《计算机控制及接口技术》

实验报告

院　　系　机械与自动控制学院

学生姓名　张利昂

学　　号　2018G0505060

指导教师　李晓明

完成日期　2018 年 12 月

浙江理工大学

课程作业 1

以 Arduino 为硬件平台， 编写程序实现以下功能：
(1) 在指定的引脚上产生一个方波信号， 要求该方波信号的周期和占空比可调。
(2) 通过串口与用户进行人机交互， 可通过发送指令调整上述参数

程序：

1、 指定引脚编号为 9,输出周期与占空比的不同通过在 arduino 平台上接发光二极管显示。硬件连接如下图所示，将发光二极管的一脚接在 9 号引脚上，另一脚接在接地上。矩形波参数设置和发光二极管亮度参数设置如下：

```
//设置 led 为引脚 9
int led = 9;
//设置矩形波参数
float T=3000;
float dc=0.5;
//设置亮度参数
float t = T*dc;
```

2、 加入框架

包括框架头文件

```
#include "TaskScheduler.h"
#include <Wire.h>
```

TaskScheduler.h 代码如下

```
// TaskScheduler.h
// A task scheduler library for arduino - header file
// http://blanboom.org

#ifndef TASK_SCHEDULER_H
#define TASK_SCHEDULER_H

#if ARDUINO >= 100
#include "Arduino.h"
#else
#include "WProgram.h"
#endif



#define RETURN_ERROR 0
#define RETURN_NORMAL 1

// Error codes
//#define REPORT_ERRORS // Remove "//" to enable error report
#define ERROR_TOO_MANY_TASKS (1)
```

```
#define ERROR_CANNOT_DELETE_TASK (2)

// The max number of tasks
#define MAX_TASKS   (10)

// The data structure of per task
typedef struct
{
 void (*pTask)(void);  // The task to be scheduled
 uint16_t delay;            // The interval before the task is first
executed
 uint16_t period;        // Repeat period of the task
 uint8_t runMe;          // Is the task should be dispatched
 uint8_t co_op;          // If it is 1, it's a co-operative task
 // If it is 0, it's a pre-emptive task
} Task;

class Schedule
{
 //Scheduler functions
private:
 void _reportStatus(void);
 void _goToSleep(void);
public:
 void init(void);
 void start(void);
 void  dispatchTasks(void);
 uint8_t addTask(void ( *)(void),  boolean);
 boolean   deleteTask(uint8_t);
};

static Schedule Sch;

#endif
```

TaskScheduler.cpp 代码如下

```
// TaskScheduler.cpp
// A task scheduler library for arduino
// http://blanboom.org


#include <avr/sleep.h>
#include "TaskScheduler.h"
```

```
uint8_t g_errorCode;
Task g_Tasks[MAX_TASKS];  // Tasks array


//////////////////////////////////////////////////////////////////
// void Schedule::init(void)
// Initialize the task scheduler.
//////////////////////////////////////////////////////////////////
void Schedule::init(void)
{
 // Delete all tasks in the task array
 uint8_t i;
 for (i = 0; i < MAX_TASKS; i++)
 {
     deleteTask(i);
 }

 // Reset error code
 g_errorCode = 0;

 //Set up timer1. 1ms per interrupt
 cli(); //disable global interrupt
 TCCR1A = 0;
 TCCR1B = 0;
 TCNT1  = 0;
 OCR1A = 1999;
 TCCR1B |= (1 << WGM12);
 TCCR1B |= (1 << CS11);
 TIMSK1 |= (1 << OCIE1A);
}

//////////////////////////////////////////////////////////////////
// void Schedule::start(void)
// Start the task scheduler
//////////////////////////////////////////////////////////////////
void Schedule::start(void)
{
 sei();  //enable global interrupt
}


unsigned char Schedule::addTask(void (*pFn)(),  boolean co_op)
{
```

```
uint8_t index = 0;

// Find a gap in the array
while ((g_Tasks[index].pTask != 0) && (index < MAX_TASKS))
{
    index++;
}

if (index == MAX_TASKS)
{
    // The task array is full.Can't add more tasks
    g_errorCode = ERROR_TOO_MANY_TASKS;
    return MAX_TASKS;
}

// Put the task into task array
g_Tasks[index].pTask = pFn;

g_Tasks[index].co_op = co_op;

g_Tasks[index].runMe  = 0;

return index; // Return task ID
}

//////////////////////////////////////////////////////////////////
// boolean Schedule::deleteTask(uint8_t taskIndex)
// Delete a task
//
//     taskIndex     -   ID of the task to be deleted
//
// Return:
//     RETURN_ERROR  -   Can't delete the task.
//     RETURN_NORMAL -   Succeed in deleting the task
//////////////////////////////////////////////////////////////////
boolean Schedule::deleteTask(uint8_t taskIndex)
{
boolean returnCode;

if (g_Tasks[taskIndex].pTask == 0)
{
    // There is no task
    g_errorCode = ERROR_CANNOT_DELETE_TASK;
    returnCode = RETURN_ERROR;  // Return an error code
```

```
    }
    else
    {
        returnCode = RETURN_NORMAL;
    }

    g_Tasks[taskIndex].pTask   = 0;
    g_Tasks[taskIndex].delay   = 0;
    g_Tasks[taskIndex].period  = 0;

    g_Tasks[taskIndex].runMe   = 0;

    return returnCode;       // return status
}

//------------------------------------------------------------------
// void Schedule::dispatchTasks(void)
// Run tasks
//------------------------------------------------------------------
void Schedule::dispatchTasks(void)
{
    uint8_t index;
    for (index = 0; index < MAX_TASKS; index++)
    {
        // Run co-operative tasks only
        if ((g_Tasks[index].co_op) && (g_Tasks[index].runMe > 0))
        {
            (*g_Tasks[index].pTask)();  // Run the task

            g_Tasks[index].runMe -= 1;

            // If period = 0, this task will only run once
            if (g_Tasks[index].period == 0)
            {
                g_Tasks[index].pTask = 0;  // Delete the task
            }
        }
    }
    _reportStatus();  // Report errors (if necessary)
    _goToSleep();  // Go to sleep (idle) mode
}

//------------------------------------------------------------------
// void Schedule::_reportStatus(void)
```

```
// Report errors
// Define REPORT_ERRORS and error codes in "Scheduler.h" first
/////////////////////////////////////////////////////////////////
void Schedule::_reportStatus(void)
{
#ifdef REPORT_ERRORS
 // Put you codes to report errors
#endif
}


/////////////////////////////////////////////////////////////////
// void Schedule::_goToSleep()
// Enable sleep mode.
/////////////////////////////////////////////////////////////////
void Schedule::_goToSleep()
{
 set_sleep_mode(SLEEP_MODE_IDLE);
 sleep_enable();
 sleep_mode();
}

/////////////////////////////////////////////////////////////////
// ISR (TIMER1_COMPA_vect)
// Update task informations and run pre-emptive tasks
/////////////////////////////////////////////////////////////////
ISR (TIMER1_COMPA_vect)
{
 uint8_t index;
 for (index = 0; index < MAX_TASKS; index++)
 {
     // Check if there is a task
     if (g_Tasks[index].pTask)
     {
         if (g_Tasks[index].delay == 0)
         {
             if (g_Tasks[index].co_op)
             {
                 // A co-operative task
                 g_Tasks[index].runMe += 1;
             }
             else
             {
                 // A pre-emptive task
```

```
                    (*g_Tasks[index].pTask)();  // Run the task now

                    g_Tasks[index].runMe -= 1;

                    // Remove the task that the period = 0
                    if (g_Tasks[index].period == 0)
                    {
                        g_Tasks[index].pTask  = 0;
                    }
                }

                if (g_Tasks[index].period)
                {
                    // Schedule the task to run again
                    g_Tasks[index].delay = g_Tasks[index].period;
                }
            }
            else
            {
                // Decrease the delay
                g_Tasks[index].delay -= 1;
            }
        }
    }
}
```

为方便删除任务，使用函数 Sch.addTask(任务名称,开始时间,执行周期,1)时，会返回这个任务的 ID,将这个 ID 赋给一个变量。需要删除任务时，用删除任务函数 Sch.deleteTask(任务 ID)，就能把任务删除。

3、 设置串口缓存区，以用来临时存储串口发送来的数据，避免干扰到波形的正常生成。当当串口收到回车字符（'\r'）时，代表用户输入完毕，arduino 再统一处理用户输入的数据，具体代码如下：

```
//设置串口缓冲、数据长度参数、标志位
char dtaUart1[15];
char dtaUart2[15];
char dtaLen = 0;
char DealFlag1=0;
char DealFlag2=0;



//读串口与写缓存区
void serialEvent()
{
```

```
char temp;
unsigned int a,c;
float m;
char b;
  while(Serial.available())
  {
      temp=Serial.read();  //读串口数据
      dtaUart[dtaLen++] =temp ;  //将串口数据写入缓存区
      if(temp=='\r')   //当串口输入为回车键时
      {
        DealFlag1=1;  //标志位 1 置 1
        //判断起始位
        if(dtaUart[0]=='T')
            {
             dtaLen=dtaLen-2;
             for(c=0,b=2;dtaLen>1;dtaLen--)
             {
                switch(dtaUart[b++])
                {
                  case '0':a=0;break;
                  case '1':a=1;break;
                  case '2':a=2;break;
                  case '3':a=3;break;
                  case '4':a=4;break;
                  case '5':a=5;break;
                  case '6':a=6;break;
                  case '7':a=7;break;
                  case '8':a=8;break;
                  case '9':a=9;break;
                }
                c=c*10+a; //计算输入值
                dtaLen=0;
              }
              T=c;  //给周期 T 赋值
            }
            else if(dtaUart[0]=='d') //检测起始位
            {  char DealFlag2=1; //标志位 2 置 1
              dtaLen=dtaLen-5;
              for(m=0,b=5;dtaLen>1;dtaLen--)
             {
              switch(dtaUart[b++])
                {
                  case '0':a=0;break;
                  case '1':a=1;break;
```

```
                    case '2':a=2;break;
                    case '3':a=3;break;
                    case '4':a=4;break;
                    case '5':a=5;break;
                    case '6':a=6;break;
                    case '7':a=7;break;
                    case '8':a=8;break;
                    case '9':a=9;break;
                  }
                  m=m*10+a;
                }
                m=m/100; //由于一开始计算时将树脂放大了 100 倍，故此处除以 100
                dtaLen=0;
               dc=m; //将占空比值赋予 dc
                }
                t=m*T;
               //输出周期及占空比
                Serial.println("T=");
                Serial.println(T);
                Serial.println("dc=");
                Serial.println(dc);
                pwm= Sch.addTask(led9pwm,1);   //重新将 pwm 发生函数加入任务
                 }
//当 T 与 dc 都接受到串口的数据时，将 setUpdate 函数加入任务框架
          if（char DealFlag1=1）
              if（char DealFlag2=1）
                  Sch.addTask(setUpdate,1);
          break;
    }
}
```

4、 Arduino 读取缓存区里的数据

```
      void setUpdate()
      {
        Sch.deleteTask(pwm);  //将旧的 pwm 函数从任务中删除
        int a;
        char b;
          Sch.deleteTask(pwm);
          if(dtaUart[0]=='T')
          {
            dtaLen=dtaLen-3;
            for(a=(int)dtaUart[2], b=2;dtaLen;dtaLen--)
            {
```

```
            a=a*10+dtaUart[++b];
          }
         T=a;
        }
      else if(dtaUart[0]=='d')
          {
            dtaLen=dtaLen-3;
            for(a=(float)dtaUart[3],b=3;dtaLen;dtaLen--)
            {
                a=a*10+dtaUart[++b];
            }
            dc=a;
          }

      DealFlag1=0;
      DealFlag2=0;

    }
```

5、 pwm 主程序

```
int pwm;//设置多任务 ID
pwm= Sch.addTask(led9pwm,1); //将 PWM 程序加入任务框架
    void led9pwm()
    {
      digitalWrite(led,HIGH); //将引脚 9 拉高
      delay(t); //设置延迟
      digitalWrite(led,LOW);// //将引脚 9 拉低
      delay(T-t);//设置延迟
}
```

6、 初始化函数

```
    void setup()
    {
       pinMode(led, OUTPUT);
       Serial.begin(9600);//串口初始化
       Sch.init();
       pwm= Sch.addTask(led9pwm,1);
       Sch.start();
    }
```

7、主循环

```
    void loop()
    {
        Sch.dispatchTasks();
    }
```

实验硬件及实验结果串口监视器部分如下图 1 所示，先输入 T=5000；回车以后在输入 dc=0.2。可根据发光二极管的闪烁频率及亮暗的时间长短来判断占空比是否准确。

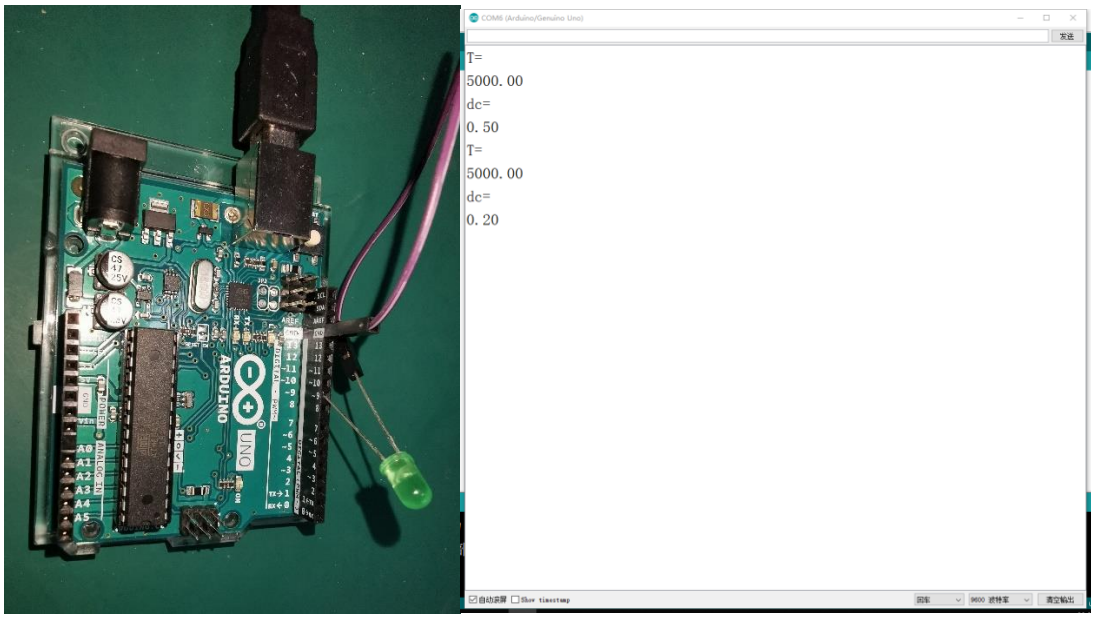
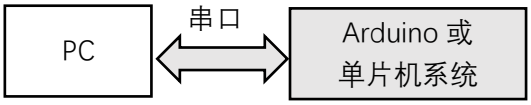


图 1 实验硬件（左）及串口监视器（右）

作业 2

一、设计目标

搭建以下实验系统：



在 PC 中编写软件，运行一仿真程序，模拟一系统（例如温度控制系统，电机控制系统等），要求包含该系统的模型，以及控制接口。该控制接口能够接收来自串口的控制指令。
在单片机中编写控制程序，实现离散 PID 控制。要求该程序包括 PID 控制算法以及控制接口实现，该控制接口能够控制 PC 里的模型程序。
分别运行上述实验系统，在 PC 端记录控制系统的状态曲线，绘制该曲线并进行说明。

二、设计内容

1、单片机程序

在本次实验中单片机部分主要是接收模拟电机控制系统传来的当前转速并进行 PID 运算得到输出值发送给模拟电机控制系统。本次实验采用的单片机为 LPC1768，软件部分使用 MDK，采用了 C 语言编程。下面主要就 main 函数，uart 函数，pid 函数进行讲解。

1.1 Main 函数

Main 函数主要用于初始化各类函数，并调用各个函数进行计算。程序如下：

```
#include "LPC17xx.h"
#include "uart.h"
#include "LED.h"
#include "PID.h"
#include "timer.h"
#include "key.h"
/*
    变量定义
*/
char state=0;
char flag_run=0;                                    //电机运行标志
//int Velocity_Integral=0;
////int Velocity_bias=0, EVelocity_goal=0, Velocity_Integral=0;
int PID_P=5000,PID_I=0,PID_D=0;                     //PID 参数
int Position,Velocity,Velocity_goal=1000;
char firstrunflag1=1,firstrunflag2=1;
int temp_Vel;
int main(void)
{
    SystemInit();                   //=====系统初始化
```

```
    Delayms_Init();                    //=====延时函数
    UART0_Init(9600);                  //=====串口 1 初始化 波特率：9600
    Timer_Init (0, SystemFrequency/200);    //5MS 进一次中断服务函数
    state=0;
}
```

1.2Uart 函数

Uart 主要用于与模拟电机控制系统进行通讯和数据传输，由于模拟电机调速系统使用 matlab Simulink ，simulink 接受和发送的数据类型为字符串，故对 uart 程序进行针对性设计，程序如下：

```
#include "lpc17xx.h"                                    /* LPC17xx definitions
*/
#include "uart.h"

#define FOSC            12000000  /*  振荡器频率  */

#define FCCLK          (FOSC  * 8)  /*  主时钟频率<=100Mhz  */
                                    /*  FOSC 的整数倍        */
#define FCCO        (FCCLK * 3)     /*  PLL 频率(275Mhz~550Mhz)      */
                                   /*  与 FCCLK 相同，或是其的偶数倍 */
#define FPCLK       (FCCLK / 4)     /*  外设时钟频率,FCCLK 的 1/2、1/4*/
                                   /*  或与 FCCLK 相同      */
#define  NVIC_PRIORTY_UART0          0x04
char Rec_date[],Rec_len=0;
/*
按默认值初始化串口 0 的引脚和通讯参数。设置为 8 位数据位，1 位停止位，无奇偶校
验
*/
void UART0_Init (int UART0_BPS)
{
    uint16_t usFdiv;
    /* UART0 */
    LPC_PINCON->PINSEL0 |= (1 << 4);     /* Pin P0.2 used as TXD0 (Com0) */
    LPC_PINCON->PINSEL0 |= (1 << 6);     /* Pin P0.3 used as RXD0 (Com0) */
    LPC_UART0->LCR  = 0x83;             /* 允许设置波特率             */
    usFdiv = (FPCLK / 16) / UART0_BPS;     /* 设置波特率                 */
    LPC_UART0->DLM  = usFdiv / 256;
    LPC_UART0->DLL  = usFdiv % 256;
    LPC_UART0->LCR  = 0x03;            /* 锁定波特率                 */
        LPC_UART0->FCR  = 0x07 | (0x02 <<6);        //时能 FIFO,8 字节触发
    LPC_UART0->IER  = 0x01; //0x03;//接收中断，发送中断
        NVIC_EnableIRQ(UART0_IRQn);
    NVIC_SetPriority(UART0_IRQn, NVIC_PRIORTY_UART0);
}
/*
```

```
从串口 0 发送数据
*/
int UART0_SendByte (int ucData)
{
    while (!(LPC_UART0->LSR & 0x20));
    return (LPC_UART0->THR = ucData);
}

/*
串口 0 读字符串
*/
int UART0_GetChar (void)
{
    while (!(LPC_UART0->LSR & 0x01));
    return (LPC_UART0->RBR);
}

/*
向串口发送字符串
*/
void UART0_SendString (unsigned char *s)
{
    while (*s != 0)
    {
        UART0_SendByte(*s++);
    }
}

//中断程序
void UART0_IRQHandler()
{
        volatile uint32_t iir,count,i;
        count = 16;
        iir = LPC_UART0->IIR;

        switch(iir &0x0F)
        {
                case 0x02:  //发送中断
                    break;
                case 0x04:  //接收中断
                    //break;
                case 0x0C:  //接收超时中断
                    if(LPC_UART0->LSR & 0x01)  // Rx FIFO is not empty
                        Rec_date[Rec_len++] = LPC_UART0->RBR;
```

```
                break;
                default:
                    break;

        }
}

/*
 拆出数字的第 i 位
*/
char Num2SingleNum(char i,int num)
{
    char ans;
    while(i--)
    {
        ans=num/10;
        num=num%10;
    }
    return ans;
}
void UART0_SendNum(char len,uint32_t disp)
{
    uint16_t dispbuf[20];
    uint8_t i;

    for(i=0;i<len;i++)
        dispbuf[i]=Num2SingleNum(i,disp);

    for(i=0;i<4;i++)
        UART0_SendByte(dispbuf[i]);
}
```

1.3PID 函数

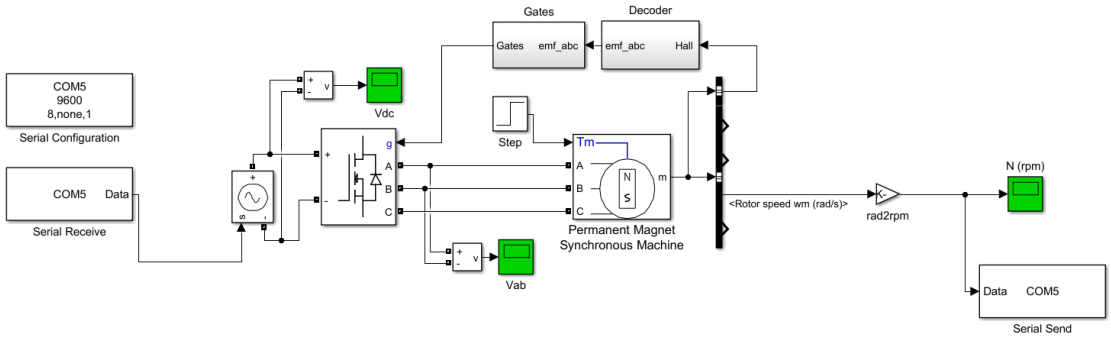
PID 函数主要用于根据系统的误差，利用比例、积分、微分计算出控制量进而控制电机转速。PID 函数程序如下：

```
#include "LPC17xx.h"
#include"PID.h"

extern char flag_run;                                   //电机运行标志
//extern int Velocity_bias, Velocity_goal, Velocity_Integral;

extern int Velocity_goal;
extern int PID_P,PID_I,PID_D,T;                 //PID 参数
```

```
int PID_Velocity(int Velocity_Least,int Velocity_goal)
{
    int offset=0;
    static                                                    int
Velocity_Integral=0,last_Velocity_bias=0,Velocity_bias=0,Velocity_PWM=0;
//静态变量，数据会保存到下一次调用函数
    last_Velocity_bias=Velocity_bias;
          //保存上一次的误差值
    Velocity_bias =  Velocity_goal - Velocity_Least ;
    Velocity_Integral += Velocity_bias ;      //===积分出位移 积分时间：5ms
    if(Velocity_Integral>3600000)   Velocity_Integral=360000;   //===积分限
幅
    if(Velocity_Integral<-3600000)      Velocity_Integral=-360000;  //===积
分限幅
    offset  =  Velocity_bias  *  PID_P  +  Velocity_Integral  *  PID_I  +
(Velocity_bias -last_Velocity_bias)* PID_D;  //===速度 PID 控制器
    if(flag_run ==0)
    {
        Velocity_Integral=0;                    //===电机关闭后清除积分
        Velocity_PWM=0;
    }
    return offset;
}
```

2、 模拟电机控制系统

模拟电机控制系统采用 matlab 的 simulink，如下图 1 所示，该模型在 simulink 自带的无刷电机模型上改进而成。

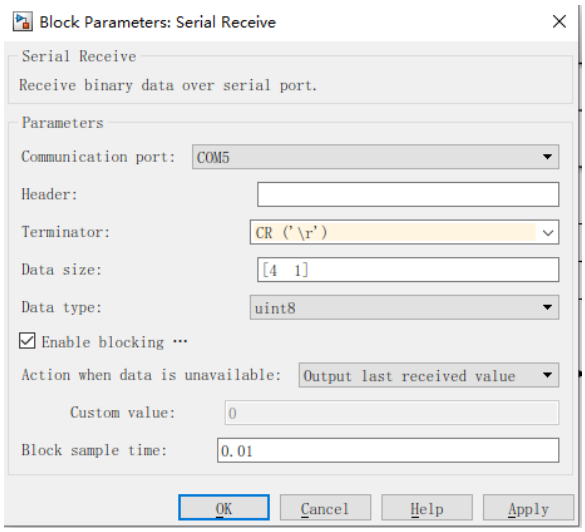


主要讲下其中的几个重要模块。

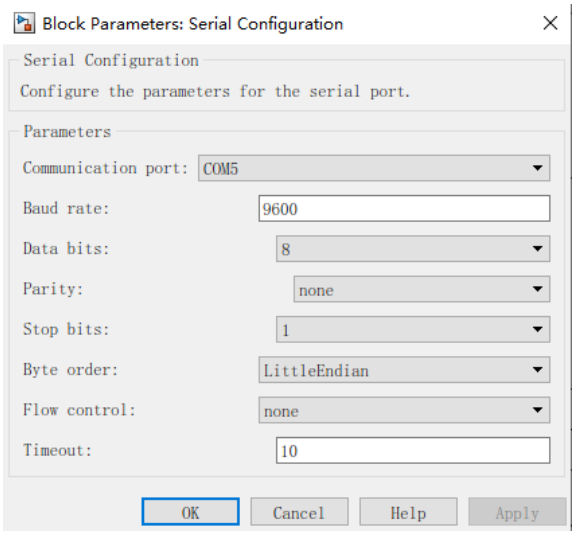
2.1 serial receive

Header 留空。Terminator 是串口输入的结尾控制符,也就是每个字符串都以换行符结尾。在这里选择 CR('\r')，表示每组数据读到结尾控制符，但不包含该控制符，也就是在 CR('\r')之前的制定位数的数据会被读取。Data size 要定义一个临时存放数据的矩阵。Enable blocking mode 勾选的情况下表示,读取输入数据时暂停其他模块。如果勾选,Serial Receive 模块只有一个 Output 为 Data，读取过程大概是这样，从 Arduino 发出的数据会不停的累积存到电脑缓存区。Serial Receive 模块会按照数据进入的顺序读取。因为其他的

操作只能在两次采样之间进行，所以这时读取数据不是实时的，而是有延迟的，随着时间延时越来越长。如果不勾选，则有两个 Output 为 Data 和 Status，status 只有两种值 1 或 0,1 表示有可用的数据，0 表是没有可用的数据。因为不会暂停其他模块，模块意外的操作（模拟）可以看做是连续的，所以现在是实时的。Action when data is unavailable（没有可读数据时的行为），有两个选项,Output last received value（可理解为输出最后进入缓存的数据）和 Out put Custom value(输出自定义值，选择该项时，可在下面的 Custom Value 进行设置，默认为零)。设置完成后会出现提示框，是否立刻生成串口配置模块(Serial Configuration)，该模块用通用的串口配置(Serial Receive 和 Serial Send)。

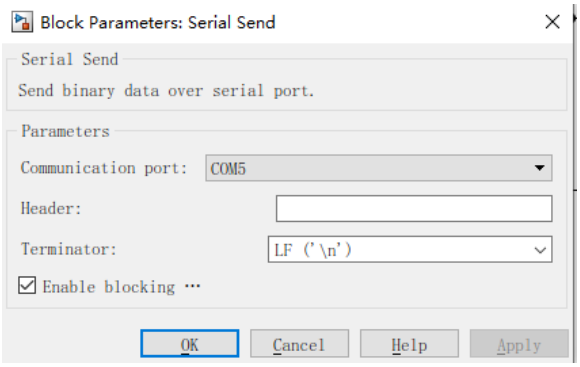


设置完成后，将会生成 Serial Configuration，打开 Serial Configuration 可以进行波特率，串口号的设置，如下图。



2.2 serial send

Simulink 中的电机转速通过 serial send 发送到 LPC1768 中进行 PID 计算，serial send 配置如下图



3、 状态体现

由于未做上位机，设定的速度直接在单片机程序内设定，故该模型电机转速直接用 simulink 自带的转速显示模块显示，如下图。