码，将 CPU 的使用权分配给状态机。由此可知，任务切入的时候状态机是被动的，对状态机来说任务切入是不可知不可控的。

根据状态机 RTC 原则可知，一个状态机响应事件时不能响应新的事件，更不能让其他状态机响应事件，所以一旦状态机获得了 CPU 使用权，状态机就成了 CPU 的主宰者（不考虑 ISR）。事件响应完成以后，状态机代码函数返回，返回到状态机管理程序，将 CPU 使用权主动让出，状态机自身也由活动状态进入休眠状态，继续等待事件。因此状态机任务切出是状态机主动完成的，任务切出是可知可控的。

c) 任务的断点

从逻辑上看，RTOS 任务就好像源源不断的水流，内核可以随时随地地“抽刀断水”。如果有需要，内核可以在任务代码的任何地方(不考虑临界段)中断任务的运行，将任务切出，当然这样做的代价就是要使用足够的RAM来保存任务的断点，各级函数调用的返回地址、PC、PSW、所有通用寄存器，这些数据全部要备份在任务堆栈中。

如果说 RTOS 任务是连续的，那么状态机任务则是断续的，从逻辑上看，一份完整的状态

机程序是由不同的程序片段拼接而成的，这些程序片段就是前文提到的 RTC-step。RTC-step 不可分割，所以状态机任务的切换只能发生在 RTC-step 的边界，也就是说只有在等待事件的时候状态机才能被切换。如果把状态机的状态转换图比作一张网，RTC-step 就是网格上的线段，而线段的交汇点就是状态机的状态，所以状态机任务断点的可能位置是人为安排的，总是位于事件响应函数返回的地方，想要保存断点只需要用一个全局变量保存状态机当前的状态即可。

(2) 并行状态机的调度

在 RTOS 中，任务调度的本质是 CPU 使用权的回收和再分配，但是对于并行状态机的调度这个定义不完全适用，由于 CPU 的使用权是状态机任务主动让出的，所以状态机任务调度仅仅是 CPU 使用权的再分配。

如果当前没有事件发生，所有的状态机都处在休眠状态，都没有运行的欲望，不会有对 CPU 使用权的需求，也就没有调度的必要。某时刻发生了一个事件，这个事件需要由并行状态机中的某个状态机响应，从事件发生到事件响应会有一个延迟，在这段延迟时间里，目标状态机对 CPU 使用权产生需求，这种需求最终会启动状态机任务的调度机制。

如果当前只有一个状态机对 CPU 使用权有需求，严格地说启动这个状态机的过程算不上是状态机调度，因为有竞争才会有调度，只有当前有多个事件需要响应且这些事件的目标状态机不止一个，目标状态机之间对 CPU 使用权存在竞争关系，这才真正需要调度机制进行仲裁，以确定 CPU 使用权的最终归属。这两种情况的差别在这里点到为止，以后如果没有特别说明则不加区别。

事件驱动+状态机这种程序结构是一种以事件处理为中心的程序结构，所以事件才是真正的主角，事件要求哪个状态机来响应，调度机制就会选择哪个状态机，因此，并行状态机调度实际上是一种根据事件对响应的迫切程度来选择事件处理顺序的决策规则。这样说好像把这种调度称为事件调度更合理一些，但是因为调度机制调度的对象是并行状态机，所以我还是乐意把它称作状态机调度。

总而言之一句话：并行状态机调度以事件为调度依据，以并行状态机为调度对象。

我把能实现并行状态机调度功能的程序称作状态机调度器，这部分程序一般会封装成一个调度器函数。根据调度机制的不同，状态机调度器可分为两种：合作式调度器和抢占式调度器。

a) 合作式调度器

合作式调度器对所有未处理的事件一视同仁，先来先处理，晚来晚处理。由于事件在事件缓冲队列中的存储方式就是先入先出的，所以合作式调度器在算法上很容易实现，调度器只需要依次从缓冲队列读出事件，然后根据事件选择对应的状态机，直到将缓冲队列中的事件清空。

合作式调度器算法简单可靠，资源消耗少，执行速度快，非常实用，如果应用层并行状态机结构设计得合理，各个 RTC-step 充分得到优化，合作式调度器足以能应付大多数应用场合。

简单是合作式调度器的优点，同样也是合作式调度器的缺点。合作式调度器默认事件缓冲队列中的所有事件对响应的迫切程度是一致的，所以才会按事件发生的先后顺序依次调度，如果事件对响应的迫切程度不一致，则可能会出现事件响应超时的情况。

假设某时刻事件缓冲队列中共有 m 个事件，按发生时间先后顺序依次为 E1、E2……Em ，这些事件对应的 RTC-step 的执行时间分别为 T1、T2……Tm ，如果调度器为合作式调度器，那么事件 Em 的响应延迟时间

$$T_{wait\_sum} = \sum_{1 \le i \le (m-1)} T_i$$

如果事件 Em 恰好是这 m 个事件中对响应需求最迫切的事件，很可能会出现响应超时。

b) 抢占式调度器

抢占式调度在 RTOS 里司空见惯，随便一个能用的 RTOS 对任务的调度都是抢占式的。需要说明的是，RTOS 中任务抢占的概念和并行状态机抢占的概念是不一样的，RTOS 任务抢占是高优先级任务抢占低优先级任务，而并行状态机抢占是指高优先级事件抢占低优先级事件。

和合作式调度器正好相反，抢占式调度器认为事件对响应的迫切程度是不一致的。假设事件 A 和事件 B 先后发生且当前均暂存在事件缓冲队列中，虽然事件 A 先于事件 B 出现，但不见得事件 A 就应该先得到响应，如果事件 B 对响应的迫切程度高于事件 A，抢占式调度器会先安排并行状态机处理事件 B。如果两个事件对响应的迫切程度一致或者很接近，则按事件出现的先后顺序依次处理。

为了描述事件对响应的迫切程度，抢占式调度器要求每个事件都要附带优先级属性，优先级属性实际上是一个非负整数值，优先级越高，事件对响应的迫切程度越高，越有可能在抢占式调度器的调度下早一点被响应。

对于类型不同的事件，它们的优先级可以是相同的，相同的优先级表示这些事件对响应的迫切程度是一致的；事件的优先级可以不是固定的，一个事件实体(事件和事件实体之间的关系就好比人类这一事物和张三这一现实个体之间的关系)的优先级可以在事件发生的时候临时赋予，所以对于两个类型相同的事件实体来说，它们的优先级可能是不同的。在抢占式调度机制下，事件优先级的动态调整可以使事件的响应更加灵活，为了实现优先级动态调整，在描述事件的结构体中可以用一个 8 位无符号整形变量来记录事件的优先级。

有了优先级，抢占式调度器的调度规则可以用下面两句话来描述：

1) 优先级不同的情况下，高优先级事件要先于低优先级事件得到响应，不考虑事件发生的先后顺序。
2) 优先级相同的情况下，先发生的事件要先于后发生的事件得到响应。

抢占式调度器的调度算法可以分为如下 2 个步骤：

1）对当前事件缓冲队列中的事件进行排序，优先级越高且发生时间越早的事件越靠前，也就是越靠近队列队头的位置。
2）完成步骤 1 之后，按合作式调度器的调度方式调度并行状态机。

由此可见，抢占式调度算法实现的重点和难点是第 1 步，第 2 步直接就是合作式调度器的

实现方法，因此接下来只讨论第 1 步如何实现。

由于事件进入事件缓冲队列的方式是先入先出的，所以在抢占式调度器开始之前，我们已经得到了一个按发生顺序排列的事件队列，因此第 1 步实现的关键就是在不打乱同优先级事件发生顺序的前提下将事件按优先级排序，图 26 所示为 8 个事件的排序示例。

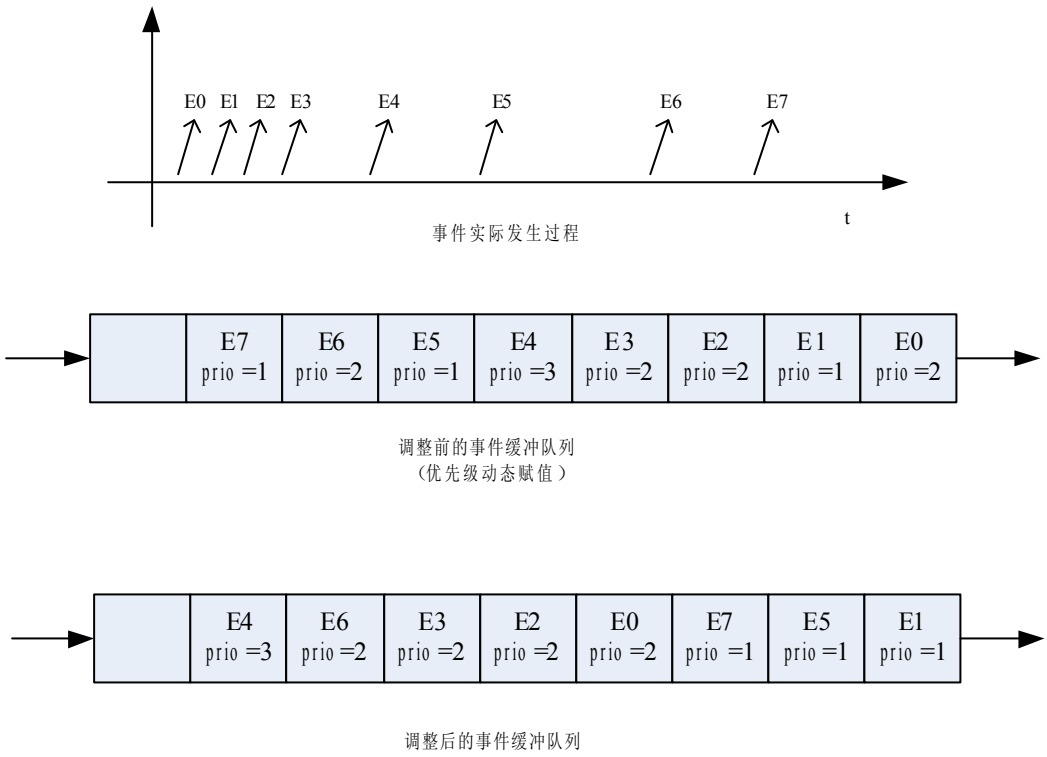

图 26 抢占式调度事件排序示例图

图 26 中规定，事件优先级属性数值越小事件优先级越高。图例假设短时间内连续发生了 8 个事件且均未得到响应，每个事件实体的优先级在事件发生时由程序动态赋予，由图中的排序结果可知，抢占式调度结束后，事件响应的顺依次是 E1、E5、E7、E0、E2、E3、E6、E4 。

本小节讨论并行状态机调度的时候一直再说的是事件，放在 GF2.0 里看这些事件其实就是应用消息，所以事件缓冲队列在 GF2.0 中就是应用消息缓冲队列。如果 GF2.0 的调度器是抢占式的且事件优先级要求动态调整，那么第五章第 3 节的第 2 小节中讨论的应用消息的消息结构要做一点改动，即在消息结构体中再加入一个 8 位无符号整型成员，用来记录应用消息的优先级，如果只使用固定优先级，这个成员可以去掉以节省 RAM，但是需要在 ROM 中提供一张应用消息类型和固定优先级的对应关系表。

由抢占式调度器的调度原理可知，应用消息缓冲区被调度器调整前是队列结构的，调整后也是队列结构的，但是调整的时候却不是队列结构的，所以抢占式调度器具体的代码实现是和应用消息缓冲队列使用的存储结构密切相关的。

如果应用消息缓冲区的存储结构是第五章第 3 节的第 2 小节给出的数组结构，抢占式调度器可以实现，但是程序的执行效率很低，在数组中按抢占式调度的调整规则调整数组元素的顺序，操作非常繁琐。如果队列中有 m 个应用消息，最坏的情况下总共需要做 m(m+1)/2 次数据

比较和 m(m+1)/2 次数据移动，再加上缓冲队列是循环的，操作会更繁琐。

相比之下，如果应用消息缓冲队列使用静态链式存储结构(单向链表即可)，抢占式调度器程序执行效率则会高得多，最坏的情况下总共需要做 m(m+1)/2 次数据比较和 m 次数据移动。虽然静态链式结构执行效率比数组结构高，但是每个元素却要多使用一部分 RAM 来存储节点索引，所以使用静态链表实现消息队列比数组结构实现的消息队列要消耗更多的 RAM(单片机中 RAM 可是珍稀资源)。

跟合作式调度器相比，抢占式调度器实现起来复杂，资源消耗也多一些，执行速度也不如合作式调度器，但是抢占式调度器对事件的管理非常灵活，能实现更加精细更加合理的调度，性能上要好于合作式调度器。

总而言之，合作式调度器和抢占式调度器各有特点，鉴于合作式调度器简单易用的特点，如果能满足应用场合的要求，建议优先选用合作式调度器。

(3) 并行状态机的数据结构

并行状态机的调度包含 2 个步骤：从应用消息缓冲队列中读取应用消息、根据应用消息找到找到目标状态机并调用状态机代码。上一小节讨论的是第 1 步，本小节讨论第 2 步。第二章第 3 节讨论过状态机的代码实现，其中比较好的实现方法是 switch-case 法和压缩表格驱动法，本小节仅就这两种方法说明并行状态机在 GF2.0 中的结构。由第五章第 4 节的第 1 小节的内容可知，GF2.0 中的状态机任务主要包含 2 个部分：状态机代码、状态存储全局变量。

1） switch-case 法状态机任务代码结构

如果用 switch-case 法实现 GF2.0 中的并行状态机，根据 switch-case 法的特点可知，状态机任务的全部代码被封装在一个任务函数中，任务函数的声明形式

```
INT8U sm_task(INT8U u8CurStat , void* pAppMsg);
```

任务函数的内部结构类似程序清单 List4 所示的程序结构，形参 u8CurStat 把状态机当前的状态告诉任务函数，根据形参 pAppMsg 任务函数可以得知应用消息的消息类型和消息参数。状态机当前状态和应用消息消息类型构成了一个二元坐标，任务函数根据这个坐标在函数内部的嵌套 switch-case 结构中找到需要运行的代码片段，该片段能接收消息参数、响应事件、确定状态机新的状态，任务函数返回状态机新状态。

2） 压缩表格驱动法状态机任务代码结构

如果用压缩表格驱动法实现 GF2.0 中的并行状态机，根据压缩表格驱动法的特点可知，状态机任务的全部代码被拆分成了一个个动作封装函数，每一个动作封装函数都和状态机的一个状态相关联，动作封装函数的声明形式

```
INT8U sm_action_Sx(void* pAppMsg);
```

动作封装函数是散乱的，必须用一个叫做压缩驱动表格的一维数组把这些动作封装函数组织起来，压缩驱动表格 SMCprsDrvTbl[]中的数组元素叫做状态机节点，节点的结构形式

```
typedef struct sm_node                          /*状态机节点结构体*/
{
    INT8U (*fpAction)(void* pAppMsg);           /*事件处理函数指针*/
    INT8U u8StatChk;                            /*状态校验*/
}SMNODE;
```

状态机节点在压缩驱动表格中的数组下标代表一个状态值，即这个状态值关联一个状态机节点 SMNODE，而这个 SMNODE 通过成员 fpAction 关联一个动作封装函数，于是状态机的一个状态就和状态机任务代码中的一个动作封装函数关联起来了。三者之间的关系如图 27 所示。

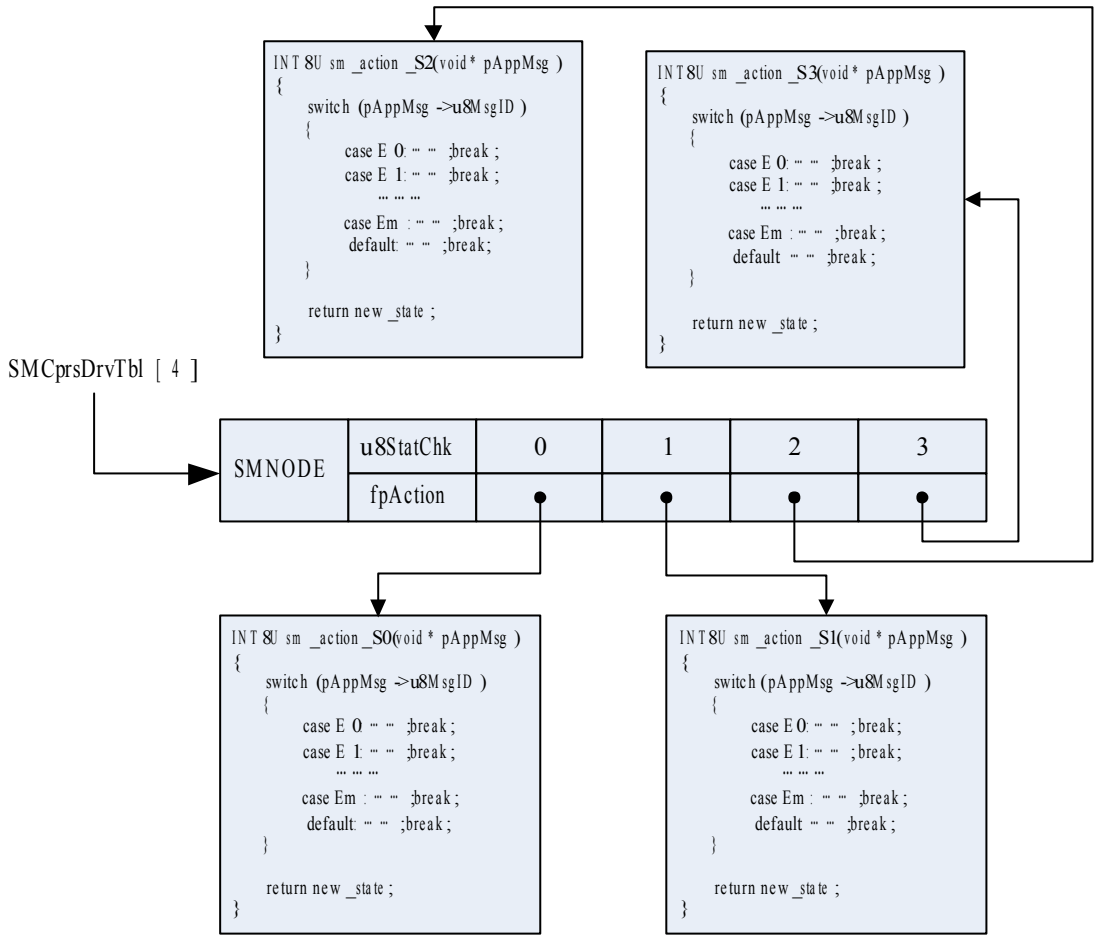

图 27 压缩表格驱动法实现的状态机任务代码结构

动作封装函数内部是一个和消息类型相关的 switch-case 结构(参见程序清单 List8)，调度程序首先根据状态机当前状态在压缩驱动表格中找到对应的 SMNODE，然后由 SMNODE 的成员 fpAction 定位动作封装函数。动作封装函数根据形参 pAppMsg 获取消息类型和消息参数，然后根据消息类型在函数内部找到目标代码片段，消息处理完毕后，动作封装函数返回状态机新状态。

3） 并行状态机任务控制块 SMTCB

仿照 RTOS 的任务控制块 TCB，GF2.0 也可以为每个状态机任务设置一个状态机任务控制块 SMTCB，结构体如下：

```
typedef struct sm_tcb
{
    BOOL    bMeth;          /*状态机任务代码结构标识*/
    INT8U   u8CurStat;      /*存储状态机当前状态*/
    void*   pCodeORTbl;     /*指向 switch-case 任务函数或压缩驱动表格*/
}SMTCB;
```

成员 bMeth 指明状态机的实现方式是 switch-case 法还是压缩驱动表格法，如果是 switch-case 法，成员 pCodeORTbl 指向任务函数入口，否则 pCodeORTbl 指向压缩驱动表格数组首地址，访问 pCodeORTbl 前要进行数据类型强制转换。

GF2.0 中存在多个并行状态机，每个状态机都有一个 SMTCB，GF2.0 把所有的 SMTCB 放在一个叫做并行状态机任务表 SMTaskTbl[]的数组中，ID 为 0 的状态机其 SMTCB 放入 SMTaskTbl[0]，ID 为 1 的状态机其 SMTCB 放入 SMTaskTbl[1]，以此类推。图 28 为 GF2.0 中并行状态机的结构全图。

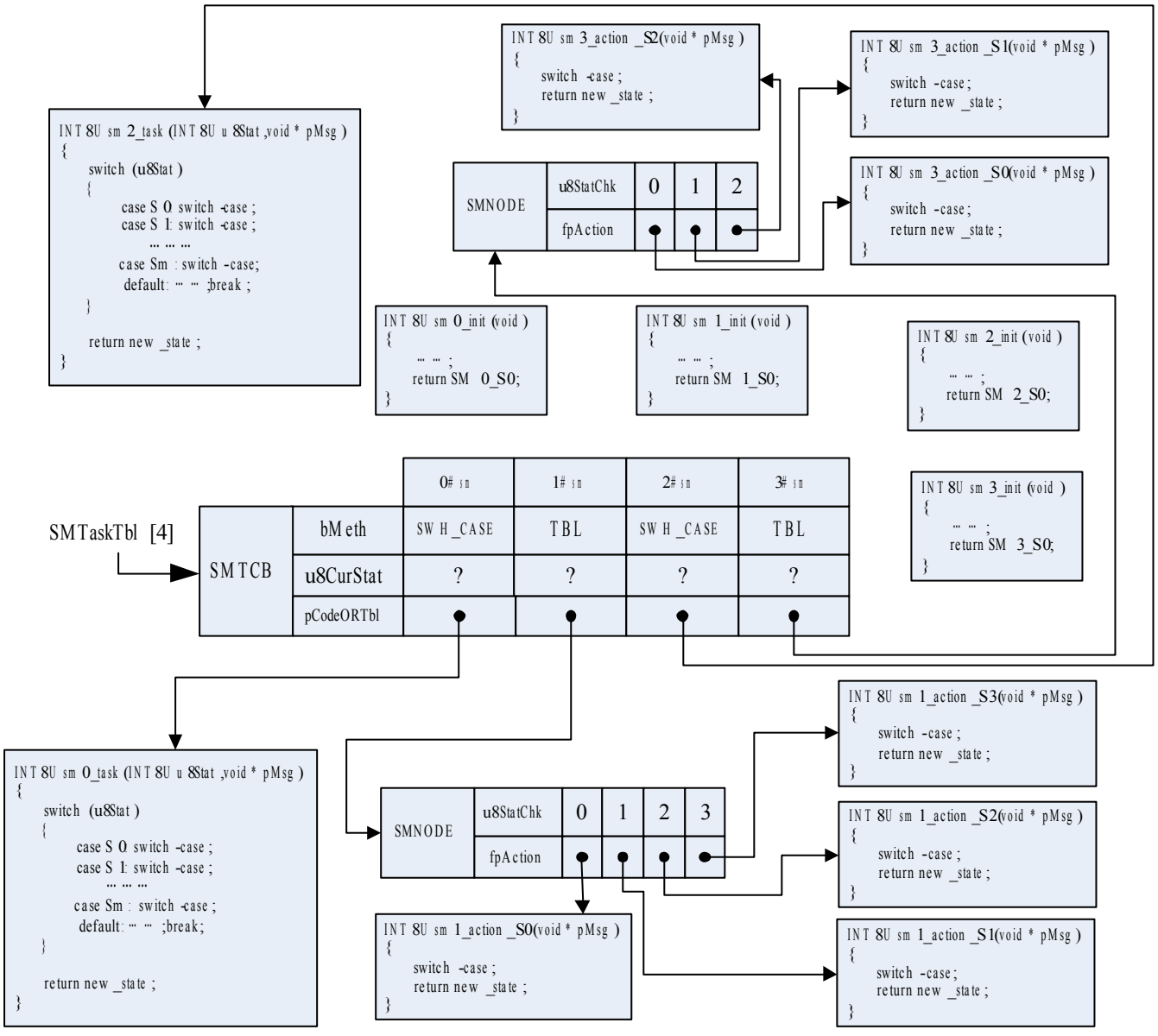

图 28 并行状态机数据结构全图

图 28 中存在 4 个没有被关联起来的函数，分别是 sm0_init()、sm1_init()、sm2_init()以及 sm3_init()，这些函数是状态机初始化函数，属于状态机任务代码的一部分，在 UML 状态图中代表由实心圆点到初始状态这个箭头上的动作，状态初始化函数返回状态机的初始状态。初始化函数要有统一的声明形式，如下

```
INT8U sm_init(void);
```

在并行状态机正式开始运行之前，系统初始化代码需要对并行状态机任务表 SMTaskTbl[]进行初始化，所以 GF2.0 还需要实现一个状态机任务初始化接口函数，函数声明形式如下

```
INT8U gf20_smtask_reg(INT8U (*fpInit)(void) , BOOL bMeth , void* pTaskEntry);
```

形参 fpInit 指向状态机初始化函数，gf20_smtask_reg()通过 fpInit 调用状态机初始化函数，并将返回值(即状态机初始状态)填入 SMTCB 的成员 u8CurStat，形参 bMeth 对应 SMTCB 的成员 bMeth，形参 pTaskEntry 对应 SMTCB 的成员 pCodeORTbl。

5、 GF2.0 为并行状态机提供的服务

截止到第五章第 4 节，GF2.0 必不可少的特性(双消息驱动、并行状态机管理)就已经完整的实现了，所以本节内容基本上就是为 GF2.0 锦上添花的。为了使 GF2.0 更加好用，也为了减少状态机编程时的重复劳动，我们应该把一些经常用到的功能集成到 GF2.0 中，这些功能可以作为 GF2.0 的服务提供给应用层的并行状态机。

(1) 状态机的定时服务

定时功能在单片机程序中用途非常广泛，常见的应用场合有程序延时、键盘扫描、数码管动态显示、数据通信超时检测等，因此 GF2.0 要把定时功能做成标准组件，为应用层并行状态机提供定时服务。并行状态机位于程序结构中的应用层，所以提供给它们的定时功能定时精度不用很高，能实现 ms 级的精度就可以了，后面讨论的实现方法都是基于这样的一个前提。

为实现定时管理，GF2.0 需要一个周期为 ms 级的定时器中断服务函数，仿照 RTOS，我把这个中断服务函数叫做系统时钟节拍 SysTick，SysTick 为定时服务提供时基信号，相当于定时管理模块的时钟源，由此可知，定时管理模块实际上是一个中断消息处理器。参照嵌入式软件设计中定时功能的常规实现方法，同时考虑 GF2.0 自身程序结构的特点(双消息驱动机制+并行状态机)，在 GF2.0 中实现定时服务有 2 种实现方法：软件定时器、SysTick 分频信号广播。

1）软件定时器

软件定时器是嵌入式软件中实现定时功能的常规方法，定时功能是通过回调函数机制(CallBack)实现的，申请定时服务的任务会交给定时管理程序一段特殊的代码，这段代码被封装成一个函数，叫做回调函数(CallBack)，定时管理程序保证回调函数按任务要求的定时间隔定时执行。

定时管理程序为每个申请定时服务的任务分配一个软件定时器，并将任务提供的回调函数的函数地址和回调参数地址保存在软件定时器中。在系统时钟节拍 SysTick 的驱动下，定时管理程序周期性查看被激活的软件定时器，如果某个软件定时器定时时间到了，定时管理程序会

以函数指针的形式调用回调函数，调用前还会将软件定时器中保存的回调参数地址提供给回调函数。

根据软件定时器的实现原理可知，这种方法实现的定时服务，回调函数定时最大误差=（先于定时管理程序运行的中断消息处理环节的执行时间）+（系统时钟节拍 SysTick 的定时周期）+（在本回调函数之前被调用的那些回调函数的执行时间之和）。

关于软件定时器的具体实现方法，网上资料很多，在这里就不详细展开了，虽然这些实现方法都是以 RTOS 为目标环境，但在 GF2.0 中也是适用的，图 29 所示是 GF2.0 使用软件定时器为应用层并行状态机任务提供定时服务的原理。

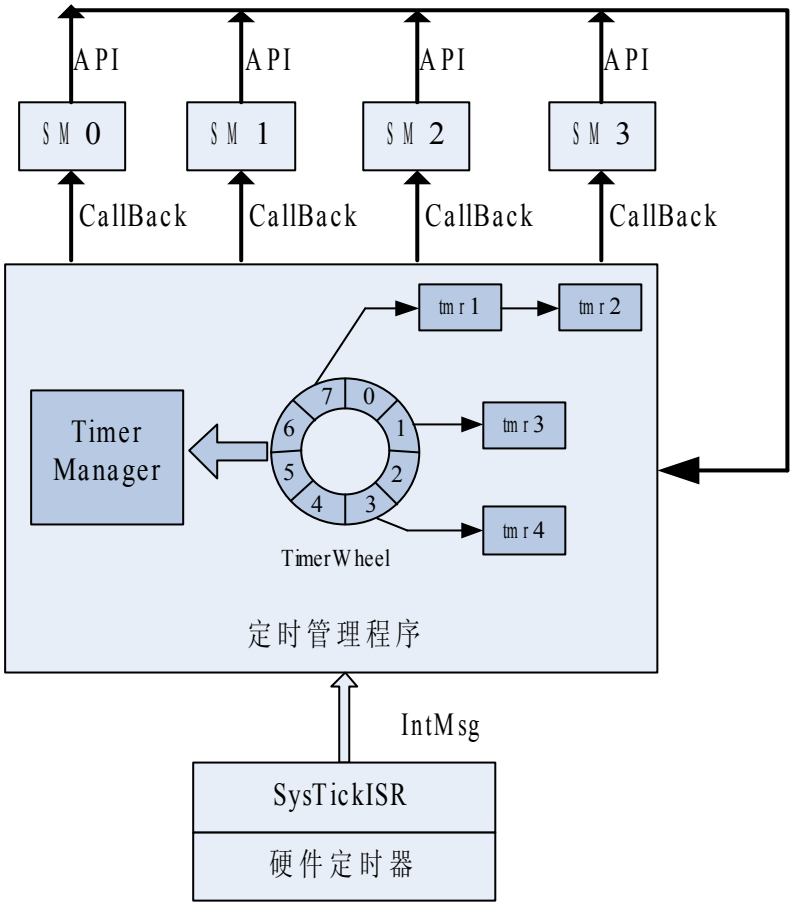

图 29 使用软件定时器实现 GF2.0 的定时服务

软件定时器可以实现非常完善的定时功能，但是却是以消耗大量的 RAM 资源为代价的，一个软件定时器结构体至少要能存储以下内容：

链表前驱节点指针 ；
链表后继节点指针 ；
定时器工作方式 ；
回调函数函数地址 ；
回调参数地址 ；
定时匹配计数器 ；
周期定时重载计数器 ；

由此可知，一个软件定时器结构体变量至少要占用十几个字节的 RAM，如果程序中软件定时器比较多，RAM 资源的消耗量是很可观的，这对于 8 位单片机来说是非常奢侈的，所以面对 8 位机的应用，GF2.0 需要使用一种资源消耗少的方法来实现定时服务。

2) SysTick 分频信号广播

软件定时器的实质是在 SysTick 的驱动下，定时管理程序对隶属于各个状态机任务的定时计数器进行集中管理。如果把定时计数器交给状态机任务自己管理，然后由定时管理程序向这些状态机任务提供节拍信号，这样只需要耗费少量的 RAM 存储定时计数器的计数值，极大地削减了定时服务 RAM 资源的消耗。SysTick 分频信号广播就是利用了这样的思路来实现的，图 30 所示是这种方法的原理图。

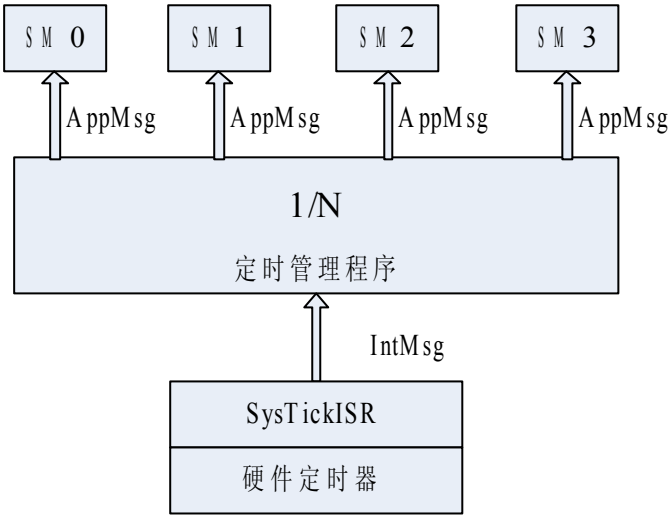

图 30 使用 SysTick 分频信号广播实现 GF2.0 的定时服务

顾名思义，SysTick 分频信号广播就是在定时管理程序中将 SysTickISR 发送中断消息的频率进行分频，再以这个较低的频率不间断地向状态机任务发送定时应用消息，需要定时的状态机任务对定时应用消息进行计数，配合定时计数器实现定时功能。定时管理程序默认所有的状态机任务都需要定时，所以会以广播的方式将定时应用消息发给所有的状态机任务，如果状态机任务不需要定时，忽略此消息即可。需要注意的是，状态机任务使用的定时计数器变量必须是全局变量或者静态局部变量。

发送定时应用消息的时候，如果为每个状态机任务都生成一条消息，很可能会使应用消息缓冲队列瞬间溢出，可以在广播式的应用消息上做特殊标记。应用消息结构体(参见程序清单 List10)中有个成员 u8UsrID，用来存储本条消息的目标状态机 ID，调度程序也是根据这个变量来调用状态机任务的，可以规定 u8UsrID 值为 0xFF 的应用消息是广播消息，如果调度程序发现 u8UsrID 值为 0xFF，就会将所有的状态机任务都调用一遍，这样只要在应用消息缓冲队列中放入一条广播式消息就行了。如果使用广播消息功能，则并行状态机任务不能使用 0xFF 作为状态机 ID 的值。

SysTick 分频信号广播可以为定时服务大量节省 RAM 资源，但是和软件定时器相比，这种实现方法提供的定时功能是非常粗糙的。

首先，定时精度上不如软件定时器。在软件定时器实现的定时服务中，回调函数是在定时管理程序中调用的，也就是说回调函数是在中断消息处理环节被调用的；在 SysTick 分频信号广播实现的定时服务中，需要被定时执行的代码是在状态机任务代码中运行的。由 GF2.0 的主循环流程图(第五章第 2 节图 20)可知，中断消息处理环节总是先于应用消息处理环节，再加上定时应用消息之前可能还会有一些应用消息需要被处理，所以第二种实现方法在定时精度上不如第一种方法。

其次，SysTick 分频信号广播使得状态机任务的定时功能实现起来更加复杂。定时计数器是状态机任务自己管理的，因此定时功能里的很多工作要由状态机任务自己来做，状态机本身也变得更加复杂。

举个例子，假设某状态机任务的唯一功能是每隔 200ms 向外界发送一次数据，SysTick 的频率是 200Hz，SysTick 分频信号广播使用的分频系数 N=5。图 31、32 所示为这个状态机在不同定时服务中的状态转换图。

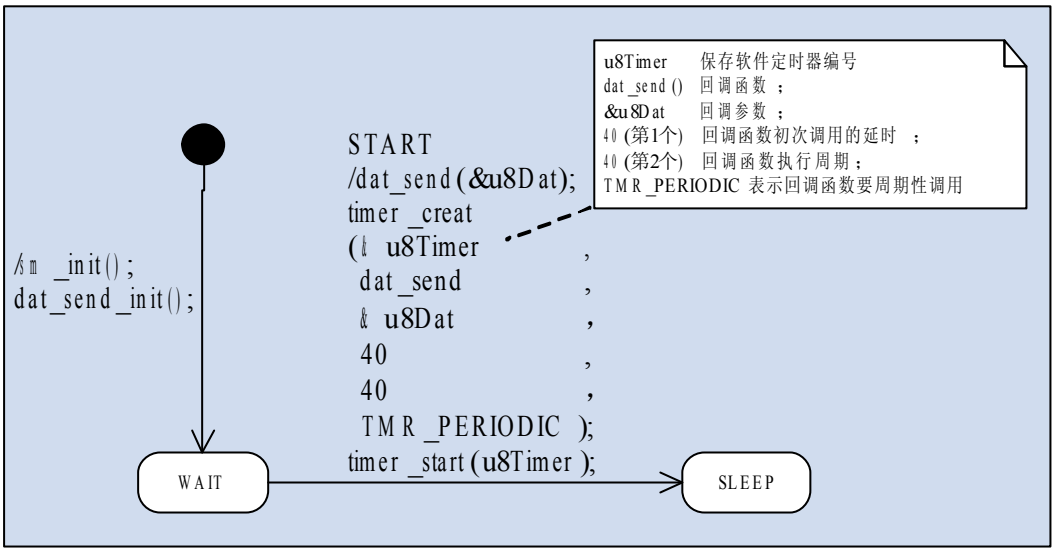

图 31 状态机使用软件定时器提供的定时服务

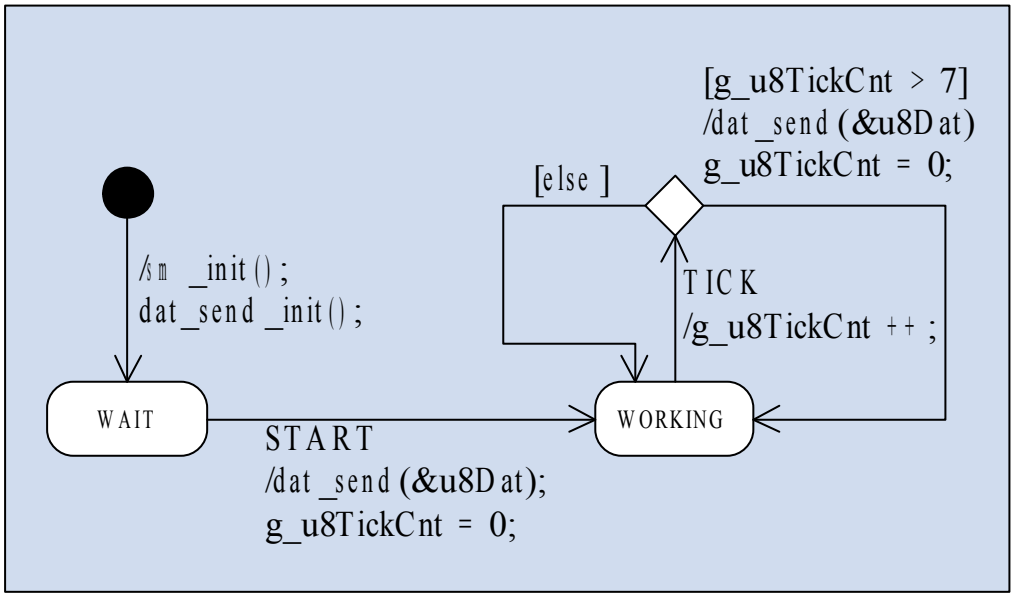

图 32 状态机使用 SysTick 分频信号提供的定时服务

(2) 状态机的同步和通信

本小节的内容是在 GF2.0 中如何实现并行状态机任务之间的同步和通信。在第五章第 4 节开始的时候谈到过正交化设计的概念，把每个正交域设计成一个独立的模块，这样模块之间就没有任何关联，彼此之间毫不影响。

遗憾的是，在实际应用中这种绝对的正交关系几乎是不存在的。同在一个大系统中，都是这个系统内的一部分，想要彼此完全独立是很难做到的。我们能做的只能是合理的划分整体功能，使各个子功能之间关联尽可能的少，这样在每个子功能模块具体实现的时候，我们只需要重点关注模块内部如何实现而不必把大量的精力放在对外关系(和其他功能模块之间的关系)的协调上。这就是高内聚低耦合的设计思想，是正交化设计面对实际问题的一种变通方式。

并行状态机的设计也要严格遵循高内聚低耦合的设计思想，各个状态机之间的关联越少越好。如果某项功能需要多个状态机之间大量的通信和同步才能实现，这种情况多半是因为并行状态机设计得不合理，这时就应该重新设计这些状态机，把那些需要频繁合作尤其是经常同步操作的状态机合并。万事总有例外，如果某些场合确实需要多个状态机之间的通信和同步，那么 GF2.0 就应该为状态机任务提供同步和通信服务，以防万一。

在 GF2.0 中，消息是一个重要且常见的环节，而消息的功能就是通信，所以并行状态机之间的通信很容易实现，通信的发起者只需要将数据内容和接收者的 ID 填入应用消息，然后把应用消息放入应用消息缓冲队列即可。因此本小节的重点还是状态机之间的同步。

同步是为了并行状态机能使用共享资源，GF2.0 的同步服务也是利用应用消息实现的，图 33 所示为 GF2.0 同步服务实现原理。

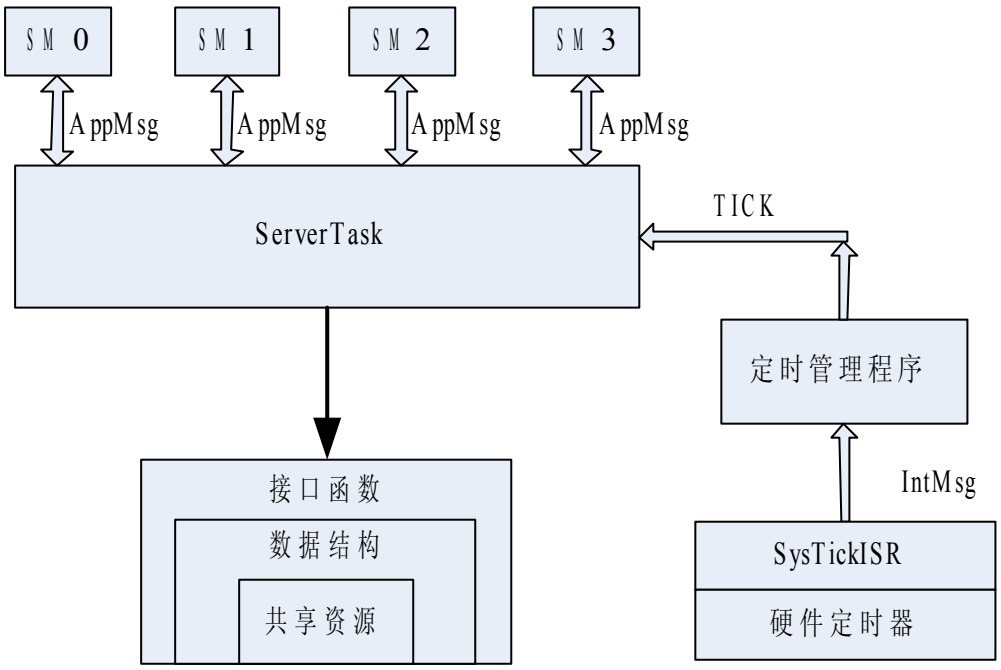

图 33 GF2.0 同步服务的实现原理

GF2.0 的同步服务使用了 C/S 的结构，共享资源由一个特殊的状态机任务集中管理，这个

状态机任务叫做 ServerTask ，除了管理共享资源，ServerTask 跟其他状态机任务相比没有任何特权，但其他的状态机任务都可以是 ServerTask 的客户。

为了处理客户状态机的等待超时，ServerTask 需要一个节拍信号，即要求 GF2.0 提供定时服务。如果定时服务使用软件定时器实现，则 ServerTask 会安排回调函数向自己发送 TICK 应用消息，如果定时服务使用 SysTick 分频信号广播实现，定时应用消息就是 TICK 消息。

某状态机任务想使用共享资源，首先必须以发送消息的方式向 ServerTask 提交申请 ，如果当前共享资源可用，ServerTask 收到申请后会将共享资源分配给这个客户，然后以发送消息的方式通知客户，客户状态机收到回应消息后继续运行。如果当前共享资源不可用，ServerTask 会记录此次申请，将客户状态机连同客户状态机指定的等待时限放入存储等待任务的数据结构如资源等待队列中。在 TICK 驱动下，ServerTask 对等待任务进行统一的等待时限管理，如果发现某个等待任务超时了，ServerTask 会将这个任务从等待队列中移除，然后向这个状态机任务发送消息，告之等待超时。

图 34 所示是一个互斥信号量 ServerTask 的状态转换图，结构比较简单，没有出错处理，客户状态机也没有优先级，只是用了先入先出的等待队列管理客户状态机申请。

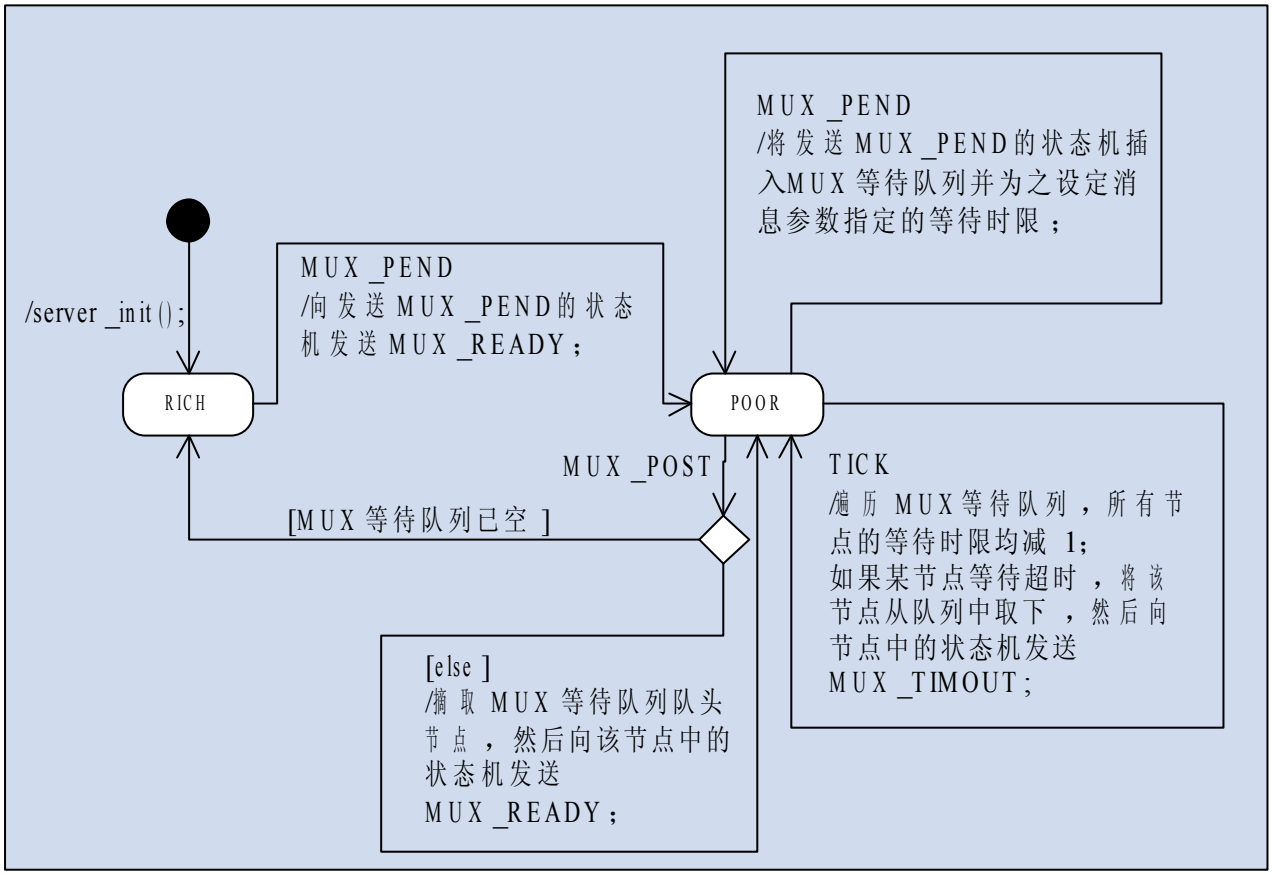

图 34 互斥信号量 ServerTask 状态转换图

图 35 是一个简单的互斥信号量客户状态机的状态转换图。状态机发送了申请之后进入等待状态 PENDING，收到 MUX_READY 消息表示状态机任务得到了资源，收到 MUX_TIMOUT 消息表示互斥信号量等待超时。从图中可以看出，PENDING 状态的状态机在等待资源的同时还可以处理其他消息(图中的 SOME_EVNT)，这也就是说，等待共享资源的状态机任务不是阻

塞的，而在 RTOS 中等待共享资源的任务一定是阻塞的。

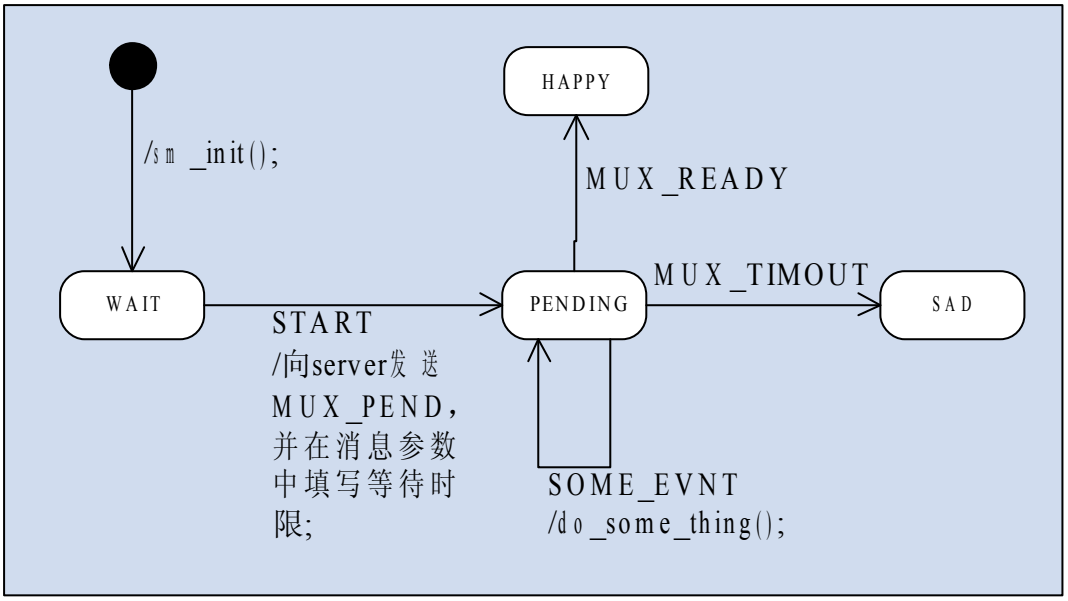

图 35 互斥信号量的客户状态机状态转换图

虽然 GF2.0 可以提供同步服务，但是同步会使得状态机结构变得复杂，建议在设计状态机的时候慎用同步功能，较好的处理方式是把有同步关系的状态机合并为一个状态机。

6、GF2.0 的可移植性

GF2.0 只是一个裸奔程序框架，从程序结构上说是一个前后台系统框架，所以它的移植是非常简单的。

1）临界段的进入和退出

在前后台系统中，访问前台后台共享资源的时候都要进行临界段保护。进入临界段的时候要禁用全局中断，退出临界段的时候要使能全局中断，C 语言中可以使用宏定义来描述这两个操作：

```
GF20_GBL_INT_DISABLE();     /*禁用全局中断*/
GF20_GBL_INT_ENABLE();      /*使能全局中断*/
```

这两个宏定义是和具体的单片机以及具体的开发环境有关的，如果开发环境中直接使用 C 代码就能实现开关全局中断，宏定义直接用 C 代码声明就行了，例如在 Keil C51 开发环境中可以这样声明：

```
#define   GF20_GBL_INT_DISABLE()   EA=0;
#define   GF20_GBL_INT_ENABLE()    EA=1;
```

如果编译环境支持嵌入汇编，则宏定义可以用嵌入汇编代码声明。
如果单片机有软中断功能且编译环境支持 C 语言调用软中断，则也可以用软中断来实现。

2）实现 SysTickISR

如果状态机任务要求 GF2.0 能提供定时服务或者同步功能，需要在移植的时候实现系统时钟节拍中断服务函数 SysTickISR。

3）修改配置头文件

在移植某些 RTOS 例如 uC/OS-II 的时候，需要修改配置头文件，对 RTOS 进行功能裁剪，一方面是为了去掉不使用的功能节省 ROM 和 RAM，另一方面是为了指定保留功能的资源消耗(主要是 RAM)。仿照 RTOS 的作法，也可以为 GF2.0 添加一个配置头文件。

根据第五章前 4 节的内容，GF2.0 需要配置的地方主要有 7 处：

中断消息和应用消息的结构

中断消息和应用消息都可以传递多种类型的消息参数，为了节省 RAM 这些消息参数的类型应该设置成可选的，配置形式类似程序清单 List10 的方式，最简的时候消息参数可以只占用 1 个字节的空间，这样的话中断消息占用 2 个字节，应用消息在不使用动态优先级的情况下只占用 3 个字节。

消息缓冲队列长度

中断消息缓冲队列和应用消息缓冲队列的长度都应该是可配置的，两者的长度应该保证即使在队列最繁忙的时候也不能溢出。应用消息缓冲队列一方面要接收来自中断消息处理环节的应用消息，另一方面还要负担着并行状态机之间的通信，所以应用消息缓冲队列的长度要大于中断消息缓冲队列的长度。

中断消息处理器向量表

如果要使用查表法调用中断消息处理器，还应该对这个向量表数组进行配置，至少要指明数组长度，防止读取函数指针的时候越界访问，造成程序跑飞。

并行状态机调度方式

如果 GF2.0 同时实现了合作式调度器和抢占式调度器，配置文件应该指明到底使用哪一种调度器。

并行状态机任务表 SMTaskTbl[]

并行状态机任务表存储状态机任务控制块 SMTCB，所以 SMTaskTbl[]数组长度也是要配置的，数组长度和并行状态机任务的数量是一致的。

定时服务

GF2.0 中的定时服务不算是程序框架的必要部分，可列为可选功能，在配置文件中使用编译常量决定是否编译此部分代码。如果使能定时服务，SysTickISR 的中断频率应该列为配置常

量；如果使用软件定时器实现定时服务，应该使用编译常量指明软件定时器的数量；如果定时服务使用 SysTick 分频信号广播实现定时服务，SysTick 分频系数需要在配置头文件中设置。

状态机任务的同步和通信

状态机之间的通信属于必选的功能，不需要配置，为状态机任务提供同步服务属于可选的功能，也应该在配置文件中设置常量使能或禁用这部分代码。

7、小结

GF2.0 的具体实现已经说完了，在小结里只谈一谈这种程序框架的意义。

在本文的第一章有过一个“单片机系统都是反应式系统”的论调，反应式系统又是交互式系统，在交互式系统中信息的流动是双向的，而 GF2.0 的重要意义就在于它把信息的流入和流出完全剥离了。流入的信息就是事件，GF2.0 提供了一个事件输入的管道，所有的事件全部由这个管道进入程序，应用程序只需要从这个管道中读取事件，就能得到外界所有的输入，这个管道的存在大大减轻了应用程序获取外界输入的负担。

GF2.0 还是一种设计单片机程序的工具，它为使用者提前圈定了解决问题的框架。双消息驱动机制强制把外界输入的处理分成了 3 个部分：ISR、中断消息处理、应用消息处理，这种处理方式显然把功能分成了 3 层，我把这三层分别叫做驱动层、协议层、事务层。状态机是解决实际问题的有力工具，但是单一的状态机在面对较复杂的功能要求时又会变得规模庞大难以把握，并行状态机以正交化的设计原则对复杂问题进行拆分，使得功能变得易于实现。

最后，GF2.0 是一种编程框架，更是一种设计思想，它在事件驱动的基础上又加入了状态机，使事件的处理更加细致。这种思想具备通用性，实现起来可繁可简，精简一下就是 GF1.0 的处理方式，细致一下就是 GF2.0 的处理方式。思想有了，剩下的就只是形式了。