FreeModbus 学习笔记

一、 FreeModbus 简介

FreeMODBUS 一个奥地利人写的 Modbus 协议。它是一个针对嵌入式应用的一个免费（自由）的通用 MODBUS 协议的移植。Modbus 是一个工业制造环境中应用的一个通用协议。Modbus 通信协议栈包括两层：Modbus 应用层协议，该层定义了数据模式和功能；另外一层是网络层。

FreeMODBUS 提供了 RTU/ASCII 传输模式及 TCP 协议支持。FreeModbus 遵循 BSD 许可证，这意味着用户可以将 FreeModbus 应用于商业环境中。目前版本 FreeModbus-V1.5 提供如下的功能支持：

表 1 FreeModbus-V1.5 功能支持

代码	描述	是否支持	备注
Master	主机	否	
Slave	从机	是	
MB_RTU	RTU 模式	是	
MB_ASCII	ASCII 模式	是	
MB_TCP	TCP 模式	是	
0x01	读线圈	是	
0x02	读离散输入	是	
0x03	读保持寄存器	是	
0x04	读输入寄存器	是	
0x05	写单个线圈	是	
0x06	写单个寄存器	是	
0x07	读异常状态	否	
0x08	诊断	否	
0x0B	获取事件计数器	否	
0x0C	获取事件记录	否	
0x0F	写多个线圈	是	
0x10	写多个寄存器	是	
0x11	报告从机 ID	是	协议与文档不一致
0x14	读文件记录	否	
0x15	写文件记录	否	
0x16	屏蔽写寄存器	否	
0x17	读/写多个寄存器	是	
0x18	写 FIFO	否	
0x2B	封装接口传输	否	
0x2B/0x0D	CANopen 参考请求与应答	否	
0x2B/0x0E	读设备身份表示	否	

二、 FreeModbus 对硬件的需求

FreeModbus 协议对硬件的需求非常少——基本上任何具有串行接口，并且有一些能够容纳 modbus 数据帧的 RAM 的微控制器都足够了。

- 一个异步串行接口，能够支持接收缓冲区满和发送缓存区空中断。
- 一个能够产生 RTU 传输所需要的 t3.5 字符超时定时器的时钟。

对于软件部分，仅仅需要一个简单的事件队列。在使用操作系统的处理器上，可通过单独定义一个任务完成 Modbus 时间的查询。小点的微控制器往往不允许使用操作系统，在那种情况下，可以使用一个全局变量来实现该事件队列(Atmel AVR 移植使用这种方式实现)。

实际的存储器需求决定于所使用的 Modbus 模块的多少。下表列出了所支持的功能编译后所需要的存储器。ARM 是使用 GNUARM 编译器 3.4.4 使用-O1 选项得到的。AVR 项数值是使用 WinAVR 编译器 3.4.5 使用-Os 选项编译得到的。

表 2 FreeModbus 对硬件的需求

Module	ARM Code	ARM RAM (static)	AVR Code	AVR RAM (static)
Modbus RTU (Required)	1132Byte	272Byte	1456Byte	266Byte
Modbus ASCII (Optional)	1612Byte	28Byte	1222Byte	16Byte
Modbus Functions [1]	1180Byte	34Byte	1602Byte	34Byte
Modbus Core (Required)	924Byte	180Byte	608Byte	75Byte
Porting Layer (Required [2])	1756Byte	16Byte	704Byte	7Byte
Totals	7304Byte	530Byte	5592Byte	398Byte

[1] 实际大小决定于可支持的 Modbus 功能码的多少。功能码可以在头文件 mbconfig.h 中进行配置。

[2] 决定于硬件。

三、 FreeModbus 的移植

1、物理层接口文件的修改

在物理层，用户只需完成串行口及超时定时器的配置即可。具体应修改接口文件 **portserial.c** 及 **porttimer.c**。

- **portserial.c** 中函数的修改：

1) void vMBPortSerialEnable(BOOL xRxEnable, BOOL xTxEnable)

此函数的功能为设置串口状态。有两个参数：xRxEnable 及 xTxEnable。当 xRxEnable 为真时，应使能串口接收及接收中断。在 RS485 通讯系统中，还要注意将 RS485 接口芯片设为接收使能状态；当 xTxEnable 为真时，应使能串口发送及发送中断。在 RS485 通讯系统中，还要注意将 RS485 接口芯片设为发送使能状态。

2) void vMBPortClose(void)

此函数的功能是关闭 Modbus 通讯端口，具体的，应在此函数中关闭通讯端口的发送使能及接收使能。

3) BOOL xMBPortSerialInit(UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity eParity)

此函数的功能是初始化串行通讯端口。有四个参数：ucPORT、ulBaudRate、ucDataBits 及 eParity。参数 ucPORT 可以忽略；参数 ulBaudRate 是通讯端口的波特率，应根据此数值设置所使用硬件端口的波特率；参数 ucDataBits 为通讯时所使用的数据位宽，注意，若使用 RTU 模式，则有 ucDataBits=8，若使用 ASCII 模式，则有 ucDataBits=7，应根据此参数设置所使用硬件端口的数据位宽；eParity 为校验方式，eParity=MB_PAR_NONE 为无校验，此时

硬件端口应设置为无校验方式及两个停止位，eParity=MB_PAR_ODD 为奇校验，此时硬件端口应设置为奇校验方式及一个停止位，eParity= MB_PAR_EVEN 为偶校验，此时硬件端口应设置为偶校验方式及一个停止位。函数返回值务必为 TRUE。

4) BOOL xMBPortSerialPutByte(CHAR ucByte)

此函数的功能为通讯端口发送一字节数据。参数为：ucByte，待发送的数据。应在此函数中编写发送一字节数据的函数。注意，由于使用的是中断发送，故只需将数据放到发送寄存器即可。函数返回值务必为 TRUE。

5) BOOL xMBPortSerialGetByte(CHAR * pucByte)

此函数的功能为通讯端口接收一字节数据。参数为：* pucByte，接收到的数据。应在此函数中编写接收的函数。注意，由于使用的是中断接收，故只需将接收寄存器的值放到* pucByte 即可。函数返回值务必为 TRUE。

6) void prvvUARTTxReadyISR(void)

发送中断函数。此函数无需修改。只需在用户的发送中断函数中调用此函数即可，同时，用户应在调用此函数后，清除发送中断标志位。

7) void prvvUARTRxISR(void)

发送中断函数。此函数无需修改。只需在用户的接收中断函数中调用此函数即可，同时，用户应在调用此函数后，清除接收中断标志位。

◆ **portserial.c** 中函数的修改：

1) BOOL xMBPortTimersInit(USHORT usTim1Timerout50us)

此函数的功能为初始化超时定时器。参数为：usTim1Timerout50us，50us 的个数。用户应根据所使用的硬件初始化超时定时器，使之能产生中断时间为 usTim1Timerout50us*50us 的中断。函数返回值务必为 TRUE。

2) void vMBPortTimersEnable( )

此函数的功能为使能超时定时器。用户需在此函数中清除中断标志位、清零定时器计数值，并重新使能定时器中断。

3) void vMBPortTimersDisable( )

此函数的功能为关闭超时定时器。用户需在此函数中清零定时器计数值，并关闭定时器中断。

4) void TIMERExpiredISR(void)

定时器中断函数。此函数无需修改。只需在用户的定时器中断中调用此函数即可，同时，用户应在调用此函数后清除中断标志位。

2、应用层回函数的修改

在应用层，用户需要定义所需要使用的寄存器，并修改对应的回函数。回函数有如下几个：

1) eMBErrorCode eMBRegInputCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs)

输入寄存器回函数。* pucRegBuffer 为要添加到协议中的数据，usAddress 为输入寄存器地址，usNRegs 为要读取寄存器的个数。用户应根据要访问的寄存器地址 usAddress 将相应输入寄存器的值按顺序添加到 pucRegBuffer 中。

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

保持寄存器回函数。* pucRegBuffer 为要协议中的数据，usAddress 为输入寄存器地址，usNRegs 为访问寄存器的个数，eMode 为访问类型（MB_REG_READ 为读保持寄存器，MB_REG_WRITE 为写保持寄存器）。用户应根据要访问的寄存器地址 usAddress 将相应输入寄存器的值按顺序添加到 pucRegBuffer 中，或将协议中的数据根据要访问的寄存器地址 usAddress 放到相应保持寄存器中。

3) eMBErrorCode eMBRegCoilsCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT

usNCoils, eMBRegisterMode eMode)

读写线圈回函数。* pucRegBuffer 为要添加到协议中的数据，usAddress 为线圈地址，usNCoils 为要访问线圈的个数，eMode 为访问类型（MB_REG_READ 为读线圈状态，MB_REG_WRITE 为写线圈）。用户应根据要访问的线圈地址 usAddress 将相应线圈的值按顺序添加到 pucRegBuffer 中，或将协议中的数据根据要访问的线圈地址 usAddress 放到相应线圈中。

4) eMBErrorCode eMBRegDiscreteCB(UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete)

读离散线圈回函数。* pucRegBuffer 为要添加到协议中的数据，usAddress 为线圈地址，usNDiscrete 为要访问线圈的个数。用户应根据要访问的线圈地址 usAddress 将相应线圈的值按顺序添加到 pucRegBuffer 中。

3、应用层初始化及协议访问

用户只需在主函数中调用协议初始化代码，及消息处理函数即可。需用户调用的函数有如下几个：

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

协议初始化函数。eMode 为所要使用的模式，用户可选 MB_RTU（RTU 模式）、MB_ASCII（ASCII 模式）或 MB_TCP（TCP 模式）；ucSlaveAddress 为从机地址，用户根据需要，取值为 1~247（0 为广播地址，248~255 协议保留）；ulBaudRate 为通信波特率，用户根据需要选用，但务必使主机能支持此波特率；eParity 为校验方式，用户根据需要选用，但务必使主机能支持此校验方式。

2) eMBErrorCode eMBSetSlaveID(UCHAR ucSlaveID, BOOL xIsRunning, UCHAR const *pucAdditional, USHORT usAdditionalLen)

从机 ID 设置函数。注意，ID 表示的是设备的类型，不同于 ucSlaveAddress（从机地址）。对同一通讯系统中，可以有相同的 ucSlaveID，但不可以有相同的 ucSlaveAddress。ucSlaveID 为一字节的设备 ID 号；xIsRunning 为设备的运行状态，0xFF 为运行，0x00 为停止；* pucAdditional 为设备的附加描述，根据需要添加；usAdditionalLen 为附加描述的长度（按字节计算）。此函数不是必须调用的。但当一个 Modbus 通讯系统中有不同种设备时，应调用此函数添加对应设备的描述。

3) eMBErrorCode eMBPoll(void)

轮询事件查询处理函数。用户需在主循环中调用此函数。对于使用操作系统的程序，应单独创建一个任务，使操作系统能周期调用此函数。

四、 FreeModbus 初始化及运行流程

FreeModbus 是基于消息队列的协议。协议通过检测相应的消息来完成对应功能。协议栈的初始化及运行流程如下：

1) 首先调用 eMBErrorCode eMBInit(eMBMode eMode, UCHAR ucSlaveAddress, UCHAR ucPort, ULONG ulBaudRate, eMBParity eParity)完成物理层设备的初始化，主要包括：BOOL xMBPortSerialInit(UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity eParity)串口初始化，设定波特率、数据位数、校验方式；BOOL xMBPortTimersInit(USHORT usTim1Timerout50us)定时器初始化，设定 T35 定时所需要的定时器常数。

2) 调用（此处非必需）eMBErrorCode eMBSetSlaveID(UCHAR ucSlaveID, BOOL xIsRunning,UCHAR const *pucAdditional, USHORT usAdditionalLen)指定设备 ID。

3) 调用 eMBErrorCode eMBEnable(void)使能协议栈，主要包括：static pvMBFrameStart pvMBFrameStartCur（函数指针）协议栈开始，将 eRcvState 设为 STATE_RX_INIT 状态，调用 void vMBPortSerialEnable(BOOL xRxEnable, BOOL xTxEnable)使能接收，调用 void vMBPortTimersEnable( )使能超时定时器。

4) 在 3 中使能了超时定时器，故经过 T35 时间后，发生第一次超时中断，在中断中，向协议栈发送消息 EV_READY（Startup finished），并调用 void vMBPortTimersDisable(　)关闭超时定时器，同时将 eRcvState 设为 STATE_RX_IDLE。此时，协议栈可以接收串口数据。注意，此处首先启用一次超时定时器是因为初始化完成时，串口有可能已经有数据，因为无法判断第一个数据是请求的开始，故等待 T35，接收下一帧请求。

5) 此时，主函数调用 eMBErrorCode eMBPoll(void)检测事件。

6) 若发生串口接收中断，且 eRcvState 为 STATE_RX_IDLE（4 中已将 eRcvState 设为 STATE_RX_IDLE），则向接收缓存中存入接收到的字符，同时将 eRcvState 设为 STATE_RX_RCV 状态，并清零超时定时器。在下一个数据来到时，不断将数据存入接收缓存，并清零超时定时器。

7) 如果没有接收完成，则不可能发生超时中断。发生超时中断，说明 T35 时间内未收到新的串口数据，根据 Modbus 协议的规定，这指示着一帧请求数据接收完成。在中断中，向协议栈发送消息 EV_FRAME_RECEIVED（Frame received），等待协议栈处理此消息。

8) 主函数调用 eMBErrorCode eMBPoll(void)检测到事件 EV_FRAME_RECEIVED 后，调用 static peMBFrameReceive peMBFrameReceiveCur 简单判断请求帧数据，并向协议栈发送消息 EV_EXECUTE（Execute function）。

9) 主函数调用 eMBErrorCode eMBPoll(void)检测到事件 EV_EXECUTE 后，根据相应的请求代码查找处理该功能的函数指针来处理该功能。若不是广播消息，则调用 static peMBFrameSend peMBFrameSendCur 发送回复消息，在此函数中，只把要回复的数据复制到了串口缓存中，同时将 eSndState 设为 STATE_TX_XMIT（Transmitter is in transfer state），并通过调用 void vMBPortSerialEnable(BOOL xRxEnable, BOOL xTxEnable)使能发送中断。注意，发送中断使能后，由于串口发送寄存器本来就是空的，故在使能后将进入发送中断中。

10) 发送中断中，且 eSndState 为 STATE_TX_XMIT（9 中已将 eSndState 设为 STATE_TX_XMIT），则将串口缓存中的数据发送出去，同时不断对发送字符个数统计，当发送完成后，向协议栈发送消息 EV_FRAME_SENT（Frame sent）。

11) 主函数调用 eMBErrorCode eMBPoll(void)检测到事件 EV_FRAME_SENT 后，不处理此消息。

12) 当串口接收到数据后，协议栈将重复 6-11 处理消息。

五、 一些理解

1. 关于 **mbrtu.c** 文件的理解

◆ 宏定义与变量

mbrtu.c 文件中定义了 RTU 模式下的宏定义、全局变量与功能函数。所包含的宏定义与全局变量定义如下：

```
/* ---------------------- Defines ------------------------------------------*/
#define MB_SER_PDU_SIZE_MIN     4       /*!< Minimum size of a Modbus RTU frame. */
#define MB_SER_PDU_SIZE_MAX     256     /*!< Maximum size of a Modbus RTU frame. */
#define MB_SER_PDU_SIZE_CRC     2       /*!< Size of CRC field in PDU. */
#define MB_SER_PDU_ADDR_OFF     0       /*!< Offset of slave address in Ser-PDU. */
#define MB_SER_PDU_PDU_OFF      1       /*!< Offset of Modbus-PDU in Ser-PDU. */

/* ---------------------- Type definitions ---------------------------------*/
typedef enum
{
    STATE_RX_INIT,              /*!< Receiver is in initial state. */
```

```
    STATE_RX_IDLE,              /*!< Receiver is in idle state. */
    STATE_RX_RCV,               /*!< Frame is beeing received. */
    STATE_RX_ERROR              /*!< If the frame is invalid. */
} eMBRcvState;

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

/* ----------------------- Static variables ---------------------------------*/
static volatile eMBSndState eSndState;
static volatile eMBRcvState eRcvState;

volatile UCHAR    ucRTUBuf[MB_SER_PDU_SIZE_MAX];

static volatile UCHAR *pucSndBufferCur;
static volatile USHORT usSndBufferCount;

static volatile USHORT usRcvBufferPos;
```

首先在宏定义中，指明了该模式下所支持的最小请求帧长度为 4（1 字节地址+1 字节命令+2 字节校验），最大请求帧长度为 256，CRC 为两字节，地址为第一字节，PDU 开始于第二字节。

在全局变量中，只定义了一个串口缓存数组 ucRTUBuf[MB_SER_PDU_SIZE_MAX]。由于发送与接收不是同步的，故可采用该缓存数组实现 Modbus 协议。在接收过程中，将所接收到的数据直接存放于缓存 ucRTUBuf 中，在发送过程中，通过指针*pucSndBufferCur 来访问该数组。

- **eMBErrorCode eMBRTUInit(UCHAR ucSlaveAddress, UCHAR ucPort, ULONG ulBaudRate, eMBParity eParity)**

此函数为 RTU 模式的初始化函数。此函数中判断串行口初始化是否成功（通过判断串行口初始化函数的返回值实现。当然，查看返回值必然先调用该函数，从而完成端口初始化），如果成功，则根据波特率计算 T35，初始化超时定时器。

- **void eMBRTUStart(void)**

此函数为 RTU 模式开始函数。函数主要功能是，将接收状态 eRcvState 设为 STATE_RX_INIT（Receiver is in initial state），使能接收同时关闭发送，使能超时定时器。

- **void eMBRTUStop(void)**

此函数为 RTU 模式终止函数。函数主要功能是，关闭接收与发送，关闭超时定时器。

- **eMBErrorCode eMBRTUReceive(UCHAR * pucRcvAddress, UCHAR ** pucFrame, USHORT * pusLength)**

此函数为 RTU 接收数据帧信息提取函数。函数主要功能是，将接收帧（存放于缓存）的地址指针赋给指针变量 pucRcvAddress，将 PDU 编码首地址赋给指针* pucFrame，将 PDU 长度地址赋给指针变量 pusLength。使用指针访问缓存数组，而不是额外开辟缓存存放帧信息，大大减少了内存的开支。

- **eMBErrorCode eMBRTUSend(UCHAR ucSlaveAddress, const UCHAR * pucFrame, USHORT usLength)**

此函数为 RTU 回复帧信息组织函数。函数的功能是，此函数首先使发送内容指针

pucSndBufferCur 指向 pucFrame 之前的一个地址，并将该地址内容填充为 ucSlaveAddress，并使用直接访问方式向缓存数组 ucRTUBuf 的相应地址内存入 CRC 校验值。注意，此函数中，对 ucRTUBuf 的访问既有间接方式（指针 pucSndBufferCur 与 pucFrame），又有直接方式（直接向相应地址内写值），比较难理解。

回复帧组织完后，将发送状态 eSndState 设为 STATE_TX_XMIT（Transmitter is in transfer state），并禁止接收使能发送。发送一旦使能，就会进入发送中断，完成相应字符的发送。

◆ **BOOL xMBRTUReceiveFSM(void)**

此函数描述了一个接收状态机，供接收中断调用。状态机中，首先完成串口接收寄存器读取，然后判断相应接收状态 eRcvState，实现接收。在 STATE_RX_INIT 状态，重置超时定时器，等待超时中断（超时中断会把 eRcvState 设为 STATE_RX_IDLE）；在 STATE_RX_ERROR 状态，同样会重置超时定时器等待超时中断；在 STATE_RX_IDLE 状态，会将接收字符个数置零，同时向缓存数组 ucRTUBuf 中存入接收到的字符，跳入状态 STATE_RX_RCV，并使重置超时定时器；在 STATE_RX_RCV 状态，不断将接收到的字符存入缓存，并统计接收计数，重置超时定时器，接收计数大于帧最大长度时，会跳入 STATE_RX_ERROR 状态。

在任何一处发生超时中断，都会将状态 eRcvState 置为 STATE_RX_IDLE。在接收过程（STATE_RX_RCV）中，发生超时中断，指示着一帧数据接收完成。

接收状态机如图 1 所示：

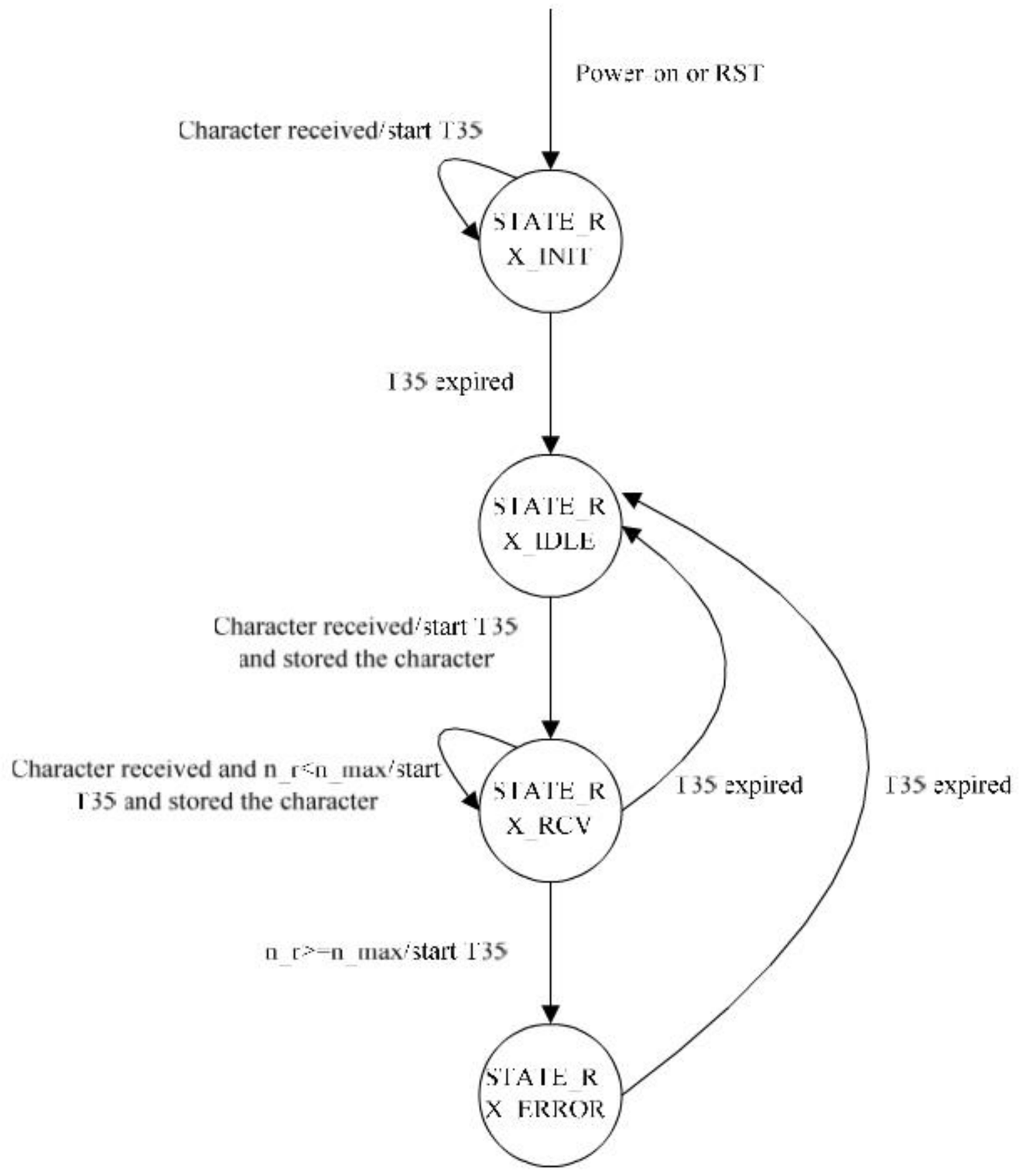

图 1 接收状态机图

◆ **BOOL xMBRTUTransmitFSM(void)**

此函数描述了一个接收状态机，供发送中断调用。状态机中，判断相应发送状态 eSndState，实现发送。在 STATE_TX_IDLE 状态，使能接收关闭发送；在 STATE_TX_XMIT 状态，调用底层串口发送函数将缓存中的字符发送出去，并使发送指针加 1，待发送字符数

减 1，待发送数为 0 时，将向系统发送事件 EV_FRAME_SENT（Frame sent），同时使能接收关闭发送，并转向 STATE_TX_IDLE 状态。

发送状态机如图 2 所示：

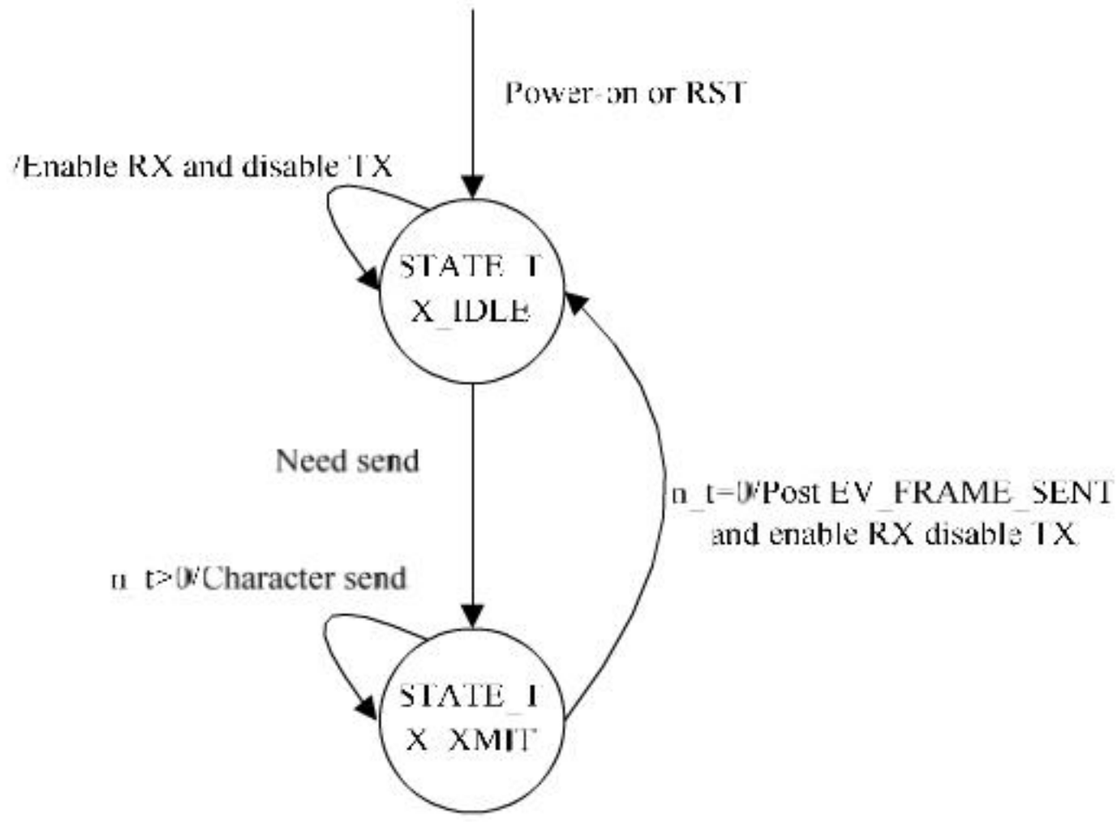

图 2 发送状态机图

◆ **BOOL xMBRTUTimerT35Expired(void)**

此函数描述了发生超时中断时应处理的事务，供超时中断调用。通过判读接收状态 eRcvState 来决定要处理的事务，思想上有点像摩尔类型的 FSM 的输出逻辑。若中断发生于 STATE_RX_INIT，则向系统发送事件 EV_READY（Startup finished）；若中断发生于 STATE_RX_RCV，则向系统发送事件 EV_FRAME_RECEIVED（Frame received）；若中断发生于 STATE_RX_ERROR，则跳出，不执行。在每个执行分支结束后，均关闭超时定时器，并将 eRcvState 转为 STATE_RX_IDLE。当然，这儿不像 FSM 的输出逻辑。

2. 关于 **mb.c** 文件的理解

◆ 宏定义与变量

mb.c 文件中定义了一系列的宏定义、函数指针及全局变量，并使用优先编译指令预编译一些程序代码。定义与优先编译部分如下：

```
#if MB_RTU_ENABLED == 1
#include "mbrtu.h"
#endif
#if MB_ASCII_ENABLED == 1
#include "mbascii.h"
#endif
#if MB_TCP_ENABLED == 1
#include "mbtcp.h"
#endif

#ifndef MB_PORT_HAS_CLOSE
#define MB_PORT_HAS_CLOSE 0
#endif

/* ----------------------- Static variables ---------------------------------*/

static UCHAR        ucMBAddress;
static eMBMode      eMBCurrentMode;
```

```
static enum
{
    STATE_ENABLED,
    STATE_DISABLED,
    STATE_NOT_INITIALIZED
} eMBState = STATE_NOT_INITIALIZED;

/* Functions pointer which are initialized in eMBInit( ). Depending on the
 * mode (RTU or ASCII) the are set to the correct implementations.
 */
static peMBFrameSend peMBFrameSendCur;
static pvMBFrameStart pvMBFrameStartCur;
static pvMBFrameStop pvMBFrameStopCur;
static peMBFrameReceive peMBFrameReceiveCur;
static pvMBFrameClose pvMBFrameCloseCur;

/* Callback functions required by the porting layer. They are called when
 * an external event has happend which includes a timeout or the reception
 * or transmission of a character.
 */
BOOL( *pxMBFrameCBByteReceived ) ( void );
BOOL( *pxMBFrameCBTransmitterEmpty ) ( void );
BOOL( *pxMBPortCBTimerExpired ) ( void );

BOOL( *pxMBFrameCBReceiveFSMCur ) ( void );
BOOL( *pxMBFrameCBTransmitFSMCur ) ( void );

/* 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
};
```

头文件使用优先编译指令，根据 Modbus 的配置文件中的相应宏开关，预编译所需的头文件，村儿减小协议代码量。

全局变量 ucMBAddress 与 eMBCurrentMode 分别表示从机地址与当前所选用的 Modbus 模式。

接下来定义了一系列的函数指针。在初始化函数中，会根据当前所选用的 Modbus 模式使这些函数指针指向相应模式下的功能函数。

关于功能代码与功能函数，写的特别巧妙：首先定义 xMBFunctionHandler 类型的结构体数组 xFuncHandlers，对于数组中的每一个元素，都可看成一个结构体。
xMBFunctionHandler 结构体类型在文件 mbproto.h 中定义如下：

```
typedef struct
{
    UCHAR               ucFunctionCode;
    pxMBFunctionHandler pxHandler;
} xMBFunctionHandler;
```

pxMBFunctionHandler 描述的是一种函数指针类型，在 mbproto.h 中定义如下：

```
typedef    eMBException(*pxMBFunctionHandler) (UCHAR *pucFrame,USHORT *pusLength);
```

故 xFuncHandlers 中的每一个元素都具有两个成员：ucFunctionCode（功能码）与 pxHandler（功能函数指针）。通过相应的宏开关，可选择预编译相应的功能函数（宏开关在文件 mbconfig.h 中定义）。

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

此函数为 Modbus 协议初始化函数。函数首先判断从机地址 ucSlaveAddress，若为广播地址，或协议保留地址，或配置文件中未规定的地址，均会使该函数返回一个错误 MB_EINVAL（illegal argument）。若地址合法，则会将该地址赋给全局变量 ucMBAddress，同时根据所选用的模式 eMode（MB_RTU、MB_ASCII 或 MB_TCP）初始化相应的函数指针。

以 RTU 模式为例，pvMBFrameStartCur 将指向协议开始函数 void eMBRTUStart(void)；pvMBFrameStopCur 将指向协议终止函数 eMBErrorCode eMBRTUSend(UCHAR

ucSlaveAddress, const UCHAR * pucFrame, USHORT usLength)；peMBFrameReceiveCur 将指向接收帧信息提取函数 eMBErrorCode eMBRTUReceive(UCHAR * pucRcvAddress, UCHAR ** pucFrame, USHORT * pusLength)；pvMBFrameCloseCur 将指向端口关闭函数 void vMBPortClose(void)（portserial.c 中定义)；pxMBFrameCBByteReceived 将指向接收中断状态机函数 BOOL xMBRTUReceiveFSM(void)；pxMBFrameCBTransmitterEmpty 将指向发送中断状态机函数 BOOL xMBRTUTransmitFSM(void)；pxMBPortCBTimerExpired 将指向超时中断函数 BOOL xMBRTUTimerT35Expired(void)。

完成相应函数指针的初始化之后，会调用该模式的初始化函数完成相应从机地址 ucMBAddress、从机端口 ucPort、从机通信速率 ulBaudRate、从机校验方式 eParity 的初始化。

◆ **eMBErrorCode eMBTCPInit(USHORT ucTCPPort)**

Modbus TCP 模式初始化函数。只有当配置文件使能对应的宏开关 MB_TCP_ENABLED 时，才会编译该函数。该函数会初始化所使用的 TCP/IP 端口号，并初始化相应的函数指针。

◆ **eMBErrorCode eMBRegisterCB(UCHAR ucFunctionCode, pxMBFunctionHandler pxHandler)**

Modbus 功能注册函数。通过该函数，可以定义 FreeModbus 协议外的功能代码，并注册相应的功能函数。具体如何实现还没具体看。

◆ **eMBErrorCode eMBClose(void)**

Modbus 端口关闭函数。该函数通过函数指针 pvMBFrameCloseCur（指向通讯端口关闭函数）来停止止 Modbus 端口上的通讯。

◆ **eMBErrorCode eMBEnable(void)**

Modbus 协议开始函数。该函数通过函数指针 pvMBFrameStartCur（指向相应模式下的使能函数）来使能 Modbus 通讯。

◆ **eMBErrorCode eMBDisable(void)**

Modbus 协议终止函数。该函数通过函数指针 pvMBFrameStopCur（指向相应模式下的终止函数）来终止 Modbus 协议。

◆ **eMBErrorCode eMBPoll(void)**

Modbus 事件轮询处理函数。该函数通过查询底层返回来的事件 eEvent 来决定当前该处理的事务。

处理过程为：若事件为 EV_READY（Startup finished)，则跳出，等待下一次查询；若事件为 EV_FRAME_RECEIVED（Frame received)，则通过函数指针 peMBFrameReceiveCur（指向接收帧信息提取函数）来完成帧信息的提取，并向系统发送 EV_EXECUTE（Execute function）事件；若事件为 EV_EXECUTE，则根据已经从帧信息中提取到的功能码在 xFuncHandlers 中查询对应的功能函数指针，查找到后通过函数指针调用相应的功能处理函数来完成帧信息的处理（向缓存数组中存放回复 PDU)，完成处理后，通过函数指针 peMBFrameSendCur 调用帧发送函数完成回复帧的发送；若事件为 EV_FRAME_SENT（Frame sent)，则跳出，等待下一次查询。

3. 关于 **mbconfig.h** 文件的理解

此文件为 Modbus 协议的配置文件。在移植时，应根据所选用的目标处理器灵活修改此文件，使之满足需要而代码最小。当然，若果你的处理器处理能力足够强，可以保持默认配置，或是根据需要，增加相应的功能的配置宏。文件内容及相应解释如下：

```
#ifndef _MB_CONFIG_H
#define _MB_CONFIG_H

//外部 C 编译器宏开关
#ifdef __cplusplus
PR_BEGIN_EXTERN_C
#endif
```

```
/* ----------------------- Defines ------------------------------------------*/
/*! \brief If Modbus ASCII support is enabled. */
#define MB_ASCII_ENABLED                    (  1 )

/*! \brief If Modbus RTU support is enabled. */
#define MB_RTU_ENABLED                      (  1 )

/*! \brief If Modbus TCP support is enabled. */
#define MB_TCP_ENABLED                      (  0 )

/*! \brief The character timeout value for Modbus ASCII. */
#define MB_ASCII_TIMEOUT_SEC                (  1 )

/*! \brief Timeout to wait in ASCII prior to enabling transmitter. */
#ifndef MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS
#define MB_ASCII_TIMEOUT_WAIT_BEFORE_SEND_MS    ( 0 )
#endif

/*! \brief Maximum number of Modbus functions codes the protocol stack */
#define MB_FUNC_HANDLERS_MAX                ( 16 )

/*! \brief Number of bytes which should be allocated for the <em>Report Slave ID */
#define MB_FUNC_OTHER_REP_SLAVEID_BUF       ( 32 )

/*! \brief If the <em>Report Slave ID</em> function should be enabled. */
#define MB_FUNC_OTHER_REP_SLAVEID_ENABLED   (  1 )

/*! \brief If the <em>Read Input Registers</em> function should be enabled. */
#define MB_FUNC_READ_INPUT_ENABLED          (  1 )

/*! \brief If the <em>Read Holding Registers</em> function should be enabled. */
#define MB_FUNC_READ_HOLDING_ENABLED        (  1 )

/*! \brief If the <em>Write Single Register</em> function should be enabled. */
#define MB_FUNC_WRITE_HOLDING_ENABLED       (  1 )

/*! \brief If the <em>Write Multiple registers</em> function should be enabled. */
#define MB_FUNC_WRITE_MULTIPLE_HOLDING_ENABLED  (  1 )

/*! \brief If the <em>Read Coils</em> function should be enabled. */
#define MB_FUNC_READ_COILS_ENABLED          (  1 )

/*! \brief If the <em>Write Coils</em> function should be enabled. */
#define MB_FUNC_WRITE_COIL_ENABLED          (  1 )

/*! \brief If the <em>Write Multiple Coils</em> function should be enabled. */
```

```
#define MB_FUNC_WRITE_MULTIPLE_COILS_ENABLED       (  1 )

/*! \brief If the <em>Read Discrete Inputs</em> function should be enabled. */
#define MB_FUNC_READ_DISCRETE_INPUTS_ENABLED       (  1 )

/*! \brief If the <em>Read/Write Multiple Registers</em> function should be enabled. */
#define MB_FUNC_READWRITE_HOLDING_ENABLED          (  1 )

/*! @} */
#ifdef __cplusplus
    PR_END_EXTERN_C
#endif
#endif
```

六、 一些小技巧

1、若 Buffer 的最后两个字节为 16 为 CRC 校验值，则对整个 Buffer 校验时，值为 0；