从单片机编程到操作系统产生

nicekwell

前言

在 2011 年年初，笔者开始了单片机的学习，那时还是学生的我已经深深地被单片机吸引。之后几乎放弃了学校课程，把所有精力都放在单片机上，参加了一些比赛，并取得不错的成绩。

在开始工作之后，笔者从事单片机开发。在工作期间的高强度编程下，尝试了多种单片机编程结构，对单片机各种结构的构建做了分析和总结，并深刻感受到操作系统产生的必要性，对操作系统的探索有着越来越强的欲望。

在工作一段时间之后，笔者决定辞职。我不想凭仅有的那么一点单片机知识做一个嵌入式码农，我渴望学习新的知识，掌握更高级的理论……好吧~至少做一个高级一点的码农。而下一步的目标就是——linux。

但是笔者并没有直接进行 linux 的学习，因为操作系统对我来说还是个新事物，我甚至不知道操作系统和单片机编程的最本质区别是什么，于是我想先搞清楚操作系统的本质原理之后再进行 linux 的学习。笔者是物理学专业的，并没有系统学习过操作系统的概念。事实上很多介绍操作系统的书籍也没有提到操作系统的底层实现，更没有介绍它与单片机编程之间的关系。正如我在辞职的这段时间学完 uCOS 之后，对操作系统内核构建有了了解，但是却仍然不知底层的任务切换是如何实现的。

However，经过两个月的学习，笔者终于清楚了操作系统的基本原理，也深刻体会到操作系统和单片机各种编程结构的区别和联系。于是我想到把之前的单片机编程经验和近期对操作系统的学习结合到一起，写一个从单片机到操作系统过渡的文章。一方面对单片机编程结构做一个系统总结，另一方面对操作系统原理也进行一次整理。把操作系统和单片机编程整合到一个知识体系中去，以便日后接受更多的操作系统内核知识。

本文分为两部分——“单片机编程篇”和“操作系统篇”。

单片机编程篇主要介绍单片机的各种编程结构，及其实现方法。是在假设已经可以驱动单片机和各个模块的前提下，讨论如何整合和使用这些资源，以实现功能。在这一篇中，将会看到程序主体逐渐从主函数转移到定时器，并且明确这些变化的目的。最后还会接触到面向对象的程序设计方式，并体会这种方式带来的好处。

操作系统篇将会介绍操作系统的最基本的任务切换原理，以及操作系统是如何实现在单片机编程中难以实现的功能的。并在 arm 平台上实现任务切换和简单的操作系统。至于更复杂的操作系统内核构建本文没有多说，各种常用的操作系统内核都有很多资料可以查阅，不过我会向大家推荐一本在网上公开但没有出版的书——《底层工作者手册之嵌入式操作系统内核》，这本书详细阐述了操作系统内核的构建方法。

本文中提到的各个工程的代码作为附件可以从网上下载到：http://download.csdn.net/detail/nicekwell/6668033

本文不足以成书，更适合作为分享经验的帖子，通过这样类似于书的排版可以使调理更加清晰。作为一个高考语文不及格的人，希望文章的内容配得上这种比较正规的形式。

当然还请保留作者信息：

作者：nicekwell

主页：http://www.nicekwell.net

CSDN：http://my.csdn.net/nicekwell

目录

第一篇 单片机编程

第二篇 操作系统

第一篇

单片机编程

第 1 章 主函数顺序调用

这应该是最常见的结构了吧，学过单片机的都知道在 main 函数里面的那个“while(1)”。

笔者也对这个 while(1) 印象深刻，因为它让我明白了单片机程序运行的归宿就在这。在不考虑中断的情况下，整个单片机的最根本任务就是这个 while(1) 循环。在此称它为“主循环”，认为 main 函数及其调用的所有子函数（以及子函数再次调用的函数……）都在一个“主进程”里。

1.1 主函数顺序调用的一般结构

这种结构基本上都是在main函数开始完成一些初始化，然后在主循环里周期性地调用一些函数：

```
void main()
{
    /*模块初始化*/
    while(1)
    {
        Fun1();
        Fun2();
        ……
    }
}
```

在初学单片机时，大部分精力都放在单片机和各个模块的驱动上，所以在开始相当长的一段时间里采用的都是这种程序结构。

而 Fun1、Fun2……这些函数完成的功能也都是比较简单的，每个函数完成一个简单的小功能，然后顺序执行就可以组合完成某个功能。

需要强调的是，这些函数虽然功能简单，但是占用 CPU 资源不一定少，比如最简单的一个独立按键扫描程序：

```
sbit key=P1^0;
unsigned char keyscan()//返回0代表按下，1代表没按下
{
    if(key==0)    //说明按键按下
    {
        delay5(1);    //延时5ms去抖
        if(key==0)    //确认按键按下
        {
            while(key==0);//等待按键释放
            return 0;
        }
    }
    return 1;
}
```

注意到这个程序里有一个 5ms 延时函数，在延时的这段时间里单片机运行一些无意义的指令消耗时间。在此期间其他任务得不到运行，整个进程阻塞在延时函数这个地方。并且，如果按键一直按下没有释放的话，程序将停留在 while(key==0); 处。

1.2 主函数顺序调用结构的特点

首先，正如它的名称是“顺序调用”，任务之间的运行顺序是固定不变的，当然不可能有优先级区别，它只适合完成那些周期性循环的工作。不管它是不是缺点，总之这是第一点。

第二点正如 1.1 节所分析的那样，在这个任务运行的时候，其他任务是得不到运行的。并且如果这个任务由于某种原因卡住了，它将阻塞整个进程的运行。

而第二点到底可不可以接受要看具体要求。比如每个任务需要 5ms 的执行时间（内部可能有一些必要的延时函数），总共四个任务。如果整个单片机系统完成的只是简单的人机交互之类的功能，这是完全可以接受的，因为我们根本察觉不到每个任务在分开运行，在我们看来它们就是并行的。但是如果完成的是像通信协议之类的驱动的话，这是接受不了的，某个任务在执行的过程中可能其他任务有更迫切的需求。

在这里我想说的是，任务执行的并行与否是相对而言的，要根据具体的情况。如果我们的要求不高，当然用这种简单的结构是最方便的了，但是这种简单的结构也确实存在很多不足，有很多可以改进的地方。这是接下来几章要讨论的问题。

总之，在此我们明确一下这种结构的特点：

1、由主循环调用的任务的执行顺序是固定的。

2、由主循环调用的任务都只能单独地运行，进入一个任务，就不能处理其他任务。

3、这些任务执行时间一般会比较长（相对后面几章改造过的任务函数而言），某一个任务里面的延时函数会造成整个进程被延时。

在操作系统的书籍中把这种结构也叫做“前后台系统”，他们把主循环称作后台，中断称作前台。用操作系统的语言来说“应用程序是一个无限的循环，循环中调用相应的函数完成相应的功能”。由于函数是循环调用的，所以“在最坏情况下的任务级响应时间取决于整个循环的执行时间”。

本文在这里没有考虑中断，也就纯粹讨论的是后台系统。

第 2 章将介绍如何构建复杂的后台系统结构，并将程序框架和任务函数分立，明确系统的构建和任务函数的实现在整个单片机编程系统中的区别。而不要混为一谈盲目的构建系统。

在接下来的第 3 章到第 6 章将会发现程序的主体逐渐从后台转移到了前台，在转移的过程中会对任务函数进行改造，并明确这些改造的目的和好处，同时也能体会到这些改造其实就是在一步步通往操作系统，操作系统的产生是必然的结果。

第 2 章 界面函数结构

先无视“界面函数”这个词吧，这是我起的名字，因为我没找到这种结构的通用叫法。

这一章将介绍的是主循环调用任务函数的一种非常常用的结构。到目前为止，在主进程的构建方面，除了顺序调用我只用到过这一种构建方式，并且用得非常多。在 2011 年的电子设计大赛上，笔者就用了这种程序结构获得了安徽省第一的成绩，可见这种结构的威力。当年的题目是两辆车交替超车，在附件中会给出源代码，同时在本章也会以此为例介绍这种结构的特点。

在第 1 章中介绍到由主循环顺序调用其他函数的结构，这种顺序执行的方式是最简单的情况了。当情况复杂时，各个函数之间存在着某种逻辑关系，一个函数执行完毕后可能要根据具体情况决定下一个要执行的是哪个函数，同时也可能由于外部原因需要立马切换到另一个函数里。

这就产生了“界面函数”这种结构，之所以叫它界面函数，是因为它们就像是一个个界面一样，每个界面完成不同的功能。下面先介绍一下这种程序结构的一般形式：

2.1 界面函数一般结构

上代码：

```
unsigned char FlagPage=0;

void main()
{
    /*初始化*/

    while(1)
    {
        if(FlagPage==0)
            Task0();
        else if(FlagPage==1)
            Task1();
        else if(FlagPage==2)
            Task2();
        ……
    }
}

void Task0()
{
    /*Do Something*/
    while(1)
    {
        /*Do Something*/
        if(FlagPage!=0)
            return;
    }
}
void Task1()
{
    /*Do Something*/
    while(1)
    {
        /*Do Something*/
        if(FlagPage!=1)
            return;
    }
}
```

```
void Task2()
{
    /*Do Something*/
    while(1)
    {
        /*Do Something*/
        if(FlagPage!=2)
            return;
    }
}
```

可以看到，主循环其实不进行任何实际功能的处理，它完成的只是调用各个任务函数。对于比较大型复杂的系统，main 函数的主循环里根本不放要实际处理的代码，而是把所有任务函数归到一起，根据选择进入相应的任务函数，当处理完该任务之后又会回到主循环，由主循环再次分配任务。

此时主循环的作用就是调配任务(当然用来调配任务的主循环本身也是一个最基本的任务)，而在被调配的任务里面可能还会再次被该任务调配的子任务。

再来看看被调用的任务函数，这些函数已经不只是完成一些简单功能了，它并不是执行一些固定操作后返回，每个任务函数都有自己的一套控制逻辑，并且“不那么容易返回”。

这些任务函数同属于一个进程，但是同一时刻只有一个可以运行。当进入某个函数时，可以说进程被这个函数阻塞，其他函数得不到运行。但这也就是我们需要的效果，因为每个函数都有自己的一套控制逻辑，完全不需要考虑其他界面函数。

而在函数退出时，可以由该函数本身指定下一个要进入的函数，或者本来就是由于外部修改了 FlagPage 变量才导致该函数退出的。

这种结构是非常常用的，并且尤其适合那些有多种界面(或者说多种工作模式)的场合。

比如电子钟里可能有时钟界面、设置界面等，从一个界面进入到另一个界面都是由按键控制的。如：在时间界面按下设置键进入到设置界面，按下返回键就进入到 logo 界面。这一个个界面也就是任务函数，只不过这个任务函数不会自动跳出，而是根据按键情况决定是否跳出、并通知主循环要跳到哪。(每个界面里也会有选择地对其他进程提供的信息进行处理，比如时间界面就会对时间累加进程所提供的时间信息进行显示，同时也会对按键扫描进程提供的按键序号进行处理；而 logo 界面只会对按键信息进行响应，忽略时间进程提供的时间，但是时间进程仍然在运行，不然时间岂不是不准了。 这些进程都是由定时器进行的，在后面会说。)

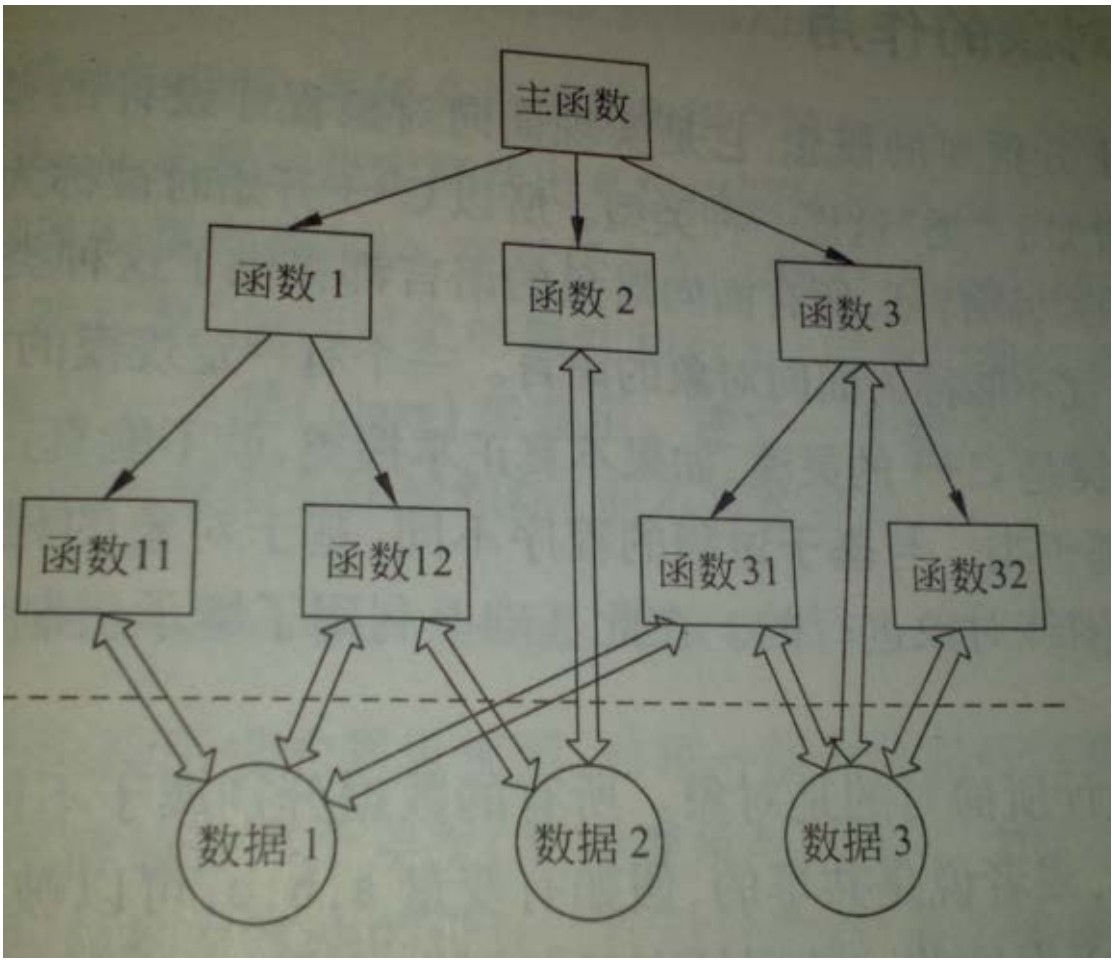
再如电子设计大赛的小车程序就是，它分为领跑模式、跟随模式、超车模式等，每种模式就是一个界面函数，只不过在运行的过程中不允许有外部操作，完全由每个界面函数本身根据采集到的当前状态信息决定是否退出，并结合一些其他全局变量判断下一个要进入的模式。

以上把这个典型结构做了一个简单介绍。

2.2 更高的角度分析这种结构

在 2.1 节分析了一下界面函数这种典型结构，在此我想借这种结构分析一下整个系统的构成。

把上述系统用下面这个图表示（取自谭浩强 C++程序设计 P227)：



这里面所有函数都是由主函数调用的，属于主进程，并且这些函数也都体现出了系统结构。

例如在函数 1 里面想进入函数 2，不是直接调用函数 2，而是先返回函数 1，再由主循环分配到函数 2。

正如前面所说，这种程序结构特别适合于多种“界面”的功能。一般情况下，主进程不会停留在主循环里，而是偶尔退出到主循环重新分配下一个将要进入的函数，大部分时间会停留在某个界面函数里。

此外，这些函数之间有一些公共变量，这些变量的作用就是被各个函数使用，甚至用于函数间通信，辅助完成这些函数之间的逻辑结构的构建。比如 1.1 节中的那个重要的 FlagPage 变量，这个标志变量就指明了当前工作于哪种工作模式下，任何函数（包括中断进程中的函数）都可以通过改变此变量来切换工作模式。。

也有一些与函数对应的用于完成特定功能的变量。比如用于数码管或者显示屏显示的现存，这些显存是有特定用处的，一般其他函数不会使用（但确实是公共变量，是可以被使用的）。

上面是把变量进行了分类，与之对应地我们也把函数进行分类。

图中画出的函数都是所谓的“界面函数”，是用于完成某一特定任务的函数，一般进入这个函数后主进程就会停在里面，当达到特殊目的后返回。而这些“界面函数”也会不断地调用其他函数完成功能，比如延时等。

这些被界面函数调用的函数把它们称作“工具函数”。这些功能函数中有一些是公用的，比如延时函数，很多地方都会用到。而也有一些是某一个界面函数才会用到的，用于完成这个特殊功能的函数，比如用于完成屏幕显示用到的屏幕驱动函数、字符显示函数等，这些函

数在其他地方几乎不会被调用。

以上这些分类有什么意义吗？貌似都是废话，这些是在编程中自然产生的，但不妨在此将它们明确分类一下，整个系统都有哪些东西呢：

1、整体的程序框架是由各个界面函数和少数关键的全局变量构建起来的。这是构成系统的主体框架。

2、每个界面函数在完成特定功能时，会携带一些为自己服务的“私有的”变量和函数。

3、为整个框架服务的还有一些常用的变量和函数，它们完成的是一些通用功能，可以把它们理解为“库函数”。

需要提一下，到目前为止讨论的程序结构都没有考虑中断，或者说到目前为止都是把中断当作程序运行过程中的特殊情况处理，并没有融入到整个系统构建的问题内讨论。但是中断确实是存在的。这里想说的是：

主进程所构建的程序框架会有一些服务函数，这些服务函数在中断处理程序里也可能会被用到。比如主进程在调用延时函数时被中断，而中断处理程序也调用了该延时函数，这会怎么样呢？

这就是所谓“函数重入”问题，在此不想过多地讨论该问题，要知道 51 单片机的编译器 keil 默认是不支持函数重入的（事实上，如果你这样做了编译器会发出警告的）。

所以，至少对于 51 单片机编程来说，在遇到多个进程编程时要注意这样一个原则：中断处理程序不要调用到可能被中断的函数。必要时可以为中断进程单独写一个服务函数，函数内容可能跟主进程中的某个函数一模一样，但这样可以避免上述问题。

以上就是对“界面函数”这个典型结构的介绍，也分析了整个系统的构成。但是，正如上面说过的，这些都是对一个进程的结构的讨论，并没有涉及到中断。

第 3 章 定时器分配任务

在前两章的内容里都没有考虑中断，本章将引入定时器。为什么要引入定时器呢？因为一些功能的完成必然会需要定时器。

每种结构的产生都有一个项目背景，本章的背景是一个电子钟：

这是我在 2012 年 6 月制作完成的，一方面想找个伴陪我一起走过考研路，另一方面想看看一个单片机的寿命到底有多长。（目前的情况来看，连续运行了一年半之后单片机没有出现什么问题，但是 1602 的显示效果远不如开始了）

本章将以这个电子钟为背景介绍用定时器分配任务的程序结构，源代码也会在附件中给出。

3.1 用界面函数构成的基础框架

这个电子钟的大体硬件租成有：按键、1602 显示屏、18b20 温度传感器、电源管理模块（两个 AD 转换和一路 PWM 输出），由于没用时钟芯片，采用的是 8 位自动填装定时器每隔 200us 一次中断来计时的（很准哦）。

任务处理方面，除计时和日期计算以外，要处理的任务还有：1602 显示、按键扫描、温度采集和后来添加的电源管理（控制电池充放电）。

完成的显示界面有时间显示界面、时间设置界面、电源管理界面 和 logo 界面。每个界面完成不同的功能，时间显示界面就是上图所示的样子，时间设置界面用来完成时间的设置，点晕管理界面用来查看当前电池状态以及设置一些充放电参数，logo 界面显示 3 秒后自动跳转到时间显示界面。

这些界面是如何完成的呢？没错！它就是在主进程里用界面函数完成的。每个界面函数

先初始化本界面显示，然后在实现本界面具体功能：

```
void Page1()
{
    /*显示初始化*/
    /*其他初始化*/

    while(1)
    {
        /*本界面任务*/

        if(FlagPage!=1)
            return;
    }
}
```

具体而言，logo 界面的代码如下（因为 logo 界面代码最短）：

```
void logopage()    //logo界面，定时器还在工作，只是停留在logo界面s中后返回
{
    /*下面是显示初始化*/
    unsigned char i;
    code char logo0[]="hello world!    ";
    code char logo1[]="    hello nicek!";
    wcom(0x80);
    for(i=0;logo0[i]!='\0';i++)
        wdat(logo0[i]);
    wcom(0x80+0x40);
    for(i=0;logo1[i]!='\0';i++)
        wdat(logo1[i]);

    timenum=15000;     //准备延时3s，这个变量会在定时器里每200us减1
    while(timenum)     //等待延时结束，等待过程中仍然对一些按键进行相应
    {
        switch(keynum)
        {
        case 5:  //按下返回键
            flagpage=0;   //返回后跳转到界面0
            return;  //返回
        case 6:  //按下灯光间
            led=0;   //点亮背光灯
            timenumlight=5;    //点亮5s，时间到了之后会在定时器中自动关闭灯光
            keynum=7;//按键响应结束，标志没有按键按下
            break;
        }
    }
    flagpage=0;   //返回后跳转到界面0
}
```

有了这些界面函数，也就构成了整个系统的基本框架，但是到目前为止还是没有用到定时器。下面就以“时间显示”界面为例，分析在这个界面中如何结合定时器完成相应功能。

3.2 结合定时器编程分析

先来讨论一下在时间显示界面里需要做的这么几个任务：

1602 需要 500ms 刷新一次（小时和分钟之间有一个冒号“:”需要 500ms 闪动一次，因为不显示秒，所以如果分钟发生了变化整屏也要刷新一次）；

按键扫描 5ms 扫描一次（如果放在时间界面的主循环里进行按键扫描的话就不用考虑这么多，但是那样有很多缺点，我这里是把按键扫描当做一个固定的进程，其他所有任务都能用利用这个扫描结果）；

电源管理 500ms 检测一次；

温度采集 500ms 一次，但比较复杂，涉及到“任务分割”问题（自己起的名字哈），后面单独讨论。

很自然地想到利用定时器计时来进行，那么定时器会以什么样的工作方式来调配整个系统呢？

首先，定时器 200us 一次中断，肯定要有一个变量累加，当累加到 5ms 时，进行一次按键扫描；

然后继续累加，每累加到一个 5ms 都要进行一次按键扫描，当累加到 500ms 时进行一次按键扫描、1602 刷新显示、电源管理，（温度采集暂且忽略）。

那么，我们可以直接把这些代码写入中断处理程序吗？

比如按键扫描我可以把这段代码写入中断处理程序吗：

```
for(i=0;i<=6;i++)  //总共7个独立按键
{
    if(P1&pow2[i]==0)  //pow2[i]就是2的i次方
    {
        delay5(1);    //延时5ms，以确认是否真的按下
        if(P1&pow2[i]==0)
        {
            keynum=i;
            break;
        }
    }
}
```

这是当然不行的，执行完这段代码所需的时间就超过 5ms 了，而定时器是 200us 一次中断。如果把这段代码放到时间界面主循环里是可以的，但是这样的话在其他界面就不能使用（除非也加入相同的代码），正如我上面所说：那样有很多缺点，我这里是把按键扫描当做一个固定的进程，其他所有任务都能用利用这个扫描结果。

所以这里要提出一个定时器分配任务的程序结构原则一：定时器中断里的代码执行长度一定不能超过定时器中断时间。

这不是废话吗，肯定的啊。所以我们要想想办法，把按键扫描程序改成了如下：

```
static unsigned char reslast;    //保存上次扫描结果，0-没按下，1-按下
unsigned char res;
unsigned char i;
numforkey=0;
res=P1;
for(i=0;i<=6;i++)
{
    if(((res&pow2[i])==0)&&((reslast&pow2[i])!=0)) //这次按下，上次断开
        break;
}
//从这里出来，如果i==7则表示没有按键按下，i<=6的任意一个值表示那个键被按下了
keynum=i;
reslast=res;
```

这段代码里面没有延时，执行一次是很快的，而且也可以很好地完成按键扫描，比上面的那种延时扫描更有优势（不占用资源，而且稍加改造可以识别同时按下多个按键）。

所以，把原则一加上一句，定时器分配任务的程序结构原则一：定时器中断里的代码执行长度一定不能超过定时器中断时间，要想办法把任务改成不占用定时器时间的结构，给主进程让出更多的时间。关于把“任务改成不占用定时器时间”这个问题会在后面的“占用式与非占用式程序结构分析”里详细讨论。

按键扫描在中断处理程序里算是可以完成了，但是 500ms 时的那么多任务在 200us 以内可以完成吗？好像有可能是可以的，就算这里可以，以后其他地方肯定会遇到处理量很大，在一个中断里完成不了的任务。而且我在电子钟里面把 1602 刷新就没有放在定时器中断里。

此时，就是定时器分配任务的程序结构原则二：当节拍时间到来时，要处理的任务真的很多，可以通过标志变量通知主进程执行。但通知让主进程做的事对实时性要求不能太高。

比如我在程序里设置了一个 flag500ms 标志变量，当此变量为 1 时标志到了 500ms，时间界面的主循环检测这个变量，当发现这个变量为 1 时就执行 1602 刷新。(按键扫描和电源管理由于在任何界面都会用到，所以把它们独立出来，放在定时器中断里进行)。

这里的 1602 刷新对实时性要求不高，所以可以用定时器通知主进程执行。

下面要说的就是比较复杂的温度采集了，上面为什么没讨论它，就是因为它比较复杂。它既不能满足原则一（在一个定时器中断时间内完成）也不能满足原则二（对实时性要求不高）。

温度传感器用的是 18b20，由单总线协议决定了对它进行一次读写大约需要 18ms，而且读写它对实时性要求也很高。

这里隆重推出一个很高深的自己起名的概念——“任务分割”。顺便引出定时器分配任务的程序结构原则三：当既不满足原则一又不满足原则二，即既不能在一个定时器中断时间里完成又对实时性要求很高的任务，对它进行任务分割。

下一节将详细介绍任务分割的概念，以及如何对任务进行“分割”。

3.3 任务分割

所谓任务分割就是把不能在一个定时器中断时间里完成的任务分割成多个可以在一个定时器中断时间里完成的任务。在这里，把分割完成之后的任务函数仍然放在主进程里。

要完成任务分割，首先需要让定时器的计时功能可以被外部使用，设一个全局变量 TimeNum，然后在定时器中断处理程序里让 TimeNum 自减：

```
unsigned char TimeNum=0;
void t0_int() interrupt 1
{
    /*Do Something*/

    if(TimeNum!=0)
        TimeNum--;
}
```

这样，外部就可以通过 TimeNum 变量使用定时器的计时资源了。在主进程里只要这么做就行：

```
/*Do Something*/
TimeNum=100;  //准备等待个定时器中断周期
while(TimeNum)     //等待
{
    /*Do Something*/
}
/*继续 Do Something*/
```

可以利用定时器延时之后就可以对主进程里的长任务进行分割了，分割方法就是把原来顺序执行的任务函数，找到合适的节点，在节点处进行合适时间的延时。

这个步骤做起来是比较麻烦的，因为要进行任务分割的任务一般对时间精确性要求是比较高的（否则直接放到主进程就可以了），需要充分了解运行过程。既要保证每个节点之间的运行时间小于一个定时器中断周期，又要保证任务时序的正确性。

需要强调一下，任务分割这个方法是个下策，是在没办法的情况下才进行这种处理，我也只在这个电子钟工程的温度测量用过。

这种情况在操作系统中也是不好处理的，当某个不能被中断的函数的运行时间确实长于时钟节拍周期的话，也只有先关闭中断，等该函数运行结束后再打开中断。操作系统中把这种函数叫做“临界段”代码。而在电子中这个例子中，由于要进行精确的计时，是不可以关闭中断的，只有进行任务分割（当然也有其他的处理方式，加个协处理器什么的）。

以下是电子钟里进行温度采集的函数：

```
void reftemp()     //读取并刷新温度。          //技术：原本的温度扫描是一个连续完整的函数，而这个函数完成的时间大约是18ms。但是由于定时器绝对不能停止工作，而定时器的中断时间是200us，在一个中断周期内不能完成所有的工作。定时器会对温度扫描造成影响。
//这里的解决方案是把原来的温度扫描函数中的延时全部用定时器及时处理，也就是把整个温度读取函数分割成许多可以在一个定时器中断周期内完成的程序片段，分成多个定时器周期完成。
//这种方法以前是没有用到过的，这是第一次用。测试完成，帅气！这个方法竟然成功了！！
{
     float temperature; //保存温度信息
     unsigned int temp;
     temp=gettemperature(); /*这个函数就是已经被任务分割过的函数*/
     if((temp&0x8000)!=0)    //是负的
     {
          sign=1;
          temp=~temp;
          temp+=1;
     }
     else
          sign=0;
     temperature=temp*0.0625;     //获取温度的浮点数
     temp1=((int)temperature)/10;      //获取温度的十位
     temp0=((int)temperature)%10;      //获取温度的个位
     temperature*=10;
     tempdp=((int)temperature)%10;     //获取温度的一位小数

     wcom(0x80+0x40+8);
     if(sign==1)    //负的
          wdat('-');
     else //正的
          wdat('+');
     wdat(0x30+temp1);
     wdat(0x30+temp0);
     wdat('.');
     wdat(0x30+tempdp);
     wdat(0xdf);    //写入℃的圆圈
     wdat('C');
}
```

3.4 定时器分配任务程序结构总结

1、 整个系统有一个主进程：main 函数的主循环及其调用的所用任务函数，以及所有任务函数调用的子任务函数。

这个主进程的特点是所有函数都在一个函数调用链里，运行时精力只能放在一处；优先级低，任何中断所调用的任务都会使其停止工作。

2、定时器也可开辟一道进程，所有由定时器直接调用的任务都属于这个进程。

定时器进程可以通过一些标志变量通知主进程进行某种动作，最常用的控制方法是用定时器产生节拍信号，通知主进程进行相应动作;

同时，定时器也可以直接调用一些函数，在定时器中断处理程序里完成任务。所有由定时器直接调用的程序都属于定时器进程，优先级高于主进程;

用定时器分配任务有一下三点原则：

定时器分配任务的程序结构原则一：定时器中断里的代码执行长度一定不能超过定时器中断时间，要想办法把任务改成不占用定时器时间的结构，给主进程让出更多的时间。

定时器分配任务的程序结构原则二：当节拍时间到来时，要处理的任务真的很多，可以通过标志变量通知主进程执行。但通知让主进程做的事对实时性要求不能太高。

定时器分配任务的程序结构原则三：当既不满足原则一又不满足原则二，即既不能在一个定时器中断时间里完成又对实时性要求很高的任务，对它进行任务分割。

3、整个系统来看有两个并行的进程——主进程和定时器进程。主进程一次只能执行一

个任务，而定时器进程由于任务一般比较小（如按键扫描、计时、数码管扫描等），所以认为定时器进程的任务也一并完成了。

看上去就像是多个进程在同时运行，这些进程之间可以通过公共变量进行通信，比如节拍时间的标识变量、计时产生的时间、按键扫描结果变量 keynum 等，所有其他进程可以有选择地对这些标识变量进行响应。类似于进程间通信。

用定时器命令主函数执行任务的原因有两点：1、利用定时器的时钟节拍使主函数也可以节拍性地执行任务。2、利用主函数构建的逻辑结构。

对于第 1 点，该任务的实时性确实会受影响，因为毕竟主函数是用查询方式查询标识变量的。

但是第 2 点带来的好处也是非常大的。还拿电子钟举例来说吧，电子钟里面的各个界面之间的逻辑是通过主函数构建出来的，定时器在任何界面都会中断，并且在定时器中执行的任务会通过标识变量向主函数发送信息（比如当前时间、按键扫描结果(我的按键扫描是放在定时器里执行的)等），虽然在各个界面向主函数发送的信息是一样的，但是主函数中的各个界面对这些信息的反应却是不同的（比如各个界面对同一按键的反应是不同的，对定时器提供的时间所做的的反应也是不同的）。 这些都依赖于主函数所构建出的逻辑结构。

关于这种结构的介绍暂且到此吧，下面将引出一个新的问题。

注意在 3.2 节的分析中，有些任务是直接放在定时器里执行的，这些任务都有一些共同的特点——执行时间短。

执行时间短带来了什么？在 1.2 节有这么一句话:“任务执行的并行与否是相对而言的。”，在本节的总结中还有这句话：“定时器进程由于任务一般比较小（如按键扫描、计时、数码管扫描等），所以认为定时器进程的任务也一并完成了。”

定时器的中断处理函数的执行肯定不会被主进程阻挡（只有它阻挡人家），这里面的任务全都可以看成是并行的多个进程，它们各自完成不同的功能，把自己的运行结果作为资源供其他进程使用。

在第 5 章中将详细分析这种结构，并称它为“准操作系统”（又是自己起的名字），在那一章，我们将会明确我们到底在追求什么，我们要追求的结构到底是什么样的？

不过在此之前，先要补充一点理论知识，这就是下一章的内容。

下一章将会分析和明确那些“完成某个任务的函数”到底应该做些什么，哪些东西是有用的。并对这些函数进行改造，分析改造成不同形式的函数有什么样的特点。

第 4 章 占用式与非占用式程序结构分析

本章内容曾公开在 CSDN 论坛上（事实上第 2 章也第 3 章也是）：http://bbs.csdn.net/topics/390504115。

在此我直接将其复制过来。

4.1 什么是占用式程序

一个进程在一个时刻只能处理一个任务。

每个任务是为了完成一个功能，如果这个功能的实现过程是一直占用进程处理资源的话，就称这个任务函数是占用式程序结构。

最常见的占用式程序结构就是延时函数了，比如最常用的 5ms 延时函数

```
void delay5(unsigned char n)
{
    unsigned int i;
    for(;n>0;n--)
        for(i=4700;i>0;i--);    //12MHz, 1T
}
```

在完成 5ms 功能过程中是一直占用调用它的进程处理资源的，在此期间不能进行其他任务。

还有一个很常见的占用式程序——数码管扫描，不过在这里我不举数码管扫描的例子，而举这次在一个项目中使用的 8*8 彩色点阵屏的扫描程序：

```
void refresh7()
{
    unsigned char r;
    for(r=0;r<8;r++)
    {
        //扫描红色
        DPw = ~(0x01<<r);  //修改完了再导通指定行
        DPr = ~vm7r[r];//送入R灯IO接口显示
        DELAY7 (light7);//显示时间长度
        DPw=0xff;
        DPr=0xff;
        DPg=0xff;
        DPb=0xff;
        DELAY7 (32-light7);//灭灯时间长度
        //为了简洁，这里把绿色和蓝色的扫描程序省略，它们的结构和红色扫描是一样的
    }
}
```

这个函数是 7 色模式下的屏幕扫描程序，调用一次此函数会把整个屏幕扫描一遍。

r 代表行数，r 循环 8 次代表屏幕的 8 个行；在每次循环里，先导通对应的行和需点亮的灯，然后延时 light7 个单位，再关闭所有显示，再延时 32-light7 个单位。

4.2 占用式程序的缺点

占用式程序最大的缺点就是执行时间太长，耽误对其他任务的响应。另外就是资源浪费，很多时间浪费在执行中的延时上。

当然，可以在这些占用式程序中嵌入其他代码以及时处理其他任务，但是这样会造成程序结构混乱，嵌入的其他代码还会影响本程序的执行。如果嵌入的代码功能简单还好，如果功能复杂，尤其是当嵌入的代码也是占用式的，就会严重影响程序执行速度。

4.3 对占用式程序的改造

在此以该项目的扫描程序为例，对其进行改造。

首先，每次调用就扫描 8 行，耗时太长，现将其改成每次扫描一行：

```
void refresh7()
{
     static unsigned char r=0;
     //扫描红色
     DPw = ~(0x01<<r);  //修改完了再导通指定行
     DPr = ~vm7r[r];//送入R灯IO接口显示
     DELAY7 (light7);//显示时间长度
     DPw=0xff;
     DPr=0xff;
     DPg=0xff;
     DPb=0xff;
     DELAY7 (32-light7);//灭灯时间长度
     //为了简洁，这里把绿色和蓝色的扫描程序省略，它们的结构和红色扫描是一样的

     r++;
     if(r==8)
          r=0;
     }
}
```

用一个静态变量 r 来记忆行数，这样每次调用此函数只需扫描一行，执行速度是原来的 8 倍，可以比较快地响应其他任务了。

但是这样还不够，每次扫描都会扫描三种颜色，时间还是有点长，下面再次改造，改为每次只扫描一种颜色：

```
void refresh7()
{
     static unsigned char r=0;
     static unsigned char flagrgb=0;  //当前需要点亮的颜色，0-R,1-G,2-B
     flagrgb++;
     if(flagrgb==3)     //说明三种颜色都扫描完了
     {
          flagrgb=0;     //从红色开始扫描
          r++;      //开始扫描下一行
          if(r==8)  //如果发现行都扫描结束则从第行开始扫描
               r=0;
     }

     switch(flagrgb)
     {
     case 0:   //扫描红色
          DPw = ~(0x01<<r);  //修改完了再导通指定行
          DPr = ~vm7r[r];//送入R灯IO接口显示
          DELAY7 (light7);//显示时间长度
          DPw=0xff;
          DPr=0xff;
          DPg=0xff;
          DPb=0xff;
          DELAY7 (32-light7);//灭灯时间长度
          break;
     case 1:   //扫描绿色
          /*省略代码*/
          break;
     case 2:   //扫描蓝色
          /*省略代码*/
          break;
     }
}
```

改造完成之后，执行时间再次缩短，变成了刚才的 1/3。

这下还没完，我们发现每次扫描中都有延时，延时过程中什么也不做，这是极大的浪费，我们需要再此改造，把延时去掉：

```
void refresh7()
{
    static unsigned char r=0;
    static unsigned char flagrgb=0;  //当前需要点亮的颜色，0-R,1-G,2-B
    static unsigned char num=0;
    num++;
    if(num==32)
    {
        num=0;
        flagrgb++;
        if(flagrgb==3)     //说明三种颜色都扫描完了
        {
            flagrgb=0;     //从红色开始扫描
            r++;      //开始扫描下一行
            if(r==8)  //如果发现行都扫描结束则从第行开始扫描
                r=0;
        }
    }

    if(num<light7)     //说明需要点亮
    {
        switch(flagrgb)
        {
        case 0:  //扫描红色
            DPw = ~(0x01<<r);
            DPr = ~vm7r[r];//送入R灯IO接口显示
            break;
        case 1:  //扫描绿色
            DPw = ~(0x01<<r);
            DPg = ~vm7g[r];
            break;
        case 2:  //扫描蓝色
            DPw = ~(0x01<<r);
            DPb = ~vm7b[r];
            break;
        }
    }
    else //说明不需要点亮
    {
        DPw=0xff;
        DPr=0xff;
        DPg=0xff;
        DPb=0xff;
    }
}
```

现在，这个函数中没有任何延时和循环，执行所消耗的时间是非常少的，可以很快地响应响应其他任务。

4.4 改造的本质

上面我们对这个项目的扫描程序进行了“三大改造”，分别是：1、各个行扫描的分离；2、各个颜色扫描的分离；3、延时函数的消除。

这些改造的本质都是对原程序的分割，把一大坨程序分成多个步骤分别执行，以减小耗时，提高对外部的响应速度。

还记得在第一章说的主函数顺序调用吗？最后说过这样一句话：“在最坏情况下的任务级响应时间取决于整个循环的执行时间”，而通过这样的改造之后，其实就是在缩短这个循环的时间。

但就整个进程的执行来看，有效代码的比例是降低的，包括上面“三大改造”的第三点延时函数的消除，看上去是消除了延时函数，提高了执行效率，但从“扫描一次整屏”这个任务来看，其执行的代码量反而是增加的。（但并不是所有的改造都一定会使效率降低，有些改造确实可以达到“消除延时函数”的目的）

那为什么还要对其进行改造呢，见下节分析。

4.5 非占用式程序结构的优势

1、非占用式程序相比于占用式程序，增加了一定的代码，虽然会使整体效率降低，但是提高了各个任务之间的切换速度，可以对各个任务都能很快地响应。这点类似于操作系统，虽然降低了效率，但是各个任务间的快速切换可以达到各个任务“并行处理”的效果，光是这点的好处就已经很大了。

2、非占用式程序结构可以放进定时器

第 3 章已经发现用定时器分配任务的好处，有些简短的代码可以直接放进定时器里。

在改造之前的扫描程序是不适合放在定时器中断处理程序里执行的，因为太长，可能还没执行完就来了下一个中断。就算勉强执行完了，留给主进程处理其他事情的时间也不多了。

而改为非占用式之后，可以在中断处理程序里直接调用扫描程序，它会很快地执行完，然后有充足的时间留给其他任务。

3、非占用式程序并不是一定会降低效率。

就拿“三大改造”的第三点说明，它虽然形式上消除了延时函数，但是每次调用此函数时对 num 变量的处理，以及有其产生的相关判断语句，总的代码量比原来的要多。

但是，这真的就仅仅是这样了吗？改造之前的函数，执行完退出之后所有的 led 全是熄灭的，只有在此函数执行过程中（延时阶段）才会点亮（传统数码管扫描亦是如此）。

而改造之后的函数，它的功能就是指定一下每个灯的亮灭，然后立马退出，在执行其他任务的过程中该点亮的灯是在点亮的状态。这样就提高了整体的亮度，在执行其他任务的过程中，从某种意义上说也是在执行当前任务。

这可能还不能太清楚地说明问题，下面再举一例，传统的按键扫描一般是这样：

```
if(key==0)     //key是某个引脚
{
     delay5(1);
     if(key==0)     //确认按键已按下
     {
          /*do something*/
     }
}
```

这段代码也是很浪费时间的，中间有个 5ms 延时白白浪费。

通过对它改造之后，结合定时器，可以几乎完全地把这 5ms 时间省出来，把如下代码放进定时器中断处理程序：

```
static unsigned char keylast;     //保存上次的按键值
if(key==0 && keylast==1)     //检测到一个下降沿
{
     /*do something*/
}
keylast=key;
```

这段代码每 5ms 执行一次，而执行的时间是非常短的，剩下有大量的时间可留给其他任务。

结合定时器进行改造，是真的可以把占用式函数的延时时间节省出来的。

4.6 非占用式程序的一般结构

非占用式程序将占用式程序分割执行，需要用到静态变量对当前步骤进行记忆，其一般结构如下：

逻辑变量计算就是根据任务功能构建出一个合理的逻辑结构。

对逻辑变量的响应就是对构建好的逻辑结构的结果的响应和执行。

4.7 吐槽

最近开发这个项目写了不少程序，以前在写程序的过程中就隐约发现了所谓占用式和非占用式程序结构的区别，程序写多了肯定能发现问题，但是如果不停下来总结，而是一味的开发，那是不会有进步的。

组织庞大的程序需要正确的理论指导，学习很多的知识也需要进行总结。知识点太多不可能学完，只有将他们提升到理论层次，将这种思维方式刻在脑子里才能灵活地运用，并从容地接受新的知识。

第 5 章 定时器执行任务

先来回顾一下，在第 2 章中介绍了界面函数结构，它的思想是主进程为主体，外部的按键等作为特殊情况单独处理。但是当接触到的程序更复杂时，尤其是当程序里还要进行精确定时时，用纯粹的单进程结构已经满足不了要求了，这就进一步产生了第二种结构——定时器分配任务。

定时器分配任务结构在主进程结构基础上开辟了一个定时器进程，在这个进程里进行按键扫描任务、计时任务等。此时这些任务是不会被中断的，定时并且精确地每隔一段时间执行一次。当时的看法是：这些定时器里的进程完成任务后把结果保存，主进程可以选用这些结果进行处理。

此外定时器还要进行一个特殊的功能——给主进程下达命令，通知主进程进行某种动作。这个功能的本质就是向主进程提供了时间信息。

这种结构已经结合了定时器，并且已经把一些简短的代码直接放到了定时器中断处理程序里了，但还有相当一部分代码放在主进程里。不是说不能放在主进程里，而是当时没有明确出定时器中的各个进程是如何形成的，这些定时器中的任务有什么更深刻的特征。

本章就是专门讨论这些放在定时器里执行的任务。

这种结构当然也有产生的背景，它源于一个“温度控制系统”的项目，具体内容大概有：数码管扫描、按键扫描、时间计时、蜂鸣器控制、温度控制。

由于这是给公司做的项目，所以和上一章的背景工程一样，不能公开源代码，不过不需要源代码也完全可以理解这个结构。

这个项目并不复杂，功能要求很明确，没有多个工作模式和界面。按开始的想法，这里面除了计时任务需要定时器外，其他任务都可以放在主进程里完成。但是这样的话可能就会出现各个任务之间的相互干扰，比如按下按键时进程被阻塞，数码管扫描就无法得到运行。

所以，在做这个项目时我尝试了另一种方法，那就是把这些任务全部放在定时器中断处理程序里，由定时器驱动每个任务的运行，主循环什么也不做。

完成之后事实证明这种结构效果很不错，并且体现出了很多操作系统的思想。下面就来分析一下这种结构。

5.1 定时器执行任务的程序结构

【时间分配系统】

这种结构的任务需要在定时器中断里执行，而定时器中断的时间不一定是任务想要调用的时间，并且不同的任务的调用时间可能不同。所以肯定有一个时间分配系统，这个系统在特定的时间调用不同的任务函数。

比如在这次的工程中，51 单片机的定时器 8 位自动填装模式的定时时间不会太长，定时时间设为 250us。而按键扫描、数码管扫描、蜂鸣器控制、温度控制、时间控制，这几个任务的执行时间是不一样的：

任务名称	调用时间
按键扫描	5ms 一次
按键处理	5ms 一次

数码管扫描	250us 一次
蜂鸣器控制	5ms 一次
温度检测	1s 一次
温度控制	1s 一次
计时和时间控制	1min 一次

需要强调的是，有些任务的执行调用时间周期虽然是相同的，但是不一定是同时调用的。比如按键扫描和蜂鸣器控制都是 5ms 执行一次，但是它们却不在同一次定时器中断内执行，因为定时器中断周期是 250us，如果在一个中断时处理多个任务可能时间比较长，不能在 250us 内处理完，所以将它们错开分到不同的时间段内执行，但是周期仍然是 5ms。

时间分配代码如下：

```
void t0() interrupt 1  //250us一次中断
{
    static unsigned char numto5ms=0; //用于5ms计时
    static unsigned char numto1min=0;    //用于1min计时
    /***数码管扫描任务***/ //数码管扫描，每次中断都执行

    numto5ms++;
    if(numto5ms==5)    //5ms一次，温度检测
    {
        /***温度检测任务***/
    }
    else if(numto5ms==10)  //5ms一次，蜂鸣器发生
    {
        /***蜂鸣器发声任务***/
    }
    else if(numto5ms==15)  //5ms一次，按键扫描
    {
        /***按键扫描任务***/
    }
    else if(numto5ms==20)  //5ms到了，进行分钟判断，此时numto5ms要清
    {
        numto5ms=0;
        numto1min++;
        if(numto1min==12000)    //1min到了
        {
            numto1min=0;
            /***时间处理任务***/
        }
    }
}
```

【任务执行函数】

任务执行函数就是在定时器里调用的用于完成某种任务的函数。

这个函数的具体特点将在下节介绍，因为很重要。

5.2 定时器里面任务函数的特点

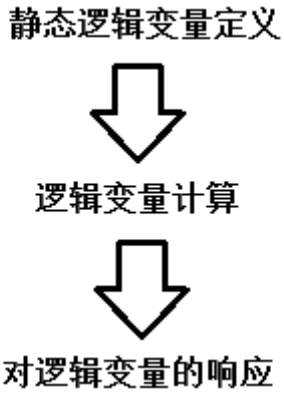
首先分析一下这种任务函数的特点和要求：

1、这些函数由定时器调用，所以对于它们的调用时间是很精确地每隔固定时间调用一次。

2、由于这些函数是放在定时器里，所以这些函数必须简短，不能占用过长时间，必须可以在一个定时器中断周期内全部处理完。

3、与在主进程连续执行的任务函数相比，这种函数的调用是周期、间断的，这种周期间断性的调用，决定它自己必须具有记忆以前的状态的能力，只有这样才能在本次被调用时决定应该进行什么样的操作。

在上一章“占用式与非占用式程序结构分析”中已经明确了非占用式程序结构的优势，并且也明确了其内部结构：



有了上一章的基础就好理解这些在定时器里的任务函数了，实际上这些任务函数和在主进程连续执行的任务函数相比，就是把它们改为了非占用式程序放在定时器里执行。

4、由于要保证每个任务都要在很短的时间内执行结束，所以就要求每个任务不能阻塞进程，不论这个任务当前处于什么情况，应当执行一次之后立马退出。

这样，不论某个进程的执行情况如何，其他进程绝对都会继续执行，也就把各个进程独立开了，保证每个进程都会得到及时的执行。

5、各个进程相对独立，但各个进程间也有通信。比如按键扫描进程把按键码传递给其他进程；蜂鸣器进程通过变量接收外部下达的响铃指令。

要注意一个特点：由于定时器里面的任务函数是被周期性的调用的，所以如果想使用某个进程的功能，必然不可能像以前那样通过调用函数来实现（因为它本身就一直在被调用着），必然是通过这个进程对外设置的接口变量来实现，这也就是与进程通信的过程。

5.3 过程任务的定时器化

这里讨论如何把一个过程化的程序改成定时器化的程序。

貌似没有找到通用的方法，可以确定的是定时器化的任务结构肯定就是像非占用式程序结构那样。

下面举几个定时器化的程序的例子。

数码管扫描

```
void smgdisp()
{
    static unsigned char n;
    n++;
    if(n==8)
        n=0;

    smgndisp(n,?);
}
```

用静态变量 n 记忆点亮的数码管序号，这样轮换点亮完成扫描。

还想举一下上一章里面的扫描全彩点阵的程序：

```
void refresh7()
{
    static unsigned char r=0;
    static unsigned char flagrgb=0;  //当前需要点亮的颜色，0-R,1-G,2-B
    static unsigned char num=0;
    num++;
```

```
        if(num==32)
        {
            num=0;
            flagrgb++;
            if(flagrgb==3)      //说明三种颜色都扫描完了
            {
                flagrgb=0;      //从红色开始扫描
                r++;        //开始扫描下一行
                if(r==8)  //如果发现行都扫描结束则从第行开始扫描
                    r=0;
            }
        }

        if(num<light7)      //说明需要点亮
        {
            switch(flagrgb)
            {
            case 0:   //扫描红色
                DPw = ~(0x01<<r);
                DPr = ~vm7r[r];//送入R灯IO接口显示
                break;
            case 1:   //扫描绿色
                DPw = ~(0x01<<r);
                DPg = ~vm7g[r];
                break;
            case 2:   //扫描蓝色
                DPw = ~(0x01<<r);
                DPb = ~vm7b[r];
                break;
            }
        }
        else //说明不需要点亮
        {
            DPw=0xff;
            DPr=0xff;
            DPg=0xff;
            DPb=0xff;
        }
    }
```

也是通过静态变量记忆，完成某行某个颜色的亮度判断。

按键扫描

简单的一个按键扫描程序在上一章也例举过：

```
static unsigned char keylast;     //保存上次的按键值
if(key==0 && keylast==1)      //检测到一个下降沿
{
    /*do something*/
}
keylast=key;
```

下面例举一个增强型的按键扫描程序，它可以识别多个按键按下、释放：

```
void keyscan()      //5ms调用一次
{
    static unsigned int key;     //本次扫描结果
    static unsigned int keylast=0xffff;   //上次扫描结果
    unsigned char i,j;

    //开始扫描
    for(i=0;i<=3;i++)
    {
        DPkey=~pow2[i];
        for(j=4;j<=7;j++)
        {
            if(DPkey&pow2[j])  //是1
            {
                key|=pow2[4*i+7-j];
```

```
                }
                else //是0
                {
                    key&=~pow2[4*i+7-j];
                }
            }
        }
        //开始判断上升沿和下降沿
        if(key^keylast)    //说明按键状态有变化
        {
            for(i=0;i<=15;i++)
            {
                if((key&pow2[i])==0 && (keylast&pow2[i])) //下降沿，按键按下
                {
                    /****在此添加i号按键按下时要做的事****/
                }
                else if((key&pow2[i]) && (keylast&pow2[i])==0) //上升沿，按键释放
                {
                    /****在此添加i号按键释放时要做的事****/
                }
            }
        }

        keylast=key;
    }
```

在这里我想说明的是，任何复杂的功能都可以写成这种“定时器化”的形式。

如果希望更多的功能则需要添加其他结构，比如想要识别按键长按，则需要添加静态变量记忆按键状态并进行计时。

However，不管需要的功能如何，都是可以用这种结构实现的。

蜂鸣器控制

这个相比较于上面两个比较特别，因为上面两个结构比较简单，目的明确（要干什么很清楚），而且用过多次了。但这个蜂鸣器控制任务是这个工程独有的需求。

所以想以此为例，再次说明任意功能都可以写成定时器内部任务的结构。虽然没有证明它，但目前看来是的，事实应该也是。

对于这个蜂鸣器控制进程来说，需要有一下几种功能：短响 1 声、短响 2 声、短响 3 声、长响 1s。

这些功能在定时器任务里完成，通过一个变量通知这个进程执行哪种任务。

具体完成代码如下：

```
/*****************************以下是蜂鸣器任务相关**********************************
蜂鸣器的任务函数会由定时器5ms调用一次，由一个标志变量标志完成什么样的声音，使用时只要修改一下变量就行了。
总共有这么及几种声音：
    短响1声、短响2声、短响3声、长响1s
*******************************************************************************/
unsigned char music=0; /*对外接口，外部想要发声直接修改这个变量就可以了。
                         0-不响
                         1-短响声
                         2-短响声
                         3-短响声
                         4-长响1s      */
void beep()   //发声任务，由定时器每5ms调用一次
{
    static unsigned char flagDoing=0;    /*标志当前是否正在执行某个任务，0-不在，1-在
                                           当检测到music变量不为时说明有任务，此时将此变量置，标志正在执行；
                                           具体执行哪个任务根据music变量指示；
                                           当任务执行完毕时清此变量，标志没有任务，是空闲状态，清零music变量，标志没有任务将要执行。   */
```

```
if(music==0)  //没有任务
{
    buzzer=1;//关闭蜂鸣器,      buzzer是蜂鸣器控制引脚，0-响，1-不响
}
else //说明有任务
{
    if(flagDoing==0)   //如果当前没有任务正在执行，则给当前任务赋值为想要执行的任务
        flagDoing=music;

    if(flagDoing==1)   //短响声
    {
        static unsigned char count=0;    //用来计时
        buzzer=0;//打开蜂鸣器
        count++;
        if(count==20) //任务结束判断条件
        {
            count=0; //为下次任务做准备
            buzzer=1;//关闭蜂鸣器
            flagDoing=0;
            music=0;
        }
    }
    else if(flagDoing==2)  //短响声
    {
        static unsigned char count=0;
        count++;
        if(count<=20)
            buzzer=0;//打开蜂鸣器
        else if(count>20 && count<=30)
            buzzer=1;//关闭
        else if(count>30 && count<=50)
            buzzer=0;//打开
        else //执行结束
        {
            count=0;
            buzzer=1;
            flagDoing=0;
            music=0;
        }
    }
    else if(flagDoing==3)  //短响声
    {
        static unsigned char count=0;
        count++;
        if(count<=20)
            buzzer=0;//打开蜂鸣器
        else if(count>20 && count<=30)
            buzzer=1;//关闭
        else if(count>30 && count<=50)
            buzzer=0;//打开
        else if(count>50 && count<=60)
            buzzer=1;//关闭
        else if(count>60 && count<=80)
            buzzer=0;
        else //执行结束
        {
            count=0;
            buzzer=1;
            flagDoing=0;
            music=0;
        }
    }
    else if(flagDoing==4)  //长响s
    {
        static unsigned char count=0;
        buzzer=0;//打开蜂鸣器
        count++;
        if(count==140)     //结束条件
        {
```

```
                    count=0;
                    buzzer=1;
                    flagDoing=0;
                    music=0;
                }
            }
        }
    }
```

用一个变量 music 来作为对外的 API，通过它通知本进程执行哪个任务；

music 和具体任务之间有一个中间变量 flagDoing，这个变量是用来缓冲外部对此进程发送的指令的，只有当进程内部的某个任务执行结束后才会响应下一个任务请求；

在本进程的每个小任务中，都各自有自己的静态变量，用于完成各自特定的功能；

在每个小任务执行结束后，都会做一些处理，让本进程准备好接收下一个任务。

总之，不管什么复杂的控制结构，都是可以写成这种被间断调用的“定时器化”的形式。

5.4 定时器执行任务程序结构总结

1、每个任务函数不会因外部状态不同而阻塞，不管外部状态如何，这个任务的本次执行都能够顺利通畅地执行完。

2、每个任务函数执行时间都很短，有时间限制，都有自己的时间段。所以不管一个进程的状态如何，它绝对不会影响到其他进程的执行，整个系统不会因为一个进程而停下来，仍然随着定时器的节拍不断地运行。

2、在这种结构中，所有被执行的代码都是高效的，因为没有延时等函数。

3、可以用一个定时器完成多个精确的时间控制任务，事实上整个系统都在精确的时间控制下运行。

5.5 我们追求的是什么

从第 3 章开始，程序的主体逐渐从主进程转移到了定时器中断（从后台转移到前台），也对任务函数进行了一系列改造。

首先，我想强调这些都是在大量编程时自然产生的，是整个系统越来越复杂的必然结果。

另外，我们进行了这么多改变到底是在追求什么？我们渴望的系统结构是什么样的？

在此我想引用《底层工作者手册之嵌入式操作系统内核》中的一段话，也算是为操作系统做个铺垫：

“在没有操作系统的情况下，C 语言是以函数为单位实现功能的，一个函数一个函数串行地执行，一个完整的功能会由多个函数共同完成。然而当软件系统的功能变得多而庞大的时候，这种方法几乎无法使用，因为此时各个功能之间必然会有千丝万缕的联系，不可能依次串行地完成每个功能，各个功能必然需要交替执行。以函数为功能单元的程序很难在执行一个函数的时候转而去执行另外一个不相关的函数，即使是使用一些技巧实现了，也会使整个软件的结构变得混乱不堪，不利于软件的维护和扩展。函数的工作方式就决定了并不适合以它为功能单元运行复杂的程序，在这种情况下就要使用操作系统了。操作系统是对函数运行管理的系统，它可以在一个函数还没有运行完就转而去执行另一个函数，并且还可以恢复到原来的函数继续执行，这样就可以根据需要及时调整到需要运行的函数来满足各种要求。”

这段话的大概意思是：

1、传统的过程式程序是以函数为单位执行的。

2、以函数为单位的程序，在一个函数的执行过程中不能立刻跳转到另一个函数，也就是说可能会耽误另一个函数的响应。

3、就算在一个函数中嵌入了另一个函数的代码使得另一个函数也能及时得到响应，那也会是整个程序结构混乱，不利于维护和扩展。

而相比之下，操作系统具有很大的好处：

1、操作系统中的编程是以功能为单位的。在实现一个任务时根本不需要考虑其他的任务函数，更不需要在一个任务里嵌入另一个任务的代码。

2、操作系统可以在一个函数没有运行完之前直接跳转到另一个地方运行，并且以后可以恢复到原来地方继续运行。这样就可以及时对重要的函数进行响应。

再来回头看看我们做的更改，我们把占用式程序改为了非占用式程序，实际上就是细化了各个任务，让每个任务函数的单次执行时间很短，也就可以及时地响应其他任务。 对比第 1 章所说的主函数顺序调用的结构，如果把里面的函数全都换成非占用式结构，就能大幅缩短循环时间。当时有么一句话：“在最坏情况下的任务级响应时间取决于整个循环的执行时间”，为什么这里把这句话搬过来，你懂的。

另外一点，在改为定时器执行任务后，保证了各个任务都有自己固定的时间段执行，每个任务都绝对不会阻塞进程、绝对不会影响到其他任务的运行。 这一点和操作系统相比甚至更有优势，因为操作系统的高优先级任务是可以阻塞低优先级任务的，而定时器执行任务结构中的所有任务都一定会及时得到执行。只要能够把这个任务以“定时器化”的形式写入到定时器中断处理程序里，它就绝对不会因为意外而被阻塞（这也是对任务函数的要求之一）。

我们的追求和操作系统是一样的，我们希望各个任务功能独立实现，不要相互影响。同时各个任务都能够及时得到响应。操作系统采用将寄存器入栈的方式保存状态信息，而在定时器执行任务的结构中是用静态变量的方式保存任务信息。

在此把这个定时器执行任务的结构称为“准操作系统”，因为它的很多特征已经非常接近操作系统了。

下一篇将会开始介绍操作系统的原理，不过在操作系统开始前还要介绍另外一个非常特别的编程结构，这就是本篇最后一章的内容。

第 6 章 面向对象思想+事件驱动结构

先来看一下这个东西吧：http://v.youku.com/v_show/id_XNTk2NzExMjg4.html（或者从 http://pan.baidu.com/s/1mZsas 下载）。

看完之后应该会觉得这个东西的结构非常复杂，这是笔者做过的最复杂的项目之一。由于是给公司开发的，所以和上一章一样不能公开源代码，但是会举一些简单的例子说明。

刚接到这个项目了解了大体功能后，第一反应是用“界面函数”的结构。确实，这个东西是非常适合用界面函数完成的，但是由于当时笔者正在自学 C++，于是用 C++的思路分析了一下这个项目：

1、总共有 4 个界面，而且有两个和 VB 的列表框好像啊。

2、几乎所有的动作都是由旋转编码器触发的。

经过一番思考之后，笔者决定做一次尝试，用面向对象的思想加上事件驱动的机制完成它。当时对面向对象和事件驱动什么的概念都还没完全搞清楚，不过完成此项目之后理解加深了一点。

下面就介绍一下这种结构，也希望能借此说明“对象”和“事件”的概念。

6.1 对象和事件

基本上每个面向对象语言的书都会把对象的概念说一下。在这个项目里有 4 个界面，把这四个界面看作四个对象，这四个对象的所有动作都由事件驱动。

什么又是“事件”呢？简单来说“一个对象发生了某个事情”就是这个对象的某种事件。事件一定是基于某个特定的对象而言的，不能简单地说“发生了某个事情”，应该说“某个对象发生了某个事情”。而我们要做的就是确定“每个对象有哪些事情会发生”，并完成“某个对象在发生某个事情时要做的事”。

所以对于一个对象而言，它应该有：

1、与它对应的事件函数，用于执行“某个事件发生时要做的事”，注意：一个对象所拥有的这些函数的个数和它可能发生的事件数是相等的。

2、完成上述函数所需要的辅助函数。

这些函数有些可能是公共的，是由系统提供的 API 或者其他工具函数；

也可能是这个对象特有的，是对这个对象做的某种更改。

3、每个对象都是实体的存在，它们都有自己的属性，这些属性在程序中的体现就是变量。

这里面，1 和 2 的函数本质上都是一样的，都是这个对象所包含的函数。不同的是：

与事件相对应的函数称它为这个对象的“事件”，这些函数数量与这个对象的事件数是相等的；

完成某种特定操作的函数称它为这个对象的“方法”，也就是“这个对象可以做的事”。并且这些方法中，有些是允许被外界调用的，有些只允许在本对象内使用，所以又分为“公共方法”和“私有方法”。

下面就以一张表来说明一个对象内部的组成，以及各个成员的含义：

函数	事件		完成当这个对象发生某个事情后要做的事
	方法	公共方法	这个对象能做的事，并且外部也可以命令这个对象做这个事
		私有方法	这个对象能做的事，但是只允许本对象内部其他方法和事件调用
变量	公共变量		标志着这个对象的属性
	私有变量		这个对象的一些不想被外部知道和没必要被外部知道的属性，或者就是一些辅助变量

有了这些了解之后我们就可以开始构建整个系统了。

6.2 C 语言对一个对象的封装

在 C++ 中有专门的对象结构，它可以把对象里的函数和变量分为公共、私有等类型。

而在 C 语言中没有这样的结构，我们通过使用上的约定也可以达到同样的效果。比如一个内部的函数，我们约定有些函数外部可以调用，它就是公共函数；约定有些函数外部不要调用，它就是私有函数。

下面是对象的各个部分在 C 语言中的形式及意义：

名称	形式	意义
公共变量	全局变量	这些变量可以看作对象的属性。一般只允许外部读，不允许外部直接修改，想要修改只有通过本对象提供的公共方法。
私有变量	全局变量	用于辅助对象完成某些功能，但是对外部又没有什么特殊意义，不足以成为属性，或者就是临时变量。总之无需对外公开。
私有方法	内部函数或外部函数	不对外公开，用于辅助公共方法的函数。约定在本对象的事件中也不能调用。
公共方法	外部函数	本对象对外（本对象的事件也看作外部）提供的接口，封装好之后的这些函数是不允许修改的，它属于本对象固有的内容。允许本对象的事件和其他对象的事件使用。
事件	外部函数	发生在该对象上的事件。 事件对所有对象的操作（包括与该事件对应的本对象），原则上只能通过该对象提供的公共方法进行。

公共方法和事件本质上都是对外公开的函数。公共方法完成对本对象的某一主动的操作，而事件是完成对外部被动的响应。

甚至可以把事件从对象的组成中取出来，把事件归到系统范畴。

下面就以一个简单的列表框的例子说明如何封装一个对象，假设这个列表框（对象）的名字叫做 List1，一共只有 3 个列表项，每个列表项都有一个自己的名称和内容（0~255），有一个光标指示当前选中列表项，分配的事件有“列表项+1”“列表项-1”“列表项内容+1”“列表项内容-1”，并且这些改变是可以循环的（255+1=0，0-1=255）。

假设屏幕显示的网格是 4 行 16 列，外部提供的绘图函数有：

```
Disp(r,c,unsigned char *)   //往 r 行 c 列写入一个字符串
```

Clear()　//清屏

有了这些条件后，对这个对象的封装如下：

List1.c :

```
/* List1 */
void Clear(); //清屏函数
void Disp(unsigned char r,unsigned char c,unsigned char *p); //在r行c列显示字符串
/**************公共变量*****************/
#define List1_ListCount 3   //列表长度为固定
unsigned char List1_ListIndex=0; //当前选中的列表项，从开始数
#define List1_StartIndex 0  /*当前屏幕显示的第一个列表项序号，从开始数，
                                    由于屏幕能一次性把个列表项都显示出来，所以这里是固
定。*/
unsigned char List1_ListData[3]; //三个列表项的数据
/**************私有变量***************/
unsigned char code List1_Name0[]="Power"; //第号列表项的名称
unsigned char code List1_Name1[]="Mode "; //第号列表项的名称
unsigned char code List1_Name2[]="K    "; //第号列表项的名称
/**************私有方法***************/
void List1_DispName()   //在固定位置显示个列表项的名称
{
    Disp(0,1,ListName0);    //显示号列表项名称
    Disp(1,1,ListName1);    //显示号列表项名称
    Disp(2,1,ListName2);    //显示号列表项名称
}
void List1_DispCursor()//在当前选中列表项前显示“>”，没选中的显示空格
{
    unsigned char i;
    for(i=0;i<List1_ListCount;i++)
    {
        if(i==List1_ListIndex)
            Disp(i,0,">");
        else
            Disp(i,0," ");
    }
}
void List1_DispData(unsigned char n) //显示n号列表项的数据
{
    unsigned char vm[4];    //现存，以十进制显示，总共三位数
    //先计算现存
    vm[0]=List1_ListData[n]/100+0x30;     //计算百位的现存
    vm[1]=(List1_ListData[n]%100)/10+0x30;     //计算十位的现存
    vm[2]=List1_ListData[n]%10+0x30; //计算个位的现存
    vm[3]='\0';   //字符串结尾
    //下面开始显示
    Disp(n,10,vm);      //从n行列开始写入数据
}
/***************公共方法************************/
void List1_Show()
{
    //先清屏
    Clear();
    //再显示所有列表项的名称
    List1_DispName();
    //再显示所有列表项的数据
    {
        unsigned char i;
        for(i=0;i<List1_ListCount;i++)
            List1_DispData(i);
    }
    //再显示光标
    List1_DiapCursor();
}
void List1_Hide()
{
    Clear();
}
void List1_SelectedListP1() //当前选中项内容+1
{
```

```
    List1_ListData[List1_ListIndex]++;
    List1_DispData(List1_ListIndex);
}
void List1_SelectedListM1() //当前选中项内容-1
{
    List1_ListData[List1_ListIndex]--;
    List1_DispData(List1_ListIndex);
}
void List1_ListP1()     //选中项序号+1
{
    List1_ListIndex++;
    if(List1_ListIndex==3) //实现循环
        List1_ListIndex=0;
    List1_DispCursor();     //刷新光标
}
void List1_ListM1()     //选中项序号-1
{
    List_ListIndex--;
    if(List_ListIndex==0xff)     //实现循环
        List1_ListIndex=2;
    List1_DispCursor();     //刷新光标
}
/***************事件******************/
void List1_Key0Down()  //0号键按下，让选中项序号-1
{
    List1_ListM1();
}
void List1_Key1Down()  //1号键按下，让选中项序号+1
{
    List1_ListP1();
}
void List1_Key2Down()  //2号键按下，让选中项内容-1
{
    List1_SelectedListM1();
}
void List1_Key3Down()  //3号键按下，让选中项内容+1
{
    List1_SelectedListP1();
}
```

在其他地方只要包含这个 List1.c 文件就可以调用该对象里的各种方法和事件函数了，里面的私有方法和私有变量虽然约定外部是不要使用的，但是外部确实是有能力使用的。所以也可以为这个对象写一个 List1.h 文件，只把公开的部分进行声明：

List1.h:

```
/* List1 */
#ifndef LIST1_H
#define LIST1_H

#define List1_ListCount 3   //列表长度为固定
unsigned char List1_ListIndex=0; //当前选中的列表项，从开始数
#define List1_StartIndex 0  /*当前屏幕显示的第一个列表项序号，从开始数，
                        由于屏幕能一次性把个列表项都显示出来，所以这里是固定。*/
unsigned char List1_ListData[3]; //三个列表项的数据
/***************公共方法************************/
void List1_Show();
void List1_Hide();
void List1_SelectedListP1();     //当前选中项内容+1
void List1_SelectedListM1();     //当前选中项内容-1
void List1_ListP1();//选中项序号+1
void List1_ListM1();//选中项序号-1
/***************事件******************/
void List1_Key0Down(); //0号键按下，让选中项序号-1
void List1_Key1Down(); //1号键按下，让选中项序号+1
void List1_Key2Down(); //2号键按下，让选中项内容-1
void List1_Key3Down(); //3号键按下，让选中项内容+1

#endif
```

如果采用了 List1.h 的话，要把 List1.c 文件里重复定义的部分给去掉。

每个对象有每个对象的特点，它们差别很大，构建方法也是大不相同的。但是每个对象构建好之后就是一个模板，它是非常独立的，在其他地方只要把代码直接复制过去做少量更改就可以使用了。

6.3 事件分配机制

一个工程中会有多个对象，每个对象都有一些可能会发生的事件，这些事件函数是由系统调用的，由系统来判断什么对象发生了什么事件。

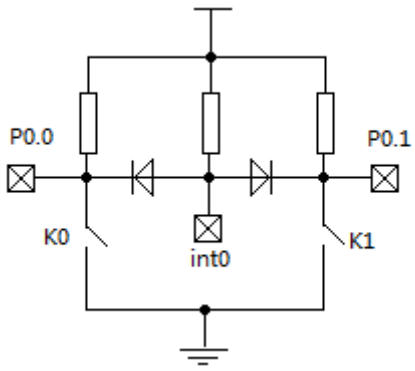
比如在这个项目中，基本的事件有：左编码器按下、右编码器按下、左编码器左旋、左编码器右旋、右编码器左旋、右编码器右旋、左编码器按下左编码器左旋、左编码器按下左编码器右旋、右编码器按下右编码器左旋、右编码器按下右编码器右旋、串口接收数据事件、IIC 接收数据事件。

由于旋转编码器的驱动本身用的就是外部中断，外部中断对单片机来说就是一种意外事件。所以只要在中断里判断当前进行的是什么操作，并记录当前哪个界面正在被使用，就可以调用相应对象的相应事件了。

然而，并不是所有的对象都一定具有所有的这些事件，根据该对象功能的需要，选用一部分有用的事件进行响应。比如在 IIC 界面里不考虑串口，这个界面就只响应 IIC 接收数据事件，而忽略串口接收数据事件。

事件并不一定非要由外部中断产生，也可能是系统虚拟的。总之事件分配有系统完成，系统利用各个资源抽象出事件概念，然后分配到相应对象上。

下面是一个完成事件分配的例子：



有两个按键，任意一个按键按下都会触发 int0 中断。假设共有 2 个对象，下面是事件分配的代码：

```
unsigned char FocusNum=0;   //标志当前获焦的对象序号，在此只有两个对象，范围是0~1
sbit kint0=P3^2;
sbit k0=P0^0;
sbit k1=P0^1;

……

void int0() interrupt 0
{
    EA=0;     //关中断

    if(k0==0)//说明是k0按下
    {
        switch(FocusNum)
        {
        case 0:   //对象0
            Form0_k0Down();
```

```
                break;
            case 1:  //对象1
                Form1_k0Down();
                break;
            }
        }
        else if(k1==0)     //说明是k1按下
        {
            switch(FocusNum)
            {
            case 0:  //对象0
                Form0_k1Down();
                break;
            case 1:  //对象1
                Form1_k1Down();
                break;
            }
        }

        while(kint0==0);   //等待两个按键都释放
        IE0=0;    //清除中断标志，防止在中断处理程序执行过程中再次触发了中断
        EA=1;     //开中断
    }
```

采用 FocusNum 来记录当前激活的对象。

在此想说明的是，k0 和 k1 两个操作都是放在一个中断里的，具体产生什么事件则是再次通过代码判断的，这些代码属于系统层，是根据实际需要对事件的抽象。

比如如果需要产生“两个按键同时按下”事件，则可按如下方式分配：

```
    void int0() interrupt 0
    {
        unsigned char
        EA=0;     //关中断

        if(k0==0 && k1!=0) //说明是k0按下
        {
            switch(FocusNum)
            {
            case 0:  //对象0
                Form0_k0Down();
                break;
            case 1:  //对象1
                Form1_k0Down();
                break;
            }
        }
        else if(k1==0 && k0!=0)//说明是k1按下
        {
            switch(FocusNum)
            {
            case 0:  //对象0
                Form0_k1Down();
                break;
            case 1:  //对象1
                Form1_k1Down();
                break;
            }
        }
        else if(k0==0 && k0==0)//两个按键都按下
        {
            switch(FocusNum)
            {
            case 0:  //对象0
                Form0_BothDown();
                break;
            case 1:  //对象1
                Form1_BothDown();
                break;
            }
```

```
        }

        while(kint0==0);   //等待两个按键都释放
        IE0=0;    //清除中断标志，防止在中断处理程序执行过程中再次触发了中断
        EA=1;     //开中断
    }
```

甚至可以继续改，实现“k0 按下时 k1 按下”“k1 按下时 k0 按下”“k0 长按”“k0 连按”等事件，这里就不一个个实现了。

总之只要系统层能够识别的动作都可以抽象成事件。

6.4 系统层构建

在以前的编程中从来没把程序这么明确地分层，可能是因为这次项目比较复杂，并且用来面向对象的方法，所以这种结构自然就产生了。在各个对象中有一些公共方法，这些函数完成特定的功能，而他们都依赖于底层的支持。

当然可以让对象直接操作驱动函数，从最底层开始。但是这样的话，一方面用起来会很麻烦，另一方面可能不是所有的底层功能都会用到。

所以让系统层根据上层对象的需要把这些功能封装，并想上层提供使用接口。这是系统层做的事情之一。

另外系统层也会构建一些控制逻辑，这些功能并不在底层有实体的驱动函数，它是系统在软件层面抽象出来的。比如当前系统的数据显示进制、背光灯时间等。对于上层的对象来说，根本不需要管这些，只要调用系统层提供的 API 就行了。

此外，系统层要做的当然还有事件分配，上一节介绍的事件分配系统是系统层核心的一部分，而系统层还有其他很多功能。事件分配系统是利用各个资源抽象出事件概念，并分配给各个对象。而本节讨论的系统层是利用各个资源，在底层驱动的支持下，根据需要构建出一些控制逻辑，并封装成系统 API，供上层软件使用。

综上，系统层做的事情有：

1、构建事件分配系统。

2、对底层驱动封装，并向上层提供操作接口。

3、根据需要再构建一些其他控制结构，并向上提供接口。

6.5 库函数

库函数也属于对系统对底层的封装。某些功能可能比较复杂，可以构成一整套体系，那么就可以把这些函数归位一类，作为完成某个功能的库函数。

这点和类似于操作系统的 GTK 库或者 QT 库，它们提供了大量的绘图函数。

在本项目中，使用了绘图库，它就是在底层对液晶屏操作的基础上建立的文字和图片显示函数，提供了在指定位置写入字符、汉字，以及反色写入、垂直镜像写入等功能。上层的对象调用这些函数会非常方便。

总结一下库的作用：利用系统层的 API，或者跳过系统层直接调用驱动函数，构建出自己的一套控制逻辑，并对外提供基于这个控制逻辑的函数库。

第二篇

操作系统

在经历了第一篇的各种编程结构之后，笔者忍不住开始操作系统的学习了。为了了解操作系统的原理，笔者选择先从 uCOS 这个简单的操作系统开始。

然而在看完 uCOS 之后，我发现还是没找到真正想要的东西。虽然弄清楚了操作系统的大体工作原理，了解了 uCOS 是如何实现优先级并控制调度器调度的，但是最根本的任务切换功能却不知是怎么实现的。

很多介绍操作系统的书都是介绍操作系统内核结构的，介绍的是操作系统原理和各个功能的实现，但是却很少介绍更底层的任务切换原理。这就使得我很难接受操作系统，因为我不知道操作系统到底是怎么产生的。

这时候有幸在网上看到了一篇文章《底层工作者手册之嵌入式操作系统内核》，这是一份在网上公开的长文。

下载：http://bbs.csdn.net/topics/390598440

作者：Ifreecoding

新浪博客：http://blog.sina.com.cn/u/2425202502

CSDN：http://my.csdn.net/ifreecoding

这本书简单介绍了操作系统的概念，然后在 arm7 平台上一步步实现操作系统，同时也公开了源代码。在此我向大家强烈推荐此手册，对了解操作系统原理有很大帮助。

本篇将完成从单片机编程到操作系统的过渡。介绍原来单片机编程方式的缺点，并对比和分析使用操作系统的好处。然后介绍操作系统的原理，明确操作系统要做哪些事情。最后将在 s3c6410 上实现简单的操作系统。

关于操作系统更复杂的功能构建方面本篇不作介绍，这些内容在《底层工作者手册之嵌入式操作系统内核》中有更为详细的描述，本篇的重点在于。

第 7 章 为什么要有操作系统

传统单片机编程的方式也叫“前后台系统”，把主进程称为后台，中断叫做前台。“后台”也称作“任务级”，“前台”也称作“中断级”。

关于这种结构的缺点在第 1 章和第 5 章已经做过分析，主要有以下几点：

1、任务执行顺序固定，必须等一个任务执行完毕才能执行下一个任务。

2、任务执行时间长，在处理一个任务的过程中不能响应其他任务。在最坏的情况下，任务的响应时间是所有任务循环一次的时间。

3、这种结构以函数为单位，构建复杂系统时结构混乱，不易编写程序和维护。

面对这些问题，在第一篇我们对编程结构进行了各种改造。把占用式程序改为了非占用式程序，提高任务单次执行速度，同时也就提高了每个任务的响应速度；把任务放进定时器里运行，并且保证每个任务都不会被卡住，保证每个任务都能及时得到运行。

而操作系统有一个特殊功能，是以前的编程方式所不具备的，同时也是操作系统的核心功能，那就是可以在一个函数没有执行完时就跳转到另一个函数。

在第一篇中，我们认为 main 函数及其调用的所有子函数（以及子函数再次调用的函数……）都在一个“主进程”里。它们都属于一个进程，因为它们都在一个函数调用链里面。在以前的单进程结构下，如果定义了一个函数没有被这个函数调用链的任何函数调用，那么它将永远无法得到执行。也正因为这样，我们才要把任务写成一个个函数链接成一个进程依次执行。

但是有了操作系统之后，我们可以把任务按功能写成多个进程，每一个功能用一条函数调用链实现。然后操作系统要做的就是决定什么时候跳转到哪一条链里运行，这个决定要考虑的因素也是很多的，并且还要保证来跳转时每个进程运行结果的正确性。任务切换功能是操作系统的最基本功能，也可以认为是操作系统管理任务的最基本工具。而操作系统内核要做的就是利用好这个工具，在合适的时间切换到合适的任务，这些是内核所要构建的更庞大的系统。

明确一下，操作系统要做的主要事情有 2 件：

1、“任务切换”功能，完成在各个进程间的跳转。这是操作系统最根本的功能。

2、管理任务进行任务切换，这是在 1 的基础上，利用 1 提供的任务切换功能建立的更上一层的控制逻辑。这也就是俗称的操作系统内核。

除了这两点之外，操作系统还可能会提供文件系统、网络管理等功能，但这些都不是必须的。

对比看一下以前遇到的问题。采用操作系统之后，任务之间没有执行顺序一说，只有优先级区别，对于抢占式内核而言，任何时刻都能保证需要运行的最高优先级任务得到运行。编程时当然也不需要考虑其他任务，每个进程只要专心完成自己的任务就行了。

本篇将主要介绍任务切换功能。对于更复杂的内核构建，不同的操作系统有着不同的管理方式，并且这些内容也是很多的。如果想要学习如何构建操作系统内核向大家推荐《底层工作者手册之嵌入式操作系统内核》，它介绍了两个操作系统的构建，从简单的任务间切换到信号量的实现都有详细说明。当然也可以看一些著名的操作系统原理书，或者对着现有的优秀操作系统学习，如 uCOS、Linux 等。

第 8 章 任务切换的具体工作

上一章已经明确任务切换是操作系统最基本的功能，本章将介绍任务切换具体做些什么工作，同时明确一个任务会占用哪些资源。

8.1 CPU 工作原理

本节讨论 CPU 执行指令的过程，通过这个过程来分析一个任务有哪些东西是它特有的，即在任务切换时需要保存和恢复的内容有哪些。

【CPU 对指令的执行】

C 编译器把 C 语言源代码编译为一条条指令，这些指令也是以一个个数字代码的形式保存的。不管它保存在什么地方（ROM 也好，RAM 也好），CPU 中总有一个寄存器 PC 指向下一条要运行的指令。CPU 在每执行一条新指令时，会从 PC 指针所指的地方取出指令，然后 PC 指针会自动增加，指向下一条指令。

所以修改 PC 指针也就能够控制 CPU 执行代码的位置。C 编译器中的分支结构、循环结构还有函数的调用，这些过程中的一个很重要的一点就是对 PC 指针的修改。比如调用一个子函数实际上就是修改 PC 指针，使其指向子函数所在的代码空间，然后就会直接从子函数处取指令执行；子函数返回时也是修改 PC 指针，使其指向父函数在调用子函数处的下一条指令，使父函数能够接着运行。（注意在函数调用时除了修改 PC 指针还有其他事情要做，这里暂不考虑）

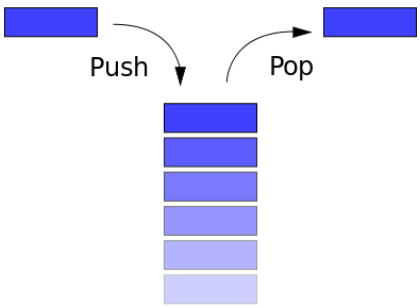
C 编译器会安排好 PC 指针的修改，它会根据具体函数调用的情况让程序在各个代码空间之间跳转，也就是在各个函数之间跳转。在同一个函数调用链里的函数都有可能得到执行，但是函数调用链之外的函数却得不到执行。此时就要手动修改 PC 指针，让 CPU 从一个函数调用链跳转到另一个函数调用链，也就是从一个进程跳转到另一个进程执行。

【全局变量】

在 C 中申请的全局变量将会始终占用一段内存，它们的地址在程序执行的过程中固定不变，并且 C 编译器会保证这段地址不会再次分配给其他地方。

【堆栈和 SP 指针】

在一条函数调用链（一个进程）中还有一个重要概念——堆栈。堆栈的数据结构简单表示如下（取自维基百科）：



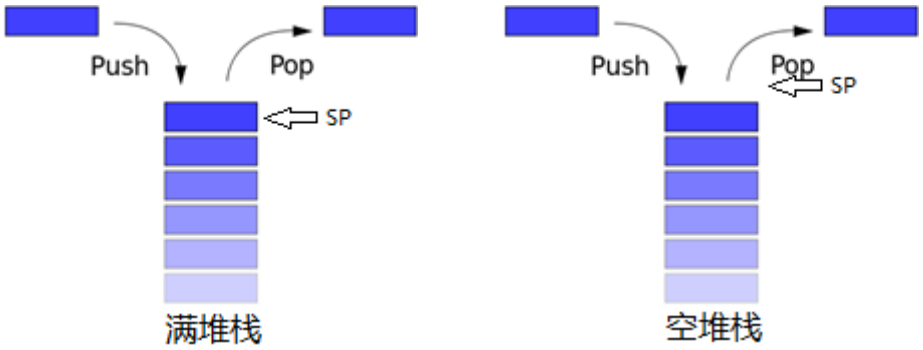
堆栈在逻辑上有栈顶和栈底之分，栈底保存的是先入栈的元素，栈顶保存的是后入栈的元素。有新数据存入时，会把数据保存在栈顶，要取出数据时也会从栈顶取出。所以它是一种“先入后出”的方式。

此外，堆栈的使用需要一个栈顶指针（SP），它标识着栈顶的位置（但是不一定直接指向栈顶的元素，参看后面堆栈的分类）。CPU 寄存器中有一个专门的 SP 指针。

堆栈的具体作用后面分析，先来看一下堆栈的分类。

堆栈根据在内存中的存储方向不同可分为递增栈和递减栈。递增栈的栈顶往内存高地址方向延伸，递减栈的栈顶往内存低地址方向延伸。

根据栈顶指针标识栈顶方式的不同可分为满堆栈和空堆栈。满堆栈的 SP 等于最后一个入栈的元素的地址，空堆栈的 SP 等于下一个入栈元素将要保存的地址。



所以堆栈共可分为空增栈、满增栈、空减栈和满减栈。需要强调，以何种方式使用堆栈与 CPU 无关，这是可以按自己意愿决定的，但是在某个平台下的 C 编译器使用哪种方式是约定好的，我们在使用堆栈时最好遵循这种约定，以便和该平台下的 C 代码兼容。

【函数调用过程】

堆栈对于函数的调用是必不可少的，具体过程在此不做介绍，事实上是我有些地方不能确定，说出来可能是错的，可以参看《底层工作者手册之嵌入式操作系统内核》等资料。不过在函数调用过程中有几点是可以确定的：

1、在发生函数调用前，需要把一部分寄存器保存到堆栈中，以便函数返回时恢复。

2、如果 CPU 寄存器不足以传递所有参数，肯定要用到堆栈在父函数和子函数之间传递参数。

3、子函数的所有动态局部变量开销都从堆栈中获得。

4、子函数如果再调用子函数，则其所有的内存开销继续从堆栈中获取。

另外，在函数调用过程中需要强调一点的是，CPU 中有一个 LR 寄存器，在函数调用前 C 编译器会把父函数下一条指令的地址存入 LR 寄存器中，子函数在返回时实际上就是跳转到 LR 寄存器指向的位置运行。

所以，关于堆栈有一下几点可以明确：

1、程序正常运行肯定离不开堆栈，CPU 寄存器中 SP 寄存器和 PC 寄存器都是需要保存的。

2、一个函数调用链需要一个堆栈，如果开辟了多个进程，则每个进程都要有自己的堆栈。

3、程序使用堆栈的位置是由 SP 决定的。主进程的堆栈位置是由 C 编译器确定，也就是由编译器确定 SP 的初试值。而其他进程的 SP 是后期决定的，为了保证每个进程都有自己的堆栈空间、确保堆栈内容不被破坏，需要在 C 中事先申请一段内存空间，之后在该任务第一次运行时让 SP 指向该段内存，这样就能保证该内存空间不被其他变量占用。

【CPU 中的其他寄存器】

在程序运行过程中，CPU 中还有一些通用寄存器和状态寄存器，这里不做介绍，可以查阅其他资料。

总之这些寄存器和程序运行密切相关，在任务切换时是需要保存和恢复的。不同 CPU 有哪些寄存器也是不同的，后面在 s3c6410 上实现任务切换时会介绍 6410 上的各个寄存器。

8.2 任务切换做的事

任务切换简单地说就是保存当前进程状态、恢复另一个进程状态。那么一个进程有哪些东西是需要保存和恢复的呢？

PC 是指向程序运行的地方，每个进程在切换前都要记住本进程当前运行的位置，下次切换到本进程时能够继续当前位置运行；

任务的运行离不开堆栈，所以 SP 指针也要保存；

CPU 中的其他寄存器也与程序运行有关，也需要保存。

简单地说，CPU 中所有寄存器都要保存，保存了这些寄存器也就保存了当前的状态，恢复这些寄存器也就恢复到了另一个状态。

然而并不是所有的寄存器都要保存的，因为编译器在使用寄存器时有些规定，这些规定决定了有些寄存器可以不保存。我想说的是 PC 寄存器，如果我们把任务切换写成了一个函数形式，那么在调用这个任务切换函数前就会把返回地址保存到 LR 寄存器中，所以我们没必要保存 PC，只要在恢复任务时把 PC 恢复为 LR 的值就可以切换到原来的地方运行了。除 PC 寄存器之外，其他寄存器也可能不需要保存，这要根据具体编译器规则了，但是全部保存总是没错的。

至此还有一个问题：这些 CPU 寄存器保存在什么地方？首先可以明确，每个进程都需要自己的一段内存空间来保存该进程的各个 CPU 寄存器结果，只要能够保证保存和恢复的正确性，放在什么地方都是可以的。不过为了方便，通常把它们就保存在该进程运行时的堆栈里。

以上就是对任务切换的大体介绍，接下来将在一个具体的硬件平台上实现它。

第 9 章 在 s3c6410 上实现任务切换

9.1 了解 s3c6410 的寄存器

任务切换要做的就是 CPU 寄存器的保存和恢复，下面来了解一下 6410 的各个寄存器以及它们的作用：

ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined	Secure monitor
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und	R13_mon
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und	R14_mon
R15	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und	SPSR_mon

= banked register

这是数据手册中的寄存器表。

ARM11 处理器共有 7 种工作模式，每种工作模式下有不同的寄存器。在此我们只关心 User（用户）模式下的寄存器，如果想深入了解可以查阅 ARM11 核的数据手册和其他相关资料。

在这个模式下，有 R0~R15 共 16 个通用寄存器和 1 个 CPSR 状态寄存器，每个寄存器都是 32 位的。

通用寄存器 R0~R15 中，有些寄存器有特殊的用途，每个寄存器的用途如下图：

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

我们关心的有下面几个寄存器：

R15 作为 PC 指针，指向内存中下一条指令的地址。ARM11 的工作状态有 ARM 状态和 Thumb 状态，ARM 状态下认为所有指令都是 32 位的，Thumb 状态认为所有指令都是 16 位的。在此我们都使用 32 位的指令，一条指令在内存中占 4 个字节，也就是说没执行完一条指令 PC 指针就会加 4（没有跳转的情况下）。

R14 作为 LR 寄存器，关于 LR 寄存器的功能上一章也已经提到，会在发生函数调用前保存父函数下一条指令的地址，子函数返回时直接跳转到 LR 指向的内存处运行。

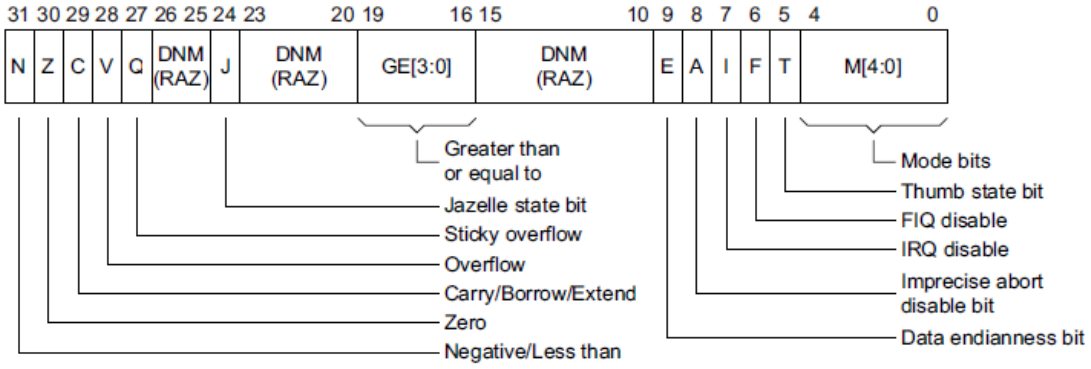
R13 作为堆栈指针，上一章介绍过程序的运行离不开堆栈，这个 R13 就充当栈顶指针的功能。

其他寄存器的功能就不作介绍了，因为在本文实现任务切换时用不到。但也有比较重要的，比如 R0~R3 在发生函数调用时有传递参数的功能，这在 C 语言和汇编混合编程时很重要。由于接下来我们写的任务切换函数不需要传递参数，在此也就不介绍了。如果你想详细了解可以查阅其他资料。

是什么规定了这些寄存器的功能的呢？这就是 AAPCS（Procedure Call Standard for the ARM Architecture），它规定了编译器应该如何使用这些寄存器，全球同志都遵守这个协议，我们在进行混合编程时也应该遵守它。

总之，不管各个寄存器功能是什么，它们和程序运行是密切相关的，在任务切换时要将它们保存和恢复。

除了保存这些通用寄存器，也不要忘了保存状态寄存器 CPSR。而关于 CPSR 中各个位的作用要特别说一下，因为在操作系统的任务初始化时要给予一个明确的 CPSR 值。而前提当然是我们知道该赋什么值。



N、Z、C、V、Q 是 5 个重要的标志位，在初始化时不需要用到，我们现在关心的是 D5~D0 位，如果想要详细了解可查阅数据手册。

D5 位：1-Thumb 模式，0-ARM 模式。

D4~D0：

M[4:0]	Mode
b10000	User
b10001	FIQ
b10010	IRQ
b10011	Supervisor
b10111	Abort
b11011	Undefined
b11111	System

在任务第一次执行前，要给任务一个初始状态，也就是初始化这个任务的堆栈。其中一步就是要给这个 CPSR 寄存器赋初值。

在后面的例子中，我们选择的是 ARM 工作状态，工作模式选择的是 Supervisor（管理）模式。一般使用时程序工作在 User 模式居多，但是由于后面是用 J-Link 连接运行的，所以要工作在 Supervisor 模式。

9.2 要用到的几条汇编指令

任务切换直接对 CPU 寄存器进行操作，这些是 C 语言办不到的，下面就介绍一下后面要用到的几条汇编指令。

MOV 目的寄存器 ，源操作数

【功能】把源操作数的内容赋给目的寄存器。

【例】

```
MOV R0,R1          ;把 R1 的内容赋给 R0
MOV R0,#0x12345678  ;把 0x12345678 赋给 R0
```

注 1：“#”开头表示立即数寻址，意思就是直接传入立即数。后面还会遇到其他寻址方式，遇到时会说明。

注 2：MOV 指令还可以加一些其他条件完成不同的功能，比如 MOVEQ、MOVNE 等，后面的参数也有其他形式。在此只介绍在后面完成任务切换时用到的方法，更多用法请参阅其他资料，有兴趣可以系统了解一下 6410 上的汇编指令。

SUB 目的寄存器 ，被减数,减数

【功能】目的寄存器 = 被减数 - 减数

【例】

```
SUB R13,R13,#64          ;R13 = R13 – 64
SUB R13,R0,R1            ;R13 = R0 – R1
```

MRS 通用寄存器 ，状态寄存器

【功能】读取状态寄存器的值，保存到通用寄存器。注：MOV 指令无法访问状态寄存器。

【例】

```
MRS R0,CPSR          ;读取 CPSR 的值放到 R0
```

MRS 状态寄存器_域 ，操作数

【功能】把操作数的指定位赋给状态寄存器的对应位，传递哪些位由域确定：

域	设置的位
f	[31:24]
s	[23:16]
x	[15:8]
c	[7:0]

这些域可以一次使用多个。

【例】

```
MSR CPSR_fsxc,R0         ;把 R0 的所有位传递到 CPSR 中
MSR CPSR_f,R0            ;把 R0 的[31:24]位传递到 CPSR 的[31:24]位中
```

BX 寄存器

【功能】跳转到寄存器指向的内存单元。BX 指令可以跳转并切换工作状态（ARM 状态或 Thumb 状态），但是这里只用到了跳转功能。

【例】

```
BX R14    ;跳转到 R14 指向的内存单元运行
```

LDR 目的寄存器 ,[存储器地址]

【功能】从内存的指定位置读取 32 位数据到寄存器。

【例】

```
LDR R1,[R0]    ;读取 R0 指向的内存单元到 R1，相当于 C 语言中的 R1=*R0，把 R0
```

的值看作地址，从内存中寻址取出数据。

注：一个 CPU 的寄存器是 32 位的，而内存是以字节为单位，所以一个寄存器在内存中的保存肯定占用 4 个字节。但是这 4 个字节又是如何存放的呢？这里要介绍“大端存储”和“小端存储”的概念。

可以认为小端存储就是寄存器的高字节存放在内存的高地址中，寄存器低字节存放在内存低地址中；

大端存储反过来，是寄存器的高字节存放在内存的低地址中，寄存器低字节存放在内存的高地址中。

在 6410 的汇编中使用的是小端存储。

LDR 目的寄存器 ,=C 中定义的变量

【功能】获取在 C 中定义的变量的地址，保存到目的寄存器中。

【例】

假设在 C 代码中有定义：

```
unsigned long a;
unsigned long *p=&a;    //p 指向 a
```

则在汇编代码中：

```
LDR R0,=a       ;获取 a 的地址，类似于 R0=&a，此时 R0 的值和 p 的值相等
LDR R1,=p       ;获取 p 的地址，类似于 R1=&p，此时 R1 指向 p，p 指向 a，R0 也指向 a，R0 的值和 a 的值相等
```

STR 源寄存器 ,[存储器地址]

【功能】把源寄存器的内容写入到内存的指定位置。

【例】

```
STR R1,[R0]    ;把 R1 的值写入到 R0 指向的位置
```

LDM 基址寄存器(!),{寄存器列表}

【功能】一次性从内存中读取多个数据保存到寄存器列表中。

基址寄存器后面如果加了“!”表示操作完之后更新基址寄存器的值。

【例】

```
LDMIA R13,{R0-R3}       ;从 R13 指向的内存单元读取数据，保存到 R0~R3 寄存器中，并且执行完之后 R13 的值不变
LDMIA R13!,{R0-R3}      ;从 R13 指向的内存单元读取数据，保存到 R0~R3 寄存器中，并且执行完之后 R13 的值增加 4*4=16
```

注：LDM 后面的 IA 暂且不管，等介绍完下一个指令后一起说明。

STM 基址寄存器(!),{寄存器列表}

【功能】一次性把寄存器列表中的寄存器写入到内存的指定位置。

基址寄存器后面如果加了“!”表示操作完之后更新基址寄存器的值。

【例】

```
STMDB R13,{R0-R3}       ;把 R0~R3 寄存器写入到 R13 指向的位置，并且执行完之后 R13 的值不变
STMDB R13!,{R0-R3}      ;把 R0~R3 寄存器写入到 R13 指向的位置，并且执行完之后 R13 的值减小 4*4=16
```

注 1： LDM 和 STM 指令后面都要加后缀，后缀可以是 IA、IB、DA 和 DB。还记得第 8 章说过堆栈的分类，可以分为空增栈、满增栈、空减栈和满减栈，当时说堆栈的种类是可以凭自己意愿选择的，而实际操作正是通过这几个后缀实现的。

IA	IB	DA	DB
先操作再增加地址	先增加地址再操作	先操作在减小地址	先减小地址再操作

如果想要正确地保存和恢复寄存器，需要 LDM 和 STM 的后缀对应使用：

	STMIA	STMIB	STMDA	STMDB
LDMIA	不对应	不对应	不对应	满减栈
LDMIB	不对应	不对应	空减栈	不对应
LDMDA	不对应	满增栈	不对应	不对应
LDMDB	空增栈	不对应	不对应	不对应

6410 的 C 编译器使用的是满减栈，所以我们要用 STMDB 和 LDMIA 来配合完成寄存器的保存和恢复。

注 2：在保存寄存器时，后缀是 I（增）开头则往内存地址增加方向延伸堆栈，D（减）开头则往内存地址减小方向延伸堆栈。但是不管往哪个方向，寄存器列表在内存中的顺序都是一样的。读取寄存器时也是一样。

什么意思呢？就是说保存时不管后缀是什么，{寄存器列表} 里写在前面的寄存器在内存中的存放地址一定比写在后面的寄存器所在地址小；读取时也是一样，低地址内存一定会被还原到写在前面的寄存器，高地址内存一定会被还原到写在后面的寄存器。下面举个例子来说明：

```
LDR R13,=0x50300008          ;把 R13 赋值 0x50300008
STMIA R13,{R0-R1}        ;R0->[0x50300008~0x5030000B]，R1->[0x5030000C~0x5030000F]
STMDB R13,{R0-R1}        ;R0->[0x50300000~0x50300003]，R1->[0x50300004~0x50300007]
```

可以看见，不管是递增栈还是递减栈存储方式，R0 始终处于低地址，R1 始终处于高地址。读取时也是一样：

```
LDMIA R13,{R0-R1}        ;[0x50300008~0x5030000B]->R0，[0x5030000C~0x5030000F]->R1
LDMDB R13,{R0-R1}        ;[0x50300000~0x50300003]->R0，[0x50300004~0x50300007]->R1
```

注 3：先看一下下面这个例子吧：

```
STMDB R13! , {R0-R14}
```

这个语句的内容是把 R0~R14 以满减栈格式保存到 R13 所指的地方，并且操作完之后更新栈顶指针 R13 的值（R13 = R13 - 4*15）。那么保存在内存中的 R13 是多少呢？是原来的 R13 还是 R13-60 呢？答案是 R13-60。

要注意：不管是 STM**还是 LDM**，只要后面的基址寄存器有“!”，都是先对基址寄存器进行修改，然后再进行保存或恢复操作。

读取时的情况也一样：

```
LDMIA R13! , {R0-R14}
```

虽然加了“!”表示操作完之后修改栈顶指针，但是修改栈顶指针是在从内存读取数据之前完成的，修改完 R13 之后从内存中读取的 R13 把修改好的 R13 给覆盖了，所以它的执行结果和 LDMIA R13 , {R0-R14} 相同。

EXPORT 代码段名

【功能】声明一个外部可调用的代码段。汇编中的代码段通过此声明之后就可以在 C 代码中被调用（当然 C 中也要声明该函数）。

【例】

汇编中：

```
    EXPORT OSSwitch_s      ;对外声明函数 OSSwitch_s
OSSwitch_s
    ……
```

C 中：

```
void OSSwitch_s(void);          //声明函数，把 OSSwitch_s 当作无参数的空函数使用
```

IMPORT 变量名

【功能】引入 C 中定义的变量，通过此声明之后就可以访问 C 中定义的变量了。

【例】
C 中：
unsigned long a;
汇编中：
IMPORT a
LDR R0,=a ;获取 a 的地址

以上只介绍了后面完成任务切换时用到的几条指令，更多其他指令和指令的用法请参阅其他资料。

9.3 在 s3c6410 上实现任务切换

首先来明确一下完成任务切换要做的事情：

1、保存当前 CPU 寄存器到内存。

2、记录刚刚保存的寄存器在内存中的位置。

3、从内存的另一个地方恢复 CPU 寄存器。

来分析一下要如何完成这 3 点。

1、首先，CPU 寄存器保存在什么地方？理论上可以保存到内存的任何地方，只要能够找回来就行了。实际上为了方便，我们把 CPU 寄存器就保存在当前任务所使用的堆栈处，也就是 R13 指向的位置。

2、把 CPU 寄存器保存到了当前任务的堆栈，栈顶指针肯定会发生变化，并且要用一个变量记录保存后的栈顶位置，这样才能在下次切换到该任务时从内存中找到寄存器存放的位置。显然，每个任务都应有一个这样的变量用于在本任务挂起时记录本任务栈顶地址。

*为了能够实现通用型的任务切换函数，任务切换函数内部的执行不应直接与任务相关联，所以设置了一个 unsigned long **OSCurTaskStkPtrPtr 变量，这个变量指向“记录当前任务堆栈地址的指针”。在任务切换函数中，修改这个变量指向的变量也就修改了“记录当前任务栈顶地址的指针变量”。*

3、这一点比较简单，只要传入下一个任务的栈顶指针就可以知道寄存器的存放位置了。

也就是说，完成任务切换对外提供的接口有：记录当前任务栈顶地址的指针变量的地址（提供这个变量的地址给 OSCurTaskStkPtrPtr 用于在任务切换函数中修改它）、将要恢复的任务堆栈的栈顶地址。（当前任务保存地址无需传入，直接保存到 R13 寄存器所指向的位置）

接下来使用的方法中对上述参数的传递并不是通过函数参数方式传递的，而是定义全局变量的方式。需要强调，这些全局变量都是在 C 中定义的，因为 C 编译器会为整个系统分配内存，在汇编中不能确定哪些内存是空闲的。

下面就来看一下任务切换部分的代码：

C 中：

```
void OSSwitch_s(); /*函数声明，汇编代码提供的函数保存当前寄存器至当前堆栈，
                     并从 OSNextTaskStkPtr处恢复寄存器      */

unsigned long **OSCurTaskStkPtrPtr;  /*在进行任务切换时，此变量所指向的变量将会被赋
```

```
                                        予当前任务的新的栈顶指针。所以在进行任务切换之
                                        前， 要让此变量指向原任务的栈顶指针，这样才能
                                        正确地更新该任务的栈顶指针，保证恢复时的正确性。
                                        */
    unsigned long *OSNextTaskStkPtr; /*将要恢复的任务的栈顶指针。在任务切换前，把将要
                                       恢复的任务的栈顶地址赋给此变量，在进行任务切换时
                                       就会从该处恢复 CPU 寄存器。   */
```

汇编中：

```
    OSSwitch_s     ;代码段定义完成任务间切换，先把当前寄存器保存到OSCurTaskStkPtr位置（也
                   ;就是SP指;针的位置），然后通过OSCurTaskStkPtrPtr修改OSCurTaskStkPtr指
                   ;向新的栈顶，最后从OSNextTaskStkPtr处恢复下一个任务的寄存器
        ;先保存当前寄存器
        STMDB R13,{R0-R14} ;注意，这里没有用“R13!”，因为那样的话对于R13寄存器来说保存的
                           ;将是R13-0x3c的值，而不是原来栈顶地址的值
        SUB R13,R13,#0x3c  ;保存了个寄存器，栈顶指针减小（x3c）
        MRS R0,CPSR        ;读取CPSR寄存器到R0，准备保存
        STMDB R13!,{R0}    ;保存状态寄存器

        ;下面开始更新原来任务的栈顶指针
        LDR R0,=OSCurTaskStkPtrPtr   ;获取指向当前任务栈顶指针的指针的地址，
                                     ;也就是OSCurTaskStkPtrPtr的地址
        LDR R1,[R0]        ;获取OSCurTaskStkPtrPtr的值，也就是OSCurTaskStkPtr的地址
        STR R13,[R1]       ;修改当前栈顶指针OSCurTaskStkPtr，使其指向最后一个存入的CPSR
                           ;寄存器所在内存单元

        ;下面开始恢复下一个任务的寄存器
        LDR R0,=OSNextTaskStkPtr     ;获取下一个任务的栈顶指针的地址，
                                     ;也就是OSNextTaskStkPtr的地址
        LDR R13,[R0]       ;获取下一个任务的栈顶指针，也就是OSNextTaskStkPtr的值
        LDMIA R13!,{R0}    ;从内存中获取CPSR的值，保存到R0寄存器
        MSR CPSR_fsxc,R0   ;恢复CPSR的值
        LDMIA R13,{R0-R14} ;恢复R0-R14，此时R13恢复到该任务上次保存前的值

        BX R14     ;跳转到R14，由于编译器在调用OSSwitch_s时已经把下一个指令地址保存
                   ;在了R14，所以跳转到R14即可
```

上述代码提供了一个 OSSwitch_s 函数。

在调用该函数前：

先给 OSNextTaskStkPtr 赋值下一个任务的栈顶地址；

再给 OSCurTaskStkPtrPtr 赋值“记录当前任务栈顶地址的指针”的地址；

然后调用 OSSwitch_s() 函数。

就可：

把当前 CPU 寄存器保存到 R13 指向的地方（满减栈格式保存）；

给 OSCurTaskStkPtrPtr 指向的内存单元赋值，赋上原任务新的栈顶地址（满减栈格式）；

从 OSNextTaskStkPtr 指向的内存单元读取数据，恢复 CPU 寄存器；

跳转到 R14 指向的位置运行。

以上是完成任务切换的最基本功能，要实现多个任务的运行还需构建一些其他逻辑结构来管理各个任务，这就是操作系统内核。

9.4 在 s3c6410 上实现简单操作系统

本节将利用上一节做好的任务切换函数，实现一个很小的操作系统。这个操作系统的功能很简单，用于调度两个任务来回切换。

9.4.1 资源获取

首先，系统要调配两个任务，肯定要有一个变量来记录当前正在运行的任务：

```
unsigned long CurTaskNum;   //当前正在运行的任务的序号，1或2
```

另外，上一节已经说过，每个任务都应有一个自己的栈顶指针，用于记录当本任务在挂起状态时的栈顶地址，所以有如下两个变量：

```
unsigned long *Task1StkPtr; //用于保存任务1在挂起状态时的栈顶地址
unsigned long *Task2StkPtr; //用于保存任务2在挂起状态时的栈顶地址
```

每个任务的运行都要有一个自己的堆栈空间，所以每个任务要申请一段内存空间给自己用：

```
#define Task1StkSize 200    //单位是字节
#define Task2StkSize 200    //单位是字节
unsigned long Task1Stk[Task1StkSize/4];   //用于给任务1作堆栈的数组
unsogned long Task2Stk[Task2StkSize/4];   //用于给任务2作堆栈的数组
```

以上是每个任务所需的资源，这些资源都是由 C 编译器来分配地址的。

9.4.2 任务第一次运行和堆栈初始化

下面要讨论每个任务第一次运行的问题：

由于整个工程是用 C 编译器编译的，所以系统一开始还是从 main 函数开始运行，在 main 函数完成初始化等工作之后，我们会把 CPU 切换到任务 1 里去（而不像传统的单片机编程是一个死循环）。切换到任务 1 里之后，只要跳转指令跳转回 main 函数的话，那么 CPU 就再也不会回到 main 函数了。

我们现在要重点关注从 main 函数跳转到任务 1 时做了些什么？这虽然也是任务切换（从 main 任务切换到任务 1），但我们可以用之前的 OSSwitch_s 函数吗？

如果想用 OSSwitch_s 函数的话，通过一些准备也是可以的。但我们一般不这么做，因为 main 函数在跳转之后就被抛弃了，我们没必要保存 main 任务的 CPU 寄存器，也没必要更新 main 任务的栈顶指针（事实上根本没有为 main 任务准备变量来记录栈顶地址），我们要做的只是直接跳转到任务 1 就行了。

所以在汇编代码中除了 OSSwitch_s 函数外，还写了一个 OSStart_s 函数：

```
OSStart_s;指定下一个要恢复的任务的栈顶地址，不保存当前寄存器，直接从该处恢复寄存器
    LDR R0,=OSNextTaskStkPtr     ;获取下一个任务的栈顶指针的地址，
                                 ;也就是OSNextTaskStkPtr的地址
    LDR R13,[R0]   ;获取下一个任务的栈顶指针，也就是OSNextTaskStkPtr的值
    LDMIA R13!,{R0}     ;从内存中获取CPSR的值，保存到R0寄存器
    MSR CPSR_fsxc,R0    ;恢复CPSR的值
    LDMIA R13,{R0-R14} ;恢复R0-R14，此时R13恢复到该任务上次保存前的值

    BX R14     ;跳转到 R14，堆栈初始化时应当把 R14 处设置为任务地址，所以跳转到 R14
               ;即可
```

这个 OSStart_s 函数不保存当前 CPU 寄存器，直接从 OSNextTaskStkPtr 指向的位置恢复寄存器。

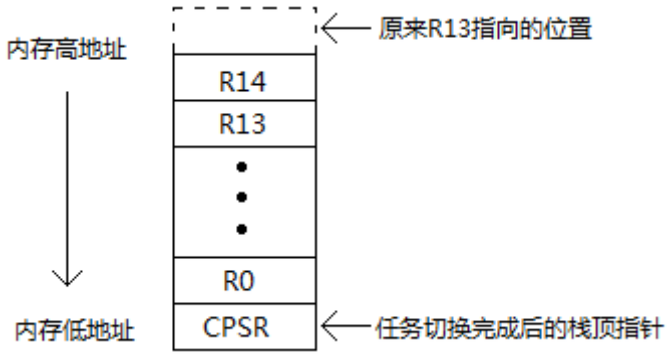
现在又来一个问题，任务 1（和任务 2）在第一次被恢复时并没有被保存过，它们堆栈中的数据是无效的，直接恢复肯定是错误的。所以在每个任务创建的时候都要对其堆栈进行初始化，使其看起来就像是被保存过一样，保证任务恢复时能够正确运行。

而如何进行初始化，我们就要来分析一下 CPU 寄存器在堆栈中的保存结构。

先回过头看一下保存 CPU 寄存器的汇编代码：

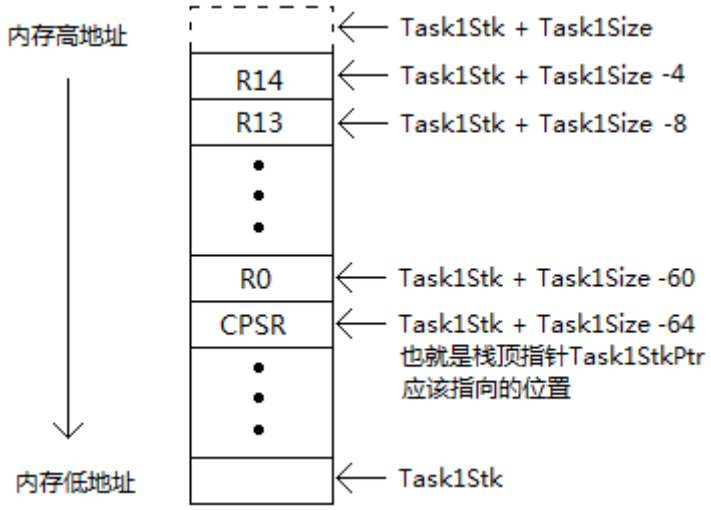
```
;先保存当前寄存器
STMDB R13,{R0-R14} ;注意，这里没有用“R13!”，因为那样的话对于R13寄存器来说保存的
                   ;将是R13-0x3c的值，而不是原来栈顶地址的值
SUB R13,R13,#0x3c  ;保存了个寄存器，栈顶指针减小（x3c）
MRS R0,CPSR        ;读取CPSR寄存器到R0，准备保存
STMDB R13!,{R0}    ;保存状态寄存器
```

通过代码可知，在内存中从高地址到低地址保存的内容依次是 R14-R0:CPSR，如下图：



此外，由于堆栈是满减栈，而申请的堆栈数组 Task1Stk[Task1Size/4] 却是往内存大的地方延伸的，所以 Task1Stk + Task1Size-1 才是申请到的堆栈数组的最高地址。

综上，在任务没有运行之前，整个堆栈在内存的内容应该是这样的：



上面说的是结构，知道了 CPU 的各个寄存器保存在什么地方。下面再来看一下内容，哪些是需要初始化的，应该初始化为什么内容。

1、首先，记录任务 1 栈顶地址的指针 Task1StkPtr 应该指向 CPSR 的位置，也就是 Task1Stk + Task1Size - 64。

2、R14 是 LR 指针，任务恢复函数的最后要跳转到该处运行。在此我们当然希望它跳转到任务 1 的函数，所以 Task1Stk + Task1Size - 4 处应该赋值为任务 1 的任务函数地址。

3、R13 是栈顶指针，在任务恢复后，它应当是任务堆栈的栈顶。而我们申请到的堆栈的最高地址为 Task1Stk + Task1Size - 1，由于堆栈是满减栈格式，所以 R13 应当等于 Task1Stk +Task1StkSize。所以 Task1Stk + Task1Size – 8 处应赋值为 Task1Stk + Task1Size。

4、CPSR 状态寄存器里面有一些设置项。根据 9.1 节的寄存器介绍，我们应把它赋值为 0x000000D3，也就是 ARM 状态（32 位指令）、Supervisor（管理）模式。（因为使用的是 J-Link 连接运行程序的，所以用管理模式）

综上，我们可以写出任务 1 的堆栈初始化函数：

```
void OSCreateTask1(void (*TaskPtr)(void),unsigned long *TaskArrayPtr,unsigned long
TaskStkSize)
{
    *((unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-4))=(unsigned
long)TaskPtr; //修改保存R14处的内存单元，使其指向函数所在地址
    *((unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-8))=(unsigned
long)(unsigned long)TaskArrayPtr+TaskStkSize;  //修改保存R13处的内存单元
    *((unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-64))=0x000000d3;    //
```

```
由于是在J-Link调试状态，所以是管理模式
    Task1StkPtr=(unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-64); //更新号
任务的栈顶指针
}
```

同理，任务 2 的堆栈初始化函数：

```
void OSCreateTask2(void (*TaskPtr)(void),unsigned long *TaskArrayPtr,unsigned long
TaskStkSize)
{
    *((unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-4))=(unsigned
long)TaskPtr; //修改保存R14处的内存单元，使其指向函数所在地址
    *((unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-8))=(unsigned
long)(unsigned long)TaskArrayPtr+TaskStkSize;  //修改保存R13处的内存单元
    *((unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-64))=0x000000d3;    //
由于是在J-Link调试状态，所以是管理模式
    Task2StkPtr=(unsigned long *)((unsigned long)TaskArrayPtr+TaskStkSize-64); //更新号
任务的栈顶指针
}
```

9.4.3 内核提供的任务切换函数

汇编代码已经提供了任务切换函数 OSSwitch_s 和任务开始函数 OSStart_s。但是这两个函数完成的是最基本的切换功能。

为了管理任务，内核在此功能基础上封装了新的任务切换函数 OSSwitch() 和任务开始函数 OSStart()。这两个函数除了调用底层的 OSSwitch_s() 和 OSStart_s() 之外，还根据内核的管理逻辑进行一些其他操作。

内核提供的任务切换函数：

```
void OSSwitch()
{
    if(CurTaskNum==1)  //当前正在运行号任务
    {
        OSCurTaskStkPtrPtr=&Task1StkPtr; /*把当前正在运行任务的栈顶指针的地址赋给
                                OSCUrTaskStkPtrPtr，汇编代码提供的任务切换函数在切
                                换完毕时会更新这个栈顶指针的值。 */
        OSNextTaskStkPtr=Task2StkPtr;//把将要恢复的堆栈指针赋给OSNextTaskStkPtr
        CurTaskNum=2; //标志接下来正在运行的任务是号
    }
    else if(CurTaskNum==2)
    {
        OSCurTaskStkPtrPtr=&Task2StkPtr; /*把当前正在运行任务的栈顶指针的地址赋给
                                OSCUrTaskStkPtrPtr，汇编代码提供的任务切换函数在切
                                换完毕时会更新这个栈顶指针的值。 */
        OSNextTaskStkPtr=Task1StkPtr;//把将要恢复的堆栈指针赋给OSNextTaskStkPtr
        CurTaskNum=1; //标志接下来正在运行的任务是号
    }

    OSSwitch_s();
}
```

内核提供的任务开始函数：

```
void OSStart()
{
    CurTaskNum=1; //接下来将要运行号任务
    OSNextTaskStkPtr=Task1StkPtr;    //把号任务的堆栈指针赋给OSNextTaskStkPtr
    OSStart_s();
}
```

任务运行时进行任务切换都是通过调用内核提供的切换函数完成的。

9.4.4 任务的实现

接下来倒数第二个问题，这属于应用层了，也就是具体的任务如何完成，以及如何使用内核提供的函数。

在此以控制两个 led 灯的任务为例。笔者使用的是 Tiny6410 开发板，有 4 个 led 灯可用，分别位于 GPK4~GPK7 引脚上。

需要做的事有：

1、在 C 语言中申请堆栈；

2、初始化 IO 口；

3、初始化堆栈；

4、OSStart()

以下是完成任务的文件代码：

```
/************************和系统相关************************/
#define Task1StkSize 200    //单位是字节
#define Task2StkSize 200    //单位是字节
unsigned long Task1Stk[Task1StkSize/4],Task2Stk[Task2StkSize/4];
void OSCreateTask1(void *(void),unsigned long *,unsigned long);
void OSCreateTask2(void *(void),unsigned long *,unsigned long);
void OSStart();
void OSSwitch();

/************************以下是任务相关*********************/
#define GPKCON0 (*((volatile unsigned long *)0X7F008800))    //定义IO设置寄存器的地址
#define GPKCON1 (*((volatile unsigned long *)0X7F008804))    //定义IO设置寄存器的地址
#define GPKDATA (*((volatile unsigned long *)0X7F008808))    //定义IO数据寄存器的地址

void led_init()    //led初始化
{
    GPKCON0=0x11110000;    //设置GPK4~GPK7为输出
}

void delay(unsigned long n) //软件延时
{
    unsigned long i;
    for(;n>0;n--)
        for(i=100000;i>0;i--);
}

void Task1()  //任务的任务函数
{
    while(1)
    {
        GPKDATA&=0xffffffef;    //点亮GPK4的led
        delay(100);
        GPKDATA|=0x00000010;    //熄灭GPK4的led
        delay(100);
        OSSwitch();   //任务切换
    }
}

void Task2()  //任务的任务函数
{
    while(1)
    {
        GPKDATA&=0xffffff7f;    //点亮GPK7的led
        delay(100);
        GPKDATA|=0x00000080;    //熄灭GPK7的led
        delay(100);
        OSSwitch();   //任务切换
    }
}

void main()
{
    led_init();   //led初始化
    OSCreateTask1(Task1,Task1Stk,Task1StkSize);    //任务堆栈初始化
    OSCreateTask2(Task2,Task2Stk,Task2StkSize);    //任务堆栈初始化

    OSStart();    //开始任务，跳转到任务运行
```

```
}
```

注 1：

以上实现的是一个非常简单的内核，没有考虑中断。复杂的操作系统有复杂的任务管理逻辑，可添加时间管理、信号量等功能，也可实现抢占式内核。

但是不管操作系统如何复杂，它都是利用最底层的任务切换功能实现的，内核就是在这个功能基础上建立的上层控制系统。

注 2：这个操作系统的工程可以在附件中找到。

参考文献

《C 程序设计》 第三版，谭浩强
《C++程序设计》 第二版，谭浩强
STC12C5A60S2 数据手册
《MicroC/OS-II, The Real-Time Kernel》 Second Edition，Jeal J.Labrosee
《嵌入式实时操作系统 uC/OS-II》 第二版，邵贝贝等译
《底层工作者手册》，Ifreecoding，http://blog.sina.com.cn/u/2425202502
《ARM1176JZF-S Technical Reference Manual》（ARM1176JZF-S 核的数据手册）
《S3C6410X RISC Microprocessor》（S3C6410 数据手册）