**FreeModbus源码详解**

**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)

/\*函数功能：

\*1:实现RTU模式和ASCALL模式的协议栈初始化;

\*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;

\*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;

\*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;

\*/

eMBErrorCode

eMBInit( eMBMode eMode, UCHAR ucSlaveAddress, UCHAR ucPort, ULONG ulBaudRate, eMBParity eParity )

{

//错误状态初始值

eMBErrorCode eStatus = MB\_ENOERR;

//验证从机地址

if( ( ucSlaveAddress == MB\_ADDRESS\_BROADCAST ) ||

( ucSlaveAddress < MB\_ADDRESS\_MIN ) || ( ucSlaveAddress > MB\_ADDRESS\_MAX ))

{

eStatus = MB\_EINVAL;

}

else

{

ucMBAddress = ucSlaveAddress; /\*从机地址的赋值\*/

switch ( eMode )

{

#if MB\_RTU\_ENABLED > 0

case MB\_RTU:

pvMBFrameStartCur = eMBRTUStart; /\*使能modbus协议栈\*/

pvMBFrameStopCur = eMBRTUStop; /\*禁用modbus协议栈\*/

peMBFrameSendCur = eMBRTUSend; /\*modbus从机响应函数\*/

peMBFrameReceiveCur = eMBRTUReceive; /\*modbus报文接收函数\*/

pvMBFrameCloseCur = MB\_PORT\_HAS\_CLOSE ? vMBPortClose : NULL;

//接收状态机

pxMBFrameCBByteReceived = xMBRTUReceiveFSM; /\*串口接收中断最终调用此函数接收数据\*/

//发送状态机

pxMBFrameCBTransmitterEmpty = xMBRTUTransmitFSM; /\*串口发送中断最终调用此函数发送数据\*/

//报文到达间隔检查

pxMBPortCBTimerExpired = xMBRTUTimerT35Expired; /\*定时器中断函数最终调用次函数完成定时器中断\*/

//初始化RTU

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

default:

eStatus = MB\_EINVAL;

}

//

if( eStatus == MB\_ENOERR )

{

if( !xMBPortEventInit() )

{

/\* port dependent event module initalization failed. \*/

eStatus = MB\_EPORTERR;

}

else

{

//设定当前状态

eMBCurrentMode = eMode; //设定RTU模式

eMBState = STATE\_DISABLED; //modbus协议栈初始化状态,在此初始化为禁止

}

}

}

return eStatus;

}

eMBEnable()函数：(mb.c)

/\*函数功能

\*1:设置Modbus协议栈工作状态eMBState为STATE\_ENABLED;

\*2:调用pvMBFrameStartCur()函数激活协议栈

\*/

eMBErrorCode

eMBEnable( void )

{

eMBErrorCode eStatus = MB\_ENOERR;

if( eMBState == STATE\_DISABLED )

{

/\* Activate the protocol stack. \*/

pvMBFrameStartCur( ); /\*pvMBFrameStartCur = eMBRTUStart;调用eMBRTUStart函数\*/

eMBState = STATE\_ENABLED;

}

else

{

eStatus = MB\_EILLSTATE;

}

return eStatus;

}

eMBRTUStart()函数：(mbrtu.c)

/\*函数功能

\*1:设置接收状态机eRcvState为STATE\_RX\_INIT；

\*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;

\*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;

\*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;

\*/

void

eMBRTUStart( void )

{

ENTER\_CRITICAL\_SECTION( );

/\* Initially the receiver is in the state STATE\_RX\_INIT. we start

\* the timer and if no character is received within t3.5 we change

\* to STATE\_RX\_IDLE. This makes sure that we delay startup of the

\* modbus protocol stack until the bus is free.

\*/

eRcvState = STATE\_RX\_INIT;

vMBPortSerialEnable( TRUE, FALSE );

vMBPortTimersEnable();

EXIT\_CRITICAL\_SECTION( );

}

eMBPoll()函数：(mb.c)

/\*函数功能:

\*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;

\*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;

\*/

eMBErrorCode

eMBPoll( void )

{

static UCHAR \*ucMBFrame; //接收和发送报文数据缓存区

static UCHAR ucRcvAddress; //modbus从机地址

static UCHAR ucFunctionCode; //功能码

static USHORT usLength; //报文长度

static eMBException eException; //错误码响应枚举

int i;

eMBErrorCode eStatus = MB\_ENOERR; //modbus协议栈错误码

eMBEventType eEvent; //事件标志枚举

/\* Check if the protocol stack is ready. \*/

if( eMBState != STATE\_ENABLED ) //检查协议栈是否使能

{

return MB\_EILLSTATE; //协议栈未使能，返回协议栈无效错误码

}

/\* Check if there is a event available. If not return control to caller.

\* Otherwise we will handle the event. \*/

//查询事件

if( xMBPortEventGet( &eEvent ) == TRUE ) //查询哪个事件发生

{

switch ( eEvent )

{

case EV\_READY:

break;

case EV\_FRAME\_RECEIVED: /\*接收到一帧数据，此事件发生\*/

eStatus = peMBFrameReceiveCur( &ucRcvAddress, &ucMBFrame, &usLength );

if( eStatus == MB\_ENOERR ) /\*报文长度和CRC校验正确\*/

{

/\* Check if the frame is for us. If not ignore the frame. \*/

/\*判断接收到的报文数据是否可接受，如果是，处理报文数据\*/

if( ( ucRcvAddress == ucMBAddress ) || ( ucRcvAddress == MB\_ADDRESS\_BROADCAST ) )

{

( void )xMBPortEventPost( EV\_EXECUTE ); //修改事件标志为EV\_EXECUTE执行事件

}

}

break;

case EV\_EXECUTE: //对接收到的报文进行处理事件

ucFunctionCode = ucMBFrame[MB\_PDU\_FUNC\_OFF]; //获取PDU中第一个字节，为功能码

eException = MB\_EX\_ILLEGAL\_FUNCTION; //赋错误码初值为无效的功能码

for( i = 0; i < MB\_FUNC\_HANDLERS\_MAX; i++ )

{

/\* No more function handlers registered. Abort. \*/

if( xFuncHandlers[i].ucFunctionCode == 0 )

{

break;

}

else if( xFuncHandlers[i].ucFunctionCode == ucFunctionCode ) /\*根据报文中的功能码，处理报文\*/

{

eException = xFuncHandlers[i].pxHandler( ucMBFrame, &usLength );/\*对接收到的报文进行解析\*/

break;

}

}

/\* If the request was not sent to the broadcast address we

\* return a reply. \*/

if( ucRcvAddress != MB\_ADDRESS\_BROADCAST )

{

if( eException != MB\_EX\_NONE ) /\*接收到的报文有错误\*/

{

/\* An exception occured. Build an error frame. \*/

usLength = 0; /\*响应发送数据的首字节为从机地址\*/

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\*/

ucMBFrame[usLength++] = eException; /\*响应发送数据帧的第三个字节为错误码标识\*/

}

if( ( eMBCurrentMode == MB\_ASCII ) && MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS )

{

vMBPortTimersDelay( MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS );

}

eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength ); /\*modbus从机响应函数,发送响应给主机\*/

}

break;

case EV\_FRAME\_SENT:

break;

}

}

return MB\_ENOERR;

}

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)

static void prvvUARTRxISR( void )

{

pxMBFrameCBByteReceived( );

}

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)

/\*函数功能

\*1:将接收到的数据存入ucRTUBuf[]中;

\*2:usRcvBufferPos为全局变量，表示接收数据的个数;

\*3:每接收到一个字节的数据，3.5T定时器清0

\*/

BOOL

xMBRTUReceiveFSM( void )

{

BOOL xTaskNeedSwitch = FALSE;

UCHAR ucByte;

assert( eSndState == STATE\_TX\_IDLE ); /\*确保没有数据在发送\*/

( void )xMBPortSerialGetByte( ( CHAR \* ) & ucByte ); /\*从串口数据寄存器读取一个字节数据\*/

//根据不同的状态转移

switch ( eRcvState )

{

/\* If we have received a character in the init state we have to

\* wait until the frame is finished.

\*/

case STATE\_RX\_INIT:

vMBPortTimersEnable(); /\*开启3.5T定时器\*/

break;

/\* In the error state we wait until all characters in the

\* damaged frame are transmitted.

\*/

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\*/

vMBPortTimersEnable();

break;

/\* In the idle state we wait for a new character. If a character

\* is received the t1.5 and t3.5 timers are started and the

\* receiver is in the state STATE\_RX\_RECEIVCE.

\*/

case STATE\_RX\_IDLE: /\*接收器空闲，开始接收，进入STATE\_RX\_RCV状态\*/

usRcvBufferPos = 0;

ucRTUBuf[usRcvBufferPos++] = ucByte; /\*保存数据\*/

eRcvState = STATE\_RX\_RCV;

/\* Enable t3.5 timers. \*/

vMBPortTimersEnable(); /\*每收到一个字节，都重启3.5T定时器\*/

break;

/\* We are currently receiving a frame. Reset the timer after

\* every character received. If more than the maximum possible

\* number of bytes in a modbus frame is received the frame is

\* ignored.

\*/

case STATE\_RX\_RCV:

if( usRcvBufferPos < MB\_SER\_PDU\_SIZE\_MAX)

{

ucRTUBuf[usRcvBufferPos++] = ucByte; /\*接收数据\*/

}

else

{

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\*/

}

vMBPortTimersEnable(); /\*每收到一个字节，都重启3.5T定时器\*/

break;

}

return xTaskNeedSwitch;

}

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)

/\*函数功能

\*1:从机接受完成一帧数据后，接收状态机eRcvState为STATE\_RX\_RCV；

\*2:上报“接收到报文”事件(EV\_FRAME\_RECEIVED)

\*3:禁止3.5T定时器，设置接收状态机eRcvState状态为STATE\_RX\_IDLE空闲;

\*/

BOOL

xMBRTUTimerT35Expired( void )

{

BOOL xNeedPoll = FALSE;

switch ( eRcvState )

{

/\* Timer t35 expired. Startup phase is finished. \*/

/\*上报modbus协议栈的事件状态给poll函数,EV\_READY:初始化完成事件\*/

case STATE\_RX\_INIT:

xNeedPoll = xMBPortEventPost( EV\_READY );

break;

/\* A frame was received and t35 expired. Notify the listener that

\* a new frame was received. \*/

case STATE\_RX\_RCV: /\*一帧数据接收完成\*/

xNeedPoll = xMBPortEventPost( EV\_FRAME\_RECEIVED ); /\*上报协议栈事件,接收到一帧完整的数据\*/

break;

/\* An error occured while receiving the frame. \*/

case STATE\_RX\_ERROR:

break;

/\* Function called in an illegal state. \*/

default:

assert( ( eRcvState == STATE\_RX\_INIT ) ||

( eRcvState == STATE\_RX\_RCV ) || ( eRcvState == STATE\_RX\_ERROR ) );

}

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\*/

eRcvState = STATE\_RX\_IDLE; /\*处理完一帧数据，接收器状态为空闲\*/

return xNeedPoll;

}

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)

/\*函数功能:

\*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;

\*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;

\*/

eMBErrorCode

eMBPoll( void )

{

static UCHAR \*ucMBFrame; //接收和发送报文数据缓存区

static UCHAR ucRcvAddress; //modbus从机地址

static UCHAR ucFunctionCode; //功能码

static USHORT usLength; //报文长度

static eMBException eException; //错误码响应枚举

int i;

eMBErrorCode eStatus = MB\_ENOERR; //modbus协议栈错误码

eMBEventType eEvent; //事件标志枚举

/\* Check if the protocol stack is ready. \*/

if( eMBState != STATE\_ENABLED ) //检查协议栈是否使能

{

return MB\_EILLSTATE; //协议栈未使能，返回协议栈无效错误码

}

/\* Check if there is a event available. If not return control to caller.

\* Otherwise we will handle the event. \*/

//查询事件

if( xMBPortEventGet( &eEvent ) == TRUE ) //查询哪个事件发生

{

switch ( eEvent )

{

case EV\_READY:

break;

case EV\_FRAME\_RECEIVED: /\*接收到一帧数据，此事件发生\*/

eStatus = peMBFrameReceiveCur( &ucRcvAddress, &ucMBFrame, &usLength );

if( eStatus == MB\_ENOERR ) /\*报文长度和CRC校验正确\*/

{

/\* Check if the frame is for us. If not ignore the frame. \*/

/\*判断接收到的报文数据是否可接受，如果是，处理报文数据\*/

if( ( ucRcvAddress == ucMBAddress ) || ( ucRcvAddress == MB\_ADDRESS\_BROADCAST ) )

{

( void )xMBPortEventPost( EV\_EXECUTE ); //修改事件标志为EV\_EXECUTE执行事件

}

}

break;

case EV\_EXECUTE: //对接收到的报文进行处理事件

ucFunctionCode = ucMBFrame[MB\_PDU\_FUNC\_OFF]; //获取PDU中第一个字节，为功能码

eException = MB\_EX\_ILLEGAL\_FUNCTION; //赋错误码初值为无效的功能码

for( i = 0; i < MB\_FUNC\_HANDLERS\_MAX; i++ )

{

/\* No more function handlers registered. Abort. \*/

if( xFuncHandlers[i].ucFunctionCode == 0 )

{

break;

}

else if( xFuncHandlers[i].ucFunctionCode == ucFunctionCode ) /\*根据报文中的功能码，处理报文\*/

{

eException = xFuncHandlers[i].pxHandler( ucMBFrame, &usLength );/\*对接收到的报文进行解析\*/

break;

}

}

/\* If the request was not sent to the broadcast address we

\* return a reply. \*/

if( ucRcvAddress != MB\_ADDRESS\_BROADCAST )

{

if( eException != MB\_EX\_NONE ) /\*接收到的报文有错误\*/

{

/\* An exception occured. Build an error frame. \*/

usLength = 0; /\*响应发送数据的首字节为从机地址\*/

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\*/

ucMBFrame[usLength++] = eException; /\*响应发送数据帧的第三个字节为错误码标识\*/

}

if( ( eMBCurrentMode == MB\_ASCII ) && MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS )

{

vMBPortTimersDelay( MB\_ASCII\_TIMEOUT\_WAIT\_BEFORE\_SEND\_MS );

}

eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength ); /\*modbus从机响应函数,发送响应给主机\*/

}

break;

case EV\_FRAME\_SENT:

break;

}

}

return MB\_ENOERR;

}

eMBRTUReceive()函数：(mbrtu.c)

/\*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

\*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

\*形参全部为地址传递\*/

eMBErrorCode

eMBRTUReceive( UCHAR \* pucRcvAddress, UCHAR \*\* pucFrame, USHORT \* pusLength )

{

BOOL xFrameReceived = FALSE;

eMBErrorCode eStatus = MB\_ENOERR;

ENTER\_CRITICAL\_SECTION();

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\*/

/\* Length and CRC check \*/

if( ( usRcvBufferPos >= MB\_SER\_PDU\_SIZE\_MIN )

&& ( usMBCRC16( ( UCHAR \* ) ucRTUBuf, usRcvBufferPos ) == 0 ) )

{

/\* Save the address field. All frames are passed to the upper layed

\* and the decision if a frame is used is done there.

\*/

\*pucRcvAddress = ucRTUBuf[MB\_SER\_PDU\_ADDR\_OFF]; //取接收到的第一个字节，modbus从机地址

/\* Total length of Modbus-PDU is Modbus-Serial-Line-PDU minus

\* size of address field and CRC checksum.

\*/

\*pusLength = ( USHORT )( usRcvBufferPos - MB\_SER\_PDU\_PDU\_OFF - MB\_SER\_PDU\_SIZE\_CRC ); //减3

/\* Return the start of the Modbus PDU to the caller. \*/

\*pucFrame = ( UCHAR \* ) & ucRTUBuf[MB\_SER\_PDU\_PDU\_OFF];

xFrameReceived = TRUE;

}

else

{

eStatus = MB\_EIO;

}

EXIT\_CRITICAL\_SECTION();

return eStatus;

}

xMBPortEventPost()函数：(portevent.c)

BOOL

xMBPortEventPost( eMBEventType eEvent )

{

xEventInQueue = TRUE;

eQueuedEvent = eEvent;

return TRUE;

}

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：

typedef struct

{

UCHAR ucFunctionCode;

pxMBFunctionHandler pxHandler;

} xMBFunctionHandler;

以下列举读线圈函数举例：

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

#if MB\_FUNC\_READ\_COILS\_ENABLED > 0

eMBException

eMBFuncReadCoils( UCHAR \* pucFrame, USHORT \* usLen )

{

USHORT usRegAddress;

USHORT usCoilCount;

UCHAR ucNBytes;

UCHAR \*pucFrameCur;

eMBException eStatus = MB\_EX\_NONE;

eMBErrorCode eRegStatus;

if( \*usLen == ( MB\_PDU\_FUNC\_READ\_SIZE + MB\_PDU\_SIZE\_MIN ) )

{

/\*线圈寄存器的起始地址\*/

usRegAddress = ( USHORT )( pucFrame[MB\_PDU\_FUNC\_READ\_ADDR\_OFF] << 8 );

usRegAddress |= ( USHORT )( pucFrame[MB\_PDU\_FUNC\_READ\_ADDR\_OFF + 1] );

//usRegAddress++;

/\*线圈寄存器个数\*/

usCoilCount = ( USHORT )( pucFrame[MB\_PDU\_FUNC\_READ\_COILCNT\_OFF] << 8 );

usCoilCount |= ( USHORT )( pucFrame[MB\_PDU\_FUNC\_READ\_COILCNT\_OFF + 1] );

/\* Check if the number of registers to read is valid. If not

\* return Modbus illegal data value exception.

\*/

/\*判断线圈寄存器个数是否合理\*/

if( ( usCoilCount >= 1 ) &&

( usCoilCount < MB\_PDU\_FUNC\_READ\_COILCNT\_MAX ) )

{

/\* Set the current PDU data pointer to the beginning. \*/

/\*为发送缓冲pucFrameCur赋值\*/

pucFrameCur = &pucFrame[MB\_PDU\_FUNC\_OFF];

\*usLen = MB\_PDU\_FUNC\_OFF;

/\* First byte contains the function code. \*/

/\*响应报文第一个字节赋值为功能码0x01\*/

\*pucFrameCur++ = MB\_FUNC\_READ\_COILS;

\*usLen += 1;

/\* Test if the quantity of coils is a multiple of 8. If not last

\* byte is only partially field with unused coils set to zero. \*/

/\*usCoilCount%8有余数，ucNBytes加1,不够的位填充0\*/

if( ( usCoilCount & 0x0007 ) != 0 )

{

ucNBytes = ( UCHAR )( usCoilCount / 8 + 1 );

}

else

{

ucNBytes = ( UCHAR )( usCoilCount / 8 );

}

\*pucFrameCur++ = ucNBytes;

\*usLen += 1;

eRegStatus =

eMBRegCoilsCB( pucFrameCur, usRegAddress, usCoilCount,

MB\_REG\_READ );

/\* If an error occured convert it into a Modbus exception. \*/

if( eRegStatus != MB\_ENOERR )

{

eStatus = prveMBError2Exception( eRegStatus );

}

else

{

/\* The response contains the function code, the starting address

\* and the quantity of registers. We reuse the old values in the

\* buffer because they are still valid. \*/

\*usLen += ucNBytes;;

}

}

else

{

eStatus = MB\_EX\_ILLEGAL\_DATA\_VALUE;

}

}

else

{

/\* Can't be a valid read coil register request because the length

\* is incorrect. \*/

eStatus = MB\_EX\_ILLEGAL\_DATA\_VALUE;

}

return eStatus;

}

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：

/\*函数功能

\*1:对响应报文PDU前面加上从机地址;

\*2:对响应报文PDU后加上CRC校;

\*3:使能发送，启动传输;

\*/

eMBErrorCode

eMBRTUSend( UCHAR ucSlaveAddress, const UCHAR \* pucFrame, USHORT usLength )

{

eMBErrorCode eStatus = MB\_ENOERR;

USHORT usCRC16;

ENTER\_CRITICAL\_SECTION( );

/\* Check if the receiver is still in idle state. If not we where to

\* slow with processing the received frame and the master sent another

\* frame on the network. We have to abort sending the frame.

\*/

if( eRcvState == STATE\_RX\_IDLE )

{

/\* First byte before the Modbus-PDU is the slave address. \*/

/\*在协议数据单元前加从机地址\*/

pucSndBufferCur = ( UCHAR \* ) pucFrame - 1;

usSndBufferCount = 1;

/\* Now copy the Modbus-PDU into the Modbus-Serial-Line-PDU. \*/

pucSndBufferCur[MB\_SER\_PDU\_ADDR\_OFF] = ucSlaveAddress;

usSndBufferCount += usLength;

/\* Calculate CRC16 checksum for Modbus-Serial-Line-PDU. \*/

usCRC16 = usMBCRC16( ( UCHAR \* ) pucSndBufferCur, usSndBufferCount );

ucRTUBuf[usSndBufferCount++] = ( UCHAR )( usCRC16 & 0xFF );

ucRTUBuf[usSndBufferCount++] = ( UCHAR )( usCRC16 >> 8 );

/\* Activate the transmitter. \*/

eSndState = STATE\_TX\_XMIT; //发送状态

xMBPortSerialPutByte( ( CHAR )\*pucSndBufferCur ); /\*发送一个字节的数据，进入发送中断函数，启动传输\*/

pucSndBufferCur++; /\* next byte in sendbuffer. \*/

usSndBufferCount--;

vMBPortSerialEnable( FALSE, TRUE ); /\*使能发送，禁止接收\*/

}

else

{

eStatus = MB\_EIO;

}

EXIT\_CRITICAL\_SECTION( );

return eStatus;

}

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)

BOOL

xMBRTUTransmitFSM( void )

{

BOOL xNeedPoll = FALSE;

assert( eRcvState == STATE\_RX\_IDLE );

switch ( eSndState )

{

/\* We should not get a transmitter event if the transmitter is in

\* idle state.\*/

case STATE\_TX\_IDLE: /\*发送器处于空闲状态，使能接收，禁止发送\*/

/\* enable receiver/disable transmitter. \*/

vMBPortSerialEnable( TRUE, FALSE );

break;

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\*/

/\* check if we are finished. \*/

if( usSndBufferCount != 0 )

{

//发送数据

xMBPortSerialPutByte( ( CHAR )\*pucSndBufferCur );

pucSndBufferCur++; /\* next byte in sendbuffer. \*/

usSndBufferCount--;

}

else

{

//传递任务，发送完成

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\*/

/\* Disable transmitter. This prevents another transmit buffer

\* empty interrupt. \*/

vMBPortSerialEnable( TRUE, FALSE ); /\*使能接收，禁止发送\*/

eSndState = STATE\_TX\_IDLE; /\*发送器状态为空闲状态\*/

}

break;

}

return xNeedPoll;

}

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；