作者：粤嵌.温子祺

一、概述

FreeMODBUS 是针对通用的Modbus协议栈在嵌入式系统中应用的一个实现。Modbus协议是一个在工业制造领域中得到广泛应用的一个通讯协议。

在FreeMODBUS的当前版本中，提供了Modbus Application Protocol v1.1a 的实现并且支持在Modbus over serial line specification 1.0中定义的RTU/ASCII传输模式。从0.7版本开始，FreeModbus也支持在TCP defined in Modbus Messaging on TCP/IP Implementation Guide v1.0a中定义的TCP传输。Freemodbus遵循BSD ，这意味着本协议栈的实现代码可以应用于商业用途。

官网：
https://www.embedded-solutions.at/en/freemodbus/

下载地址：

https://github.com/cwalter-at/freemodbus

1.1 支持的功能码

目前版本的FreeModbus支持如下的功能码：

- 读输入寄存器 (0x04)
- 读保持寄存器 (0x03)
- 写单个寄存器 (0x06)
- 写多个寄存器 (0x10)
- 读/写多个寄存器 (0x17)
- 读取线圈状态 (0x01)
- 写单个线圈 (0x05)
- 写多个线圈 (0x0F)
- 读输入状态 (0x02)
- 报告从机标识 (0x11)

1.2 硬件需求

FreeModbus协议对硬件的需求非常少——基本上任何具有串行接口，并且有一些能够容纳modbus数据帧的RAM的微控制器都足够了。

- 一个异步串行接口，能够支持接收缓冲区满和发送缓存区空中断。
- 一个能够产生RTU传输所需要的t3.5字符超时定时器的时钟。

对于软件部分，仅仅需要一个简单的事件队列。在使用操作系统的处理器上，可通过单独定义一个任务完成Modbus时间的查询。小点的微控制器往往不允许使用操作系统，在那种情况下，可以使用一个全局变量来实现该事件队列。

实际的存储器需求决定于所使用的Modbus模块的多少，功能码可以在头文件mbconfig.h中进行配置。

1.3 文件结构

FreeModbus\modbus\mb.c
给应用层提供Modbus从机设置及轮询相关接口

FreeModbus\modbus\mb_m.c
给应用层提供Modbus主机设置及轮询相关接口

FreeModbus\modbus\ascii\mbascii.c
ASCII模式设置及其状态机

FreeModbus\modbus\functions\mbfunccoils.c
从机线圈相关功能

FreeModbus\modbus\functions\mbfunccoils_m.c
主机线圈相关功能

FreeModbus\modbus\functions\mbfuncdisc.c
从机离散输入相关功能

FreeModbus\modbus\functions\mbfuncdisc_m.c
主机离散输入相关功能

FreeModbus\modbus\functions\mbfuncholding.c
从机保持寄存器相关功能

FreeModbus\modbus\functions\mbfuncholding_m.c
主机保持寄存器相关功能

FreeModbus\modbus\functions\mbfuncinput.c
从机输入寄存器相关功能

FreeModbus\modbus\functions\mbfuncinput_m.c
主机输入寄存器相关功能

FreeModbus\modbus\functions\mbfuncother.c
其余Modbus功能

FreeModbus\modbus\functions\mbutils.c
一些协议栈中需要用到的小工具

FreeModbus\modbus\rtu\mbcrc.c
CRC校验功能

FreeModbus\modbus\rtu\mbrtu.c
从机RTU模式设置及其状态机

FreeModbus\modbus\rtu\mbrtu_m.c
主机RTU模式设置及其状态机

FreeModbus\modbus\tcp\mbtcp.c
TCP模式设置及其状态机

FreeModbus\port\port.c
实现硬件移植部分接口

FreeModbus\port\portevent.c
实现从机事件移植接口

FreeModbus\port\portevent_m.c
实现主机事件及错误处理移植接口

FreeModbus\port\portserial.c
从机串口移植

FreeModbus\port\portserial_m.c
主机串口移植

FreeModbus\port\porttimer.c
从机定时器移植

FreeModbus\port\porttimer_m.c
主机定时器移植

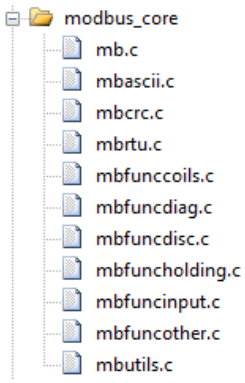
FreeModbus\port\user_mb_app.c
定义从机数据缓冲区，实现从机Modbus功能的回调接口

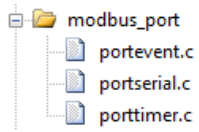
FreeModbus\port\user_mb_app_m.c
定义主机数据缓冲区，实现主机Modbus功能的回调接口
注：所有带_m后缀的文件为主机模式下必须使用的文件，如使用从机模式则无需这些文件。

二、移植

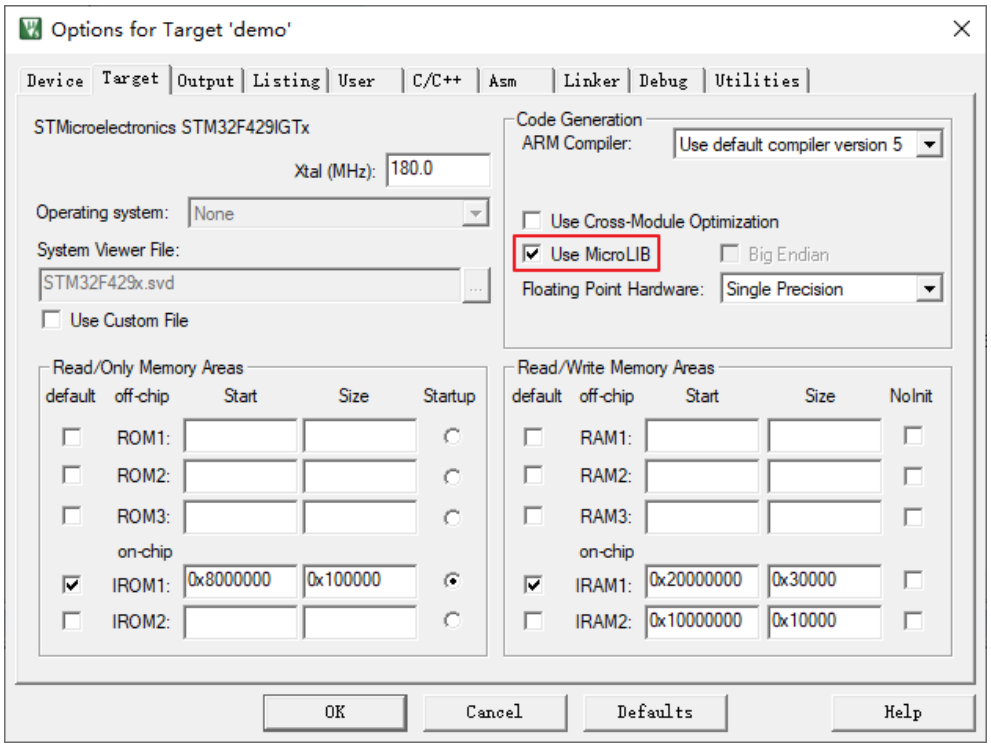
2.1 源码添加与工程配置

1.将modbus源码添加到工程





2.由于CubeMx导出的代码默认使用了“MicroLib”，则需要将assert语句注释；否则断言语句会报错。



将modbus源码多处出现的assert代码，注释如下即可。

```
//    assert( ucNBits <= 8 );
//    assert( ( size_t )BITS_UCHAR == sizeof( UCHAR ) * 8 );
```

2.2 底层文件

物理层接口文件的修改,用户只需完成串行口及超时定时器的配置即可。具体应修改接口文件portserial.c及porttimer.c。

2.2.1 port.h

设置变量类型、临界区代码保护开关、内联函数关键字等。

```
#ifndef _PORT_H
#define _PORT_H

#include <assert.h>
#include <inttypes.h>
#include "stm32f4xx_hal.h"
#define INLINE inline
#define PR_BEGIN_EXTERN_C extern "C" {
#define PR_END_EXTERN_C }

 #define ENTER_CRITICAL_SECTION( ) __set_PRIMASK(1)
 #define EXIT_CRITICAL_SECTION( ) __set_PRIMASK(0)

 typedef uint8_t BOOL;

 typedef unsigned char UCHAR;
 typedef char CHAR;

 typedef uint16_t USHORT;
```

```
typedef int16_t SHORT;

typedef uint32_t ULONG;
typedef int32_t LONG;

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#endif
```

2.2.2 portserial.c

1. 设置串口状态

```
void
vMBPortSerialEnable( BOOL xRxEnable, BOOL xTxEnable )
{
 /* If xRXEnable enable serial receive interrupts. If xTxENable enable
 * transmitter empty interrupts.
 */
 /* 使用了485芯片，同一时刻只能接收或者发送 */
 if (xRxEnable)
 {
  __HAL_UART_ENABLE_IT(&huart2,UART_IT_RXNE);
  }
  else
  {
  __HAL_UART_DISABLE_IT(&huart2,UART_IT_RXNE);
  }

  if (xTxEnable)
  {
  __HAL_UART_ENABLE_IT(&huart2,UART_IT_TXE);
  }
  else
  {
  __HAL_UART_DISABLE_IT(&huart2,UART_IT_TXE);
  }
}
```

此函数的功能为设置串口状态。有两个参数：xRxEnable及xTxEnable。当xRxEnable为真时，应使能串口接收及接收中断。在RS485通讯系统中，还要注意将RS485接口芯片设为接收使能状态；当xTxEnable为真时，应使能串口发送及发送中断。在RS485通讯系统中，还要注意将RS485接口芯片设为发送使能状态。

2. 发送一字节数据

```
BOOL
```

```
xMBPortSerialPutByte( CHAR ucByte )
{
 /* Put a byte in the UARTs transmit buffer. This function is called
 * by the protocol stack if pxMBFrameCBTransmitterEmpty( ) has been
 * called. */

 if(HAL_UART_Transmit(&huart2 ,(uint8_t *)&ucByte,1,0x01) != HAL_OK )
 {
  return FALSE ;
  }
  else
  {
  return TRUE;
  }
}
```

此函数的功能为通讯端口发送一字节数据。参数为：ucByte，待发送的数据。应在此函数中编写发送一字节数据的函数。注意，由于使用的是中断发送，故只需将数据放到发送寄存器即可。函数返回值务必为TRUE。

3. 获取一字节数据

```
BOOL
xMBPortSerialGetByte( CHAR * pucByte )
{
 /* Return the byte in the UARTs receive buffer. This function is called
 * by the protocol stack after pxMBFrameCBByteReceived( ) has been called.
 */
 if(HAL_UART_Receive (&huart2 ,(uint8_t *)pucByte,1,0x01) != HAL_OK )
 {
 return FALSE ;
  }
  else
  {
  return TRUE;
  }
}
```

2.2.3 porttimer.c

1. 初始化超时定时器

```
BOOL
xMBPortTimersInit( USHORT usTim1Timerout50us )
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};

 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 9000-1;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = usTim1Timerout50us*50-1;//t3.5字符超时定时器的时钟
```

```
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
  Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
  Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
  Error_Handler();
  }
  return TRUE;
}
```

此函数的功能为初始化超时定时器。参数为：usTim1Timerout50us，50us的个数。用户应根据所使用的硬件初始化超时定时器，使之能产生中断时间为usTim1Timerout50us*50us的中断。函数返回值务必为TRUE。

2. 使能超时定时器

```
void
vMBPortTimersEnable( )
{
  /* Enable the timer with the timeout passed to xMBPortTimersInit( ) */
  __HAL_TIM_CLEAR_IT(&htim2,TIM_IT_UPDATE);
  __HAL_TIM_SetCounter(&htim2,0);

  /* 在中断模式下启动定时器 */
  HAL_TIM_Base_Start_IT(&htim2);
}
```

此函数的功能为使能超时定时器。用户需在此函数中清除中断标志位、清零定时器计数值，并重新使能定时器中断。

3. 关闭超时定时器

```
void
vMBPortTimersDisable( )
{
  /* Disable any pending timers. */
  HAL_TIM_Base_Stop_IT(&htim2);

  __HAL_TIM_SetCounter(&htim2,0);
  __HAL_TIM_CLEAR_IT(&htim2,TIM_IT_UPDATE);
}
```

此函数的功能为关闭超时定时器。用户需在此函数中清零定时器计数值，并关闭定时器中断。

2.2.4 mbrtu.c

1.发送数据时，需设置RS485为发送模式。

```
BOOL
xMBRTUTransmitFSM( void )
{
 ..................................
 //设置为发送模式
 HAL_GPIO_WritePin(GPIOD, GPIO_PIN_11, GPIO_PIN_SET);

 xMBPortSerialPutByte( ( CHAR )*pucSndBufferCur );

 //设置为接收模式
 HAL_GPIO_WritePin(GPIOD, GPIO_PIN_11, GPIO_PIN_RESET);
 ..................................
}
```

2.2.5 stm32f4xx_it.c

1.定时器2中断服务函数

```
void TIM2_IRQHandler(void)
{

 /* USER CODE BEGIN TIM2_IRQn 1 */
 if(__HAL_TIM_GET_FLAG(&htim2, TIM_FLAG_UPDATE)) // 更新中断标记被置位
 {
 __HAL_TIM_CLEAR_FLAG(&htim2, TIM_FLAG_UPDATE); // 清除中断标记
 prvvTIMERExpiredISR(); // 通知modbus3.5个字符等待时间到
 }

 /* USER CODE END TIM2_IRQn 1 */
}
```

定时器中断函数。此函数无需修改。只需在用户的定时器中断中调用此函数即可，同时，用户应在调用此函数后清除中断标志位。

2.串口2中断处理函数

```
void USART2_IRQHandler(void)
{
 /* USER CODE BEGIN USART2_IRQn 0 */
 if(__HAL_UART_GET_IT_SOURCE(&huart2, UART_IT_RXNE)!= RESET)
 {

 prvvUARTRxISR(); //接收完成中断
 __HAL_UART_CLEAR_FLAG(&huart2, UART_FLAG_RXNE);
 }

 if(__HAL_UART_GET_IT_SOURCE(&huart2, UART_IT_TXE)!= RESET)
```

```
{
prvvUARTTxReadyISR(); //发送完成中断
__HAL_UART_CLEAR_FLAG(&huart2, UART_FLAG_TXE);
}

HAL_NVIC_ClearPendingIRQ(USART2_IRQn);
/* USER CODE END USART2_IRQn 0 */

}
```

void prvvUARTTxReadyISR(void)
发送中断函数。此函数无需修改。只需在用户的发送中断函数中调用此函数即可，同时，用户应在调用此函数后，清除发送中断标志位。

void prvvUARTRxISR(void)
接送中断函数。此函数无需修改。只需在用户的接收中断函数中调用此函数即可，同时，用户应在调用此函数后，清除接收中断标志位。

2.3 应用层

2.3.1 函数描述

在应用层，用户需要定义所需要使用的寄存器，并修改对应的回函数。回函数有如下几个：
1) eMBErrorCode eMBRegInputCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs)
输入寄存器回调函数。* pucRegBuffer为要添加到协议中的数据，usAddress为输入寄存器地址，usNRegs为要读取寄存器的个数。用户应根据要访问的寄存器地址usAddress将相应输入寄存器的值按顺序添加到pucRegBuffer中。

2) eMBErrorCode eMBRegHoldingCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs, eMBRegisterMode eMode)
保持寄存器回调函数。* pucRegBuffer为要协议中的数据，usAddress为输入寄存器地址，usNRegs为访问寄存器的个数，eMode为访问类型（MB_REG_READ为读保持寄存器，MB_REG_WRITE为写保持寄存器）。用户应根据要访问的寄存器地址usAddress将相应输入寄存器的值按顺序添加到pucRegBuffer中，或将协议中的数据根据要访问的寄存器地址usAddress放到相应保持寄存器中。

3) eMBErrorCode eMBRegCoilsCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils, eMBRegisterMode eMode)
读写线圈回调函数。* pucRegBuffer为要添加到协议中的数据，usAddress为线圈地址，usNCoils为要访问线圈的个数，eMode为访问类型（MB_REG_READ为读线圈状态，MB_REG_WRITE为写线圈）。用户应根据要访问的线圈地址usAddress将相应线圈的值按顺序添加到pucRegBuffer中，或将协议中的数据根据要访问的线圈地址usAddress放到相应线圈中。

4) eMBErrorCode eMBRegDiscreteCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete)

读离散线圈回调函数。* pucRegBuffer为要添加到协议中的数据，usAddress为线圈地址，usNDiscrete为要访问线圈的个数。用户应根据要访问的线圈地址usAddress将相应线圈的值按顺序添加到pucRegBuffer中。

2.3.2 实现过程

```
#include "mb.h"
#include "mbport.h"

// 十路输入寄存器
#define REG_INPUT_SIZE 10
uint16_t REG_INPUT_BUF[REG_INPUT_SIZE];


// 十路保持寄存器
 #define REG_HOLD_SIZE 10
 uint16_t REG_HOLD_BUF[REG_HOLD_SIZE];


 // 十路线圈
 #define REG_COILS_SIZE 10
 uint8_t REG_COILS_BUF[REG_COILS_SIZE];


 // 十路离散量
 #define REG_DISC_SIZE 10
 uint8_t REG_DISC_BUF[10];

 int main(void)
 {
  eMBInit(MB_RTU, 0x01, 3, 9600, MB_PAR_NONE);

  /* Enable the Modbus Protocol Stack. */
  eMBEnable();

  while(1)
  {
  (void)eMBPoll();
  }
 }

 // CMD4
 eMBErrorCode eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
 {
  USHORT usRegIndex = usAddress - 1;

  // 非法检测
  if((usRegIndex + usNRegs) > REG_INPUT_SIZE)
  {
  return MB_ENOREG;
  }

  // 循环读取
```

```
 while( usNRegs > 0 )
 {
 *pucRegBuffer++ = ( unsigned char )( REG_INPUT_BUF[usRegIndex] >> 8 );
 *pucRegBuffer++ = ( unsigned char )( REG_INPUT_BUF[usRegIndex] & 0xFF );
 usRegIndex++;
 usNRegs--;
 }

 // 模拟输入寄存器被改变
 for(usRegIndex = 0; usRegIndex < REG_INPUT_SIZE; usRegIndex++)
 {
 REG_INPUT_BUF[usRegIndex]++;
 }

 return MB_ENOERR;
}

// CMD6、3、16
 eMBErrorCode eMBRegHoldingCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs, eMBRegisterMode eMode )
{
 USHORT usRegIndex = usAddress - 1;

 // 非法检测
 if((usRegIndex + usNRegs) > REG_HOLD_SIZE)
 {
 return MB_ENOREG;
 }

 // 写寄存器
 if(eMode == MB_REG_WRITE)
 {
 if(pucRegBuffer[1] ==0)
 {
 HAL_GPIO_WritePin(GPIOH, GPIO_PIN_10|GPIO_PIN_11|GPIO_PIN_12, GPIO_PIN_SET);
 }

 if(pucRegBuffer[1] ==1)
 {
 HAL_GPIO_WritePin(GPIOH, GPIO_PIN_11|GPIO_PIN_12, GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOH, GPIO_PIN_10, GPIO_PIN_RESET);
 }

 if(pucRegBuffer[1] ==2)
 {
 HAL_GPIO_WritePin(GPIOH, GPIO_PIN_12, GPIO_PIN_SET);
 HAL_GPIO_WritePin(GPIOH, GPIO_PIN_10|GPIO_PIN_11, GPIO_PIN_RESET);
 }

 if(pucRegBuffer[1] ==3)
 {
 HAL_GPIO_WritePin(GPIOH, GPIO_PIN_10|GPIO_PIN_11|GPIO_PIN_12, GPIO_PIN_RESET);
```

```
}


 while( usNRegs > 0 )
 {
 REG_HOLD_BUF[usRegIndex] = (pucRegBuffer[0] << 8) | pucRegBuffer[1];
 pucRegBuffer += 2;
 usRegIndex++;
 usNRegs--;
 }
 }

 // 读寄存器
 else
 {
 while( usNRegs > 0 )
 {
 *pucRegBuffer++ = ( unsigned char )( REG_HOLD_BUF[usRegIndex] >> 8 );
 *pucRegBuffer++ = ( unsigned char )( REG_HOLD_BUF[usRegIndex] & 0xFF );
 usRegIndex++;
 usNRegs--;
 }
 }

 return MB_ENOERR;
}

// CMD1、5、15
eMBErrorCode eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils, eMBRegisterMode eMode )
{
 USHORT usRegIndex = usAddress - 1;
 USHORT usCoilGroups = ((usNCoils - 1) / 8 + 1);
 UCHAR ucStatus = 0;
 UCHAR ucBits = 0;
 UCHAR ucDisp = 0;

 // 非法检测
 if((usRegIndex + usNCoils) > REG_COILS_SIZE)
 {
 return MB_ENOREG;
 }

 // 写线圈
 if(eMode == MB_REG_WRITE)
 {
 while(usCoilGroups--)
 {
 ucStatus = *pucRegBuffer++;
 ucBits = 8;
 while((usNCoils--) != 0 && (ucBits--) != 0)
 {
```

```
 REG_COILS_BUF[usRegIndex++] = ucStatus & 0X01;
 ucStatus >>= 1;
 }
 }
 }

 // 读线圈
 else
 {
 while(usCoilGroups--)
 {
 ucDisp = 0;
 ucBits = 8;
 while((usNCoils--) != 0 && (ucBits--) != 0)
 {
 ucStatus |= (REG_COILS_BUF[usRegIndex++] << (ucDisp++));
 }
 *pucRegBuffer++ = ucStatus;
 }
 }
 return MB_ENOERR;
}


// CMD4
eMBErrorCode eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
{
 USHORT usRegIndex = usAddress - 1;
 USHORT usCoilGroups = ((usNDiscrete - 1) / 8 + 1);
 UCHAR ucStatus = 0;
 UCHAR ucBits = 0;
 UCHAR ucDisp = 0;

 // 非法检测
 if((usRegIndex + usNDiscrete) > REG_DISC_SIZE)
 {
 return MB_ENOREG;
 }

 // 读离散输入
 while(usCoilGroups--)
 {
 ucDisp = 0;
 ucBits = 8;
 while((usNDiscrete--) != 0 && (ucBits--) != 0)
 {
 if(REG_DISC_BUF[usRegIndex])
 {
 ucStatus |= (1 << ucDisp);
 }
 ucDisp++;
 }
```

```
202   *pucRegBuffer++ = ucStatus;
203   }
204
205   // 模拟改变
206   for(usRegIndex = 0; usRegIndex < REG_DISC_SIZE; usRegIndex++)
207   {
208   REG_DISC_BUF[usRegIndex] = !REG_DISC_BUF[usRegIndex];
209   }
210
211   return MB_ENOERR;
212 }
```

2.4 演示验证

通过RS485发送

功能码，04，读线圈

```
1  “01 04 00 00 00 02 71 CB”,
```

功能码，06，写保持寄存器

```
1  “01 06 00 01 00 01 19 CA”, //点亮一盏LED
2  “01 06 00 01 00 02 59 CB”, //点亮二盏LED
3  “01 06 00 01 00 03 98 0B”, //点亮三盏LED
4  “01 06 00 01 00 00 D8 0A”, //熄灭所有LED
```

若能够成功返回数据，则移植成功，演示如下：