|  |  |  |
| --- | --- | --- |
| 日 期 | 内 容 | 结 果 |
| 2018.3.31  下 午 | 1、查芯片74HC154;  2、电路各功能检测； | 如 下 ： |
| 2018．4.2-3 | 1、源码详解  （1）初始化（1、2、3）  （2）接收一帧完整报文  （3）报文处理机制  （4）发送响应报文 | 1、Modbus协议栈的初始化准备工作  2、eMBPoll()函数轮询等待接收完成事件发生  3、接收机状态eRcvState为STATE\_RX\_IDLE空闲  4、报文解析结束，得到ucMBFrame响应缓冲和usLength响应报文长度，等待发送报文 |
| 2018.4.4 | Modbus协议整理 |  |
| 2018.4.5 | Modbus 移植 | FreeModbus源码详解1+freeModbus代码解读及移植笔记 |
| 2018.4.6 | stm32+modbus 485 | 1. 物理层接口文件的修改 2. 应用层回函数的修改 3. 应用层初始化及协议访问 |
| 2018.4.7 | 移植测试、与上位机的发送接收 |  |
|  |  |  |
|  |  |  |
|  |  |  |

1. **芯片74HC154**

**1、74HC154概述**

74HC154是一款告诉CMOS器件，74HC154引脚兼容低功耗肖特基TTL系列。

74HC154译码器可接受4位有效二进制地址输入，并提供16个互斥的低有效输出。该使能电路包含两个“逻辑与”输入，必须置为低以便使能输出端，任选一个使能输入端作为数据输入，74HC154可充当一个1-16的多路分配器。当其余的使能输入端为低时，地址输出将会跟随应用的状态。

2、特性

16线多路分配功能；

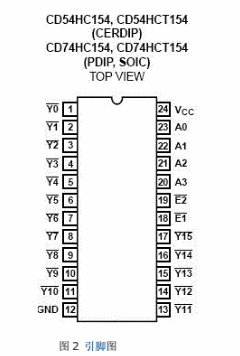
4位2进制码输入译码至16个互斥输出；

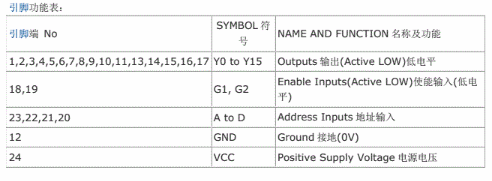
工作电压：2.0—6.0V；

驱动电流：+/-5.2mA；

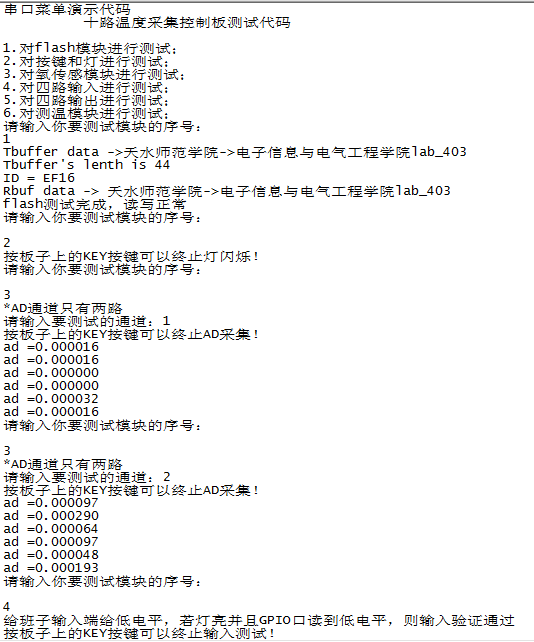
温度范围：-40--+85摄氏度

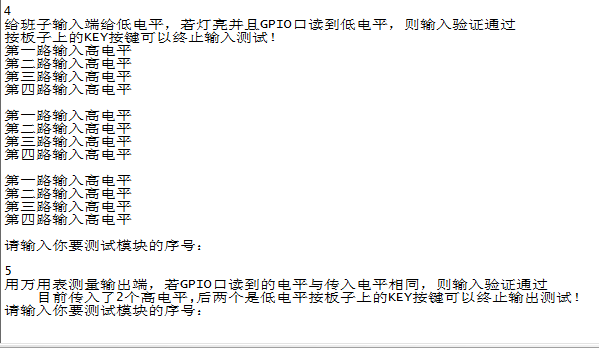
3、管脚图：





1. 板子各部分功能测量





第六部分：温度测量：

单路测量：T从T1开始，路数从0路开始。

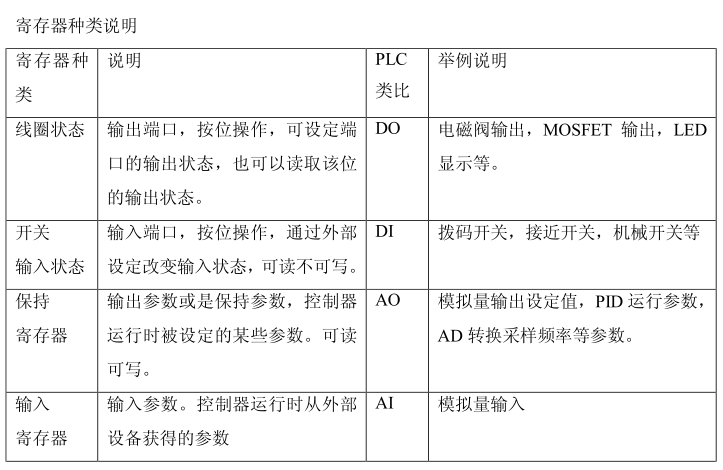
出现问题：第四路，第七路读取温度值不合适。

多路循环采集：T从T1开始，路数从1路开始。

出现问题：第五路，第八路读取温度不合适。

即板子上T5,T8路读取温度不合适。

三、



特别说明：

寄存器信息地址（PLC 地址）

寄存器信息地址指的是存放于控制器中的地址，这些控制器可以是 PLC，也可以使触

摸屏，或是文本显示器。例如 40001、30002 等，这些地址一般使用 10 进制描述。

寄存器寻址地址（ 协议地址）

寄存器寻址地址指的是通信时使用的寄存器地址，例如信息地址 40001 对应寻址地址

0x0000，40002 对应寻址地址 0x0001，寄存器寻址地址一般使用 16 进制描述。再如，信息

寄存器 40003 对应寻址地址 0002，信息寄存器 30003 对应寻址地址 0002，虽然两个信息寄

存器通信时使用相同的地址，但是需要使用不同的命令才可以访问，所以访问时不存在冲突

1. 协议理解
2. 简单分析一条modbus-RTU报文，例如：01 06 00 01 00 17 98 04

01（从机地址）06（功能号）00 01（数据地址） 00 17（数据） 98 04（CRC校验）

这一串数据意思是：把数据0x0017（十进制23）写入1号从机地址0x0001数据地址

1. 报文：一个报文就是一帧数据，一个数据帧就是一个报文：指的是一串完整的指令数据，就像上面的一串数据。
2. CRC校验

例如：98 04是它前面的数据通过算法计算出来的结果。

作用：在数据传输过程中可能数据发生错误，CRC校验检测接收的数据是否正确。

1. 功能号：意义：modbus定义 作用：指示具体的操作。
2. MODBUS-RTU
3. 一个报文分析：先声明下我们的目的，我们是要两个设备通讯，用的是MODBUS协议
4. 主机对从机写数据操作

如果单片机接收到一个报文那么就对报文进行解析执行相应的处理，如上面：01（从机地址）06（功能号）00 01（数据地址）00 17（数据） 98 04（CRC校验）

假如本机地址是1，那么单片机接收到这串数据根据数据计算CRC校验判断数据是否正确，如果判断数据无误，则结果是：HoldDataReg[1] = 0x0017;

1. 主机对从机读数据操作

主机进行读HoldDataReg[1]操作，则报文是：01（从机地址）03（功能号）00 01（数据地址）00 01（读取数据个数）D5 CA（CRC校验）那么单片机接收到这串数据根据计算CRC校验判断数据是否正确，如果正确，则结果是：返回信息给主机，返回的信息也是有格式的，返回内容：01（从机地址）03（功能号）02（数据字节个数）0017（两个字节数据）F8 4A（CRC校验）

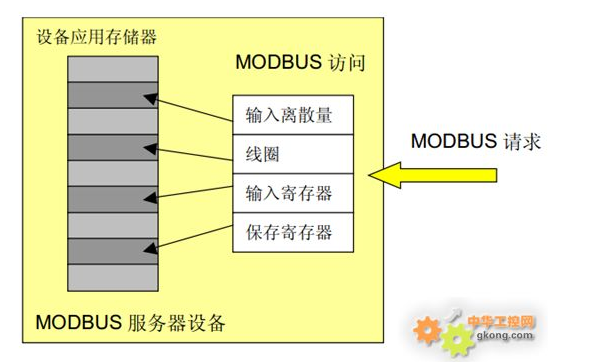
1. ModBus只是个虚拟的通讯协议，虚拟一些寄存器并向里面读写数据，一般使用0x03读寄存器模式，0x10写寄存器模式，再加上CRC校验
2. 0x01和0x03指令

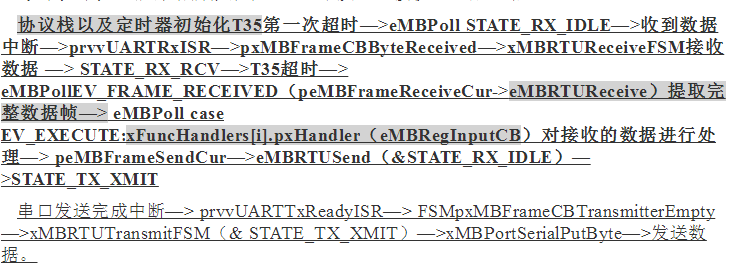
0x10（16）写多个寄存器指令协议：（向寄存器写数据，传送给下位机）  
主机请求：  
01 10 00 30 00 03 06 00 1E 00 20 00 32 CE 62    最后两位为校验码  
      从机地址：01                                //下位机地址  
          功能码：10  
寄存器起始地址：0030  
    寄存器数量：0003  
      字节计数：06  
       PWM频率：001E                                （寄存器地址0030）  
     PWM占空比：0020                                （寄存器地址0031）  
   PWM脉冲个数：0032                                （寄存器地址0032）  
     CRC校验码：CE62  
  
从机应答：  
01 10 00 30 00 03 80 07                         最后两位为校验码  
      从机地址：01  
          功能码：10  
寄存器起始地址：0030  
    寄存器数量：0003  
     CRC校验码：8007  
  
0x03(3)读保持寄存器指令协议：（读取寄存器数据，即当前寄存器的数据）  
主机请求：  
01 03 00 30 00 03 05 C4  
      从机地址：01  
          功能码：10  
寄存器起始地址：0030  
    寄存器数量：0003  
     CRC校验码：05C4  
从机应答：  
01 03 06 00 1E 00 20 00 32 09 68   
      从机地址：01  
          功能码：03  
读出数据字节数：06  
    寄存器的值：  
        1.00 1E  
        2.00 20  
        3.00 32  
     CRC校验码：0968  
上面是一个从上位机发送PWM频率、占空比、脉冲个数三个参数的Modeus通讯协议，请求是上位机的PC软件发给下位机的，然后下位机检验从机地址、功能码、寄存器数量、CRC校验值是否正确，如果正确则给予应答（正确的应答），如果不正确就回复错误代码

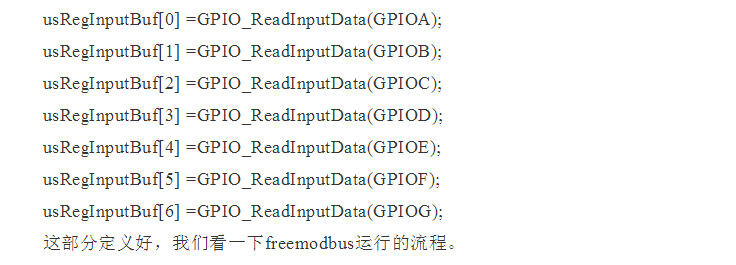
所谓的“线圈”“寄存器”就是“位变量”“16位变量”，不要被迷惑。之所以称“线圈”我觉得应该是对于应用的设备

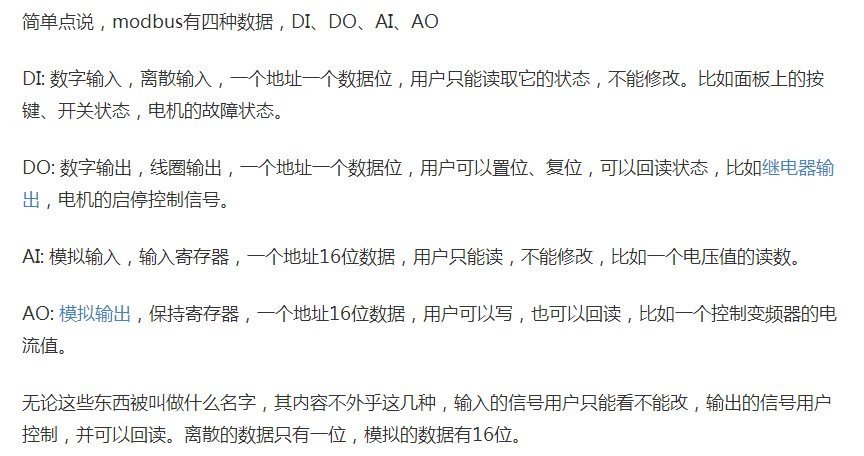
***4.30***













**关于Modbus 3区、4区寄存器地址的理解以及Freemodbus中开始地址的设定**

在Modbus实际应用中，我们对Modbus 3区、4区的地址有的时候会出现混淆，尤其是类似于404097这种表达方式的地址，就更容易乱，因为我们常常会用串口调试，这个就容易难理解。

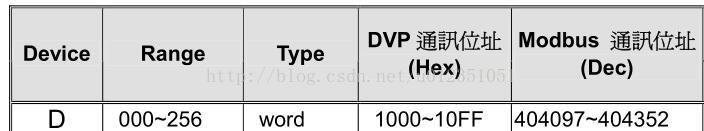
Modbus 中3区和4区的地址表示含义如下：

30001-39999是输入寄存器，也就是我们常说的输入寄存器，只读。

40001-49999是保持寄存器，可以读写。

从上面的定义可以看出来，3区，4区的寄存器开始地址不是0，而是30001和40001，但是在modbus协议中，这两区的起始地址却是可以从0开始的，那么这是为什么呢？答案是这里的30001和40001，是专用于PLC的表达，台达或者西门子都是这样表达的，我们在用modbus调试工具进行调试时，经过数据抓取，发现，30001和40001对应的命令地址，就是00 00 ，所以看到3xxxx或者4xxxx，只要减去30001或者40001就是它的命令地址了。

另外在Freemodbus中，它是参照PLC的起始地址来设计的，并不是数也是要从30001或者40001开始，而是说它不能从0开始，而是从1开始的，1就对应30001或者40001，举例来讲，如果想要仿照台达PLC的通信协议，台达通信地址D去规定：



D0对应的Modbus地址为4区的4097，那么在Freemodbus中对应的起始地址就是4097，而不是4096，但是真正的命令地址却是4096，所以读D0的命令应该是：

01 03 10 00 00 01 xx xx（这里是RTU，台达是支持ASCII的），其中0x1000=4096



***5.1***

**通讯协议**  
（一）、通讯传送方式：  
　　 通讯传送分为独立的信息头，和发送的编码数据。以下的通讯传送方式定义也与MODBUS RTU通讯规约相兼容：   
  
编 码 8位二进制   
起始位 1位   
数据位 8位   
奇偶校验位 1位（偶校验位）   
停止位 1位   
错误校检 CRC（冗余循环码）   
  
初始结构 = ≥4字节的时间   
地址码 = 1 字节  
功能码 = 1 字节  
数据区 = N 字节  
错误校检 = 16位CRC码   
结束结构 = ≥4字节的时间  
  
  
　　地址码：地址码为通讯传送的第一个字节。这个字节表明由用户设定地址码的从机将接收由主机发送来的信息。并且每个从机都有具有唯一的地址码，并且响应回送均以各自的地址码开始。主机发送的地址码表明将发送到的从机地址，而从机发送的地址码表明回送的从机地址。  
  
　　功能码：通讯传送的第二个字节。ModBus通讯规约定义功能号为1到127。本仪表只利用其中的一部分功能码。作为主机请求发送，通过功能码告诉从机执行什么动作。作为从机响应，从机发送的功能码与从主机发送来的功能码一样，并表明从机已响应主机进行操作。如果从机发送的功能码的最高位为１(比如功能码大与此同时127)，则表明从机没有响应操作或发送出错。  
  
　　数据区：数据区是根据不同的功能码而不同。数据区可以是实际数值、设置点、主机发送给从机或从机发送给主机的地址。  
  
　　 CRC码：二字节的错误检测码。  
  
（二）、通讯规约：  
  
　　 当通讯命令发送至仪器时，符合相应地址码的设备接通讯命令，并除去地址码，读取信息，如果没有出错，则执行相应的任务；然后把执行结果返送给发送者。返送的信息中包括地址码、执行动作的功能码、执行动作后结果的数据以及错误校验码。如果出错就不发送任何信息。  
  
1．信息帧结构  
  
地址码 功能码 数据区 错误校验码   
8位 8位 N × 8位 16位   
  
　　地址码：地址码是信息帧的第一字节(8位)，从0到255。这个字节表明由用户设置地址的从机将接收由主机发送来的信息。每个从机都必须有唯一的地址码，并且只有符合地址码的从机才能响应回送。当从机回送信息时，相当的地址码表明该信息来自于何处。  
  
　　 功能码：主机发送的功能码告诉从机执行什么任务。表1-1列出的功能码都有具体的含义及操作。   
  
代码 含义  操作   
03 读取数据 读取当前寄存器内一个或多个二进制值   
06 重置单一寄存器 把设置的二进制值写入单一寄存器   
  
　　数据区：数据区包含需要从机执行什么动作或由从机采集的返送信息。这些信息可以是数值、参考地址等等。例如，功能码告诉从机读取寄存器的值，则数据区必需包含要读取寄存器的起始地址及读取长度。对于不同的从机，地址和数据信息都不相同。  
  
　　错误校验码：主机或从机可用校验码进行判别接收信息是否出错。有时，由于电子噪声或其它一些干扰，信息在传输过程中会发生细微的变化，错误校验码保证了主机或从机对在传送过程中出错的信息不起作用。这样增加了系统的安全和效率。错误校验采用CRC-16校验方法。  
  
注：信息帧的格式都基本相同：地址码、功能码、数据区和错误校验码。  
  
2．错误校验  
  
　　 冗余循环码（CRC）包含2个字节，即16位二进制。CRC码由发送设备计算，放置于发送信息的尾部。接收信息的设备再重新计算接收到信息的 CRC码，比较计算得到的CRC码是否与接收到的相符，如果两者不相符，则表明出错。  
  
　　CRC码的计算方法是，先预置16位寄存器全为1。再逐步把每8位数据信息进行处理。在进行CRC码计算时只用8位数据位，起始位及停止位，如有奇偶校验位的话也包括奇偶校验位，都不参与CRC码计算。  
  
　　 在计算CRC码时，8位数据与寄存器的数据相异或，得到的结果向低位移一字节，用0填补最高位。再检查最低位，如果最低位为1，把寄存器的内容与预置数相异或，如果最低位为0，不进行异或运算。  
  
　　 这个过程一直重复8次。第8次移位后，下一个8位再与现在寄存器的内容相相异或，这个过程与以上一样重复8次。当所有的数据信息处理完后，最后寄存器的内容即为CRC码值。CRC码中的数据发送、接收时低字节在前。  
  
　　 计算CRC码的步骤为：  
  
预置16位寄存器为十六进制FFFF（即全为1）。称此寄存器为CRC寄存器；   
把第一个8位数据与16位CRC寄存器的低位相异或，把结果放于CRC寄存器；   
把寄存器的内容右移一位(朝低位)，用0填补最高位，检查最低位；   
如果最低位为0：重复第3步(再次移位); 如果最低位为1：CRC寄存器与多项式A001（1010 0000 0000 0001）进行异或；   
重复步骤3和4，直到右移8次，这样整个8位数据全部进行了处理；   
重复步骤2到步骤5，进行下一个8位数据的处理；   
最后得到的CRC寄存器即为CRC码。   
3．功能码03，读取点和返回值：  
  
　　仪表采用Modbus RTU通讯规约，利用通讯命令，可以进行读取点(“保持寄存器”) 或返回值(“输入寄存器” )的操作。保持和输入寄存器都是16位（2字节）值，并且高位在前。这样用于仪表的读取点和返回值都是2字节。一次最多可读取寄存器数是60。由于一些可编程控制器不用功能码03，所以功能码03被用作读取点和返回值。从机响应的命令格式是从机地址、功能码、数据区及CRC码。数据区中的寄存器数据都是每两个字节高字节在前。  
  
4．功能码06，单点保存  
  
　　主机利用这条命令把单点数据保存到仪表的存储器。从机也用这个功能码向主机返送信息。

一、前提

主机与从机通讯必须正确，这是进一步做命令解析的前提。使用Modbus Poll等测试程序，可以测试从机FreeModbus代码移植是否正确，如果通讯有问题，那就赶紧去查……

二、main函数与各种功能码对应的函数之间的关系

翻阅一下给的例程，主程序里包含eMBRegInputCB、eMBRegHoldingCB、eMBRegCoilsCB、eMBRegDiscreteCB、以及读从机标识（这个实现在其它文件里）等函数。这些函数可以将主机发过来的命令直接放入好定义的缓冲区，如：eMBRegHoldingCB函数内部实现了读、写两个功能，分别对应功能号0x03的数据上传，以及功能号0x10、另外一个写函数（功能号不记得了）的数据下发。主机下发的数据就包含在usRegHoldingBuf中。

当然，不同功能号使用的数据缓冲区需要单独命名，起始地址也需要单独定义。不过起始地址是不是一样都没所谓，只是自己看着舒爽就行了。

三、解析主机命令

以写功能0x10为例，下发数据是成帧的，即，MODBUS协议保证了会收到完整的一个主机序列，以CRC16做结尾。那么，判断何时收到一帧数据，也一定是在usRegHoldingBuf缓冲区填写完毕之后，此时可设定一个标识，表示收到完整的数据帧，而在main主循环处理此标识即可。

代码如下：

// FUN=0x03

eMBErrorCode eMBRegHoldingCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNRegs, eMBRegisterMode eMode )

{

    eMBErrorCode    eStatus = MB\_ENOERR;

    int             iRegIndex;

    if( ( usAddress >= REG\_HOLDING\_START ) &&

        ( usAddress + usNRegs <= REG\_HOLDING\_START + REG\_HOLDING\_NREGS ) )

    {

        iRegIndex = ( int )( usAddress - usRegHoldingStart );

        switch ( eMode )

        {

        // Pass current register values to the protocol stack.

        case MB\_REG\_READ:

            while( usNRegs > 0 )

            {

                \*pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] >> 8 );

                \*pucRegBuffer++ = ( UCHAR ) ( usRegHoldingBuf[iRegIndex] & 0xFF );

                iRegIndex++;

                usNRegs--;

            }

            break;

        // Update current register values with new values from the protocol stack.

        case MB\_REG\_WRITE:

            while( usNRegs > 0 )

            {

                usRegHoldingBuf[iRegIndex] = \*pucRegBuffer++ << 8;

                usRegHoldingBuf[iRegIndex] |= \*pucRegBuffer++;

                iRegIndex++;

                usNRegs--;

            }

**// 接收完毕主机来的一帧数据，设置标识**

**flag.bCOM0\_NewFrame = TRUE;**

             break;

        }

    }

    else

    {

        eStatus = MB\_ENOREG;

    }

    return eStatus;

}

main()

{

**flag.bCOM0\_NewFrame = FALSE;**

      ...

      for(;;)

      {

           // Call the main polling loop of the Modbus protocol stack.

           ( void )eMBPoll( );

**// 如果收到主机下发帧，则处理主机数据**

**if(flag.bCOM0\_NewFrame)**

**{**

**switch(usRegHoldingBuf[0]) // 数据存放于此，以下根据具体定义实现**

**{**

**case FUN1:**

**...**

**break;**

**case FUN2:**

**...**

**break;**

**}**

**// 清除标记**

**flag.bCOM0\_NewFrame = FALSE;**

 **}**

    }

}

自此，完成从机接收数据的处理过程。这个思路很常见，无非与他人写好的源码绕在一起了，看起来有些陌生。这样做的好处是，主机每发送一帧数据，从机只会解释执行一次。

四、主机读设备的数据

这是上述过程的逆过程。对于不再修改的数据，从机可以事先填充好对应的缓冲区，等待主机读走就完事。

main()

{

**// 变量赋值**

**usRegInputBuf[0] = 1;**

**usRegInputBuf[1] = 2;**

**usRegInputBuf[2] = 3;**

**usRegInputBuf[3] = 4;**

for(;;)

{

...

}

}

如果从机需要在每次收到主机命令之后再准备数据，可以参照上述方法实现，不赘述。

如果把准备数据的代码放在**MB\_REG\_READ**语句之后也是可以的，无非代码结构不好，读起来费劲。

五、其它

一个设计完备的MODBUS协议栈，一定要有完备的应用层的通讯协议，即，具体的功能实现。为这些上层通讯协议分配功能号，参数，缓冲区地址、长度，书写文档，都是花费时间的工作，但这是一个完整设计必不可少的部分，花费再多时间也是值得的。

工具软件：MODBUS POLL

硬件RTU：参照Freemodbus例程，自己写的。

一、缺省MODBUS命令测试

（1）功能号FUN=1，读取一组逻辑线圈的当前状态（ON/OFF）

读取起始地址3000（0x0BB8）开始的16个线圈（实际设置地址为起始地址-1，下同）

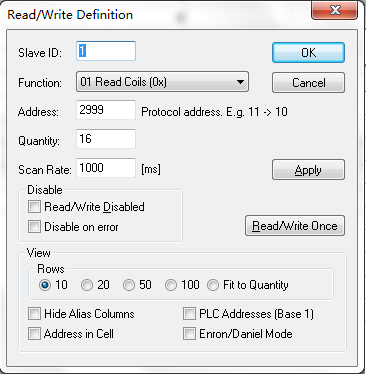
020-Tx:01 01 0B B7 00 10 8F C4 分别为从站ID、功能号、起始地址、数据长度、校验

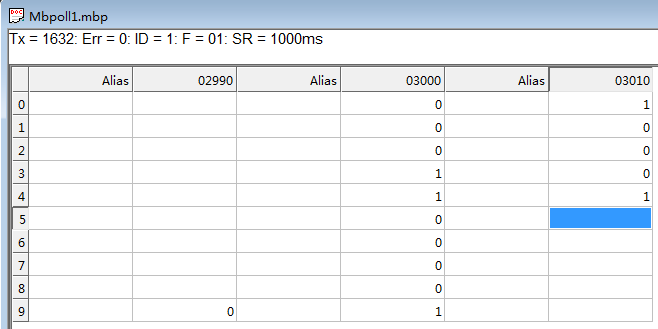
021-Rx:01 01 02 30 8C AC 59 分别为从站ID、功能号、寄存器数量、寄存器值、校验

这里要注意的有两点：

* 发送的起始地址应该是协议地址-1
* 发送的数据长度0x0010，是读取16个线圈。接收的寄存器值把16个线圈压缩到了2个字节。

在Poll界面上，做如下设置：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=493520900102x50l&url=http://album.sina.com.cn/pic/001l7tK0zy79ATLhjPSe4)

运行结果：  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=493520900102x50l&url=http://album.sina.com.cn/pic/001l7tK0zy79ATLtQIq3a)

从高到低算下来，刚好是0x30 0x8C

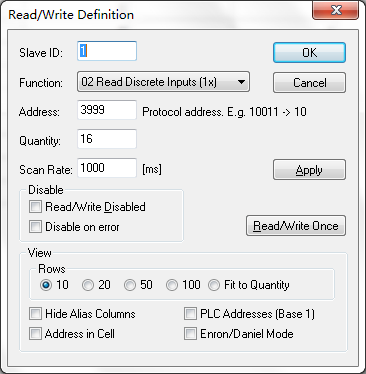
（2）功能号FUN=2，读取一组开关的当前输入状态（ON/OFF）

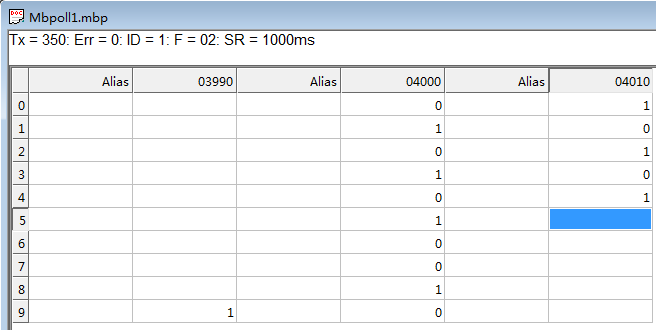
读取起始地址4000（0x0BB8）开始的16个开关

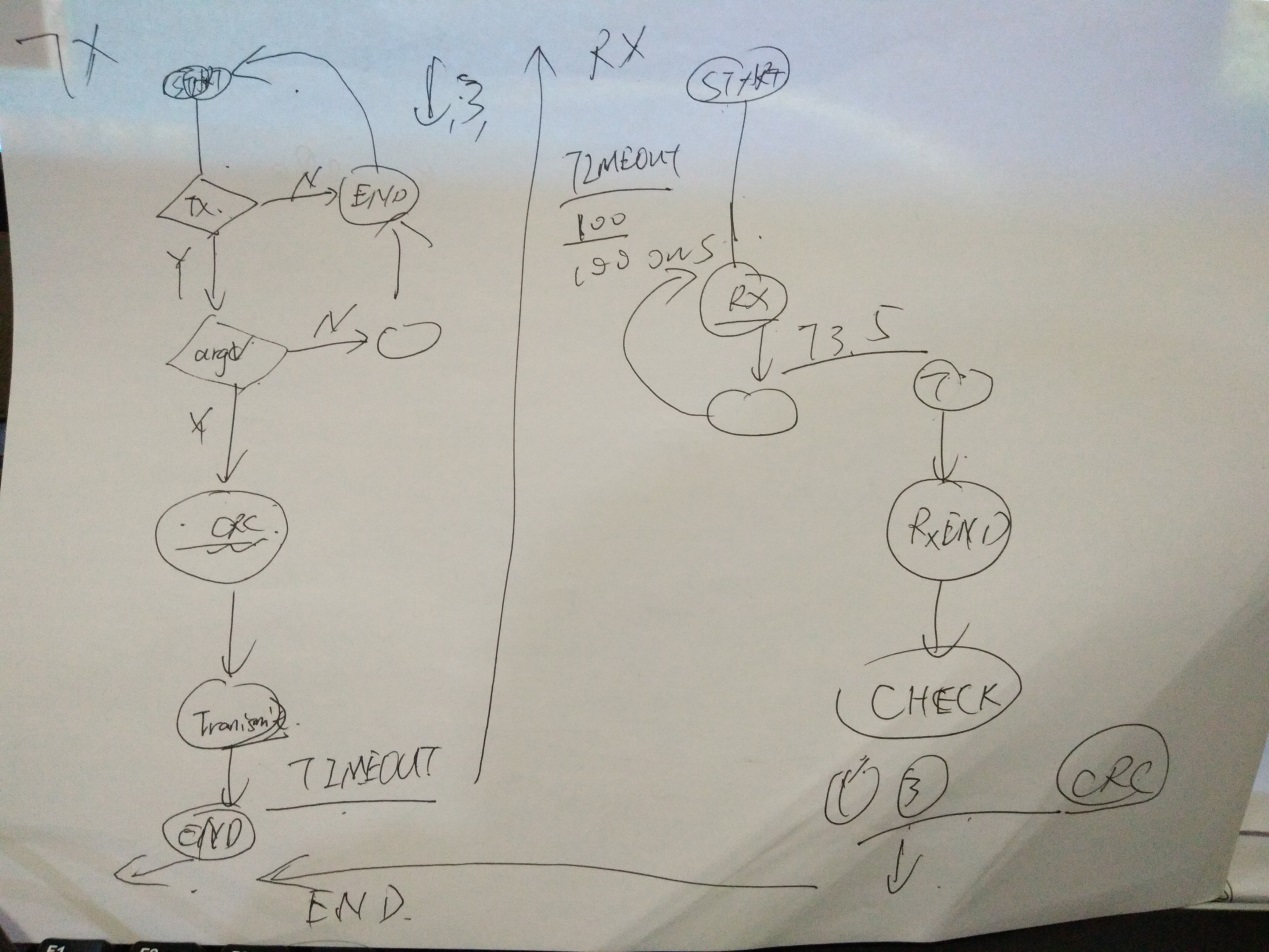
000-Tx:01 02 0F 9F 00 10 4A FC 分别为从站ID、功能号、起始地址、数据长度、校验

001-Rx:01 02 02 55 AA 06 97 分别为从站ID、功能号、寄存器数量、寄存器值、校验

在Poll界面上，做如下设置：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=493520900102x50l&url=http://album.sina.com.cn/pic/001l7tK0zy79AUJv3hBa7)

运行结果：  
[](http://photo.blog.sina.com.cn/showpic.html#blogid=493520900102x50l&url=http://album.sina.com.cn/pic/001l7tK0zy79AUJQWKIe8)  
为0x55 0xAA



***5.2***

## 主流程

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16811963) [copy](https://blog.csdn.net/rianboe917/article/details/16811963)

1. /\* ----------------------- Start implementation -----------------------------\*/
2. **int**
3. main( **void** )
4. {
5. eMBErrorCode    eStatus;
7. eStatus = eMBInit( MB\_RTU, 0x0A, 0, 38400, MB\_PAR\_EVEN );
9. /\* Enable the Modbus Protocol Stack. \*/
10. eStatus = eMBEnable(  );
12. **for**( ;; )
13. {
14. ( **void** )eMBPoll(  );
16. /\* Here we simply count the number of poll cycles. \*/
17. usRegInputBuf[0]++;
18. }
19. }

由上述主函数可知协议栈经eMBInit和eMBEnable初始化、使能后进入协议栈的循环eMBPoll中。

## eMBInit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！

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16811963) [copy](https://blog.csdn.net/rianboe917/article/details/16811963)

1. #if MB\_RTU\_ENABLED > 0
2. **case** MB\_RTU:
3. pvMBFrameStartCur = eMBRTUStart;
4. pvMBFrameStopCur = eMBRTUStop;
5. peMBFrameSendCur = eMBRTUSend;
6. peMBFrameReceiveCur = eMBRTUReceive;
7. pvMBFrameCloseCur = MB\_PORT\_HAS\_CLOSE ? vMBPortClose : **NULL**;
8. pxMBFrameCBByteReceived = xMBRTUReceiveFSM;
9. pxMBFrameCBTransmitterEmpty = xMBRTUTransmitFSM;
10. pxMBPortCBTimerExpired = xMBRTUTimerT35Expired;
12. eStatus = eMBRTUInit( ucMBAddress, ucPort, ulBaudRate, eParity );
13. **break**;
14. #endif
15. #if MB\_ASCII\_ENABLED > 0
16. **case** MB\_ASCII:
17. pvMBFrameStartCur = eMBASCIIStart;
18. pvMBFrameStopCur = eMBASCIIStop;
19. peMBFrameSendCur = eMBASCIISend;
20. peMBFrameReceiveCur = eMBASCIIReceive;
21. pvMBFrameCloseCur = MB\_PORT\_HAS\_CLOSE ? vMBPortClose : **NULL**;
22. pxMBFrameCBByteReceived = xMBASCIIReceiveFSM;
23. pxMBFrameCBTransmitterEmpty = xMBASCIITransmitFSM;
24. pxMBPortCBTimerExpired = xMBASCIITimerT1SExpired;
26. eStatus = eMBASCIIInit( ucMBAddress, ucPort, ulBaudRate, eParity );
27. **break**;
28. #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的函数进行赋值初始化！！此编写模式可以借鉴学习！！

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。

## eMBEnable的分析

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16811963) [copy](https://blog.csdn.net/rianboe917/article/details/16811963)

1. <span style="font-size:10px;">eMBErrorCode
2. eMBEnable( **void** )
3. {
4. eMBErrorCode    eStatus = MB\_ENOERR;
6. **if**( eMBState == STATE\_DISABLED )
7. {
8. /\* Activate the protocol stack. \*/
9. pvMBFrameStartCur(  );
10. eMBState = STATE\_ENABLED;
11. }
12. **else**
13. {
14. eStatus = MB\_EILLSTATE;
15. }
16. **return** eStatus;
17. }</span>

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分配到为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。

## eMBPoll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：

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16830479) [copy](https://blog.csdn.net/rianboe917/article/details/16830479)

1. **BOOL**
2. xMBRTUTimerT35Expired( **void** )
3. {
4. **BOOL**            xNeedPoll = FALSE;
6. **switch** ( eRcvState )
7. {
8. /\* Timer t35 expired. Startup phase is finished. \*/
9. **case** STATE\_RX\_INIT:
10. xNeedPoll = xMBPortEventPost( EV\_READY );
11. **break**;
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**;
19. /\* An error occured while receiving the frame. \*/
20. **case** STATE\_RX\_ERROR:
21. **break**;
23. /\* Function called in an illegal state. \*/
24. default:
25. assert( ( eRcvState == STATE\_RX\_INIT ) ||
26. ( eRcvState == STATE\_RX\_RCV ) || ( eRcvState == STATE\_RX\_ERROR ) );
27. }
29. vMBPortTimersDisable(  );
30. eRcvState = STATE\_RX\_IDLE;
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:

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16830479) [copy](https://blog.csdn.net/rianboe917/article/details/16830479)

1. **BOOL**
2. xMBRTUReceiveFSM( **void** )
3. {
4. **BOOL**            xTaskNeedSwitch = FALSE;
5. **UCHAR**           ucByte;
7. assert( eSndState == STATE\_TX\_IDLE );
9. /\* Always read the character. \*/
10. ( **void** )xMBPortSerialGetByte( ( **CHARCHAR** \* ) & ucByte );
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**;
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**;
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;
37. /\* Enable t3.5 timers. \*/
38. vMBPortTimersEnable(  );
39. **break**;
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！

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16830479) [copy](https://blog.csdn.net/rianboe917/article/details/16830479)

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。

**[objc]** [view plain](https://blog.csdn.net/rianboe917/article/details/16830479) [copy](https://blog.csdn.net/rianboe917/article/details/16830479)

1. eMBErrorCode
2. eMBRTUReceive( **UCHARUCHAR** \* pucRcvAddress, **UCHARUCHAR** \*\* pucFrame, **USHORTUSHORT** \* pusLength )
3. {
4. **BOOL**            xFrameReceived = FALSE;
5. eMBErrorCode    eStatus = MB\_ENOERR;
7. ENTER\_CRITICAL\_SECTION(  );
8. assert( usRcvBufferPos < MB\_SER\_PDU\_SIZE\_MAX );
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];
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\_SER\_PDU\_SIZE\_CRC );
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. }
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！

**本篇按照FreeModbus协议栈的工作流程，对源代码进行总结解析；FreeModbus协议栈作为从机，等待主机传送的数据，当从机接收到一帧完整的报文后，对报文进行解析，然后响应主机，发送报文给主机，实现主机和从机之间的通信；**

**1：demo.c中三个函数，完成协议栈的准备工作；**

**eMBInit()函数：(mb.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能：
2. \*1:实现RTU模式和ASCALL模式的协议栈初始化;
3. \*2:完成协议栈核心函数指针的赋值，包括Modbus协议栈的使能和禁止、报文的接收和响应、3.5T定时器中断回调函数、串口发送和接收中断回调函数;
4. \*3:eMBRTUInit完成RTU模式下串口和3.5T定时器的初始化，需用户自己移植;
5. \*4:设置Modbus协议栈的模式eMBCurrentMode为MB\_RTU，设置Modbus协议栈状态eMBState为STATE\_DISABLED;
6. \*/
7. eMBErrorCode
8. eMBInit( eMBMode eMode, **UCHAR** ucSlaveAddress, **UCHAR** ucPort, **ULONG** ulBaudRate, eMBParity eParity )
9. {
10. //错误状态初始值
11. eMBErrorCode    eStatus = MB\_ENOERR;
13. //验证从机地址
14. **if**( ( ucSlaveAddress == MB\_ADDRESS\_BROADCAST ) ||
15. ( ucSlaveAddress < MB\_ADDRESS\_MIN ) || ( ucSlaveAddress > MB\_ADDRESS\_MAX ))
16. {
17. eStatus = MB\_EINVAL;
18. }
19. **else**
20. {
21. ucMBAddress = ucSlaveAddress;            /\*从机地址的赋值\*/
23. **switch** ( eMode )
24. {
25. #if MB\_RTU\_ENABLED > 0
26. **case** MB\_RTU:
27. pvMBFrameStartCur = eMBRTUStart;      /\*使能modbus协议栈\*/
28. pvMBFrameStopCur = eMBRTUStop;        /\*禁用modbus协议栈\*/
29. peMBFrameSendCur = eMBRTUSend;        /\*modbus从机响应函数\*/
30. peMBFrameReceiveCur = eMBRTUReceive;  /\*modbus报文接收函数\*/
31. pvMBFrameCloseCur = MB\_PORT\_HAS\_CLOSE ? vMBPortClose : NULL;
32. //接收状态机
33. pxMBFrameCBByteReceived =     xMBRTUReceiveFSM;   /\*串口接收中断最终调用此函数接收数据\*/
34. //发送状态机
35. pxMBFrameCBTransmitterEmpty = xMBRTUTransmitFSM;  /\*串口发送中断最终调用此函数发送数据\*/
36. //报文到达间隔检查
37. pxMBPortCBTimerExpired =      xMBRTUTimerT35Expired; /\*定时器中断函数最终调用次函数完成定时器中断\*/
38. //初始化RTU
39. eStatus = eMBRTUInit( ucMBAddress, ucPort, ulBaudRate, eParity );
40. **break**;
41. #endif
42. #if MB\_ASCII\_ENABLED > 0
43. **case** MB\_ASCII:
44. pvMBFrameStartCur = eMBASCIIStart;
45. pvMBFrameStopCur = eMBASCIIStop;
46. peMBFrameSendCur = eMBASCIISend;
47. peMBFrameReceiveCur = eMBASCIIReceive;
48. pvMBFrameCloseCur = MB\_PORT\_HAS\_CLOSE ? vMBPortClose : NULL;
49. pxMBFrameCBByteReceived = xMBASCIIReceiveFSM;
50. pxMBFrameCBTransmitterEmpty = xMBASCIITransmitFSM;
51. pxMBPortCBTimerExpired = xMBASCIITimerT1SExpired;
53. eStatus = eMBASCIIInit( ucMBAddress, ucPort, ulBaudRate, eParity );
54. **break**;
55. #endif
56. **default**:
57. eStatus = MB\_EINVAL;
58. }
60. //
61. **if**( eStatus == MB\_ENOERR )
62. {
63. **if**( !xMBPortEventInit() )
64. {
65. /\* port dependent event module initalization failed. \*/
66. eStatus = MB\_EPORTERR;
67. }
68. **else**
69. {
70. //设定当前状态
71. eMBCurrentMode = eMode;       //设定RTU模式
72. eMBState = STATE\_DISABLED;    //modbus协议栈初始化状态,在此初始化为禁止
73. }
74. }
75. }
76. **return** eStatus;
77. }

**eMBEnable()函数：(mb.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能
2. \*1:设置Modbus协议栈工作状态eMBState为STATE\_ENABLED;
3. \*2:调用pvMBFrameStartCur()函数激活协议栈
4. \*/
5. eMBErrorCode
6. eMBEnable( **void** )
7. {
8. eMBErrorCode    eStatus = MB\_ENOERR;
10. **if**( eMBState == STATE\_DISABLED )
11. {
12. /\* Activate the protocol stack. \*/
13. pvMBFrameStartCur(  );  /\*pvMBFrameStartCur = eMBRTUStart;调用eMBRTUStart函数\*/
14. eMBState = STATE\_ENABLED;
15. }
16. **else**
17. {
18. eStatus = MB\_EILLSTATE;
19. }
20. **return** eStatus;
21. }

**eMBRTUStart()函数：(mbrtu.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能
2. \*1:设置接收状态机eRcvState为STATE\_RX\_INIT；
3. \*2:使能串口接收,禁止串口发送,作为从机,等待主机传送的数据;
4. \*3:开启定时器，3.5T时间后定时器发生第一次中断,此时eRcvState为STATE\_RX\_INIT,上报初始化完成事件,然后设置eRcvState为空闲STATE\_RX\_IDLE;
5. \*4:每次进入3.5T定时器中断,定时器被禁止，等待串口有字节接收后，才使能定时器;
6. \*/
7. **void**
8. eMBRTUStart( **void** )
9. {
10. ENTER\_CRITICAL\_SECTION(  );
11. /\* Initially the receiver is in the state STATE\_RX\_INIT. we start
12. \* the timer and if no character is received within t3.5 we change
13. \* to STATE\_RX\_IDLE. This makes sure that we delay startup of the
14. \* modbus protocol stack until the bus is free.
15. \*/
16. eRcvState = STATE\_RX\_INIT;
17. vMBPortSerialEnable( TRUE, FALSE );
18. vMBPortTimersEnable();
20. EXIT\_CRITICAL\_SECTION( );
21. }

**eMBPoll()函数：(mb.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能:
2. \*1:检查协议栈状态是否使能，eMBState初值为STATE\_NOT\_INITIALIZED，在eMBInit()函数中被赋值为STATE\_DISABLED,在eMBEnable函数中被赋值为STATE\_ENABLE;
3. \*2:轮询EV\_FRAME\_RECEIVED事件发生，若EV\_FRAME\_RECEIVED事件发生，接收一帧报文数据，上报EV\_EXECUTE事件，解析一帧报文，响应(发送)一帧数据给主机;
4. \*/
5. eMBErrorCode
6. eMBPoll( **void** )
7. {
8. **static** **UCHAR**   \*ucMBFrame;           //接收和发送报文数据缓存区
9. **static** **UCHAR**    ucRcvAddress;        //modbus从机地址
10. **static** **UCHAR**    ucFunctionCode;      //功能码
11. **static** **USHORT**   usLength;            //报文长度
12. **static** eMBException eException;      //错误码响应枚举
14. **int**             i;
15. eMBErrorCode    eStatus = MB\_ENOERR;         //modbus协议栈错误码
16. eMBEventType    eEvent;                      //事件标志枚举
18. /\* Check if the protocol stack is ready. \*/
19. **if**( eMBState != STATE\_ENABLED )              //检查协议栈是否使能
20. {
21. **return** MB\_EILLSTATE;                     //协议栈未使能，返回协议栈无效错误码
22. }
24. /\* Check if there is a event available. If not return control to caller.
25. \* Otherwise we will handle the event. \*/
27. //查询事件
28. **if**( xMBPortEventGet( &eEvent ) == TRUE )     //查询哪个事件发生
29. {
30. **switch** ( eEvent )
31. {
32. **case** EV\_READY:
33. **break**;
35. **case** EV\_FRAME\_RECEIVED:                  /\*接收到一帧数据，此事件发生\*/
36. eStatus = peMBFrameReceiveCur( &ucRcvAddress, &ucMBFrame, &usLength );
37. **if**( eStatus == MB\_ENOERR )           /\*报文长度和CRC校验正确\*/
38. {
39. /\* Check if the frame is for us. If not ignore the frame. \*/
40. /\*判断接收到的报文数据是否可接受，如果是，处理报文数据\*/
41. **if**( ( ucRcvAddress == ucMBAddress ) || ( ucRcvAddress == MB\_ADDRESS\_BROADCAST ) )
42. {
43. ( **void** )xMBPortEventPost( EV\_EXECUTE );  //修改事件标志为EV\_EXECUTE执行事件
44. }
45. }
46. **break**;
48. **case** EV\_EXECUTE:                                     //对接收到的报文进行处理事件
49. ucFunctionCode = ucMBFrame[MB\_PDU\_FUNC\_OFF];     //获取PDU中第一个字节，为功能码
50. eException = MB\_EX\_ILLEGAL\_FUNCTION;             //赋错误码初值为无效的功能码
51. **for**( i = 0; i < MB\_FUNC\_HANDLERS\_MAX; i++ )
52. {
53. /\* No more function handlers registered. Abort. \*/
54. **if**( xFuncHandlers[i].ucFunctionCode == 0 )
55. {
56. **break**;
57. }
58. **else** **if**( xFuncHandlers[i].ucFunctionCode == ucFunctionCode ) /\*根据报文中的功能码，处理报文\*/
59. {
60. eException = xFuncHandlers[i].pxHandler( ucMBFrame, &usLength );/\*对接收到的报文进行解析\*/
61. **break**;
62. }
63. }
65. /\* If the request was not sent to the broadcast address we
66. \* return a reply. \*/
67. **if**( ucRcvAddress != MB\_ADDRESS\_BROADCAST )
68. {
69. **if**( eException != MB\_EX\_NONE )     /\*接收到的报文有错误\*/
70. {
71. /\* An exception occured. Build an error frame. \*/
72. usLength = 0;                                                        /\*响应发送数据的首字节为从机地址\*/
73. ucMBFrame[usLength++] = ( **UCHAR** )( ucFunctionCode | MB\_FUNC\_ERROR ); /\*响应发送数据帧的第二个字节，功能码最高位置1\*/
74. ucMBFrame[usLength++] = eException;                                  /\*响应发送数据帧的第三个字节为错误码标识\*/
75. }
76. **if**( ( eMBCurrentMode == MB\_ASCII ) && MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS )
77. {
78. vMBPortTimersDelay( MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS );
79. }
80. eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength );           /\*modbus从机响应函数,发送响应给主机\*/
81. }
82. **break**;
84. **case** EV\_FRAME\_SENT:
85. **break**;
86. }
87. }
88. **return** MB\_ENOERR;
89. }

**至此：完成Modbus协议栈的初始化准备工作，eMBPoll()函数轮询等待接收完成事件发生,接收机状态eRcvState为STATE\_RX\_IDLE空闲；**

**2：FreeModbus协议栈接收一帧完整报文机制：**

**FreeModbus协议栈通过淳口中断接收一帧数据，用户需在串口接收中断中回调prvvUARTRxISR()函数；**

**prvvUARTRxISR()函数：(portserial.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. **static** **void** prvvUARTRxISR( **void** )
2. {
3. pxMBFrameCBByteReceived(  );
4. }

**在第一阶段中eMBInit()函数中赋值pxMBFrameCBByteReceived = xMBRTUReceiveFSM,发生接收中断时,最终调用xMBRTUReceiveFSM函数对数据进行接收；**

**xMBRTUReceiveFSM()函数：(mbrtu.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能
2. \*1:将接收到的数据存入ucRTUBuf[]中;
3. \*2:usRcvBufferPos为全局变量，表示接收数据的个数;

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. \*3:每接收到一个字节的数据，3.5T定时器清0
2. \*/
3. **BOOL**
4. xMBRTUReceiveFSM( **void** )
5. {
6. **BOOL**            xTaskNeedSwitch = FALSE;
7. **UCHAR**           ucByte;
9. assert( eSndState == STATE\_TX\_IDLE );                  /\*确保没有数据在发送\*/
11. ( **void** )xMBPortSerialGetByte( ( **CHAR** \* ) & ucByte );   /\*从串口数据寄存器读取一个字节数据\*/
13. //根据不同的状态转移
14. **switch** ( eRcvState )
15. {
16. /\* If we have received a character in the init state we have to
17. \* wait until the frame is finished.
18. \*/
19. **case** STATE\_RX\_INIT:
20. vMBPortTimersEnable();                             /\*开启3.5T定时器\*/
21. **break**;
23. /\* In the error state we wait until all characters in the
24. \* damaged frame are transmitted.
25. \*/
26. **case** STATE\_RX\_ERROR:                                   /\*数据帧被损坏，重启定时器，不保存串口接收的数据\*/
27. vMBPortTimersEnable();
28. **break**;
30. /\* In the idle state we wait for a new character. If a character
31. \* is received the t1.5 and t3.5 timers are started and the
32. \* receiver is in the state STATE\_RX\_RECEIVCE.
33. \*/
34. **case** STATE\_RX\_IDLE:                                    /\*接收器空闲，开始接收，进入STATE\_RX\_RCV状态\*/
35. usRcvBufferPos = 0;
36. ucRTUBuf[usRcvBufferPos++] = ucByte;               /\*保存数据\*/
38. eRcvState = STATE\_RX\_RCV;
40. /\* Enable t3.5 timers. \*/
41. vMBPortTimersEnable();                             /\*每收到一个字节，都重启3.5T定时器\*/
42. **break**;
44. /\* We are currently receiving a frame. Reset the timer after
45. \* every character received. If more than the maximum possible
46. \* number of bytes in a modbus frame is received the frame is
47. \* ignored.
48. \*/
49. **case** STATE\_RX\_RCV:
50. **if**( usRcvBufferPos < MB\_SER\_PDU\_SIZE\_MAX)
51. {
52. ucRTUBuf[usRcvBufferPos++] = ucByte;             /\*接收数据\*/
53. }
54. **else**
55. {
56. eRcvState = STATE\_RX\_ERROR;                      /\*一帧报文的字节数大于最大PDU长度，忽略超出的数据\*/
57. }
59. vMBPortTimersEnable();                               /\*每收到一个字节，都重启3.5T定时器\*/
60. **break**;
61. }
62. **return** xTaskNeedSwitch;
63. }

**当主机发送一帧完整的报文后，3.5T定时器中断发生，定时器中断最终回调xMBRTUTimerT35Expired函数;**

**xMBRTUTimerT35Expired()函数：(mbrtu.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能
2. \*1:从机接受完成一帧数据后，接收状态机eRcvState为STATE\_RX\_RCV；
3. \*2:上报“接收到报文”事件(EV\_FRAME\_RECEIVED)
4. \*3:禁止3.5T定时器，设置接收状态机eRcvState状态为STATE\_RX\_IDLE空闲;
5. \*/
6. **BOOL**
7. xMBRTUTimerT35Expired( **void** )
8. {
9. **BOOL**            xNeedPoll = FALSE;
11. **switch** ( eRcvState )
12. {
13. /\* Timer t35 expired. Startup phase is finished. \*/
14. /\*上报modbus协议栈的事件状态给poll函数,EV\_READY:初始化完成事件\*/
15. **case** STATE\_RX\_INIT:
16. xNeedPoll = xMBPortEventPost( EV\_READY );
17. **break**;
19. /\* A frame was received and t35 expired. Notify the listener that
20. \* a new frame was received. \*/
21. **case** STATE\_RX\_RCV:                                     /\*一帧数据接收完成\*/
22. xNeedPoll = xMBPortEventPost( EV\_FRAME\_RECEIVED ); /\*上报协议栈事件,接收到一帧完整的数据\*/
23. **break**;
25. /\* An error occured while receiving the frame. \*/
26. **case** STATE\_RX\_ERROR:
27. **break**;
29. /\* Function called in an illegal state. \*/
30. **default**:
31. assert( ( eRcvState == STATE\_RX\_INIT ) ||
32. ( eRcvState == STATE\_RX\_RCV ) || ( eRcvState == STATE\_RX\_ERROR ) );
33. }
35. vMBPortTimersDisable(  );        /\*当接收到一帧数据后，禁止3.5T定时器，只到接受下一帧数据开始，开始计时\*/
37. eRcvState = STATE\_RX\_IDLE;       /\*处理完一帧数据，接收器状态为空闲\*/
39. **return** xNeedPoll;
40. }

**至此：从机接收到一帧完整的报文，存储在ucRTUBuf[MB\_SER\_PDU\_SIZE\_MAX]全局变量中，定时器禁止，接收机状态为空闲；**

**3：解析报文机制**

**在第二阶段，从机接收到一帧完整的报文后，上报“接收到报文”事件，eMBPoll函数轮询，发现“接收到报文”事件发生，调用peMBFrameReceiveCur函数，此函数指针在eMBInit被赋值eMBRTUReceive函数，最终调用eMBRTUReceive函数，从ucRTUBuf中取得从机地址、PDU单元和PDU单元的长度，然后判断从机地址地是否一致，若一致，上报“报文解析事件”EV\_EXECUTE,(xMBPortEventPost( EV\_EXECUTE ));“报文解析事件”发生后，根据功能码，调用xFuncHandlers[i].pxHandler( ucMBFrame, &usLength )对报文进行解析，此过程全部在eMBPoll函数中执行；**

**eMBPoll()函数：(mb.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能:
2. \*1:检查协议栈状态是否使能，eMBState初值为STATE\_NOT\_INITIALIZED，在eMBInit()函数中被赋值为STATE\_DISABLED,在eMBEnable函数中被赋值为STATE\_ENABLE;
3. \*2:轮询EV\_FRAME\_RECEIVED事件发生，若EV\_FRAME\_RECEIVED事件发生，接收一帧报文数据，上报EV\_EXECUTE事件，解析一帧报文，响应(发送)一帧数据给主机;
4. \*/
5. eMBErrorCode
6. eMBPoll( **void** )
7. {
8. **static** **UCHAR**   \*ucMBFrame;           //接收和发送报文数据缓存区
9. **static** **UCHAR**    ucRcvAddress;        //modbus从机地址
10. **static** **UCHAR**    ucFunctionCode;      //功能码
11. **static** **USHORT**   usLength;            //报文长度
12. **static** eMBException eException;      //错误码响应枚举
14. **int**             i;
15. eMBErrorCode    eStatus = MB\_ENOERR;         //modbus协议栈错误码
16. eMBEventType    eEvent;                      //事件标志枚举
18. /\* Check if the protocol stack is ready. \*/
19. **if**( eMBState != STATE\_ENABLED )              //检查协议栈是否使能
20. {
21. **return** MB\_EILLSTATE;                     //协议栈未使能，返回协议栈无效错误码
22. }
24. /\* Check if there is a event available. If not return control to caller.
25. \* Otherwise we will handle the event. \*/
27. //查询事件
28. **if**( xMBPortEventGet( &eEvent ) == TRUE )     //查询哪个事件发生
29. {
30. **switch** ( eEvent )
31. {
32. **case** EV\_READY:
33. **break**;
35. **case** EV\_FRAME\_RECEIVED:                  /\*接收到一帧数据，此事件发生\*/
36. eStatus = peMBFrameReceiveCur( &ucRcvAddress, &ucMBFrame, &usLength );
37. **if**( eStatus == MB\_ENOERR )           /\*报文长度和CRC校验正确\*/
38. {
39. /\* Check if the frame is for us. If not ignore the frame. \*/
40. /\*判断接收到的报文数据是否可接受，如果是，处理报文数据\*/
41. **if**( ( ucRcvAddress == ucMBAddress ) || ( ucRcvAddress == MB\_ADDRESS\_BROADCAST ) )
42. {
43. ( **void** )xMBPortEventPost( EV\_EXECUTE );  //修改事件标志为EV\_EXECUTE执行事件
44. }
45. }
46. **break**;
48. **case** EV\_EXECUTE:                                     //对接收到的报文进行处理事件
49. ucFunctionCode = ucMBFrame[MB\_PDU\_FUNC\_OFF];     //获取PDU中第一个字节，为功能码
50. eException = MB\_EX\_ILLEGAL\_FUNCTION;             //赋错误码初值为无效的功能码
51. **for**( i = 0; i < MB\_FUNC\_HANDLERS\_MAX; i++ )
52. {
53. /\* No more function handlers registered. Abort. \*/
54. **if**( xFuncHandlers[i].ucFunctionCode == 0 )
55. {
56. **break**;
57. }
58. **else** **if**( xFuncHandlers[i].ucFunctionCode == ucFunctionCode ) /\*根据报文中的功能码，处理报文\*/
59. {
60. eException = xFuncHandlers[i].pxHandler( ucMBFrame, &usLength );/\*对接收到的报文进行解析\*/
61. **break**;
62. }
63. }
65. /\* If the request was not sent to the broadcast address we
66. \* return a reply. \*/
67. **if**( ucRcvAddress != MB\_ADDRESS\_BROADCAST )
68. {
69. **if**( eException != MB\_EX\_NONE )     /\*接收到的报文有错误\*/
70. {
71. /\* An exception occured. Build an error frame. \*/
72. usLength = 0;                                                        /\*响应发送数据的首字节为从机地址\*/
73. ucMBFrame[usLength++] = ( **UCHAR** )( ucFunctionCode | MB\_FUNC\_ERROR ); /\*响应发送数据帧的第二个字节，功能码最高位置1\*/
74. ucMBFrame[usLength++] = eException;                                  /\*响应发送数据帧的第三个字节为错误码标识\*/
75. }
76. **if**( ( eMBCurrentMode == MB\_ASCII ) && MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS )
77. {
78. vMBPortTimersDelay( MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS );
79. }
80. eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength );           /\*modbus从机响应函数,发送响应给主机\*/
81. }
82. **break**;
84. **case** EV\_FRAME\_SENT:
85. **break**;
86. }
87. }
88. **return** MB\_ENOERR;
89. }

**eMBRTUReceive()函数：(mbrtu.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*eMBPoll函数轮询到EV\_FRAME\_RECEIVED事件时,调用peMBFrameReceiveCur()，此函数是用户为函数指针peMBFrameReceiveCur()的赋值
2. \*此函数完成的功能：从一帧数据报文中，取得modbus从机地址给pucRcvAddress，PDU报文的长度给pusLength，PDU报文的首地址给pucFrame，函数
3. \*形参全部为地址传递\*/
4. eMBErrorCode
5. eMBRTUReceive( **UCHAR** \* pucRcvAddress, **UCHAR** \*\* pucFrame, **USHORT** \* pusLength )
6. {
7. **BOOL**            xFrameReceived = FALSE;
8. eMBErrorCode    eStatus = MB\_ENOERR;
10. ENTER\_CRITICAL\_SECTION();
11. assert( usRcvBufferPos < MB\_SER\_PDU\_SIZE\_MAX );    /\*断言宏，判断接收到的字节数<256，如果>256，终止程序\*/
13. /\* Length and CRC check \*/
14. **if**( ( usRcvBufferPos >= MB\_SER\_PDU\_SIZE\_MIN )
15. && ( usMBCRC16( ( **UCHAR** \* ) ucRTUBuf, usRcvBufferPos ) == 0 ) )
16. {
17. /\* Save the address field. All frames are passed to the upper layed
18. \* and the decision if a frame is used is done there.
19. \*/
20. \*pucRcvAddress = ucRTUBuf[MB\_SER\_PDU\_ADDR\_OFF];     //取接收到的第一个字节，modbus从机地址
22. /\* Total length of Modbus-PDU is Modbus-Serial-Line-PDU minus
23. \* size of address field and CRC checksum.
24. \*/
25. \*pusLength = ( **USHORT** )( usRcvBufferPos - MB\_SER\_PDU\_PDU\_OFF - MB\_SER\_PDU\_SIZE\_CRC ); //减3
27. /\* Return the start of the Modbus PDU to the caller. \*/
28. \*pucFrame = ( **UCHAR** \* ) & ucRTUBuf[MB\_SER\_PDU\_PDU\_OFF];
29. xFrameReceived = TRUE;
30. }
31. **else**
32. {
33. eStatus = MB\_EIO;
34. }
36. EXIT\_CRITICAL\_SECTION();
37. **return** eStatus;
38. }

xMBPortEventPost()函数：(portevent.c)

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. **BOOL**
2. xMBPortEventPost( eMBEventType eEvent )
3. {
4. xEventInQueue = TRUE;
5. eQueuedEvent = eEvent;
6. **return** TRUE;
7. }

**xFuncHandlers[i]是结构体数组，存放的是功能码以及对应的报文解析函数，原型如下：**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. **typedef** **struct**
2. {
3. **UCHAR**           ucFunctionCode;
4. pxMBFunctionHandler pxHandler;
5. } xMBFunctionHandler;

**以下列举读线圈函数举例：**

**eMBFuncReadCoils()读线圈寄存器函数： (mbfunccoils.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. #if MB\_FUNC\_READ\_COILS\_ENABLED > 0
3. eMBException
4. eMBFuncReadCoils( **UCHAR** \* pucFrame, **USHORT** \* usLen )
5. {
6. **USHORT**          usRegAddress;
7. **USHORT**          usCoilCount;
8. **UCHAR**           ucNBytes;
9. **UCHAR**          \*pucFrameCur;
11. eMBException    eStatus = MB\_EX\_NONE;
12. eMBErrorCode    eRegStatus;
14. **if**( \*usLen == ( MB\_PDU\_FUNC\_READ\_SIZE + MB\_PDU\_SIZE\_MIN ) )
15. {
16. /\*线圈寄存器的起始地址\*/
17. usRegAddress = ( **USHORT** )( pucFrame[MB\_PDU\_FUNC\_READ\_ADDR\_OFF] << 8 );
18. usRegAddress |= ( **USHORT** )( pucFrame[MB\_PDU\_FUNC\_READ\_ADDR\_OFF + 1] );
19. //usRegAddress++;
21. /\*线圈寄存器个数\*/
22. usCoilCount = ( **USHORT** )( pucFrame[MB\_PDU\_FUNC\_READ\_COILCNT\_OFF] << 8 );
23. usCoilCount |= ( **USHORT** )( pucFrame[MB\_PDU\_FUNC\_READ\_COILCNT\_OFF + 1] );
25. /\* Check if the number of registers to read is valid. If not
26. \* return Modbus illegal data value exception.
27. \*/
28. /\*判断线圈寄存器个数是否合理\*/
29. **if**( ( usCoilCount >= 1 ) &&
30. ( usCoilCount < MB\_PDU\_FUNC\_READ\_COILCNT\_MAX ) )
31. {
32. /\* Set the current PDU data pointer to the beginning. \*/
33. /\*为发送缓冲pucFrameCur赋值\*/
34. pucFrameCur = &pucFrame[MB\_PDU\_FUNC\_OFF];
35. \*usLen = MB\_PDU\_FUNC\_OFF;
37. /\* First byte contains the function code. \*/
38. /\*响应报文第一个字节赋值为功能码0x01\*/
39. \*pucFrameCur++ = MB\_FUNC\_READ\_COILS;
40. \*usLen += 1;
42. /\* Test if the quantity of coils is a multiple of 8. If not last
43. \* byte is only partially field with unused coils set to zero. \*/
44. /\*usCoilCount%8有余数，ucNBytes加1,不够的位填充0\*/
45. **if**( ( usCoilCount & 0x0007 ) != 0 )
46. {
47. ucNBytes = ( **UCHAR** )( usCoilCount / 8 + 1 );
48. }
49. **else**
50. {
51. ucNBytes = ( **UCHAR** )( usCoilCount / 8 );
52. }
53. \*pucFrameCur++ = ucNBytes;
54. \*usLen += 1;
56. eRegStatus =
57. eMBRegCoilsCB( pucFrameCur, usRegAddress, usCoilCount,
58. MB\_REG\_READ );
60. /\* If an error occured convert it into a Modbus exception. \*/
61. **if**( eRegStatus != MB\_ENOERR )
62. {
63. eStatus = prveMBError2Exception( eRegStatus );
64. }
65. **else**
66. {
67. /\* The response contains the function code, the starting address
68. \* and the quantity of registers. We reuse the old values in the
69. \* buffer because they are still valid. \*/
70. \*usLen += ucNBytes;;
71. }
72. }
73. **else**
74. {
75. eStatus = MB\_EX\_ILLEGAL\_DATA\_VALUE;
76. }
77. }
78. **else**
79. {
80. /\* Can't be a valid read coil register request because the length
81. \* is incorrect. \*/
82. eStatus = MB\_EX\_ILLEGAL\_DATA\_VALUE;
83. }
84. **return** eStatus;
85. }

**至此：报文解析结束，得到ucMBFrame响应缓冲和usLength响应报文长度，等待发送报文；**

**4：发送响应报文**

**解析完一帧完整的报文后，eMBPoll()函数中调用peMBFrameSendCur()函数进行响应，eMBFrameSendCur()是函数指针，最终会调用eMBRTUSend()函数发送响应；**

**eMBRTUSend()函数：**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

1. /\*函数功能
2. \*1:对响应报文PDU前面加上从机地址;
3. \*2:对响应报文PDU后加上CRC校;
4. \*3:使能发送，启动传输;
5. \*/
6. eMBErrorCode
7. eMBRTUSend( **UCHAR** ucSlaveAddress, **const** **UCHAR** \* pucFrame, **USHORT** usLength )
8. {
9. eMBErrorCode    eStatus = MB\_ENOERR;
10. **USHORT**          usCRC16;
12. ENTER\_CRITICAL\_SECTION(  );
14. /\* Check if the receiver is still in idle state. If not we where to
15. \* slow with processing the received frame and the master sent another
16. \* frame on the network. We have to abort sending the frame.
17. \*/
18. **if**( eRcvState == STATE\_RX\_IDLE )
19. {
20. /\* First byte before the Modbus-PDU is the slave address. \*/
21. /\*在协议数据单元前加从机地址\*/
22. pucSndBufferCur = ( **UCHAR** \* ) pucFrame - 1;
23. usSndBufferCount = 1;
25. /\* Now copy the Modbus-PDU into the Modbus-Serial-Line-PDU. \*/
26. pucSndBufferCur[MB\_SER\_PDU\_ADDR\_OFF] = ucSlaveAddress;
27. usSndBufferCount += usLength;
29. /\* Calculate CRC16 checksum for Modbus-Serial-Line-PDU. \*/
30. usCRC16 = usMBCRC16( ( **UCHAR** \* ) pucSndBufferCur, usSndBufferCount );
31. ucRTUBuf[usSndBufferCount++] = ( **UCHAR** )( usCRC16 & 0xFF );
32. ucRTUBuf[usSndBufferCount++] = ( **UCHAR** )( usCRC16 >> 8 );
34. /\* Activate the transmitter. \*/
35. eSndState = STATE\_TX\_XMIT;                         //发送状态
36. xMBPortSerialPutByte( ( **CHAR** )\*pucSndBufferCur );  /\*发送一个字节的数据，进入发送中断函数，启动传输\*/
37. pucSndBufferCur++;                                 /\* next byte in sendbuffer. \*/
38. usSndBufferCount--;
39. vMBPortSerialEnable( FALSE, TRUE );                /\*使能发送，禁止接收\*/
40. }
41. **else**
42. {
43. eStatus = MB\_EIO;
44. }
45. EXIT\_CRITICAL\_SECTION(  );
46. **return** eStatus;
47. }

**进入发送中断，串口发送中断中调用prvvUARTTxReadyISR()函数，继续调用pxMBFrameCBTransmitterEmpty()函数，pxMBFrameCBTransmitterEmpty为函数指针，最终调用xMBRTUTransmitFSM()函数；**

**xMBRTUTransmitFSM()函数：(mbrtu.c)**

**[cpp]** [view plain](https://blog.csdn.net/Golf_research/article/details/52240739) [copy](https://blog.csdn.net/Golf_research/article/details/52240739)

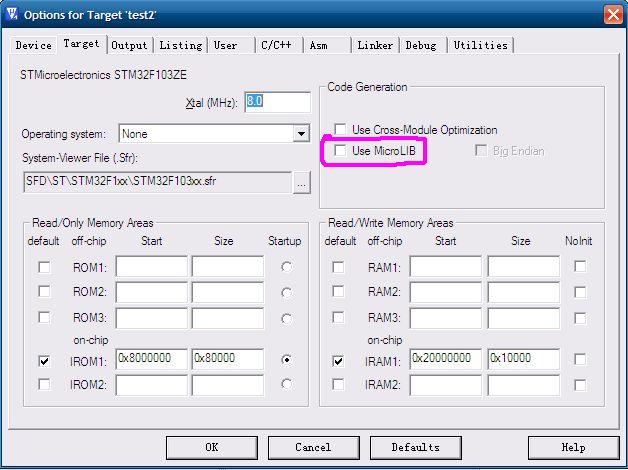
1. **BOOL**
2. xMBRTUTransmitFSM( **void** )
3. {
4. **BOOL**            xNeedPoll = FALSE;
6. assert( eRcvState == STATE\_RX\_IDLE );
8. **switch** ( eSndState )
9. {
10. /\* We should not get a transmitter event if the transmitter is in
11. \* idle state.\*/
12. **case** STATE\_TX\_IDLE:        /\*发送器处于空闲状态，使能接收，禁止发送\*/
13. /\* enable receiver/disable transmitter. \*/
14. vMBPortSerialEnable( TRUE, FALSE );
15. **break**;
17. **case** STATE\_TX\_XMIT:        /\*发送器处于发送状态,在从机发送函数eMBRTUSend中赋值STATE\_TX\_XMIT\*/
18. /\* check if we are finished. \*/
19. **if**( usSndBufferCount != 0 )
20. {
21. //发送数据
22. xMBPortSerialPutByte( ( **CHAR** )\*pucSndBufferCur );
23. pucSndBufferCur++;  /\* next byte in sendbuffer. \*/
24. usSndBufferCount--;
25. }
26. **else**
27. {
28. //传递任务，发送完成
29. xNeedPoll = xMBPortEventPost( EV\_FRAME\_SENT );   /\*协议栈事件状态赋值为EV\_FRAME\_SENT,发送完成事件,eMBPoll函数会对此事件进行处理\*/
30. /\* Disable transmitter. This prevents another transmit buffer
31. \* empty interrupt. \*/
32. vMBPortSerialEnable( TRUE, FALSE );              /\*使能接收，禁止发送\*/
33. eSndState = STATE\_TX\_IDLE;                       /\*发送器状态为空闲状态\*/
34. }
35. **break**;
36. }
38. **return** xNeedPoll;
39. }

**至此：协议栈准备工作，从机接受报文，解析报文，从机发送响应报文四部分结束；**

***5．5***

STM32程序添加printf函数后无法运行的解决方法（串口实验）

标准库函数的默认输出设备是显示器，要实现在串口或LCD输出，必须重定义标准库函数里调用的与输出设备相关的函数.

例如:printf输出到串口，需要将fputc里面的输出指向串口(重定向),方法如下:  
  
#ifdef \_\_GNUC\_\_  
  
  
#define PUTCHAR\_PROTOTYPE int \_\_io\_putchar(int ch)  
#else  
#define PUTCHAR\_PROTOTYPE int fputc(int ch, FILE \*f)  
#endif   
  
  
PUTCHAR\_PROTOTYPE  
{  
USART\_SendData(USART1, (uint8\_t) ch);  
  
while (USART\_GetFlagStatus(USART1, USART\_FLAG\_TC) == RESET);  
return ch;  
}  
  
  
因printf()之类的函数，使用了半主机模式。使用标准库会导致程序无法运行,以下是解决方法:  
  
  
方法1.使用微库,因为使用微库的话,不会使用半主机模式.  
  
  
  
方法2.仍然使用标准库,在主程序添加下面代码:  
  
  
#pragma import(\_\_use\_no\_semihosting)   
\_sys\_exit(int x)   
{   
x = x;   
}   
struct \_\_FILE   
{   
int handle;   
  
  
  
};   
  
FILE \_\_stdout;  
另一个高手的讲解  
刚开始学stm32，顺着gpio、uart。。。的顺序慢慢爬  
初始化的方法学习了马老师的STM32\_Init.h\*\*\*\*，自己英文还可以，加上avr的基础还不错，所以gpio和时钟配置都很顺利  
碰到uart就头大了，看到各种例程里都是printf（）函数，自己也想用，毕竟是avr想用却开销不了的东西。但是我自己写的程序里一旦出现printf，单片机的不干活了。查论坛首先发现要重定义fputc函数，照做了，还是不行。  
后来怀疑是uart1初始化问题，用自己写的put\_c函数却没问题。  
后来又发现一种说法，需要避免使用semihosting(半主机模式），我也把代码加进去了（改fputc去掉了），还是不行。  
再一想，重定义fputc是绝对必须的，加上了之后问题解决，成功使用printf("(敏感词0373) \n");输出了，哈哈  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
以上废话，可以不看。  
简单地说：想在mdk 3.80a中用printf，需要同时重定义fputc函数和避免使用semihosting(半主机模式），   
论坛里应该有完整介绍这个的帖子，但是我没搜到，也许是沉了。重发出来希望能帮上像我这样的菜鸟们。  
  
需要添加以下代码  
  
  
#pragma import(\_\_use\_no\_semihosting)   
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
\*标准库需要的支持函数   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   
struct \_\_FILE   
{   
int handle;   
/\* Whatever you require here. If the only file you are using is \*/   
/\* standard output using printf() for debugging, no file handling \*/   
/\* is required. \*/   
};   
/\* FILE is typedef’ d in stdio.h. \*/   
FILE \_\_stdout;   
  
/// <summary>   
/// 定义\_sys\_exit()以避免使用半主机模式   
/// </summary>   
/// <param name="x"></param>   
/// <returns></returns>   
\_sys\_exit(int x)   
{   
x = x;   
}   
  
  
  
int fputc(int ch, FILE \*f)  
{  
    //USART\_SendData(USART1, (u8) ch);  
    USART1->DR = (u8) ch;  
      
    /\* Loop until the end of transmission \*/  
    while(USART\_GetFlagStatus(USART1, USART\_FLAG\_TXE) == RESET)  
    {  
    }  
  
    return ch;  
}

***5.7***

[modbus](http://www.eemaker.com/tag/modbus)完整支持很多功能码，但是实际在应用的时候常用的也就那么几个。具体如下：

    0x01: 读线圈寄存器

    0x02: 读离散输入寄存器

    0x03: 读保持寄存器

    0x04: 读输入寄存器

    0x05: 写单个线圈寄存器

    0x06: 写单个保持寄存器

    0x0f:  写多个线圈寄存器

    0x10: 写多个保持寄存器

    如上所示一共8种功能码。这其中有涉及到线圈、离散输入、保持、输入四种寄存器。这名字也不知道谁起的，让人看了一点不通俗易懂，搞得晕晕乎乎。实际上你要是看清他的本质就很简单了。下面分别解释一下：

**线圈寄存器**，实际上就可以类比为开关量，没一个bit都对应一个信号的开关状态。所以一个byte就可以同时控制8路的信号。比如控制外部8路io的高低。 线圈寄存器支持读也支持写，写在功能码里面又分为写单个线圈寄存器和写多个线圈寄存器。对应上面的功能码也就是：0x01  0x05  0x0f

**离散输入寄存器**，如果线圈寄存器理解了这个自然也明白了。离散输入寄存器就相当于线圈寄存器的只读模式，他也是每个bit表示一个开关量，而他的开关量只能读取输入的开关信号，是不能够写的。比如我读取外部按键的按下还是松开。所以功能码也简单就一个读的 0x02

**保持寄存器**，这个寄存器的单位不再是bit而是两个byte，也就是可以存放具体的数据量的，并且是可读写的。比如我我设置时间年月日，不但可以写也可以读出来现在的时间。写也分为单个写和多个写，所以功能码有对应的三个：0x03 0x06 0x10

**输入寄存器，**只剩下这最后一个了，这个和保持寄存器类似，但是也是只支持读而不能写。一个寄存器也是占据两个byte的空间。类比我我通过读取输入寄存器获取现在的AD采集值。对应的功能码也就一个 0x04

