定时器、中断、系统滴答和程序框架

目录

1. 面临的问题

这一章所要解决的问题是编程的思路，解决问题的套路和程序的框架。一般而言，教科书上面是按照一个个小的专题讲下来，所提供的软件示例都比较简单，例如单纯的写一个 LED 灯的驱动，单纯的实现一个按键的检测等等。然而在工程实践中所面临的场景并非如此简单，考虑一个简单的应用场景：硬件上面单片机外接一个 LED 灯和一个按键。每按下一次按键则改变一次闪烁模式。

闪烁模式 0：每 1 秒闪烁一次，每个周期亮 0.5 秒，灭 0.5 秒

闪烁模式 1：每 3 秒闪烁一次，每个周期亮 0.2 秒，灭 2.8 秒

闪烁模式 2：每 1 秒闪烁一次，每个周期亮 0.2 秒，灭 0.8 秒

上电的时候 LED 灯按默认模式 0 闪烁，按一次按键后变成按模式 1 闪烁，再按一次按键后变成按模式 2 闪烁，再按一次按键又变成模式 0 闪烁，如此循环往复。

对于这个应用，如何在 LED 灯闪烁的同时又能检测按键？如何实现 LED 点亮时间的定时，用软件延时还是定时器？如果考虑更复杂的场景，例如一个基于 51 单片机的 GSM 报警系统，单片机要不停地检测用户按键，通过串口 AT 指令集控制 GSM 模块，要时刻监视报警信号，要执行用户的各种按键命令，要控制 LED 灯指示状态，要驱动 LCD 模

块显示操作菜单，报警信息等等。也就是要同时执行很多个任务，这又该如何实现呢？

这是涉及到编程思路的大问题，也是嵌入式编程的最核心的问题。一旦掌握了编程思路，那么面对任何的应用，简单的也好，复杂的也好，都知道该怎么处理，否则即使简单的应用场景也往往一筹莫展。

2. 解决问题的核心思路

2.1 分时复用

在 PC 机上面编程，例如 linux 和 windows 上面的软件开发，程序员不用关心底层驱动，例如鼠标的驱动和按键的驱动都是由 OS 来完成的，用户程序只需要调用相应的 API 接口函数就可以处理输入输出，但在嵌入式系统中，所有的外设驱动都需要自己来实现，都要占用 CPU 来运行，因此分时复用是编程最基本的原则，牢记这一点：任何任务都不可以占用 CPU 过多时间。

```
void led_Driver(void)
{
    while (1)
    {
        led_TurnOn();
        for (i = 0; i < 500; i++)Delay1ms();
        led_TurnOff();
        for (i = 0; i < 500; i++)Delay1ms();
    }
}
int main(void)
{
    sys_Initialization();
    while (1)
    {
```

```
        led_Driver();
        key_Driver();
        UserInterface();
    }
}
```

例如上面的代码，通过软件延时的方式来实现 LED 灯的闪烁，这个在教科书上面演示一下 LED 灯驱动没有问题，但是实际应用中不能这样写。单纯的一个 LED 闪烁占用了所有的 CPU 时间，按键驱动等其他的任务都得不到执行了？ 不要说延时 500ms，除了可以在一个任务中短暂延时微秒级别的时长外，毫秒级别的软件延时都是不可取的，占用了宝贵的 CPU 时间。

那么怎么来实现分时复用呢？主要有两种，在没有操作系统支持的场景下使用状态机，在有操作系统的场景下使用多线程。下面分别详述。

2.2 协作式多任务

在大部分的单片机系统中，特别是 8 位机系统，受限于资源，基本上都是一个大循环处理所有的事情。

```
int main(void)
{
    sys_Initialization();
    while (1)
    {
        Task1();
        Task2();
        Task3();
        ...
        Taskn();
    }
}
```

如上面的主函数，sys_Initialization()初始化外设，例如配置时钟，启动看门狗，配置 GPIO, Timer, 串口等。然后就进入主循环。主循环

由多个函数组成，每个函数实现一个任务。所有的任务互相协助来实现应用。例如 Task1()实现按键的驱动，它周期性的扫描按键输入，并且当有按键输入的时候延时去除机械抖动，并把检测到的按键按下、释放等动作写入到一个全局数据结构中供其他任务使用。例如 Task2()实现 LED 的驱动，根据当前的闪烁模式，点亮 LED 灯，延时一段时间，然后再熄灭 LED 灯，延时一段时间，如此周期性地执行。再比如 Task3()则是处理用户输出，根据 task1() 任务监测到的按键动作执行动作，在这里是每当按键按下的时候改变一次 LED 灯闪烁的模式。这三个任务相互配合就实现了我们在第一节中提出的应用。这是一个大的框架，根据应用的复杂程度，可以加入更多的任务，例如 Task4()实现 LCD 屏的驱动，Task5()处理串口输入等等。

在主循环中的每一个任务都不可以占用 CPU 太长时间，每个任务基本的设计原则就是检查是否有要处理的事件，有事件需要处理则马上处理，没有则马上退出函数把 CPU 出让给其他的任务。如果处理一个事件需要花费很长的时间，则需要把这个事件拆封成多个子事件，避免长时占用 CPU 是任务函数设计的首要原则。这种多个任务相互协作的方式叫做协作式多任务。这种编程方法几乎是万能的，可以处理各种各样的系统应用。

2.3 多线程

近年来发展起来的 ARM7 Cortex-M0 M3 M4 架构的 32 位 MCU 资源多，移植了很多的嵌入式 OS，例如 ucOSII 、FreeRTOS、MQX 等等，51 单

片机上面也有 RTX Tiny 等操作系统。操作系统本身实现了任务调度机制，因此可以用多线程的方式来实现多任务，例如上面的 LED 闪烁，可以创建三个线程，一个用来实现按键驱动，一个实现 LED 灯闪烁，一个则处理按键命令。这三个线程相当于协作式多任务中的三个任务函数。OS 会给每个线程分配运行时间片，当运行时间片用完时就切换到下一个线程。分时复用，大的原则是一样的，不同的是协作式多任务需要主动退出任务函数出让 CPU,而线程则是由 OS 的调度器来协调多个任务之间的分时复用。

2.4 事件驱动机制

在协作式多任务中，任务函数的设计理念就是根据事件来执行动作，有事件则执行，没事件则马上退出。这又称为“事件驱动机制”。事件是一个宽泛的概念，例如外部 IO 口电平的变化，串口接收到字符，变量计数达到一定的次数等都可以称之为事件。然而最为常用的事件则是时间。事实上，现实世界的林林总总的事务都是在时间轴上面发生的，甚至连我们的人生也不过是一个时间的函数而已。在单片机程序的微观世界里面，各种事情的发生都和时间紧密相关，例如：LED 灯的闪烁周期、点亮的时间、熄灭的时间等等显然都是靠时间事件来驱动的；按键的驱动中扫描到按键以后要延时一段时间再次检测以去抖动；I2C SPI UART 这些通信协议的实现严格依赖于时序；对于一个测控系统而言，要周期性地去采样信号等等。以上这些例子都是靠时间来驱动的。

2.5 定时器

那么怎么来计时呢？在单片机系统中靠定时器来实现这一个功能。定时器是一种对时钟或者外部脉冲进行计数的设备。定时器中的计数寄存器根据系统时钟或者外部脉冲进行增减计数，当达到一定的数值后触发中断。虽然不同的单片机在具体的实现上有所区别，例如有的是加计数，有的是减计数，时钟源可能不同等等，然而在原理上都是一样的。

2.6 中断

中断是嵌入式系统的一个重要概念。正常而言，单片机按照程序顺序执行，然而现实世界中有许多需要及时处理的事件，例如串口收到一个字符，某个输入口监控着一个重要的报警信号，当此信号电平变换时意味着系统出现严重故障必须及时处理等等。如果靠程序周期性地隔一段时间去主动检查串口字符是否收到，或者此报警信号电平是否变化等则达不到时效要求，串口字符没有及时接收则可能被下一个输入来的字符覆盖掉，报警信号没有及时处理则可能引发爆炸等严重事故。中断则提供了一种及时处理事务的机制，当上面的重大事件发生的时候，暂停前台程序的执行，转而执行处理中断事件的一小段程序，处理完毕中断事件后再返回到前台程序执行。这里面有几个重要的概念定义：

中断源： 触发中断信号的重要事件称为中断源，现实中有各种各样的事件作为中断源，例如定时器计时到一定程度，外部输入的电平变

化，串口收到字符，以太网收到网络数据报文等等。

中断服务程序：就是当中断发生时转去执行处理中断事件的一段代码。相对于主程序而言。

中断优先级：现实世界中的中断事件本身也有优先级别的高低，例如串口丢一个字符可以再重传补救，但报警信号不及时处理则可能导致爆炸等严重事故，显然报警信号的优先级要高，当两种中断同时发生时，优先级高的中断会先行处理。

中断嵌套：如果在处理串口中断的时候发生了报警事件，则要中断当前的串口中断服务程序，转去执行更高级别的报警信号处理程序，像这样高优先级的中断能够中断低优先级中断的情况叫做中断嵌套。

2.7 定时器中断和系统滴答

系统滴答是所有计算机系统上的一个重要概念，系统滴答是指一小段时间，不同的系统上面系统滴答的长度不大一样，例如一个每秒钟 100 个滴答的系统，每个滴答的时间长度是 10ms，每秒 1000 个滴答的系统，每个滴答的时间长度则是 1ms。像 windows 或者 linux 这些操作系统中，很多的事件和滴答有关，像线程运行时间片、像内存页面的切换周期等。系统滴答是通过定时器来实现的，例如一个 10ms 长度的滴答，计算好 10ms 对应的定时器计数值，设置好定时器后使之每 10ms 产生一次定时器中断，在中断处理程序中用一个全局变量进行对中断次数进行累加，就可以得到系统从开机运行以来所经历的滴答数。在本文提供的示例代码中，实现了一个 1ms 时常的系统滴

答，并且提供了一个 time_GetTicks()函数，这个函数返回从开机以来运行的滴答数，也就是经历的毫秒数。这个函数是整个系统中最为重要的一个函数，整个系统中所有靠时间驱动的代码到处可以看到它的身影。

3. STC 单片机上的定时器、中断和系统滴答的实现

3.1 STC 单片机的定时器设置

传统的 8051 单片机有两个定时器/计数器: Timer0 和 Timer1。STC12 STC15 系列单片机上面除了兼容上述传统的 Timer0 Timer1 两个定时器外，还增加了 2 个、4 个甚至更多的定时器模块以适应电机驱动、PWM 输出等应用需求。在这里我们使用 Timer0 来实现系统滴答。

Timer0 有四种工作模式，分别为:

模式 0 – 13 位定时器、计数器

模式 1 – 16 位定时器、计数器

模式 2 – 8 位自动重装模式

模式 3 – 两个 8 位定时器、计数器

在这里我们使用模式 1,也就是最为常用的 16 位的定时器模式，计数时钟为系统时钟的 1/12，如果系统时钟为 11.0592MHZ，那么每隔(12/11095200)秒计数器值增加 1，当计数到 65536 十六位定时器溢出的时候，定时器的中断标志位置位，如果使能了定时器中断将会产生

中断。此处要产生 1ms 的定时时常，也就是 1000HZ 的中断频率，则可以计算出计时器的计数初值，假设计数初值为 C 则：

65536 – C = 11059200 / 12 / 1000 = 921.6 取整数 922.

所以 C = 65536 – 922 = 64614.

对应的对 Timer0 的初始化函数如下：

```
static unsigned int code g_timer0_init_val = 65536 - (CFG_SYSFREQ/12/1000);
/*Timer 0 as system tick timer. 1KHZ */
void time_Initialization(void)
{
    /*Timer0 at 16-bit timer mode.*/
    TMOD = (TMOD & 0xf0u) | 0x01u;
    TL0 = g_timer0_init_val & 0xff;
    TH0 = g_timer0_init_val >> 8;
    ET0 = 1; /*Enable timer0 interrupt.*/
    TR0 = 1; /*Start running.*/
}
```

TMOD 设置 Timer0 工作在模式 1. TL0 和 TH0 根据设置为计算好的初值。定时器溢出频率为 1000HZ，也就是每 1ms 中断一次。然后使能定时器中断，并且启动 Timer0 运行。

在 main()函数中进入主循环之前，使能全局中断：

for(EA = 1;;)

3.2 STC 单片机的定时器中断处理程序

按照上节初始化 Timer0 之后，使能全局中断后，Timer0 将会每 1ms 产生一次,在中断处理程序中需要重新设置 TL0 和 TH1 的值。然后还需要把 g_sys_ticks 全局变量自增。

```
/*System ticks*/
static volatile unsigned long idata g_sys_ticks;
void timer0_ISR(void) interrupt 1
{
```

```
    TL0 = g_timer0_init_val & 0xff;
    TH0 = g_timer0_init_val >> 8;
    g_sys_ticks ++;
}
```

g_sys_ticks 全局变量是一个 32 位的 unsigned long 长整形的变量，最长计时可达 49.7 天时间。关于中断处理程序的一般性原则，需要注意中断 ISR 要尽可能的简短，一般而言处理最紧要的事情，设置标志，读取数据到缓冲区，然后在主循环中做处理。ISR 处理太多事情，长时占用 CPU 是编程的陋习，会导致系统的实时反应能力大大下降。这是中断处理的一般性原则需要加以注意。

3.3 STC 单片机的系统滴答函数

```
unsigned long time_GetTicks(void)
{
    unsigned long ticks;
    EA = 0;
    ticks = g_sys_ticks;
    EA = 1;
    return ticks;
}
```

上面 time_GetTicks()就是获取系统开机以来运行的滴答数的 API 函数，注意如果要获取系统滴答数要通过这个函数来读取，而不应该直接读取 g_sys_ticks()全局变量。 其原因在于 g_sys_ticks 是一个 4 字节的全局变量，在 8 位机系统上面不是原子变量，而且这个变量是一个中断处理程序和主程序共享的变量，故在主程序访问的时候必须通过禁用全局中断来进行互斥。其原因在于，假如主程序在读取 g_sys_ticks 中间，发生了 Timer0 中断，而在中断中 g_sys_ticks 将会被更新，这时 g_sys_ticks 读取的值将会是错误的。同样的在有操作系统支撑的系

统中，中断 ISR 和线程、不同的线程之间共享的变量也必须通过禁用全局中断或者使用互斥锁等方式来进行保护。

4. 协作式多任务实例分析

上面讲了解决问题的思路和基础设施-系统滴答的实现，后面就是用大量的实际案例来实践这个解决问题的思路，大家通过这些实际的案例就能逐步掌握解决问题的技巧，进而掌握嵌入式系统编程的精髓。本文中讲述四个简单的案例，后面会有更多的案例分析，大家会发现，从简单的 LED 驱动，到串口通信，到复杂的 GUI 系统，几乎一切的问题全部都是用这同一个思路解决问题的。无论是 51 架构，还是新兴的 ARM 架构，无论是这种大循环的裸奔型方案还是带嵌入式操作系统的复杂方案，只要掌握了思想的精髓，剩下的无非就是对具体机型，具体操作系统 API 的熟悉而已。掌握了编程的大思路就拿到了嵌入式系统设计的入场券。

4.1 LED 灯 0.5 秒点亮 0.5 秒熄灭的周期性闪烁

```
33 void led_Task(void)
34 {
35         static unsigned long idata oldticks;
36         static unsigned char idata on;
37         unsigned long ticks;
38 
39         ticks = time_GetTicks();
40         if(ticks - oldticks < 500) return;
41         oldticks = ticks;
42         on ^= 0x01;
43         led_TurnOn(on);
44         printf("LED running\r\n");
45 }
```

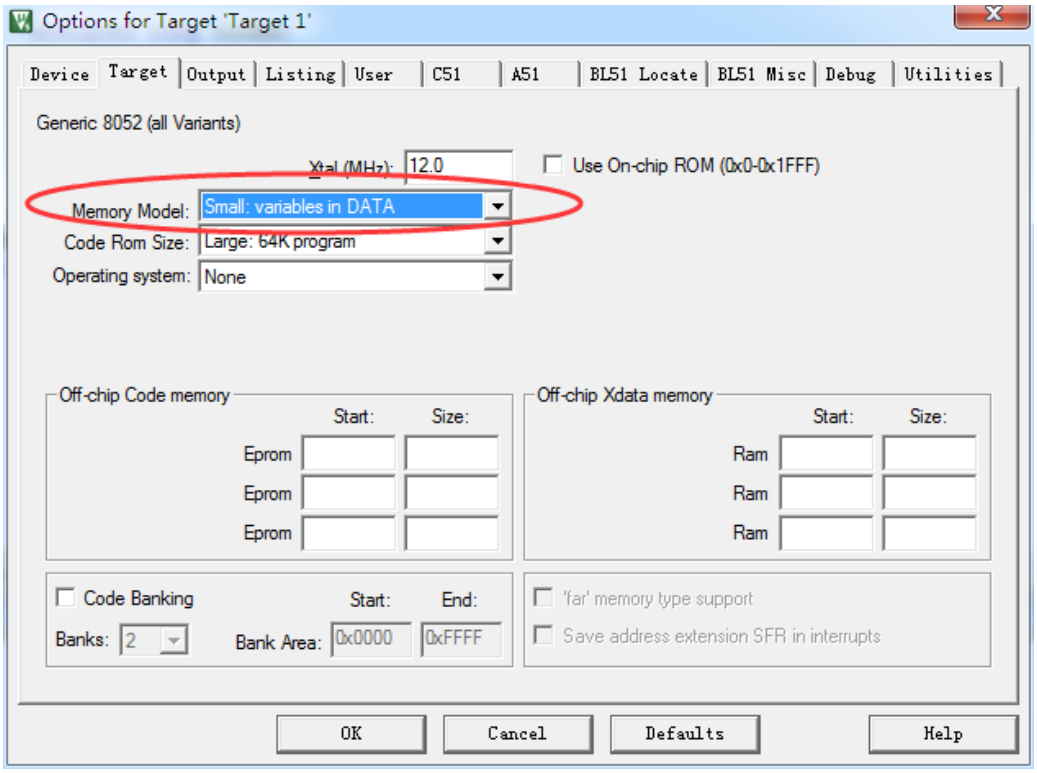
如上图所示，35 行定义一个静态局部变量 oldticks，这是一个 32 位的 unsigned long 型变量。大家在代码中看到很多的 C 语言修饰符，例如 static const struct enum，这些都是 C 语言的标准特性，可能有人会认为 C51 是用于单片机系统的 C 语言应该是标准 C 语言的子集，事实上恰恰相反， C51 是标准 C 语言的超集，标准 C 语言的特性除了部分标准 C 库函数因为 51 系统资源的局限性没有实现外，语言特性无所缺少。不仅如此，针对 51 架构的特殊性，C51 还增加了一些语言特性关键字，例如 xdata idata pdata bdata data bit code sbit sfr interrupt 等来做相应的优化和支持。所以作为一个 51 系统的软件工程师，如果使用 C51 语言来编写代码，那么首要的要求就是对 C 语言要非常熟悉，如果在这方面有欠缺，则需要补充这方面的知识技能。本文不打算展开讲解 C 语言和 C51 语言扩展的特性，仅对某些地方做一些特殊说明，如无法理解的地方可以找 C 语言的书籍来补充。在代码中常见的使用 static 关键字的地方有三处，一个是修饰函数，例如 static void Delay1ms(void) 这是为了限定函数名的作用域仅限于其所在的.c 源文件中，假如其他源文件中也有同名的 staitc 修饰的函数，那么它们的命名互不干扰，一般而言把实现同一功能的函数放在同一个文件中，这样一个文件可以看作是一个软件模块，在这个软件模块中有函数是作为对外接口的函数，也就是可能被其他软件模块调用到的函数，这样的函数不能用 static 修饰，而有些函数则是仅被同一源文件中其他函数调用的子函数，这样的函数要用 static 修饰，这样增强模块的封装性，是一个良好的编程习惯，在 linux 内核这样的大型

工程中，static 修饰符到处可见。在模块内子函数前面不使用 static 修饰对于小型工程来说无伤大雅，编译链接都不会有问题，但这是一种陋习，需要改正。事实上，模块化、不污染全局名字空间的理念是如此重要，以至于成为 C++函数的三大标准特性:封装、继承和多态中的一个。static 使用的第二个地方就是修饰全局变量，此时的作用和修饰函数是一样的，就是限定全局变量的名字作用域在本源文件中。从封装和模块化的角度来说，所有的全局变量都应该用 static 修饰。static 使用的第三个地方是在函数内部修饰变量，被 static 修饰的局部变量其作用域仅限在函数内部，其余的则和全局变量的作用完全一样。函数内部的自动变量在函数入口被初始化，在函数返回时被回收，但静态局部变量仅在上电后 main 函数执行之前被初始化，在函数进入与返回时不被初始化、不被回收，其生命周期为整个运行期间。实际上这些静态局部变量也可以在函数外声明成全局变量，但是很显然这会破坏全局命名空间，违背模块化原则，在代码中全局变量满天飞同样是一种陋习，需要改正。

35 行定义了 oldticks 用于记录上次任务函数执行时的时间。37 行的 unsigned long ticks;是局部自动变量，函数执行时 39 行会读取当前时间，然后在 40 行会比较当前时间和上次执行时间的差值，如果差值小于 500ms，则说明时间事件还没有到，则立即退出任务函数。如果大于等于 500ms 则说明时间事件已到，此时则 41 行更新 oldticks 为当前时间，然后 42 行 43 行反转 LED 灯的执行状态后退出。led_task()任务函数在主循环中每隔一小段时间运行一次，每次都是检查时间事

件（当前时间和上次事件发生时的时间超过 500ms）是否发生，若发生则执行任务，否则立即退出。这样的设计方式保证了 LED 灯按照 500ms 的周期闪烁，同时其他的并行任务例如按键扫描等都能得到及时的执行。初次接触，可能会比较难以理解，但大家需要再再的思考，直到理解其流程为止。

另外需要补充说明的一点是，全局变量和静态全局变量尽可能放在外部逻辑空间也即是 xdata 空间或者高 128 字节的 idata 空间，而局部变量则使用默认情况也就是放在 data 空间，这里面涉及到对 51 架构内存空间的理解，以及 C51 的 memory model 也就是内存模型对不同空间的使用情况的理解，详情可以参阅 51 单片机的数据手册以及 C51 用户手册。总结而言，绝大多数情况下，使用 small memory model:



在这模式下，变量默认放在 data 区，为了给运行时栈留出足够的空间，全局变量尽可能放在 xdata 区和 idata 区。

4.2 LED 灯 0.2 秒点亮 0.8 秒熄灭的周期性闪烁

```
33  void led_Task(void)
34  {
35          static unsigned long idata oldticks;
36          static unsigned char idata state;
37          unsigned long ticks;
38
39          ticks = time_GetTicks();
40          switch (state)
41          {
42          case 0:
43                  led_TurnOn(1);
44                  oldticks = ticks;
45                  state = 1;
46                  break;
47          case 1:
48                  if (ticks - oldticks < 200) break;
49                  oldticks = ticks;
50                  led_TurnOn(0);
51                  state = 2;
52                  break;
53          case 2:
54                  if (ticks - oldticks < 800) break;
55                  led_TurnOn(1);
56                  oldticks = ticks;
57                  state = 1;
58                  break;
59          default:
60                  break;
61          }
62  }
```

和上次的闪烁模式不同，上节的 LED 灯按照 500ms 的占空比进行闪烁，而本节的闪烁模式点亮时间为 200ms，熄灭时间为 800ms，对应的要对应两种不同的时间事件，一个是延时 200ms 到，一个是延时 800ms 到；为了区别这两种不同的状态，引入了一个静态变量 state，用于记录任务运行的状态，在这个任务中有三个状态：一个是初始状态，也就是任务函数第一次被执行的时候的初始化状态，第二个是点亮状态，需要等待延时 200ms 事件以切换到第三个状态也就是熄灭状态，在熄灭状态需要等待延时 800ms 的时间事件，然后再切换到

点亮状态，然后点亮和熄灭状态轮番交替就形成了闪烁的效果。这种情况类似于数字逻辑电路中的状态机模型，故又成为软件状态机，实际上软件状态机就是协作式多任务处理问题的具体实现，一切任务都可以分解为不同的状态，状态之间的切换就是靠事件驱动的，故又称为事件驱动状态机模型。这和数字逻辑设计中的状态机模型在理念上是一致的，数字设计中的状态机同样也是依靠外部事件的驱动在不同状态间进行切换的。

上节中的 LED 灯 500ms 闪烁实际上是一个状态机的特例，因为 LED 灯点亮和熄灭的等待时间相同，故使用了 on 变量代替了 state 变量。而每隔 500ms 则执行一次 on ^= 0x01;来实现状态切换。

具体实现上面，使用 switch(state)语句对状态进行散转，特别注意 state 变量必须是静态变量。任务函数第一次被调用的时候状态为默认值 0，也就是初始化状态，在这个状态下点亮 LED 灯，记录当前时间到 oldticks 变量，把状态设为 1，然后退出函数。当后面循环执行任务函数的时候，state 为 1，此时会检查延时时间是否到 200ms，未到则立即退出，已到则熄灭 LED 灯，然后切换到状态 2，并更新 oldticks 为当前运行时间。后续循环执行任务函数时散转到状态 2 分支，检测 800ms 延时事件是否到达，若未达到则退出，若达到则更新 state 为 1，点亮 LED 灯，并更新 oldticks 为当前时间。如此任务在状态 1 2 之间根据延时事件不停的循环切换，实现了 LED 灯的 200ms 点亮，800ms 熄灭的状态切换。这是我们遇到的第一个事件驱动的状态机任务函数，大家需要仔细体会，理解其运行机制最为重要。

4.3 按键驱动的实现

按键驱动的实现是事件驱动状态机模式的又一个生动的应用。按键和 LED 灯似乎是最为简单的外设，用一个 IO 口就可以驱动，但是软件上面检测按键并非像看起来那么简单。一个完整的驱动需要检测按键的按下、重复按键、抬起等事件，并且不能影响其他任务的执行。并且要能够通过软件消除抖动。

按照事件驱动的状态机模型进行分析，按键动作有以下几种状态：

按键未被按下

第一次检测到按键被按下

延时去抖动后再次检测按键是否被按下

用户一直按着按键不松开

第一次检测到用户已经松开按键

延时去抖动后再次检测按键是否被松开

松开按键后再回到初始的按键未被按下的状态。

```
void key_Task(void)
{
    unsigned char key_code;
    unsigned long ticks = time_GetTicks();
    static unsigned long idata oldticks;
    static unsigned char idata old_code,state = 0;
    switch(state)
    {
    case 0:
        old_code = ScanKeyCode();
        if(old_code == KEY_NONE) break;
        oldticks = ticks;
        state = 1;
        break;
    case 1: /*de-bounce*/
        if(ticks - oldticks < 20) break;
```

```
        key_code = ScanKeyCode();
        if(key_code == old_code)
        {
            key_InsertKeyCode(key_code, KEY_PRESSED);
            oldticks = ticks;
            state = 2;
        }
        else
        {
            state = 0;
        }
        break;
    case 2: /*Check for repeat*/
        key_code = ScanKeyCode();
        if(key_code == KEY_NONE)
        {
            oldticks = ticks;
            state = 3;
        }
        else
        {
            if(ticks - oldticks > 250)
            {
                key_InsertKeyCode(old_code, KEY_REPEATED);
                oldticks = ticks;
            }
        }
        break;
    case 3: /*check for release*/
        if(ticks - oldticks < 20) break;
        key_code = ScanKeyCode();
        if(key_code == KEY_NONE)
        {
            key_InsertKeyCode(old_code, KEY_RELEASED);
            state = 0;
        }
        else
        {
            oldticks = ticks;
        }
        break;
    }
}
```

上面的按键驱动程序就是采用这种状态机的方式来实现的，下面进行详细的讲解：

```
case 0:
    old_code = ScanKeyCode();
    if(old_code == KEY_NONE) break;
    oldticks = ticks;
    state = 1;
    break;
```

状态 0 对应的是按键没有被按下的初始状态，按键 Task 在主循环中被周期性调用。按键 task 大部分时间都是处在这个状态 0 的，每次按键 task 被调用就会执行一次按键扫描,如果按键没有被按下，则立即返回，只有扫描到按键按下的时候才会进入状态 1，进入状态 1 之前需要记录当前运行 tick 时间。

```
case 1: /*de-bounce*/
    if(ticks - oldticks < 20) break;
    key_code = ScanKeyCode();
    if(key_code == old_code)
    {
        key_InsertKeyCode(key_code, KEY_PRESSED);
        oldticks = ticks;
        state = 2;
    }
    else
    {
        state = 0;
    }
    break;
```

状态 1 是延时 20ms 后重新扫描按键，如果这一次扫描的按键和上次是一样的，则说明按键已经稳定，则通过 key_InsetKeyCode()函数把按键码和按下动作写入到一个全局数据结构中，留待其他任务读取按键并处理。如果这次扫描到的按键值和状态 0 扫描到的按键值不同，则说明刚才的按键动作是机械抖动则回复到状态 0 继续扫描。

```
    case 2: /*Check for repeat*/
        key_code = ScanKeyCode();
        if(key_code == KEY_NONE)
        {
            oldticks = ticks;
            state = 3;
        }
        else
        {
            if(ticks - oldticks > 250)
            {
                key_InsertKeyCode(old_code, KEY_REPEATED);
                oldticks = ticks;
            }
        }
        break;
```

状态 2 对应的是用户按住按键一直不松开的状态，这个时候要能识别重复按键，实际应用中很多场合下会用到长时间按键，例如按下按键超过 10 秒钟则系统恢复出厂设置等等。因此检测重复按键是很常用的功能。

在上面的代码中周期性地扫描按键，如果发现按键已经抬起则进入状态 3，否则会检测按下去的时间长度，并且每延时 250ms 则记录一次重复按键事件。

```
    case 3: /*check for release*/
        if(ticks - oldticks < 20) break;
        key_code = ScanKeyCode();
        if(key_code == KEY_NONE)
        {
            key_InsertKeyCode(old_code, KEY_RELEASED);
            state = 0;
        }
        else
        {
            oldticks = ticks;
        }
        break;
```

状态 3 是等待按键释放的状态。在状态 3 扫描按键是否已经释放，如

果已经释放了则记录释放的动作，并转变状态到 0. 如果尚未释放则每隔 20ms 重复扫描一次，直到释放为止。

4.4 用户处理按键

如我们文章开头中提出的应用场景:

每按一次按键则改变一次 LED 灯的闪烁模式:

闪烁模式 0：每 1 秒闪烁一次，每个周期亮 0.5 秒，灭 0.5 秒

闪烁模式 1：每 3 秒闪烁一次，每个周期亮 0.2 秒，灭 2.8 秒

闪烁模式 2：每 1 秒闪烁一次，每个周期亮 0.2 秒，灭 0.8 秒

为此，在主循环中有三个任务，分别是按键驱动任务，LED 灯闪烁任务和用户处理按键的任务。

```
38          for(;;)
39          {
40                  led_Task();
41                  key_Task();
42                  ProcessKey();
43          }
44  }
```

其中用户处理按键的任务，其代码如下:

```
 9  void ProcessKey(void)
10  {
11          static unsigned char idata led_flash_mode = 0;
12          unsigned char ret,keycode,keystate;
13
14          ret = key_GetKeyCode(&keycode, &keystate);
15          if(ret == 0) return;
16          if(keycode == KEY_SET && keystate == KEY_PRESSED)
17          {
18                  led_flash_mode++;
19                  if(led_flash_mode >= 3) led_falsh_mode = 0;
20                  led_SetFlashMode(led_flash_mode);
21          }
22  }
```

这个任务同样也是事件驱动，在主循环中被周期性调用，每次执行都

会读取用户按键，如果没有按键按下则立即返回，否则就检测按键的键值是否 KEY_SET 以及是否是按下事件，若满足条件则改变 LED 灯的闪烁模式。

4.5 任务之间的交互

如上面所示的三个任务，LED 灯闪烁、按键驱动和用户按键处理这三个任务。他们之间的数据交互是怎么实现的呢？LED 灯的闪烁模式的改变是通过一个 led 软件模块中的 API 函数 led_SetFlashMode()来实现的。在这个函数中设置全局变量 on_span off_span。 而按键驱动和用户按键处理之间的交互则是通过一个 FIFO 队列来进行交互的。并且对这个 FIFO 数据结构的操作使用了两个包裹函数：

```
extern unsigned char key_InsertKeyCode(unsigned char key_code,
unsigned char state);
extern unsigned char key_GetKeyCode(unsigned char *pcode,unsigned
char *pstate);
```

像这样，每个功能模块自成一体，通过 API 函数对外交互，把本功能所用到的数据结构封装在模块内部，大大提高了模块的封装性、独立性和可移植性。把数据结构杂糅在一起，看起来乱糟糟的一团，是一个编程陋习，需要改正。

5. 小结

本文讲述了单片机编程的程序框架，熟悉单片机的数据手册，掌握 C

语言以及 C51 对 C 语言的扩展固然重要，但语言只是工具，更重要的是使用工具解决问题。分时复用、协作式多任务、事件驱动机制、状态机这些单片机系统编程中重要的概念都在本文中做了详细的讲解，同时作为最重要的事件，系统滴答，在 51 系统中如何实现用代码实例做了讲解，并且讲解了定时器和中断两个重要的概念。最后用 5 个实际的例子讲述了如何使用上述思路解决实际的问题。

6.源代码下载

本文配套的 demo 工程、源代码在产品随附资料中。

深东电子： 专注 STC 单片机应用

深东电子网站 http://www.shendongmcu.com

深东电子淘宝 http://shendongmcu.taobao.com