**FreeModbus移植 经验分享**

**一   为什么要移植Freemodbus**

         为什么要移植Freemodbus，这个问题需要从两个方面来回答。第一，modbus是一个非常好的应用层协议，它很简洁也相对完善。对于还没有接触过modbus的朋友来说，我非常不建议直接移植freemodbus，应该耐心的从modbus文档入手，并充分把握身边的所有资源，例如PLC的中modbus部分。第二，其实嵌入式系统的通信协议可以自己制定，但是通过实践发现自己定制的协议漏洞百出，尤其是扩展极为困难。我始终认为借鉴他人的经验是很好的途径。借鉴他人成熟的代码，可以减少调试的时间，实现的功能也多了不少。

         个人观点，仅供参考。

         freemodbus小提示

         freemodbus只能使用从机功能。freemodbus更适合嵌入式系统，虽然例子中也有WIN32的例子，如果想要做PC机程序并实现主机功能，推荐使用另一个modbus库——NMODBUS，使用C#开发。同样WINFORM也可以通过自己编写串口代码实现modbus功能，但是这会花费很长的时间，可能是一周也可能是一个月，如果使用现成的代码库，那么开发时间可能只有10分钟。         自己整理的modbus协议http://img.orz520.com/t011c48a04ddd73f7c3.gif[MODBUS 协议整理.pdf](http://www.amobbs.com/forum.php?mod=attachment&aid=NDE3NjR8NWUyZGEwYzZ8MTM5NDcyMTU0MnwyNzU2MHw1NDkxNjE1) *(111.46 KB, 下载次数: 3418)*   
        代码参考了这个帖子，感谢你的分享。[点击这里](http://www.amobbs.com/forum.php?mod=viewthread&tid=4848191&highlight=FreeModbus" \t "_blank)**二  freeemodbus中如何通过串口发送和接收数据**

         freemodbus通过串口中断的方式接收和发送数据。采用这种做法我想可以节省程序等待的时间，并且也短充分使用CPU的资源。串口中断接收毋庸置疑，在中断服务函数中把数据保存在数组中，以便稍后处理。但是串口发送中断使用哪种形式？串口发送中断至少有两种方式，第一种，数据寄存器空中断，只要数据寄存器为空并且中断屏蔽位置位，那么中断就会发生；第二种，发送完成中断，若数据寄存器的数据发送完成并且中断屏蔽位置位，那么中断也会发送。我非常建议各位使用串口发送完成中断。freemodbus多使用RS485通信中，从机要么接收要么发送，多数情况下从机处于接收状态，要有数据发送时才进入发送状态。进入发送状态时，数据被一个一个字节发送出去，当最后一个字节被发送出去之后，从机再次进入接收状态。如果使用发送寄存器为空中断，还需要使用其他的方法才可以判断最后一个字节的数据是否发送完成。如果使用数据寄存器为空中断，那么将很有可能丢失最后一个字节。（马潮老师的AVR图书中也推荐使用发送完成中断，交流性质的文章，就没有参考文献了。）

**二  freemodbus中如何判断帧结束**

         大家应该清楚，modbus协议中没有明显的开始符和结束符，而是通过帧与帧之间的间隔时间来判断的。如果在指定的时间内，没有接收到新的字符数据，那么就认为收到了新的帧。接下来就可以处理数据了，首当其冲的就是判断帧的合法性。Modbus通过时间来判断帧是否接受完成，自然需要单片机中的定时器配合。

**三   整体代码**

下面给出一个STM32平台上使用FREEMODBUS最简单的例子，操作保持寄存器，此时操作指令可以为03，06和16；

1. #include "stm32f10x.h"
2. #include
3. #include "mb.h"
4. #include "mbutils.h"
5. //保持寄存器起始地址
6. #define REG\_HOLDING\_START 0x0000
7. //保持寄存器数量
8. #define REG\_HOLDING\_NREGS 8
9. //保持寄存器内容
10. uint16\_t usRegHoldingBuf[REG\_HOLDING\_NREGS]
11. = {0x147b,0x3f8e,0x147b,0x400e,0x1eb8,0x4055,0x147b,0x408e};
12. int main(void)
13. {
14. //初始化 RTU模式 从机地址为1 USART1 9600 无校验
15. eMBInit(MB\_RTU, 0x01, 0x01, 9600, MB\_PAR\_NONE);
16. //启动FreeModbus
17. eMBEnable();
18. while (1)
19. {
20. //FreeMODBUS不断查询
21. eMBPoll();
22. }
23. }
24. /\*\*
25. \* @brief 保持寄存器处理函数，保持寄存器可读，可读可写
26. \* @param pucRegBuffer 读操作时--返回数据指针，写操作时--输入数据指针
27. \* usAddress 寄存器起始地址
28. \* usNRegs 寄存器长度
29. \* eMode 操作方式，读或者写
30. \* @retval eStatus 寄存器状态
31. \*/
32. eMBErrorCode
33. eMBRegHoldingCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNRegs,
34. eMBRegisterMode eMode )
35. {
36. //错误状态
37. eMBErrorCode eStatus = MB\_ENOERR;
38. //偏移量
39. int16\_t iRegIndex;
40. //判断寄存器是不是在范围内
41. if( ( (int16\_t)usAddress >= REG\_HOLDING\_START ) \
42. && ( usAddress + usNRegs <= REG\_HOLDING\_START + REG\_HOLDING\_NREGS ) )
43. {
44. //计算偏移量
45. iRegIndex = ( int16\_t )( usAddress - REG\_HOLDING\_START);
46. switch ( eMode )
47. {
48. //读处理函数
49. case MB\_REG\_READ:
50. while( usNRegs > 0 )
51. {
52. \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] >> 8 );
53. \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] & 0xFF );
54. iRegIndex++;
55. usNRegs--;
56. }
57. break;
58. //写处理函数
59. case MB\_REG\_WRITE:
60. while( usNRegs > 0 )
61. {
62. usRegHoldingBuf[iRegIndex] = \*pucRegBuffer++ << 8;
63. usRegHoldingBuf[iRegIndex] |= \*pucRegBuffer++;
64. iRegIndex++;
65. usNRegs--;
66. }
67. break;
68. }
69. }
70. else
71. {
72. //返回错误状态
73. eStatus = MB\_ENOREG;
74. }
75. return eStatus;
76. }

*复制代码*

先给大家一个整体的印象，先让大家会使用FREEMODBUS，再详细描述细节

//保持寄存器起始地址

#define REG\_HOLDING\_START     0x0000

//保持寄存器数量

#define REG\_HOLDING\_NREGS     8

这两个宏定义，决定了保持寄存器的起始地址和总个数。需要强调的是，modbus寄存器的地址有两套规则，一套称为PLC地址，为5位十进制数，例如40001。另一套是协议地址，PLC地址40001意味着该参数类型为保持寄存器，协议地址为0x0000，这里面有对应关系，**去掉PLC地址的最高位，然后剩下的减1即可**。这会存在一个问题，PLC地址30002和PLC地址40002的协议地址同为0x0001，此时访问时是不是会冲突呢。亲们，当然不会了，30001为输入寄存器，需要使用04指令访问，而40001为保持寄存器，可以使用03、06和16指令访问。所以，用好modbus还是要熟悉协议本生，切不可着急。

//保持寄存器内容

uint16\_t usRegHoldingBuf[REG\_HOLDING\_NREGS]

= {0x147b,0x3f8e,0x147b,0x400e,0x1eb8,0x4055,0x147b,0x408e};

接下来定义了保持寄存器的内容，在这里请大家注意了，保持寄存器为无符号16位数据。在测试的情况下，我随便找了一些数据进行测试。看数据的本质似乎看不出说明规律，但是usRegHoldingBuf却是以16进制保存了浮点数。

1. int main(void)
2. {
3. //初始化 RTU模式 从机地址为1 USART1 9600 无校验
4. eMBInit(MB\_RTU, 0x01, 0x01, 9600, MB\_PAR\_NONE);
5. //启动FreeModbus
6. eMBEnable();
7. while (1)
8. {
9. //FreeMODBUS不断查询
10. eMBPoll();
11. }
12. }

*复制代码*

接下来就进入主函数部分。有三个FREEMODBUS提供的函数，eMBInit，eMBEnable和eMBPoll。eMBInit为modbus的初始化函数，eMBEnable为modbus的使能函数，而eMBPoll为modbus的查询函数，eMBPoll也是非常单纯的函数，查询是否有数据帧到达，如果有数据到达，便进行相依的处理。再次观察这几个函数，只有eMBInit有很多的参数，这些参数和位于系统底层的硬件有关，这个应该引起移植过程的更多关注。下面几个章节再议。

1. eMBErrorCode
2. eMBRegHoldingCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNRegs,
3. eMBRegisterMode eMode )
4. {
5. //错误状态
6. eMBErrorCode eStatus = MB\_ENOERR;
7. //偏移量
8. int16\_t iRegIndex;
9. //判断寄存器是不是在范围内
10. if( ( (int16\_t)usAddress >= REG\_HOLDING\_START ) \
11. && ( usAddress + usNRegs <= REG\_HOLDING\_START + REG\_HOLDING\_NREGS ) )
12. {
13. //计算偏移量
14. iRegIndex = ( int16\_t )( usAddress - REG\_HOLDING\_START);
15. switch ( eMode )
16. {
17. //读处理函数
18. case MB\_REG\_READ:
19. while( usNRegs > 0 )
20. {
21. \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] >> 8 );
22. \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] & 0xFF );
23. iRegIndex++;
24. usNRegs--;
25. }
26. break;
27. //写处理函数
28. case MB\_REG\_WRITE:
29. while( usNRegs > 0 )
30. {
31. usRegHoldingBuf[iRegIndex] = \*pucRegBuffer++ << 8;
32. usRegHoldingBuf[iRegIndex] |= \*pucRegBuffer++;
33. iRegIndex++;
34. usNRegs--;
35. }
36. break;
37. }
38. }
39. else
40. {
41. //返回错误状态
42. eStatus = MB\_ENOREG;
43. }
44. return eStatus;
45. }

*复制代码*

最后，如果收到一个有效的数据帧，那么就可以开始处理了。

第一步，判断寄存器的地址是否在合法的范围内。

  if( ( (int16\_t)usAddress >= REG\_HOLDING\_START ) \

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

第二步，判断需要操作寄存器的偏移地址。

         给个例子可以迅速的说明问题，例如访问寄存器的起始地址为0x0002，保持寄存器的起始地址为0x0000，那么这个访问的偏移量为2，程序就从保持寄存器数组的第2个（从0开始）开始操作。

第三步，读写操作分开处理

      case MB\_REG\_READ:

        while( usNRegs > 0 )

        {

          \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] >> 8 );

          \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] & 0xFF );

          iRegIndex++;

          usNRegs--;

        }

        break;

         以读操作为例，代码不多说了，请大家注意操作的顺序。保持寄存器以16位形式保存，但是modbus通信时以字节为单位，高位字节数据在前，低位数据字节在后。

**四   串口相关部分代码编写**

         串口部分的代码编写比较常规，主要有三个函数，串口初始化，串口数据发送和串口数据接收。除了以上三个函数之外，还有串口中断服务函数。

1. /\*\*
2. \* @brief 串口初始化
3. \* @param ucPORT 串口号
4. \* ulBaudRate 波特率
5. \* ucDataBits 数据位
6. \* eParity 校验位
7. \* @retval None
8. \*/
9. BOOL
10. xMBPortSerialInit( UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity eParity )
11. {
12. (void)ucPORT; //不修改串口
13. (void)ucDataBits; //不修改数据位长度
14. (void)eParity; //不修改校验格式
15. GPIO\_InitTypeDef GPIO\_InitStructure;
16. USART\_InitTypeDef USART\_InitStructure;
17. //使能USART1，GPIOA
18. RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA |
19. RCC\_APB2Periph\_USART1, ENABLE);
20. //GPIOA9 USART1\_Tx
21. GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_9;
22. GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;
23. GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_AF\_PP; //推挽输出
24. GPIO\_Init(GPIOA, &GPIO\_InitStructure);
25. //GPIOA.10 USART1\_Rx
26. GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_10;
27. GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;
28. GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_IN\_FLOATING; //浮动输入
29. GPIO\_Init(GPIOA, &GPIO\_InitStructure);
30. USART\_InitStructure.USART\_BaudRate = ulBaudRate; //只修改波特率
31. USART\_InitStructure.USART\_WordLength = USART\_WordLength\_8b;
32. USART\_InitStructure.USART\_StopBits = USART\_StopBits\_1;
33. USART\_InitStructure.USART\_Parity = USART\_Parity\_No;
34. USART\_InitStructure.USART\_HardwareFlowControl = USART\_HardwareFlowControl\_None;
35. USART\_InitStructure.USART\_Mode = USART\_Mode\_Rx | USART\_Mode\_Tx;
36. //串口初始化
37. USART\_Init(USART1, &USART\_InitStructure);
38. //使能USART1
39. USART\_Cmd(USART1, ENABLE);
40. NVIC\_InitTypeDef NVIC\_InitStructure;
41. NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_1);
42. //设定USART1 中断优先级
43. NVIC\_InitStructure.NVIC\_IRQChannel = USART1\_IRQn;
44. NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 0;
45. NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 0;
46. NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;
47. NVIC\_Init(&NVIC\_InitStructure);
48. //最后配置485发送和接收模式
49. RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOD, ENABLE);
50. //GPIOD.8
51. GPIO\_InitStructure.GPIO\_Pin = GPIO\_Pin\_8;
52. GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;
53. GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;
54. GPIO\_Init(GPIOD, &GPIO\_InitStructure);
55. return TRUE;
56. }

*复制代码*

传入的参数有端口号，波特率，数据位和校验位，可以根据实际的情况修改代码。在这里我并没有修改其他参数，至于传入的波特率是有效的。除了配置串口的相关参数之外，还需要配置串口的中断优先级。最后，由于使用485模式，还需要一个发送接收控制端，该IO配置为推挽输出模式。

1. /\*\*
2. \* @brief 控制接收和发送状态
3. \* @param xRxEnable 接收使能、
4. \* xTxEnable 发送使能
5. \* @retval None
6. \*/
7. void
8. vMBPortSerialEnable( BOOL xRxEnable, BOOL xTxEnable )
9. {
10. if(xRxEnable)
11. {
12. //使能接收和接收中断
13. USART\_ITConfig(USART1, USART\_IT\_RXNE, ENABLE);
14. //MAX485操作 低电平为接收模式
15. GPIO\_ResetBits(GPIOD,GPIO\_Pin\_8);
16. }
17. else
18. {
19. USART\_ITConfig(USART1, USART\_IT\_RXNE, DISABLE);
20. //MAX485操作 高电平为发送模式
21. GPIO\_SetBits(GPIOD,GPIO\_Pin\_8);
22. }
23. if(xTxEnable)
24. {
25. //使能发送完成中断
26. USART\_ITConfig(USART1, USART\_IT\_TC, ENABLE);
27. }
28. else
29. {
30. //禁止发送完成中断
31. USART\_ITConfig(USART1, USART\_IT\_TC, DISABLE);
32. }
33. }

*复制代码*

由于485使用半双工模式，从机一般处于接收状态，有数据发送时才会进入发送模式。在FreeModbus中有专门的控制接收和发送状态的函数，在这里不但可以打开或关闭接收和发送中断，还可以控制485收发芯片的发送接收端口。代码非常简单，但是还是建议各位使用发送完成中断。

1. BOOL
2. xMBPortSerialPutByte( CHAR ucByte )
3. {
4. //发送数据
5. USART\_SendData(USART1, ucByte);
6. return TRUE;
7. }
8. BOOL
9. xMBPortSerialGetByte( CHAR \* pucByte )
10. {
11. //接收数据
12. \*pucByte = USART\_ReceiveData(USART1);
13. return TRUE;
14. }
15. xMBPortSerialPutByte和xMBPortSerialGetByte两个函数用于串口发送和接收数据，在这里只要调用STM32的库函数即可。
16. static void prvvUARTTxReadyISR( void )
17. {
18. //mb.c eMBInit函数中
19. //pxMBFrameCBTransmitterEmpty = xMBRTUTransmitFSM
20. //发送状态机
21. pxMBFrameCBTransmitterEmpty();
22. }
23. static void prvvUARTRxISR( void )
24. {
25. //mb.c eMBInit函数中
26. //pxMBFrameCBByteReceived = xMBRTUReceiveFSM
27. //接收状态机
28. pxMBFrameCBByteReceived();
29. }
30. void USART1\_IRQHandler(void)
31. {
32. //发生接收中断
33. if(USART\_GetITStatus(USART1, USART\_IT\_RXNE) == SET)
34. {
35. prvvUARTRxISR();
36. //清除中断标志位
37. USART\_ClearITPendingBit(USART1, USART\_IT\_RXNE);
38. }
39. //发生完成中断
40. if(USART\_GetITStatus(USART1, USART\_IT\_TC) == SET)
41. {
42. prvvUARTTxReadyISR();
43. //清除中断标志
44. USART\_ClearITPendingBit(USART1, USART\_IT\_TC);
45. }
46. }

*复制代码*

若进入串口中断服务函数，则要调用FreeModbus中响应的函数，串口接收中断服务函数对应prvvUARTRxISR()，其代码如下

1. static void prvvUARTRxISR( void )
2. {
3. //mb.c eMBInit函数中
4. //pxMBFrameCBByteReceived = xMBRTUReceiveFSM
5. //接收状态机
6. pxMBFrameCBByteReceived();
7. }

*复制代码*

在prvvUARTRxISR中又调用了pxMBFrameCBByteReceived()，其实pxMBFrameCBTransmitterEmpty()并不是一个函数，而是一个函数指针。其定义如下，请注意函数指针的声明和函数声明的区别。

BOOL( \*pxMBFrameCBTransmitterEmpty ) ( void );

在mb.c文件的eMBInit函数完成赋值。一般情况下都会选择RTU模式，那么pxMBFrameCBByteReceived就和xMBRTUReceiveFSM等价了，

pxMBFrameCBByteReceived = xMBRTUReceiveFSM;

同理，若发生串口发送完成中断，该中断服务函数对应prvvUARTTxReadyISR，其代码如下

1. static void prvvUARTTxReadyISR( void )
2. {
3. //mb.c eMBInit函数中
4. //pxMBFrameCBTransmitterEmpty = xMBRTUTransmitFSM
5. //发送状态机
6. pxMBFrameCBTransmitterEmpty();
7. }

*复制代码*

在prvvUARTTxReadyISR中又调用了pxMBFrameCBTransmitterEmpty()，pxMBFrameCBTransmitterEmpty也是函数指针，在eMBInit函数完成赋值，它等价于xMBRTUTransmitFSM。

         特别提醒，由于我使用的是串口发送完成中断，想要进入该中断服务函数，需要发送一个字节的数据并启动串口发送中断，代码还需要少许修改。在mbRTU.c的eMBRTUSend中稍作修改，代码如下。

*复制代码*

1. /\* First byte before the Modbus-PDU is the slave address. \*/
2. pucSndBufferCur = ( UCHAR \* ) pucFrame - 1;
3. usSndBufferCount = 1;
4. /\* Now copy the Modbus-PDU into the Modbus-Serial-Line-PDU. \*/
5. pucSndBufferCur[MB\_SER\_PDU\_ADDR\_OFF] = ucSlaveAddress;
6. usSndBufferCount += usLength;
7. /\* Calculate CRC16 checksum for Modbus-Serial-Line-PDU. \*/
8. usCRC16 = usMBCRC16( ( UCHAR \* ) pucSndBufferCur, usSndBufferCount );
9. ucRTUBuf[usSndBufferCount++] = ( UCHAR )( usCRC16 & 0xFF );
10. ucRTUBuf[usSndBufferCount++] = ( UCHAR )( usCRC16 >> 8 );
11. /\* Activate the transmitter. \*/
12. //发送状态转换，在中断中不断发送
13. eSndState = STATE\_TX\_XMIT;
14. //插入代码 启动第一次发送，这样才可以进入发送完成中断
15. xMBPortSerialPutByte( ( CHAR )\*pucSndBufferCur );
16. pucSndBufferCur++;
17. usSndBufferCount--;
18. //使能发送状态，禁止接收状态
19. vMBPortSerialEnable( FALSE, TRUE );

*复制代码*

写到这里给位可能看的不是很明白，建议研究一下FreeModbus的源码，稍作一些修改使用起来才会更加方便。

**五 定时器相关部分代码编写**        定时器的作用前面已经说明了，在这里就罗列一下相关的移植代码。定时器的代码要比串口的代码简单一些。

1. static void prvvTIMERExpiredISR( void );
2. BOOL
3. xMBPortTimersInit( USHORT usTim1Timerout50us )
4. {
5. TIM\_TimeBaseInitTypeDef  TIM\_TimeBaseStructure;
6. NVIC\_InitTypeDef NVIC\_InitStructure;
7. //
8. uint16\_t PrescalerValue = 0;
10. //使能定时器4时钟
11. RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_TIM4, ENABLE);
13. //定时器时间基配置说明
14. //HCLK为72MHz，APB1经过2分频为36MHz
15. //TIM4的时钟倍频后为72MHz（硬件自动倍频,达到最大）
16. //TIM4的分频系数为3599，时间基频率为72 / (1 + Prescaler) = 20KHz,基准为50us
17. //TIM最大计数值为usTim1Timerout50u
18. PrescalerValue = (uint16\_t) (SystemCoreClock / 20000) - 1;
19. //定时器1初始化
20. TIM\_TimeBaseStructure.TIM\_Period = (uint16\_t) usTim1Timerout50us;
21. TIM\_TimeBaseStructure.TIM\_Prescaler = PrescalerValue;
22. TIM\_TimeBaseStructure.TIM\_ClockDivision = 0;
23. TIM\_TimeBaseStructure.TIM\_CounterMode = TIM\_CounterMode\_Up;
24. TIM\_TimeBaseInit(TIM4, &TIM\_TimeBaseStructure);
25. //预装载使能
26. TIM\_ARRPreloadConfig(TIM4, ENABLE);
28. NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_1);
29. //定时器4中断优先级
30. NVIC\_InitStructure.NVIC\_IRQChannel = TIM4\_IRQn;
31. NVIC\_InitStructure.NVIC\_IRQChannelPreemptionPriority = 0;
32. NVIC\_InitStructure.NVIC\_IRQChannelSubPriority = 3;
33. NVIC\_InitStructure.NVIC\_IRQChannelCmd = ENABLE;
34. NVIC\_Init(&NVIC\_InitStructure);
36. //清除溢出中断标志位
37. TIM\_ClearITPendingBit(TIM4,TIM\_IT\_Update);
38. //定时器4溢出中断关闭
39. TIM\_ITConfig(TIM4, TIM\_IT\_Update, DISABLE);
40. //定时器4禁能
41. TIM\_Cmd(TIM4,  DISABLE);
42. return TRUE;
43. }
44. void
45. vMBPortTimersEnable( )
46. {
47. TIM\_ClearITPendingBit(TIM4, TIM\_IT\_Update);
48. TIM\_ITConfig(TIM4, TIM\_IT\_Update, ENABLE);
49. //设定定时器4的初始值
50. TIM\_SetCounter(TIM4,0x0000);
51. //定时器4启动
52. TIM\_Cmd(TIM4, ENABLE);
53. }
54. void
55. vMBPortTimersDisable(  )
56. {
57. TIM\_ClearITPendingBit(TIM4, TIM\_IT\_Update);
58. TIM\_ITConfig(TIM4, TIM\_IT\_Update, DISABLE);
59. TIM\_SetCounter(TIM4,0x0000);
60. //关闭定时器4
61. TIM\_Cmd(TIM4, DISABLE);
62. }
63. static void prvvTIMERExpiredISR( void )
64. {
65. ( void )pxMBPortCBTimerExpired();
66. }
67. void TIM4\_IRQHandler(void)
68. {
69. if (TIM\_GetITStatus(TIM4, TIM\_IT\_Update) != RESET)
70. {
71. //清除定时器T4溢出中断标志位
72. TIM\_ClearITPendingBit(TIM4, TIM\_IT\_Update);
73. prvvTIMERExpiredISR( );
74. }
75. }

复制代码

在这里请注意STM32的TIM有一个自动倍频的功能，如果APB1被分频的话，TIM的时钟就会自动倍频，当然再如何倍频也不应超过72MHz。

**六 各种寄存器的读或写函数**

         Modbus通信中，总共有四类的寄存器，开关输入寄存器，线圈寄存器，保持寄存器和输入寄存器。

1. /\*\*
2. \* @brief 输入寄存器处理函数，输入寄存器可读，但不可写。
3. \* @param pucRegBuffer 返回数据指针
4. \* usAddress 寄存器起始地址
5. \* usNRegs 寄存器长度
6. \* @retval eStatus 寄存器状态
7. \*/
8. eMBErrorCode
9. eMBRegInputCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNRegs )
10. {
11. eMBErrorCode eStatus = MB\_ENOERR;
12. int16\_t iRegIndex;
13. //查询是否在寄存器范围内
14. //为了避免警告，修改为有符号整数
15. if( ( (int16\_t)usAddress >= REG\_INPUT\_START ) \
16. && ( usAddress + usNRegs <= REG\_INPUT\_START + REG\_INPUT\_NREGS ) )
17. {
18. //获得操作偏移量，本次操作起始地址-输入寄存器的初始地址
19. iRegIndex = ( int16\_t )( usAddress - REG\_INPUT\_START );
20. //逐个赋值
21. while( usNRegs > 0 )
22. {
23. //赋值高字节
24. \*pucRegBuffer++ = ( uint8\_t )( usRegInputBuf[iRegIndex] >> 8 );
25. //赋值低字节
26. \*pucRegBuffer++ = ( uint8\_t )( usRegInputBuf[iRegIndex] & 0xFF );
27. //偏移量增加
28. iRegIndex++;
29. //被操作寄存器数量递减
30. usNRegs--;
31. }
32. }
33. else
34. {
35. //返回错误状态，无寄存器
36. eStatus = MB\_ENOREG;
37. }
38. return eStatus;
39. }
40. /\*\*
41. \* @brief 保持寄存器处理函数，保持寄存器可读，可读可写
42. \* @param pucRegBuffer 读操作时--返回数据指针，写操作时--输入数据指针
43. \* usAddress 寄存器起始地址
44. \* usNRegs 寄存器长度
45. \* eMode 操作方式，读或者写
46. \* @retval eStatus 寄存器状态
47. \*/
48. eMBErrorCode
49. eMBRegHoldingCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNRegs,
50. eMBRegisterMode eMode )
51. {
52. //错误状态
53. eMBErrorCode eStatus = MB\_ENOERR;
54. //偏移量
55. int16\_t iRegIndex;
56. //判断寄存器是不是在范围内
57. if( ( (int16\_t)usAddress >= REG\_HOLDING\_START ) \
58. && ( usAddress + usNRegs <= REG\_HOLDING\_START + REG\_HOLDING\_NREGS ) )
59. {
60. //计算偏移量
61. iRegIndex = ( int16\_t )( usAddress - REG\_HOLDING\_START );
62. switch ( eMode )
63. {
64. //读处理函数
65. case MB\_REG\_READ:
66. while( usNRegs > 0 )
67. {
68. \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] >> 8 );
69. \*pucRegBuffer++ = ( uint8\_t )( usRegHoldingBuf[iRegIndex] & 0xFF );
70. iRegIndex++;
71. usNRegs--;
72. }
73. break;
74. //写处理函数
75. case MB\_REG\_WRITE:
76. while( usNRegs > 0 )
77. {
78. usRegHoldingBuf[iRegIndex] = \*pucRegBuffer++ << 8;
79. usRegHoldingBuf[iRegIndex] |= \*pucRegBuffer++;
80. iRegIndex++;
81. usNRegs--;
82. }
83. break;
84. }
85. }
86. else
87. {
88. //返回错误状态
89. eStatus = MB\_ENOREG;
90. }
91. return eStatus;
92. }
93. /\*\*
94. \* @brief 线圈寄存器处理函数，线圈寄存器可读，可读可写
95. \* @param pucRegBuffer 读操作---返回数据指针，写操作--返回数据指针
96. \* usAddress 寄存器起始地址
97. \* usNRegs 寄存器长度
98. \* eMode 操作方式，读或者写
99. \* @retval eStatus 寄存器状态
100. \*/
101. eMBErrorCode
102. eMBRegCoilsCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNCoils,
103. eMBRegisterMode eMode )
104. {
105. //错误状态
106. eMBErrorCode eStatus = MB\_ENOERR;
107. //寄存器个数
108. int16\_t iNCoils = ( int16\_t )usNCoils;
109. //寄存器偏移量
110. int16\_t usBitOffset;
111. //检查寄存器是否在指定范围内
112. if( ( (int16\_t)usAddress >= REG\_COILS\_START ) &&
113. ( usAddress + usNCoils <= REG\_COILS\_START + REG\_COILS\_SIZE ) )
114. {
115. //计算寄存器偏移量
116. usBitOffset = ( int16\_t )( usAddress - REG\_COILS\_START );
117. switch ( eMode )
118. {
119. //读操作
120. case MB\_REG\_READ:
121. while( iNCoils > 0 )
122. {
123. \*pucRegBuffer++ = xMBUtilGetBits( ucRegCoilsBuf, usBitOffset,
124. ( uint8\_t )( iNCoils > 8 ? 8 : iNCoils ) );
125. iNCoils -= 8;
126. usBitOffset += 8;
127. }
128. break;
129. //写操作
130. case MB\_REG\_WRITE:
131. while( iNCoils > 0 )
132. {
133. xMBUtilSetBits( ucRegCoilsBuf, usBitOffset,
134. ( uint8\_t )( iNCoils > 8 ? 8 : iNCoils ),
135. \*pucRegBuffer++ );
136. iNCoils -= 8;
137. }
138. break;
139. }
140. }
141. else
142. {
143. eStatus = MB\_ENOREG;
144. }
145. return eStatus;
146. }
147. /\*\*
148. \* @brief 开关输入寄存器处理函数，开关输入寄存器，可读
149. \* @param pucRegBuffer 读操作---返回数据指针，写操作--返回数据指针
150. \* usAddress 寄存器起始地址
151. \* usNRegs 寄存器长度
152. \* eMode 操作方式，读或者写
153. \* @retval eStatus 寄存器状态
154. \*/
155. eMBErrorCode
156. eMBRegDiscreteCB( UCHAR \* pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
157. {
158. //错误状态
159. eMBErrorCode eStatus = MB\_ENOERR;
160. //操作寄存器个数
161. int16\_t iNDiscrete = ( int16\_t )usNDiscrete;
162. //偏移量
163. uint16\_t usBitOffset;
164. //判断寄存器时候再制定范围内
165. if( ( (int16\_t)usAddress >= REG\_DISCRETE\_START ) &&
166. ( usAddress + usNDiscrete <= REG\_DISCRETE\_START + REG\_DISCRETE\_SIZE ) )
167. {
168. //获得偏移量
169. usBitOffset = ( uint16\_t )( usAddress - REG\_DISCRETE\_START );
170. while( iNDiscrete > 0 )
171. {
172. \*pucRegBuffer++ = xMBUtilGetBits( ucRegDiscreteBuf, usBitOffset,
173. ( uint8\_t)( iNDiscrete > 8 ? 8 : iNDiscrete ) );
174. iNDiscrete -= 8;
175. usBitOffset += 8;
176. }
177. }
178. else
179. {
180. eStatus = MB\_ENOREG;
181. }
182. return eStatus;
183. }

复制代码