湖南大學

HUNAN UNIVERSITY

嵌入式计算机系统实验一报告

目录

一、实验目标、内容、原理

（一）实验目的

1.学习软件设计的层次化抽象方法并实践；

2.实验具体目标：以简单"无人驾驶汽车"应用场景为例，设计具有同时控制三个步进功能的程序模块（分别对应"转向"、"刹车"、"油门"控制），该模块能提供适当的 API 函数（一个或多个），方便应用层程序调用编写控制电机程序；

3.获取通过查阅手册解决实际问题的能力。

（二）实验内容

1．自行查找资料，学习并熟悉四线步进电机控制方法（以 28BYJ-48 型步进电机为例）；

2．阅读所提供的 displayer 模块参考源程序和其 API 函数设计方法，规划构造控制三个步进电机程序模块的 API 函数（一个或多个函数）。三个步进电机分别假设为：SM 接口（可以 LED 灯模拟，测试时应接 1 个真实步进电机）、8 个 LED 指示灯（假设为另外 2 个步进电机控制接口（模拟））；

3.（可以基于原 StepMotor 定义进行功能扩充）从模块的初始化程序、后台服务程序（即驱动）、应用层接口 API 函数三个层次设计该模块对应程序，最终实现（个人所规划 API 函数并验证 API 函数的正确性。（所编写的程序模块可根据需要，加载到模板任何模块上，如各种回调函数或主函数中，不硬性要求按模板方式封装，比如说，可以将步进电机与导航键结合起来，导航键的调用可自行实现；当然如果实现了模板方式封装，有加分）。

4．注意：本次实验不可使用小学期中应用的 BSP。

（三）实验原理

1.基本知识

步进电机是数字控制电机，它将脉冲信号转变成角位移，即给一个脉冲信号，步进电机就转动一个角度，因此非常适合于单片机控制。步进电机可分为反应式步进电机（简称 VR）、永磁式步进电机（简称 PM）和混合式步进电机（简称 HB），后两种常用。

步进电机控制特点：

它是通过输入脉冲信号来进行控制的。

电机的总转动角度由输入脉冲数决定。

电机的转速由脉冲信号频率决定。

2.步进电机 28BYJ-48 工作原理

28BYJ-48 型步进电机的旋转是以固定的角度一步一步运行的。可以通过控制脉冲个数来控制角位移量，从而达到准确定位的目的；同时可以通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的。控制步进电机定子绕组的通电顺序可以控制步进电机的转动方向。

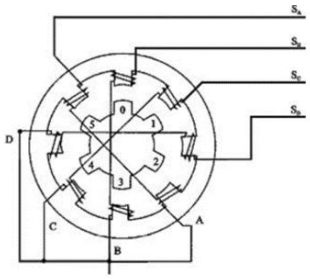


图 1 四相步进电机步进示意图

2. 28BYJ-48步进电机的驱动：

28BYJ-48 步进电机有多种减速比：1:64、1:32、1:16，以我公司的 28BYJ-48 步进电机为例，其参数如下表所示：

型号	电压	相数	步距角	减速比
28BYJ-48	5V	4	5.625/16	1:16

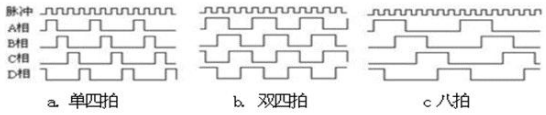
序号	颜色	描述
1	红	+5V
2	橙	A
3	黄	B
4	粉	C
5	蓝	D

该步进电机为四相八拍步进电机，采用单极性直流电源供电。只要对步进电机的各相绕组按合适的时序通电，就能使步进电机步进转动，图 1 是该四相反应式步进电机工作原理示意图。

开始时，开关 SB 接通电源，SA、SC、SD 断开，B 相磁极和转子 0、3 号齿对齐，同时，转子的 1、4 号齿就和 C、D 相绕组磁极产生错齿，2、5 号齿就和 D、A 相绕组磁极产生错齿。

当开关 SC 接通电源，SB、SA、SD 断开时，由于 C 相绕组的磁力线和 1、4 号齿之间磁力线的作用，使转子转动，1、4 号齿和 C 相绕组的磁极对齐。而 0、3 号齿和 A、B 相绕组产生错齿，2、5 号齿就和 A、D 相绕组磁极产生错齿。依次类推，A、B、C、D 四相绕组轮流供电，则转子会沿着 A、B、C、D 方向转动。

四相步进电机按照通电顺序的不同，可分为单四拍、双四拍、八拍三种工作方式。单四拍与双四拍的步距角相等，但单四拍的转动力矩小。八拍工作方式的步距角是单四拍与双四拍的一半，因此，八拍工作方式既可以保持较高的转动力矩又可以提高控制精度。单四拍、双四拍与八拍工作方式的电源通电时序与波形分别如图 2.a、b、c 所示：



a. 单四拍　　b. 双四拍　　c 八拍

图 2.步进电机工作时序波形图

旋转角度的算法：给予一个脉冲，该步进电机内部转子旋转 5.625 度，由于自带减速齿轮组，故外部主轴旋转角度为 5.625/减速比，根据要转动的角度即可推算出脉冲数。

3.步进电机的励磁方式

（1）一相励磁

在每一瞬间，步进电机只有一个线圈导通。每送一个励磁信号，步进电机旋转 5.625°，这是三种励磁方式中最简单的一种。

开始时，开关 SB 接通电源，SA、SC、SD 断开，B 相磁极和转子 0、3 号齿对齐，同时，转子的 1、4 号齿就和 C、D 相绕组磁极产生小角度错齿，2、5 号齿就和 D、A 相绕组磁极产生大角度（前小角度的 2 倍）错齿。

当开关 SC 接通电源，SB、SA、SD 断开时，由于 C 相绕组的磁力线和 1、4 号齿之间磁力线的作用，使转子转动，1、4 号齿和 C 相绕组的磁极对齐。而 0、3 号齿和 A、B 相绕组产生错齿，2、5 号齿就和 A、D 相绕组磁极产生错齿。依次类推，A、B、C、D 四相绕组轮流供电，则转子会沿着 A、B、C、D 方向转动。

表 7-3-2 一相励磁顺序表

STEP	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

（2）二相励磁

在每一瞬间，步进电机有两个线圈同时导通。每送一个励磁信号，步进电机旋转 5.625°。1/64。

表 7-3-3 二相励磁顺序表

STEP	A	B	C	D
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

（3）一-二相励磁

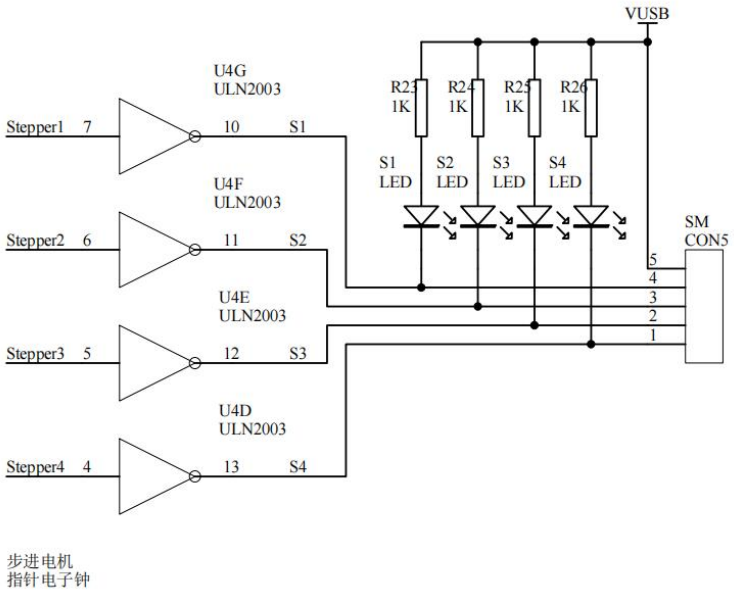
为一相励磁与二相励磁交替导通的方式。每送一个励磁信号，步进电机旋转 2.8125°。八拍模式是这类四相步进电机的最佳工作模式，能最大限度的发挥电机的各项性能。当相邻两相同时导通的节拍时，由于该两相绕组的定子齿对它们附近的转子齿同时产生相同的吸引力，这将导致这两个转子齿的中心线对到两个绕组的中心线上，也就使转动精度增加了一倍，还会增加电机的整体扭力输出。

表 7-3-4 一-二相励磁顺序表

STEP	A	B	C	D
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

4.电路图

学习板上步进电机相关电路如下图所示：



步进电机
指针电子钟

ULN2003 的工作原理是，当输入端接收到高电平时，对应的输出端会输出低电平，并且能够吸收 500mA 的电流。当输入端为低电平时，输出端则为高阻态，不会吸收电流。

因此当 stepper 端为 1 时，S 端输出低电平，LED 导通亮起，有电流通过，步进电机输入 1；当 stepper 端为 0 时，S 端输出高阻态，LED 截止，无电流通过，步进电机输入 0。

所以如果想要右端的 SM CON5 的 4-1 接受到 0001 信号，仅需在左端输入 0001 即可，无需取反。

5.按键消抖原理

4.1软件消抖

方法1：使用延时

如果按键较多，常用软件方法去抖，即检测出键闭合后执行一个延时程序，5ms～10ms的延时，让前沿抖动消失后再一次检测键的状态，如果仍保持闭合状态电平，则确认为真正有键按下。当检测到按键释放后，也要给5ms～10ms的延时，待后沿抖动消失后才能转入该键的处理程序。

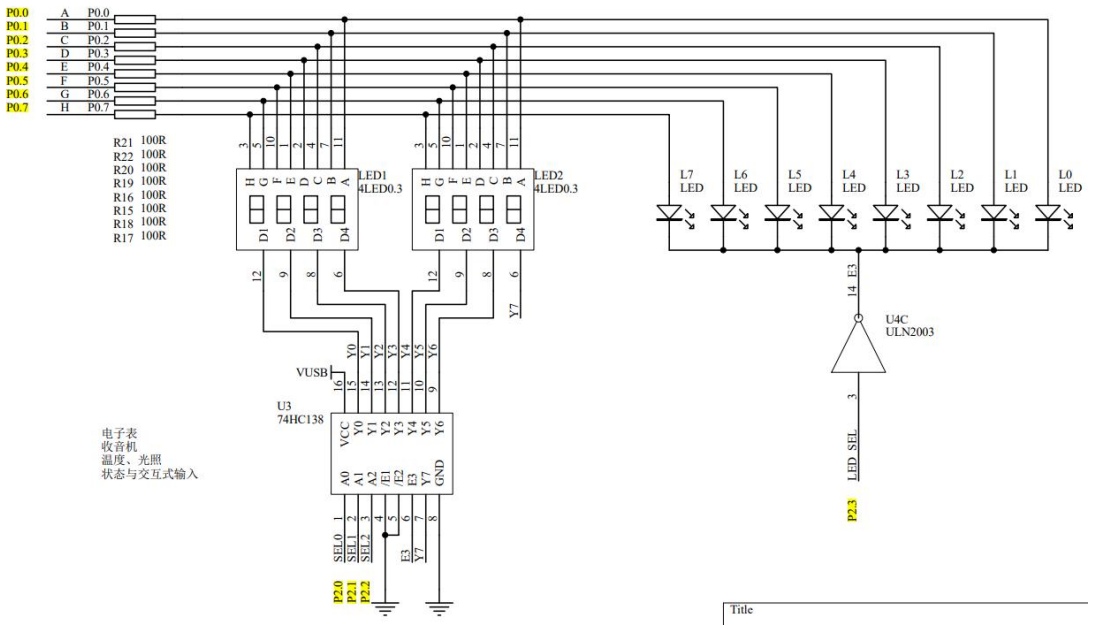
方法2：检测多次

可以设定一个检测周期，如果在一个检测周期内，按键被检测为被按下达到了一定次数，则确认为真正被按下。

在我的程序中，我使用了方法 2 检测多次，在计数器一定时间内，仅对按键进行一次计数。

二、驱动程序实例源码学习

（一）display



1. 初始化

设置 P0 为推挽模式，使输出的亮度更大。配置思路可见于数据手册，由于思路基本相同，后续端口配置推挽将不再赘述。

```
void DisplayerInit(void)
{
  P0M1 = 0x00;
  P0M0 = 0xff;              //设置P0为推挽模式:段选
  P2M1 &= 0xf0;
  P2M0 |= 0x0f;             //将P2^3、P2^2、P2^1、P2^0设置为推挽模式:位选
}
```

P0口设定〈P0.7, P0.6, P0.5, P0.4, P0.3, P0.2, P0.1, P0.0口〉(P0口地址：80H)

P0M1 [1 : 0] 寄存器P0M1地址为93H	P0M0 [1 : 0] 寄存器P0M0地址为94H	I/O 口模式
0	0	准双向口(传统8051 I/O 口模式), 灌电流可达20mA, 拉电流为270μA, 由于制造误差，实际为270uA～150uA
0	1	推挽输出(强上拉输出，可达20mA, 要加限流电阻)
1	0	仅为输入(高阻)
1	1	开漏(Open Drain)，内部上拉电阻断开。 开漏模式既可读外部状态也可对外输出(高电平或低电平)。 如要正确读外部状态或需要对外输出高电平，需外加上拉电阻，否则读不到外部状态，也对外输不出高电平。

```
举例:  MOV   P0M1, #10100000B
       MOV   P0M0, #11000000B
;P0.7为开漏,P0.6为强推挽输出,P0.5为高阻输入,P0.4/P0.3/P0.2/P0.1/P0.0为准双向口/弱上拉
```

2. 驱动函数

在 1ms 的回调函数中被调用。Switch 中的每种情况是对当前扫描位的端口设置，使得一次仅能显示一位，具体端口可见于电路图中标黄部分。加入 P23=1,P22-P20 为 0 的情况，可以使数码管和 LED 同时显示。

```
void myDrvDisplayer()     //动态扫描控制函数，针对Disp所描述。每1mS调用一次          //硬件驱动程序
{
  P0 =0;                                                  //关闭段选，消影
  switch (sys_Disp.Current_of_Scan)
      { case  0:    P23=0; P22=0; P21=0; P20=0; break;
        case  1:    P23=0; P22=0; P21=0; P20=1; break;
        case  2:    P23=0; P22=0; P21=1; P20=0; break;
        case  3:    P23=0; P22=0; P21=1; P20=1; break;
        case  4:    P23=0; P22=1; P21=0; P20=0; break;
        case  5:    P23=0; P22=1; P21=0; P20=1; break;
        case  6:    P23=0; P22=1; P21=1; P20=0; break;
        case  7:    P23=0; P22=1; P21=1; P20=1; break;
        default:    P23=1; break;
      }
  if (sys_Disp.Current_of_Scan <  8)
     P0 = decode_table[sys_Disp.Text[sys_Disp.Current_of_Scan]];  //段选：数码管时需译码
  else
    { if (sys_Disp.Current_of_Scan == 8) P0 = sys_Disp.Text[sys_Disp.Current_of_Scan];       //段选：流水灯时
      else P0=0;                                                                          //段选：空
    }
  if ( ++sys_Disp.Current_of_Scan > sys_Disp.Ending_of_Scan )                              //计算下一次要显示的位
    if (sys_Disp.Current_of_Scan <= 8) sys_Disp.Current_of_Scan=8;//使得LED和数码管可以同时显示
       else
         sys_Disp.Current_of_Scan = sys_Disp.Begin_of_Scan;
}
```

3. 应用接口 API

其余的是一些暴露给应用程序的接口 API，仅设置了一些需要用的相关参数。

```
void SetDisplayerArea(char Begin_of_scan,char Ending_of_Scan)                    //应用程序接口API
{   sys_Disp.Begin_of_Scan=Begin_of_scan;
    sys_Disp.Ending_of_Scan=Ending_of_Scan;
}

void Seg7Print(char d0,char d1,char d2,char d3,char d4,char d5,char d6,char d7)   //应用程序接口API
{ sys_Disp.Text[0]=d0;
  sys_Disp.Text[1]=d1;
  sys_Disp.Text[2]=d2;
  sys_Disp.Text[3]=d3;
  sys_Disp.Text[4]=d4;
  sys_Disp.Text[5]=d5;
  sys_Disp.Text[6]=d6;
  sys_Disp.Text[7]=d7;
}

void LedPrint(char led_val)                                                      //应用程序接口API
{   sys_Disp.Text[8]=led_val;
}
```

（二）sys

1. 初始化

相关定时器配置可见于数据手册第二章、第七章相关部分。

```
void sys_Timer0_Init(void)
{ AUXR |= 0x80;                          //定时器0工作在1T、16位自动重装初值、1mS定时模式
  TMOD &= 0xF0;
  TL0 = 65536-(SysClock/1000);
  TH0 = (65536-SysClock/1000)>>8;
  TR0 = 1;
  ET0=1;
}
```

```
void MySTC_Init()
{   sys_Timer0_Init();
    EA = 1;
}
```


2. 定时器/计数器工作模式寄存器TMOD

定时和计数功能由特殊功能寄存器TMOD的控制位C/$\overline{T}$进行选择，TMOD寄存器的各位信息如下表所列。可以看出，2个定时/计数器有4种操作模式，通过TMOD的M1和M0选择。2个定时/计数器的模式0、1和2都相同，模式3不同，各模式下的功能如下所述。

寄存器TMOD各位的功能描述

TMOD　地址：89H　复位值：00H

不可位寻址

7	6	5	4	3	2	1	0
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
定时器1				定时器0			

TCON：定时器/计数器中断控制寄存器（可位寻址）

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
TCON	88H	name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

```
void main()
{
        AUXR    |=      0x80;           //定时器0为1T模式
//      AUXR    &=      ~0x80;          //定时器0为12T模式

        TMOD    =       0x00;           //设置定时器为模式0(16位自动重装载)

        TMOD    &=      ~0x04;          //C/T0=0, 对内部时钟进行时钟输出
//      TMOD    |=      0x04;           //C/T0=1, 对T0引脚的外部时钟进行时钟输出

        TL0     =       F38_4KHz;       //初始化计时值
        TH0     =       F38_4KHz >> 8;
        TR0     =       1;
        INT_CLKO        =       0x01;   //使能定时器0的时钟输出功能

        while (1);                      //程序终止
}
```

2. 驱动函数

MySTC_OS 函数在主函数中被 while 循环调用，以达到定时回调的效果，它分别回调了 1、10、100ms 和 1s 的函数，这些函数中又分别调用了在 main 中暴露给用户的自定义函数。

```
void MySTC_OS()
{    if( flag_1mS )       { flag_1mS=0;       sys_task_1mS();       }
     if( flag_10mS )      { flag_10mS=0;      sys_task_10mS();      }
     if( flag_100mS )     { flag_100mS=0;     sys_task_100mS();     }
     if( flag_1S )        { flag_1S=0;        sys_task_1S();        }
}
```

该函数定义了一个中断历程，中断号为 1，表示是定时器 0 的溢出中断触发的，每次触发时它将 1ms 的 flag 重新设置为 1，以便于 MySTC_OS 函数中的调用。

```
void sys_Timer0_isr() interrupt 1                    //程序基本架构/任务调度/OS
{ flag_1mS = 1;                                      //每1mS
}
```

3. 应用接口

在 1ms 的回调函数中，它计数并设置了其他几个 ms 数的 flag，以实现了不同 ms 数的回调，然后它调用了驱动，在这里我对主函数 my1mS_callback 的调用代码换成了对驱动的调用，也可以放在主函数中去实现。

其他的回调函数均是对主函数的调用，相当于在用户调用之前增加了一层封装。

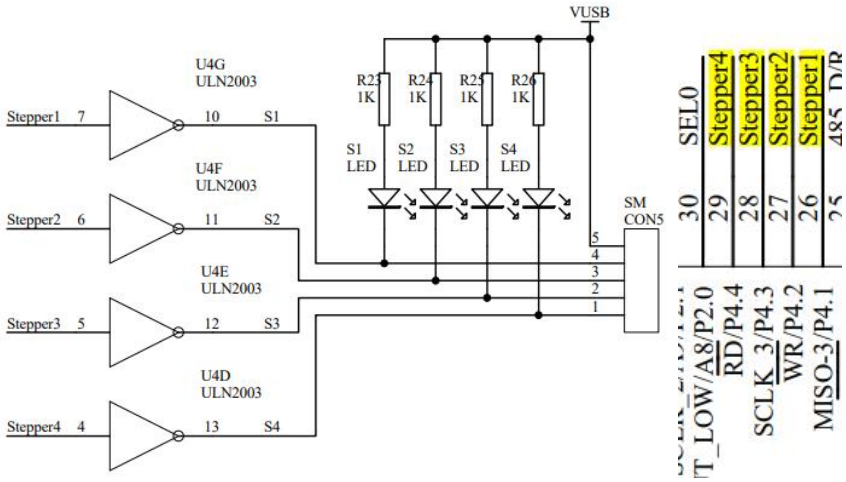
```
void sys_task_1mS()
{  if( --count_1mS == 0 )
      { count_1mS = 10;
        flag_10mS = 1;
          if( --count_10mS == 0 )
         { count_10mS = 10;
           flag_100mS = 1;
               if( --count_100mS == 0 )
            { count_100mS = 10;
              flag_1S = 1;
            }
         }
       }
    myDrvDisplayer();
    StepMotorDrv();          //显示驱动
}

void sys_task_10mS()
{  my10mS_callback();  //用户10mS回调
}

void sys_task_100mS()
{   my100mS_callback();        //用户100mS回调
}

void sys_task_1S()
{   my1S_callback();          //用户1S回调
}
```

三、程序设计

我写了五个版本，最终验收时给助教展示了三个。第一个版本简单实现了 stepmotor 的功能，以小学期提供的.h 文件格式（加入了其中没有的驱动函数）进行封装，除了实验要求的初始化、驱动、控制函数，还包括实验要求中没有给出的急停函数与获取状态函数，同时提供了 LED 可视化单步进电机的速度和剩余步数（可动态更新），并提供了步进电机的掉头和变速功能。

第二个版本在第一个的基础上，使用了 callback 的 BSP（因为没能写出 callback 函数以及不知道如何配置 callback 函数的第一个参数的状态，只能出此下策），实现了 adc 控制当前显示的步进电机切换与急停，以及 key 控制加减速功能。

第三个版本同样在第一个基础上，自己编写了 Key 的代码部分，并实现了按键消抖，使用 key 控制当前显示电机的加速以及减速，并未调用 BSP。



（一）版本一

1. stepmotor.h 文件

```
/****************************** stepmotor 说明 *********************************************

(1) 初始化函数：用于初始化I/O端口和步进电机控制相关的硬件资源
    void StepMotorInit(void);
(2) 驱动函数：用于控制步进电机的后台运行，确保按设定的模式顺序驱动步进电机旋转
    void StepMotorDrv(void);
(3) 应用层函数：提供给用户调用
      SetStepMotor(char StepMotor,unsigned char cinspeed,int cinsteps)  指定步进电机、按指定转动速度、转动指定步
          函数参数：StepMotor  指定步进电机，取值（enum StepMotorName中定义）
                enumStepMotor1:   SM 接口上的步进电机
                enumStepMotor2:   此时，用L0～L3四个LED模拟一个4相步进电机
                enumStepMotor3:   此时，用L4～L7四个LED模拟一个4相步进电机
                 speed      步进电机转动速度（0～255），单位：步/S。（实际每步时间=int(1000mS/speed) mS），与设置速度可能存在一定误差
                 steps      步进电机转动步数（-32768～32767），负值表示反转
          函数返回：enumSetStepMotorOK:   调用成功（enum StepMotorActName中定义）
                    enumSetStepMotorFail：调用失败（电机名不在指定范围，或speed=0,或调用时正在转动）
      EmStop(char StepMotor) 紧急停止指定步进电机转动
          函数参数：StepMotor  指定步进电机。函数参数不对将返回0值。
          函数返回：剩余未转完的步数
      GetStepMotorStatus(char StepMotor) 获取指定步进电机状态
          函数参数：StepMotor  指定步进电机
          函数返回：enumStepMotorFree:自由（enum StepMotorActName中定义）
                    enumStepMotorBusy,忙（正在转动）
                    enumSetStepMotorFail：调用失败（步进电机名不在指定范围）
      stepmotorreverse(char StepMotor) 令指定步进电机掉头
      stepmotorchangev(char StepMotor) 令指定步进电机变速
*/
```

```
#ifndef _stepmotor_h_
#define _stepmotor_h_

extern void StepMotorInit(void);
extern void StepMotorDrv(void);
extern char SetStepMotor(char StepMotor,unsigned char cinspeed,int cinsteps);
extern int EmStop(char StepMotor);
extern unsigned char GetStepMotorStatus(char StepMotor);
extern void stepmotorreverse(char StepMotor);
extern void stepmotorchangev(char StepMotor);

enum StepMotorName     {enumStepMotor1=0,enumStepMotor2,enumStepMotor3};
enum StepMotorActName {enumStepMotorFree,enumStepMotorBusy,enumSetStepMotorOK,enumSetStepMotorFail};

#endif
```

2. 初始化

仅是将对应端口控制为推挽模式，不再赘述。下图为用到的相应变量的声明。

```
void StepMotorInit(void){
    P4M1 &= 0xe1;        //11100001
    P4M0 |= 0x1e;        //00011110
    P4 &= 0xe1;          //11100001
}
```

```
xdata int speed[3]= {0, 0, 0};//每秒钟步进的数目  步/s
xdata int steps[3]= {0, 0, 0};//负值表示反转  步
xdata unsigned char rem=0;//计数
xdata unsigned int time[3]= {0, 0, 0};// 实际每步时间=int(1000mS/speed) mS
xdata unsigned char led=0;
xdata unsigned char i1=0,i2=0,elsestep=0;
```

3. 驱动函数

这里的 time 起到一个时间片的作用，当一个时间片用完时才进入函数执行。

对于真实的步进电机，在这里编号为 0，我们使用配置端口的方式对它进行驱动，这里使用的方法是一相励磁，具体原理见本文第一部分。四步作为一个轮回，rem 记录的是当前走到四步中的哪一步，每次调用时更新 rem，将步数减 1，并重置时间片。

```
void StepMotorDrv(void){
  if(speed[0] != 0){
    if (time[0]-- == 0 && steps[0] != 0)//已经过一步的时间间隔且还有步数未走
    {
      if (steps[0] > 0) //1号步进电机正转
      {
        switch (rem)
        {
        case 0:
          P41 = 1;P42 = 0;P43 = 0;P44 = 0;break;
        case 1:
          P41 = 0;P42 = 1;P43 = 0;P44 = 0;break;
        case 2:
          P41 = 0;P42 = 0;P43 = 1;P44 = 0;break;
        case 3:
          P41 = 0;P42 = 0;P43 = 0;P44 = 1;break;
        }
      if (rem != 3)
        rem++;
      else   rem = 0;
      steps[0]--;//步数减一
      time[0] = 1000 / speed[0];//重置时间片
      }
      else //1号步进电机反转
      {
        switch (rem)
        {
        case 0:
          P41 = 0;P42 = 0;P43 = 0;P44 = 1;break;
        case 1:
          P41 = 0;P42 = 0;P43 = 1;P44 = 0;break;
        case 2:
          P41 = 0;P42 = 1;P43 = 0;P44 = 0;break;
        case 3:
          P41 = 1;P42 = 0;P43 = 0;P44 = 0;break;
        }
      if (rem != 3)
        rem++;
      else   rem = 0;
      steps[0]++;
      time[0] = 1000 / speed[0];
      }
    }
  }
```

对于 LED 模拟的步进电机，我们仅仅需要修改 LED 的显示参数并进行其余相应参数的更新。比如，若当前为步进电机 1，且为正数，我们将当前 led 显示值与 0xf0 相与以清除后四位，并加 0x01 移动相应位数，该位数由 i1 指明。同样也是，i1 四个一轮，使用完自加。

```
if (speed[1] != 0){
  if (time[1]-- == 0 && steps[1]!= 0)
  {
    if (steps[1] > 0)
    {
      led = (led & 0xf0) + (0x01 << (il++ % 4));
      LedPrint(led);
      steps[1]--;
      time[1] = 1000 / speed[1];
    }
    else
    {
      led = (led & 0xf0) + (0x08 >> (il++ % 4));
      LedPrint(led);
      steps[1]++;
      time[1] = 1000 / speed[1];
    }
  }
}
```

```
if (speed[2] != 0){
  if (time[2]-- == 0 && steps[2]!= 0)
  {
    if (steps[2] > 0)
    {
      led = (led & 0x0f) + (0x10 << (i2++ % 4));
      LedPrint(led);
      steps[2]--;
      time[2] = 1000 / speed[2];
    }
    else
    {
      led = (led & 0x0f) + (0x80 >> (i2++ % 4));
      LedPrint(led);
      steps[2]++;
      time[2] = 1000 / speed[2];
    }
  }
}
```

4. 应用接口 API

（1）设置步数、速度和方向

修改对应参数即可。

```
char SetStepMotor(char StepMotor,unsigned char cinspeed,int cinsteps){
  switch (StepMotor)
  {
  case enumStepMotor1:
    steps[0] = cinsteps;
    speed[0] = cinspeed;
    return enumSetStepMotorOK;
  case enumStepMotor2:
    steps[1] = cinsteps;
    speed[1] = cinspeed;
    return enumSetStepMotorOK;
  case enumStepMotor3:
    steps[2] = cinsteps;
    speed[2] = cinspeed;
    return enumSetStepMotorOK;
  default:
    return enumSetStepMotorFail;
  }
}
```

（2）紧急制动

可返回剩余步数。

```
int EmStop(char StepMotor){//函数返回：剩余未转完的步数
  switch (StepMotor)
  {
  case enumStepMotor1:
    elsestep = steps[0];
    steps[0] = 0;
    return elsestep;
  case enumStepMotor2:
    elsestep = steps[1];
    steps[1] = 0;
    return elsestep;
  case enumStepMotor3:
    elsestep = steps[2];
    steps[2] = 0;
    return elsestep;
  default:
    return 0;
  }
}
```

（3）获取当前状态

```
unsigned char GetStepMotorStatus(char StepMotor){
  switch (StepMotor)
  {
  case enumStepMotor1:
    if (steps[0] == 0)
      return enumStepMotorFree;
    else
      return enumStepMotorBusy;
  case enumStepMotor2:
    if (steps[1] == 0)
      return enumStepMotorFree;
    else
      return enumStepMotorBusy;
  case enumStepMotor3:
    if (steps[2] == 0)
      return enumStepMotorFree;
    else
      return enumStepMotorBusy;
  default:
    return enumSetStepMotorFail;
  }
}
```

（4）掉头

由于步数的正负指示当前方向，取负即可。

```
void stepmotorreverse(char StepMotorrev){
    steps[StepMotorrev]=-steps[StepMotorrev];
}
```

（5）变速

设置了一个匀速增加、速度不变、再匀速减小到 0 的速度变化函数，这里的 stepcounter 由 100ms 回调函数修改。

```
void stepmotorchangev(char StepMotorcha){
    if(speedcounter<50 && speed[StepMotorcha]<255)
    speed[StepMotorcha]++;
    else if(speedcounter>100 && speed[StepMotorcha]>0)
    speed[StepMotorcha]--;
}
```

5. 主函数

（1）main

做了初始化，设置了三个电机值，步数范围为-32768-32767（16 位），而后做了一些相关测试。

```
void main() {

  DisplayerInit();
  StepMotorInit();

  SetDisplayerArea(0,7);
  Seg7Print(10,10,10,10,10,10,10,10);

  SetStepMotor(enumStepMotor1,5,32767);
  SetStepMotor(enumStepMotor2,5,-32767);
  SetStepMotor(enumStepMotor3,5,8000);

  //stepmotorreverse(enumStepMotor2);
  //EmStop(enumStepMotor3);
  Displaymotorstep(enumStepMotor2);
  //Displaymotorspeed(enumStepMotor2);
```

（2）显示当前电机的步数与速度

我将其放置在 1s 的回调函数中进行调用。

```
void Displaymotorspeed(char StepMotor)
{
  switch (StepMotor)
  {
  case enumStepMotor1:
    steppositive=speed[0];
    Seg7Print(10, 12, 1, 12, 10, steppositive / 100 % 10, steppositive / 10 % 10, steppositive % 10);
    break;
  case enumStepMotor2:
    steppositive=speed[1];
    Seg7Print(10, 12, 2, 12, 10, steppositive / 100 % 10, steppositive / 10 % 10, steppositive % 10);
    break;
  case enumStepMotor3:
    steppositive=speed[2];
    Seg7Print(10, 12, 3, 12, 10, steppositive / 100 % 10, steppositive / 10 % 10, steppositive % 10);
    break;
  }
}
```

```
void Displaymotorstep(char StepMotor)
{
  switch (StepMotor)
  {
  case enumStepMotor1:
    if(steps[0]<0) steppositive=-steps[0];
    else steppositive=steps[0];
    Seg7Print(12, 1, 12, steppositive / 10000 % 10, steppositive / 1000 % 10, steppositive / 100 % 10, steppositive / 10 % 10, steppositive % 10);
    break;
  case enumStepMotor2:
    if(steps[1]<0) steppositive=-steps[1];
    else steppositive=steps[1];
    Seg7Print(12, 2, 12, steppositive / 10000 % 10, steppositive / 1000 % 10, steppositive / 100 % 10, steppositive / 10 % 10, steppositive % 10);
    break;
  case enumStepMotor3:
    if(steps[2]<0) steppositive=-steps[2];
    else steppositive=steps[2];
    Seg7Print(12, 3, 12, steppositive / 10000 % 10, steppositive / 1000 % 10, steppositive / 100 % 10, steppositive / 10 % 10, steppositive % 10);
    break;
  }
}
```

```
void myIS_callback()
{
    Displaymotorstep(enumStepMotor2);
}
```

（3）实现加速与减速的平滑切换

我将其放置在 100ms 的回调函数中进行调用。

```
void myl00mS_callback()
{
    stepmotorchangev(enumStepMotor2);
    speedcounter++;
    //Displaymotorspeed(enumStepMotor2);
}
```

（二）版本二

在版本一的基础上，将给出增加的部分。

1. 主函数

（1）main

增加了 callback 的部分来实现按键和导航的回调，这是调库的唯一目的。

```
    SetEventCallBack(enumEventKey, mykey_callback);
    SetEventCallBack(enumEventNav, mynav_callback);
    SetEventCallBack(enumEventSyslmS,mylmS_callback);
    SetEventCallBack(enumEventSys10mS,myl0mS_callback);
    SetEventCallBack(enumEventSys100mS,myl00mS_callback);
    SetEventCallBack(enumEventSys1S,myls_callback);
```

（2）导航按键回调

上下用于切换当前步进电机，key3 用于指示急停。

```
void mynav_callback(){
  //切换步进电机
  if ((GetAdcNavAct(enumAdcNavKeyUp) == enumKeyPress) && (change < 2))
      change++;
  if ((GetAdcNavAct(enumAdcNavKeyDown) == enumKeyPress) && (change >0 ))
      change--;
  if (GetAdcNavAct(enumAdcNavKey3) == enumKeyPress) {
    if (change == 0) EmStop(enumStepMotor1);
    if (change == 1) EmStop(enumStepMotor2);
    if (change == 2) EmStop(enumStepMotor3);
  }
}
```

（3）按键回调

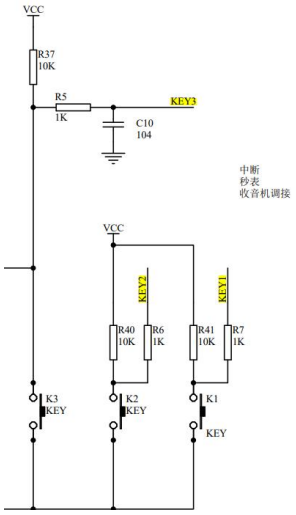
key1 加速，key2 减速。

```
void mykey_callback(){
  //key1-speedup
  if (GetKeyAct(enumKey1) == enumKeyPress) {
    if((change == 0) && (speed[0] < 255)) {speed[0]+=5; }
    if((change == 1) && (speed[1] < 255)) {speed[1]+=5; }
    if((change == 2) && (speed[2] < 255)) {speed[2]+=5; }
  }

  //key2-speeddown
  if (GetKeyAct(enumKey2) == enumKeyPress) {
    if((change == 0) && (speed[0] > 5)) {speed[0]-=5; }
    if((change == 1) && (speed[1] > 5)) {speed[1]-=5; }
    if((change == 2) && (speed[2] > 5)) {speed[2]-=5; }
  }
}
```

（三）版本三

在版本一的基础上，将给出增加的部分。

1. key



INT1/P3.3	21	KEY2
INT0/P3.2	20	KEY1
	19	TXD

（1）key.h

```
#ifndef _KARL_KEY_H_
#define _KARL_KEY_H_


#define enumKey1 0
#define enumKey2 1

#define enumKeyPress 0
#define enumKeyRelease 1

extern void Key_Init();
extern unsigned char GetKeyAct(char Key);

#endif
```

（2）初始化

设置推挽，相关配置可查看数据手册第六章。

```
void Key_Init()
{
    P3M1 |= 0x0c;
    P3M0 &= 0xf3;

    EX0 = 1;
    EX1 = 1;
    IT0 = 0;
    IT1 = 0;
}
```

IE： 中断允许寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IE	A8H	name	EA	ELVD	EADC	ES	ET1	EX1	ET0	EX0

EA： 中断允许总控制位
EA=0, 屏蔽所有的中断请求
EA=1, 开放中断, 但每个中断源还有自己的独立允许控制位。

ELVD： 低压检测中断允许位
ELVD = 0, 禁止低压检测中断
ELVD = 1, 允许低压检测中断

```
//定时器特殊功能寄存器
sfr TCON        =   0x88;   //0000,0000 T0/T1控制寄存器
sbit TF1        =   TCON^7;
sbit TR1        =   TCON^6;
sbit TF0        =   TCON^5;
sbit TR0        =   TCON^4;
sbit IE1        =   TCON^3;
sbit IT1        =   TCON^2;
sbit IE0        =   TCON^1;
sbit IT0        =   TCON^0;
sfr TMOD        =   0x89;   //0000,0000 T0/T1模式寄存器
```

（3）获取按键状态

```
unsigned char GetKeyAct(char Key)
{
    if (Key == enumKey1)
    {
        return key1_curr_act;
    }
    else if (Key == enumKey2)
    {
        return key2_curr_act;
    }
    else
    {
        return enumKeyRelease;
    }
}
```

（4）中断历程函数

数据手册：

```
void    Int0_Routine(void)      interrupt 0;
void    Timer0_Rountine(void)   interrupt 1;
void    Int1_Routine(void)      interrupt 2;
```

```
void Int0_Rountine() interrupt 0
{
    int0 = P32;
    if (int0 == 0)
    {
        key1_curr_act = enumKeyPress;
    }
    else
    {
        key1_curr_act = enumKeyRelease;
    }
    return;
}
```

```
void Int1_Rountine() interrupt 2
{
    int1 = P33;
    if (int1 == 0)
    {
        key2_curr_act = enumKeyPress;
    }
    else
    {
        key2_curr_act = enumKeyRelease;
    }
    return;
}
```

2. 主函数

（1）main

与第一版中的 callback 函数没什么不同，最终也没能实现 callback 的回调。

```
void main() {

  DisplayerInit();
  StepMotorInit();

  SetDisplayerArea(0,7);
  Seg7Print(10,10,10,10,10,10,10,10);

  //SetEventCallBack(enumEventKey,key_callback);

  SetStepMotor(enumStepMotor1,5,32767);
  SetStepMotor(enumStepMotor2,5,-32767);
  SetStepMotor(enumStepMotor3,5,8000);

  //stepmotorreverse(enumStepMotor2);
  //EmStop(enumStepMotor3);
  //Displaymotorstep(enumStepMotor2);
    Displaymotorspeed(enumStepMotor1);
```

（2）按键控制速度 && 按键消抖

最终也没能实现 callback 的回调，只能把状态检查和更新都放置在 100ms 回调中。首先调用了中断历程函数，然后对当前的按键状态进行检测，对当前显示的步进电机速度进行增加 5 或者减小 5。

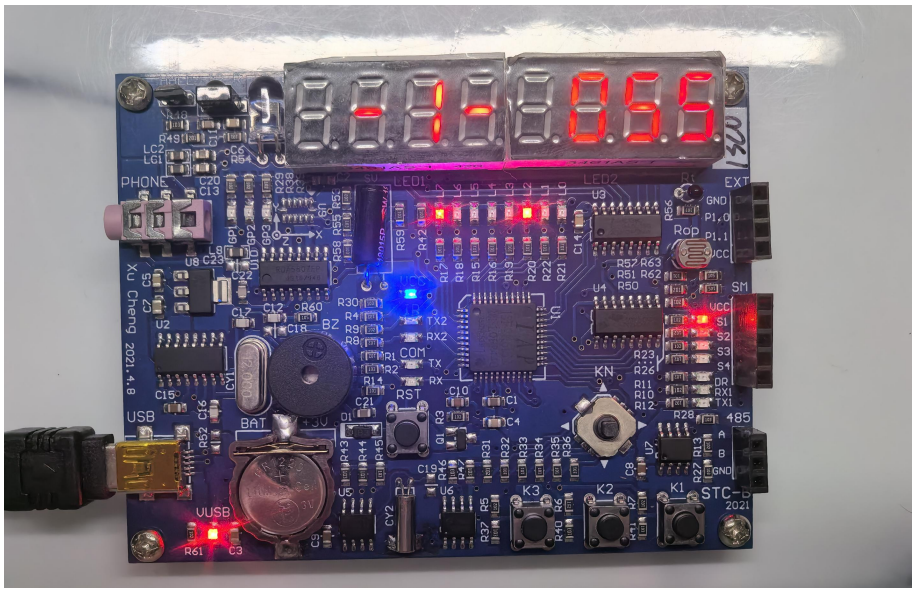
这里实现了按键消抖，使用了三个变量进行处理。对于 clock 变量，它是当前时钟的一个指示，我将其放置在 1ms 回调函数中进行自加；对于 flagclock1，它指示的是上一次调用按键自加时的时间，每次调用时，首先对二者的差进行检查，如果小于 200，才进行调用；每次离开函数前，要更新该变量。

```
void myl00mS_callback()                    //举例。100mS事件回调函数
{
    stepmotorchangev(enumStepMotor2);
    speedcounter++;
    Displaymotorspeed(enumStepMotor1);

    Int0_Rountine();
    Int1_Rountine();

    if(GetKeyAct(enumKey1)==enumKeyPress){
      if((clock-flagclock1)>200){
      if(speed[0] < 255){speed[0]+=5; }
      if(speed[1] < 255){speed[1]+=5; }
      if(speed[2] < 255){speed[2]+=5; }
      }
      flagclock1=clock;
    }
    if(GetKeyAct(enumKey2)==enumKeyPress){
      if((clock-flagclock2)>200){
      if(speed[0] > 5) {speed[0]-=5; }
      if(speed[1] > 5) {speed[1]-=5; }
      if(speed[2] > 5) {speed[2]-=5; }
      }
      flagclock2=clock;
    }

}
```

```
void sys_task_1mS()
{   if( --count_1mS == 0 )
       { count_1mS = 10;
         flag_10mS = 1;
           if( --count_10mS == 0 )
          { count_10mS = 10;
            flag_100mS = 1;
                if( --count_100mS == 0 )
             { count_100mS = 10;
               flag_1S = 1;
             }
          }
       }
    clock++;
    myDrvDisplayer();
    StepMotorDrv();          //显示驱动
    my1mS_callback();
}
```

四、实验结果

由于动态操作难以演示，放一张正常工作的图在这里。



五、实验中遇到的问题与难点

1. callback 和其第一个参数难以实现

这个问题最终也没有解决。验收时助教说有办法绕过这个函数实现功能，放在定时器 1ms 的回调里也可以，我的第三版里也是这个思路，但没有时间去实现更多的功能模块了，最后验收成绩不高。

2. 端口配置

参考数据手册中的介绍以及它提供的实例代码进行尝试。

六、实验心得

这次实验做得非常难受，花了不少时间，但结果非常令人失望。由于没能来得及自己实现 ADC 和导航模块，哪怕我做了三个版本且步进电机的部分加了多的功能，验收也没能拿到 B 及以上的评级。

我觉得实验验收的评分标准不太合理。首先实验要求中给出的实验内容只分了四点：1.查找资料学习步进电机控制方法。2.阅读参考代码规划 API 函数。3.实现 API 函数。4.不可使用 BSP。然后扩展思路也给了三点：1.实现加减速控制。2.功能模块组合，添加传感器数据以控制电机。3.远程控制。最后也提到了以上的扩展思路仅是举例，不要限制个人的思路。

按照以往其他课验收的经验来说，完成全部的实验要求就应该能拿到 80%到 90%的分数了，扩展思路属于额外的加分项，且也不需要全部完成，做一两个的分数就能拿到 90+，做出全部的扩展且验收很好的话可以拿到满分。也就是说，一般来说，实现了基本的三个步进电机功能＋一到两个扩展功能应该能拿到 90+。我一开始也是按照这个想法完成的（第一个版本），由于 callback 函数自己很难写出来，做了第二个调用 BSP 的版本以候选。

验收时间是下午 14:30 开始，当天中午 11:17 才发布验收标准，如下。

	步进电机	数码管	key	ADC	导航按键	其他扩展创新	是否允许使用BSP
A	√	√	√	√	√	√	否
B	√	√	√	√	√	×	否
C	√	√	√	√	×	×	是
D	√	√	√	×	×	×	是
E	×	×	×	×	×	×	是

这是否有些不太合理？首先验收标准并没有根据 实验的要求是否实现+扩展思路的项数 来进行给分，其次在实验要求并**没有提到按模块给分**的情况下，绝大多数的学生是不会想到做这么多模块的。据我所知，周围的几个成绩不错的同学也都没有准备这个部分，都是在实验标准出来之后瞳孔地震然后女娲补天。而且 可能 会出现在一个同学做出来了之后临时互相 copy 的情况。既然提前很多天就发了实验内容，说明是希望好好研究着做的，如果实验验收的结果是由发布验收标准后两三个小时是否空闲、是否效率高、甚至周围的人有没有做出来决定，那么这个实验验收的结果会不会太随机了？如果一开始就是打算按模块的数量给分的话，是否在实验要求中明确地说明更为合适？这样在刚开始做实验的时候就会把精力放在多做模块上而不是认为实验要求的步进电机才是主要部分了。

这是一门核心课，在小班课的课时不多的情况下，一个实验占 7.5 分，实验的验收部分就占了总评 4.5 分，且评级对分数的影响很大。大三的阶段竞争激烈到焦虑，每一分对我们来说都非常非常重要。希望下次的实验不会是同样的情况（鞠躬）。