上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

电气传动综合实验

实 验 报 告

姓名 谢弘洋

学号 515021910641

2018 年 12 月 27 日

实验一 BLDC 开环控制

1. 实验目标

a. 编写程序，实现 BLDC 无刷电机的启动、运行与停止。

b. 通过开发套件上的两个按钮外设，实现电机启动、停止和转动方向的控制。

c. 熟悉实验开发套件的基本配置和使用方式，熟悉软件编写与调试工具的使用。

2. 实验原理

2.1 BLDC 无刷直流电机的运行

不同于有刷直流电机，无刷直流电机（Brushless DC Motor）最大的特点在于其转子由永磁体构成，因此不再需要外界通以电流来产生转子磁场，也就克服了传统直流电机转子需要换向器与电刷从而带来的一系列寿命、噪声上的问题。同时，定子也相应地由三相对称分布的绕组组成，当每个绕组上通以正负电流时，可以产生对应方向上的正负磁场。通常，BLDC 的定子绕组采用两两通电方式，总共能够产生六种磁场方向的组合（如图1所示）。

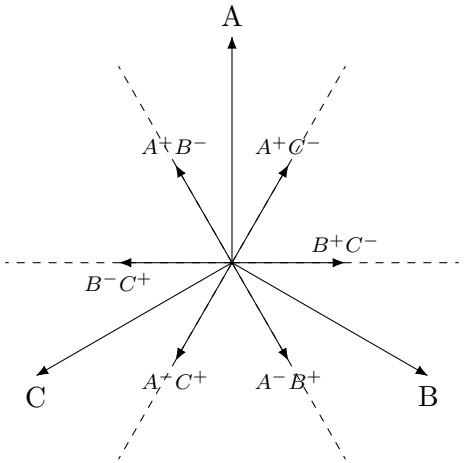

图 1: 定子绕组两两通电产生的磁场矢量组合

当定子磁场与转子磁场有一夹角时，就会产生磁场力矩，使两者有相互接近的趋势。而根据毕奥-萨法尔定律，磁场力与两个磁矢量之间夹角的正弦值成正比，因此当定子磁场方向与转子磁场方向垂直时，能够在相同的磁场大小（即定子绕组电流）下得到最大的力矩，提高电机的负载能力和效率。由此，可以按照这一准则将空间角度分为六个扇区，当转子磁场方向落在对应扇区时，即为对应的两个定子绕组同上对应方向的电流，产生与转子磁场方向正交的定子磁场，拖动转子旋转，如图2所示。

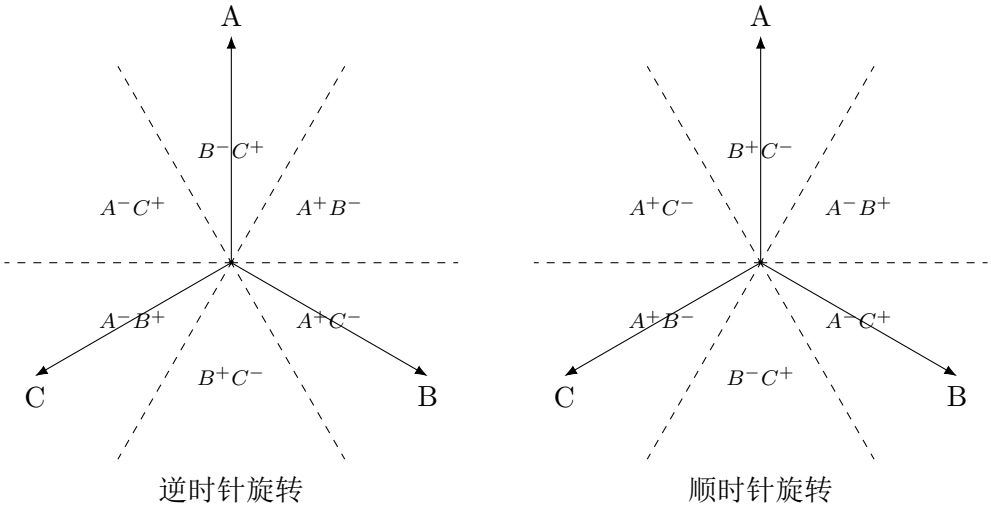

图 2: 转子磁矢量位于对应扇区时施加的定子磁场组合

2.2 转子位置检测

由于转子磁场有永磁体产生，因此转子磁矢量的方向仅与转子的角度位置有关，为了确定某一瞬间需要由定子绕组产生的磁场组合，只需要对转子磁场位置进行检测，再按照图2判断对应的扇区即可。本实验中通过霍尔传感器来检测转子磁场的位置，通过再 A、B、C 轴上安装的三个霍尔传感器，可以得到三个高低电平信号，当转子磁场进入或离开对应传感器所在的半周（180°）范围时，对应传感器的输出将发生电平反转，如图3所示。通过三个电平信号的组合进行判断，恰好可以定位出图2中的六个扇区。

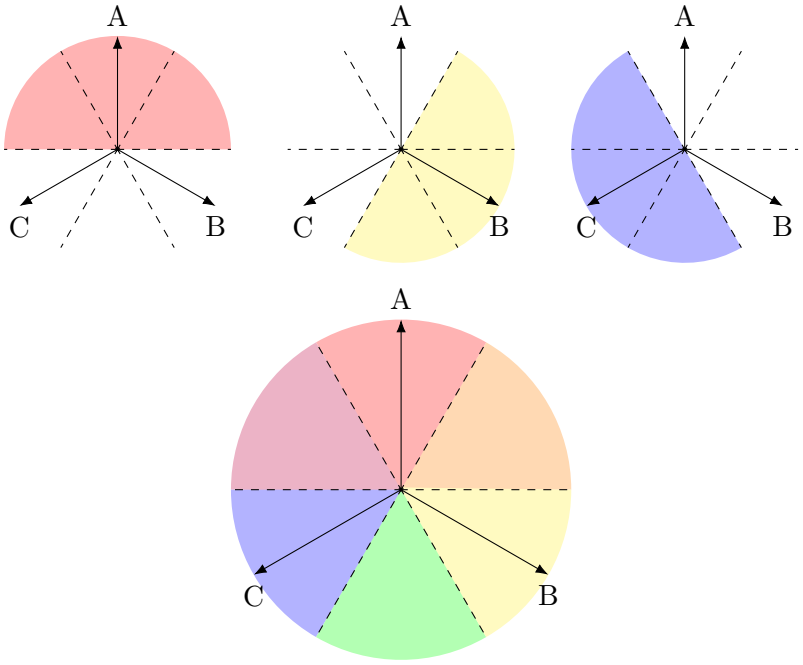

图 3: 通过霍尔传感器检测转子磁场位置

3. 实验过程

3.1 线路连接

将电机 A、B、C 三相电源线与开发板上三组 H 桥的输出端子连接，将电机三条霍尔传感器输出线与开发板三个 IO 端口连接，再为开发板连接上直流电源线即完成了电气部分连接。

此外，将开发板上的 RJ45 接口与仿真器连接，再将仿真器与计算机的 USB 口连接，即完成了调试通信部分连接。

3.2 程序实现

定子磁场的产生

通过处理器的六路 PWM 波输出控制三组 H 桥上下桥臂的六个开关管即可实现对 A、B、C 三相电压的控制，从而实现定子绕组电流方向控制，产生每个时刻需要的定子磁场组合。dsPIC33FJ32MC204 的 PWM 模块提供了 OVDCON 寄存器来方便使用者再不改变每路 PWM 输出的其他配置（如占空比）的情况下直接控制每路 PWM 信号的输出情况。

REGISTER 15-10: PxOVDCON: OVERRIDE CONTROL REGISTER(1)

U-0	U-0	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
—	—	POVD3H	POVD3L	POVD2H	POVD2L	POVD1H	POVD1L
bit 15							bit 8

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	POUT3H	POUT3L	POUT2H	POUT2L	POUT1H	POUT1L
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15-14 **Unimplemented:** Read as '0'

bit 13-8 **POVDxH<3:1>:POVDxL<3:1>:** PWM Output Override bits
1 = Output on PWMx I/O pin is controlled by the PWM generator
0 = Output on PWMx I/O pin is controlled by the value in the corresponding POUTxH:POUTxL bit

bit 7-6 **Unimplemented:** Read as '0'

bit 5-0 **POUTxH<3:1>:POUTxL<3:1>:** PWM Manual Output bits
1 = PWMx I/O pin is driven active when the corresponding POVDxH:POVDxL bit is cleared
0 = PWMx I/O pin is driven inactive when the corresponding POVDxH:POVDxL bit is cleared

通过对寄存器中的第 8-13 位置 1，可以使得对应的 PWM 正常输出，置 0，则使得对应 PWM 的输出由该寄存器中第 0-5 位决定。因此，可以得到产生前述六种磁场与对应的 OVDCON 寄存器值之间的关系如表1所示。（设 PWM1 对应 A 相，PWM2 对应 B 相，PWM3 对应 C 相）。

最终在输出的波形如图4所示，属于 H_PWM-L_ON 型调制方式。从而可

表 1: 定子磁场与 OVDCON 寄存器的对应关系

定子磁场组合	PWM1	PWM2	PWM3	OVDCON 值
A^+B^-	PWM	L	0	0x0204
B^-C^+	0	L	PWM	0x2004
A^-C^-	L	0	PWM	0x2001
A^-B^+	L	PWM	0	0x0801
B^+C^-	0	PWM	L	0x0810
A^+C^-	PWM	0	L	0x0210
零磁场	0	0	0	0x0000

以通过调节占空比，改变定子绕组的两端电压，从而调节转速。

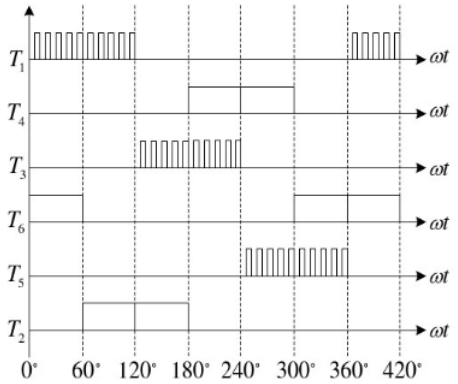

图 4: PWM 模块输出波形

转子位置检测

在任意时刻读取三路霍尔传感器的值即可得到该时刻转子磁场所在的扇区，再根据当前需要的旋转方向由图2和图3选择需要的定子磁场组合，再根据表1为 OVDCON 设置上相应的值即可完成电机运行的控制。在程序中，三相霍尔传感器的输入在寄存器中由高位到低位按照 CAB 的顺序依次排列，由此可得到控制电机旋转的逻辑关系如表2所示。

转速的开环控制

对于无刷直流电机，由于转子磁场由永磁体产生，因此只能通过调节定子磁场来改变力矩，从而改变转速。通过控制 PWM 波占空比能够改变每相的电流，也就能够改变转速。

在开环控制实验中，以开发板上的电位器旋钮作为输入源，通过 dsPIC33FJ32MC204 的一路 ADC 采样得到电位器的分压值，等比例地设置为 PWM 波占空比，即完成了电机转速地开环控制。

表 2: 霍尔传感器输入与 PWM 输出对应关系

霍尔传感器输入		逆时针转动		顺时针转动	
CAB	十进制	定子磁场	OVDCON	定子磁场	OVDCON
001	1	A^+C^-	0x0210	A^-C^+	0x2001
010	2	B^-C^+	0x2004	B^+C^-	0x0810
011	3	A^+B^-	0x0204	A^-B^+	0x0801
100	4	A^-B^+	0x0801	A^+B^-	0x0204
101	5	B^+C^-	0x0810	B^-C^+	0x2004
110	6	A^-C^+	0x2001	A^+C^-	0x0210

3.3 程序设计框图

主程序

主程序部分主要实现定时器、PWM、ADC、IO 模块的初始化与设置工作，之后进入死循环维持程序运行，并通过一系列的循环判断来实现开发板上两个按钮外设对电机启停和转向的控制。程序流程框图如5所示。

ADC 中断

在开环控制中，将 ADC 采集到的电位器分压值乘上一定系数后直接作为 PWM 的占空比，从而实现对转速的开环调节。中断内流程框图如图6所示。

3.4 代码清单及注释

SensoredBLDC.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"

_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & POSCMD_XT);

_FBS (BSS_NO_FLASH & BWRP_WRPROTECT_OFF);

_FWDT (FWDTEN_OFF);

_FGS (GSS_OFF & GCP_OFF & GWRP_OFF);

_FPOR (PWMPIN_ON & HPOL_ON & LPOL_ON & FPWRT_PWR128);

_FICD (ICS_PGD3 & JTAGEN_OFF);

void InitADC10(void);
void DelayNmSec(unsigned int N);
void InitMCPWM(void);
void InitTMR3(void);
void InitIC(void);
void CalculateDC(void);
void ResetPowerModule(void);
```

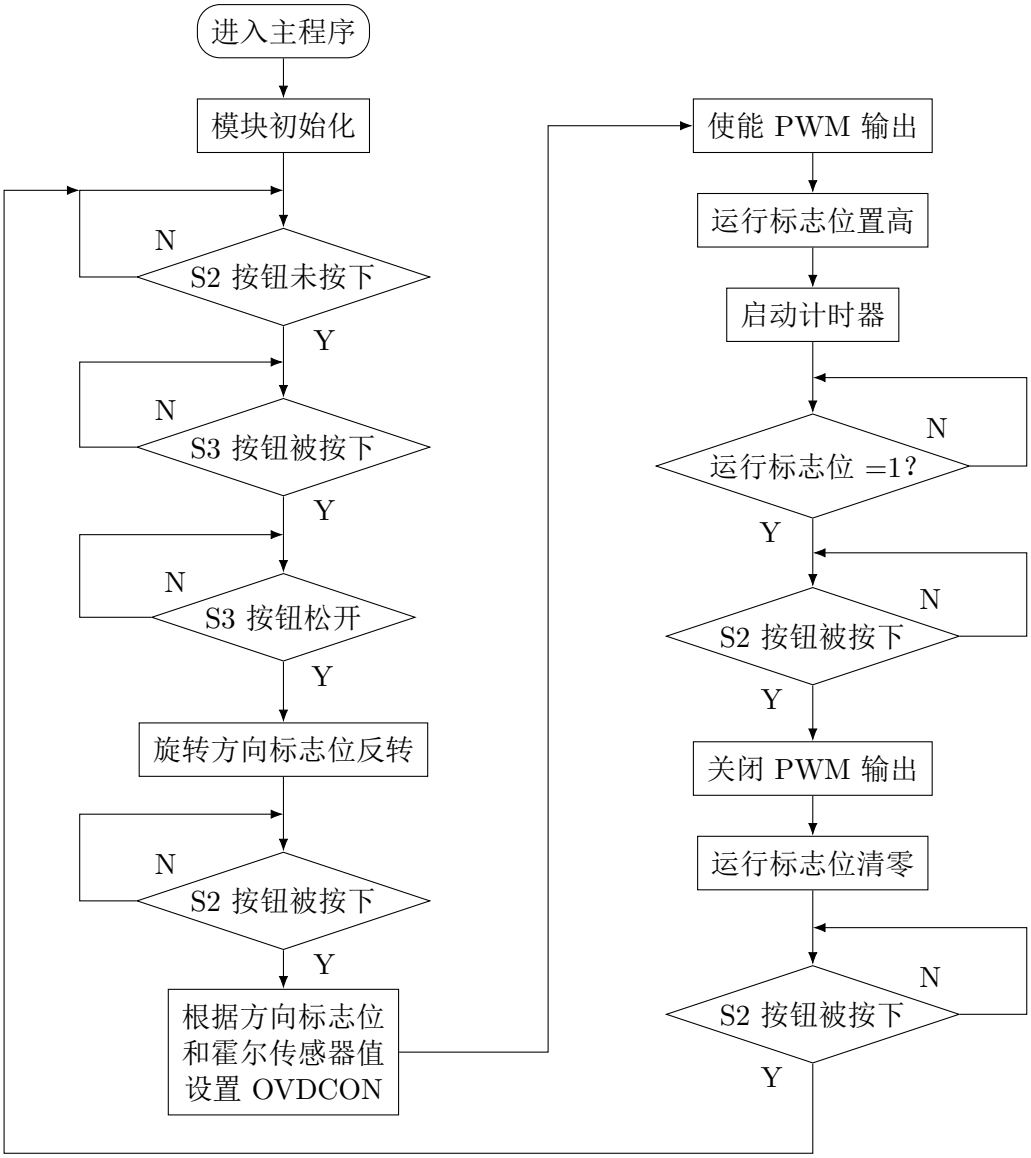

图 5: 实验一主程序流程图

```
void InitTMR1(void);
void lockIO(void);
void unlockIO(void);

struct MotorFlags Flags;

unsigned int HallValue;
unsigned int timer3value;
unsigned int timer3avg;
unsigned char polecount;

char *UartRPM,UartRPMarray[5];
int RPM, rpmBalance;

unsigned int StateTableFwd[] = {0x0000, 0x0210, 0x2004, 0x0204,
                                0x0801, 0x0810, 0x2001, 0x0000};
unsigned int StateTableRev[] = {0x0000, 0x2001, 0x0810, 0x0801,
                                0x0204, 0x2004, 0x0210, 0x0000};
```

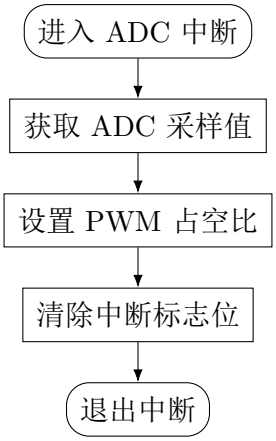

图 6: 实验一 ADC 中断框图

```
int main(void)
{
    unsigned int i;

    //初始化时钟
    PLLFBD =  8;
    CLKDIVbits.PLLPOST = 0;
    CLKDIVbits.PLLPRE = 0;
    __builtin_write_OSCCONH(0x03);
    __builtin_write_OSCCONL(0x01);

    //初始化锁相环
    while(OSCCONbits.COSC != 0b011);
    while(OSCCONbits.LOCK != 1);

    //设置按钮对应的输入端口
    TRISA |= 0x0100;
    TRISB |= 0x0010;
    //设置霍尔传感器对应的输入端口
    unlockIO();
    RPINR7bits.IC1R = 0x01;
    RPINR7bits.IC2R = 0x02;
    RPINR10bits.IC7R = 0x03;
    lockIO();

    //模块初始化
    InitADC10();
    InitTMR1();
    InitTMR3();
    timer3avg = 0;
    InitMCPWM();
    InitIC();

    //设置初始旋转方向
    Flags.Direction = 1;

    //延时等待各模块初始化完成
    for(i=0;i<1000;i++);

    //主程序循环
    while(1)
    {
        //等待S2(启停)按钮被按下
```

```
            while(S2)
            {
                //若S3(反向)按钮被按下
                if (!S3)
                {
                    //等待S3被松开
                    while (!S3)
                        DelayNmSec(10);
                    //切换旋转方向
                    Flags.Direction ^= 1;
                }
                Nop();
            }
            //等待S2按钮被松开
            while (!S2)
                DelayNmSec(10);

            //读取霍尔传感器值
            HallValue = (unsigned int)((PORTB >> 1) & 0x0007);
            //根据旋转方向为OVDCON寄存器写入对应扇区的值
            if (Flags.Direction)
                OVDCON = StateTableFwd[HallValue];
            else
                OVDCON = StateTableRev[HallValue];

            //使能PWM输出
            PWMCON1 = 0x0777;
            //电机运转标志位置1
            Flags.RunMotor = 1;
            //启动timer3计时
            T3CONbits.TON = 1;
            //初始化极数计数器
            polecount = 1;
            DelayNmSec(100);

            //当电机处于运转状态，循环等待
            while (Flags.RunMotor)
            {
                //若S2按钮被按下
                if (!S2)
                    {
                    //关闭PWM输出
                    PWMCON1 = 0x0700;
                    //OVDCON寄存器置0
                    OVDCON = 0x0000;
                    //电机运转标志位置0
                    Flags.RunMotor = 0;
                    //等待S2被松开
                    while (!S2)
                        DelayNmSec(10);
                    }
                Nop();
            }
        }
}

//延时函数
void DelayNmSec(unsigned int N)
{
unsigned int j;
while(N--)
    for(j=0;j < MILLISEC;j++);
}
```

```
void lockIO(){

asm volatile ("mov #OSCCON,w1 \n"
                "mov #0x46, w2 \n"
                "mov #0x57, w3 \n"
                "mov.b w2,[w1] \n"
                "mov.b w3,[w1] \n"
                "bset OSCCON, #6");
}

void unlockIO(){

asm volatile ("mov #OSCCON,w1 \n"
                "mov #0x46, w2 \n"
                "mov #0x57, w3 \n"
                "mov.b w2,[w1] \n"
                "mov.b w3,[w1] \n"
                "bclr OSCCON, #6");
}
```

Interrupts.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"

//ADC中断
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt (void)
{
    if (Flags.RunMotor)
    {
        //读取ADC采样得到的电位器分压值，作为占空比
        P1DC1 =  (ADC1BUF0 >> 1);
        P1DC2 = P1DC1;
        P1DC3 = P1DC1;
    }

    IFS0bits.AD1IF = 0;
}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC1Interrupt (void)
{

    IFS0bits.IC1IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
    {
        OVDCON = StateTableFwd[HallValue];
    }
    else
    {
        OVDCON = StateTableRev[HallValue];
    }

}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt (void)
{
```

```
    IFS0bits.IC2IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
        OVDCON = StateTableRev[HallValue];
}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC7Interrupt (void)
{
    IFS1bits.IC7IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
        OVDCON = StateTableRev[HallValue];
}
```

实验二 BLDC 闭环控制

1. 实验目标

a. 在实验一实现的 BLDC 启动、运行的基础上，增加转速的闭环控制。

b. 通过霍尔传感器的输出实现 BLDC 的转速测定。

c. 通过开发板上的电位器旋钮作为输入，实现对 BLDC 转速的精确给定。

2. 实验原理

2.1 转速闭环 PI 控制

根据自动控制原理课程的知识可知，为了使实际转速 n 与给定转速 n^* 相同，需要引入反馈环节，将两者比较后的结果作为控制系统的输入量，直到速度差 $\Delta n = 0$。

而为了实现无差调节，往往采用 PI 控制方式，控制系统的框图如图7所示。

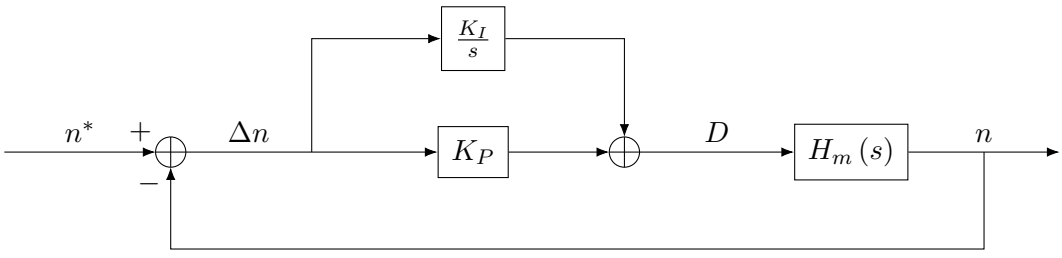

图 7: BLDC 闭环控制框图

图中，K_P 为 PI 环节的比例调节系数，K_I 为 PI 环节的积分调节系数，D 为占空比输出，$H_m(s)$ 为 H 桥及电机的传递函数。

为了达到稳定状态，要求控制系统输出的占空比应为一常数，又由于积分环节 K_I/s 的存在，只有到积分环节输入 $\Delta n = 0$ 时，积分输出才不再变化，因此，采用 PI 调节能够实现电机转速的无静差调节。而通过调节比例系数 K_P 和积分系数 K_I，能够调整控制系统的动态相应特性，如调整时间、最大超调量等。

电机速度测定

能够实现闭环 PI 调节的前提是能够准确测定电机实际转速，从而形成反馈。由于已经有安装在电机内的霍尔传感器能够获取电机转子的角度位置信息，只需要配合 dsPIC33FJ32MC204 处理器内部的定时器模块，测定转子磁场进入图3中某一扇区的时间间隔 Δt，在已知电机极对数的情况下，既可以计算得到电机实际转速：

$$n = \frac{60}{p\Delta t} \tag{1}$$

3. 实验过程

3.1 线路连接

本次实验线路连接与实验一（1.3.1）相同，这里不再赘述。

3.2 程序实现

转速测定

根据表2中霍尔传感器输入与转子磁场所在扇区关系，利用一路霍尔传感器 IO 输入跳变时产生的中断，在中断处理函数中读取定时器当前时间，并重新归零定时器，即可得到 Δt。

在实际程序中，采用 IC1 端口对应的中断处理函数进行转速的测定，根据 CAB 的霍尔传感器输入顺序，IC1 对应 B 相霍尔传感器输入，因此当电机逆时针旋转时，选择当霍尔传感器输入为 011 时作为计算转子转过 360° 电角度的起点；而当电机顺时针旋转时，选择当霍尔传感器输入为 001 时作为计算转子转过 360° 电角度的起点。当每次进入中断时判断霍尔传感器输入值，若对应于当前电角度起点，则表示转子转过电角度一周，当转过电角度周数等于电机极对数时，读取出定时器数值，得到的即为 $p\Delta t$。

IC1 中断处理函数的程序流程框图如图8所示。

闭环控制

利用 IC1 中断中得到的时间间隔值，计算出电机的实际转速，并与给定转速进行比较，将转速插值送入比例环节和积分环节得到占空比的控制输出量。由于处理器内的数字类型具有最大和最小值，因此还需要加入限幅判断环节，防止由于积分环节的累积效应，导致控制输出超过类型范围，导致程序崩溃。

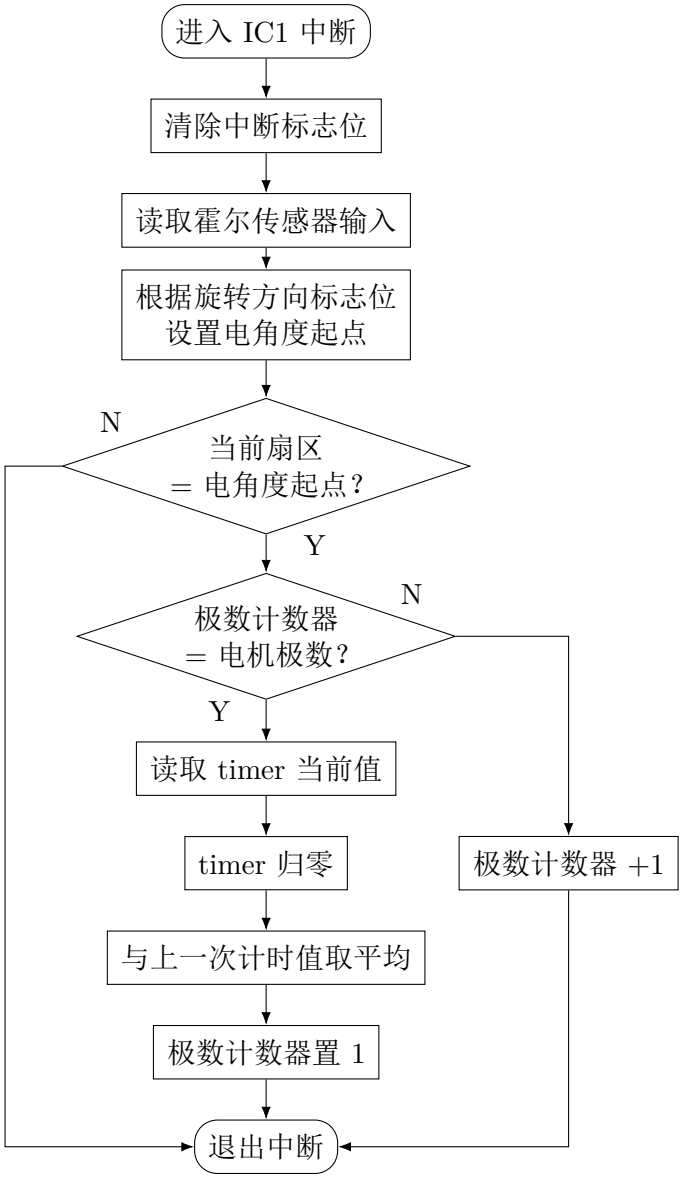

图 8: 实验二 IC1 中断程序流程框图

将闭环控制放在另一个定时器 Timer1 的中断内进行，每次进入 T1 中断，就进行依此 PI 环节的计算，并得到新的占空比输出值。Timer1 中断的程序流程框图如图9所示。

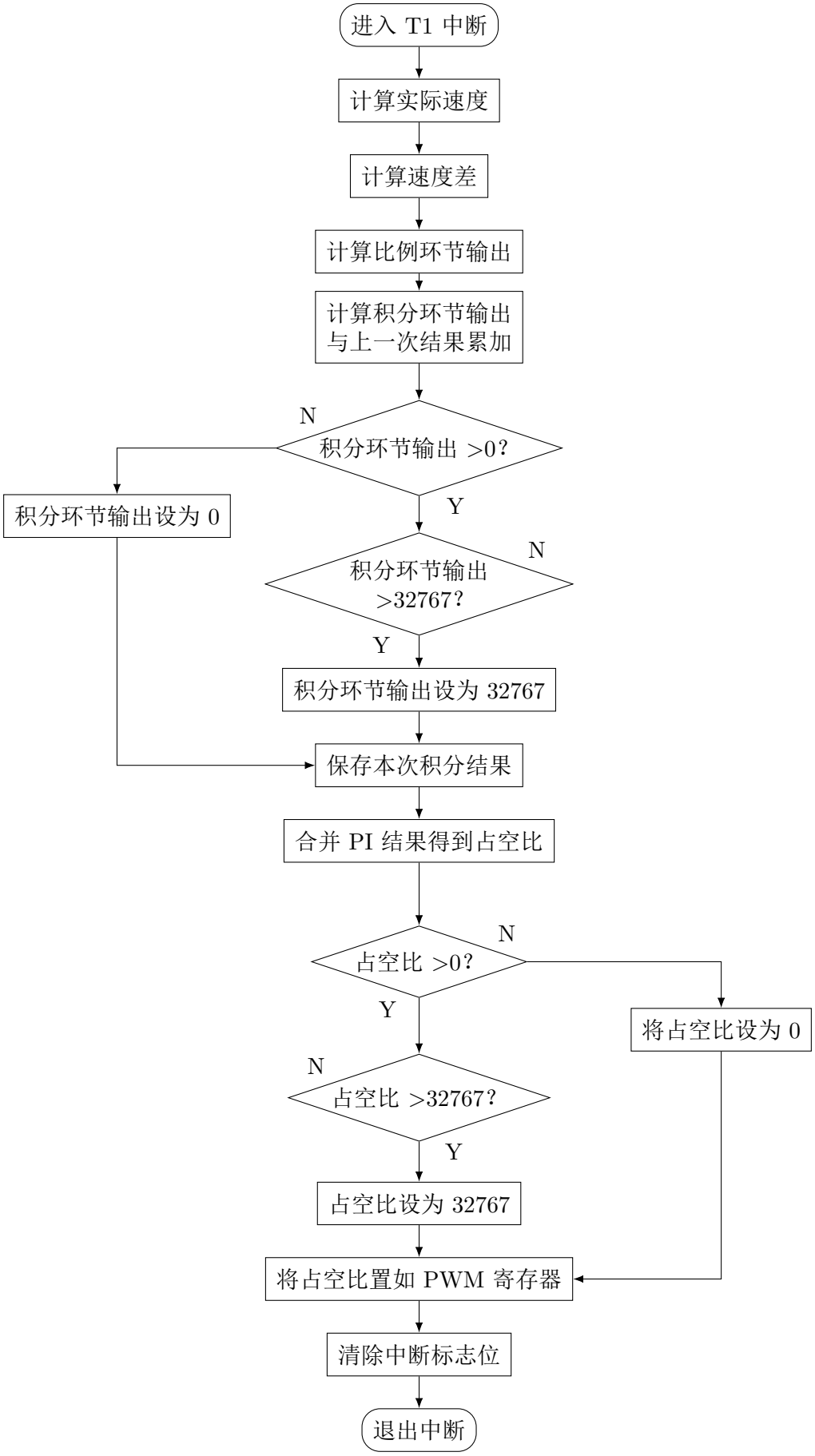

图 9: 实验二 Timer1 中断程序流程图

转速设定

本实验中，通过开发板上的电位器旋钮外设作为用户输入源，通过 ADC 采样得到电位器的分压值，等比例地换算为转速的设定值，在每次 ADC 中断中进行 ADC 采样结果与转速设定值的更新，ADC 中断处理程序的流程图如10所示。

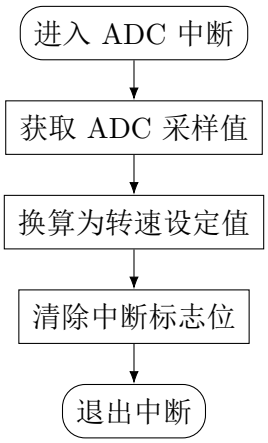

图 10: 实验二 ADC 中断框图

3.3 代码清单及注释

SensoredBLDC.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"

_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & POSCMD_XT);

_FBS (BSS_NO_FLASH & BWRP_WRPROTECT_OFF);

_FWDT (FWDTEN_OFF);

_FGS (GSS_OFF & GCP_OFF & GWRP_OFF);

_FPOR (PWMPIN_ON & HPOL_ON & LPOL_ON & FPWRT_PWR128);

_FICD (ICS_PGD3 & JTAGEN_OFF);

void InitADC10(void);
void DelayNmSec(unsigned int N);
void InitMCPWM(void);
void InitTMR3(void);
void InitIC(void);
void CalculateDC(void);
void ResetPowerModule(void);
void InitTMR1(void);
void lockIO(void);
void unlockIO(void);

struct MotorFlags Flags;

unsigned int HallValue;
```

```
unsigned int timer3value;
unsigned int timer3avg;
unsigned char polecount;

char *UartRPM,UartRPMarray[5];
int RPM, rpmBalance;

unsigned int StateTableFwd[] = {0x0000, 0x0210, 0x2004, 0x0204,
                                0x0801, 0x0810, 0x2001, 0x0000};
unsigned int StateTableRev[] = {0x0000, 0x2001, 0x0810, 0x0801,
                                0x0204, 0x2004, 0x0210, 0x0000};



int main(void)
{
    unsigned int i;

    //初始化时钟
    PLLFBD =  8;
    CLKDIVbits.PLLPOST = 0;
    CLKDIVbits.PLLPRE = 0;
    __builtin_write_OSCCONH(0x03);
    __builtin_write_OSCCONL(0x01);

    //初始化锁相环
    while(OSCCONbits.COSC != 0b011);
    while(OSCCONbits.LOCK != 1);

    //设置按钮对应的输入端口
    TRISA |= 0x0100;
    TRISB |= 0x0010;
    //设置霍尔传感器对应的输入端口
    unlockIO();
    RPINR7bits.IC1R = 0x01;
    RPINR7bits.IC2R = 0x02;
    RPINR10bits.IC7R = 0x03;
    lockIO();

    //模块初始化
    InitADC10();
    InitTMR1();
    InitTMR3();
    timer3avg = 0;
    InitMCPWM();
    InitIC();

    //设置初始旋转方向
    Flags.Direction = 1;

    //延时等待各模块初始化完成
    for(i=0;i<1000;i++);

    //主程序循环
    while(1)
    {
        //等待S2(启停)按钮被按下
        while(S2)
        {
            //若S3(反向)按钮被按下
            if (!S3)
            {
                //等待S3被松开
```

```
                while (!S3)
                    DelayNmSec(10);
                //切换旋转方向
                Flags.Direction ^= 1;
            }
            Nop();
        }
        //等待S2按钮被松开
        while (!S2)
            DelayNmSec(10);

        //读取霍尔传感器值
        HallValue = (unsigned int)((PORTB >> 1) & 0x0007);
        //根据旋转方向为OVDCON寄存器写入对应扇区的值
        if (Flags.Direction)
            OVDCON = StateTableFwd[HallValue];
        else
            OVDCON = StateTableRev[HallValue];

        //使能PWM输出
        PWMCON1 = 0x0777;
        //电机运转标志位置1
        Flags.RunMotor = 1;
        //启动timer3计时
        T3CONbits.TON = 1;
        //初始化极数计数器
        polecount = 1;
        DelayNmSec(100);

        //当电机处于运转状态，循环等待
        while (Flags.RunMotor)
        {
            //若S2按钮被按下
            if (!S2)
                {
                //关闭PWM输出
                PWMCON1 = 0x0700;
                //OVDCON寄存器置0
                OVDCON = 0x0000;
                //电机运转标志位置0
                Flags.RunMotor = 0;
                //等待S2被松开
                while (!S2)
                    DelayNmSec(10);
                }
            Nop();
        }
    }
}

//延时函数
void DelayNmSec(unsigned int N)
{
unsigned int j;
while(N--)
    for(j=0;j < MILLISEC;j++);
}


void lockIO(){

asm volatile ("mov #OSCCON,w1 \n"
              "mov #0x46, w2 \n"
```

```
                    "mov #0x57, w3 \n"
                    "mov.b w2,[w1] \n"
                    "mov.b w3,[w1] \n"
                    "bset OSCCON, #6");
}

void unlockIO(){

asm volatile ("mov #OSCCON,w1 \n"
                    "mov #0x46, w2 \n"
                    "mov #0x57, w3 \n"
                    "mov.b w2,[w1] \n"
                    "mov.b w3,[w1] \n"
                    "bclr OSCCON, #6");
}
```

Interrupts.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"
int DesiredSpeed;    //给定速度
int ActualSpeed;     //实际速度
int SpeedError;      //速度误差
//误差积分
long SpeedIntegral = 0, SpeedIntegral_n_1 = 0, SpeedProportional = 0;
long DutyCycle = 0; //占空比
unsigned int Kps = 20000;    //比例系数
unsigned int Kis = 2000;     //积分系数

//ADC中断
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt (void)
{
    if (Flags.RunMotor)
        //读取ADC采样值，作为给定速度
        DesiredSpeed = ADC1BUF0 * POTMULT;

    IFS0bits.AD1IF = 0;
}

//IO端口中断
void __attribute__((interrupt, no_auto_psv)) _IC1Interrupt (void)
{
    int Hall_Index;

    IFS0bits.IC1IF = 0;
    //读取霍尔传感器输入
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
    {
        //根据霍尔传感器输入与旋转方向，设置OVDCON寄存器值
        OVDCON = StateTableFwd[HallValue];
        //设置电角度起始扇区位置
        Hall_Index = HALL_INDEX_F;
    }
    else
    {
        OVDCON = StateTableRev[HallValue];
        Hall_Index = HALL_INDEX_R;
    }

    //电角度旋转一周
```

```
    if (HallValue == Hall_Index)
        //判断是否完成机械角度一周
        if (polecount++ == POLEPAIRS)
        {
            //读取timer3计时器读数
            timer3value = TMR3;
            //清零timer3计时器读数
            TMR3 = 0;
            //计时器读书求平均，减小随机误差
            timer3avg = ((timer3avg + timer3value) >> 1);
            //重置极数计数器
            polecount = 1;
        }
}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt (void)
{
    IFS0bits.IC2IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
        OVDCON = StateTableRev[HallValue];
}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC7Interrupt (void)
{
    IFS1bits.IC7IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
        OVDCON = StateTableRev[HallValue];
}

//定时器Timer1中断
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt (void)
{
    //根据时间间隔计算出实际转速
    ActualSpeed = SPEEDMULT/timer3avg;
    //计算转速误差
    SpeedError = DesiredSpeed - ActualSpeed;
    //计算比例环节输出
    SpeedProportional = (int)(((long)Kps*(long)SpeedError) >> 15);
    //计算积分环节输出
    SpeedIntegral = SpeedIntegral_n_1 + (int)(((long)Kis*(long)
        SpeedError) >> 15);

    //积分环节输出做限幅处理
    if (SpeedIntegral < 0)
        SpeedIntegral = 0;
    else if (SpeedIntegral > 32767)
        SpeedIntegral = 32767;
    //将本次积分环节设置为历史值，供下一次继续使用
    SpeedIntegral_n_1 = SpeedIntegral;
    //计算出占空比输出
    DutyCycle = SpeedIntegral + SpeedProportional;
    //占空比输出做限幅处理
    if (DutyCycle < 0)
```

```
        DutyCycle = 0;
    else if (DutyCycle > 32767)
        DutyCycle = 32767;

    //将占空比输出结果写入PWM模块
    PDC1 = (int)(((long)(PTPER*2)*(long)DutyCycle) >> 15);
    PDC2 = PDC1;
    PDC3 = PDC1;

    //清除中断标志位
    IFS0bits.T1IF = 0;
}
```

实验三　跑马灯实验

1. 实验目标

a. 利用开发板上的 LED 外设，实现跑马灯。

b. 进一步熟悉 PWM 模块的控制与使用。

2. 实验原理

在进行前两个实验时可以发现，当电机转速较低时，可以看到开发板上的 LED 跳变闪动，结合开发板电路原理图11，可知开发板上的六枚 LED 采用共阴极接地，阳极分别与六路 PWM 输出相接，因此当 PWM 输出为高电平时，既可以点亮对应的 LED。

结合之前进行电机旋转控制时 PWM 模块的控制策略，通过 OVDCON 寄存器可以方便地选通需要的 PWM 输出，而使其余 PWM 输出无效，因此在本实验中也可以采用相同的控制方式，对应于跑马灯的六种状态，建立 OVDCON 寄存器值表，经过一定延时依次将表中数值放入 OVDCON 中，即可完成跑马灯控制。

3. 实验过程

3.1 线路连接

本实验中 PWM 输出作为跑马灯的控制，因此不再需要接入电机三相电源线和霍尔传感器输出线，只需要为开发板接入直流电源即可。

3.2 程序实现

令 LED 按照 D10、D11 D15 的顺序依次点亮，此时六种状态与 OVDCON 寄存器值的关系如表3所示。

在主程序循环中，每经过一段延时时间，就依次改变 OVDCON 表的值，并将新的值放入 OVDCON 寄存器中，由此即可实现跑马灯，程序流程图如图12所

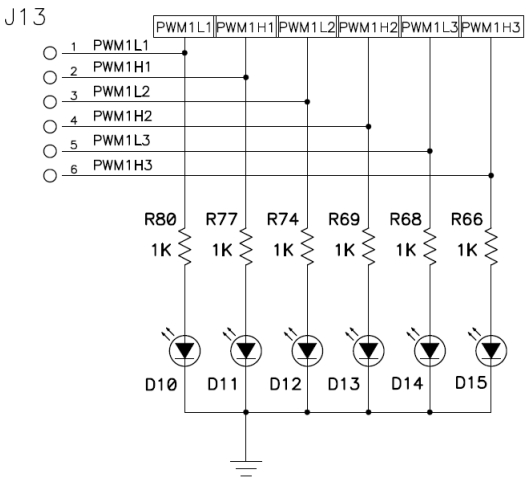

图 11: 开发板 LED 部分原理图

表 3: LED 状态与 OVDCON 寄存器的对应关系

点亮的 LED	PWM1	PWM2	PWM3	OVDCON 值
D10	L	0	0	0x0100
D11	H	0	H	0x0200
D12	0	L	0	0x0400
D13	0	H	0	0x0800
D14	0	0	L	0x1000
D15	0	0	H	0x2000

示。

3.3 代码清单及注释

SensoredBLDC.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"

_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & POSCMD_XT);

_FBS (BSS_NO_FLASH & BWRP_WRPROTECT_OFF);

_FWDT (FWDTEN_OFF);

_FGS (GSS_OFF & GCP_OFF & GWRP_OFF);

_FPOR (PWMPIN_ON & HPOL_ON & LPOL_ON & FPWRT_PWR128);

_FICD (ICS_PGD3 & JTAGEN_OFF);

void InitADC10(void);
void DelayNmSec(unsigned int N);
void InitMCPWM(void);
```

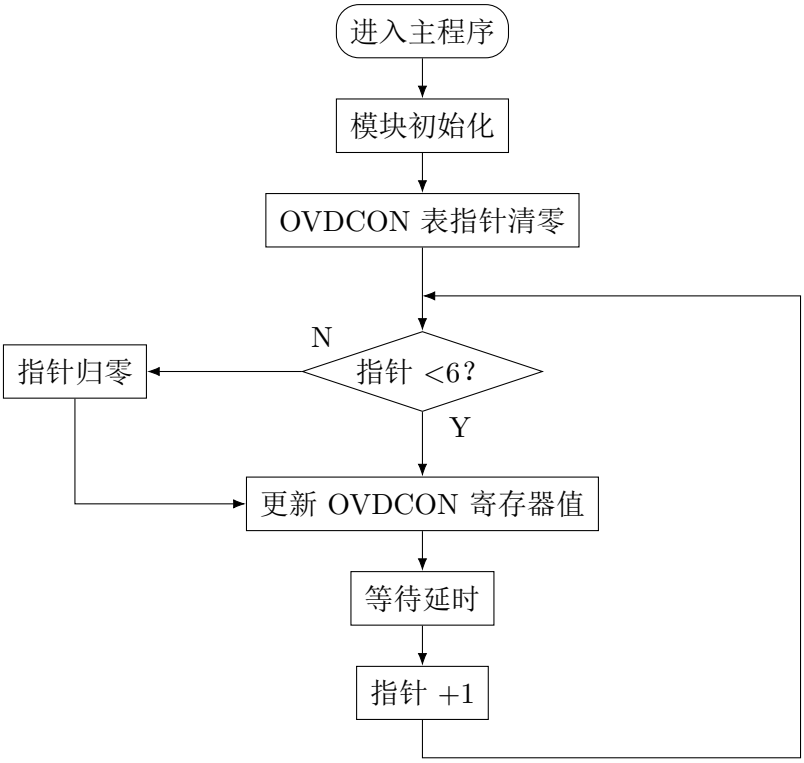

图 12: 实验三主程序流程图

```
void InitTMR3(void);
void InitIC(void);
void CalculateDC(void);
void ResetPowerModule(void);
void InitTMR1(void);
void lockIO(void);
void unlockIO(void);

struct MotorFlags Flags;

unsigned int index = 0;

unsigned int StateTableFwd[] = {0x0100, 0x0200, 0x0400, 0x0800,
                                0x1000, 0x2000};



int main(void)
{
    unsigned int i;

    //初始化时钟
    PLLFBD =  8;
    CLKDIVbits.PLLPOST = 0;
    CLKDIVbits.PLLPRE = 0;
    __builtin_write_OSCCONH(0x03);
    __builtin_write_OSCCONL(0x01);

    //初始化锁相环
    while(OSCCONbits.COSC != 0b011);
    while(OSCCONbits.LOCK != 1);

    //模块初始化
```

```
    InitMCPWM();
    InitIC();

    //延时等待各模块初始化完成
    for(i=0;i<1000;i++);

    //使能PWM输出
    PWMCON1 = 0x0777;

    //主程序循环
    while(1)
    {

        if (++index == 6)
            index = 0;
        OVDCON = StateTableFwd[index];

        DelayNmSec(200);
    }
}

//延时函数
void DelayNmSec(unsigned int N)
{
unsigned int j;
while(N--)
    for(j=0;j < MILLISEC;j++);
}


void lockIO(){

asm volatile ("mov #OSCCON,w1 \n"
              "mov #0x46, w2 \n"
              "mov #0x57, w3 \n"
              "mov.b w2,[w1] \n"
              "mov.b w3,[w1] \n"
              "bset OSCCON, #6");
}

void unlockIO(){

asm volatile ("mov #OSCCON,w1 \n"
              "mov #0x46, w2 \n"
              "mov #0x57, w3 \n"
              "mov.b w2,[w1] \n"
              "mov.b w3,[w1] \n"
              "bclr OSCCON, #6");
}
```

实验四 UART 通信

1. 实验目标

a. 利用 UART 模块实现上位机（PC）与开发板之间的通讯。

b. 实现通过上位机向处理器发送调速指令完成调速的功能。

c. 实现处理器向上位机定时发送当前速度功能，观察开闭环调速特性。

2. 实验原理

2.1 BLDC 调速系统的动态特性

无刷直流电机的动态特性

对无刷直流电机，可以将其定子绕组等效为绕组电阻、绕组电感与反电动势的串联，如图13所示。由此可得定子绕组的微分方程：

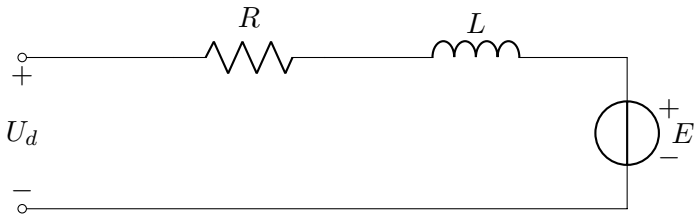

图 13: 定子绕组等效电路

$$U_d = RI_d + L\frac{\mathrm{d}I_d}{\mathrm{d}t} + E \tag{2}$$

式中，I_d 为定子绕组电流，E 为电枢反应产生的反电动势，在电机的结构确定后，与转速成正比：

$$E = C_e n \tag{3}$$

而对与转子，可以列写出电磁转矩 T_e、负载转矩 T_L 和转轴飞轮矩 GD^2 组成的动力学方程：

$$T_e - T_L = \frac{GD^2}{375}\frac{\mathrm{d}n}{\mathrm{d}t} \tag{4}$$

而电磁转矩又于电枢电流成正比：

$$T_e = C_m I_d \tag{5}$$

设 T_l 为定子绕组的时间常数，T_m 为传动系统的动力学时间常数：

$$T_l = \frac{L}{R} \tag{6}$$

$$T_m = \frac{GD^2 R}{375 C_e C_m} \tag{7}$$

将上述方程改写为复频域形式，整理后得到无刷直流电机的动态特性框图如图14所示：进一步化简框图，打开闭环并将负载电流移到输入端得到如图15所

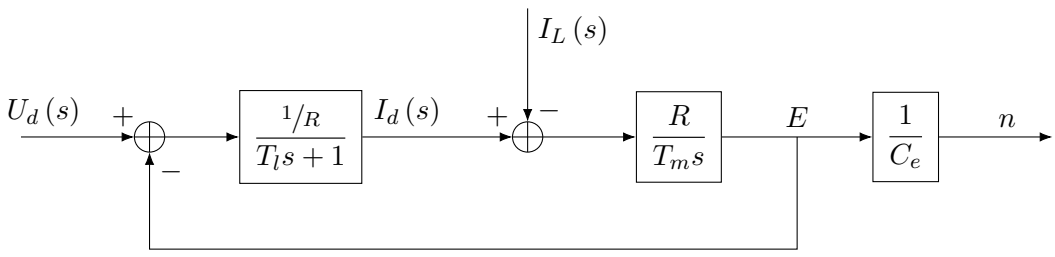

图 14: 无刷直流电机动态特性框图

示。可已看出，当无刷直流电机处于理想空载（$I_L = 0$）时，无刷直流电机的转速与输入电压构成一个二阶系统。

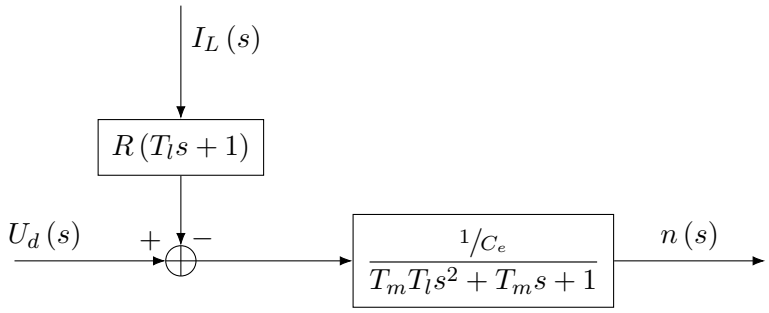

图 15: 简化的无刷直流电机动态特性框图

开环控制系统调速特性

对于数字 DSP 处理器和 MOSFET 构成的调速系统而言，处理器的采样频率和 MOSFET 的开关频率往往在 10 kHz 以上，控制系统的时间常数相比于具有较大感性与惯性的电机而言非常小，可以忽略不计，其速度给定 n^* 与电机电压 U_d 间即为一简单的比例环节。因此对于开环控制系统的动态框图即如图15所示。

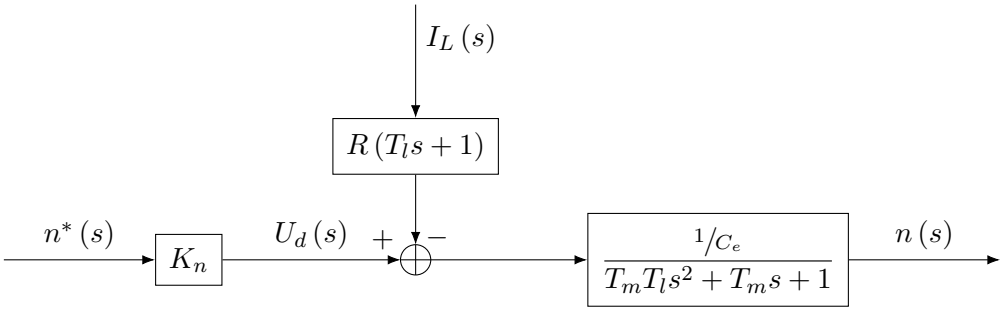

图 16: 开环调速系统动态特性框图

可以看到，在开环调节下，负载电流直接作为干扰量进入系统中，并且无法得到抑制，因此将造成实际的转速将随着负载电流的变化而波动，绝大部分情况下都将与给定转速存在差值，并且由于电机具有一定的惯性，时间常数较大，电机的动态响应速度也较慢。

闭环控制系统调速特性

闭环调速系统中引入了反馈机制，并通过 PI 调节实现转速的无静差控制，当控制系统两次采样、计算间隔远小于电机的时间常数时，可以认为其为连续系

统，PI 环节的传递函数可以写为：

$$d(s) = \left(K_P + \frac{K_I}{s}\right)\Delta n(s) \tag{8}$$

$$U_d(s) = K_s d(s) \tag{9}$$

式中，d 为控制器输出的占空比，K_s 为 H 桥将占空比转化为输出平均电压的比例系数。由此可以得到闭环系统的动态特性框图如图17所示。

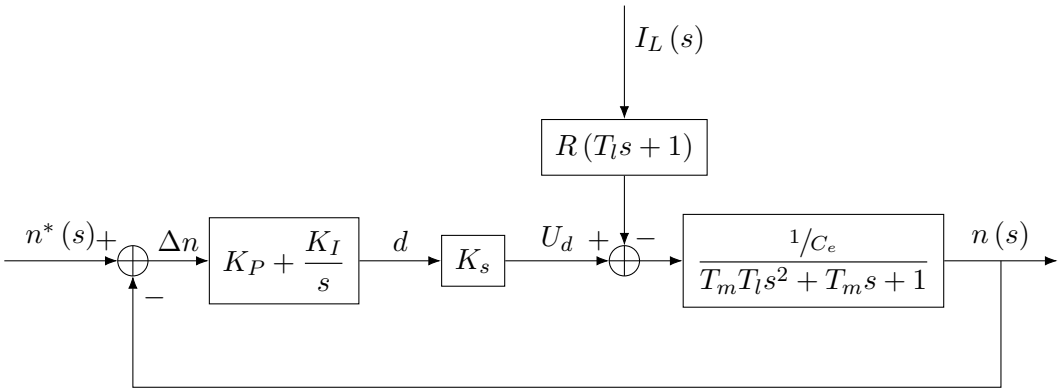

图 17: 闭环调速系统动态特性框图

当忽略负载电流（即认为理想空载）时，系统的闭环传递函数可以写为：

$$G(s) = \frac{K_P K_s s + K_I K_s}{C_e T_m T_l s^3 + C_e T_m s^2 + (C_e + K_P K_s)s + K_I K_s} \tag{10}$$

而对于负载电流 $I_L(s)$ 而言，其闭环传递函数为：

$$N(s) = \frac{RT_l s^2 + Rs}{-C_e T_m T_l s^3 - C_e T_m s^2 + (K_P K_s - C_e)s + K_I K_s} \tag{11}$$

则根据终值定理，干扰项产生的输出最终将趋于零，即被闭环系统抑制，在稳态时达到实际转速等于给定转速的效果。

2.2 自定义通讯缓存区

dsPIC33FJ32MC204 的 UART 模块内置了 4 个字节深度的发送队列 TX_FIFO 和接收队列 RX_FIFO，并能够设置 TX 中断和 RX 中断的出发条件与对应队列当前占用字节数的关系，但是由于队列长度短，造成在实际使用过程中的灵活度不够。例如在发送长度超过四位的字符串时，整个程序必须在发送完四个字节后进入阻塞状态，直到判断到 TX_FIFO 中有新的空位后再继续发送，导致处理器的效率降低；发过来在接受字符过程中，则当需要接收的指令长度超过四个字节时也需要让程序停留在接收字符的进程中，一旦有字符进入 FIFO 就将其及时取出，以免 FIFO 溢出，导致接收的错误。

因此，在程序中重新开辟了两块空间作为 TX 和 RX 的缓存区，并编写配

套的字符与字符串操作函数，实现对字符发送与接收任务的完全接管，直接与对应 TX 中断与 RX 中断进行交互。其中，RX 缓存区采用堆栈的数据格式，当 RX_FIFO 接收到字符产生 RX 中断时，自动将 FIFO 中的内容压入堆栈，当发现指令结束字符时，再将堆栈中内容弹出，进行对指令的判断和后续操作。而 TX 缓存区采用循环队列结构，当 TX 缓存区首尾指针相等，即队列为空时，TX 中断处于关闭状态，当有内容需要发送时，将发送内容一次性全部写入缓存区，此时，TX 中断将被打开，并自动将缓存区内内容依次放入 FIFO 中，直到缓存区再次为空，TX 中断关闭，而其他程序不要阻塞等待发送过程，提高了处理器效率，此外，当一次写入的字符串长度过大，超过缓存区长度时，将自动发出预警信息，提示开发者适当增加缓存区长度。

3. 实验过程

3.1 线路连接

本实验中的线路连接与实验一与实验二1.3.1的线路连接相同，此外，使用 UART 功能与上位机进行通讯时，还需要确保开发板上的两个跳帽安装在了正确的位置。根据开发板线路原理图18所示，将跳帽安装在 UART 对应位置。

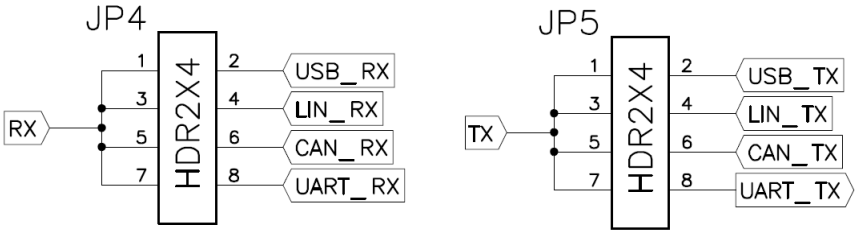

图 18: 开发板跳帽连接部分原理图

程序实现

通过 dsPIC33FJ32MC204 的 UART 模块，来实现开发板向上位机发送电机转速，上位机向开发板发送控制指令的功能，该部分主要的逻辑过程在 UART 的 RX 中断处理函数中完成：进入中断后，开始从 RX_FIFO 中读取字符，并对读取到的字符进行判断是否为换行字符（即一条指令的结束字符），当判断到指令结束时，从 RX 缓存区中取出之前存储的整条指令，按照正则表达式对指令中的指令名称和参数进行提取，并完成对应的速度返回、速度设定和恢复到 ADC 电位器调速模式等操作。此外，还有输入回显、退格键删除输入等交互功能。程序流程框图如19所示。

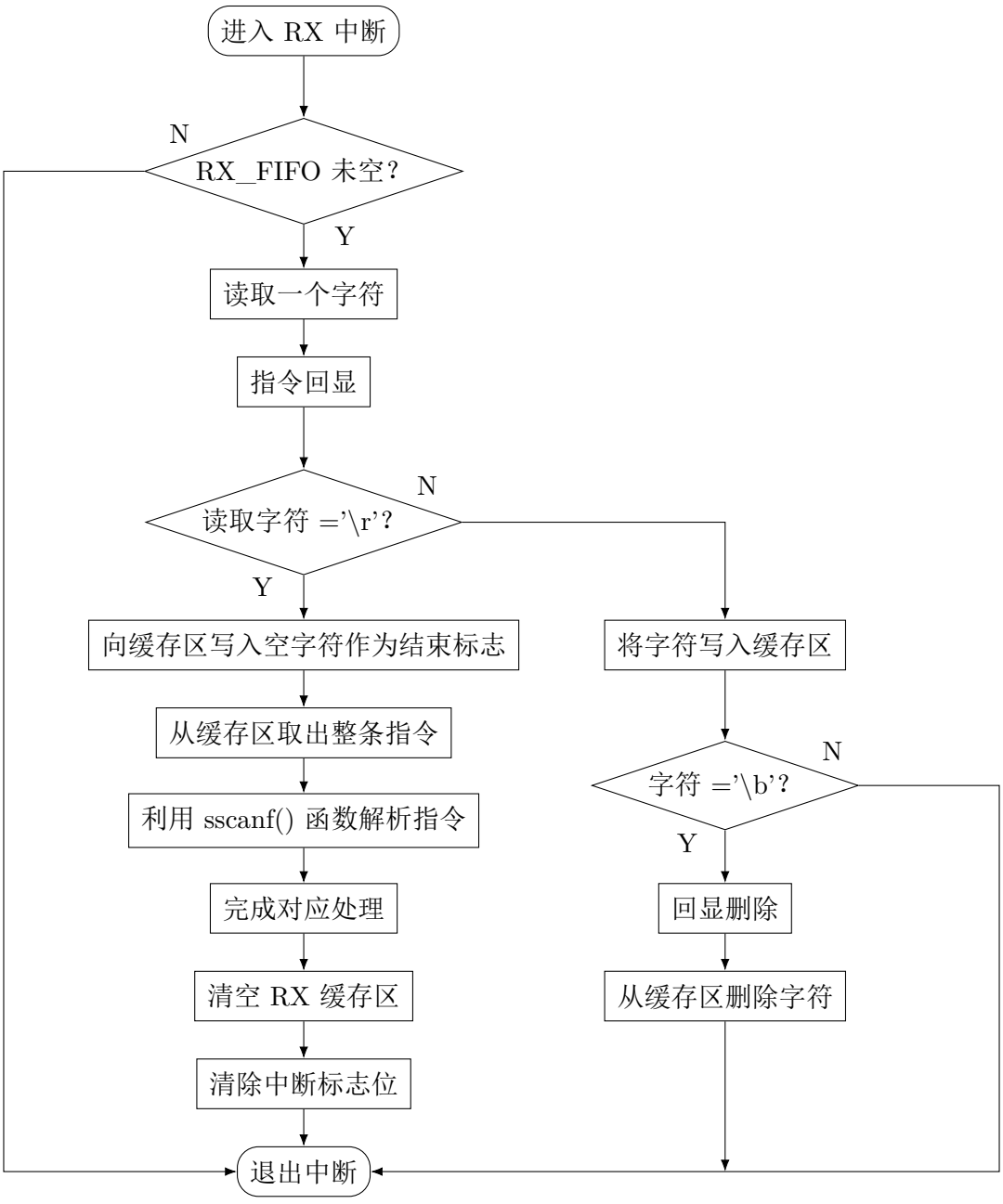

图 19: 实验四 RX 中断程序流程框图

3.2 代码清单及注释

SensoredBLDC.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"

_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & POSCMD_XT);

_FBS (BSS_NO_FLASH & BWRP_WRPROTECT_OFF);

_FWDT (FWDTEN_OFF);

_FGS (GSS_OFF & GCP_OFF & GWRP_OFF);

_FPOR (PWMPIN_ON & HPOL_ON & LPOL_ON & FPWRT_PWR128);

_FICD (ICS_PGD3 & JTAGEN_OFF);

void InitADC10(void);
void DelayNmSec(unsigned int N);
void InitMCPWM(void);
void InitTMR3(void);
void InitIC(void);
void CalculateDC(void);
void ResetPowerModule(void);
void InitTMR1(void);
void lockIO(void);
void unlockIO(void);

struct MotorFlags Flags;

unsigned int HallValue;
unsigned int timer3value;
unsigned int timer3avg;
unsigned char polecount;

char *UartRPM,UartRPMarray[5];
int RPM, rpmBalance;

unsigned int StateTableFwd[] = {0x0000, 0x0210, 0x2004, 0x0204,
                                0x0801, 0x0810, 0x2001, 0x0000};
unsigned int StateTableRev[] = {0x0000, 0x2001, 0x0810, 0x0801,
                                0x0204, 0x2004, 0x0210, 0x0000};



void __attribute__ ((interrupt, no_auto_psv)) _U1RXInterrupt(void) {


    int data = 0;

    //RX_FIFO未空
    if(U1STAbits.URXDA != 0)
    {
        //从RX_FIFO取出一个字符
        data = U1RXREG;

        //输入回显
        TX_Buffer_WriteChar(mySciBuffer, data);

        //输入为回车，一条指令结束
```

```
if ( data == '\r' )
{
    //回传换行符，完成换行回显
    TX_Buffer_WriteChar(mySciBuffer, '\n');
    //向RX_BUFFER写入空字符，标志指令结束
    RX_Buffer_WriteChar(mySciBuffer, 0);

    //RX_BUFFER未空
    while (mySciBuffer->RX_Buffer[i])
    {
        //取出字符
        CMDLINE[i] = mySciBuffer->RX_Buffer[i];
        i++;
    }
    //从取出的指令中提取指令名称与参数
    key = sscanf(CMDLINE,"%s %d",cmdline, &speed_set);
    //指令为SPEED
    if (!strcmp(cmdline,"SPEED"))
    {
        //不带参数，为返回速度指令
        if (key == 1)
        {
            Flags.RunMotor = 1;
            sprintf(ANSWER,"The speed is %d rpm.\r\n",
                ActualSpeed);
            TX_Buffer_WriteString(mySciBuffer, ANSWER);
        }
        //带参数，为调速指令
        if (key == 2)
        {
            Flags.RunMotor = 1;
            //设置速度为0，令电机停转
            if (speed_set == 0)
            {
                Flags.RunMotor = 0;
            }
            report_flag = 1;
            //设置当前调速模式，令ADC中断失效
            set_mode = 1;
        }
    }
    //指令为RESERVE
    if (!strcmp(cmdline,"RESERVE"))
    {
        //不带参数，以当前给定速度反转
        if (key == 1)
        {
            Flags.RunMotor = 1;
            Flags.Direction ^= 1;
            sprintf(ANSWER,"The motor is reserved.\r\n");
            TX_Buffer_WriteString(mySciBuffer, ANSWER);
        }
        //带速度参数，以新的给定速度反转
        if (key == 2)
        {
            Flags.RunMotor = 1;
            if (speed_set == 0)
            {
                Flags.RunMotor = 0;
            }
            report_flag = 1;
            set_mode = 1;
        }
```

```
            }
            //指令为ADC，回复ADC采样值调速模式
            if (!strcmp(cmdline,"ADC"))
            {
                Flags.RunMotor = 1;
                set_mode = 0;
            }

            //清除RX_BUFFER
            RX_Buffer_Flush(mySciBuffer);
        }
        else
        {
            //向RX_BUFFER中写入收到的字符
            RX_Buffer_WriteChar(mySciBuffer, data);
        }
        //收到退格键
        if ( data == '\b' )
        {
            //先回退光标，输入空格覆盖需要删除的字符，再回退到原位置
            TX_Buffer_WriteChars(mySciBuffer, " \b", 2);
            //从RX_BUFFER中删除字符和退格键
            RX_Buffer_DeleteLastNChars(mySciBuffer, 2);
        }

    }

    U1STAbits.OERR = 0;
    IFS0bits.U1RXIF = 0;
}
void __attribute__ ((interrupt, no_auto_psv)) _U1TXInterrupt(void) {

    //从TX_BUFFER中向TX_FIFO放入字符
    TX_Buffer_Export2TX_FIFO(mySciBuffer, 0);

    //TX_BUFFER为空，则关闭TX中断
    if ( TX_Buffer_isEmpty(mySciBuffer) )
    {
        IEC0bits.U1TXIE = 0;
    }

    IFS0bits.U1TXIF = 0;
}

int main(void)
{
    unsigned int i;

    //初始化时钟
    PLLFBD =  8;
    CLKDIVbits.PLLPOST = 0;
    CLKDIVbits.PLLPRE = 0;
    __builtin_write_OSCCONH(0x03);
    __builtin_write_OSCCONL(0x01);

    //初始化锁相环
    while(OSCCONbits.COSC != 0b011);
    while(OSCCONbits.LOCK != 1);

    //设置按钮对应的输入端口
    TRISA |= 0x0100;
    TRISB |= 0x0010;
    //设置霍尔传感器对应的输入端口
```

```
unlockIO();
RPINR7bits.IC1R = 0x01;
RPINR7bits.IC2R = 0x02;
RPINR10bits.IC7R = 0x03;
lockIO();

//模块初始化
InitADC10();
InitTMR1();
InitTMR3();
timer3avg = 0;
InitMCPWM();
InitIC();

//设置初始旋转方向
Flags.Direction = 1;

//延时等待各模块初始化完成
for(i=0;i<1000;i++);

//主程序循环
while(1)
{
    //等待S2(启停)按钮被按下
    while(S2)
    {
        //若S3(反向)按钮被按下
        if (!S3)
        {
            //等待S3被松开
            while (!S3)
                DelayNmSec(10);
            //切换旋转方向
            Flags.Direction ^= 1;
        }
        Nop();
    }
    //等待S2按钮被松开
    while (!S2)
        DelayNmSec(10);

    //读取霍尔传感器值
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);
    //根据旋转方向为OVDCON寄存器写入对应扇区的值
    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
        OVDCON = StateTableRev[HallValue];

    //使能PWM输出
    PWMCON1 = 0x0777;
    //电机运转标志位置1
    Flags.RunMotor = 1;
    //启动timer3计时
    T3CONbits.TON = 1;
    //初始化极数计数器
    polecount = 1;
    DelayNmSec(100);

    //当电机处于运转状态，循环等待
    while (Flags.RunMotor)
    {
        //若S2按钮被按下
```

```
                if (!S2)
                    {
                    //关闭PWM输出
                    PWMCON1 = 0x0700;
                    //OVDCON寄存器置0
                    OVDCON = 0x0000;
                    //电机运转标志位置0
                    Flags.RunMotor = 0;
                    //等待S2被松开
                    while (!S2)
                        DelayNmSec(10);
                    }
                Nop();
            }
        }
}

//延时函数
void DelayNmSec(unsigned int N)
{
unsigned int j;
while(N--)
    for(j=0;j < MILLISEC;j++);
}


void lockIO(){

asm volatile ("mov #OSCCON,w1 \n"
                "mov #0x46, w2 \n"
                "mov #0x57, w3 \n"
                "mov.b w2,[w1] \n"
                "mov.b w3,[w1] \n"
                "bset OSCCON, #6");
}

void unlockIO(){

asm volatile ("mov #OSCCON,w1 \n"
                "mov #0x46, w2 \n"
                "mov #0x57, w3 \n"
                "mov.b w2,[w1] \n"
                "mov.b w3,[w1] \n"
                "bclr OSCCON, #6");
}
```

Interrupts.c

```
#include "p33FJ32MC204.h"
#include "SensoredBLDC.h"
extern int speed_set, set_mode;
int DesiredSpeed;    //给定速度
int ActualSpeed;     //实际速度
int SpeedError;      //速度误差
//误差积分
long SpeedIntegral = 0, SpeedIntegral_n_1 = 0, SpeedProportional = 0;
long DutyCycle = 0; //占空比
unsigned int Kps = 20000;    //比例系数
unsigned int Kis = 2000;     //积分系数

//ADC中断
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt (void)
```

```
{
    if (Flags.RunMotor)
    {
        if (set_mode == 0)
        {
            DesiredSpeed = ADC1BUF0 * POTMULT;
        }
        if (set_mode == 1)
        {
            DesiredSpeed = speed_set;
        }
    }

    IFS0bits.AD1IF = 0;
}

//IO端口中断
void __attribute__((interrupt, no_auto_psv)) _IC1Interrupt (void)
{
    int Hall_Index;

    IFS0bits.IC1IF = 0;
    //读取霍尔传感器输入
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
    {
        //根据霍尔传感器输入与旋转方向，设置OVDCON寄存器值
        OVDCON = StateTableFwd[HallValue];
        //设置电角度起始扇区位置
        Hall_Index = HALL_INDEX_F;
    }
    else
    {
        OVDCON = StateTableRev[HallValue];
        Hall_Index = HALL_INDEX_R;
    }

    //电角度旋转一周
    if (HallValue == Hall_Index)
        //判断是否完成机械角度一周
        if (polecount++ == POLEPAIRS)
        {
            //读取timer3计时器读数
            timer3value = TMR3;
            //清零timer3计时器读数
            TMR3 = 0;
            //计时器读书求平均，减小随机误差
            timer3avg = ((timer3avg + timer3value) >> 1);
            //重置极数计数器
            polecount = 1;
        }
}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt (void)
{
    IFS0bits.IC2IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
```

```
        OVDCON = StateTableRev[HallValue];
}

//IO端口中断，当霍尔传感器输入变化时改变OVDCON寄存器的值
void __attribute__((interrupt, no_auto_psv)) _IC7Interrupt (void)
{
    IFS1bits.IC7IF = 0;
    HallValue = (unsigned int)((PORTB >> 1) & 0x0007);

    if (Flags.Direction)
        OVDCON = StateTableFwd[HallValue];
    else
        OVDCON = StateTableRev[HallValue];
}

//定时器Timer1中断
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt (void)
{
    //根据时间间隔计算出实际转速
    ActualSpeed = SPEEDMULT/timer3avg;
    //计算转速误差
    SpeedError = DesiredSpeed - ActualSpeed;
    //计算比例环节输出
    SpeedProportional = (int)(((long)Kps*(long)SpeedError) >> 15);
    //计算积分环节输出
    SpeedIntegral = SpeedIntegral_n_1 + (int)(((long)Kis*(long)
        SpeedError) >> 15);

    //积分环节输出做限幅处理
    if (SpeedIntegral < 0)
        SpeedIntegral = 0;
    else if (SpeedIntegral > 32767)
        SpeedIntegral = 32767;
    //将本次积分环节设置为历史值，供下一次继续使用
    SpeedIntegral_n_1 = SpeedIntegral;
    //计算出占空比输出
    DutyCycle = SpeedIntegral + SpeedProportional;
    //占空比输出做限幅处理
    if (DutyCycle < 0)
        DutyCycle = 0;
    else if (DutyCycle > 32767)
        DutyCycle = 32767;

    //将占空比输出结果写入PWM模块
    PDC1 = (int)(((long)(PTPER*2)*(long)DutyCycle) >> 15);
    PDC2 = PDC1;
    PDC3 = PDC1;

    //清除中断标志位
    IFS0bits.T1IF = 0;
}
```

sci_buffer.h

```
/*
 * SCI_BUFFER以结构体形式定义了SCI_BUFFER对象，结构体中开辟了RX_BUFFER和
    TX_BUFFER空间，以及相应的数据结构指针
 *
 * 还定义了配套的函数，能够实现RX_BUFFER的写入、删除、读取、清零等功能，
 * 和TX_BUFFER的写入（字符、多个字符、字符串）、删除、判断是否为空、输出
    TX_BUFFER中内容到TX_FIFO等功能
 *
```

```
 * 代替UART模块中的RX_FIFO和TX_FIFO，实现更为简洁的程序设计和更高的运行
     效率，并能实现更为复杂的信息交互功能
 *
 */

#ifndef __SCI_BUFFER_H_
#define __SCI_BUFFER_H_

#include "string.h"
#include "p33FJ32MC204.h"

//定义RX_BUFFER和TX_BUFFER长度
#define RX_BUFFER_LEN 128
#define TX_BUFFER_LEN 128

//定义SCI_BUFFER结构体对象
//依此包含：
//RX_BUFFER
//TX_BUFFER
//RX_BUFFER(尾)指针
//TX_BUFFER头指针
//TX_BUFFER尾指针        从而TX_BUFFER能够实现循环队列的结构
typedef struct _SCI_Buffer_Obj_
{
    volatile int RX_Buffer[RX_BUFFER_LEN];
    volatile int TX_Buffer[TX_BUFFER_LEN];
    volatile int RX_Buffer_Index;
    volatile int TX_Buffer_Head_Index;
    volatile int TX_Buffer_Tail_Index;
}SCI_Buffer_Obj;

//定义SCI_BUFFER结构体指针
typedef struct _SCI_Buffer_Obj_ *SCI_Buffer_Handle;

//初始化SCI_BUFFER结构体指针程序
SCI_Buffer_Handle SCI_Buffer_init(void *pMemory);

//RX_BUFFER写入单个字符
int RX_Buffer_WriteChar(SCI_Buffer_Handle sci_buffer_handle, int RX_Char
    );
//RX_BUFFER删除末尾单个字符
int RX_Buffer_DeleteLastChar(SCI_Buffer_Handle sci_buffer_handle);
//RX_BUFFER删除末尾n个字符
int RX_Buffer_DeleteLastNChars(SCI_Buffer_Handle sci_buffer_handle, int
    n);
//RX_BUFFER清空
void RX_Buffer_Flush(SCI_Buffer_Handle sci_buffer_handle);
//RX_BUFFER读取整行数据
char* RX_Buffer_PickLine(SCI_Buffer_Handle sci_buffer_handle);

//TX_BUFFER写入单个字符
int TX_Buffer_WriteChar(SCI_Buffer_Handle sci_buffer_handle, int TX_Char
    );
//TX_BUFFER写入N个字符
int TX_Buffer_WriteChars(SCI_Buffer_Handle sci_buffer_handle, char *
    TX_msg, int len);
//TX_BUFFER写入字符串
int TX_Buffer_WriteString(SCI_Buffer_Handle sci_buffer_handle, const
    char *TX_msg);
//TX_BUFFER溢出预检查
int TX_Buffer_OverflowPreCheck(SCI_Buffer_Handle sci_buffer_handle, int
    len);
//将TX_BUFFER中的字符导出到TX_FIFO中
```

```
int TX_Buffer_Export2TX_FIFO(SCI_Buffer_Handle sci_buffer_handle, int
    TX_FIFO_Level);
//判断TX_BUFFER是否为空
int TX_Buffer_isEmpty(SCI_Buffer_Handle sci_buffer_handle);

#endif /* SCI_BUFFER_H_ */
```

sci_buffer.c

```
#include "sci_buffer.h"
#include "p33FJ32MC204.h"


//初始化SCI_BUFFER对象
//*pMemory : 主函数中定义的SCI_BUFFER对象的内存地址
SCI_Buffer_Handle SCI_Buffer_init(void *pMemory)
{
    //创建SCI_BUFFER结构体指针指向该内存空间并返回
    SCI_Buffer_Handle sci_buffer_handle;
    sci_buffer_handle = (SCI_Buffer_Handle)pMemory;

    //初始化BUFFER数组指针
    sci_buffer_handle->RX_Buffer_Index = 0;
    sci_buffer_handle->TX_Buffer_Head_Index = 0;
    sci_buffer_handle->TX_Buffer_Tail_Index = 0;

    return(sci_buffer_handle);
}

//向RX_BUFFER中写入单个字符
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//RX_Char : 要向RX_BUFFER中写入的字符
int RX_Buffer_WriteChar(SCI_Buffer_Handle sci_buffer_handle, int RX_Char
    )
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    //将字符写入RX_BUFFER
    sci_buffer->RX_Buffer[sci_buffer->RX_Buffer_Index] = RX_Char;
    //RX_BUFFER指针递增
    sci_buffer->RX_Buffer_Index ++;
    //判断RX_BUFFER是否溢出
    if ( sci_buffer->RX_Buffer_Index == RX_BUFFER_LEN )
    {
        //若溢出，则发送警告信息
        TX_Buffer_WriteString(sci_buffer, "WARNING! RX_Buffer Just
            Overflowed! A Larger RX_Buffer is Required!");
        return(-1);
    }
    return(0);
}

//删除RX_BUFFER末尾的单个字符
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
int RX_Buffer_DeleteLastChar(SCI_Buffer_Handle sci_buffer_handle)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    //将RX_BUFFER指针前移，即删除了末尾字符
    if ( sci_buffer->RX_Buffer_Index != 0 )
    {
        sci_buffer->RX_Buffer_Index --;
```

```
    }

    return(0);
}

//删除RX_BUFFER末尾的n字符
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//n : 删除的字符数
int RX_Buffer_DeleteLastNChars(SCI_Buffer_Handle sci_buffer_handle, int
    n)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    int i;

    //循环n次，将RX_BUFFER递减n次
    for (i=0; i < n; i++)
    {
        if ( sci_buffer->RX_Buffer_Index != 0 )
        {
            sci_buffer->RX_Buffer_Index --;
        }
        else
        {
            break;
        }
    }

    return(0);
}

//清空RX_BUFFER
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
void RX_Buffer_Flush(SCI_Buffer_Handle sci_buffer_handle)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    //将RX_BUFFER指针指向0
    sci_buffer->RX_Buffer_Index = 0;
}

//读取RX_BUFFER中的全部数据
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
char* RX_Buffer_PickLine(SCI_Buffer_Handle sci_buffer_handle)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    return((char *)sci_buffer->RX_Buffer);
}

//向TX_BUFFER中写入单个字符
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//TX_Char : 要写入TX_BUFFER的字符
int TX_Buffer_WriteChar(SCI_Buffer_Handle sci_buffer_handle, int TX_Char
    )
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    //先进行TX_BUFFER溢出预检查
    TX_Buffer_OverflowPreCheck(sci_buffer, 1);
    //预检查完成，写入该字符
    sci_buffer->TX_Buffer[sci_buffer->TX_Buffer_Tail_Index %
        TX_BUFFER_LEN] = TX_Char;
```

```
    //递增TX_BUFFER尾指针
    sci_buffer->TX_Buffer_Tail_Index ++;
    //使能TX_FIFO中断，开始通过SCI_TX发送字符
    if (U1STAbits.TRMT == 1)
    {
        TX_Buffer_Export2TX_FIFO(sci_buffer, 0);
    }
    IEC0bits.U1TXIE = 1;
    return(0);
}

//向TX_BUFFER中写入n个字符
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//TX_msg : 要写入TX_BUFFER的字符串
//len : 写入字符的个数
int TX_Buffer_WriteChars(SCI_Buffer_Handle sci_buffer_handle, char *
    TX_msg, int len)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    int i;

    //先进行TX_BUFFER溢出预检查
    TX_Buffer_OverflowPreCheck(sci_buffer, len);
    //循环len次，写入len个字符，并依此递增TX_BUFFER尾指针
    for (i=0; i<len; i++)
    {
        sci_buffer->TX_Buffer[sci_buffer->TX_Buffer_Tail_Index %
            TX_BUFFER_LEN] = TX_msg[i];
        sci_buffer->TX_Buffer_Tail_Index ++;
    }
    if (U1STAbits.TRMT == 1)
    {
        TX_Buffer_Export2TX_FIFO(sci_buffer, 0);
    }
    //使能TX_FIFO中断，开始通过SCI_TX发送字符
    IEC0bits.U1TXIE = 1;
    return(0);
}

//向TX_BUFFER中写入n个字符
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//TX_msg : 要写入TX_BUFFER的字符串
int TX_Buffer_WriteString(SCI_Buffer_Handle sci_buffer_handle, const
    char *TX_msg)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    const char *pChar;
    pChar = TX_msg;
    //先进行TX_BUFFER溢出预检测
    if ( TX_Buffer_OverflowPreCheck(sci_buffer, strlen(TX_msg)) == -2 )
    {
        //长度过长，返回错误信息
        return(-1);
    }
    //依此写入整个字符串到TX_BUFFER
    while ( *pChar )
    {
        sci_buffer->TX_Buffer[sci_buffer->TX_Buffer_Tail_Index %
            TX_BUFFER_LEN] = *pChar;
        sci_buffer->TX_Buffer_Tail_Index ++;
        pChar ++;
```

```
    }
    if (U1STAbits.TRMT == 1)
    {
        TX_Buffer_Export2TX_FIFO(sci_buffer, 0);
    }
    //使能TX_FIFO中断，开始通过SCI_TX发送字符
    IEC0bits.U1TXIE = 1;
    return(0);
}

//TX_BUFFER溢出预检测
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//len : 写入字符的个数
int TX_Buffer_OverflowPreCheck(SCI_Buffer_Handle sci_buffer_handle, int
    len)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    //如果写入字符的长度已经大于整个TX_BUFFER的大小，则直接输出错误信息
        并返回错误
    if ( len > TX_BUFFER_LEN )
    {
        TX_Buffer_WriteString(sci_buffer, "Too Long Text! A Larger
            TX_BUFFER is Required!\r\n");
        return(-2);
    }

    //如果当前TX_BUFFER剩余大小不能容纳需要写入的字符串，则为了输出数据
        完整，暂时以阻塞式等待TX_BUFFER中有足够的空间
    //并发出错误信息，提醒使用者扩大TX_BUFFER大小，以免程序总是运行在阻
        塞模式，降低效率
    if ( ( sci_buffer->TX_Buffer_Tail_Index + len ) >= ( sci_buffer->
        TX_Buffer_Head_Index + TX_BUFFER_LEN ) )
    {
        while ( ( sci_buffer->TX_Buffer_Tail_Index + 52 ) == (
            sci_buffer->TX_Buffer_Head_Index + TX_BUFFER_LEN ) ) {}
        TX_Buffer_WriteString(sci_buffer, "A Block Just Occurred! A
            Larger TX_BUFFER is Required!\r\n");
        while ( ( sci_buffer->TX_Buffer_Tail_Index + len ) == (
            sci_buffer->TX_Buffer_Head_Index + TX_BUFFER_LEN ) ) {}
        return(-1);
    }
    return(0);
}

//将TX_BUFFER中的内容写入TX_FIFO
//sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
//TX_FIFO_Level : 设定的TX_FIFO的中断深度，对应一次可以从TX_BUFFER写入
    TX_FIFO中的字符数量
int TX_Buffer_Export2TX_FIFO(SCI_Buffer_Handle sci_buffer_handle, int
    TX_FIFO_Level)
{
    SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

    //计算可以本次可以写入TX_FIFO的字符数量
    int i = 4 - TX_FIFO_Level;
    //依此写入i个字符到TX_FIFO中，并更新TX_BUFFER头指针
    while ( ( --i >= 0 ) && ( sci_buffer->TX_Buffer_Head_Index <
        sci_buffer->TX_Buffer_Tail_Index) )
    {
       U1TXREG = sci_buffer->TX_Buffer[sci_buffer->TX_Buffer_Head_Index
           ];
```

```
            //循环队列的已进行了一次循环，则头尾指针同时减去TX_BUFER长度
            if ( ++(sci_buffer->TX_Buffer_Head_Index) >= TX_BUFFER_LEN )
            {
                sci_buffer->TX_Buffer_Head_Index = sci_buffer->
                    TX_Buffer_Head_Index - TX_BUFFER_LEN;
                sci_buffer->TX_Buffer_Tail_Index = sci_buffer->
                    TX_Buffer_Tail_Index - TX_BUFFER_LEN;
            }
        }

        return(0);
    }

    //判断TX_BUFFER是否为空
    //sci_buffer_handle : 指向SCI_BUFFER结构体对象的指针
    int TX_Buffer_isEmpty(SCI_Buffer_Handle sci_buffer_handle)
    {
        SCI_Buffer_Obj *sci_buffer = (SCI_Buffer_Obj *)sci_buffer_handle;

        //若头指针等于尾指针，则TX_BUFFER为空
        if ( sci_buffer->TX_Buffer_Head_Index == sci_buffer->
            TX_Buffer_Tail_Index )
        {
            return(1);
        }
        return(0);
    }
```

心得与体会

通过本次课程的学习，对于无刷直流电机的运行原理，控制电路和控制方法有了较为完整的理解与掌握。通过四个循序渐进的实验内容，逐步完成无刷直流电机的启动运行到闭环调速再到通讯，对无刷直流高效率、结构简单、调速性能优秀的特点有了更为直观的认识与体会，也在结合 Datasheet、User Guide 的学习、摸索中锻炼了自己解决问题的能力。在此，要感谢同组同学在课程过程中的相互合作、共同进步。也要感谢助教每次课都耐心地为我们准备实验器材，在我们遇到问题时，即使已经过了下课时间，也依然认真帮我们分析问题，寻找解决途径。也要感谢朱莉老师为我们细致地分析讲解无刷直流电机的原理与控制方法，为我们提供了这一个学习与动手实践的平台，并从中获益匪浅。