一、MODBUS 简介

Modbus 是由 Modicon（现为施耐德电气公司的一个品牌）在 1979 年发明的，是全球第一个真正用于工业现场的总线协议。

ModBus 网络只有一个主机，所有通信都由他发出。网络可支持 247 个之多的远程从属控制器，但实际所支持的从机数要由所用通信设备决定。主机一般使用 PC 机，从机一般为单片机、PLC 等设备。

1.1 通信模式

在 ModBus 系统中有 2 种传输模式可选择。。选择时应视所用 ModBus 主机而定，每个 ModBus 系统只能使用一种模式，不允许 2 种模式混用。一种模式是 ASCII（美国信息交换码），另一种模式是 RTU（远程终端设备）。下面简单说一下两种模式的区别，即使你没有彻底搞懂也无大碍，并不影响你做本实验。等需要你必须搞懂时你再去看相关资料也不迟。

ASCII 模式：当控制器设为在 Modbus 网络上以 ASCII（美国标准信息交换代码）模式通信，在消息中的每个 8Bit 字节都作为一个 ASCII 码（两个十六进制字符）发送。这种方式的主要优点是字符发送的时间间隔可达到 1 秒而不产生错误。

代码系统

· 十六进制，ASCII 字符 0...9,A...F

· 消息中的每个 ASCII 字符都是一个十六进制字符组成

每个字节的位

· 1 个起始位

· 7 个数据位，最小的有效位先发送

· 1 个奇偶校验位，无校验则无

1 个停止位（有校验时），2 个 Bit（无校验时）

错误检测域

· LRC(纵向冗长检测)

RTU 模式：当控制器设为在 Modbus 网络上以 RTU（远程终端单元）模式通信，在消息中的每个 8Bit 字节包含两个 4Bit 的 十六进制字符。这种方式的主要优点是：在同样的波特率下，可比 ASCII 方式传送更多的数据。

代码系统

8 位二进制，十六进制数 0...9，A...F

消息中的每个 8 位域都是一或两个十六进制字符组成

每个字节的位

1 个起始位

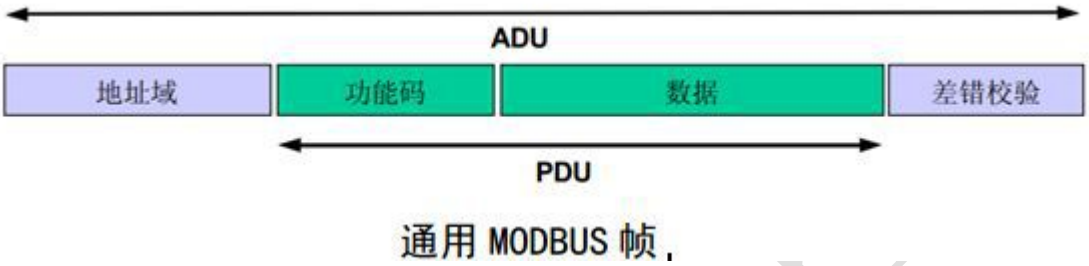
8 个数据位，最小的有效位先发送

1 个奇偶校验位，无校验则无

1 个停止位（有校验时），2 个 Bit（无校验时）

二、MODBUS 数据帧的介绍

MODBUS 的工作模式是应答式的，由主机提出问题或发出命令，从机来回答问题或执行命令。其数据帧（报文）一般分为 4 个部分一次是：从机地址、功能码（功能号）、数据、校验值。如图 2.1 所示，图中 ADU 是“应用数据单元”的意思，PDU 是“协议数据单元”的意思。



通用 MODBUS 帧

图 2.1

1.从机地址：每个从机都有一个唯一的地址，相当于居民身份证号或者计算机中的 IP 地址。从机在接收到数据帧后会根据从机地址来判断当前数据是否是发给自己的，如果不是发给自己的就不做处理，如果是发给自己的就进行回应。

2.功能号：功能号是用来表示所要执行的功能，通俗的说就是主机要求从机干什么。Modbus 对功能号有具体的定义，如表 2.1 所示。

功能号	名称	作用
01	读取线圈状态	取得一组逻辑线圈的当前状态（ON/OFF)
02	读取输入状态	取得一组开关输入的当前状态（ON/OFF)
03	读取保持寄存器	在一个或多个保持寄存器中取得当前的二进制值
04	读取输入寄存器	在一个或多个输入寄存器中取得当前的二进制值
05	强置单线圈	强置一个逻辑线圈的通断状态
06	预置单寄存器	把具体二进值装入一个保持寄存器
07	读取异常状态	取得 8 个内部线圈的通断状态，这 8 个线圈的地址由控制器决定
08	回送诊断校验	把诊断校验报文送从机，以对通信处理进行评鉴
09	编程（只用于 484）	使主机模拟编程器作用，修改 PC 从机逻辑
10	控询（只用于 484）	可使主机与一台正在执行长程序任务从机通信，探询该从机是否已完成其操作任务，仅在含有功能码 9 的报文发送后，本功能码才发送
11	读取事件计数	可使主机发出单询问，并随即判定操作是否成功，尤其是该命令或其他应答产生通信错误时
12	读取通信事件记录	可是主机检索每台从机的 ModBus 事务处理通信事件记录。如果某项事务处理完成，记录会给出有关错误
13	编程（184/384 484 584）	可使主机模拟编程器功能修改 PC 从机逻辑
14	探询（184/384 484 584）	可使主机与正在执行任务的从机通信，定期控询该从机是否已完成其程序操作，仅在含有功能 13 的报文发送后，本功能码才得发送
15	强置多线圈	强置一串连续逻辑线圈的通断
16	预置多寄存器	把具体的二进制值装入一串连续的保持寄存器
17	报告从机标识	可使主机判断编址从机的类型及该从机运行指示灯的状态

18	（884 和 MICRO 84）	可使主机模拟编程功能，修改 PC 状态逻辑
19	重置通信链路	发生非可修改错误后，是从机复位于已知状态，可重置顺序字节
20	读取通用参数（584L）	显示扩展存储器文件中的数据信息
21	写入通用参数（584L）	把通用参数写入扩展存储文件，或修改之
22～64	保留作扩展功能备用	
65～72	保留以备用户功能所用	留作用户功能的扩展编码
73～119	非法功能	
120～127	保留	留作内部作用
128～255	保留	用于异常应答

表 2.1

3.数据：数据内容会因命令码的不同而不同。

4.校验值：数据在传输过程中有可能出错，校验值就是用来检测是否出错的。

三、关于实验

实验开始前首先要 USB 转 485 线连接 PC 机和开发板。然后将实验代码下载到开发板中。如果有多个从机，那么各从机的地址域必须唯一。从机地址在协议栈初始化过程中赋值。图 3.1 中第 52 行便是协议栈的初始化，eMBInit 函数的第 3 个参数就是从机地址。实验中我们只用了一个从机，从机地址设为 1。

```
48  int main(void)
49  {
50   LedInit();
51   KeyInit();  //-----按键端口初始化------------
52   eMBInit(MB_RTU, 0x01, 0x01, 9600, MB_PAR_NONE); //初始化 RTU模式 从机地址为1 USART1 9600 无校验
53   eMBEnable(); //启动FreeModbus
54   while(1)
55   {
56     eMBPoll();
57     LED_Poll();
58     Button_Poll();
59   }
60  }
```

图 3.1

打开软件 ECOMV280（已收录到“配套软件”文件夹中）。进行相关配置后如图 3.2 所示。注意：串口号需要根据实际情况调整。

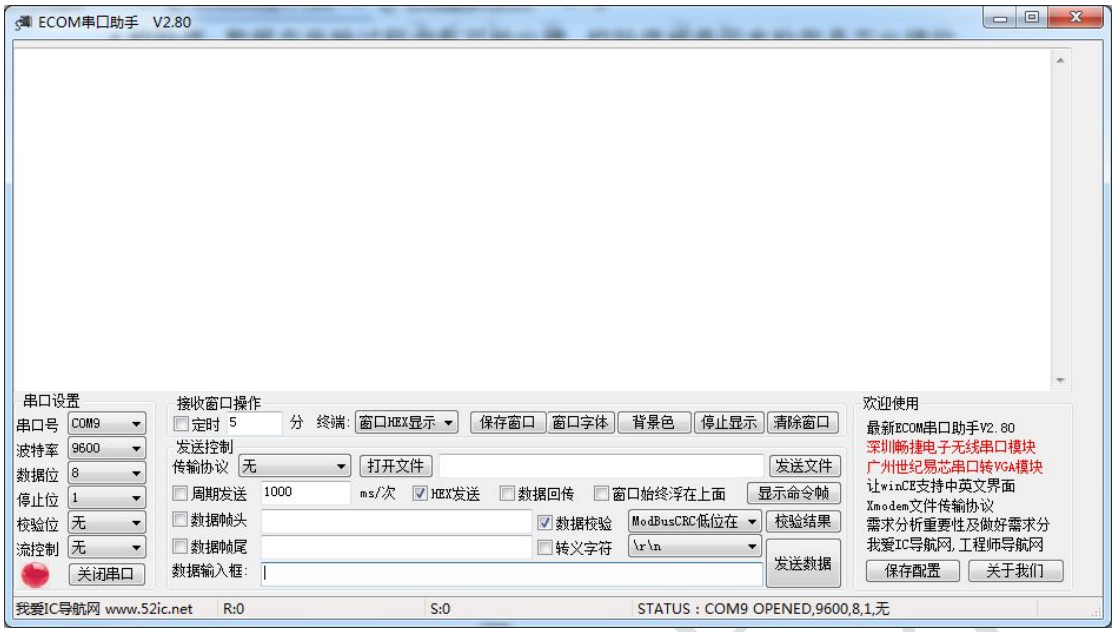


图 3.2

3.1 测试功能码 2

数字量输入采集指令：（采集板子上 3 个按键的状态）

发送：01 02 00 00 00 03 38 0B 十六进制

注意：发送时在数据框中只填入红色部分数据，蓝色部分为 CRC 校验，软件自动添加。

数据	字节数	数据说明	备注
01	1	模块地址	地址范围 01-FE
02	1	功能码	02-读取输入状态
0000	2	输入地址	0000-输入起始位地址
0003	2	读取输入位长度	0003-读取 3 个输入位
380B	2	CRC 校验码	前面所有数据 CRC 计算后得到的值

备注：如果没有板子按键按下，收到 01 02 01 00 A1 88

板子上按键 S2 按下，收到 01 02 01 01 60 48

板子上按键 S3 按下，收到 01 02 01 02 20 49

读取的数据是一个字节，转化为二进制为 8 个位 0000 0000 我们的 3 个按键的状态对应为：S2 对应 D0 位，S3 对应 D1 位，如果按键按下对应的位为 1，如果按键没有按下对应的位为 0。

3.2 测试功能码 15

发送：01 0F 00 00 00 03 01 00 8F 57 十六进制

数据	字节	数据说明	备注
01	1	模块地址	地址范围 01-FE
0F	1	功能码	强置多线圈
0000	2	输入地址	0000-线圈起始位地址
0003	2	写入线圈长度	0003-写 3 个线圈（3 个位）
01	1	写入数据字节	写入 1 个字节数据

00	1	写入数据	00-写 3 个线圈的输出状态

接收：01 0F 00 00 00 03 15 CA 十六进制

写入的一个字节数据转换成二进制为 8 个位 0000 0000，LED1 对应 DO，LED2 对应 D1, LED3 对应 D2。 “1”表示点亮 LED， “0”表示熄灭 LED。

例如

发送：01 0F 00 00 00 03 01 01 4E 97 表示点亮 LED1.

发送：01 0F 00 00 00 03 01 02 0E 96 表示点亮 LED2

发送：01 0F 00 00 00 03 01 04 8E 94 表示点亮 LED3

发送：01 0F 00 00 00 03 01 00 8F 57 表示熄灭所有 LED

四、FreeModbus 代码简单分析

协议必须首先调用初始化功能 eMBinit()函数。后调用 eMBEnable()，最后，在循环体或者单独一个任务中调用 eMBPoll()函数。

4.1eMBInit 分析

首先使用 eMBInit 初始化协议栈，根据你使用的参数 eMBMode eMode 初始化相应的函数入口。

```
#if MB_RTU_ENABLED > 0
        case MB_RTU:
            pvMBFrameStartCur = eMBRTUStart;
            pvMBFrameStopCur = eMBRTUStop;
            peMBFrameSendCur = eMBRTUSend;
            peMBFrameReceiveCur = eMBRTUReceive;
            pvMBFrameCloseCur = MB_PORT_HAS_CLOSE ? vMBPortClose : NULL;
            pxMBFrameCBByteReceived = xMBRTUReceiveFSM;
            pxMBFrameCBTransmitterEmpty = xMBRTUTransmitFSM;
            pxMBPortCBTimerExpired = xMBRTUTimerT35Expired;

            eStatus = eMBRTUInit( ucMBAddress, ucPort, ulBaudRate, eParity );

            break;
#endif
#if MB_ASCII_ENABLED > 0
        case MB_ASCII:
            pvMBFrameStartCur = eMBASCIIStart;
            pvMBFrameStopCur = eMBASCIIStop;
            peMBFrameSendCur = eMBASCIISend;
            peMBFrameReceiveCur = eMBASCIIReceive;
            pvMBFrameCloseCur = MB_PORT_HAS_CLOSE ? vMBPortClose : NULL;
            pxMBFrameCBByteReceived = xMBASCIIReceiveFSM;
            pxMBFrameCBTransmitterEmpty = xMBASCIITransmitFSM;
```

```
            pxMBPortCBTimerExpired = xMBASCIITimerT1SExpired;

            eStatus = eMBASCIIInit( ucMBAddress, ucPort, ulBaudRate, eParity
    );
            break;
#endif
```

以上代码中 pvMBFrameStartCur、pvMBFrameStopCur 等即协议栈函数的接口，对于不同模式使用不同函数进行初始化。其中 eMBRTUInit 函数对底层驱动(串口和定时器)进行了初始化。在上述初始化完成并且成功后对事件功能也进了初始化，最后全局变量 eMBState = STATE_DISABLED。

4.2eMBEnable 的分析

```
eMBEnable( void )
{
    eMBErrorCode    eStatus = MB_ENOERR;

    if( eMBState == STATE_DISABLED )
    {
        /* Activate the protocol stack. */
        pvMBFrameStartCur(  );
        eMBState = STATE_ENABLED;
    }
    else
    {
        eStatus = MB_EILLSTATE;
    }
    return eStatus;
}
```

由第一节的分析，此时将启动协议栈 pvMBFrameStartCur，查看程序该函数指针被赋值为 eMBRTUStart。该函数中将全局变量 eRcvState = STATE_RX_INIT，并使能串口和定时器，注意此时的定时开始工作。全局变量 eMBState =STATE_ENABLED。

4.3eMBPoll 的分析

在此循环函数中 xMBPortEventGet(&eEvent) == TRUE 先判断是否有事件，无事件发生则不进入状态机。还记得第二节定时器开始工作了吗？我们先看看该定时器如果超时了会发生什么事件。在超时中断中我们将会调用 pxMBPortCBTimerExpired 函数，其中有以下代码：

```
1.  BOOL
2.  xMBRTUTimerT35Expired( void )
3.  {
4.      BOOL            xNeedPoll = FALSE;
5.
6.      switch ( eRcvState )
7.      {
8.          /* Timer t35 expired. Startup phase is finished. */
9.      case STATE_RX_INIT:
10.         xNeedPoll = xMBPortEventPost( EV_READY );
11.         break;
12.
13.         /* A frame was received and t35 expired. Notify the listener that
14.          * a new frame was received. */
15.     case STATE_RX_RCV:
16.         xNeedPoll = xMBPortEventPost( EV_FRAME_RECEIVED );
17.         break;
18.
19.         /* An error occured while receiving the frame. */
20.     case STATE_RX_ERROR:
21.         break;
22.
23.         /* Function called in an illegal state. */
24.     default:
25.         assert( ( eRcvState == STATE_RX_INIT ) ||
26.                 ( eRcvState == STATE_RX_RCV ) || ( eRcvState == STATE_RX_ERR
    OR ) );
27.     }
28.
29.     vMBPortTimersDisable(  );
30.     eRcvState = STATE_RX_IDLE;
31.
32.     return xNeedPoll;
33. }
```

上一节分析中全局变量 eRcvState =STATE_RX_INIT，因此第二节所说的定时器第一次超时将会发送 xNeedPoll =xMBPortEventPost(EV_READY)事件，然后关闭定时器，全局变量 eRcvState =STATE_RX_IDLE。此时在主循环 eMBPoll 中将会执行一次 EV_READY 下的操作，之后会一直执行 eMBPoll，整个协议栈开始运行。接收数据分析由于 FreeModbus 只支持从机模式，因此我们分析一下其在接收到数据后的操作。

接收数据在上三节的操作中，我们可以知道进入 eMBPoll 循环后串口中断是开启的。因此在接收到数据的时候首先响应的应该是串口中断程序。

接收中断中将会调用接收状态机:

```
1.  BOOL
2.  xMBRTUReceiveFSM( void )
3.  {
4.      BOOL            xTaskNeedSwitch = FALSE;
5.      UCHAR           ucByte;
6.
7.      assert( eSndState == STATE_TX_IDLE );
8.
9.      /* Always read the character. */
10.     ( void )xMBPortSerialGetByte( ( CHARCHAR * ) & ucByte );
11.
12.     switch ( eRcvState )
13.     {
14.         /* If we have received a character in the init state we have to
15.          * wait until the frame is finished.
16.          */
17.     case STATE_RX_INIT:
18.         vMBPortTimersEnable(  );
19.         break;
20.
21.         /* In the error state we wait until all characters in the
22.          * damaged frame are transmitted.
23.          */
24.     case STATE_RX_ERROR:
25.         vMBPortTimersEnable(  );
26.         break;
27.
28.         /* In the idle state we wait for a new character. If a character
29.          * is received the t1.5 and t3.5 timers are started and the
30.          * receiver is in the state STATE_RX_RECEIVCE.
31.          */
32.     case STATE_RX_IDLE:
33.         usRcvBufferPos = 0;
34.         ucRTUBuf[usRcvBufferPos++] = ucByte;
35.         eRcvState = STATE_RX_RCV;
36.
37.         /* Enable t3.5 timers. */
38.         vMBPortTimersEnable(  );
39.         break;
40.
```

```
41.         /* We are currently receiving a frame. Reset the timer after
42.          * every character received. If more than the maximum possible
43.          * number of bytes in a modbus frame is received the frame is
44.          * ignored.
45.          */
46.     case STATE_RX_RCV:
47.         if( usRcvBufferPos < MB_SER_PDU_SIZE_MAX )
48.         {
49.             ucRTUBuf[usRcvBufferPos++] = ucByte;
50.         }
51.         else
52.         {
53.             eRcvState = STATE_RX_ERROR;
54.         }
55.         vMBPortTimersEnable(  );
56.         break;
57.     }
58.     return xTaskNeedSwitch;
59. }
```

经过第 3 节的分析，此时全局变量 eRcvState =STATE_RX_IDLE。接收状态机开始后，读取 UART 串口缓存中的数据，并进入 STATE_RX_IDLE 分支中存储一次数据后开启超时定时器，进入 STATE_RX_RCV 分支继续接收后续的数据，直至定时器超时。为什么要等待超时才能停止接收呢？

```
1.     /* A frame was received and t35 expired. Notify the listener that
2.      * a new frame was received. */
3.  case STATE_RX_RCV:
4.     xNeedPoll = xMBPortEventPost( EV_FRAME_RECEIVED );
5.     break;
```

可以发现接收数据时发生超时后，协议栈会发送 EV_FRAME_RECEIVED 接收完成这个信号。此时 eMBPoll 接收到此信号后会调用 eMBRTUReceive 函数。

```
1.  eMBErrorCode
2.  eMBRTUReceive( UCHARUCHAR * pucRcvAddress, UCHARUCHAR ** pucFrame, USHORTUSH
    ORT * pusLength )
3.  {
4.      BOOL            xFrameReceived = FALSE;
5.      eMBErrorCode    eStatus = MB_ENOERR;
6.
7.      ENTER_CRITICAL_SECTION(  );
```

```
8.      assert( usRcvBufferPos < MB_SER_PDU_SIZE_MAX );
9.
10.     /* Length and CRC check */
11.     if( ( usRcvBufferPos >= MB_SER_PDU_SIZE_MIN )
12.         && ( usMBCRC16( ( UCHARUCHAR * ) ucRTUBuf, usRcvBufferPos ) == 0 ) )
13.     {
14.         /* Save the address field. All frames are passed to the upper layed
15.          * and the decision if a frame is used is done there.
16.          */
17.         *pucRcvAddress = ucRTUBuf[MB_SER_PDU_ADDR_OFF];
18.
19.         /* Total length of Modbus-PDU is Modbus-Serial-Line-PDU minus
20.          * size of address field and CRC checksum.
21.          */
22.         *pusLength = ( USHORT )( usRcvBufferPos - MB_SER_PDU_PDU_OFF - MB_SE
    R_PDU_SIZE_CRC );
23.
24.         /* Return the start of the Modbus PDU to the caller. */
25.         *pucFrame = ( UCHARUCHAR * ) & ucRTUBuf[MB_SER_PDU_PDU_OFF];
26.         xFrameReceived = TRUE;
27.     }
28.     else
29.     {
30.         eStatus = MB_EIO;
31.     }
32.
33.     EXIT_CRITICAL_SECTION(  );
34.     return eStatus;
35. }
```

eMBRTUReceive 函数完成了 CRC 校验、帧数据地址、长度的赋值，便于给上层进行处理。之后发送(void)xMBPortEventPost(EV_EXECUTE)事件。处理数据时根据功能码调用相应的函数，这些函数存储在 xFuncHandlers 数组中。之后发送响应，完成一次操作。