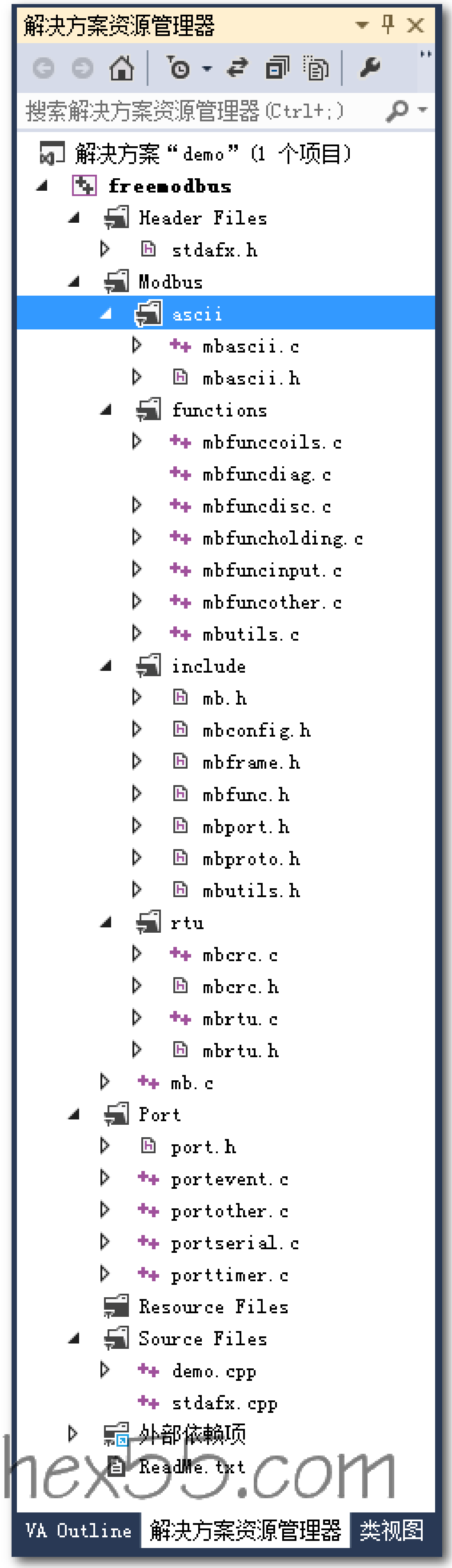
freemodbus 源码分析详解

2016 年 1 月 6 日 by shawge

我这里为了方便代码浏览，用了 VS2013，DEMO 自然用 WIN32 的，选用哪个 DEMO 进行分析也并不影响我们对 FREEMODBUS 的解剖。

代码组织结构


解决方案资源管理器
搜索解决方案资源管理器(Ctrl+;)
解决方案"demo"(1 个项目)
freemodbus
Header Files
stdafx.h
Modbus
ascii
mbascii.c
mbascii.h
functions
mbfunccoils.c
mbfuncdiag.c
mbfuncdisc.c
mbfuncholding.c
mbfuncinput.c
mbfuncother.c
mbutils.c
include
mb.h
mbconfig.h
mbframe.h
mbfunc.h
mbport.h
mbproto.h
mbutils.h
rtu
mbcrc.c
mbcrc.h
mbrtu.c
mbrtu.h
mb.c
Port
port.h
portevent.c
portother.c
portserial.c
porttimer.c
Resource Files
Source Files
demo.cpp
stdafx.cpp
外部依赖项
ReadMe.txt
VA Outline
解决方案资源管理器
类视图

首先是 modbus 这个大文件下的文件：

- **ascii** 目录的文件是用于实现 MODBUS ASCII 模式的，这个在 modbus 里是可选实现代码比较简单，看完 RTU 的分析我相信你对比着自己也就看明白 ASCII 模式了，这将不是本文的重点。
- **funcions** 是与 RTU 的执行功能码相关的代码，主要就是读、写寄存器开关线圈之类的，根据你自己的需要在去实现里面回调，按照相应参数去执行相应功能。
- **include** 是 freemodbus 的一些定义，这里先不作分析，在看源代码的时候我们再去看每个数据结构的相关定义。
- **rtu** 这个文件夹就是 RTU 模式的实现了，本文分析重点之一。
- **port 这个是移植相关**，port.h 是移植需要的函数声明。portevent.c 这个是事件队列的实现，freemodbus 只是用了一个消息作为队列简单赋值处理，portother.c 是一相从字节里取位等与 MODBUS 没多大关系的函数，portserial.c 是串口移植相关函数，porttimer.c 是定时器相关移植（由于 RTU 方式依赖时间来判断帧头帧尾），移植相关可以参见我的另一篇博文(译自官方文档)freemodbus RTU/ASCII 官方移植文档。
- **mb.c** 这个就是 modbus 的应用层实现，本文分析重点之一。

  Source Files 目录中的 demo.cpp 是例程，stdafx.cpp 是 WIN32 的预编译文件与 modbusfree 无关。

流程分析

win32 的 main（）分析，不感兴趣直接跳到 *mb.c* 一节

一切还是从 main 开始吧。

```
int
_tmain( int argc, _TCHAR * argv[] )
{
    int             iExitCode;
    TCHAR           cCh;
    BOOL            bDoExit;

    const UCHAR     ucSlaveID[] = { 0xAA, 0xBB, 0xCC };

    if( eMBInit( MB_RTU, 0x01, 1, 115200, MB_PAR_EVEN ) != MB_ENOERR )
    {
        _ftprintf( stderr, _T( "%s: can't initialize modbus stack!\r\n" ), PROG );
        iExitCode = EXIT_FAILURE;
    }
    else if( eMBSetSlaveID( 0x34, TRUE, ucSlaveID, 3 ) != MB_ENOERR )
    {
        _ftprintf( stderr, _T( "%s: can't set slave id!\r\n" ), PROG );
        iExitCode = EXIT_FAILURE;
    }
    else
    {
```

```
    /* Create synchronization primitives and set the current state
     * of the thread to STOPPED.
     */
    InitializeCriticalSection( &hPollLock );
    eSetPollingThreadState( STOPPED );

    /* CLI interface. */
    _tprintf( _T( "Type 'q' for quit or 'h' for help!\r\n" ) );
    bDoExit = FALSE;
    do
    {
        _tprintf( _T( "> " ) );
        cCh = _gettchar(  );
        switch ( cCh )
        {
        case _TCHAR( 'q' ):
            bDoExit = TRUE;
            break;
        case _TCHAR( 'd' ):
            eSetPollingThreadState( SHUTDOWN );
            break;
        case _TCHAR( 'e' ):
            if( bCreatePollingThread(  ) != TRUE )
            {
                _tprintf( _T( "Can't start protocol stack! Already running?\r\n" ) );
            }
            break;
        case _TCHAR( 's' ):
            switch ( eGetPollingThreadState(  ) )
            {
            case RUNNING:
                _tprintf( _T( "Protocol stack is running.\r\n" ) );
                break;
            case STOPPED:
                _tprintf( _T( "Protocol stack is stopped.\r\n" ) );
                break;
            case SHUTDOWN:
                _tprintf( _T( "Protocol stack is shuting down.\r\n" ) );
                break;
            }
            break;
        case _TCHAR( 'h' ):
            _tprintf( _T( "FreeModbus demo application help:\r\n" ) );
            _tprintf( _T( "  'd' ... disable protocol stack.\r\n" ) );
```

```
            _tprintf( _T( "  'e' ... enabled the protocol stack\r\n" ) );
            _tprintf( _T( "  's' ... show current status\r\n" ) );
            _tprintf( _T( "  'q' ... quit applicationr\r\n" ) );
            _tprintf( _T( "  'h' ... this information\r\n" ) );
            _tprintf( _T( "\r\n" ) );
            _tprintf( _T( "Copyright 2006 Christian Walter <wolti@sil.at>\r\n" ) );
            break;
        default:
            if( cCh != _TCHAR('\n') )
            {
                _tprintf( _T( "illegal command '%c'!\r\n" ), cCh );
            }
            break;
        }

        /* eat up everything untill return character. */
        while( cCh != '\n' )
        {
            cCh = _gettchar(  );
        }
    }
    while( !bDoExit );

    /* Release hardware resources. */
    ( void )eMBClose(  );
    iExitCode = EXIT_SUCCESS;
  }
  return iExitCode;
}
```

eMBInit(MB_RTU, 0x01, 1, 115200, MB_PAR_EVEN) != MB_ENOERR 初始化 modbus 协议栈，如果实始化失败则打印错误信息并退出，否则打印命令提示符，要求输入指令。

```
      do
       {
          _tprintf( _T( "> " ) );
          cCh = _gettchar(  );
          switch ( cCh )
          {
          case _TCHAR( 'q' ):
              bDoExit = TRUE;
              break;
          case _TCHAR( 'd' ):
```

```
            eSetPollingThreadState( SHUTDOWN );
            break;
        case _TCHAR( 'e' ):
            if( bCreatePollingThread(  ) != TRUE )
            {
                _tprintf( _T( "Can't start protocol stack! Already running?\r\n" ) );
            }
            break;
        case _TCHAR( 's' ):
            switch ( eGetPollingThreadState(  ) )
            {
            case RUNNING:
                _tprintf( _T( "Protocol stack is running.\r\n" ) );
                break;
            case STOPPED:
                _tprintf( _T( "Protocol stack is stopped.\r\n" ) );
                break;
            case SHUTDOWN:
                _tprintf( _T( "Protocol stack is shuting down.\r\n" ) );
                break;
            }
            break;
        case _TCHAR( 'h' ):
            _tprintf( _T( "FreeModbus demo application help:\r\n" ) );
            _tprintf( _T( "  'd' ... disable protocol stack.\r\n" ) );
            _tprintf( _T( "  'e' ... enabled the protocol stack\r\n" ) );
            _tprintf( _T( "  's' ... show current status\r\n" ) );
            _tprintf( _T( "  'q' ... quit applicationr\r\n" ) );
            _tprintf( _T( "  'h' ... this information\r\n" ) );
            _tprintf( _T( "\r\n" ) );
            _tprintf( _T( "Copyright 2006 Christian Walter <wolti@sil.at>\r\n" ) );
            break;
        default:
            if( cCh != _TCHAR('\n') )
            {
                _tprintf( _T( "illegal command '%c'!\r\n" ), cCh );
            }
            break;
        }

        /* eat up everything untill return character. */
        while( cCh != '\n' )
        {
            cCh = _gettchar(  );
```

```
        }
    }
    while( !bDoExit );
```

如果用户输入 e,则会调用 bCreatePollingThread( )启动协议栈线程。那么我们跟进 bCreatePollingThread( )去看看。

```
BOOL
bCreatePollingThread( void )
{
    BOOL            bResult;

    if( eGetPollingThreadState(  ) == STOPPED )
    {
        if( ( hPollThread = CreateThread( NULL, 0, dwPollingThread, NULL, 0, NULL ) ) == NULL )
        {
            /* Can't create the polling thread. */
            bResult = FALSE;
        }
        else
        {
            bResult = TRUE;
        }
    }
    else
    {
        bResult = FALSE;
    }

    return bResult;
}
```

先是确认一下线程状态，然后创建并启动线程函数 dwPollingThread（），

```
DWORD           WINAPI
dwPollingThread( LPVOID lpParameter )
{
    eSetPollingThreadState( RUNNING );

    if( eMBEnable(  ) == MB_ENOERR )
    {
        do
```

```
        {
            if( eMBPoll(  ) != MB_ENOERR )
                break;
        }
        while( eGetPollingThreadState(  ) != SHUTDOWN );
    }

    ( void )eMBDisable(  );

    eSetPollingThreadState( STOPPED );

    return 0;
}
```

从这里就跟 MCU\ARM 上应用 freemodbus 一样一样的了，无法是先使能协议栈，然后循环调用 eMBPoll( )，同时用 eGetPollingThreadState（）检测线程状态。eMBPoll(void)就是我们的重点咯，我们现在已经进入 mb.c 这个文件啦，这个是 freemodbus 实现的 modbus 应用层，虽然代码里面对数据链路层以及应用层分的不是很清晰，但这个 mb.c 是完完全全的应用层了。

mb.c

```
eMBErrorCode
eMBPoll( void )
{
    static UCHAR   *ucMBFrame;
    static UCHAR    ucRcvAddress;
    static UCHAR    ucFunctionCode;
    static USHORT   usLength;
    static eMBException eException;

    int             i;
    eMBErrorCode    eStatus = MB_ENOERR;
    eMBEventType    eEvent;

    /* Check if the protocol stack is ready. */
    if( eMBState != STATE_ENABLED )
    {
        return MB_EILLSTATE;
    }

    /* Check if there is a event available. If not return control to caller.
     * Otherwise we will handle the event. */
    if( xMBPortEventGet( &eEvent ) == TRUE )
    {
```

```
    switch ( eEvent )
    {
    case EV_READY:
        break;

    case EV_FRAME_RECEIVED:
        eStatus = peMBFrameReceiveCur( &ucRcvAddress, &ucMBFrame, &usLength );
        if( eStatus == MB_ENOERR )
        {
            /* Check if the frame is for us. If not ignore the frame. */
            if( ( ucRcvAddress == ucMBAddress ) || ( ucRcvAddress == MB_ADDRESS_BROADCAST ) )
            {
                ( void )xMBPortEventPost( EV_EXECUTE );
            }
        }
        break;

    case EV_EXECUTE:
        ucFunctionCode = ucMBFrame[MB_PDU_FUNC_OFF];
        eException = MB_EX_ILLEGAL_FUNCTION;
        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 )
            {
                eException = xFuncHandlers[i].pxHandler( ucMBFrame, &usLength );
                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 )
            {
                /* An exception occured. Build an error frame. */
                usLength = 0;
                ucMBFrame[usLength++] = ( UCHAR )( ucFunctionCode | MB_FUNC_ERROR );
                ucMBFrame[usLength++] = eException;
```

```
            }
            if( ( eMBCurrentMode == MB_ASCII ) && MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS )
            {
                vMBPortTimersDelay( MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS );
            }
            eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength );
        }
        break;

    case EV_FRAME_SENT:
        break;
    }
  }
  return MB_ENOERR;
}
```

eMBPoll（）就是一个状态机。它只有下面四种状态：

```
typedef enum
{
    EV_READY,                  /*!< Startup finished. */
    EV_FRAME_RECEIVED,          /*!< Frame received. */
    EV_EXECUTE,                /*!< Execute function. */
    EV_FRAME_SENT              /*!< Frame sent. */
} eMBEventType;
```

从注释中可以看出，分别是启动完成，帧接收完成，执行功能码，执行帧发送。

这个状态机通过 xMBPortEventGet(&eEvent) 获取事件状态，而事件状态的投递方是谁呢？这里我们先不关注（咱们自上向下分析吧）。我们先分析一下这个状态机的流程。

由于我在写这篇文章之前做过功课，所以比较清楚，这里大家过一下就可以了。

在整个协议栈运行的最初肯定是 EV_READY 态，然后过了一个 3.5T（这个就是 modbus 的帧头帧尾确认时间啦，不清楚？去翻翻协议吧，我当然不建议你去读国人写的那些“modbus 协议整理”之类的葵花宝典，而是建议你去 modbus 官网下载。找不到下载链接？看这里 Modbus Specifications and Implementation Guides,点那个 I Accept 就可以进去啦。）如果这个时候接收到一个完整的帧那么就会进入 EV_FRAME_RECEIVED 态，至于是谁负责去接收和检验帧我们后面再去理，你要记住我们还在应用层里打转转。

```
            /* Check if the frame is for us. If not ignore the frame. */
            if( ( ucRcvAddress == ucMBAddress ) || ( ucRcvAddress == MB_ADDRESS_BROADCAST ) )
            {
                ( void )xMBPortEventPost( EV_EXECUTE );
```

```
            }
```

在 EV_READY 态如果检测收到的地址跟从机地址（freemodbus 的开源版本只支持从机，如果你想要主机的可以参考一下 FreeModbus_Slave-Master-RTT-STM32）匹配，或是广播地址就自己给自己投递一个 EV_EXECUTE 事件。

```
    case EV_EXECUTE:
        ucFunctionCode = ucMBFrame[MB_PDU_FUNC_OFF];
        eException = MB_EX_ILLEGAL_FUNCTION;
        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 )
            {
                eException = xFuncHandlers[i].pxHandler( ucMBFrame, &usLength );
                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 )
            {
                /* An exception occured. Build an error frame. */
                usLength = 0;
                ucMBFrame[usLength++] = ( UCHAR )( ucFunctionCode | MB_FUNC_ERROR );
                ucMBFrame[usLength++] = eException;
            }
            if( ( eMBCurrentMode == MB_ASCII ) && MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS )
            {
                vMBPortTimersDelay( MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS );
            }
            eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength );
        }
        break;
```

在 EV_EXECUTE 的第一段就是执行相应的功能码回调，也就是读写寄存器或者是打开线圈什么的，实现上就是执行 mbfunctions 里面的代码，因为在协议栈初始化的时候这些文件里面的函数都被值给了 xFuncHandlers[]，去看看 xFuncHandlers[]的定义吧。

```
/* An array of Modbus functions handlers which associates Modbus function
 * codes with implementing functions.
 */
static xMBFunctionHandler xFuncHandlers[MB_FUNC_HANDLERS_MAX] = {
#if MB_FUNC_OTHER_REP_SLAVEID_ENABLED > 0
    {MB_FUNC_OTHER_REPORT_SLAVEID, eMBFuncReportSlaveID},
#endif
#if MB_FUNC_READ_INPUT_ENABLED > 0
    {MB_FUNC_READ_INPUT_REGISTER, eMBFuncReadInputRegister},
#endif
#if MB_FUNC_READ_HOLDING_ENABLED > 0
    {MB_FUNC_READ_HOLDING_REGISTER, eMBFuncReadHoldingRegister},
#endif
#if MB_FUNC_WRITE_MULTIPLE_HOLDING_ENABLED > 0
    {MB_FUNC_WRITE_MULTIPLE_REGISTERS, eMBFuncWriteMultipleHoldingRegister},
#endif
#if MB_FUNC_WRITE_HOLDING_ENABLED > 0
    {MB_FUNC_WRITE_REGISTER, eMBFuncWriteHoldingRegister},
#endif
#if MB_FUNC_READWRITE_HOLDING_ENABLED > 0
    {MB_FUNC_READWRITE_MULTIPLE_REGISTERS, eMBFuncReadWriteMultipleHoldingRegister},
#endif
#if MB_FUNC_READ_COILS_ENABLED > 0
    {MB_FUNC_READ_COILS, eMBFuncReadCoils},
#endif
#if MB_FUNC_WRITE_COIL_ENABLED > 0
    {MB_FUNC_WRITE_SINGLE_COIL, eMBFuncWriteCoil},
#endif
#if MB_FUNC_WRITE_MULTIPLE_COILS_ENABLED > 0
    {MB_FUNC_WRITE_MULTIPLE_COILS, eMBFuncWriteMultipleCoils},
#endif
#if MB_FUNC_READ_DISCRETE_INPUTS_ENABLED > 0
    {MB_FUNC_READ_DISCRETE_INPUTS, eMBFuncReadDiscreteInputs},
#endif
};
```

看到这里你就明白了 xFuncHandlers 不过是一个功能码和功能回调函数的对应表，eMBFuncWriteHoldingRegister（）就是写保持寄存器回调。我们还是接着看 EV_EXECUTE，第一段里面需要注意 `if( xFuncHandlers[i].ucFunctionCode == 0 )`这一句是用来在结束遍历表，freemodbus 提供了一个

eMBRegisterCB（）eMBRegisterCB 函数专门用来注册功能码和与之相应的回调，但是对于不响应的功能码 freemodbus 通过 `xFuncHandlers[i].ucFunctionCode = 0;`将其直接置 0。
EV_EXECUTE 第二段就是对主机作出回应。讲到这里接收处理就讲完了。在 mb.c 中我们可以看到这一层并不对 EV_FRAME_SENT 作处理。

mbrtu.c 分析

在 mb.c 里面我们留了一个疑惑，是谁在投递事件？或者说是谁在改变 mb.c 里面状态机的状态？

如果是 RTU 模式，那么就是这 mbrtu.c 里面的这个函数了

```
BOOL
xMBRTUTimerT35Expired( void )
{
    BOOL            xNeedPoll = FALSE;

    switch ( eRcvState )
    {
        /* Timer t35 expired. Startup phase is finished. */
    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:
        xNeedPoll = xMBPortEventPost( EV_FRAME_RECEIVED );
        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(  );
    eRcvState = STATE_RX_IDLE;

    return xNeedPoll;
}
```

这个函数是被 vMBPortTimerPoll（）被调用的，vMBPortTimerPoll（）又是被 xMBPortEventGet（）调用的，这里我们看一下 vMBPortTimerPoll（）是在什么情况下调用 xMBRTUTimerT35Expired：

```
void
    vMBPortTimerPoll(  )
{

    /* Timers are called from the serial layer because we have no high
    * res timer in Win32. */
    if( bTimeoutEnable )
    {
        DWORD          dwTimeCurrent = GetTickCount(  );

        if( ( dwTimeCurrent - dwTimeLast ) > dwTimeOut )
        {
            bTimeoutEnable = FALSE;
            ( void )pxMBPortCBTimerExpired(  );
        }
    }
}
```

可以看到就当系统的 tickCount 间隔达到一定时间时就调用 xMBRTUTimerT35Expired（）（pxMBPortCBTimerExpired 在 eMBInit()中被赋值为 xMBRTUTimerT35Expired），简单点说吧，就相当于单片机定时器中断函数，定时执行 xMBRTUTimerT35Expired()函数。

回到 mbrtu.c 中来吧，跟踪的第一要点是不能迷路，方向感要好！

xMBRTUTimerT35Expired 就是根据 eRcvState 的不同状态来投递不同的事件给 mb.c 中的 eMBPoll（）这个状态机。而 eRcvState 又是怎么来的呢？在 xMBRTUReceiveFSM（）中我们看到了它。

```
BOOL
xMBRTUReceiveFSM( void )
{
    BOOL           xTaskNeedSwitch = FALSE;
    UCHAR          ucByte;

    assert( eSndState == STATE_TX_IDLE );

    /* Always read the character. */
    ( void )xMBPortSerialGetByte( ( CHAR * ) & ucByte );
```

```
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(  );
    break;

    /* In the error state we wait until all characters in the
     * damaged frame are transmitted.
     */
case STATE_RX_ERROR:
    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:
    usRcvBufferPos = 0;
    ucRTUBuf[usRcvBufferPos++] = ucByte;
    eRcvState = STATE_RX_RCV;

    /* Enable t3.5 timers. */
    vMBPortTimersEnable(  );
    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;
    }
    else
    {
        eRcvState = STATE_RX_ERROR;
    }
    vMBPortTimersEnable(  );
```

```
        break;
    }
    return xTaskNeedSwitch;
}
```

这里不兜圈子，直接告诉你 xMBRTUReceiveFSM 会在串口接收函数中被调用（虽然在这个 WIN32 例程中并没有中断例程）。我们这里主要分析一下 xMBRTUReceiveFSM 的流程。

首先 xMBRTUReceiveFSM 会进入 STATE_RX_INIT 态，这个时候它调用 vMBPortTimersEnable 开启定时器，当达到 3.5T 时间后 xMBRTUTimerT35Expired 会让 eRcvState = STATE_RX_IDLE，这样 xMBRTUReceiveFSM 会进入 STATE_RX_IDLE 态，在 STATE_RX_IDLE 态一旦通过 xMBPortSerialGetByte 收到了一个字符，那么就会 进入 STATE_RX_RCV 态，在这里就是持续的接收字符同时进行两种检测，一种是如果接收的字符超过了 MB_SER_PDU_SIZE_MAX（RTU 帧的最大值）就会进入 STATE_RX_ERROR 态，另一种就是检测是否超时，vMBPortTimersEnable( )就是用来清零定时器的。如果超时则会由 xMBRTUTimerT35Expired 向 mb.c 状态机投递一个 EV_FRAME_RECEIVED 帧结束事件，这个时候帧数据就会被交给 mb.c 中的状态机去处理。在 xMBRTUTimerT35Expired 退出前会再次将 xMBRTUReceiveFSM 的状态置为 STATE_RX_IDLE 空闲态。

至此从上到下整个接收流程都理清楚了。那么我再看一看发送流程吧，这个比较轻松。

```
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:
        /* enable receiver/disable transmitter. */
        vMBPortSerialEnable( TRUE, FALSE );
        break;

    case STATE_TX_XMIT:
        /* check if we are finished. */
        if( usSndBufferCount != 0 )
        {
            xMBPortSerialPutByte( ( CHAR )*pucSndBufferCur );
            pucSndBufferCur++;  /* next byte in sendbuffer. */
            usSndBufferCount--;
        }
```

```
        else
        {
            xNeedPoll = xMBPortEventPost( EV_FRAME_SENT );
            /* Disable transmitter. This prevents another transmit buffer
             * empty interrupt. */
            vMBPortSerialEnable( TRUE, FALSE );
            eSndState = STATE_TX_IDLE;
        }
        break;
    }

    return xNeedPoll;
}
```

xMBRTUTransmitFSM 在 eMBInit（）中被赋值给了 pxMBFrameCBTransmitterEmpty，而 pxMBFrameCBTransmitterEmpty 又被 xMBPortSerialPoll 调用，最后 xMBPortSerialPoll 被 xMBPortEventGet 中被调用。

xMBRTUTransmitFSM 只有两个状态。

```
typedef enum
{
    STATE_TX_IDLE,              /*!< Transmitter is in idle state. */
    STATE_TX_XMIT               /*!< Transmitter is in transfer state. */
} eMBSndState;
```

在没有发送任务的时候，它是处理 STATE_TX_IDLE 态，在 modbus 协议栈初始化的时候它就是这个态，而这个 STATE_TX_XMIT 发送态则是用来将要发送的数据推送到发送缓冲的（这里你可以用你的串口中断来做，但我觉得用 DMA 会更好一些），发送完数据后又返回到 STATE_TX_IDLE 态，但是 STATE_TX_XMIT 是谁让它进入的呢？

```
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. */
        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;
        vMBPortSerialEnable( FALSE, TRUE );
    }
    else
    {
        eStatus = MB_EIO;
    }
    EXIT_CRITICAL_SECTION(  );
    return eStatus;
}
```

这个 eMBRTUSend 就是用来将 xMBRTUTransmitFSM 置为 STATE_TX_XMIT 的函数，同时它还使能串口发送功能。eMBRTUSend 本身却是在 eMBPoll（）的 EV_EXECUTE 状态的第二段被调用的，就是当收到功能码时我们回应给主机的这一部分。

```
            /* If the request was not sent to the broadcast address we
             * return a reply. */
            if( ucRcvAddress != MB_ADDRESS_BROADCAST )
            {
                if( eException != MB_EX_NONE )
                {
                    /* An exception occured. Build an error frame. */
                    usLength = 0;
                    ucMBFrame[usLength++] = ( UCHAR )( ucFunctionCode | MB_FUNC_ERROR );
                    ucMBFrame[usLength++] = eException;
```

```
            }
            if( ( eMBCurrentMode == MB_ASCII ) && MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS )
            {
                vMBPortTimersDelay( MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS );
            }
            eStatus = peMBFrameSendCur( ucMBAddress, ucMBFrame, usLength );
        }
        break;
```

其中的 peMBFrameSendCur（）就是 eMBRTUSend（），在 eMBInit 我们将 eMBRTUSend 赋值给了 peMBFrameSendCur（）。 现在咱们终于绕出来了，发送流程也介绍清楚了。

写到这里，我估计你可能会有一些疑惑，在这个例程中真正完成发送和接收串口的代码在哪里？

```
BOOL
xMBPortEventGet( eMBEventType * eEvent )
{
    BOOL            xEventHappened = FALSE;

    if( xEventInQueue )
    {
        *eEvent = eQueuedEvent;
        xEventInQueue = FALSE;
        xEventHappened = TRUE;
    }
    else
    {
        /* Poll the serial device. The serial device timeouts if no
         * characters have been received within for t3.5 during an
         * active transmission or if nothing happens within a specified
         * amount of time. Both timeouts are configured from the timer
         * init functions.
         */
        ( void )xMBPortSerialPoll(  );

        /* Check if any of the timers have expired. */
        vMBPortTimerPoll(  );

    }
    return xEventHappened;
}
```

其实它们就在 xMBPortSerialPoll 里，换句话说，每次当 mb.c 的状态机调用 xMBPortEventGet（）都在进行串口操作，要么是发送要么是接收。

```
BOOL
xMBPortSerialPoll(  )
{
    BOOL            bStatus = TRUE;
    DWORD           dwBytesRead;
    DWORD           dwBytesWritten;
    DWORD           i;

    while( bRxEnabled )
    {
        /* buffer wrap around. */
        if( uiRxBufferPos >= BUF_SIZE )
            uiRxBufferPos = 0;

        if( ReadFile( g_hSerial, &ucBuffer[uiRxBufferPos],
                      BUF_SIZE - uiRxBufferPos, &dwBytesRead, NULL ) )
        {
            if( dwBytesRead == 0 )
            {
                /* timeout with no bytes. */
                break;
            }
            else if( dwBytesRead > 0 )
            {
                vMBPortLog( MB_LOG_DEBUG, _T( "SER-POLL" ),
                            _T( "detected end of frame (t3.5 expired.)\r\n" ) );
                for( i = 0; i < dwBytesRead; i++ )
                {
                    /* Call the modbus stack and let him fill the buffers. */
                    ( void )pxMBFrameCBByteReceived(  );
                }
            }
        }
        else
        {
            vMBPortLog( MB_LOG_ERROR, _T( "SER-POLL" ), _T( "I/O error on serial device: %s" ),
                        Error2String( GetLastError ( ) ) );
            bStatus = FALSE;
        }
    }
```

```
    if( bTxEnabled )
    {
        while( bTxEnabled )
        {
            ( void )pxMBFrameCBTransmitterEmpty(  );
            /* Call the modbus stack to let him fill the buffer. */
        }
        dwBytesWritten = 0;
        if( !WriteFile
            ( g_hSerial, &ucBuffer[0], uiTxBufferPos, &dwBytesWritten, NULL )
            || ( dwBytesWritten != uiTxBufferPos ) )
        {
            vMBPortLog( MB_LOG_ERROR, _T( "SER-POLL" ), _T( "I/O error on serial device: %s" ),
                        Error2String( GetLastError ( ) ) );
            bStatus = FALSE;
        }
    }

    return bStatus;
}
```

xMBPortSerialPoll 依据 bRxEnabled 和 bTxEnabled 来区分到底是发送还是接收。

我看到有些人说 freemodbus 只能通过阻塞方式发送和接收串口数据很显然是错误的，它可以用普通串口中断或者是串口 DMA 来做。

写第一版的时候忘了分析一下事件队列，虽然说是叫事件队列，其实就是很简单的对一个变量进行了封装,提供了抽象接口，代码也只有这么几行：

```
/* ----------------------- Variables ----------------------------------------*/
static eMBEventType eQueuedEvent;
static BOOL     xEventInQueue;

/* ----------------------- Start implementation -----------------------------*/
BOOL
xMBPortEventInit( void )
{
    xEventInQueue = FALSE;
    return TRUE;
}

BOOL
xMBPortEventPost( eMBEventType eEvent )
{
```

```
    xEventInQueue = TRUE;
    eQueuedEvent = eEvent;
    return TRUE;
}

BOOL
xMBPortEventGet( eMBEventType * eEvent )
{
    BOOL            xEventHappened = FALSE;

    if( xEventInQueue )
    {
        *eEvent = eQueuedEvent;
        xEventInQueue = FALSE;
        xEventHappened = TRUE;
    }
    else
    {
        /* Poll the serial device. The serial device timeouts if no
         * characters have been received within for t3.5 during an
         * active transmission or if nothing happens within a specified
         * amount of time. Both timeouts are configured from the timer
         * init functions.
         */
        ( void )xMBPortSerialPoll(  );

        /* Check if any of the timers have expired. */
        vMBPortTimerPoll(  );

    }
    return xEventHappened;
}
```

xMBPortEventPost 这个投递事件的函数只是将事件枚举赋值给这个模块的变量，同时将 xEventInQueue 置为真表示队列中有数据，xMBPortEventGet 的逻辑稍微复杂一点，它会在 eMBPoll 状态机中被反复调用，它首先将 xEventInQueue 置为 FALSE，然后如果队列中有数据将就队列中的数字赋给 eMBPoll 传入的指针，没有事件的话就进行一下串口的接收和发送处理。

注意：freemodbus 并没有用到 T1.5（同一帧内两个字符之间的最大时间间隔）检测，你可以去看源代码里面 xMBRTUTimerT15Expired 这个函数仅仅只是声明了，我个人猜测是因为 T1.5 这个时间粒度太小（波特率为 19200，按协议 t1.5 取为 750us），一般的 MCU 根本没精力去做这个检测。

最后提一下 asc 模式，在 eMBInit()函数中我们看到如果你的初时化时候的选择 MB_ASCII 作为参数，与 modbus 协议相关的回调和状态都会被替换成 maasiic 中的内容，顺着这条藤去摸一下 ASC 模式的瓜应该不难。

```
#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
```