西安科技大学
XI'AN UNIVERSITY OF SCIENCE AND TECHNOLOGY

虚拟仪器（LabVIEW）

课程设计报告

指导教师： 訾宏洋　彭倩

院　（系）： 电气与控制工程学院

专业班级： 测控技术与仪器 1702 班

姓　　名： 徐华金

学　　号： 1706070221

报告日期： 2021 年 1 月 6 日

基于 LabVIEW 的直流有刷电机 PID 调试及控制助手

摘　要

PID 控制是最早发展起来的控制策略之一，由于其算法简单、鲁棒性好和可靠性高，被广泛应用于过程控制和运动控制中。传统单片机 PID 控制在 PID 参数整定等方面有诸多不便，如 PID 参数调节不便、控制效果不易判断等缺点。本设计基于 LabVIEW 设计一个直流有刷电机 PID 调试及控制助手，实现 PID 参数调节及电机转速及位置的控制。

系统下位机采用 STM32F103 作为直流有刷电机控制的主控芯片，采用位置控制增量式 PID 算法及速度控制增量式 PID 算法分别实现了对直流有刷电机的位置及速度的控制。上位机运用 LabVIEW 的 VISA 模块实现与下位机的串口通信，上位机可实现对直流有刷电机的位置及速度 PID 控制参数的调节、目标值波形及实际波形的显示和直流有刷电机位置及速度的控制。

该系统对 PID 参数的调节方便、控制效果更直观；人机界面美观、人机交互性强、系统稳定。

关键字：LabVIEW，PID，直流有刷电机控制，串口通信

西安科技大学电气与控制工程学院
测控技术与仪器专业
《虚拟仪器设计》课程设计任务书

学生姓名：徐华金　　学号：17406070221

一、 设计题目

基于 LabVIEW 的直流有刷电机 PID 调试及控制助手

二、设计目的和要求

通过对虚拟仪器的设计，了解虚拟仪器设计的基本原理及常用的对象使用方法；通过设计基于虚拟仪器的数据采集系统、图形识别系统、通过串口、并口控制、测温系统、转速测量系统等，了解虚拟仪器数据采集的法实现方法，并口或串口的数据传输原理，明确虚拟编程中如何驱动非 NI 公司的数据采集卡或电脑中常用的声卡，运用知识将外界物理信号采集到计算机并进行简单分析或者通过虚拟仪器产生符合要求的信号并通过声卡输出或者学会利用 NI 的 ELVIS 系统设计简单的测量软件。加深虚拟仪器知识、单片机的了解，培养学生运用虚拟仪器思想解决工程实际问题的能力。

二、 设计内容

A．详细要求：

基于 LabVIEW 的直流有刷电机 PID 调试及控制助手设计：使用 LabVIEW 开发环境和 VISA 串口、单片机电机控制系统，实现（1）上位机对下位机 PID 控制器 PID 参数的调整（2）下位机系统被控量波形的显示（3）上位机对下位机电机的转速和位置的控制。

B．前面板上要设置必要的控件对象以设置相应参数。

（1） PID 调试界面：实现对 PID 参数的调整及曲线的显示。

（2） 电机控制界面：实现对电机速度及位置的控制。

C．设计界面要美观，程序可读性好。

三、 设计进度安排

设计时间总计 1 周

课程设计任务及要求讲解（0.5 天）

设计任务分析及查找资料（0.5 天）

程序编制及调试　　（2.5 天）

设计说明书撰写　　（1 天）

验收与答辩　　（0.5 天）

四、 设计报告应包括的主要内容

目录、设计任务分析、实现过程、下位机硬件图及流程图、上位机程序前面板及流程图、结束语、心得体会、参考文献

五、 考核方法与成绩评定

考核方式由三部分组成：平时学习态度（含考勤）、设计完成情况（含方案、程序质量、界面、设计报告等）及答辩情况确定。

成绩由三部分构成，平时学习态度（10%），设计完成情况（40%），设计报告（30%），答辩情况（20%）。

六、 补充说明

无

指导教师：咎宏洋、彭倩（虚拟仪器课程设计）

黄梦涛、彭倩（组态软件课程设计）

2021 年 1 月 2 日

目　录

1 绪论

1.1 研究的目的与意义

直流有刷电机（Brushed DC motor）具有结构简单、易于控制、成本低等特点，在一些功能简单的应用场合，或者说在能够满足必要的性能、低成本和足够的可靠性的前提下，直流有刷电机往往是一个很好的选择。例如便宜的电子玩具、各种风扇和汽车的电动座椅等。基本的直流有刷电机在电源和电机之间只需要两根电缆，这样就可以节省配线和连接器所需的空间，并降低电缆和连接器的成本。此外，还可以使用 MOSFET/IGBT 开关对直流有刷电机进行控制，给电机提供足够好的性能的同时，整个电机控制系统也会比较便宜。

目前,直流调速控制系统大多采用在额定电压范围内改变电枢电压的方法,使得电机的额定转速下降。通过改变电动机的电枢电压接通时间和通电周期的比值,这就是常见的脉冲宽度调制技术(PWM 技术)。但单一的 PWM 控制并不能使电机的转速达到良好的控制效果，因此需要加入 PID 控制算法使电机转速控制更加精准。

直流有刷电机位置控制模式一般是通过编码器产生的脉冲的个数来确定转动的角度或者是转的圈数，由于位置模式可以位置进行严格的控制，所以一般应用于定位装置。应用领域如数控机床、印刷机械等等。

以上两种电机控制都需要 PID 算法的控制才可以达到较好的控制效果。但传统单片机 PID 控制系统 PID 参数的整定一直是个不太方便的作用。一方面 PID 参数不好调节，另一方面调节 PID 后的控制效果也不好与之前的效果比较，缺乏有效的数据支撑。基于以上两点开发一款基于 LabVIEW 的直流有刷电机 PID 调试及控制助手软件，方便电机的 PID 参数调试及控制。

1.2 PID 控制算法介绍

PID 算法是控制领域非常常见的算法，小到控制温度，大到控制飞机的飞行姿态和速度等等，都会涉及到 PID 控制，在控制领域可以算是万能的算法。

通过图 1-1 不难看出，PID 控制其实就是对偏差的控制过程；如果偏差为 0,则比例环节不起作用，只有存在偏差时，比例环节才起作用； 积分环节主要是用来消除静差，所谓静差，就是系统稳定后输出值和设定值之间的差值，积分环节实际上就是偏差累计的过程， 把累计的误差加到原有系统上以抵消系统造成的静差；而微分信号则反应了偏差信号的变化规律，也可以说是变化趋势，根据偏差信号的变化趋势来进行超前调节，从而增加了系统的预知性。

PID 控制器是一种线性控制器，他根据给定 $y_d(t)$与实际输出 y(t)构成控制偏差

$$error(t) = y_d(t) - y(t) \tag{1-1}$$

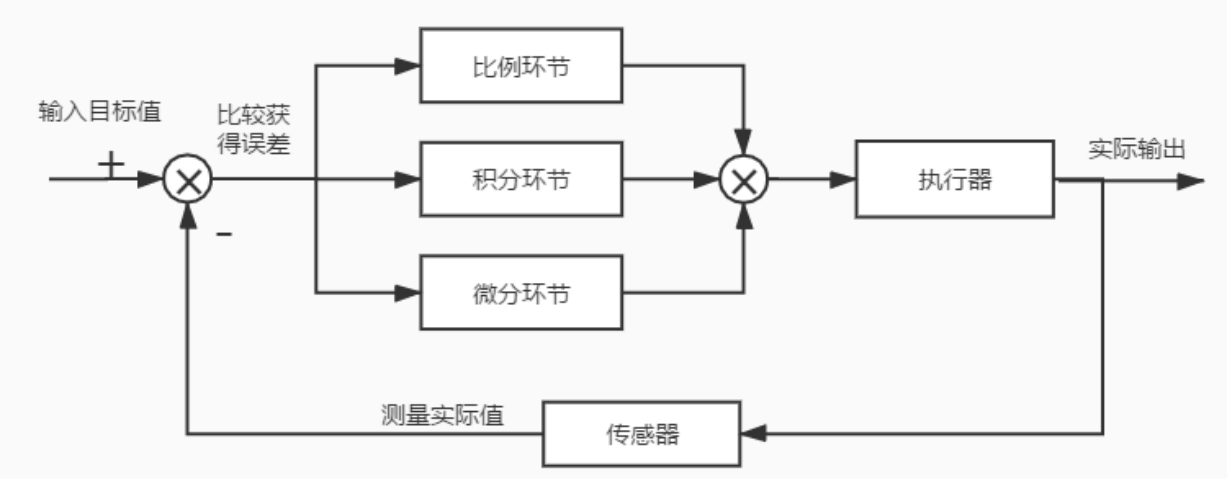


图 1-1 PID 框图

PID 控制规律为

$$u(t) = k_p[error(t) + \frac{1}{T_I}\int_0^t error(t)dt + \frac{T_D derror(t)}{dt}] \quad (1\text{-}2)$$

对上述 PID 系统进行离散化，离散化后方便在程序上进行数字处理。

1) 假设采集数据的间隔时间为 T，则在 kT 时刻有：

2) 误差等于第 k 个周期时刻的误差等于输入（目标）值减输出（实际）值，则有：

$$error(k) = errint(k) - errout(k) \quad (1\text{-}3)$$

3) 积分环节为所有时刻的误差和，则有：

$$error(k) + error(k+1) + error(k+2) + \cdots \quad (1\text{-}4)$$

4) 微分环节为第 k 时刻误差的变化率，则有：

$$\frac{error(k) - error(k-1)}{T} \quad (1\text{-}5)$$

综上获得如下 PID 离散形式：

$$u(k) = K_p\left[error(k) + \frac{T}{T_i}\sum error(j) + \frac{T_D}{T}\big(error(k) - error(k-1)\big)\right] \quad (1\text{-}6)$$

位置式 PID 可表示为：

$$u(k) = K_p error(k) + K_i\sum error(k) + K_d\big(error(k) - error(k-1)\big) \quad (1\text{-}7)$$

进一步推导出增量式 PID 表达式：

$$\Delta u(k) = K_p(error(k) - error(k-1) + K_i error(k) + K_d(error(k) - 2error(k-1) + error(k-2))) \quad (1\text{-}8)$$

增量式 PID 优缺点：

- 优点：
  1) 误动作时影响小，必要时可用逻辑判断的方法去掉出错数据。
  2) 手动、自动切换时冲击小，便于实现无扰动切换。
  3) 算式中不需要累加。控制增量 Δ u(k)的确定仅与最近 3 次的采样值有关。在速度闭环控制中有很好的实时性。
- 缺点：
  1) 积分截断效应大，有稳态误差；

2) 溢出的影响大。有的被控对象用增量式则不太好

1.3 直流有刷电机调速原理

要实现对直流电机的闭环控制，必须先了解其调速原理，直流电机的转速关系如下：

$$n = \frac{U-IR}{k_e\varphi} \tag{1-9}$$

式（1-9）中 I——电枢电流 A；　　k_e——电枢结构决定的电动势常数；
n——转数 r/min；　　R——电枢回路总电阻 Ω；
U——电枢电压 V；　　φ——励磁磁通 Wb；

改变U、φ、R都可以改变电机转速，分别称为调压调速、改变磁通调速和改变电枢回路电阻调速。PWM 调速是调压调速的一种，应用也最为广泛，本次系统采用 PWM 调速的方式。

1.4 系统指标

1) 速度控制范围：0~210rpm
2) 位置控制范围：0~360°
3) PID 参数输入范围 0~100
4) 速度控制误差：±1rpm

2 下位机设计

2.1 总体方案

下位机系统组成框图如图 2-1 所示，此系统以 STM32F103C8T6 单片机作为主控，控制系统部分由 USB 供电，电机驱动部分由 12V 动力电池供电，12V 动力电池电压由 L298N 电机驱动模块降压为 5V 后给霍尔式编码器测速模块供电。

STM32F103C8T6 单片机采集霍尔式编码器测速模块测量的电机转速编码信息，经过处理后输出 PWM 波给 L298N 电机驱动模块进而控制电机转速或位置，单片机通过 USB 串口与上位机进行通信，发送数据给上位机或接收上位机发送的数据。

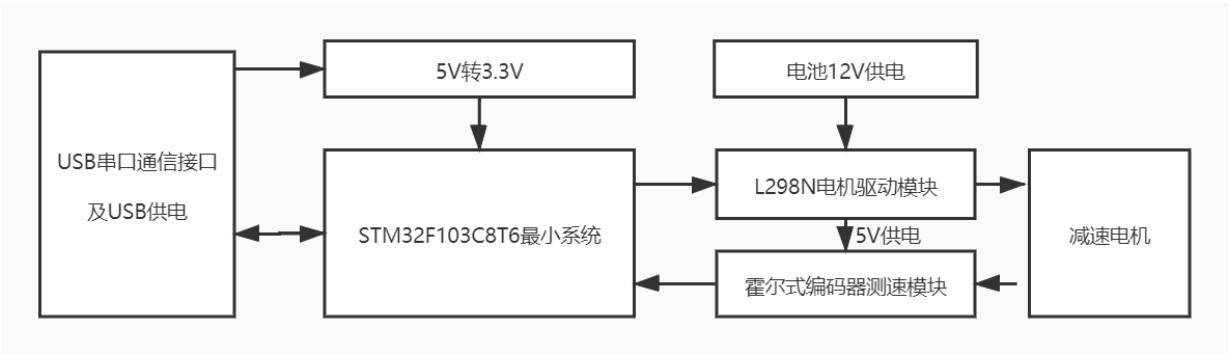


图 2-1 下位机系统框图

2.2 下位机硬件设计

2.2.1 STM32 最小系统的设计

STM32 最小系统由 STM32 主控芯片、时钟电路、复位电路、电源电路、串口通信电路等组成。STM32 最小系统采用杨桃电子出品的最小系统板，其电路如图 2-2 所示（附录一）。此最小系统板采用 CH340C 芯片将 TTL 电平转换为 USB 电平，实现与电脑的通信电路。采用 AMS1117-3.3V 线性稳压芯片将 5V 电源转 3.3V 电源为 STM32 最小系统供电。8MHz 无源晶振为单片机提供外部时钟。

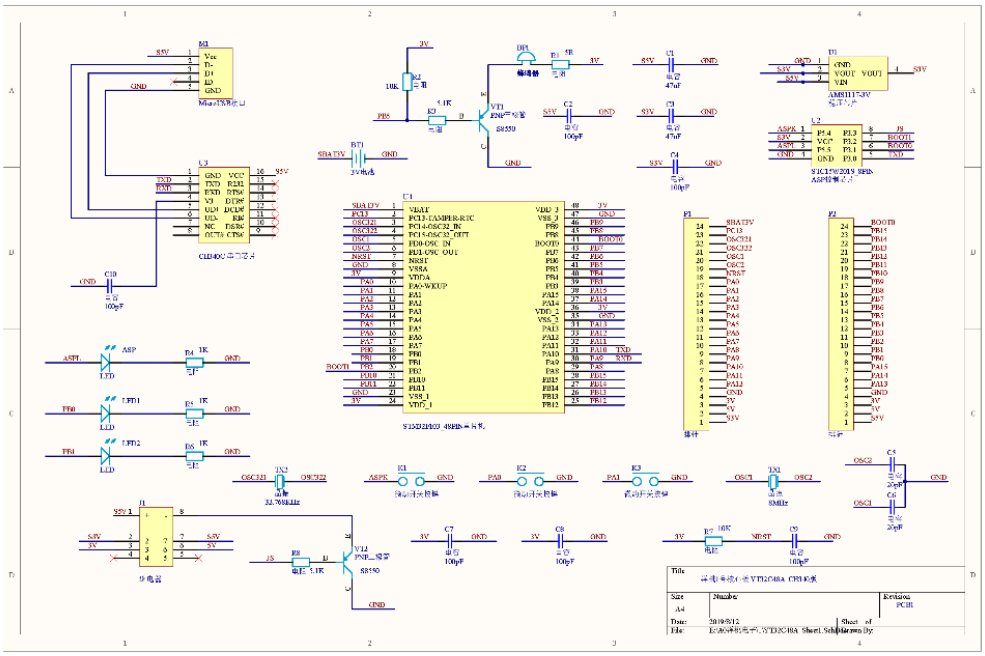


图 2-2 STM32 最小系统原理图

2.2.2 L298N 电机驱动模块

L298N 是 ST 公司的产品，内部包含 4 通道逻辑驱动电路，是一种二相和四相电机的专门驱动芯片，即内含两个 H 桥的高电压大电流双桥式驱动器，接收标准的 TTL 逻辑电平信号，可驱动 4.5V~46V、2A 以下的电机，电流峰值输出可达 3A，其内部结构如下图 2-3 所示。

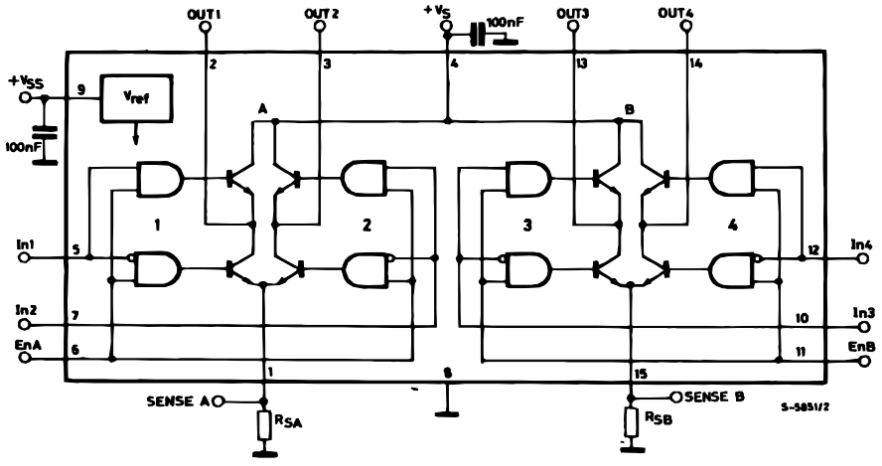


图 2-3 L298N 内部结构图

本次采用 L298N 模块用作电机驱动，其模块实物图如图 2-4 所示，该模块可以驱动两路直流电机，其原理图见附录二

L298N 逻辑功能见表 2-1。

表 2-1 L298N 逻辑功能表

IN1	IN2	ENA	电机状态
×	×	0	电机停止
1	0	1	电机正转
0	1	1	电机反转
0	0	1	电机停止
1	1	1	电机停止

IN3，IN4 的逻辑图与表 2-1 相同。由上表可知 ENA 为低电平时，INx 输入电平对电机控制不起作用，当 ENA 为高电平，输入电平为一高一低，电机正或反转。同为低电平电机停止，同为高电平电机停止。

2.2.3 电机及编码器

此次系统控制的电机型号为 JGB37-520，此款直流减速电机是一款微型减速电机，其上带有一款霍尔传感器测速码盘，其电机的减速比为 1:30。电机的其他参数如下：

- 额定电压：12V
- 减速前转速：630rpm
- 空转电流：250MA
- 堵转最大电流：6.5A

JGB37-520 电机使用的测速码盘模组配有 13 线强磁码盘，AB 双相输出共同利用下，通过计算可得出电机转动 1 圈时，脉冲数可达 780 个。

2.2.4 各模块间的连接

1) 单片机最小系统板与 L298N 电机驱动模块的连接见表 2-2 所示：

表 2-2 单片机与电机驱动模块连接

单片机最小系统板	L298N 电机驱动模块
PA2	EN1
PA11	IN1
PA8	IN2
GND	GND

2) 单片机与编码器连接见表 2-3 所示：

表 2-3 单片机与编码器测速模块连接

单片机最小系统板	编码器测速模块
PB7	A
PB8	B

3) L298N 电机驱动模块与其他模块连接见表 2-4 所示：

表 2-4 L298N 模块与其他模块的连接

L298N 电机驱动模块	电机	电池	编码器测速模块
+12V		正极	
GND		负极	GND
OUT1	M-		
OUT2	M+		
+5V			VCC

2.3 下位机软件设计

下位机软件只要包括电机驱动控制程序、编码器驱动程序、PID 算法程序和通信协议处理程序，本节介绍电机驱动控制程序、编码器驱动程序和 PID 算法程序部分，通信协议处理程序将在第 3 章介绍。下位机软件的总体流程图如图 2-4 所示

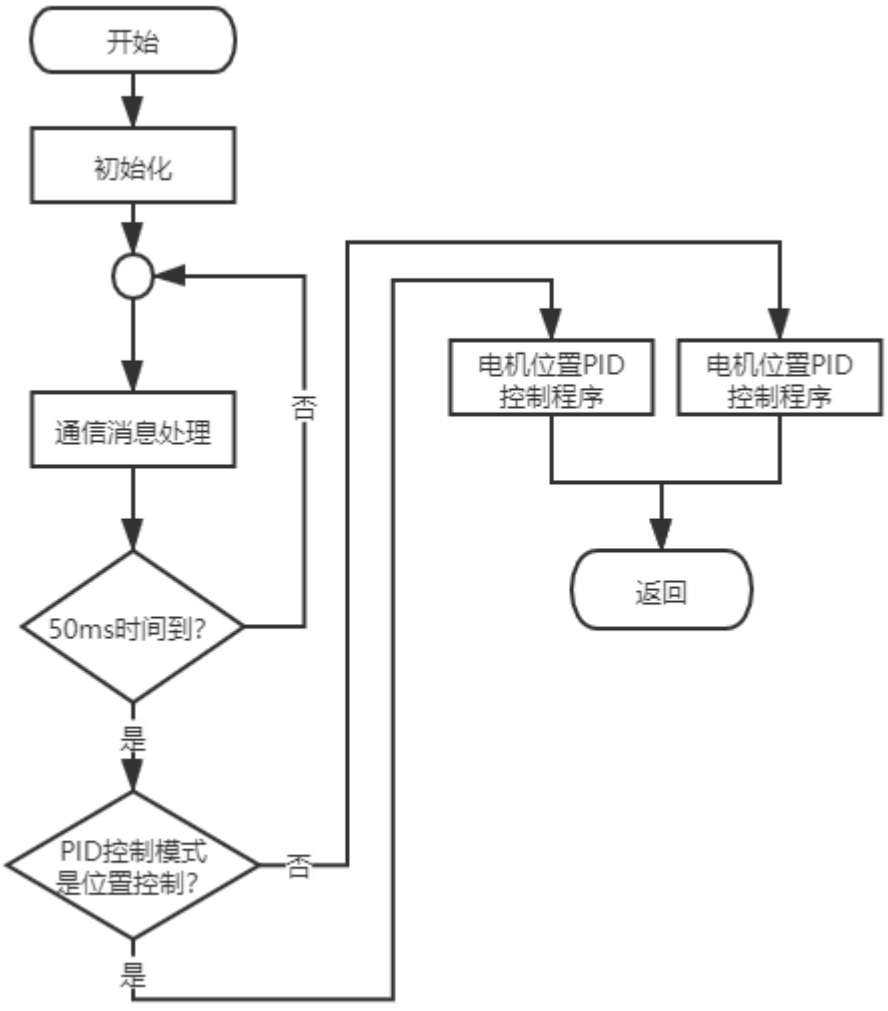


图 2-4 下位机软件总体流程图

2.3.1 电机驱动

这里只讲解核心的部分代码，有些变量的设置，头文件的包含等并没有涉及到，下文出现的所有程序均为核心部分代码，完整的代码请参考附录四。

bsp_motor_tim.h 文件宏定义

```
/*宏定义*/
#define PWM_TIM                          TIM1
#define PWM_TIM_GPIO_AF_ENALBE()         __HAL_AFIO_REMAP_TIM1_ENABLE();
#define PWM_TIM_CLK_ENABLE()
    __HAL_RCC_TIM1_CLK_ENABLE()
#define PWM_CHANNEL_1                    TIM_CHANNEL_1
#define PWM_CHANNEL_2                    TIM_CHANNEL_4
/* 累计 TIM_Period 个后产生一个更新或者中断*/
/* 当定时器从 0 计数到 PWM_PERIOD_COUNT，即为 PWM_PERIOD_COUNT+1 次，为一个定时周期 */
#define PWM_PERIOD_COUNT     (3600)
/* 通用控制定时器时钟源 TIMxCLK = HCLK=72MHz */
/* 设定定时器频率为=TIMxCLK/(PWM_PRESCALER_COUNT)/PWM_PERIOD_COUNT = 10khz */
#define PWM_PRESCALER_COUNT     (2)
/* 最大比较值 */
#define PWM_MAX_PERIOD_COUNT              (PWM_PERIOD_COUNT - 100)
/*PWM 引脚*/
#define PWM_TIM_CH1_GPIO_CLK()
__HAL_RCC_GPIOA_CLK_ENABLE();
#define PWM_TIM_CH1_GPIO_PORT             GPIOA
#define PWM_TIM_CH1_PIN                   GPIO_PIN_8
#define PWM_TIM_CH2_GPIO_CLK()
__HAL_RCC_GPIOA_CLK_ENABLE();
#define PWM_TIM_CH2_GPIO_PORT             GPIOA
#define PWM_TIM_CH2_PIN                   GPIO_PIN_11
```

bsp_motor_tim.c 文件定时器配置

```
TIM_HandleTypeDef  DCM_TimeBaseStructure;
static void TIM_PWMOUTPUT_Config(void)
{
        TIM_OC_InitTypeDef  TIM_OCInitStructure;

        /*使能定时器*/
        PWM_TIM_CLK_ENABLE();
        DCM_TimeBaseStructure.Instance = PWM_TIM;
        /* 累计 TIM_Period 个后产生一个更新或者中断*/
        //当定时器从 0 计数到 PWM_PERIOD_COUNT，即为 PWM_PERIOD_COUNT+1 次，为一个定时周期
        DCM_TimeBaseStructure.Init.Period = PWM_PERIOD_COUNT - 1;
        // 通用控制定时器时钟源 TIMxCLK = HCLK/2=84MHz
        // 设定定时器频率为=TIMxCLK/(PWM_PRESCALER_COUNT+1)
        DCM_TimeBaseStructure.Init.Prescaler = PWM_PRESCALER_COUNT - 1;
        /*计数方式*/
        DCM_TimeBaseStructure.Init.CounterMode = TIM_COUNTERMODE_UP;
        /*采样时钟分频*/
        DCM_TimeBaseStructure.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
        /*初始化定时器*/
        HAL_TIM_PWM_Init(&DCM_TimeBaseStructure);
```

```
21
22          /*PWM 模式配置*/
23          TIM_OCInitStructure.OCMode = TIM_OCMODE_PWM1;
24          TIM_OCInitStructure.Pulse = 0;
25          TIM_OCInitStructure.OCPolarity = TIM_OCPOLARITY_HIGH;
26          TIM_OCInitStructure.OCNPolarity = TIM_OCPOLARITY_HIGH;
27          TIM_OCInitStructure.OCIdleState = TIM_OCIDLESTATE_RESET;
28          TIM_OCInitStructure.OCNIdleState = TIM_OCNIDLESTATE_RESET;
29          TIM_OCInitStructure.OCFastMode = TIM_OCFAST_DISABLE;
30
31          /*配置 PWM 通道*/
32          HAL_TIM_PWM_ConfigChannel(&DCM_TimeBaseStructure,
&TIM_OCInitStructure, PWM_CHANNEL_1);
33          /*开始输出 PWM*/
34          HAL_TIM_PWM_Start(&DCM_TimeBaseStructure,PWM_CHANNEL_1);
35
36          /*配置脉宽*/
37          TIM_OCInitStructure.Pulse = 500;    // 默认占空比为 50%
38          /*配置 PWM 通道*/
39          HAL_TIM_PWM_ConfigChannel(&DCM_TimeBaseStructure,
&TIM_OCInitStructure, PWM_CHANNEL_2);
40          /*开始输出 PWM*/
41          HAL_TIM_PWM_Start(&DCM_TimeBaseStructure,PWM_CHANNEL_2);
42  }
```

配置 STM32 定时器 1 为 PWM 输出模式，用于电机控制。TIM1 的通道 1(PA8)和通道 4(PA11)用于 PWM 输出，连接 L298N 电机驱动模块的 IN1 与 IN2 引脚。

2.3.2 编码器驱动

bsp_encoder.h 文件宏定义

```
1   /* 定时器选择 */
2   #define ENCODER_TIM                          TIM4
3   #define ENCODER_TIM_CLK_ENABLE()        __HAL_RCC_TIM4_CLK_ENABLE()
4   #define ENCODER_TIM_AF_CLK_ENABLE()     __HAL_AFIO_REMAP_TIM4_ENABLE()
5   /* 定时器溢出值 */
6   #define ENCODER_TIM_PERIOD                   65535
7   /* 定时器预分频值 */
8   #define ENCODER_TIM_PRESCALER                0
9   /* 定时器中断 */
10  #define ENCODER_TIM_IRQn                     TIM4_IRQn
11  #define ENCODER_TIM_IRQHandler               TIM4_IRQHandler
12  /* 编码器接口引脚 */
13  #define ENCODER_TIM_CH1_GPIO_CLK_ENABLE()
__HAL_RCC_GPIOB_CLK_ENABLE()
14  #define ENCODER_TIM_CH1_GPIO_PORT            GPIOB
15  #define ENCODER_TIM_CH1_PIN                  GPIO_PIN_6
16  #define ENCODER_TIM_CH2_GPIO_CLK_ENABLE()
__HAL_RCC_GPIOB_CLK_ENABLE()
17  #define ENCODER_TIM_CH2_GPIO_PORT            GPIOB
18  #define ENCODER_TIM_CH2_PIN                  GPIO_PIN_7
19  /* 编码器接口倍频数 */
20  #define ENCODER_MODE                         TIM_ENCODERMODE_TI12
21  /* 编码器接口输入捕获通道相位设置 */
22  #define ENCODER_IC1_POLARITY                 TIM_ICPOLARITY_RISING
```

```
23  #define ENCODER_IC2_POLARITY                    TIM_ICPOLARITY_RISING
24  /* 编码器物理分辨率 */
25  #define ENCODER_RESOLUTION                      13
26  /* 经过倍频之后的总分辨率 */
27  #if (ENCODER_MODE == TIM_ENCODERMODE_TI12)
28  #define ENCODER_TOTAL_RESOLUTION             (ENCODER_RESOLUTION * 4)
/* 4 倍频后的总分辨率 */
29  #else
30  #define ENCODER_TOTAL_RESOLUTION             (ENCODER_RESOLUTION * 2)
/* 2 倍频后的总分辨率 */
31  #endif
32  /* 减速电机减速比 */
33  #define REDUCTION_RATIO                         30
```

bsp_encoder.c 文件定时器配置

```
1   static void TIM_Encoder_Init(void)
2   {
3           TIM_Encoder_InitTypeDef Encoder_ConfigStructure;
4
5           /* 使能编码器接口时钟 */
6           ENCODER_TIM_CLK_ENABLE();
7           /* 定时器初始化设置 */
8           TIM_EncoderHandle.Instance = ENCODER_TIM;
9           TIM_EncoderHandle.Init.Prescaler = ENCODER_TIM_PRESCALER;
10          TIM_EncoderHandle.Init.CounterMode = TIM_COUNTERMODE_UP;
11          TIM_EncoderHandle.Init.Period = ENCODER_TIM_PERIOD;
12          TIM_EncoderHandle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
13          TIM_EncoderHandle.Init.AutoReloadPreload =
TIM_AUTORELOAD_PRELOAD_DISABLE;
14          /* 设置编码器倍频数 */
15          Encoder_ConfigStructure.EncoderMode = ENCODER_MODE;
16          /* 编码器接口通道 1 设置 */
17          Encoder_ConfigStructure.IC1Polarity = ENCODER_IC1_POLARITY;
18          Encoder_ConfigStructure.IC1Selection =
TIM_ICSELECTION_DIRECTTI;
19          Encoder_ConfigStructure.IC1Prescaler = TIM_ICPSC_DIV1;
20          Encoder_ConfigStructure.IC1Filter = 0;
21          /* 编码器接口通道 2 设置 */
22          Encoder_ConfigStructure.IC2Polarity = ENCODER_IC2_POLARITY;
23          Encoder_ConfigStructure.IC2Selection =
TIM_ICSELECTION_DIRECTTI;
24          Encoder_ConfigStructure.IC2Prescaler = TIM_ICPSC_DIV1;
25          Encoder_ConfigStructure.IC2Filter = 0;
26          /* 初始化编码器接口 */
27          HAL_TIM_Encoder_Init(&TIM_EncoderHandle,
&Encoder_ConfigStructure);
28
29          /* 清零计数器 */
30          __HAL_TIM_SET_COUNTER(&TIM_EncoderHandle, 0);
31
32          /* 清零中断标志位 */
33          __HAL_TIM_CLEAR_IT(&TIM_EncoderHandle,TIM_IT_UPDATE);
34          /* 使能定时器的更新事件中断 */
35          __HAL_TIM_ENABLE_IT(&TIM_EncoderHandle,TIM_IT_UPDATE);
36          /* 设置更新事件请求源为：计数器溢出 */
37          __HAL_TIM_URS_ENABLE(&TIM_EncoderHandle);
38
39          /* 设置中断优先级 */
```

```
40          HAL_NVIC_SetPriority(ENCODER_TIM_IRQn, 1, 0);
41          /* 使能定时器中断 */
42          HAL_NVIC_EnableIRQ(ENCODER_TIM_IRQn);
43
44          /* 使能编码器接口 */
45          HAL_TIM_Encoder_Start(&TIM_EncoderHandle, TIM_CHANNEL_ALL);
46  }
```

STM32 芯片内部有专门用来采集增量式编码器方波信号的接口，这些接口实际上是 STM32 定时器的其中一种功能。使用 STM32 的定时器 4 用作编码的驱动定时器，将 TIM4 配置为编码器模式，PB6、PB7 作输入引脚，可以很方便的获取到编码器的值。

2.3.3 PID 算法

本设计采用增量式 PID 算法，分别实现了电机位置 PID 控制算法和电机速度 PID 控制算法。

bsp_pid.h 文件变量定义

```
1   typedef struct{
2           float (*PID_realize)(float);
3   }pid_realize;
4
5   typedef struct
6   {
7           float target_val,Starget_val;     //目标值
8           float actual_val,Sactual_val;     //实际值
9           float err,Serr;             //定义当前偏差值
10          float err_next,Serr_next;       //定义下一个偏差值
11          float err_last,Serr_last;       //定义最后一个偏差值
12          float Kp, Ki, Kd,SKp,SKi,SKd;     //定义比例、积分、微分系数
13          uint32_t Mode;//控制模式(0x55)，位置控制 or 速度控制(0xAA)
14  }_pid;
```

bsp_pid.c 文件位置 PID 算法实现

```
1   float W_PID_realize(float actual_val)
2   {
3           /*计算目标值与实际值的误差*/
4           pid.err=pid.target_val-actual_val;
5           if (pid.err > -5 && pid.err < 5)
6           {
7                   pid.err = 0;
8           }
9           /*PID 算法实现*/
10          pid.actual_val += pid.Kp * (pid.err - pid.err_next) +
11                                  pid.Ki * pid.err +
a)                    pid.Kd * (pid.err - 2 * pid.err_next + pid.err_last);
12          /*传递误差*/
13          pid.err_last = pid.err_next;
14          pid.err_next = pid.err;
15
16          /*返回当前实际值*/
17          return pid.actual_val;
18  }
```

bsp_pid.c 文件速度 PID 算法实现

```
19  float S_PID_realize(float actual_val)
```

```
20  {
21          /*计算目标值与实际值的误差*/
22          pid.Serr=pid.Starget_val-actual_val;
23          /*PID 算法实现*/
24          pid.Sactual_val += pid.SKp*(pid.Serr - pid.Serr_next)
25                                  + pid.SKi*pid.Serr
26                   + pid.SKd*(pid.Serr - 2 * pid.Serr_next +
pid.Serr_last);
27          /*传递误差*/
28          pid.Serr_last = pid.Serr_next;
29          pid.Serr_next = pid.Serr;
30          /*返回当前实际值*/
31          return pid.Sactual_val;
32  }
```

2.3.4 两种 PID 控制算法间的切换

因为此次的设计采用的是单级位置 PID 控制或单级速度 PID 控制，而非串级速度位置 PID 控制，因而需要设计两种 PID 算法的切换程序。

bsp_pid.c 文件 PID 算法回调函数

```
1   float PID_realize(float actual_val,float (*PIDRealize)(float))
2   {
3           return ((*PIDRealize)(actual_val));
4   }
```

bsp_motor_control.c 文件电机 PID 控制函数

```
1   void motor_pid_control(void)
2   {
3     if (is_motor_en == 1)     // 电机在使能状态下才进行控制处理
4     {
5       float cont_val = 0;           // 当前控制值
6       static __IO int32_t Capture_Count = 0;    // 当前时刻总计数值
7       static __IO int32_t Last_Count = 0;       // 上一时刻总计数值
8       int32_t actual_speed = 0;                 // 实际测得速度
9       pid_realize pid_fun;
10      /* 当前时刻总计数值 = 计数器值 + 计数溢出次数 * ENCODER_TIM_PERIOD  */
11      Capture_Count = __HAL_TIM_GET_COUNTER(&TIM_EncoderHandle) +
(Encoder_Overflow_Count * ENCODER_TIM_PERIOD);
12          /* 转轴转速 = 单位时间内的计数值 / 编码器总分辨率 * 时间系数  */
13          actual_speed = ((float)(Capture_Count - Last_Count) /
ENCODER_TOTAL_RESOLUTION / REDUCTION_RATIO) /
(GET_BASIC_TIM_PERIOD()/1000.0/60.0);
14          /* 记录当前总计数值，供下一时刻计算使用 */
15          Last_Count = Capture_Count;
16          if(pid.Mode==0x55)
17          {
18                  pid_fun.PID_realize = W_PID_realize;
19                  cont_val =
PID_realize(Capture_Count,pid_fun.PID_realize);    // 进行 PID 计算
20          #if 1
21                  set_computer_value(SEND_FACT_CMD, CURVES_CH1,
&Capture_Count, 1);               // 给通道 1 发送实际值
22          #endif
23          }
24          else if(pid.Mode==0xAA)
```

```
25          {
26                  pid_fun.PID_realize = S_PID_realize;
27                  cont_val =
PID_realize(actual_speed,pid_fun.PID_realize);    // 进行 PID 计算
28          #if 1
29                  set_computer_value(SEND_FACT_CMD, CURVES_CH1,
&actual_speed,1);               // 给通道 1 发送实际值
30          #endif
31          }
32      if (cont_val > 0)    // 判断电机方向
33      {
34        set_motor_direction(MOTOR_FWD);  //正传
35      }
36      else
37      {
38        cont_val = -cont_val;
39        set_motor_direction(MOTOR_REV);
40      }
41          if(pid.Mode==0x55)
42          {
43                  cont_val = (cont_val > PWM_MAX_PERIOD_COUNT*0.48) ?
PWM_MAX_PERIOD_COUNT*0.48 : cont_val;    // 速度上限处理
44          }
45          else if(pid.Mode==0xAA)
46          {
47                  cont_val = (cont_val > PWM_MAX_PERIOD_COUNT) ?
PWM_MAX_PERIOD_COUNT : cont_val;    // 速度上限处理
48          }
49          set_motor_speed(cont_val);
// 设置 PWM 占空比
50    #if 0
51      set_computer_value(SEND_FACT_CMD, CURVES_CH1, &Capture_Count, 1);
// 给通道 1 发送实际值
52    //#else
53      printf("实际值：%d. 目标值：%.0f\n", actual_speed, get_pid_target());
// 打印实际值和目标值
54    #endif
55    }
56  }
```

3 通信协议设计

串口是一种全双工串行异步通信接口，LabVIEW 与单片机之间的通信需要遵循一个通信协议才能够正确解析发送和接收的指令，执行正确的功能。

3.1 指令包格式

字节数	4bytes	1bytes	4bytes	1bytes	……	……	……	1 bytes
名称	包头	通道地址	包长度	指令	参数 1	…	参数 2	校验和
内容	0x59485A53	xxxx	xxxx	xxxx	xxxx	…	xxxx	xxxx

1) 所有多字节的低字节在前
2) 包头固定为四字节的 0x59485A53；
3) 通道地址为 0x01，可做扩展；
4) 包长度为从包头到校验的所有数据长度。
5) 指令为相应的功能码。
6) 参数为指令需要参数时加入。
7) 校验为校验和方式—8 位。

3.2 指令汇总

下位机到上位机的通信协议指令见表 3-1，上位机到下位机通信协议指令见表 3-2。各指令详解见 3.3 小节指令详解。

表 3-1 下位机到上位机指令集

下位机 ---->>上位机		
指令	参数	功能
0x01	1 个，目标值，int 类型	设置上位机通道的目标值
0x02	1 个，实际值，int 类型	设置上位机通道实际值
0x03	3 个，P、I、D，int 类型	设置上位机位置控制 PID 值
0x04	无	设置上位机启动指令（同步上位机的按钮状态）
0x05	无	设置上位机停止指令（同步上位机的按钮状态）
0x06	保留	保留
0x07	1 个，PID 控制模式，int 类型	设置上位机 PID 控制模式（同步上位机的按钮状态）
0x08	3 个，P、I、D，int 类型	设置上位机速度控制 PID 值

表 3-2 上位机到下位机指令集

上位机---->>下位机		
指令	参数	功能
0x10	3 个，P、I、D，int 类型	设置下位机的 PID 值
0x11	1 个，目标值，int 类型	设置下位机的目标值
0x12	无	启动指令
0x13	无	停止指令
0x14	无	复位指令
0x15	保留	保留
0x16	1 个，int 类型	设置 PID 控制模式

3.3 指令详解

1) 设置上位机通道的目标值

功能：下位机向上位机发送目标值

目标值：位置控制模式下为编码器目标脉冲数；速度控制模式下为电机的目标转动速度。

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	目标值	校验和
内容	0x59485A53	0x01	0x0F	0x01	xxxx	xxxx

2) 设置上位机通道实际值

功能：下位机向上位机发送实际值

实际值：位置控制模式下为编码器实际脉冲数；速度控制模式下为电机的实际转动速度。

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	实际值	校验和
内容	0x59485A53	0x01	0x0F	0x02	xxxx	xxxx

3) 设置上位机位置控制 PID 值

功能：下位机向上位机发送位置 PID 控制 PID 参数

P 参数：比例系数

I 参数：积分系数

D 参数：微分系数

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	4bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	P 参数	I 参数	D 参数	校验和
内容	0x59485A53	0x01	0x17	0x03	xxxx	xxxx	xxxx	xxxx

4) 设置上位机启动指令

功能：同步上位机启动按钮状态

字节数	4bytes	1bytes	4bytes	1bytes	1 bytes
名称	包头	通道地址	包长度	指令	校验和
内容	0x59485A53	0x01	0x0B	0x04	xxxx

5) 设置上位机停止指令

功能：同步上位机停止按钮状态

字节数	4bytes	1bytes	4bytes	1bytes	1 bytes
名称	包头	通道地址	包长度	指令	校验和
内容	0x59485A53	0x01	0x0B	0x05	xxxx

6) 设置上位机 PID 控制模式

功能：下位机向上位机发送 PID 控制模式，用于初始化上位机 PID 控制模式按钮状态

目标值：0x55 位置控制模式；0xAA 速度控制模式

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	目标值	校验和
内容	0x59485A53	0x01	0x0F	0x07	xxxx	xxxx

7) 设置上位机速度控制 PID 值

功能：下位机向上位机发送速度 PID 控制 PID 参数

P 参数：比例系数

I 参数：积分系数

D 参数：微分系数

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	4bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	P 参数	I 参数	D 参数	校验和
内容	0x59485A53	0x01	0x17	0x08	xxxx	xxxx	xxxx	xxxx

8) 设置下位机的 PID 值

功能：上位机向下位机发送 PID 控制 PID 参数

P 参数：比例系数

I 参数：积分系数

D 参数：微分系数

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	4bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	P 参数	I 参数	D 参数	校验和
内容	0x59485A53	0x01	0x17	0x10	xxxx	xxxx	xxxx	xxxx

9) 上位机发送目标值

功能：上位机向下位机发送目标值

目标值：位置控制模式下为编码器目标脉冲数；速度控制模式下为电机的目标转动速度。

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	目标值	校验和
内容	0x59485A53	0x01	0x0F	0x11	xxxx	xxxx

10) 上位机发送启动指令

功能：上位机发送启动指令启动下位机电机

字节数	4bytes	1bytes	4bytes	1bytes	1 bytes
名称	包头	通道地址	包长度	指令	校验和
内容	0x59485A53	0x01	0x0B	0x12	xxxx

11) 上位机发送停止指令

功能：上位机发送停止指令停止下位机电机

字节数	4bytes	1bytes	4bytes	1bytes	1 bytes
名称	包头	通道地址	包长度	指令	校验和
内容	0x59485A53	0x01	0x0B	0x13	xxxx

12) 上位机发送复位指令

功能：上位机向下位机发送复位指令用来复位下位机控制器

字节数	4bytes	1bytes	4bytes	1bytes	1 bytes
名称	包头	通道地址	包长度	指令	校验和
内容	0x59485A53	0x01	0x0B	0x14	xxxx

13) 上位机发送 PID 控制模式指令

功能：上位机向下位机发送 PID 控制模式改变指令

控制模式：0x55 位置控制模式；0xAA 速度控制模式

字节数	4bytes	1bytes	4bytes	1bytes	4bytes	1 bytes
名称	包头	通道地址	包长度	指令	控制模式	校验和
内容	0x59485A53	0x01	0x0F	0x16	xxxx	xxxx

3.3 下位机通信协议程序实现

protocol.c 文件核心代码

```
57  struct prot_frame_parser_t
58  {
59      uint8_t *recv_ptr;
60      uint16_t r_oft;
61      uint16_t w_oft;
62      uint16_t frame_len;
63      uint16_t found_frame_head;
64  };
65
66  static struct prot_frame_parser_t parser;
67
68  static uint8_t recv_buf[PROT_FRAME_LEN_RECV];
69
70  uint8_t check_sum(uint8_t init, uint8_t *ptr, uint8_t len )
71  {
72    uint8_t sum = init;
73
74    while(len--)
75    {
76      sum += *ptr;
77      ptr++;
```

```
78    }
79
80    return sum;
81  }
82
83  static uint8_t get_frame_type(uint8_t *frame, uint16_t head_oft)
84  {
85      return (frame[(head_oft + CMD_INDEX_VAL) % PROT_FRAME_LEN_RECV] &
0xFF);
86  }
87
88  static uint16_t get_frame_len(uint8_t *frame, uint16_t head_oft)
89  {
90      return ((frame[(head_oft + LEN_INDEX_VAL + 0) %
PROT_FRAME_LEN_RECV] <<  0) |
91              (frame[(head_oft + LEN_INDEX_VAL + 1) %
PROT_FRAME_LEN_RECV] <<  8) |
92              (frame[(head_oft + LEN_INDEX_VAL + 2) %
PROT_FRAME_LEN_RECV] << 16) |
93              (frame[(head_oft + LEN_INDEX_VAL + 3) %
PROT_FRAME_LEN_RECV] << 24));    // 合成帧长度
94  }
95
96  static uint8_t get_frame_checksum(uint8_t *frame, uint16_t head_oft,
uint16_t frame_len)
97  {
98      return (frame[(head_oft + frame_len - 1) % PROT_FRAME_LEN_RECV]);
99  }
100
101 static int32_t recvbuf_find_header(uint8_t *buf, uint16_t ring_buf_len,
uint16_t start, uint16_t len)
102 {
103     uint16_t i = 0;
104
105     for (i = 0; i < (len - 3); i++)
106     {
107         if (((buf[(start + i + 0) % ring_buf_len] <<  0) |
108              (buf[(start + i + 1) % ring_buf_len] <<  8) |
109              (buf[(start + i + 2) % ring_buf_len] << 16) |
110              (buf[(start + i + 3) % ring_buf_len] << 24)) ==
FRAME_HEADER)
111         {
112             return ((start + i) % ring_buf_len);
113         }
114     }
115     return -1;
116 }
117
118 static int32_t recvbuf_get_len_to_parse(uint16_t frame_len, uint16_t
ring_buf_len,uint16_t start, uint16_t end)
119 {
120     uint16_t unparsed_data_len = 0;
121
122     if (start <= end)
123         unparsed_data_len = end - start;
124     else
125         unparsed_data_len = ring_buf_len - start + end;
126
```

```
127     if (frame_len > unparsed_data_len)
128         return 0;
129     else
130         return unparsed_data_len;
131 }
132
133 static void recvbuf_put_data(uint8_t *buf, uint16_t ring_buf_len,
uint16_t w_oft,
134         uint8_t *data, uint16_t data_len)
135 {
136     if ((w_oft + data_len) > ring_buf_len)               // 超过缓冲区尾
137     {
138         uint16_t data_len_part = ring_buf_len - w_oft;     // 缓冲区剩余
长度
139
140         /* 数据分两段写入缓冲区*/
141         memcpy(buf + w_oft, data, data_len_part);
// 写入缓冲区尾
142         memcpy(buf, data + data_len_part, data_len - data_len_part);
// 写入缓冲区头
143     }
144     else
145         memcpy(buf + w_oft, data, data_len);    // 数据写入缓冲区
146 }
147
148 static uint8_t protocol_frame_parse(uint8_t *data, uint16_t *data_len)
149 {
150     uint8_t frame_type = CMD_NONE;
151     uint16_t need_to_parse_len = 0;
152     int16_t header_oft = -1;
153     uint8_t checksum = 0;
154
155     need_to_parse_len = recvbuf_get_len_to_parse(parser.frame_len,
PROT_FRAME_LEN_RECV, parser.r_oft, parser.w_oft);    // 得到为解析的数据长度
156     if (need_to_parse_len < 9)     // 肯定还不能同时找到帧头和帧长度
157         return frame_type;
158
159     /* 还未找到帧头，需要进行查找*/
160     if (0 == parser.found_frame_head)
161     {
162         /* 同步头为四字节，可能存在未解析的数据中最后一个字节刚好为同步头第一
个字节的情况，
163            因此查找同步头时，最后一个字节将不解析，也不会被丢弃*/
164         header_oft = recvbuf_find_header(parser.recv_ptr,
PROT_FRAME_LEN_RECV, parser.r_oft, need_to_parse_len);
165         if (0 <= header_oft)
166         {
167             /* 已找到帧头*/
168             parser.found_frame_head = 1;
169             parser.r_oft = header_oft;
170
171             /* 确认是否可以计算帧长*/
172             if (recvbuf_get_len_to_parse(parser.frame_len,
PROT_FRAME_LEN_RECV,
173                     parser.r_oft, parser.w_oft) < 9)
174                 return frame_type;
175         }
176         else
```

```
177         {
178             /* 未解析的数据中依然未找到帧头，丢掉此次解析过的所有数据*/
179             parser.r_oft = ((parser.r_oft + need_to_parse_len - 3) %
PROT_FRAME_LEN_RECV);
180             return frame_type;
181         }
182     }
183
184     /* 计算帧长，并确定是否可以进行数据解析*/
185     if (0 == parser.frame_len)
186     {
187         parser.frame_len = get_frame_len(parser.recv_ptr,
parser.r_oft);
188         if(need_to_parse_len < parser.frame_len)
189             return frame_type;
190     }
191
192     /* 帧头位置确认，且未解析的数据超过帧长，可以计算校验和*/
193     if ((parser.frame_len + parser.r_oft - PROT_FRAME_LEN_CHECKSUM) >
PROT_FRAME_LEN_RECV)
194     {
195         /* 数据帧被分为两部分，一部分在缓冲区尾，一部分在缓冲区头 */
196         checksum = check_sum(checksum, parser.recv_ptr + parser.r_oft,
197                 PROT_FRAME_LEN_RECV - parser.r_oft);
198         checksum = check_sum(checksum, parser.recv_ptr,
parser.frame_len -
199                 PROT_FRAME_LEN_CHECKSUM + parser.r_oft -
PROT_FRAME_LEN_RECV);
200     }
201     else
202     {
203         /* 数据帧可以一次性取完*/
204         checksum = check_sum(checksum, parser.recv_ptr + parser.r_oft,
parser.frame_len - PROT_FRAME_LEN_CHECKSUM);
205     }
206
207     if (checksum == get_frame_checksum(parser.recv_ptr, parser.r_oft,
parser.frame_len))
208     {
209         /* 校验成功，拷贝整帧数据 */
210         if ((parser.r_oft + parser.frame_len) > PROT_FRAME_LEN_RECV)
211         {
212             /* 数据帧被分为两部分，一部分在缓冲区尾，一部分在缓冲区头*/
213             uint16_t data_len_part = PROT_FRAME_LEN_RECV -
parser.r_oft;
214             memcpy(data, parser.recv_ptr + parser.r_oft,
data_len_part);
215             memcpy(data + data_len_part, parser.recv_ptr,
parser.frame_len - data_len_part);
216         }
217         else
218         {
219             /* 数据帧可以一次性取完*/
220             memcpy(data, parser.recv_ptr + parser.r_oft,
parser.frame_len);
221         }
222         *data_len = parser.frame_len;
223         frame_type = get_frame_type(parser.recv_ptr, parser.r_oft);
```

```
224
225         /* 丢弃缓冲区中的命令帧*/
226         parser.r_oft = (parser.r_oft + parser.frame_len) %
PROT_FRAME_LEN_RECV;
227     }
228     else
229     {
230         /* 校验错误，说明之前找到的帧头只是偶然出现的废数据*/
231         parser.r_oft = (parser.r_oft + 1) % PROT_FRAME_LEN_RECV;
232     }
233     parser.frame_len = 0;
234     parser.found_frame_head = 0;
235
236     return frame_type;
237 }
238
239 void protocol_data_recv(uint8_t *data, uint16_t data_len)
240 {
241     recvbuf_put_data(parser.recv_ptr, PROT_FRAME_LEN_RECV,
parser.w_oft, data, data_len);    // 接收数据
242     parser.w_oft = (parser.w_oft + data_len) % PROT_FRAME_LEN_RECV;
// 计算写偏移
243 }
```

4 上位机设计

4.1 整体方案设计

LabVIEW 上位机主要由前面板和后面板组成，其整体架构如图 4-1 所示。前面板由两个选项卡组成，默认选项卡界面是 PID 调试界面，负责 PID 参数的调整和输出曲线的显示；另一个选项卡为电机控制界面，用与控制电机的速度和位置。

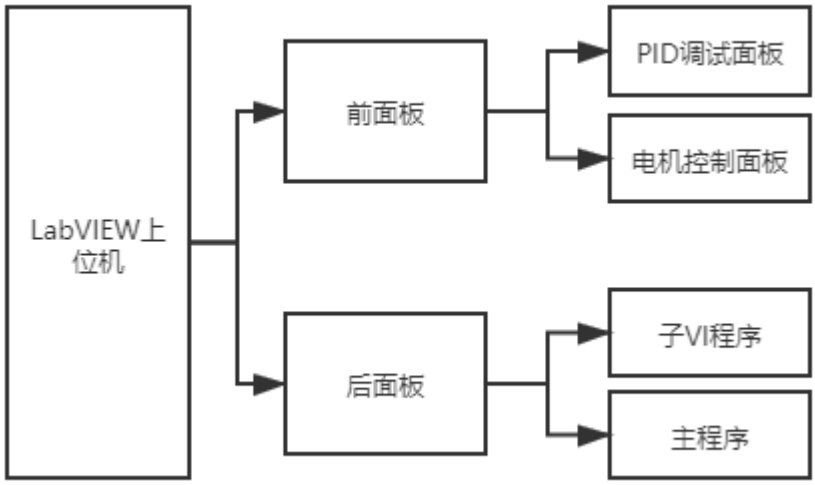


图 4-1 Lab VIE 上位机系统整体框架

4.2 前面板设计

本设计上位机的前面板采用了一个选项卡控件实现了不同界面的切换，如图 4-2 和图 4-3 所示。PID 调参界面主要由三部分组成：串口配置部分、PID 配置部分和曲线显示部分。

串口配置部分实现串口参数的选择，LabVIEW 的 VISA 控件需要输入串口配置参数才能工作，需要输入的参数包括但不限于 VISA 资源名称、波特率、校验方式、数据位、停止位。

PID 配置部分用与电机 PID 调参，PID 参数对应 3 个数值输入控件，当点击发送 PID 按钮时，PID 参数会被打包为符合通信协议规定的数据包通过 VISA 串口发送给下位机。目标值既为电机的输入，当点击发送目标值按钮时，目标值会被打包为符合通信协议规定的数据包发送给下位机。

曲线显示部分利用波形图表显示实际值与目标值，通过对两条曲线的观察可以随时直观的观察 PID 调整的效果，方便 PID 参数的调节。

速度控制按钮可以切换电机的控制方式，有位置控制和速度控制，对应电机控制界

面的位置控制和速度控制功能。

电机控制界面主要由四部分组成：电机转速控制部分、电机位置控制部分、转速显示部分和曲线显示部分。

在速度控制模式下旋转转速控制旋钮可以控制电机的转动速度，同时转速表会实时显示实际的转速，波形图表会绘制出转速的实时曲线；在位置控制模式下旋转电机位置

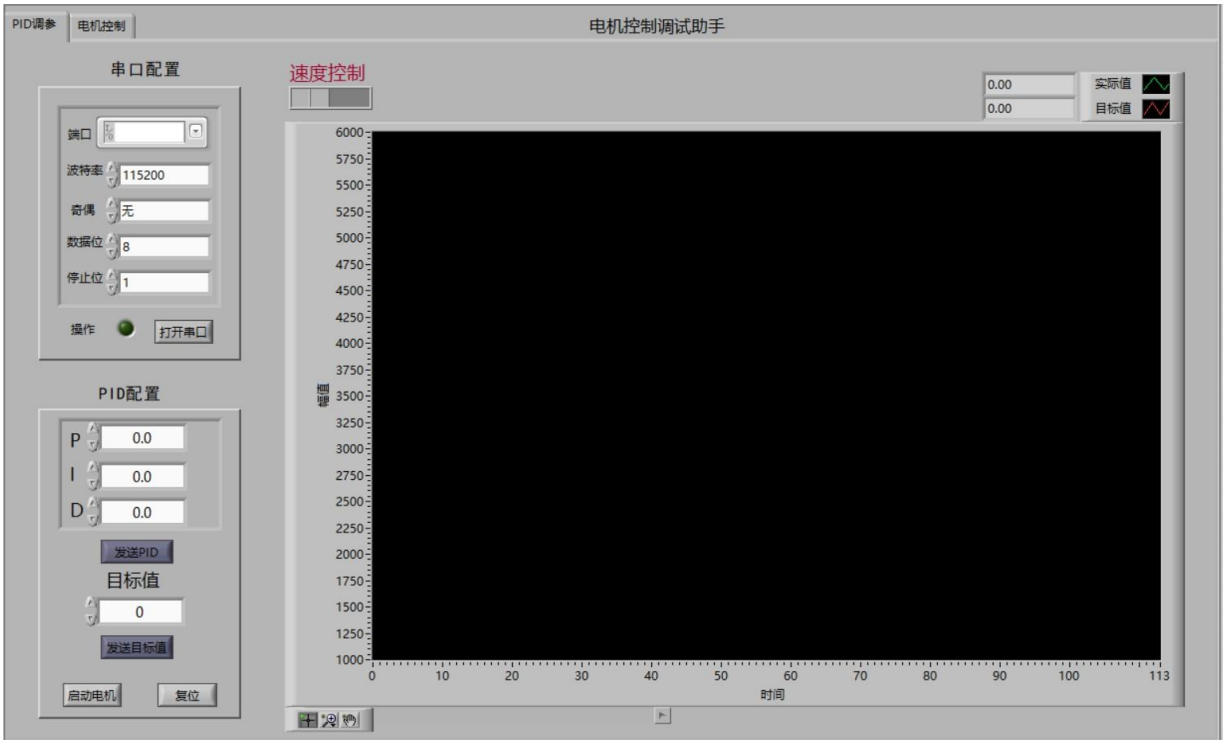


图 4-2 LabVIEW 前面板 PID 调参界面

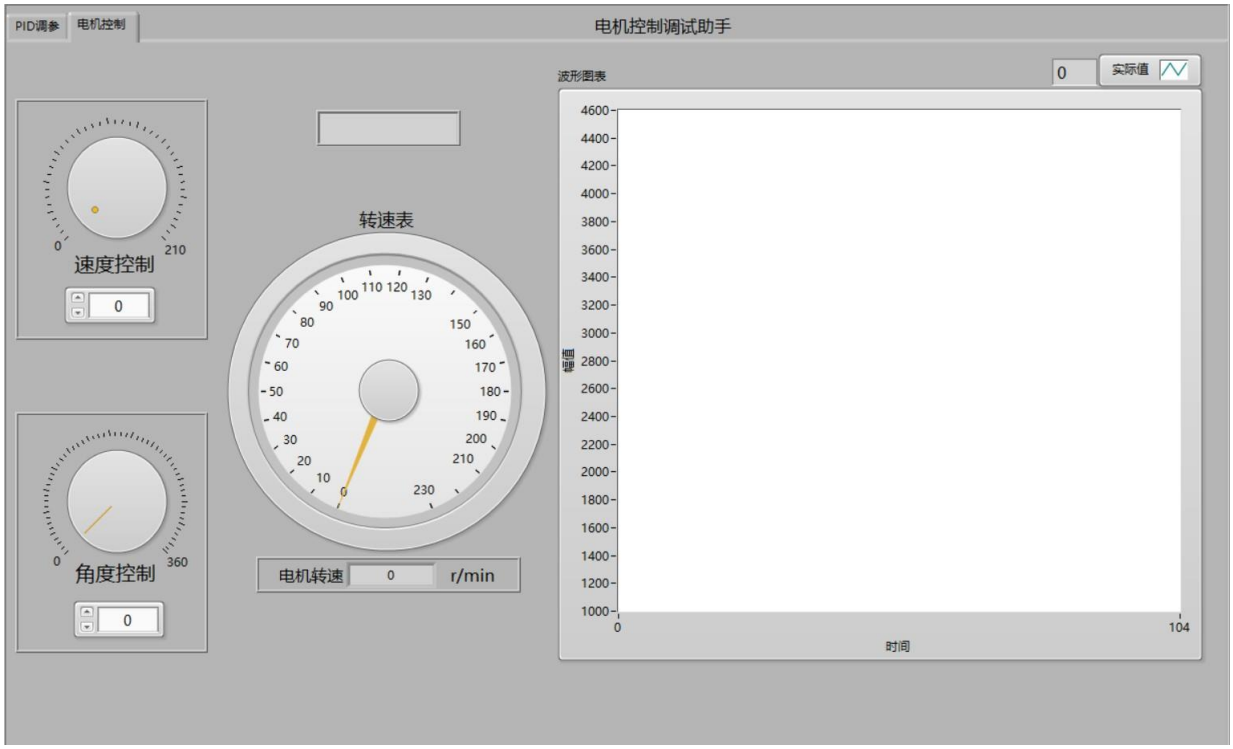


图 4-3 LabVIEW 前面板电机控制界面

旋钮可以控制电机转动的角度，波形图表会绘制出电机位置的实时曲线。

4.3 后面板程序设计

后面板程序包括两部分：主 VI 和子 VI。此设计用到的子 VI 有： PID 转换子 VI、获取 PID 参数子 VI、获取实际值子 VI、获取指令字节子 VI、数据打包子 VI。

4.3.1 主 VI 的设计

主 VI 如图 4-2（附录三）所示，它主要由一个 while 主循环、顺序结构和条件结构组成。在第一个顺序结构的第一帧中处理了一些控件的使能、禁用状态，使界面操作符合逻辑。在第二帧中嵌套了一个叠层顺序结构用于处理串口收发数据。

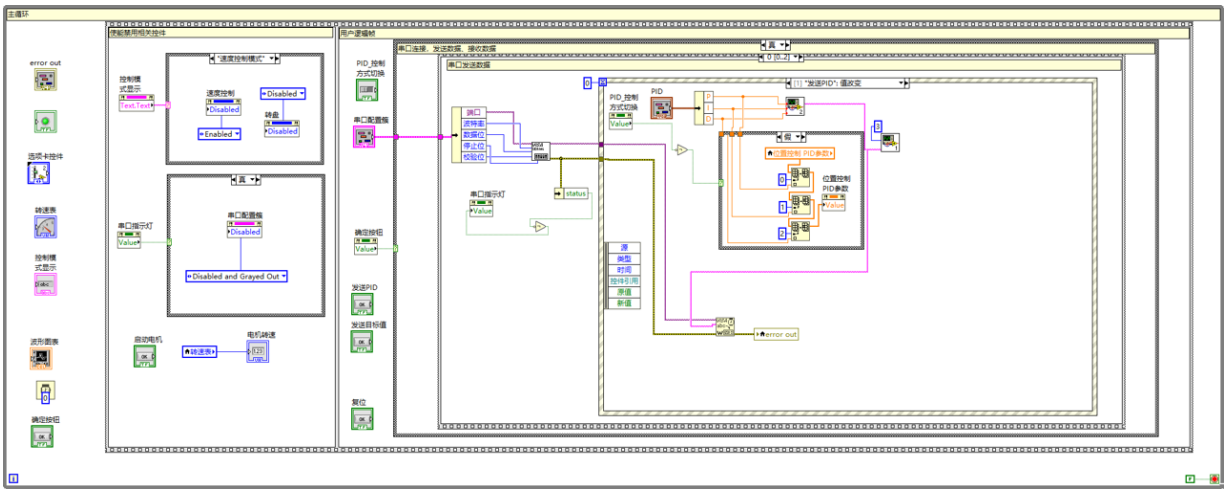


图 4-4 LabVIEW 主程序

以下是对主程序的分块介绍：

1) 串口收发叠层顺序结构部分

第 0 帧如图 4-5 所示，这一帧主要实现串口发送部分的功能，在其中嵌入了一个事件结构，用来处理不同的事件。事件结构所包含的事件如图 4-8 所示。

第 1 帧如图 4-6 所示，这一帧用作延时处理。

第 2 帧如图 4-7 所示，这一帧主要实现串口接收部分的功能，在验证接收到的数据为下位机发送的指令包后，会进入外层条件结构作数据缓存，然后由获取指令字节子 VI 提取指令字节，之后将指令字节输入条件结构由不同条件执行不同的程序做进一步处理。

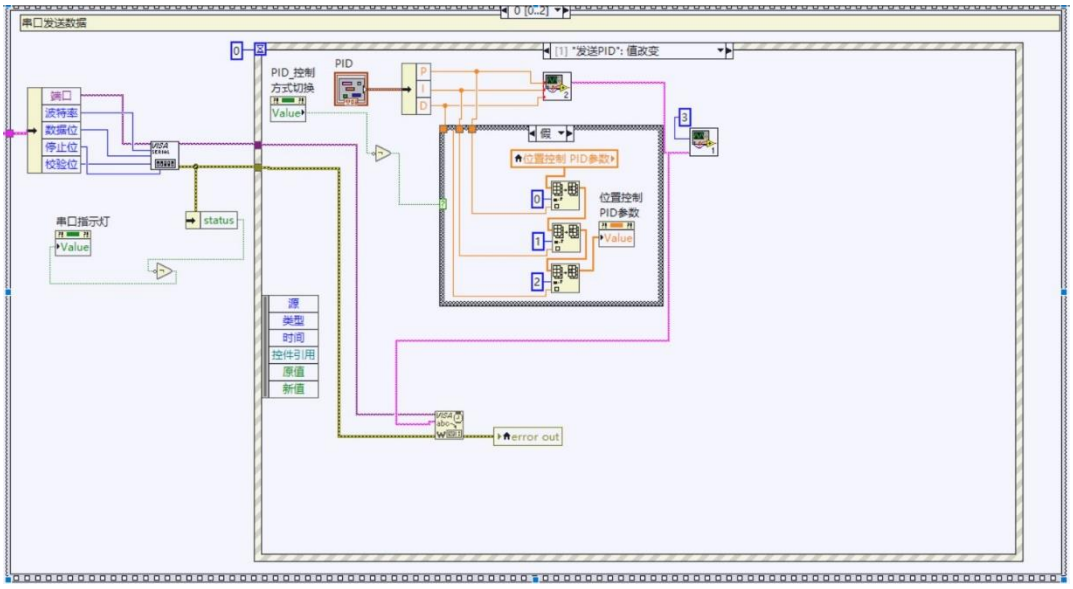


图 4-5 串口收发叠层顺序结构第 0 帧

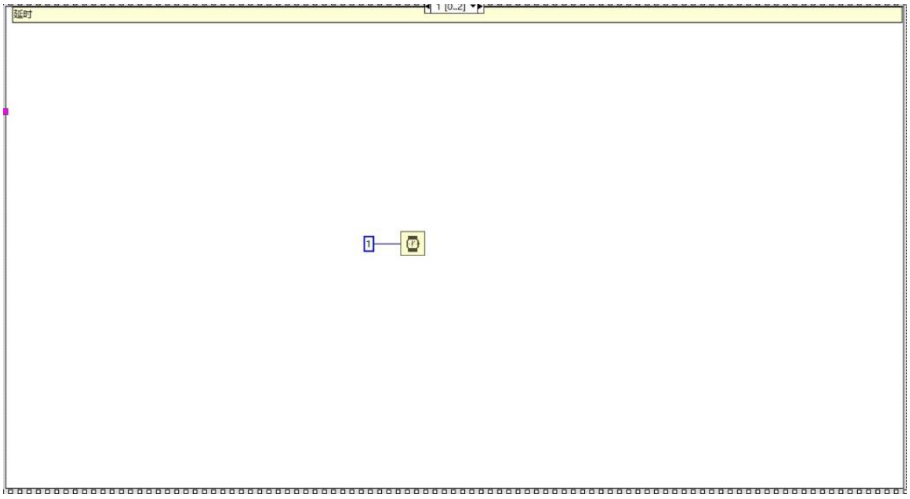


图 4-6 串口收发叠层顺序结构第 1 帧

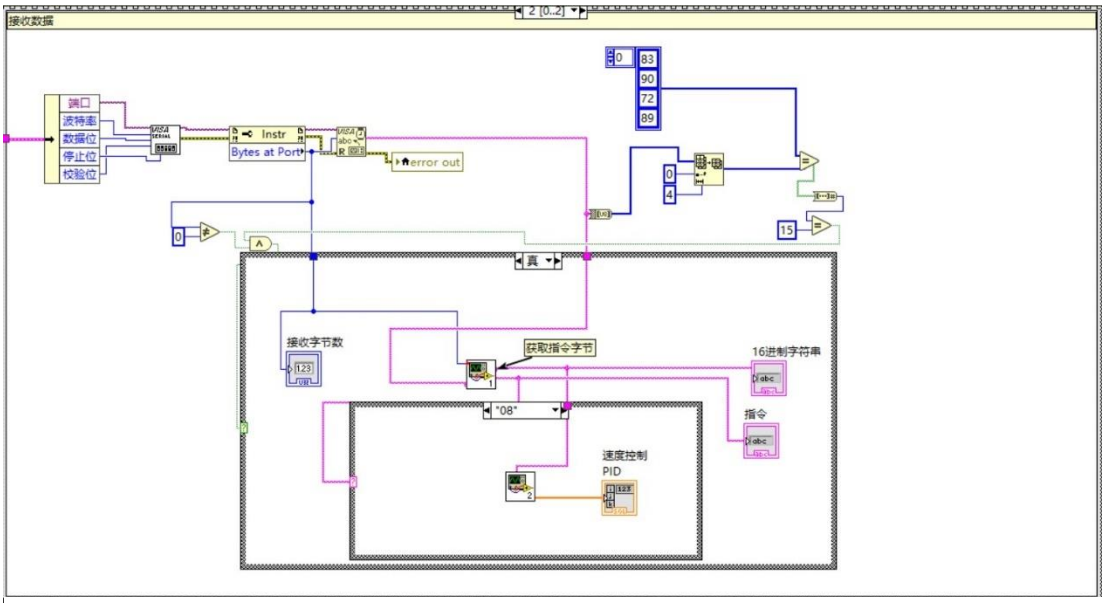


图 4-7 串口收发叠层顺序结构第 2 帧

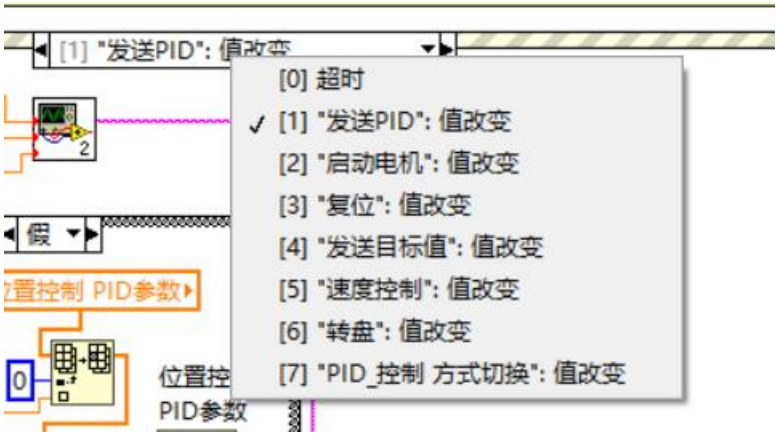


图 4-8 事件结构所含事件图

2) 串口发送帧事件结构

如图 4-9 所示，分别为事件结构的事件 1 到事件 7 的程序图，事件结构的超时事件在此并没有使用因而并没有截图。

事件 1 为“发送 PID”值改变事件。当点击前面板发送 PID 按钮时会触发该事件，在此事件分支内，前面板 PID 数值控件内的 P、I、D 值会根据不同的 PID 控制模式存储到不同的数组变量内，并将 PID 值打包发送给下位机。

事件 2 为“启动电机”值改变事件。当点击前面板启动电机\停止电机按钮时会触发此事件。在此事件分支内，根据前面板启动电机\停止电机按钮按钮的值来判断需要向下位机发送指令的类别。之后由判断结果向下位机发送指令。

事件 3 为“复位”值改变事件。当点击前面板复位按钮时会触发该事件向下位机发送复位指令。

事件 4 为“发送目标值”值改变事件。当点击前面板发送目标值按钮时会触发该事件。在此事件分支内，会将前面板中目标值数值控件中的数值打包发送给下位机。

事件 5、6 分别为“速度控制”值改变事件、“转盘”值改变事件。当切换到电机控制界面并改变速度控制或转盘旋钮的值时会触发该事件。并通过发送目标值指令向下位机发送控件中的数值。

事件 7 为“PID_控制方式切换”值改变事件。当点击前面板 PID 控制模式切换按钮时会触发该事件。在此事件分支内会根据 PID 控制模式切换按钮的值发送相应的控制指令给下位机。下位机由此指令来判断该执行什么 PID 控制算法。

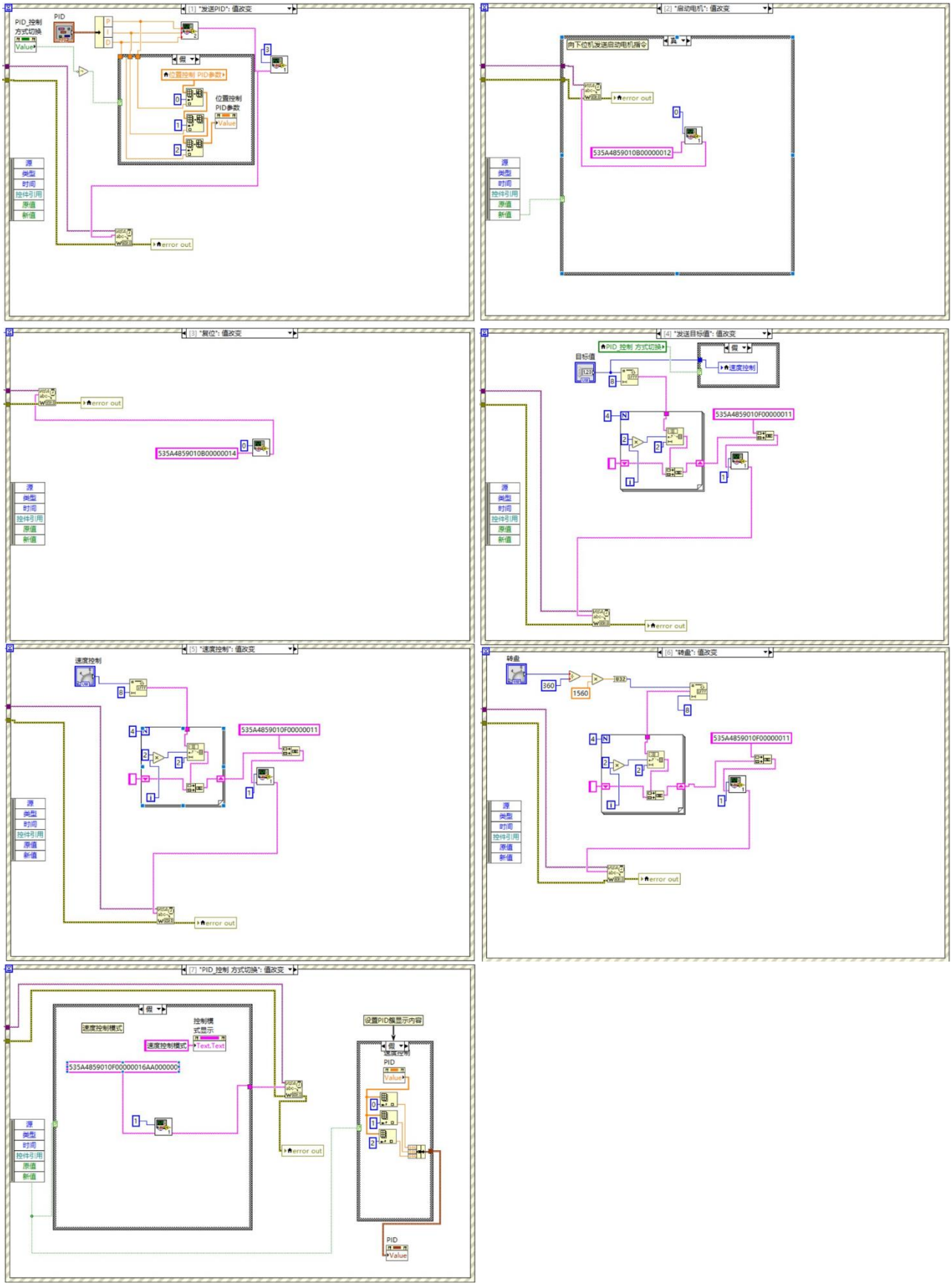


图 4-9 事件结构中各事件程序图

4.3.2 获取指令字节子 VI

获取指令字节子 VI 的功能是将串口接收的命令字符串中的指令字节提取出来。其核心思想是将字符串转换为字节数组，然后将指令字节对应的数组元素取出转化为 16 进制字符串后拼接在一起。具体程序如图 4-11 所示。

图 4-10 获取指令字节子 VI 前面板

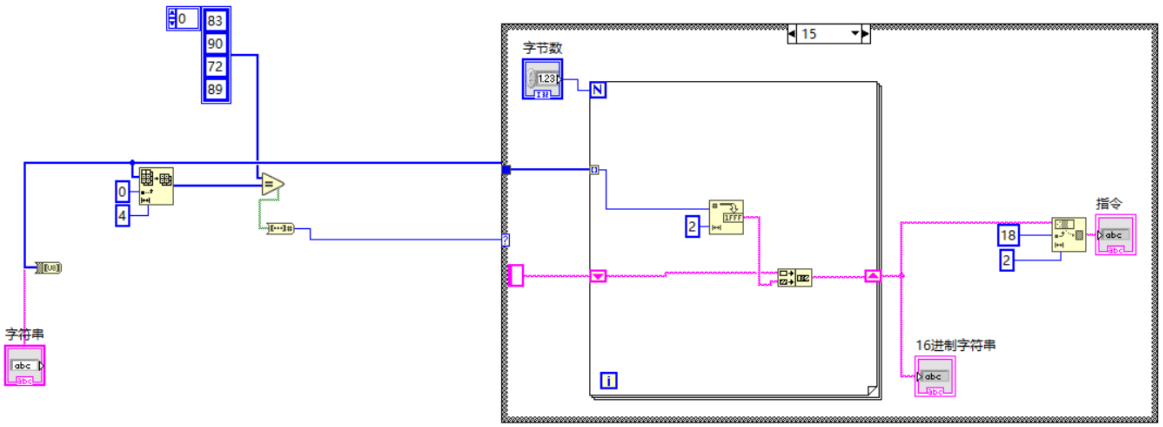


图 4-11 获取指令字节子 VI 后面板

4.3.3 获取实际值子 VI

获取实际值子 VI 的功能是将串口接收的命令字符串中的下位机发送的实际值提取出来，其实现原理与湖区指令字节子 VI 相同。其后面板如图所示。

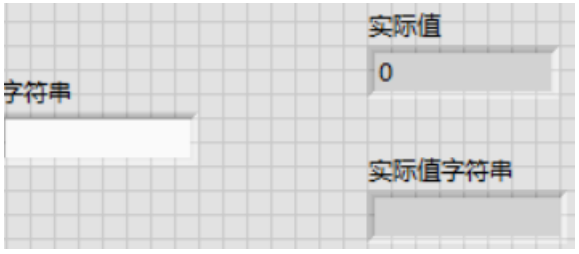


图 4-12 获取实际值子 VI 前面板

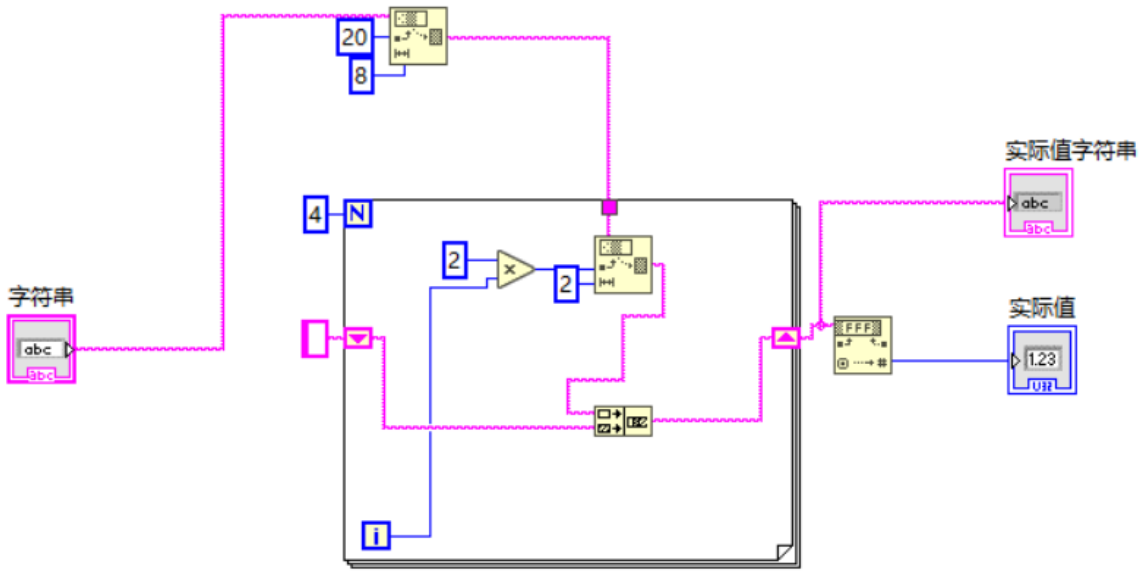


图 4-13 获取实际值子 VI 后面板

4.3.4 获取 PID 参数子 VI

获取 PID 参数子 VI 的主要功能是将串口接收的命令字符串中的下位机发送的 PID 值提取出来，实现原理与上面的大体相同，只是因为 PID 参数是 4 字节的，下位机发送时是低字节在前，所以需要按字节反转字符串，这部分代码借助与 for 循环与移位寄存器器实现。具体程序如图 4-14 所示：

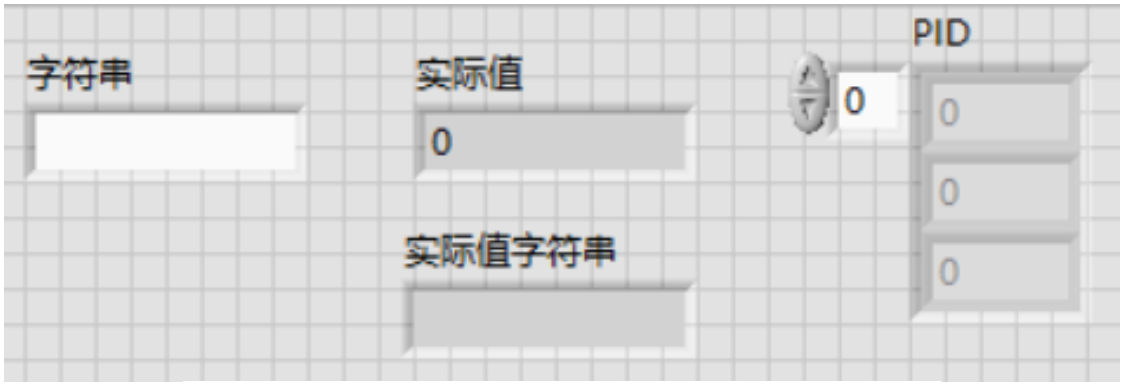


图 4-13 获取 PID 参数子 VI 前面板

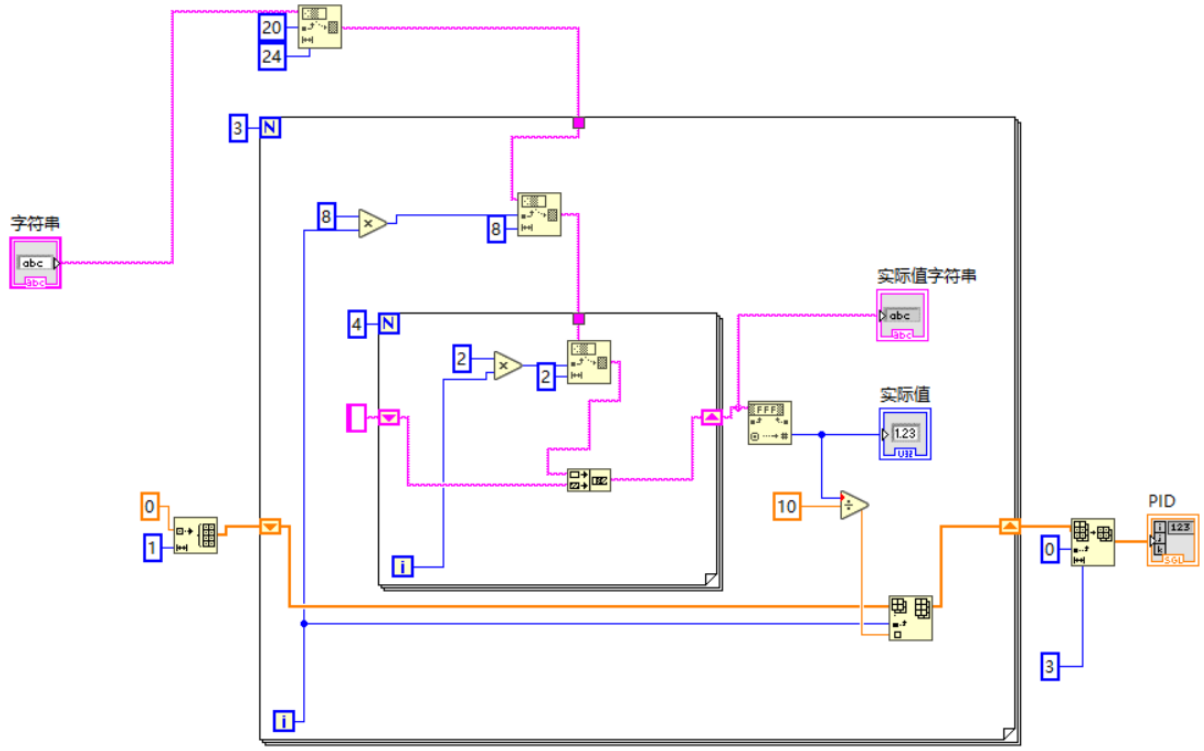


图 4-14 获取 PID 参数子 VI 后面板

4.3.5 数据打包子 VI

数据打包子 VI 的功能为计算除校验位之外的校验和，并将其添加到协议包中，最后将字节数组转换为字符串。结果字符串可直接通过串口发送给下位机。命令数即要发送几个命令字节，1 个命令有 4 个字节。其具体程序如图 4-16 所示。

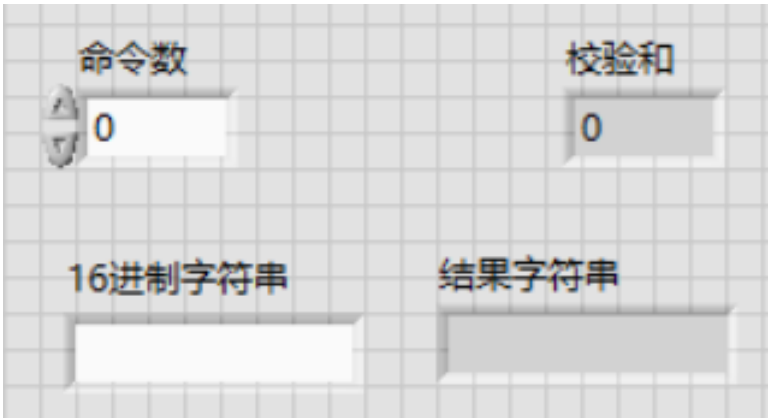


图 4-15 数据打包子 VI 前面板

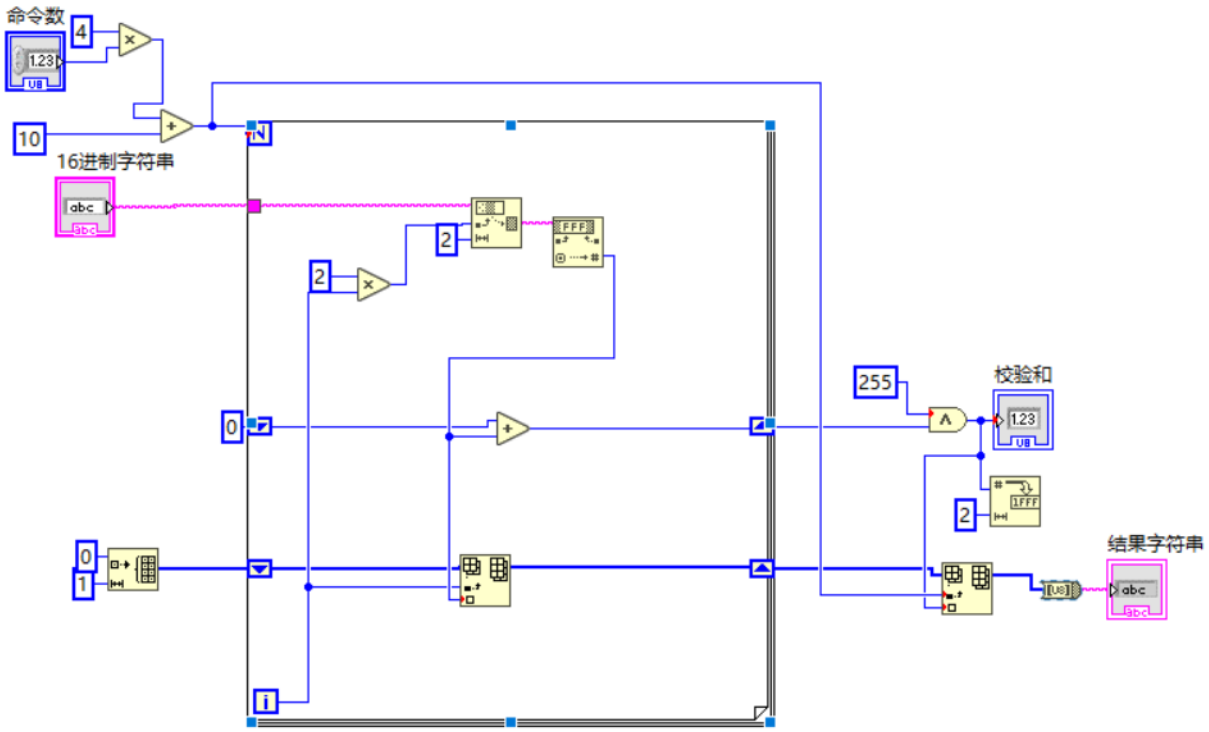


图 4-16 数据打包子 VI 后面板

5 系统调试

5.1 硬件调试

5.1.1 硬件连接

将单片机最小系统、L298N 电机驱动模块、电机及电源按第二章 2.2 节下位机硬件设计中的各模块连接表用线连接起来，连接后如图 5-1 所示：

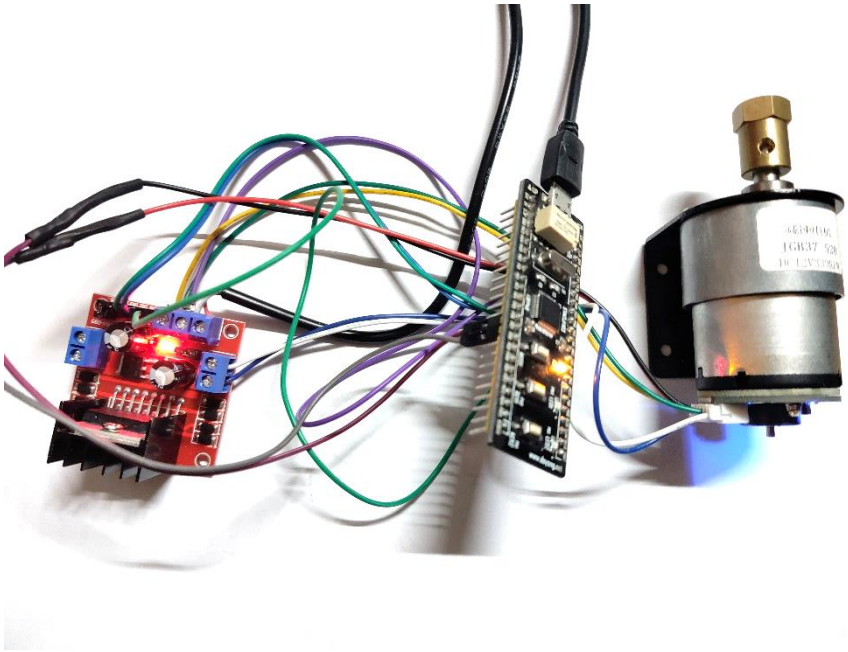


图 5-1 下位机系统实物图

5.1.2 程序编译下载

用 Keil 软件编写下位机 STM32 程序，编译没有错误后用串口下载软件将编译生成的.hex 文件下载到 STM32 单片机。

5.2 联合调试

通过 USB 数据线将 STM32 最小系统板与电脑相连，打开 LabVIEW 主 VI 并运行，端口选择 COM5、波特率选择 115200、奇偶校验选择无、数据为选 8 位、停止位 1 位。点击打开串口按钮即可同下位机通信。经过测试上下位机之间的通信均正常，上位机可以完整的解析下位机发送来的数据，下位机也可正确的执行上位机发送来的控制指令，整个系统可正常运行。如图 5-2 所示。系统设置为位置控制模式，目标值设置为 5000，发

送目标值后，电机开始转动最终实际值达到 4997 电机停止。

改变 PID 参数点击发送 PID，重新设置目标值点击发送目标值。结果如图 5-3 所示。

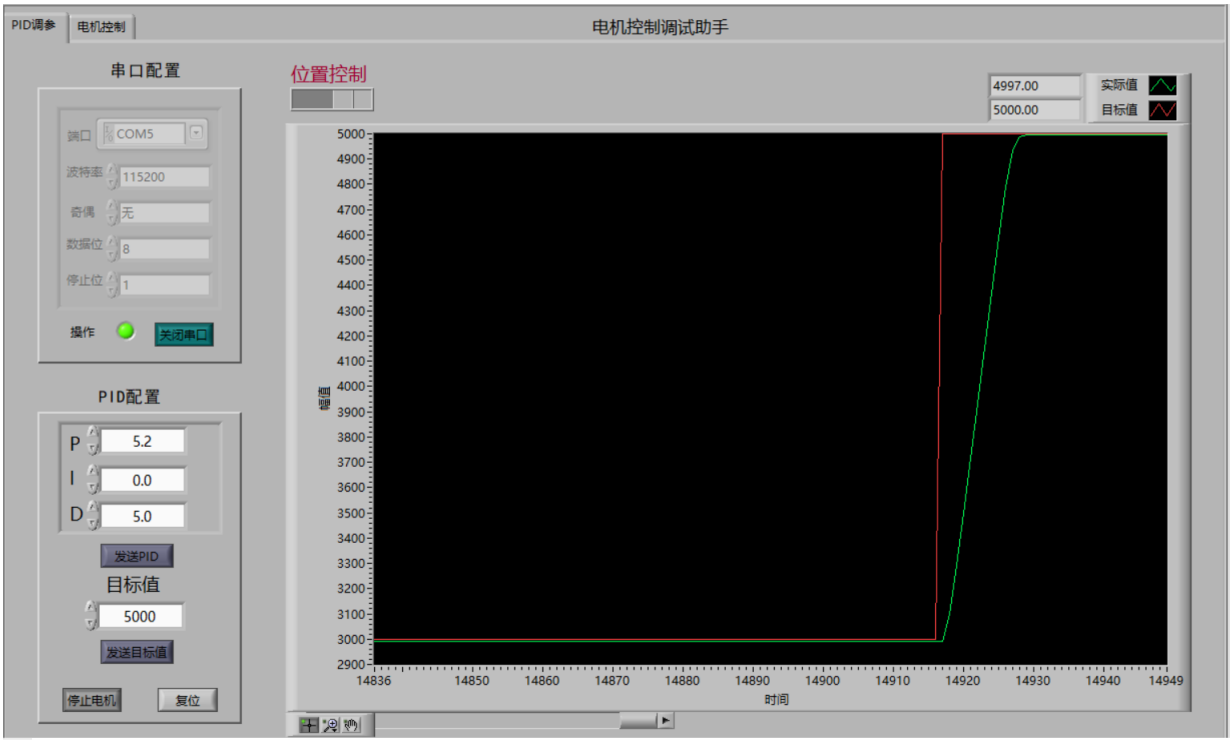


图 5-2 PID 调节面板运行效果图（1）

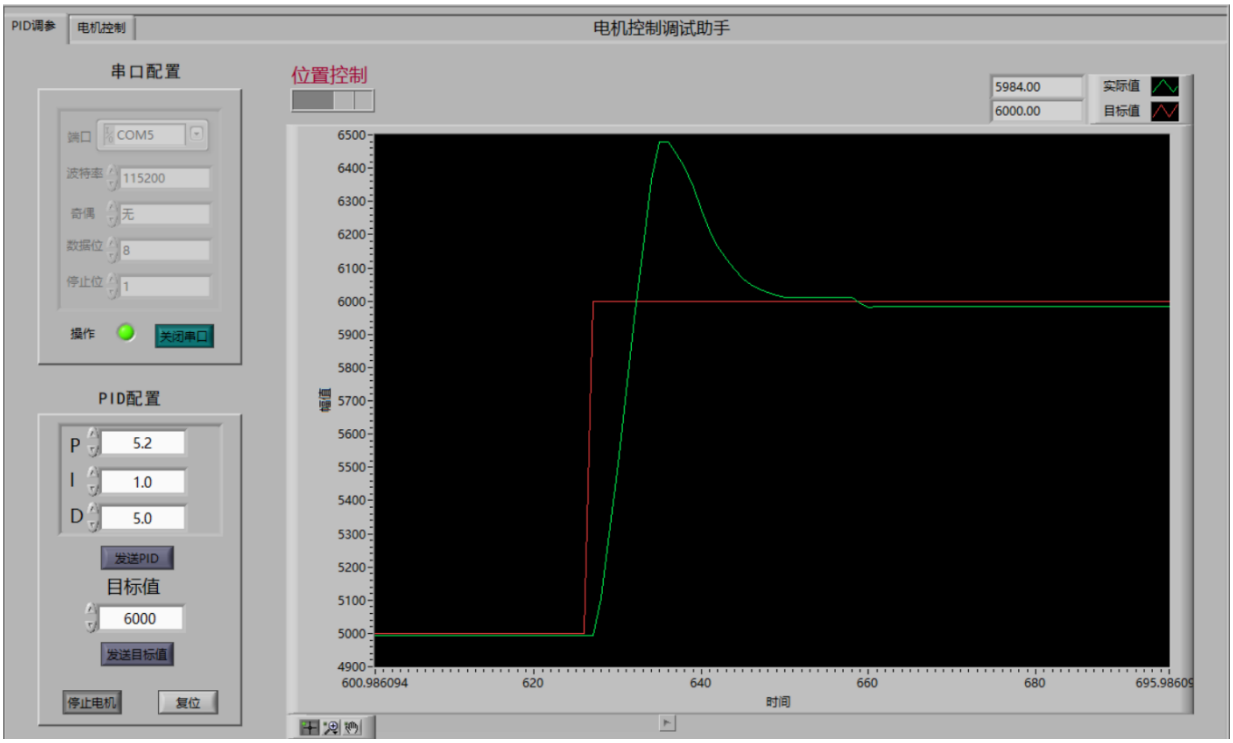


图 5-3 PID 调节面板运行效果图（2）

PID 参数调节后输出曲线发生了变化，新的 PID 参数已经起了作用，比较两幅图可以知道第二组 PID 参数的控制效果并不如第一组 PID 参数。

点击电机控制选项卡切换到电机控制界面，分别对电机速度和位置进行控制，电机速度和位置控制响应很快，控制精度较高。

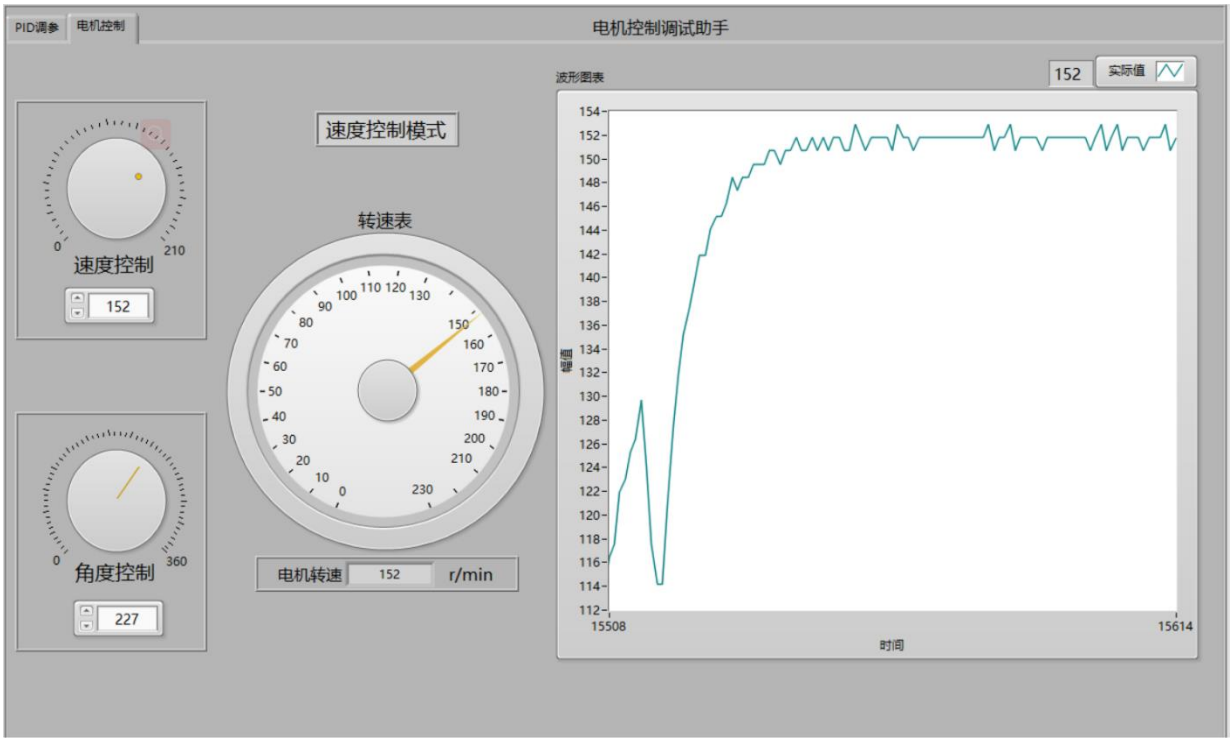


图 5-4 电机控制界面速度控制运行效果图

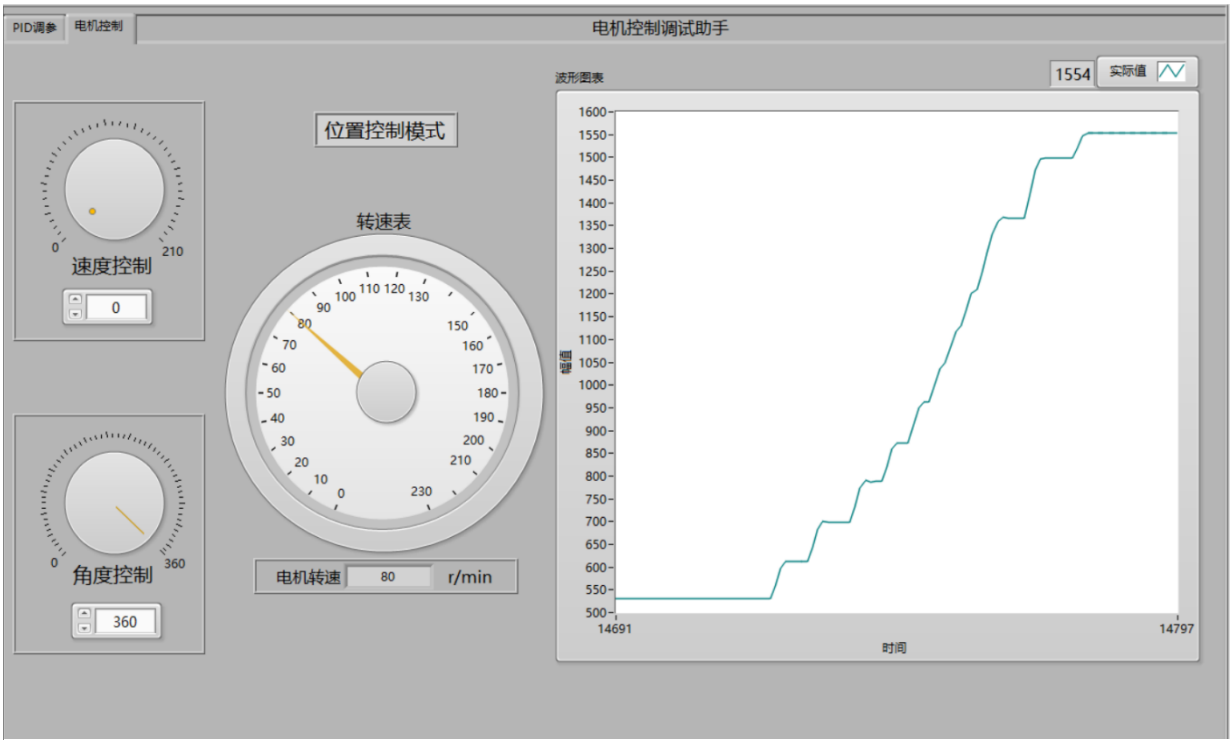


图 5-5 电机控制界面位置控制效果图

6 结论与展望

此次设计基于 LabVIEW 开发的直流有刷电机 PID 调试和控制助手，解决了单片机上 PID 参数调节不方便问题，运用 LabVIEW 编写的上位机可以方便的对下位机中 PID 控制程序中的参数进行动态的调整，同时也可对电机进行转速和位置的控制。经过调试验证此系统达到了系统设计的要求。

1) 电机转速控制范围宽：0~210rpm。
2) 电机转速控制精度满足$\pm 1rpm$的设计精度要求。
3) 拥有 0~100 的 PID 参数调试范围，PID 调试方便，效果好。
4) 电机位置控制模式下有 0~360° 的控制范围。

此次设计的系统上、下位机相结合，这对我小型测控系统的设计能力有很大的提升，在系统的设计过程中遇到了比较多的问题，在不断的查找问题解决问题的过程中不断的提高自己的动手实践能力。由于本人的水平有限和设计的条件有限，设计中尚存在许多未尽人意的地方。主要有以下几个方面需要改进和研究：

1) 波形显示控件未作优化，在一些情况下波形的显示效果不尽人意，会出现波形卡顿、波形显示位置不便观察的问题。
2) 电机控制界面功能比较单一，可以加入方向控制功能，紧急制动等功能等。

PC 技术与嵌入式系统融合发展，虚拟仪器的功能得到进一步的发展，例如更多的嵌入式和实时功能。随着 PC 技术和相关技术的发展，虚拟仪器技术已成为一项前沿学科，代表着仪器发展的最新方向之一，不断的被推向各个新的领域，在新的世纪将大行其道。虚拟仪器设计已经成为测试与仪器技术发展的一个重要方向。未来虚拟仪器技术的应用会越来越广泛。

7 参考文献

[1]王海星.基于 LabVIEW 和 STM32 的直流电机闭环调速系统设计[J].化工自动化及仪表,2019,46(04):310-312+315.

[2]施尚英.基于 LabVIEW 数字 PID 直流电机调速系统的实现[J].四川职业技术学院学报,2016,26(04):174-176.

[3]李海春,蔡燕,姜文涛.基于 LabVIEW 的电机转速监测系统的设计[J].电子科技,2012,25(12):80-82+85.

[4]林若波.基于 LabVIEW 的电机调速数据采集与处理系统[J].测控技术,2012,31(07):16-19.

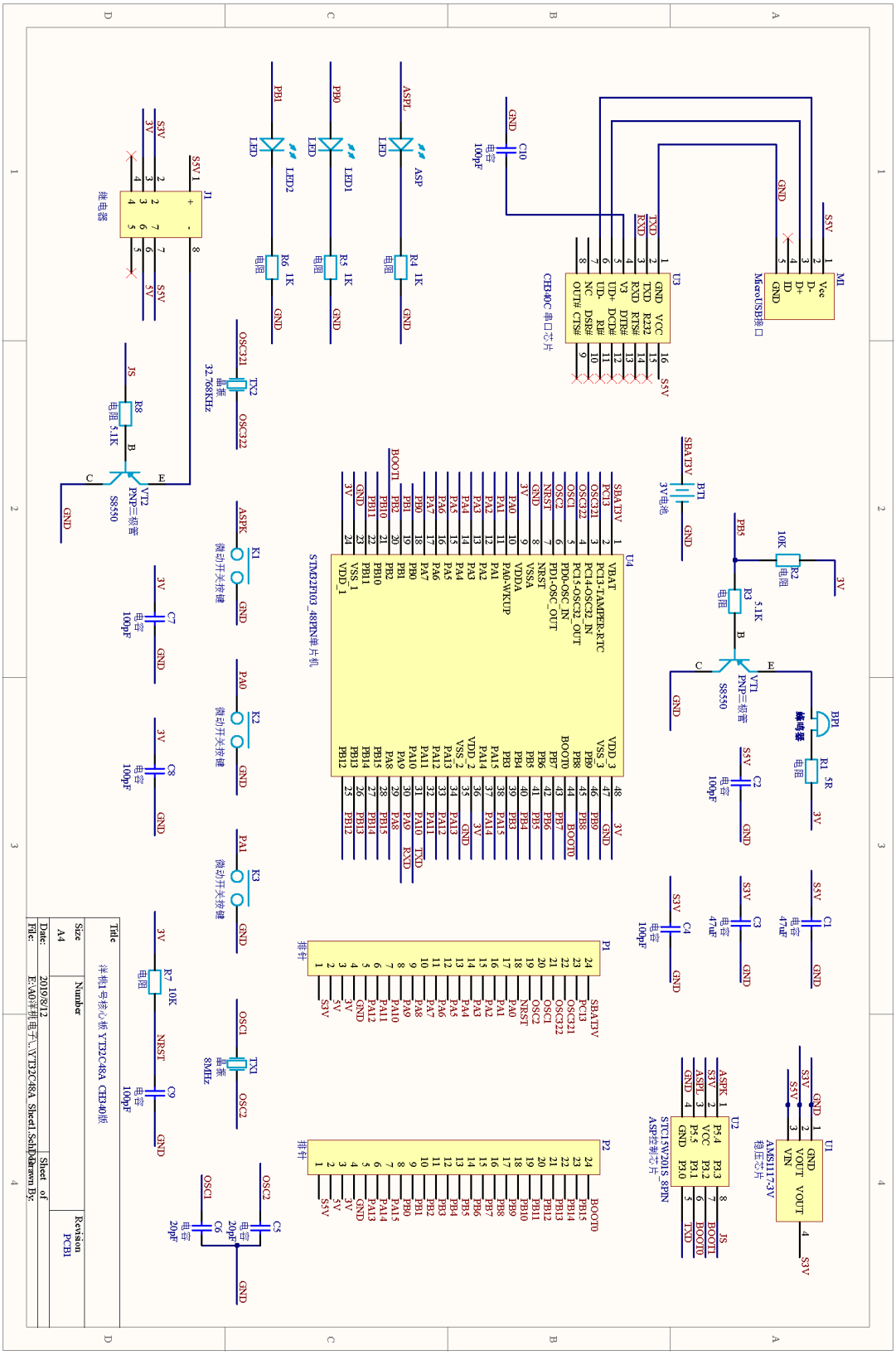
[5]林若波,彭燕标,廖兴展.基于 LabVIEW 的直流电机测速与控制系统设计[J].云南民族大学学报(自然科学版),2012,21(03):231-234.

[6]杨春旭,林若波,彭燕标.基于 PWM 控制的直流电机调速系统的设计[J].齐齐哈尔大学学报(自然科学版),2011,27(03):10-13.

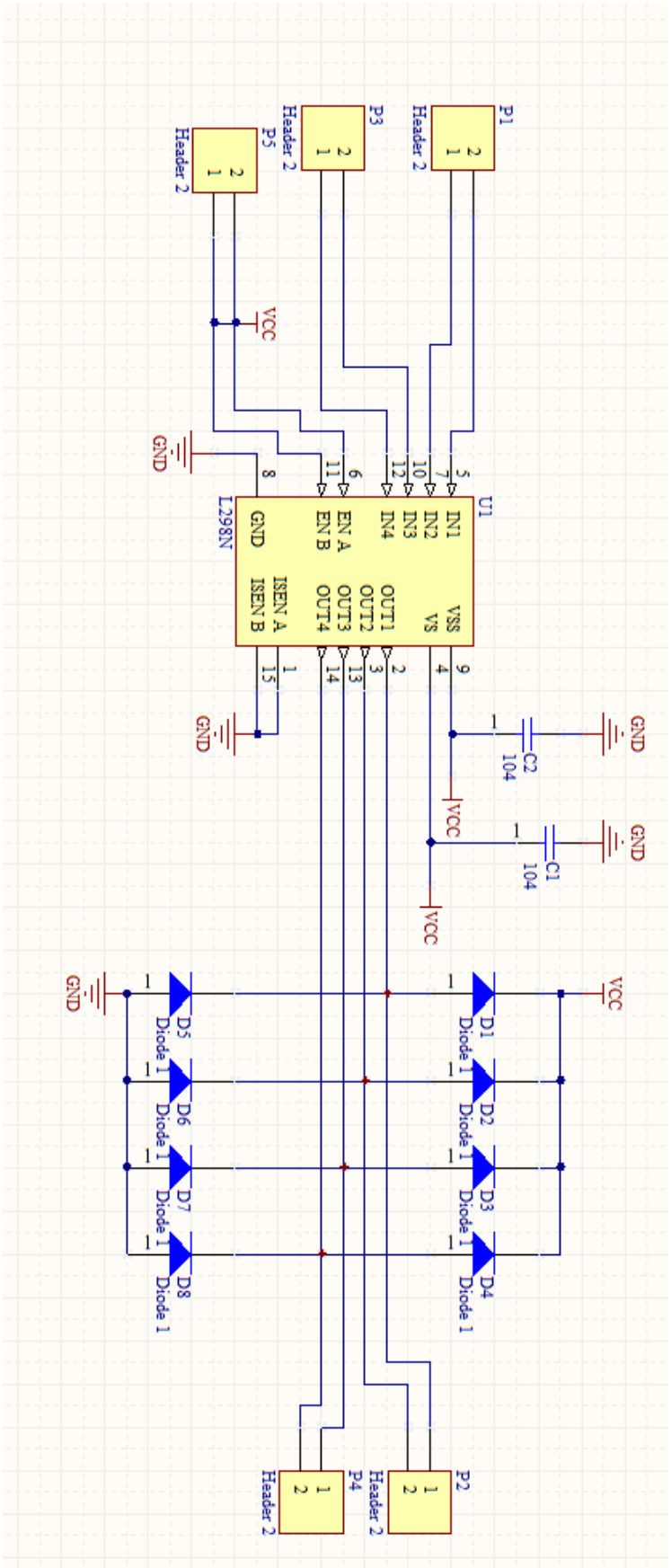
[7]王颖,章蔚中.基于 LabVIEW 的电机测控系统设计[J].微计算机信息,2008,(28):114-115.

8 附录

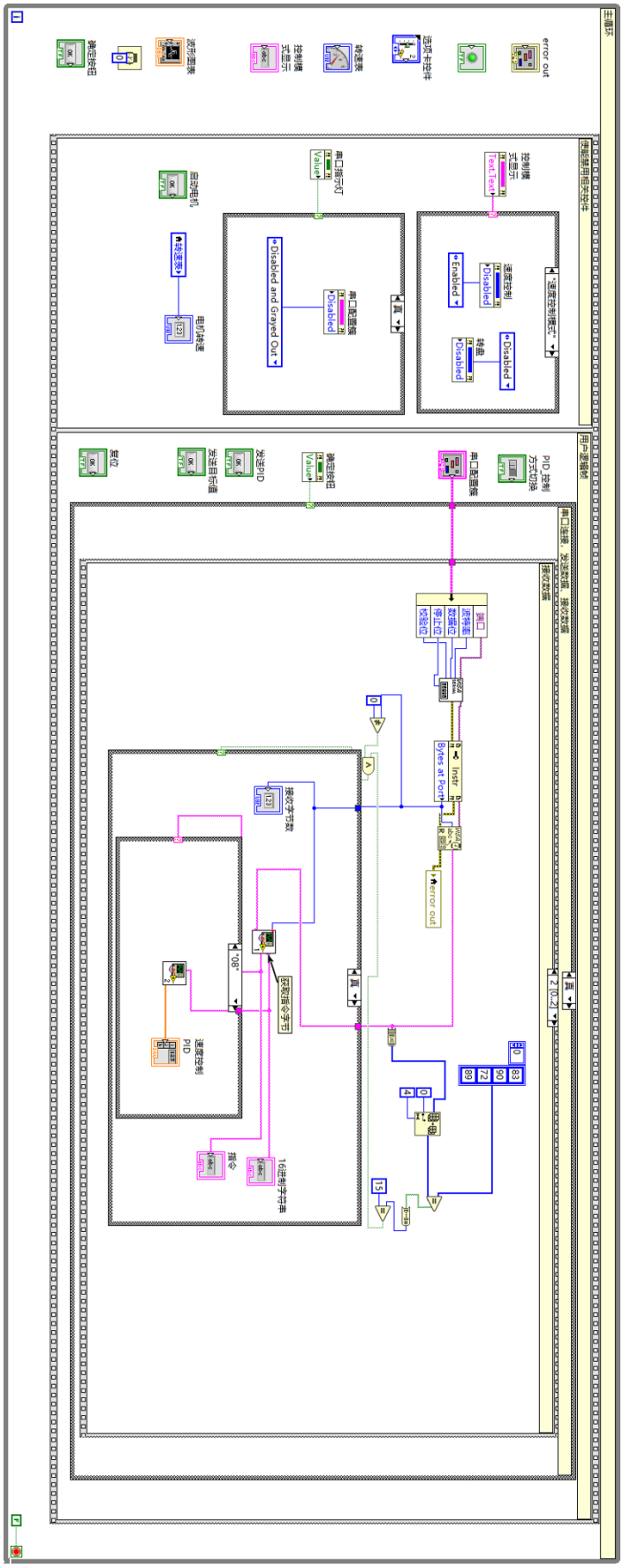
附录一：单片机最小系统板电路图



附录二：L298N 模块电路图



附录三：LabVIEW 主 VI



附录四：下位机程序

main.c

```
/**
  * @brief  主函数
  * @param  无
  * @retval 无
  */
int main(void)
{
    int32_t target_location = CIRCLE_PULSES;
    /* HAL 库初始化*/
    HAL_Init();
    /* 初始化系统时钟为 72MHz */
    SystemClock_Config();
    /* 开启复用寄存器时钟 */
    __HAL_RCC_SYSCFG_CLK_ENABLE();
    /* 初始化按键 GPIO */
    Key_GPIO_Config();
    /* 初始化 LED */
    //LED_GPIO_Config();
    /* 协议初始化 */
    protocol_init();
    /* 初始化串口 */
    DEBUG_USART_Config();
    /* 电机初始化 */
    motor_init();
    /* 编码器接口初始化 */
    Encoder_Init();
    /* 初始化基本定时器，用于处理定时任务 */
    TIMx_Configuration();
    /* PID 参数初始化 */
    PID_param_init();
#if 1
    uint32_t i;
    for(i=100000;i>0;i--);
    set_computer_value(SEND_STOP_CMD, CURVES_CH1, NULL, 0);     // 同步上位机的启
动按钮状态
    for(i=100000;i>0;i--);
    set_computer_value(SEND_TARGET_CMD, CURVES_CH1, &target_location, 1);       // 给
通道 1 发送目标值
    #endif
    while(1)
    {
        #if 0
        uint32_t i;
        for(i=10000000;i>0;i--);
        set_computer_value(SEND_STOP_CMD, CURVES_CH1, NULL, 0);     // 同步上位机
的启动按钮状态
        #endif
        /* 接收数据处理 */
        receiving_process();
        #if 1
        /* 扫描 KEY1 */
        if( Key_Scan(KEY1_GPIO_PORT, KEY1_PIN) == KEY_ON)
        {
        #if 1
          set_computer_value(SEND_START_CMD,      CURVES_CH1,      NULL,      0);
```

```
// 同步上位机的启动按钮状态
        #endif
          set_pid_target(target_location);      // 设置目标值
          set_motor_enable();                   // 使能电机
        }

        /* 扫描 KEY2 */
        if( Key_Scan(KEY2_GPIO_PORT, KEY2_PIN) == KEY_ON)
        {
          set_motor_disable();       // 停止电机
          set_computer_value(SEND_STOP_CMD,       CURVES_CH1,       NULL,       0);
// 同步上位机的启动按钮状态
        }
        #endif
    }
}

/**
  * @brief   System Clock Configuration
  *          The system Clock is configured as follow :
  *              System Clock source                  = PLL (HSE)
  *              SYSCLK(Hz)                           = 72000000
  *              HCLK(Hz)                             = 72000000
  *              AHB Prescaler                        = 1
  *              APB1 Prescaler                       = 2
  *              APB2 Prescaler                       = 1
  *              HSE Frequency(Hz)                    = 8000000
  *              HSE PREDIV1                          = 2
  *              PLLMUL                               = 9
  *              Flash Latency(WS)                    = 0
  * @param   None
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_ClkInitTypeDef clkinitstruct = {0};
  RCC_OscInitTypeDef oscinitstruct = {0};
  RCC_PeriphCLKInitTypeDef   PeriphClkInit = {0}
  /* Enable HSE Oscillator and activate PLL with HSE as source */
  oscinitstruct.OscillatorType   = RCC_OSCILLATORTYPE_HSE;
  oscinitstruct.HSEState         = RCC_HSE_ON;
  oscinitstruct.HSEPredivValue   = RCC_HSE_PREDIV_DIV1;
  oscinitstruct.PLL.PLLState     = RCC_PLL_ON;
  oscinitstruct.PLL.PLLSource    = RCC_PLLSOURCE_HSE;
  oscinitstruct.PLL.PLLMUL       = RCC_PLL_MUL9;
  if (HAL_RCC_OscConfig(&oscinitstruct)!= HAL_OK)
  {
    /* Initialization Error */
    while(1);
  }
  /* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2
     clocks dividers */
  clkinitstruct.ClockType   =   (RCC_CLOCKTYPE_SYSCLK   |   RCC_CLOCKTYPE_HCLK   |
RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
  clkinitstruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  clkinitstruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  clkinitstruct.APB2CLKDivider = RCC_HCLK_DIV1;
  clkinitstruct.APB1CLKDivider = RCC_HCLK_DIV2;
```

```
  if (HAL_RCC_ClockConfig(&clkinitstruct, FLASH_LATENCY_2)!= HAL_OK)
  {
    /* Initialization Error */
    while(1);
  }
    PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_ADC;
    PeriphClkInit.AdcClockSelection = RCC_ADCPCLK2_DIV6;
    if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK )
    {
    /* Initialization Error */
    while(1);
  }
}
```

stm32f1xx_it.c

```
/**
  * @brief   This function handles SysTick Handler.
  * @param   None
  * @retval None
  */
void SysTick_Handler(void)
{
    HAL_IncTick();
  HAL_SYSTICK_Callback();
}
/**
  * @brief 编码器接口中断服务函数
  */
void ENCODER_TIM_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&TIM_EncoderHandle);
}
/**
  * @brief   基本定时器中断服务函数----PID 计算使用
  * @param   None
  * @retval None
  */
void BASIC_TIM_IRQHandler (void)
{
    HAL_TIM_IRQHandler(&TIM_TimeBaseStructure);
}

uint8_t dr;
volatile uint8_t sr_status;
/**
  * @brief 串口中断服务函数
  */
void DEBUG_USART_IRQHandler(void)
{
    sr_status = UartHandle.Instance->SR & (1<<3);//clear SR register ORE bit status
    dr = UartHandle.Instance->DR;
    protocol_data_recv(&dr, 1);
    HAL_UART_IRQHandler(&UartHandle);
}

/**
  * @brief   编码器定时器更新事件回调函数
  * @param   无
  * @retval  无
```

```
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if(htim==(&TIM_EncoderHandle))
  {
    /* 判断当前计数器计数方向 */
    if(__HAL_TIM_IS_TIM_COUNTING_DOWN(&TIM_EncoderHandle))
      /* 下溢 */
      Encoder_Overflow_Count--;
    else
      /* 上溢 */
      Encoder_Overflow_Count++;
  }
  else if(htim==(&TIM_TimeBaseStructure))
  {
    #if 0
    int32_t Capture_Count = 0;       // 当前时刻总计数值
    /* 当前时刻总计数值 = 计数器值 + 计数溢出次数 * ENCODER_TIM_PERIOD */
    Capture_Count = __HAL_TIM_GET_COUNTER(&TIM_EncoderHandle) +
(Encoder_Overflow_Count * ENCODER_TIM_PERIOD);
    printf("Capture_Count=%d\n",Capture_Count);
    #endif
    motor_pid_control();
  }
}
```

bsp_debug_usart.h

```
//串口波特率
#define DEBUG_USART_BAUDRATE                    115200

//引脚定义
/************************************************/
#define DEBUG_USART                             USART1
#define DEBUG_USART_CLK_ENABLE()                __HAL_RCC_USART1_CLK_ENABLE();

#define RCC_PERIPHCLK_UARTx                     RCC_PERIPHCLK_USART1
#define RCC_UARTxCLKSOURCE_SYSCLK               RCC_USART1CLKSOURCE_SYSCLK

#define DEBUG_USART_RX_GPIO_PORT                GPIOA
#define DEBUG_USART_RX_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define DEBUG_USART_RX_PIN                      GPIO_PIN_10

#define DEBUG_USART_TX_GPIO_PORT                GPIOA
#define DEBUG_USART_TX_GPIO_CLK_ENABLE()        __HAL_RCC_GPIOA_CLK_ENABLE()
#define DEBUG_USART_TX_PIN                      GPIO_PIN_9

#define DEBUG_USART_IRQHandler                  USART1_IRQHandler
#define DEBUG_USART_IRQ                         USART1_IRQn
```

bsp_debug_usart.c

```
UART_HandleTypeDef UartHandle;

 /**
  * @brief  DEBUG_USART GPIO 配置,工作模式配置。115200 8-N-1
  * @param  无
  * @retval 无
  */
void DEBUG_USART_Config(void)
{
```

```
  UartHandle.Instance                 = DEBUG_USART;

  UartHandle.Init.BaudRate         = DEBUG_USART_BAUDRATE;
  UartHandle.Init.WordLength     = UART_WORDLENGTH_8B;
  UartHandle.Init.StopBits         = UART_STOPBITS_1;
  UartHandle.Init.Parity             = UART_PARITY_NONE;
  UartHandle.Init.HwFlowCtl       = UART_HWCONTROL_NONE;
  UartHandle.Init.Mode               = UART_MODE_TX_RX;

  HAL_UART_Init(&UartHandle);

 /*使能串口接收断 */
  __HAL_UART_ENABLE_IT(&UartHandle,UART_IT_RXNE);
}


/**
  * @brief UART MSP 初始化
  * @param huart: UART handle
  * @retval 无
  */
void HAL_UART_MspInit(UART_HandleTypeDef *huart)
{
    GPIO_InitTypeDef    GPIO_InitStruct;

    DEBUG_USART_CLK_ENABLE();
    //DEBUG_USART_AF_ENABLE();
    DEBUG_USART_RX_GPIO_CLK_ENABLE();
    DEBUG_USART_TX_GPIO_CLK_ENABLE();
    //__HAL_RCC_AFIO_CLK_ENABLE();

    /* 配置 Tx 引脚  */
    GPIO_InitStruct.Pin = DEBUG_USART_TX_PIN;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed =    GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(DEBUG_USART_TX_GPIO_PORT, &GPIO_InitStruct);

    /* 配置 Rx 引脚为复用功能 */
    GPIO_InitStruct.Pin = DEBUG_USART_RX_PIN;
    GPIO_InitStruct.Mode=GPIO_MODE_AF_INPUT;     //模式要设置为复用输入模式!
    HAL_GPIO_Init(DEBUG_USART_RX_GPIO_PORT, &GPIO_InitStruct);

    HAL_NVIC_SetPriority(DEBUG_USART_IRQ ,0,1);    //抢占优先级 0，子优先级 1
    HAL_NVIC_EnableIRQ(DEBUG_USART_IRQ );                //使能 USART1 中断通道
    }

/*****************  发送字符 **********************/
void Usart_SendByte(uint8_t str)
{
  HAL_UART_Transmit(&UartHandle, &str, 1, 1000);
}

/*****************  发送字符串 **********************/
void Usart_SendString(uint8_t *str)
{
    unsigned int k=0;
```

```
   do
   {
        HAL_UART_Transmit(&UartHandle,(uint8_t *)(str + k) ,1,1000);
        k++;
   } while(*(str + k)!='\0');

}
//重定向 c 库函数 printf 到串口 DEBUG_USART，重定向后可使用 printf 函数
int fputc(int ch, FILE *f)
{
    /* 发送一个字节数据到串口 DEBUG_USART */
    HAL_UART_Transmit(&UartHandle, (uint8_t *)&ch, 1, 1000);

    return (ch);
}

//重定向 c 库函数 scanf 到串口 DEBUG_USART，重写向后可使用 scanf、getchar 等函数
int fgetc(FILE *f)
{
    int ch;
    HAL_UART_Receive(&UartHandle, (uint8_t *)&ch, 1, 1000);
    return (ch);
}
```

bsp_motor_control.h

```
//引脚定义
/*******************************************************/
// 连接 MOS 管搭建板的 SD 脚，或者连接 L298N 板的 EN 脚
#define SHUTDOWN_PIN                    GPIO_PIN_2
#define SHUTDOWN_GPIO_PORT               GPIOA
#define SHUTDOWN_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOA_CLK_ENABLE()
/*******************************************************/

/* 电机 SD or EN 使能脚 */
#define                                                 MOTOR_ENABLE_SD()
HAL_GPIO_WritePin(SHUTDOWN_GPIO_PORT, SHUTDOWN_PIN, GPIO_PIN_SET)       // 高电平打开-高电平使能
#define                                                 MOTOR_DISABLE_SD()
HAL_GPIO_WritePin(SHUTDOWN_GPIO_PORT, SHUTDOWN_PIN, GPIO_PIN_RESET)     // 低电平关断-低电平禁用

/* 电机方向控制枚举 */
typedef enum
{
  MOTOR_FWD = 0,
  MOTOR_REV,
}motor_dir_t;

#define CIRCLE_PULSES      (ENCODER_TOTAL_RESOLUTION * REDUCTION_RATIO)     // 编码器一圈可以捕获的脉冲

/* 设置速度（占空比） */
#define                                             SET_FWD_COMPAER(ChannelPulse)
TIM1_SetPWM_pulse(PWM_CHANNEL_1,ChannelPulse)     // 设置比较寄存器的值
#define                                             SET_REV_COMPAER(ChannelPulse)
TIM1_SetPWM_pulse(PWM_CHANNEL_2,ChannelPulse)     // 设置比较寄存器的值

/* 使能输出 */
#define                                                 MOTOR_FWD_ENABLE()
```

```
HAL_TIM_PWM_Start(&DCM_TimeBaseStructure,PWM_CHANNEL_1);
#define                                                    MOTOR_REV_ENABLE()
HAL_TIM_PWM_Start(&DCM_TimeBaseStructure,PWM_CHANNEL_2);

/* 禁用输出 */
#define                                                  MOTOR_FWD_DISABLE()
HAL_TIM_PWM_Stop(&DCM_TimeBaseStructure,PWM_CHANNEL_1);
#define                                                   MOTOR_REV_DISABLE()
HAL_TIM_PWM_Stop(&DCM_TimeBaseStructure,PWM_CHANNEL_2);
```

bsp_motor_control.c

```
static motor_dir_t direction    = MOTOR_FWD;        // 记录方向
static uint16_t       dutyfactor = 0;                    // 记录占空比
static uint8_t       is_motor_en = 0;                    // 电机使能
extern _pid;
static void sd_gpio_config(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* 定时器通道功能引脚端口时钟使能 */
    SHUTDOWN_GPIO_CLK_ENABLE();

    /* 引脚 IO 初始化 */
    /*设置输出类型*/
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    /*设置引脚速率 */
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    /*选择要控制的 GPIO 引脚*/
    GPIO_InitStruct.Pin = SHUTDOWN_PIN;

    /*调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
    HAL_GPIO_Init(SHUTDOWN_GPIO_PORT, &GPIO_InitStruct);
}

/**
  * @brief   电机初始化
  * @param   无
  * @retval 无
  */
void motor_init(void)
{
  Motor_TIMx_Configuration();        // 初始化电机 1
  sd_gpio_config();
}

/**
  * @brief   设置电机速度
  * @param   v: 速度（占空比）
  * @retval 无
  */
void set_motor_speed(uint16_t v)
{
  v = (v > PWM_PERIOD_COUNT) ? PWM_PERIOD_COUNT : v;        // 上限处理

  dutyfactor = v;

  if (direction == MOTOR_FWD)
  {
```

```
        SET_FWD_COMPAER(dutyfactor);        // 设置速度
    }
    else
    {
        SET_REV_COMPAER(dutyfactor);        // 设置速度
    }
}

/**
  * @brief   设置电机方向
  * @param   无
  * @retval 无
  */
void set_motor_direction(motor_dir_t dir)
{
    direction = dir;

    if (direction == MOTOR_FWD)   //正传
    {
        SET_FWD_COMPAER(dutyfactor);        // 设置速度
        SET_REV_COMPAER(0);                 // 设置速度
    }
    else
    {
        SET_FWD_COMPAER(0);                 // 设置速度
        SET_REV_COMPAER(dutyfactor);        // 设置速度
    }
}

/**
  * @brief   使能电机
  * @param   无
  * @retval 无
  */
void set_motor_enable(void)
{
    is_motor_en = 1;
    MOTOR_ENABLE_SD();
    MOTOR_FWD_ENABLE();
    MOTOR_REV_ENABLE();
}

/**
  * @brief   禁用电机
  * @param   无
  * @retval 无
  */
void set_motor_disable(void)
{
    is_motor_en = 0;
    MOTOR_DISABLE_SD();
    MOTOR_FWD_DISABLE();
    MOTOR_REV_DISABLE();
}

/**
  * @brief   电机位置式 PID 控制实现(定时调用)
  * @param   无
```

```
  * @retval 无
  */
void motor_pid_control(void)
{
  if (is_motor_en == 1)        // 电机在使能状态下才进行控制处理
  {
    float cont_val = 0;                  // 当前控制值
    static __IO int32_t Capture_Count = 0;       // 当前时刻总计数值
    static __IO int32_t Last_Count = 0;          // 上一时刻总计数值
    int32_t actual_speed = 0;                    // 实际测得速度
    pid_realize pid_fun;

    /* 当前时刻总计数值 = 计数器值 + 计数溢出次数 * ENCODER_TIM_PERIOD */
    Capture_Count = __HAL_TIM_GET_COUNTER(&TIM_EncoderHandle) +
(Encoder_Overflow_Count * ENCODER_TIM_PERIOD);
    /* 转轴转速 = 单位时间内的计数值 / 编码器总分辨率 * 时间系数 */
    actual_speed = ((float)(Capture_Count - Last_Count) / ENCODER_TOTAL_RESOLUTION /
REDUCTION_RATIO) / (GET_BASIC_TIM_PERIOD()/1000.0/60.0);
    /* 记录当前总计数值，供下一时刻计算使用 */
    Last_Count = Capture_Count;
    if(pid.Mode==0x55)
    {
        pid_fun.PID_realize = W_PID_realize;
        cont_val = PID_realize(Capture_Count,pid_fun.PID_realize);     // 进行 PID 计算
        #if 1
            set_computer_value(SEND_FACT_CMD, CURVES_CH1, &Capture_Count, 1);
// 给通道 1 发送实际值
        #endif
    }
    else if(pid.Mode==0xAA)
    {
        pid_fun.PID_realize = S_PID_realize;
        cont_val = PID_realize(actual_speed,pid_fun.PID_realize);     // 进行 PID 计算
        #if 1
            set_computer_value(SEND_FACT_CMD, CURVES_CH1, &actual_speed,1);
// 给通道 1 发送实际值
        #endif
    }
     if (cont_val > 0)      // 判断电机方向
     {
       set_motor_direction(MOTOR_FWD);  //正传
     }
     else
     {
       cont_val = -cont_val;
       set_motor_direction(MOTOR_REV);
     }
    if(pid.Mode==0x55)
    {
        cont_val = (cont_val > PWM_MAX_PERIOD_COUNT*0.48) ?
PWM_MAX_PERIOD_COUNT*0.48 : cont_val;     // 速度上限处理
    }
    else if(pid.Mode==0xAA)
    {
        cont_val = (cont_val > PWM_MAX_PERIOD_COUNT) ? PWM_MAX_PERIOD_COUNT :
cont_val;     // 速度上限处理
    }
    set_motor_speed(cont_val);                                  // 设置 PWM 占空比
```

```
  }
}
```

bsp_basic_tim.h

```
#define BASIC_TIM                   TIM3
#define BASIC_TIM_CLK_ENABLE()      __HAL_RCC_TIM3_CLK_ENABLE()
#define BASIC_TIM_IRQn              TIM3_IRQn
#define BASIC_TIM_IRQHandler        TIM3_IRQHandler
/* 累计 TIM_Period 个后产生一个更新或者中断*/
//当定时器从 0 计数到 BASIC_PERIOD_COUNT-1，即为 BASIC_PERIOD_COUNT 次，为一个定时周期
//PID 计算周期:50ms 计算一次--频率 20Hz
#define BASIC_PERIOD_MS      (100)//50ms
#define BASIC_PERIOD_COUNT      (BASIC_PERIOD_MS*100)
#define BASIC_PRESCALER_COUNT    (720)//100Khz
/* 获取定时器的周期，单位 ms */
/* 以下两宏仅适用于定时器时钟源 TIMxCLK=72MHz，预分频器为：720-1 的情况 */
#define                                     SET_BASIC_TIM_PERIOD(T)
__HAL_TIM_SET_AUTORELOAD(&TIM_TimeBaseStructure, (T)*100 - 1)     // 设置定时器的周期（1~1000ms）
#define                                     GET_BASIC_TIM_PERIOD()
((__HAL_TIM_GET_AUTORELOAD(&TIM_TimeBaseStructure)+1)/100.0f)       // 获取定时器的周期，单位 ms
```

bsp_basic_tim.c

```
TIM_HandleTypeDef TIM_TimeBaseStructure;
 /**
  * @brief   基本定时器 TIMx,x[6,7]中断优先级配置
  * @param   无
  * @retval 无
  */
static void TIMx_NVIC_Configuration(void)
{
    //设置抢占优先级，子优先级
    HAL_NVIC_SetPriority(BASIC_TIM_IRQn, 1,1);
    // 设置中断来源
    HAL_NVIC_EnableIRQ(BASIC_TIM_IRQn);
}
/*
 * 注意：TIM_TimeBaseInitTypeDef 结构体里面有 5 个成员，TIM6 和 TIM7 的寄存器里面只有
 * TIM_Prescaler 和 TIM_Period，所以使用 TIM6 和 TIM7 的时候只需初始化这两个成员即可，
 * 另外三个成员是通用定时器和高级定时器才有.
 *-----------------------------------------------------------------------------
 * TIM_Prescaler                都有
 * TIM_CounterMode                TIMx,x[6,7]没有，其他都有（基本定时器）
 * TIM_Period                   都有
 * TIM_ClockDivision        TIMx,x[6,7]没有，其他都有(基本定时器)
 * TIM_RepetitionCounter TIMx,x[1,8]才有(高级定时器)
 *-----------------------------------------------------------------------------
 */
static void TIM_Mode_Config(void)
{
    // 开启 TIMx_CLK,x[6,7]
    BASIC_TIM_CLK_ENABLE();
    TIM_TimeBaseStructure.Instance = BASIC_TIM;
    /* 累计 TIM_Period 个后产生一个更新或者中断*/
    //当定时器从 0 计数到 4999，即为 5000 次，为一个定时周期
```

```
    TIM_TimeBaseStructure.Init.Period = BASIC_PERIOD_COUNT - 1;
    //定时器时钟源 TIMxCLK = 2 * PCLK1
    //                    PCLK1 = HCLK / 2
    //                    => TIMxCLK=HCLK/2=SystemCoreClock/2*2=72MHz
    // 设定定时器频率为=TIMxCLK/(TIM_Prescaler+1)=100KHz
    TIM_TimeBaseStructure.Init.Prescaler = BASIC_PRESCALER_COUNT - 1;
    TIM_TimeBaseStructure.Init.CounterMode = TIM_COUNTERMODE_UP;                    //
向上计数
    TIM_TimeBaseStructure.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;        // 时钟分
频
    // 初始化定时器 TIMx, x[2,3,4,5]
    HAL_TIM_Base_Init(&TIM_TimeBaseStructure);

    // 开启定时器更新中断
    HAL_TIM_Base_Start_IT(&TIM_TimeBaseStructure);
}

/**
  * @brief  初始化基本定时器定时，1ms 产生一次中断
  * @param  无
  * @retval 无
  */
void TIMx_Configuration(void)
{
    TIMx_NVIC_Configuration();
    TIM_Mode_Config();
#if PID_ASSISTANT_EN
  uint32_t temp = GET_BASIC_TIM_PERIOD();        // 计算周期，单位 ms
  set_computer_value(SEND_PERIOD_CMD, CURVES_CH1, &temp, 1);        // 给通道 1 发
送目标值
#endif

}
```

bsp_motor_tim.h

```
/*宏定义*/
#define PWM_TIM                                    TIM1
#define PWM_TIM_GPIO_AF_ENALBE()            __HAL_AFIO_REMAP_TIM1_ENABLE();
#define PWM_TIM_CLK_ENABLE()                __HAL_RCC_TIM1_CLK_ENABLE()
#define PWM_CHANNEL_1                       TIM_CHANNEL_1
#define PWM_CHANNEL_2                       TIM_CHANNEL_4
/* 累计 TIM_Period 个后产生一个更新或者中断*/
/* 当定时器从 0 计数到 PWM_PERIOD_COUNT，即为 PWM_PERIOD_COUNT+1 次，为一个
定时周期 */
#define PWM_PERIOD_COUNT        (3600)
/* 通用控制定时器时钟源 TIMxCLK = HCLK=72MHz */
/* 设定定时器频率为=TIMxCLK/(PWM_PRESCALER_COUNT)/PWM_PERIOD_COUNT = 10khz
*/
#define PWM_PRESCALER_COUNT        (2)
/* 最大比较值 */
#define PWM_MAX_PERIOD_COUNT                (PWM_PERIOD_COUNT - 100)
/*PWM 引脚*/
#define PWM_TIM_CH1_GPIO_CLK()              __HAL_RCC_GPIOA_CLK_ENABLE();
#define PWM_TIM_CH1_GPIO_PORT               GPIOA
#define PWM_TIM_CH1_PIN                     GPIO_PIN_8
#define PWM_TIM_CH2_GPIO_CLK()              __HAL_RCC_GPIOA_CLK_ENABLE();
#define PWM_TIM_CH2_GPIO_PORT               GPIOA
#define PWM_TIM_CH2_PIN                     GPIO_PIN_11
```

bsp_motor_tim.c

```
void TIM_SetTIMxCompare(TIM_TypeDef *TIMx,uint32_t channel,uint32_t compare);
void TIM_SetPWM_period(TIM_TypeDef* TIMx,uint32_t TIM_period);

/**
  * @brief   配置 TIM 复用输出 PWM 时用到的 I/O
  * @param   无
  * @retval  无
  */
static void TIMx_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /*  定时器通道功能引脚端口时钟使能  */

    PWM_TIM_CH1_GPIO_CLK();
    PWM_TIM_CH2_GPIO_CLK();

    /*  定时器通道 1 功能引脚 IO 初始化  */
    /*设置输出类型*/
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    /*设置引脚速率  */
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    /*设置复用*/
    //PWM_TIM_GPIO_AF_ENALBE();

    /*选择要控制的 GPIO 引脚*/
    GPIO_InitStruct.Pin = PWM_TIM_CH1_PIN;
    /*调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
    HAL_GPIO_Init(PWM_TIM_CH1_GPIO_PORT, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = PWM_TIM_CH2_PIN;
    HAL_GPIO_Init(PWM_TIM_CH2_GPIO_PORT, &GPIO_InitStruct);

}


/*
 *  注意：TIM_TimeBaseInitTypeDef 结构体里面有 5 个成员，TIM6 和 TIM7 的寄存器里面只有
 * TIM_Prescaler 和 TIM_Period，所以使用 TIM6 和 TIM7 的时候只需初始化这两个成员即可，
 *  另外三个成员是通用定时器和高级定时器才有.
 *-----------------------------------------------------------------------------
 * TIM_Prescaler                都有
 * TIM_CounterMode                    TIMx,x[6,7]没有，其他都有（基本定时器）
 * TIM_Period                   都有
 * TIM_ClockDivision         TIMx,x[6,7]没有，其他都有(基本定时器)
 * TIM_RepetitionCounter TIMx,x[1,8]才有(高级定时器)
 *-----------------------------------------------------------------------------
 */
TIM_HandleTypeDef    DCM_TimeBaseStructure;
static void TIM_PWMOUTPUT_Config(void)
{
    TIM_OC_InitTypeDef    TIM_OCInitStructure;

    /*使能定时器*/
    PWM_TIM_CLK_ENABLE();
```

```
    DCM_TimeBaseStructure.Instance = PWM_TIM;
    /*  累计  TIM_Period 个后产生一个更新或者中断*/
    //当定时器从 0 计数到 PWM_PERIOD_COUNT，即为 PWM_PERIOD_COUNT+1 次，为一个定时周期
    DCM_TimeBaseStructure.Init.Period = PWM_PERIOD_COUNT - 1;
    //  通用控制定时器时钟源 TIMxCLK = HCLK/2=84MHz
    //  设定定时器频率为=TIMxCLK/(PWM_PRESCALER_COUNT+1)
    DCM_TimeBaseStructure.Init.Prescaler = PWM_PRESCALER_COUNT - 1;

    /*计数方式*/
    DCM_TimeBaseStructure.Init.CounterMode = TIM_COUNTERMODE_UP;
    /*采样时钟分频*/
    DCM_TimeBaseStructure.Init.ClockDivision=TIM_CLOCKDIVISION_DIV1;
    /*初始化定时器*/
    HAL_TIM_PWM_Init(&DCM_TimeBaseStructure);

    /*PWM 模式配置*/
    TIM_OCInitStructure.OCMode = TIM_OCMODE_PWM1;
    TIM_OCInitStructure.Pulse = 0;
    TIM_OCInitStructure.OCPolarity = TIM_OCPOLARITY_HIGH;
    TIM_OCInitStructure.OCNPolarity = TIM_OCPOLARITY_HIGH;
    TIM_OCInitStructure.OCIdleState = TIM_OCIDLESTATE_RESET;
    TIM_OCInitStructure.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    TIM_OCInitStructure.OCFastMode = TIM_OCFAST_DISABLE;

    /*配置 PWM 通道*/
    HAL_TIM_PWM_ConfigChannel(&DCM_TimeBaseStructure,        &TIM_OCInitStructure,
PWM_CHANNEL_1);
    /*开始输出 PWM*/
    HAL_TIM_PWM_Start(&DCM_TimeBaseStructure,PWM_CHANNEL_1);

    /*配置脉宽*/
    TIM_OCInitStructure.Pulse = 500;      // 默认占空比为 50%
    /*配置 PWM 通道*/
    HAL_TIM_PWM_ConfigChannel(&DCM_TimeBaseStructure,        &TIM_OCInitStructure,
PWM_CHANNEL_2);
    /*开始输出 PWM*/
    HAL_TIM_PWM_Start(&DCM_TimeBaseStructure,PWM_CHANNEL_2);
}

/**
  * @brief   设置 TIM 通道的占空比
  *@param   channel        通道     （1,2,3,4）
  *@param   compare        占空比
  *@note   无
  * @retval 无
  */
void TIM1_SetPWM_pulse(uint32_t channel,int compare)
{
    switch(channel)
    {
        case                                                    TIM_CHANNEL_1:
__HAL_TIM_SET_COMPARE(&DCM_TimeBaseStructure,TIM_CHANNEL_1,compare);break;
        case TIM_CHANNEL_2:
__HAL_TIM_SET_COMPARE(&DCM_TimeBaseStructure,TIM_CHANNEL_2,compare);break;
        case TIM_CHANNEL_3:
__HAL_TIM_SET_COMPARE(&DCM_TimeBaseStructure,TIM_CHANNEL_3,compare);break;
```

```
        case TIM_CHANNEL_4:
__HAL_TIM_SET_COMPARE(&DCM_TimeBaseStructure,TIM_CHANNEL_4,compare);break;
    }
}


/**
  * @brief  初始化控制通用定时器
  * @param  无
  * @retval 无
  */
void Motor_TIMx_Configuration(void)
{
    TIMx_GPIO_Config();
    TIM_PWMOUTPUT_Config();
}
```

bsp_encoder.h

```
/* 定时器选择 */
#define ENCODER_TIM                                 TIM4
#define ENCODER_TIM_CLK_ENABLE()                    __HAL_RCC_TIM4_CLK_ENABLE()
#define ENCODER_TIM_AF_CLK_ENABLE()                 __HAL_AFIO_REMAP_TIM4_ENABLE()
/* 定时器溢出值 */
#define ENCODER_TIM_PERIOD                          65535
/* 定时器预分频值 */
#define ENCODER_TIM_PRESCALER                       0
/* 定时器中断 */
#define ENCODER_TIM_IRQn                            TIM4_IRQn
#define ENCODER_TIM_IRQHandler                      TIM4_IRQHandler
/* 编码器接口引脚 */
#define ENCODER_TIM_CH1_GPIO_CLK_ENABLE()           __HAL_RCC_GPIOB_CLK_ENABLE()
#define ENCODER_TIM_CH1_GPIO_PORT                   GPIOB
#define ENCODER_TIM_CH1_PIN                         GPIO_PIN_6
#define ENCODER_TIM_CH2_GPIO_CLK_ENABLE()           __HAL_RCC_GPIOB_CLK_ENABLE()
#define ENCODER_TIM_CH2_GPIO_PORT                   GPIOB
#define ENCODER_TIM_CH2_PIN                         GPIO_PIN_7
/* 编码器接口倍频数 */
#define ENCODER_MODE                                TIM_ENCODERMODE_TI12
/* 编码器接口输入捕获通道相位设置 */
#define ENCODER_IC1_POLARITY                        TIM_ICPOLARITY_RISING
#define ENCODER_IC2_POLARITY                        TIM_ICPOLARITY_RISING
/* 编码器物理分辨率 */
#define ENCODER_RESOLUTION                          13
/* 经过倍频之后的总分辨率 */
#if (ENCODER_MODE == TIM_ENCODERMODE_TI12)
  #define ENCODER_TOTAL_RESOLUTION                  (ENCODER_RESOLUTION * 4)   /* 4
倍频后的总分辨率 */
#else
  #define ENCODER_TOTAL_RESOLUTION                  (ENCODER_RESOLUTION * 2)   /* 2
倍频后的总分辨率 */
#endif
/* 减速电机减速比 */
#define REDUCTION_RATIO                             30
extern __IO int16_t Encoder_Overflow_Count;
extern TIM_HandleTypeDef TIM_EncoderHandle;
```

bsp_encoder.c

```
/* 定时器溢出次数 */
__IO int16_t Encoder_Overflow_Count = 0;
```

```
TIM_HandleTypeDef TIM_EncoderHandle;

/**
  * @brief  编码器接口引脚初始化
  * @param  无
  * @retval 无
  */
static void Encoder_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* 定时器通道引脚端口时钟使能 */
    ENCODER_TIM_CH1_GPIO_CLK_ENABLE();
    ENCODER_TIM_CH2_GPIO_CLK_ENABLE();
    /* 设置重映射 */
    //ENCODER_TIM_AF_CLK_ENABLE();
    /**TIM3 GPIO Configuration
    PC6         ------> TIM3_CH1
    PC7         ------> TIM3_CH2
    */
    /* 设置输入类型 */
    GPIO_InitStruct.Mode = GPIO_MODE_AF_INPUT;
    /* 设置上拉 */
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    /* 设置引脚速率 */
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = ENCODER_TIM_CH1_PIN;
    /* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO */
    HAL_GPIO_Init(ENCODER_TIM_CH1_GPIO_PORT, &GPIO_InitStruct);
    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStruct.Pin = ENCODER_TIM_CH2_PIN;
    /* 调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO */
    HAL_GPIO_Init(ENCODER_TIM_CH2_GPIO_PORT, &GPIO_InitStruct);
}

/**
  * @brief  配置 TIMx 编码器模式
  * @param  无
  * @retval 无
  */
static void TIM_Encoder_Init(void)
{
    TIM_Encoder_InitTypeDef Encoder_ConfigStructure;
    /* 使能编码器接口时钟 */
    ENCODER_TIM_CLK_ENABLE();
    /* 定时器初始化设置 */
    TIM_EncoderHandle.Instance = ENCODER_TIM;
    TIM_EncoderHandle.Init.Prescaler = ENCODER_TIM_PRESCALER;
    TIM_EncoderHandle.Init.CounterMode = TIM_COUNTERMODE_UP;
    TIM_EncoderHandle.Init.Period = ENCODER_TIM_PERIOD;
    TIM_EncoderHandle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    TIM_EncoderHandle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;

    /* 设置编码器倍频数 */
    Encoder_ConfigStructure.EncoderMode = ENCODER_MODE;
    /* 编码器接口通道 1 设置 */
    Encoder_ConfigStructure.IC1Polarity = ENCODER_IC1_POLARITY;
    Encoder_ConfigStructure.IC1Selection = TIM_ICSELECTION_DIRECTTI;
```

```
    Encoder_ConfigStructure.IC1Prescaler = TIM_ICPSC_DIV1;
    Encoder_ConfigStructure.IC1Filter = 0;
    /* 编码器接口通道 2 设置 */
    Encoder_ConfigStructure.IC2Polarity = ENCODER_IC2_POLARITY;
    Encoder_ConfigStructure.IC2Selection = TIM_ICSELECTION_DIRECTTI;
    Encoder_ConfigStructure.IC2Prescaler = TIM_ICPSC_DIV1;
    Encoder_ConfigStructure.IC2Filter = 0;
    /* 初始化编码器接口 */
    HAL_TIM_Encoder_Init(&TIM_EncoderHandle, &Encoder_ConfigStructure);

    /* 清零计数器 */
    __HAL_TIM_SET_COUNTER(&TIM_EncoderHandle, 0);

    /* 清零中断标志位 */
    __HAL_TIM_CLEAR_IT(&TIM_EncoderHandle,TIM_IT_UPDATE);
    /* 使能定时器的更新事件中断 */
    __HAL_TIM_ENABLE_IT(&TIM_EncoderHandle,TIM_IT_UPDATE);
    /* 设置更新事件请求源为：计数器溢出 */
    __HAL_TIM_URS_ENABLE(&TIM_EncoderHandle);

    /* 设置中断优先级 */
    HAL_NVIC_SetPriority(ENCODER_TIM_IRQn, 1, 0);
    /* 使能定时器中断 */
    HAL_NVIC_EnableIRQ(ENCODER_TIM_IRQn);

    /* 使能编码器接口 */
    HAL_TIM_Encoder_Start(&TIM_EncoderHandle, TIM_CHANNEL_ALL);
}

/**
  * @brief  编码器接口初始化
  * @param  无
  * @retval 无
  */
void Encoder_Init(void)
{
    Encoder_GPIO_Init();      /* 引脚初始化 */
    TIM_Encoder_Init();       /* 配置编码器接口 */
}
```

bsp_pid.h

```
typedef struct{
    float (*PID_realize)(float);
}pid_realize;

typedef struct
{
    float target_val,Starget_val;        //目标值
    float actual_val,Sactual_val;        //实际值
    float err,Serr;                  //定义当前偏差值
    float err_next,Serr_next;            //定义下一个偏差值
    float err_last,Serr_last;           //定义最后一个偏差值
    float Kp, Ki, Kd,SKp,SKi,SKd;        //定义比例、积分、微分系数
    uint32_t Mode;//控制模式(0x55)，位置控制 or 速度控制(0xAA)
}_pid;
```

bsp_pid.c

```
//定义全局变量
_pid;
```

```
/**
  * @brief  PID 参数初始化
    *   @note  无
  * @retval 无
  */
void PID_param_init()
{
    /*位置控制初始化参数 */
    pid.target_val=CIRCLE_PULSES;
    pid.actual_val=0.0;
    pid.err = 0.0;
    pid.err_last = 0.0;
    pid.err_next = 0.0;
    pid.Kp = 5.2;
    pid.Ki = 0;
    pid.Kd = 5;
    /* 速度控制初始化参数 */
    pid.Starget_val=100;
    pid.Sactual_val=0.0;
    pid.Serr = 0.0;
    pid.Serr_last = 0.0;
    pid.Serr_next = 0.0;
    pid.SKp = 6;
    pid.SKi = 2.5;
    pid.SKd = 0;
    //控制模式
    pid.Mode = 0x55; //位置控制
#if 1
    uint32_t i;
    for(i=100000;i>0;i--);
     //float pid_temp[3] = {pid.Kp, pid.Ki, pid.Kd};
    uint32_t pid_temp[3] = {(uint32_t)(pid.Kp*10), (uint32_t)(pid.Ki*10),
(uint32_t)(pid.Kd*10)};
     set_computer_value(SEND_P_I_D_CMD, CURVES_CH1, pid_temp, 3);  // 给通道 1
发送 P I D 值
    for(i=100000;i>0;i--);
    uint32_t Spid_temp[3] = {(uint32_t)(pid.SKp*10), (uint32_t)(pid.SKi*10),
(uint32_t)(pid.SKd*10)};
     set_computer_value(SEND_S_P_I_D_CMD, CURVES_CH1, Spid_temp, 3);  // 给通道
1 发送 P I D 值
    for(i=100000;i>0;i--);
    set_computer_value(SEND_MODE_CMD, CURVES_CH1,&(pid.Mode), 1);  // 给通道
1 发送 PID 控制模式
#endif
}

/**
  * @brief  设置目标值
  * @param  val      目标值
    *   @note  无
  * @retval 无
  */
void set_pid_target(float temp_val)
{
    if(pid.Mode==0x55)
        pid.target_val = temp_val;  // 设置位置当前的目标值
    else if(pid.Mode==0xAA)
        pid.Starget_val = temp_val;  // 设置速度当前的目标值
```

```
}

/**
  * @brief  获取目标值
  * @param  无
  *    @note  无
  * @retval 目标值
  */
float get_pid_target(void)
{
    float tar;
    if(pid.Mode==0x55)
    {
        tar = pid.target_val;      // 设置当前的目标值
    }
    else if(pid.Mode==0xAA)
    {
        tar = pid.Starget_val;      // 设置当前的目标值
    }
    return tar;
}

/**
  * @brief  设置比例、积分、微分系数
  * @param  p：比例系数 P
  * @param  i：积分系数 i
  * @param  d：微分系数 d
  *    @note  无
  * @retval 无
  */
void set_p_i_d(float p, float i, float d)
{
    uint32_t mode;
    mode = pid.Mode;
    if(mode==0x55) //位置控制 PID 参数
    {
        pid.Kp = p;      // 设置比例系数 P
        pid.Ki = i;      // 设置积分系数 I
        pid.Kd = d;      // 设置微分系数 D
    }
    else if(mode==0xAA)//速度控制 PID 参数
    {
        pid.SKp = p;      // 设置比例系数 P
        pid.SKi = i;      // 设置积分系数 I
        pid.SKd = d;      // 设置微分系数 D
    }
}
/**
  * @brief  设置 PID 控制模式
  * @param  mode：控制模式
  * @note  无
  * @retval 无
  */
void set_control_mode(uint32_t mode)
{
    pid.Mode = mode;      // 设置控制模式
}
```

```
/**
  * @brief   位置控制 PID 算法实现
  * @param   actual_val:实际值
  * @note   无
  * @retval  通过 PID 计算后的输出
  */
float W_PID_realize(float actual_val)
{
    /*计算目标值与实际值的误差*/
    pid.err=pid.target_val-actual_val;
    //printf("pid.err=%f\n",pid.err);
    if (pid.err > -5 && pid.err < 5)
    {
        pid.err = 0;
    }
    /*PID 算法实现*/
    pid.actual_val += pid.Kp * (pid.err - pid.err_next) +
                      pid.Ki * pid.err +
                      pid.Kd * (pid.err - 2 * pid.err_next + pid.err_last);
    /*传递误差*/
    pid.err_last = pid.err_next;
    pid.err_next = pid.err;
/*返回当前实际值*/
    return pid.actual_val;
}
/**
  * @brief   速度 PID 算法实现
  * @param   actual_val:实际值
  * @note   无
  * @retval  通过 PID 计算后的输出
  */
float S_PID_realize(float actual_val)
{
    /*计算目标值与实际值的误差*/
    pid.Serr=pid.Starget_val-actual_val;
    /*PID 算法实现*/
    pid.Sactual_val += pid.SKp*(pid.Serr - pid.Serr_next)
                    + pid.SKi*pid.Serr
                      + pid.SKd*(pid.Serr - 2 * pid.Serr_next + pid.Serr_last);
    /*传递误差*/
    pid.Serr_last = pid.Serr_next;
    pid.Serr_next = pid.Serr;
    /*返回当前实际值*/
    return pid.Sactual_val;
}
/**
 *@brief PID 算法回调函数
 *@param actual_val:实际值
 *@param (*PIDRealize)(float)  具体 PID 算法函数，有位置控制、和速度控制
 *@return  通过 PID 计算后的输出
 */
float PID_realize(float actual_val,float (*PIDRealize)(float))
{
    return ((*PIDRealize)(actual_val));
}
```

protocol.h

```
/*  数据接收缓冲区大小  */
#define PROT_FRAME_LEN_RECV    128
```

```
/* 校验数据的长度 */
#define PROT_FRAME_LEN_CHECKSUM       1
/* 数据头结构体 */
typedef __packed struct
{
   uint32_t head;       // 包头
   uint8_t ch;          // 通道
   uint32_t len;        // 包长度
   uint8_t cmd;         // 命令
//   uint8_t sum;         // 校验和
}packet_head_t;
#define FRAME_HEADER        0x59485A53      // 帧头
/* 通道宏定义 */
#define CURVES_CH1          0x01
#define CURVES_CH2          0x02
#define CURVES_CH3          0x03
#define CURVES_CH4          0x04
#define CURVES_CH5          0x05
/* 指令(下位机 -> 上位机) */
#define SEND_TARGET_CMD         0x01       // 发送上位机通道的目标值
#define SEND_FACT_CMD           0x02       // 发送通道实际值
#define SEND_P_I_D_CMD          0x03       // 发送位置控制 PID 值（同步上位机显示的值）
#define SEND_START_CMD          0x04       // 发送启动指令（同步上位机按钮状态）
#define SEND_STOP_CMD           0x05       // 发送停止指令（同步上位机按钮状态）
#define SEND_PERIOD_CMD         0x06       // 发送周期（同步上位机显示的值）
#define SEND_MODE_CMD           0x07       // 发送 PID 控制模式(同步上位机显示的值)
#define SEND_S_P_I_D_CMD        0x08       // 发送速度控制 PID 值（同步上位机显示的值）
/* 指令(上位机 -> 下位机) */
#define SET_P_I_D_CMD           0x10       // 设置 PID 值
#define SET_TARGET_CMD          0x11       // 设置目标值
#define START_CMD               0x12       // 启动指令
#define STOP_CMD                0x13       // 停止指令
#define RESET_CMD               0x14       // 复位指令
#define SET_PERIOD_CMD          0x15       // 设置周期
#define SET_MODE_CMD        0x16       // 设置 PID 控制模式
/* 空指令 */
#define CMD_NONE                0xFF       // 空指令
/* 索引值宏定义 */
#define HEAD_INDEX_VAL          0x3u       // 包头索引值（4 字节）
#define CHX_INDEX_VAL           0x4u       // 通道索引值（1 字节）
#define LEN_INDEX_VAL           0x5u       // 包长索引值（4 字节）
#define CMD_INDEX_VAL           0x9u       // 命令索引值（1 字节）
#define EXCHANGE_H_L_BIT(data)          ((((data) << 24) & 0xFF000000) |\
                                        (((data) <<  8) & 0x00FF0000) |\
                                        (((data) >>  8) & 0x0000FF00) |\
                                        (((data) >> 24) & 0x000000FF))      // 交换高低字节
#define COMPOUND_32BIT(data)            (((*(data-0) << 24) & 0xFF000000) |\
                                        ((*(data-1) << 16) & 0x00FF0000) |\
                                        ((*(data-2) <<  8) & 0x0000FF00) |\
                                        ((*(data-3) <<  0) & 0x000000FF))       // 合成为一个字
```

protocol.c

```
struct prot_frame_parser_t
{
```

```
    uint8_t *recv_ptr;
    uint16_t r_oft;
    uint16_t w_oft;
    uint16_t frame_len;
    uint16_t found_frame_head;
};
static struct prot_frame_parser_t parser;
static uint8_t recv_buf[PROT_FRAME_LEN_RECV];
/**
  * @brief 计算校验和
  * @param ptr：需要计算的数据
  * @param len：需要计算的长度
  * @retval 校验和
  */
uint8_t check_sum(uint8_t init, uint8_t *ptr, uint8_t len )
{
  uint8_t sum = init;
  while(len--)
  {
    sum += *ptr;
    ptr++;
  }
  return sum;
}
/**
 * @brief    得到帧类型（帧命令）
 * @param    *frame:  数据帧
 * @param    head_oft: 帧头的偏移位置
 * @return   帧长度.
 */
static uint8_t get_frame_type(uint8_t *frame, uint16_t head_oft)
{
    return (frame[(head_oft + CMD_INDEX_VAL) % PROT_FRAME_LEN_RECV] & 0xFF);
}
/**
 * @brief    得到帧长度
 * @param    *buf:  数据缓冲区.
 * @param    head_oft: 帧头的偏移位置
 * @return   帧长度.
 */
static uint16_t get_frame_len(uint8_t *frame, uint16_t head_oft)
{
    return ((frame[(head_oft + LEN_INDEX_VAL + 0) % PROT_FRAME_LEN_RECV] <<  0) |
            (frame[(head_oft + LEN_INDEX_VAL + 1) % PROT_FRAME_LEN_RECV] <<  8) |
            (frame[(head_oft + LEN_INDEX_VAL + 2) % PROT_FRAME_LEN_RECV] << 16) |
            (frame[(head_oft + LEN_INDEX_VAL + 3) % PROT_FRAME_LEN_RECV] << 24));
// 合成帧长度
}
/**
 * @brief    获取 crc-16 校验值
 * @param    *frame:  数据缓冲区.
 * @param    head_oft: 帧头的偏移位置
 * @param    head_oft: 帧长
 * @return   帧长度.
 */
static uint8_t get_frame_checksum(uint8_t *frame, uint16_t head_oft, uint16_t frame_len)
{
    return (frame[(head_oft + frame_len - 1) % PROT_FRAME_LEN_RECV]);
```

```
}

/**
 * @brief     查找帧头
 * @param     *buf:   数据缓冲区.
 * @param     ring_buf_len:  缓冲区大小
 * @param     start:  起始位置
 * @param     len:  需要查找的长度
 * @return    -1：没有找到帧头，其他值：帧头的位置.
 */
static int32_t recvbuf_find_header(uint8_t *buf, uint16_t ring_buf_len, uint16_t start, uint16_t
len)
{
    uint16_t i = 0;
    for (i = 0; i < (len - 3); i++)
    {
        if (((buf[(start + i + 0) % ring_buf_len] <<  0) |
             (buf[(start + i + 1) % ring_buf_len] <<  8) |
             (buf[(start + i + 2) % ring_buf_len] << 16) |
             (buf[(start + i + 3) % ring_buf_len] << 24)) == FRAME_HEADER)
        {
            return ((start + i) % ring_buf_len);
        }
    }
    return -1;
}
/**
 * @brief     计算为解析的数据长度
 * @param     *buf:   数据缓冲区.
 * @param     ring_buf_len:  缓冲区大小
 * @param     start:  起始位置
 * @param     end:  结束位置
 * @return    为解析的数据长度
 */
static int32_t recvbuf_get_len_to_parse(uint16_t frame_len, uint16_t ring_buf_len,uint16_t
start, uint16_t end)
{
    uint16_t unparsed_data_len = 0;
    if (start <= end)
        unparsed_data_len = end - start;
    else
        unparsed_data_len = ring_buf_len - start + end;
    if (frame_len > unparsed_data_len)
        return 0;
    else
        return unparsed_data_len;
}

/**
 * @brief     接收数据写入缓冲区
 * @param     *buf:   数据缓冲区.
 * @param     ring_buf_len:  缓冲区大小
 * @param     w_oft:  写偏移
 * @param     *data:  需要写入的数据
 * @param     *data_len:  需要写入数据的长度
 * @return    void.
 */
static void recvbuf_put_data(uint8_t *buf, uint16_t ring_buf_len, uint16_t w_oft,
```

```
        uint8_t *data, uint16_t data_len)
{
    if ((w_oft + data_len) > ring_buf_len)                    // 超过缓冲区尾
    {
        uint16_t data_len_part = ring_buf_len - w_oft;        // 缓冲区剩余长度
        /* 数据分两段写入缓冲区*/
        memcpy(buf + w_oft, data, data_len_part);                                  // 写入缓
冲区尾
        memcpy(buf, data + data_len_part, data_len - data_len_part);          // 写入缓冲区
头
    }
    else
        memcpy(buf + w_oft, data, data_len);      // 数据写入缓冲区
}
/**
 * @brief     查询帧类型（命令）
 * @param     *data:   帧数据
 * @param     data_len: 帧数据的大小
 * @return    帧类型（命令）.
 */
static uint8_t protocol_frame_parse(uint8_t *data, uint16_t *data_len)
{
    uint8_t frame_type = CMD_NONE;
    uint16_t need_to_parse_len = 0;
    int16_t header_oft = -1;
    uint8_t checksum = 0;
    need_to_parse_len              =              recvbuf_get_len_to_parse(parser.frame_len,
PROT_FRAME_LEN_RECV, parser.r_oft, parser.w_oft);      // 得到为解析的数据长度
    if (need_to_parse_len < 9)        // 肯定还不能同时找到帧头和帧长度
        return frame_type;
    /* 还未找到帧头，需要进行查找*/
    if (0 == parser.found_frame_head)
    {
        /* 同步头为四字节，可能存在未解析的数据中最后一个字节刚好为同步头第一
个字节的情况，
           因此查找同步头时，最后一个字节将不解析，也不会被丢弃*/
        header_oft    =    recvbuf_find_header(parser.recv_ptr,    PROT_FRAME_LEN_RECV,
parser.r_oft, need_to_parse_len);
        if (0 <= header_oft)
        {
            /* 已找到帧头*/
            parser.found_frame_head = 1;
            parser.r_oft = header_oft;
            /* 确认是否可以计算帧长*/
            if (recvbuf_get_len_to_parse(parser.frame_len, PROT_FRAME_LEN_RECV,
                    parser.r_oft, parser.w_oft) < 9)
                return frame_type;
        }
        else
        {
            /* 未解析的数据中依然未找到帧头，丢掉此次解析过的所有数据*/
            parser.r_oft    =    ((parser.r_oft    +    need_to_parse_len    -    3)    %
PROT_FRAME_LEN_RECV);
            return frame_type;
        }
    }
    /* 计算帧长，并确定是否可以进行数据解析*/
    if (0 == parser.frame_len)
```

```
    {
        parser.frame_len = get_frame_len(parser.recv_ptr, parser.r_oft);
        if(need_to_parse_len < parser.frame_len)
            return frame_type;
    }
    /* 帧头位置确认，且未解析的数据超过帧长，可以计算校验和*/
    if ((parser.frame_len + parser.r_oft - PROT_FRAME_LEN_CHECKSUM) >
PROT_FRAME_LEN_RECV)
    {
        /* 数据帧被分为两部分，一部分在缓冲区尾，一部分在缓冲区头 */
        checksum = check_sum(checksum, parser.recv_ptr + parser.r_oft,
                PROT_FRAME_LEN_RECV - parser.r_oft);
        checksum = check_sum(checksum, parser.recv_ptr, parser.frame_len -
                PROT_FRAME_LEN_CHECKSUM + parser.r_oft - PROT_FRAME_LEN_RECV);
    }
    else
    {
        /* 数据帧可以一次性取完*/
        checksum = check_sum(checksum, parser.recv_ptr + parser.r_oft, parser.frame_len -
PROT_FRAME_LEN_CHECKSUM);
    }
    if (checksum == get_frame_checksum(parser.recv_ptr, parser.r_oft, parser.frame_len))
    {
        /* 校验成功，拷贝整帧数据 */
        if ((parser.r_oft + parser.frame_len) > PROT_FRAME_LEN_RECV)
        {
            /* 数据帧被分为两部分，一部分在缓冲区尾，一部分在缓冲区头*/
            uint16_t data_len_part = PROT_FRAME_LEN_RECV - parser.r_oft;
            memcpy(data, parser.recv_ptr + parser.r_oft, data_len_part);
            memcpy(data + data_len_part, parser.recv_ptr, parser.frame_len -
data_len_part);
        }
        else
        {
            /* 数据帧可以一次性取完*/
            memcpy(data, parser.recv_ptr + parser.r_oft, parser.frame_len);
        }
        *data_len = parser.frame_len;
        frame_type = get_frame_type(parser.recv_ptr, parser.r_oft);

        /* 丢弃缓冲区中的命令帧*/
        parser.r_oft = (parser.r_oft + parser.frame_len) % PROT_FRAME_LEN_RECV;
    }
    else
    {
        /* 校验错误，说明之前找到的帧头只是偶然出现的废数据*/
        parser.r_oft = (parser.r_oft + 1) % PROT_FRAME_LEN_RECV;
    }
    parser.frame_len = 0;
    parser.found_frame_head = 0;
    return frame_type;
}
/**
 * @brief     接收数据处理
 * @param     *data:   要计算的数据的数组.
 * @param     data_len: 数据的大小
 * @return    void.
 */
```

```
void protocol_data_recv(uint8_t *data, uint16_t data_len)
{
    recvbuf_put_data(parser.recv_ptr, PROT_FRAME_LEN_RECV, parser.w_oft, data, data_len);
// 接收数据
    parser.w_oft = (parser.w_oft + data_len) % PROT_FRAME_LEN_RECV;
// 计算写偏移
}

/**
 * @brief   初始化接收协议
 * @param   void
 * @return  初始化结果.
 */
int32_t protocol_init(void)
{
    memset(&parser, 0, sizeof(struct prot_frame_parser_t));

    /* 初始化分配数据接收与解析缓冲区*/
    parser.recv_ptr = recv_buf;
    return 0;
}
/**
 * @brief   接收的数据处理
 * @param   void
 * @return  -1：没有找到一个正确的命令.
 */
int8_t receiving_process(void)
{
  uint8_t frame_data[128];            // 要能放下最长的帧
  uint16_t frame_len = 0;             // 帧长度
  uint8_t cmd_type = CMD_NONE;        // 命令类型
  while(1)
  {
    cmd_type = protocol_frame_parse(frame_data, &frame_len);
    switch (cmd_type)
    {
      case CMD_NONE:
      {
        return -1;
      }
      case SET_P_I_D_CMD:
      {
        uint32_t temp0 = COMPOUND_32BIT(&frame_data[13]);
        uint32_t temp1 = COMPOUND_32BIT(&frame_data[17]);
        uint32_t temp2 = COMPOUND_32BIT(&frame_data[21]);

        float p_temp, i_temp, d_temp;
//          p_temp = *(float *)&temp0;
//          i_temp = *(float *)&temp1;
//          d_temp = *(float *)&temp2;
        p_temp = (float)(temp0/10.0);
        i_temp = (float)(temp1/10.0);
        d_temp = (float)(temp2/10.0);
        set_p_i_d(p_temp, i_temp, d_temp);      // 设置 P I D
      }
      break;
      case SET_TARGET_CMD:
      {
```

```
            int actual_temp = COMPOUND_32BIT(&frame_data[13]);      // 得到数据

            set_pid_target(actual_temp);      // 设置目标值
        }
        break;

        case START_CMD:
        {
            set_motor_enable();                    // 启动电机
        }
        break;

        case STOP_CMD:
        {
            set_motor_disable();                   // 停止电机
        }
        break;

        case RESET_CMD:
        {
            HAL_NVIC_SystemReset();                // 复位系统
        }
        break;

        case SET_PERIOD_CMD:
        {
            uint32_t temp = COMPOUND_32BIT(&frame_data[13]);       // 周期数
            SET_BASIC_TIM_PERIOD(temp);                                  // 设置定时器
周期 1~1000ms
        }
        break;
       case SET_MODE_CMD:
        {
            uint32_t mode = COMPOUND_32BIT(&frame_data[13]);       // 周期数
            set_control_mode(mode);                                   // 设置定时器周期
1~1000ms
        }
        break;
        default:
            return -1;
    }
  }
}
/**
  * @brief 设置上位机的值
  * @param cmd：命令
  * @param ch: 曲线通道
  * @param data：参数指针
  * @param num：参数个数
  * @retval 无
  */
void set_computer_value(uint8_t cmd, uint8_t ch, void *data, uint8_t num)
{
  uint8_t sum = 0;      // 校验和
  num *= 4;               // 一个参数 4 个字节

  static packet_head_t set_packet;
```

```
    set_packet.head = FRAME_HEADER;         // 包头 0x59485A53
    set_packet.len    = 0x0B + num;         // 包长
    set_packet.ch     = ch;                 // 设置通道
    set_packet.cmd    = cmd;                // 设置命令

    sum = check_sum(0, (uint8_t *)&set_packet, sizeof(set_packet));          // 计算包头校验
和
    sum = check_sum(sum, (uint8_t *)data, num);                              // 计算参数
校验和
    HAL_UART_Transmit(&UartHandle, (uint8_t *)&set_packet, sizeof(set_packet), 0xFFFFF);
// 发送数据头
    HAL_UART_Transmit(&UartHandle, (uint8_t *)data, num, 0xFFFFF);
// 发送参数
    HAL_UART_Transmit(&UartHandle, (uint8_t *)&sum, sizeof(sum), 0xFFFFF);
// 发送校验和
}
```