伍陆电子 Modbus 开发攻略--基于 STM32F4xx



开源共享　　携手共进

产品购买地址：

（1）购买地址（伍陆电子授权店铺）

http://www.56dz.com

（2）资料下载

http://www.56dz.com

（3）技术支持

伍陆电子官网：www.56dz.com

伍陆电子

伍陆电子 Modbus 开发攻略基于 STM32F407.....1
产品购买地址：.....1
第 1 章 Modbus 协议简介.....1
1.1 Modbus 协议概述.....2
1.2 Modbus 主/从协议原理.....3
1.3 Modbus 消息帧.....4
1.4 传输模式.....5
1.4.1 RTU 传输模式.....5
1.4.1.1 Modbus 报文 RTU 帧.....6
1.4.1.2 CRC 校验.....8
1.4.2 ASCII 传输模式.....9
1.4.2.1 Modbus ASCII 报文帧.....10
1.4.2.2 LRC 校验.....12
1.5 Modbus 事务处理流程.....12
1.6 异常码定义.....13
课后作业.....14
第 2 章 Modbus 功能码定义.....15
2.1 功能码分类.....16
2.2 公共功能码定义.....17
2.3 常用功能码介绍.....17
2.3.1 读线圈寄存器 01H.....17
2.3.2 读离散输入寄存器 02H.....19
2.3.3 读保持寄存器 03H.....20
2.3.4 输入寄存器 04H.....21
2.3.5 写单个线圈寄存器 05H.....23
2.3.6 写单个保持寄存器 06H.....24
2.3.7 写多个保持寄存器 10H.....24
2.3.8 写多个线圈寄存器 0FH.....26
课后作业.....27
第 3 章 Modbus 移植实验.....28
3.1 FreeModbus 源码获取及介绍.....29
3.1.1 FreeModbus 简介.....29
3.1.2 FreeModbus 源码的获取.....30
3.1.3 FreeModbus 源码文件的介绍.....31
3.2 FreeModbus 移植到 STM32.....33
3.2.1 基础工程准备.....33
3.2.2 添加 FreeModbus 至基础工程.....33
3.2.3 修改 FreeModbus 中 portserial.c 文件.....35
3.2.4 修改 FreeModbus 中 porttimer.c 文件.....41
3.2.5 修改 FreeModbus 中 port.h 文件.....43
3.2.6 修改 FreeModbus 中 demo.c 文件.....46
3.2.7 修改 main.c 文件.....52
3.3 实验现象.....54
3.3.1 ModbusPoll 工具的安装和使用.....55

3.3.2 Modbus 功能测试..... 57
课后作业..... 61

伍陆电子

第 1 章 Modbus 协议简介

我们知道，基于STM32 设计的产品在工业控制应用领域是非常广泛的，而工业控制应用中Modbus协议又是极其重要的。相信很多学习电子技术的朋友不太了解甚至没有听说过Modbus协议，这是因为它主要应用在工业控制领域。基于STM32 芯片在工业应用领域的定位，那么就很有必要理解和掌握Modbus 协议的应用，也为将来开发工业控制产品做好铺垫。本章分为如下几部分内容：

1.1 Modbus 协议概述

1.2 Modbus 主/从协议原理

1.3 Modbus 消息帧

1.4 传输模式

1.5 Modbus 事务处理流程

1.6 异常码定义

伍陆电子

1.1 Modbus 协议概述

Modbus 是由 Modicon（现为施耐德电气公司的一个品牌）在 1979 年发明的，是全球第一个真正用于工业现场的总线协议。

ModBus 网络是一个工业通信系统，由带智能终端的可编程序控制器和计算机通过公用线路或局部专用线路连接而成。其系统结构既包括硬件、亦包括软件。它可应用于各种数据采集和过程监控。

为更好地普及和推动 Modbus 在基于以太网上的分布式应用，目前施耐德公司已将 Modbus 协议的所有权移交给 IDA（Interface for Distributed Automation，分布式自动化接口）组织，并成立了 Modbus-IDA 组织，为 Modbus 今后的发展奠定了基础。

在中国，Modbus 已经成为国家标准。

标准编号：GB/T19582-2008

标准名称：《基于 Modbus 协议的工业自动化网络规范》

分 3 个部分：

《GB/T 19582.1-2008 第 1 部分：Modbus 应用协议》

《GB/T 19582.2-2008 第 2 部分：Modbus 协议在串行链路上的实现指南》

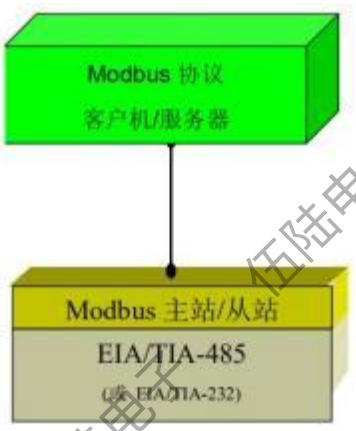
《GB/T 19582.3-2008 第 3 部分：Modbus 协议在 TCP/IP 上的实现指南》

本教程仅涉及第 1 部分和第 2 部分，串行链路仅介绍 RS485 网络。

本章节的介绍文字，摘自 GB/T19582 国标文件，我们摘抄重要的知识点进行整理汇总。

按照 7 层 OSI 通信模型，Modbus 标准包括应用层、数据链路层、物理层。

层	ISO/OSI 模型	
7	应用层	Modbus 协议
6	表示层	空
5	会话层	空
4	传输层	空
3	网络层	空
2	数据链路层	Modbus 串行链路协议
1	物理层	EIA/TIA-485 (或 EIA/TIA-232)



Modbus 串行链路协议是一个主/从协议。该协议位于 OSI 模型的第二层。

一个主从类型的系统有一个向某个“子”节点发出显式命令并处理响应的节点(主节点)。典型的子节点在没有收到主节点的请求时并不主动发送数据，也不与其它子节点通信。

在物理层，Modbus 串行链路系统可以使用不同的物理接口(RS485、RS232)。最常用的是 TIA/EIA-485 (RS485) 两线制接口

1.2 Modbus 主/从协议原理

Modbus 串行链路协议是一个主-从协议。在同一时刻，只有一个主节点连接于总线，一个或多个子节点 (最大编号为 247) 连接于同一个串行总线。Modbus 通信总是由主节点发起。子节点在没有收到来自主节点的请求时，从不会发送数据。子节点之间从不会互相通信。主节点在同一时刻只会发起一个Modbus 事务处理。**为了方便理解，我们通常也将主节点及子节点称为主设备和从设备。**

主节点以两种模式对子节点发出 Modbus 请求:

●单播模式

主节点以特定地址访问某个子节点，子节点接到并处理完请求后，子节点向主节点返回一个报文(一个’应答’)。在这种模式，一个 Modbus 事务处理包含 2 个报文：一个来自主节点的请求，一个来自子节点的应答。

每个子节点必须有唯一的地址 **(1 到 247)**，这样才能区别于其它节点被独立的寻址。

●广播模式

主节点向所有的子节点发送请求。对于主节点广播的请求没有应答返回。广

播请求一般用于写命令。

所有设备必须接受广播模式的写功能。**地址 0** 是专门用于表示广播数据的。

地址规则：

Modbus 寻址空间有 256 个不同地址。

0	1～247	248～55
广播地址	子节点单独地址	保留

地址 0 为广播地址。所有的子节点必须识别广播地址。

Modbus 主节点没有地址，只有子节点必须有一个地址。 该地址必须在 Modbus 串行总线上唯一。

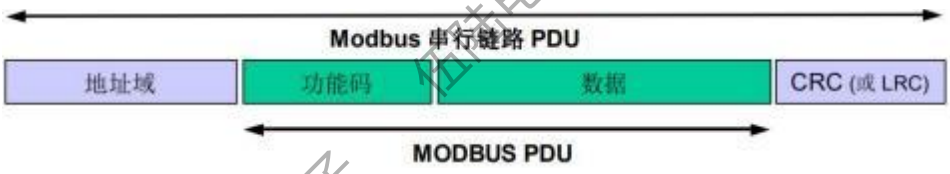
当数据帧到达终端设备（从设备）时，它通过一个简单的”端口“进入被寻址到的设备，该设备去掉数据帧的“信头”（数据头），读取数据，如果没有错误，就执行数据所请求的任务，然后将自己生成的数据加入到取得的“信封”中，把数据帧返回给发送者。返回的响应数据中包含了以下内容：终端从机地址、被执行了的命令、执行命令生成的被请求数据和一个校验码。发生任何错误都不会有成功的响应，或者返回一个错误指示帧。

1.3 Modbus 消息帧

《Modbus 应用协议》定义了简单的协议数据单元(PDU - Protocol Data Unit)。



发起 Modbus 事务处理的客户端构造 Modbus PDU，然后添加附加的域以构造通信 PDU。



●在 Modbus 串行链路，地址域只含有子节点地址。

如前面所述，合法的子节点地址为十进制 0 -247。 每个子设备被赋予 1 - 247 范围中的地址。主节点通过将子节点的地址放到报文的地址域对子节点寻址。当子节点返回应答时，它将自己的地址放到应答报文的地址域以让主节点知

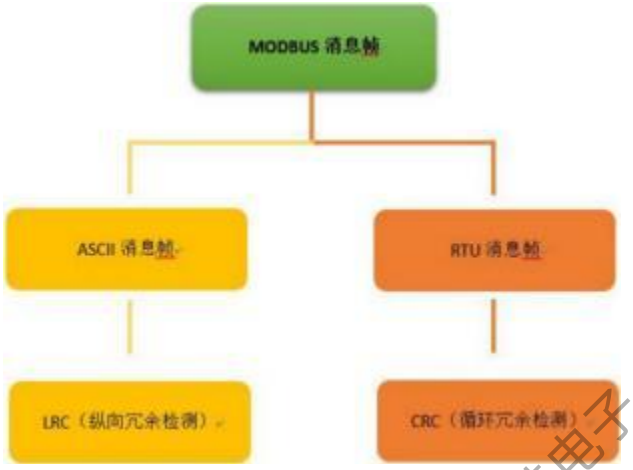
道哪个子节点在回答。

●功能码指明服务器要执行的动作。功能码后面可跟有表示含有请求和响应参数的数据域。

●错误检验域是对报文内容执行"冗余校验"的计算结果。根据不同的传输模式 (RTU or ASCII) 使用两种不同的计算方法。

1.4 传输模式

控制器能设置为两种传输模式（ASCII和 RTU）中的任何一种在标准的Modbus网络通信。用户选择想要的模式，包括串口通信参数（波特率、校验方式等），在配置每个控制器的时候，在一个Modbus 网络上所有设备都必须选择相同的传输模式和串口参数。下图为两种模式的分类：



1.4.1 RTU 传输模式

当设备使用 RTU (Remote Terminal Unit) 模式在 Modbus 串行链路通信，消息报文中每个 8 位字节含有两个 4 位十六进制字符。这种模式的主要优点是较高的数据密度，在相同的波特率下比 ASCII 模式有更高的吞吐率。RTU 模式的每个报文必须以连续的字符流传送。

RTU 模式每个字节 （ 11 位 ） 的格式为：

●编码系统：8 位二进制，报文中每个 8 位字节含有两个 4 位十六进制字符(0 -9， A-F)。

●每字节的 bit 流：1 起始位、8 数据位，首先发送最低有效位、1 位作

为奇偶校验、1 停止位。

偶校验是要求的，其它模式(奇校验，无校验)也可以使用。为了保证与其它产品的最大兼容性，同时支持无校验模式是建议的。默认校验模式模式必须为偶校验。

注：使用无校验要求 2 个停止位。

字符是如何串行传送的：

每个字符或字节均由此顺序发送(从左到右)：最低有效位 (LSB)...最高有效位(MSB)。



设备配置为奇校验、偶校验或无校验都可以接受。如果无奇偶校验，将传送一个附加的停止位以填充字符帧：



帧检验域：循环冗余校验 (CRC)，2 字节。

帧描述：

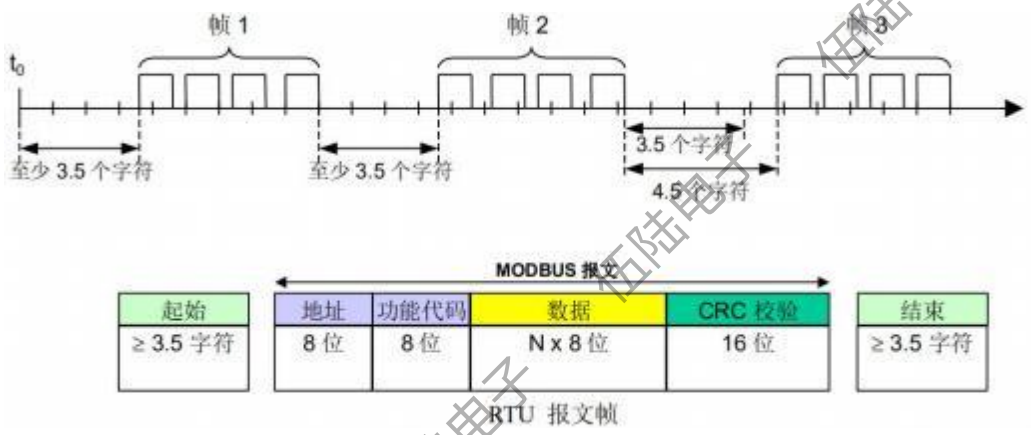
子节点地址	功能代码	数据	CRC
1 字节	1 字节	0 到 252 字节	2 字节 CRC 低 | CRC 高

Modbus RTU 帧总长度最大为 256 字节。

1.4.1.1 Modbus 报文 RTU 帧

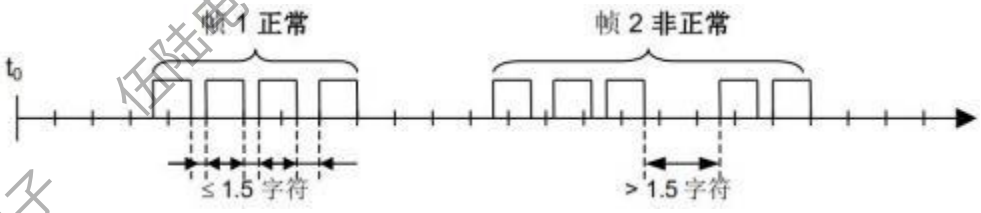
由发送设备将 Modbus 报文构造为带有已知起始和结束标记的帧。这使设备可以在报文的开始接收新帧，并且知道何时报文结束。不完整的报文必须能够被检测到而错误标志必须作为结果被设置。

在 RTU 模式，**报文帧由时长至少为 3.5 个字符时间的空闲间隔区分**。在后续的部分，这个时间区间被称作 t3.5。



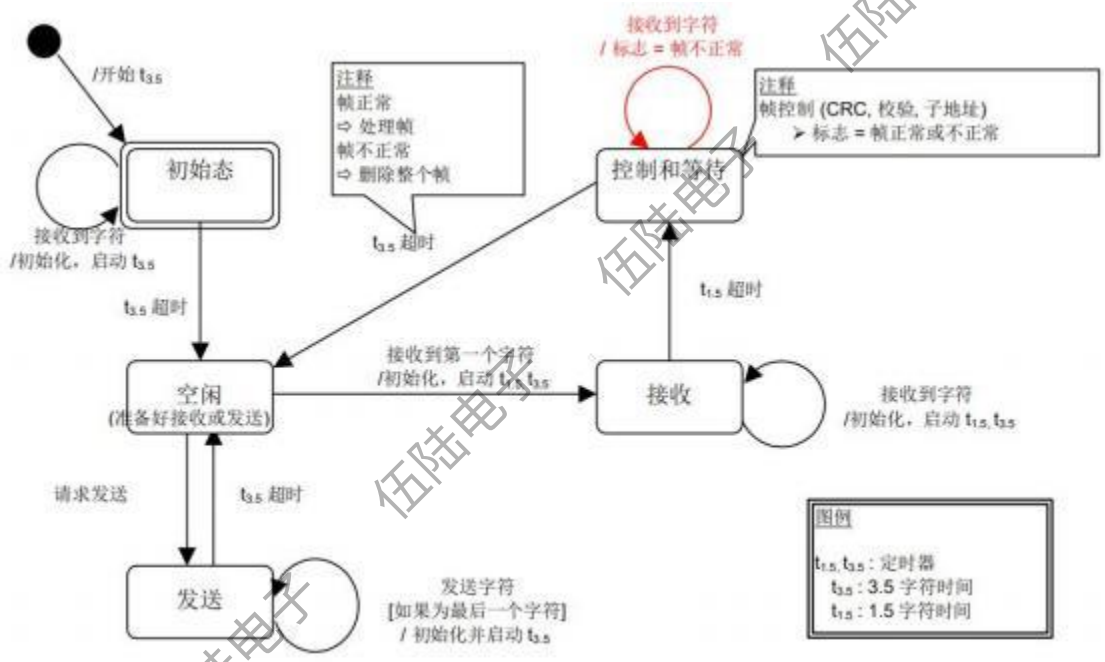
整个报文帧必须以连续的字符流发送。

如果两个字符之间的空闲间隔大于 1.5 个字符时间，则报文帧被认为不完整应该被接收节点丢弃。如下：



注：RTU 接收驱动程序的实现，由于 t1.5 和 t3.5 的定时，隐含着大量的对中断的管理。在高通信速率下，这导致 CPU 负担加重。因此，在通信速率等于或低于 19200 bps 时，这两个定时必须严格遵守；对于波特率大于 19200 bps 的情形，应该使用 2 个定时的固定值：**建议的字符间超时时间(t1.5)为 750μs，帧间的超时时间 (t3.5) 为 1.750ms。**

下图表示了对 RTU 传输模式状态图的描述。"主节点"和"子节点"的不同角度均在相同的图中表示：



RTU 传输模式状态图

上面状态图的一些解释:

●从"初始"态到 "空闲" 态转换需要 t3.5 定时超时: 这保证帧间延迟

●"空闲"态是没有发送和接收报文要处理的正常状态。

●在 RTU 模式，当没有活动的传输的时间间隔达 3.5 个字符长时，通信链路被认为在"空闲"态。

●当链路空闲时，在链路上检测到的任何传输的字符被识别为帧起始。链路变为"活动"状态。 然后，当链路上没有字符传输的时间间个达到 t3.5 后，被识别为帧结束。

●检测到帧结束后，完成 CRC 计算和检验。然后，分析地址域以确定帧是否发往此设备，如果不是，则丢弃此帧。为了减少接收处理时间，地址域可以在一接到就分析，而不需要等到整个帧结束。这样，CRC 计算只需要在帧寻址到该节点 (包括广播帧) 时进行。

1.4.1.2 CRC 校验

在 RTU 模式包含一个对全部报文内容执行的，基于循环冗余校验 (CRC - Cyclical RedundancyChecking) 算法的错误检验域。CRC 域检验整个报文的内容。不管报文有无奇偶校验，均执行此检验。

CRC 包含由两个 8 位字节组成的一个 16 位值。

CRC 域作为报文的最后的域附加在报文之后。计算后，首先附加低字节，然后是高字节。CRC 高字节为报文发送的最后一个子节。

附加在报文后面的 CRC 的值由发送设备计算。接收设备在接收报文时重新计算 CRC 的值，并将计算结果于实际接收到的 CRC 值相比较。如果两个值不相等，则为错误。

CRC 的计算，开始对一个 16 位寄存器预装全 1。然后将报文中的连续的 8 位字节对其进行后续的计算。只有字符中的 8 个数据位参与生成 CRC 的运算，起始位，停止位和校验位不参与 CRC 计算。

CRC 的生成过程中，每个 8 位字符与寄存器中的值异或。然后结果向最低有效位(LSB)方向移动(Shift) 1 位，而最高有效位(MSB)位置充零。然后提取并检查 LSB：如果 LSB 为 1，则寄存器中的值与一个固定的预置值异或；如果 LSB 为 0，则不进行异或操作。

这个过程将重复直到执行完 8 次移位。完成最后一次（第 8 次）移位及相关操作后，下一个 8 位字节与寄存器的当前值异或，然后又同上面描述过的一样重复 8 次。当所有报文中子节都运算之后得到的寄存器中的最终值，就是 CRC。

1.4.2 ASCII 传输模式

当 Modbus 串行链路的设备被配置为使用 ASCII (American Standard Code for Information Interchange) 模式通信时，报文中的每个 8 位字节以两个 ASCII 字符发送。当通信链路或者设备无法符合 RTU 模式的定时管理时使用该模式。

注：由于一个子节需要两个字符，此模式比 RTU 效率低。

例：子节 0X5B 会被编码为两个字符：0x35 和 0x42 （ ASCII 编码 0x35 ="5"，0x42 ="B" ）。

ASCII 模式每个字节（ 10 位 ） 的格式为：

●编码系统：十六进制，ASCII 字符 0-9，A-F。报文中每个 ASCII 字符含有 1 个十六进制字符。

●每字节bit流: 1 起始位、7 数据位，首先发送最低有效位、1 位作为奇偶校验、1 停止位。

偶校验是要求的，其它模式 （ 奇校验， 无校验 ） 也可以使用。为了保证与其它产品的最大兼容性，同时支持无校验模式是建议的。默认校验模式模式 必须为偶校验。

注：使用无校验要求 2 个停止位。

字符是如何串行传送的:

每个字符或字节均由此顺序发送(从左到右):最低有效位 (LSB)...最高有效位 (MSB)。



设备配置为奇校验、偶校验或无校验都可以接受。如果无奇偶校验，将传送一个附加的停止位以填充字符帧:



帧检验域: 纵向冗余校验 (LRC - Longitudinal Redundancy Checking)。

1.4.2.1 Modbus ASCII 报文帧

由发送设备将 Modbus 报文构造为带有已知起始和结束标记的帧。这使设备可以在报文的开始接收新帧，并且知道何时报文结束。不完整的报文必须能够被检测到而错误标志必须作为结果被设置。

报文帧的地址域含有两个字符。

在 ASCII 模式，报文用特殊的字符区分帧起始和帧结束。一个报文必须以一个‘冒号’(:) (ASCII 十六进制 3A)起始，以‘回车-换行’(CR LF) 对 (ASCII 十六进制 OD 和 OA) 结束。注：LF 字符可以通过特定的 Modbus 应用命令 (参见 Modbus 应用协议规范) 改变。

对于所有的域，允许传送的字符为十六进制 0 -9，A -F (ASCII 编码)。设备连续的监视总线上的‘冒号’字符。 当收到这个字符后，每个设备解码后续

的字符一直到帧结束。

报文中字符间的时间间隔可以达一秒。如果有更大的间隔，则接受设备认为发生了错误。

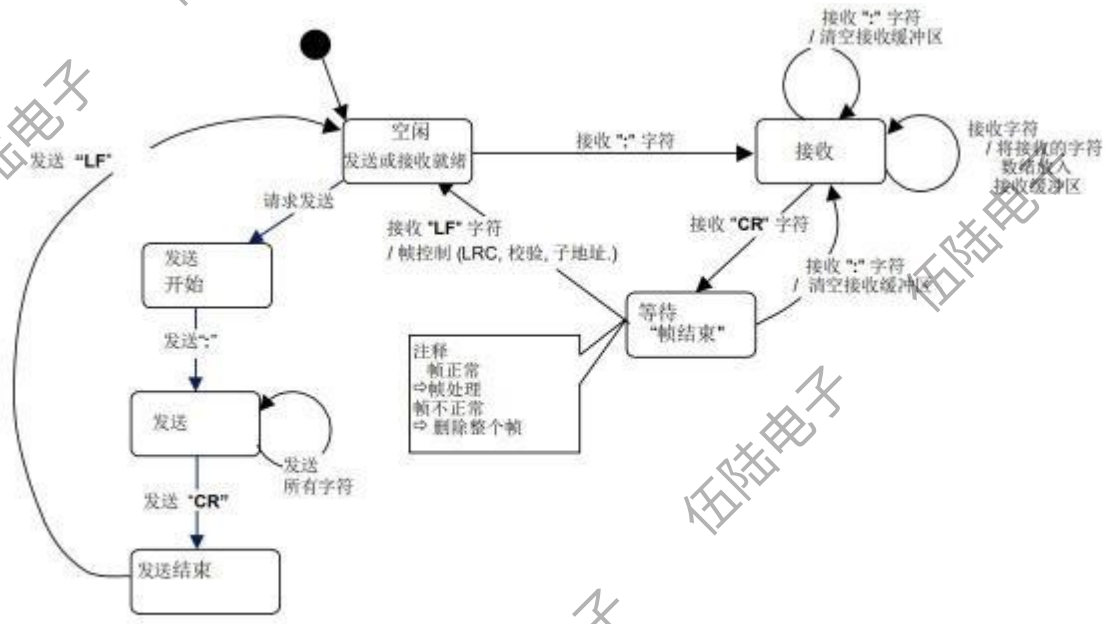
下图显示了一个典型的ASCII报文帧：

起始	地址	功能	数据	LRC	结束
1 字符 :	2 字符	2 字符	0 到 to 2x252 字符	2 字符	2 字符 CR,LF

注：每个字符字节需要用两个字符编码。因此，为了确保 ASCII 模式 和 RTU 模式在 Modbus 应用级兼容，ASCII 数据域最大数据长度为 (2x252) 是 RTU 数据域 (252) 的两倍。

必然的，Modbus ASCII 帧的最大尺寸为 513 个字符。

ASCII 报文帧的要求在下面的状态图中综合。 ″主节点″ 和 ″子节点″ 的不同角度均在相同的图中表示：



ASCII 传输模式状态图

上面状态图的一些解释：

●“空闲 ”态是没有发送和接收报文要处理的正常状态。

●每次接收到 ″:″ 字符表示新的报文的开始。如果在一个报文的接收过程中收到该字符，则当前地报文被认为不完整并被丢弃。而一个新的接收缓冲区被重新分配。

●检测到帧结束后，完成 LRC 计算和检验。然后，分析地址域以确定帧是

否发往此设备，如果不是，则丢弃此帧。为了减少接收处理时间，地址域可以在一接到就分析，而不需要等到整个帧结束。

1.4.2.2 LRC 校验

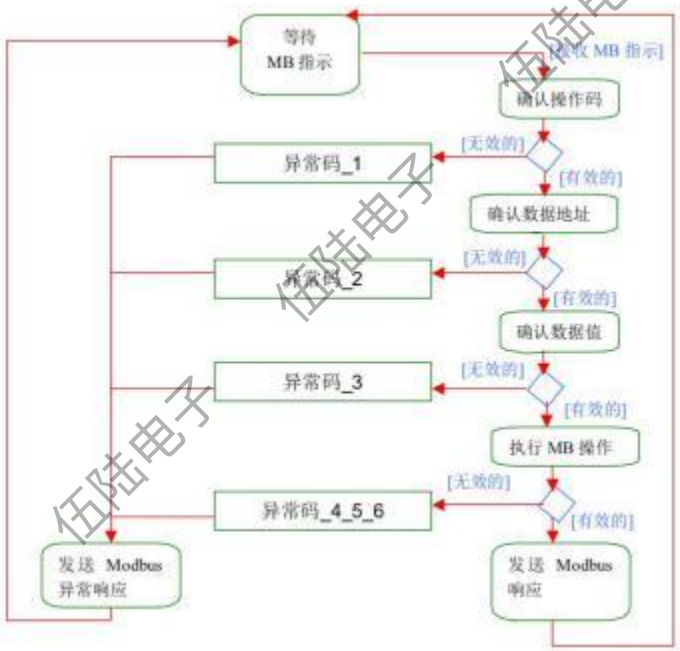
在 ASCII 模式，包含一个对全部报文内容执行的，基于纵向冗余校验 (LRC - Longitudinal Redundancy Checking) 算法的错误检验域。LRC 域检验不包括起始“冒号”和结尾 CRLF 对的整个报文的内容。不管报文有无奇偶校验，均执行此检验。

LRC 域为一个子节，包含一个 8 位二进制值。LRC 值由发送设备计算，然后将 LRC 附在报文后面。接收设备在接收报文时重新计算 LRC 的值，并将计算结果于实际接收到的 LRC 值相比较。如果两个值不相等，则为错误。

LRC 的计算，对报文中的所有的连续 8 位字节相加，忽略任何进位，然后求出其二进制补码。执行检验针对不包括起始“冒号”和结尾 CRLF 对的整个 ASCII 报文域的内容。在 ASCII 模式， LRC 的结果被 ASCII 编码为两个字节并放置于 ASCII 模式报文帧的结尾， CRLF 之前。

1.5 Modbus 事务处理流程

下列状态图描述了在服务器侧 MODBUS 事务处理的一般处理过程。



一旦服务器处理请求，使用合适的 MODBUS 服务器事务建立 MODBUS 响应。根据处理结果，可以建立两种类型响应：

●一个正确的 MODBUS 响应：响应功能码 = 请求功能码

●一个 MODBUS 异常响应

●用来为客户机提供处理过程中与被发现的差错相关的信息；

●响应功能码 = 请求功能码 + 0x80；

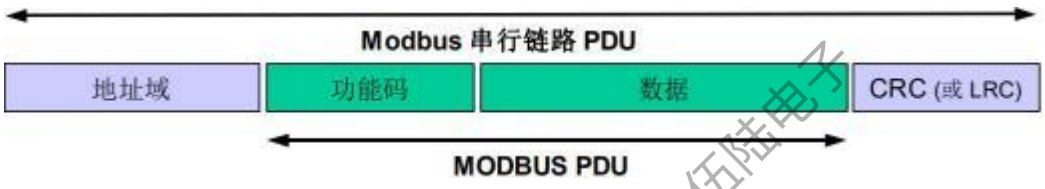
●提供一个异常码来指示差错原因。

1.6 异常码定义

客户机请求和服务器异常响应的实例：（仅功能码和数据区）

请求		响应	
域名	（十六进制）	域名	（十六进制）
功能	01	功能	81
起始地址 Hi	04	异常码	02
起始地址 Lo	A1		
输出数量 Hi	00		
输出数量 Lo	01		

上面仅列出了功能码域和数据域，对于完整的 RTU 帧，还应该有地址域和 CRC 域。



在这个实例中，客户机对服务器设备寻址请求。功能码(01)用于读输出状态操作。它将请求地址 1245(十六进制 04A1)的输出状态。值得注意的是，像输出域(0001)号码说明的那样，只读出一个输出。

如果在服务器设备中不存在输出地址，那么服务器将返回异常码(02)的异常响应。这就说明从站的非法数据地址。

MODBUS 异常码		
代码	名称	含义
01	非法功能	对于服务器(或从站)来说，询问中接收到的功能码是不可允许的操作。这也许是因为功能码仅仅适用于新设备而在被选单元中是不可实现的。同时，还指出服务器(或从站)在错误状态中处理这种请求，例如：因为它是未配置的，并且要求返回寄存器值。
02	非法数据地址	对于服务器(或从站)来说，询问中接收到的数据地址是不可允许的地址。特别是，参考号和传输长度的组合是无效的。对于带有 100 个寄存器的控制器来说，带有偏移量 96 和长度 4 的请求会成功，带有偏移量 96 和长度 5 的请求将产生异常码 02。
03	非法数据值	对于服务器(或从站)来说，询问中包括的值是不可允许的值。这个值指示了组合请求剩余结构中的故障，例如：隐含长度是不正确的。并不意味着，因为 MODBUS 协议不知道任何特殊寄存器的任何特殊值的重要意义，寄存器中被提交存储的数据项有一个应用程序期望之外的值。
04	从站设备故障	当服务器(或从站)正在设法执行请求的操作时，产生不可重新获得的差错。

下面的不常用		
05	*确认*	*与编程命令一起使用。服务器(或从站)已经接受请求，并切正在处理这个请求，但是需要长的持续时间进行这些操作。返回这个响应防止在客户机(或主站)中发生超时错误。客户机(或主站)可以继续发送轮询程序完成报文来确定是否完成处理。*
06	*从属设备忙*	*与编程命令一起使用。服务器(或从站)正在处理长持续时间的程序命令。张服务器(或从站)空闲时，用户(或主站)应该稍后重新传输报文。*
08	*存储奇偶性差错*	*与功能码 20 和 21 以及参考类型 6 一起使用，指示扩展文件区不能通过一致性校验。* *服务器(或从站)设法读取记录文件，但是在存储器中发现一个奇偶校验错误。客户机(或主方)可以重新发送请求，但可以在服务器(或从站)设备上要求服务。*
0A	*不可用网关路径*	*与网关一起使用，指示网关不能为处理请求分配输入端口至输出端口的内部通信路径。通常意味着网关是错误配置的或过载的。*
0B	*网关目标设备响应失败*	*与网关一起使用，指示没有从目标设备中获得响应。通常意味着设备未在网络中。*

课后作业

第 2 章 Modbus 功能码定义

本章将向大家介绍Modbus 功能码，在使用Modbus 时，这些功能码是关键指令。本章分为如下几部分内容：

2.1 功能码分类

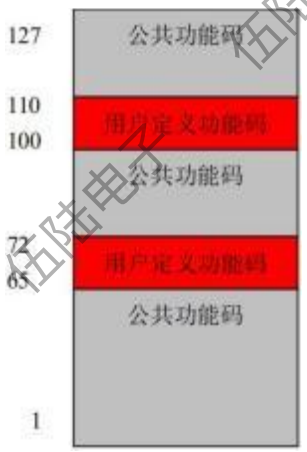
2.2 公共功能码定义

2.3 常用功能码介绍

伍陆电子

2.1 功能码分类

有三类 MODBUS 功能码。它们是：公共功能码、用户定义功能码、保留功能码。



公共功能码：

- ●是较好地被定义的功能码，
- ●保证是唯一的，
- ●MODBUS 组织可改变的，
- ●公开证明的，
- ●具有可用的一致性测试，
- ●MB IETF RFC 中证明的，
- ●包含已被定义的公共指配功能码和未来使用的未指配保留供功能码。

用户定义功能码：

- ●有两个用户定义功能码的定义范围，即 65 至 72 和十进制 100 至 110。
- ●用户没有 MODBUS 组织的任何批准就可以选择和实现一个功能码
- ●不能保证被选功能码的使用是唯一的。
- ●如果用户要重新设置功能作为一个公共功能码，那么用户必须启动 RFC，以便将改变引入公共分类中，并且指配一个新的公共功能码

保留功能码：

- ●一些公司对传统产品通常使用的功能码，并且对公共使用是无效的功能码。

如果我们需要实现自定义的一些功能码，必须在 65-72 或 100-110 中进行选择。

2.2 公共功能码定义

Modbus数据模型有四种，通过不同的功能码来读写这些数据对象。

对象类别	对象类型	访问类型	内容
离散量输入	单个比特	只读	I/O 系统提供这种类型数据
线圈	单个比特	读写	通过应用程序改变这种类型数据
输入寄存器	16-比特字	只读	I/O 系统提供这种类型数据
保持寄存器	16-比特字	读写	通过应用程序改变这种类型数据

功能码定义如下：

		数据类型	功能描述	功能码		
				码	子码	（十六进制）
数据访问	比特访问	物理离散量输入	读输入离散量	02		02
		内部比特或物理线圈	读线圈	01		01
			写单个线圈	05		05
			写多个线圈	15		0F
	16比特访问	输入存储器	读输入寄存器	04		04
		内部存储器或物理输出存储器	读多个寄存器	03		03
			写单个寄存器	06		06
			写多个寄存器	16		10
			读/写多个寄存器	23		17
			屏蔽寄存器	22		16
	文件记录访问		读文件记录	20	6	14
			写文件记录	21	6	15
封装接口			读设备识别码	43	14	2B

最常用的功能码有：01、02、03、04、05、15、16。

下面，我们会介绍每个功能码对应的数据域格式。

2.3 常用功能码介绍

2.3.1 读线圈寄存器 01H

●描述：

读 MODBUS 从机线圈寄存器当前状态。

●查询：

例如从机地址为 11H，线圈寄存器的起始地址为 0013H，结束地址为 0037H。该次查询总共访问 37 个线圈寄存器（0X37-0X13+1=0X25，将其转换为十进制=37）。

	Hex
从机地址	11
功能码	01
寄存器起始地址高字节	00
寄存器起始地址低字节	13
寄存器数量高字节	00
寄存器数量低字节	25
CRC 校验高字节	0E
CRC 校验低字节	84

●响应：

响应负载中的各线圈状态与数据内容每位相对应。1 代表 ON，0 代表 OFF。若返回的线圈数不为 8 的倍数，则在最后数据字节未尾使用 0 代替（因为传输是从第到高的，即字节高位填充0）。

	Hex
从机地址	11
功能码	01
返回字节数	05
数据 1(线圈 0013H-线圈 001AH)	CD
数据 2(线圈 001BH-线圈 0022H)	6B
数据 3(线圈 0023H-线圈 002AH)	B2
数据 4(线圈 0032H-线圈 002BH)	0E
数据 5(线圈 0037H-线圈 0033H)	1B
CRC 校验高字节	45
CRC 校验低字节	E6

线圈 0013H 到线圈 001AH 的状态为 CDH，二进制值为 1100 1101，该字节的最高位为线圈 001AH，最低位为线圈 0013H。线圈 001AH 到线圈 0013H 的状态分别为 ON-ON-OFF-OFF-ON-ON-OFF-ON。如下：（可以理解为小端模式）

001AH	0019H	0018H	0017H	0016H	0015H	0014H	0013H
ON	ON	OFF	OFF	ON	ON	OFF	ON

最后一个数据字节中，线圈 0033H 到线圈 0037 状态为 1BH(二进制 0001 1011)，线圈 0037H 是左数第 4 位（从 0 开始），线圈 0033H 为该字节的最低位，线圈 0037H 至线圈 0033H 的状态分别为 ON-ON-OFF-ON-ON，剩余 3 位使用 0 填充。

003AH	0039H	0038H	0037H	0036H	0035H	0034H	0033H
填充	填充	填充	ON	ON	OFF	ON	ON

2.3.2 读离散输入寄存器 02H

●说明：

读离散输入寄存器状态。

●查询：

从机地址为 11H。离散输入寄存器的起始地址为 00C4H，结束寄存器地址为 00D9H。总共访问 32 个离散输入寄存器。例如：

	Hex
从机地址	11
功能码	02
寄存器地址高字节	00
寄存器地址低字节	C4
寄存器数量高字节	00
寄存器数量低字节	16
CRC 校验高字节	BA
CRC 校验低字节	A9

●响应：

响应各离散输入寄存器状态，分别对应数据区中的每位值，1 代表 ON；0 代表 OFF。第一个数据字节的 LSB(最低字节)为查询的寻址地址，其他输入口按顺序在该字节中由低字节向高字节排列，直到填充满 8 位。下一个字节中的 8 个输入位也是从低字节到高字节排列。若返回的输入位数不是 8 的倍数，则在最后的数据字节中的剩余位至该字节的最高位使用 0 填充。

	Hex
从机地址	11
功能码	02
返回字节数	03
数据 1(00C4H-00CBH)	AC
数据 2(00CCH-00D3H)	DB
数据 3(00D4H-00D9H)	35
CRC 校验高字节	20
CRC 校验低字节	18

离散输入寄存器 00D4H 到 00D9H 的状态为 35H (二进制 0011 0101)。输入寄存器 00D9H 为左数第 3 位，输入寄存器 00D4 为最低位，输入寄存器 00D9H 到 00D4H 的状态分别为 ON-ON-OFF-ON-OFF-ON。00DBH 寄存器和 00DAH 寄存器被 0 填充。如下：

00CBH	00CAH	00C9H	00C8H	00C7H	00C6H	00C5H	00C4H
0	0	1	1	0	1	0	1
00D3H	00D2H	00D1H	00D0H	00CFH	00CEH	00CDH	00CCH
1	1	1	0	1	0	1	1
00DBH	00DAH	00D9H	00D8H	00D7H	00D6H	00D5H	00D4H
填充	填充	1	1	0	1	0	1

2.3.3 读保持寄存器 03H

●说明：

读保持寄存器。可读取单个或多个保持寄存器。

●查询：

从机地址为 11H。保持寄存器的起始地址为 006BH，结束地址为 006DH。该次查询总共访问多个保持寄存器。

	Hex
从机地址	11
功能码	03
寄存器地址高字节	00
寄存器地址低字节	6B
寄存器数量高字节	00
寄存器数量低字节	03
CRC 高字节	76
CRC 低字节	87

●响应：

保持寄存器的长度为 2 个字节。对于单个保持寄存器而言，寄存器高字节数据先被传输，低字节数据后被传输。保持寄存器之间，低地址寄存器先被传输，高地址寄存器后被传输。

	Hex
从机地址	11
功能码	03
字节数	06
数据 1 高字节(006BH)	00
数据 1 低字节(006BH)	6B
数据 2 高字节(006CH)	00
数据 2 低字节(006CH)	13
数据 3 高字节(006DH)	00
数据 3 低字节(006DH)	00
CRC 高字节	38
CRC 低字节	B9

006BH 高字节	006BH 低字节	006CH 高字节	006CH 低字节	006DH 高字节	006DH 低字节
00	6B	00	13	00	00

2.3.4 输入寄存器 04H

●说明：

读输入寄存器命令。该命令支持单个寄存器访问也支持多个寄存器访问。

●查询：

从机地址为 11H。输入寄存器的起始地址为 0008H，寄存器的结束地址为 0009H。本次访问访 2 个输入寄存器。

	Hex 格式
从机地址	11
功能码	04
寄存器起始地址高字节	00
寄存器起始地址低字节	08
寄存器个数高字节	00
寄存器个数低字节	02
CRC 高字节	F2
CRC 低字节	99

●响应：

输入寄存器长度为 2 个字节。对于单个输入寄存器而言，寄存器高字节数据先被传输，低字节数据被传输。输入寄存器之间，低地址寄存器先被传输，高地址寄存器后被传输。

	Hex 格式
从机地址	11
功能码	04
字节数	04
数据 1 高字节(0008H)	00
数据 1 低字节(0008H)	0A
数据 2 高字节(0009H)	00
数据 2 低字节(0009H)	0B
CRC 高字节	8B
CRC 低字节	80

006BH 高字节	006BH 低字节	006CH 高字节	006CH 低字节
00	0A	00	0B

2.3.5 写单个线圈寄存器 05H

●说明：

写单个线圈寄存器。FF00H 值请求线圈处于 ON 状态，0000H 值请求线圈处于 OFF 状态。05H 指令设置单个线圈的状态，15H 指令可以设置多个线圈的状态，两个指令虽然都设定线圈的 ON/OFF 状态，但是 ON/OFF 的表达方式却不同。

●查询：

从机地址为 11H，线圈寄存器的地址为 00ACH。使 00ACH 线圈处于 ON 状态，即数据内容为 FF00H。

	Hex
从机地址	11
功能码	05
寄存器地址高字节	00
寄存器地址低字节	AC
数据 1 高字节	FF
数据 2 低字节	00
CRC 校验高字节	4E
CRC 校验低字节	8B

●响应：

	Hex
从机地址	11
功能码	05
寄存器地址高字节	00
寄存器地址低字节	AC
寄存器 1 高字节	FF
寄存器 1 低字节	00
CRC 校验高字节	4E
CRC 校验低字节	8B

2.3.6 写单个保持寄存器 06H

●说明：

写保持寄存器。注意 06 指令只能操作单个保持寄存器，16 指令可以设置单个或多个保持寄存器。

●查询：

从机地址为 11H。保持寄存器地址为 0001H。寄存器内容为 0003H。

	Hex
从机地址	11
功能码	06
寄存器地址高字节	00
寄存器地址低字节	01
数据 1 高字节	00
数据 1 低字节	01
CRC 校验高字节	9A
CRC 校验低字节	9B

●响应：

	Hex
从机地址	11
功能码	06
寄存器地址高字节	00
寄存器地址低字节	01
寄存器数量高字节	00
寄存器数量低字节	01
CRC 校验高字节	1B
CRC 校验低字节	5A

2.3.7 写多个保持寄存器 10H

●说明：

写多个保持寄存器。

●查询：

从机地址为 11H。保持寄存器的起始地址为 0001H，寄存器的结束地址为 0002H。总共访问 2 个寄存器。保持寄存器 0001H 的内容为 000AH，保持寄存器 0002H 的内容为 0102H。

	Hex
从机地址	11
功能码	10
寄存器起始地址高字节	00
寄存器起始地址低字节	01
寄存器数量高字节	00
寄存器数量低字节	02
字节数	04
数据 1 高字节	00
数据 1 低字节	0A
数据 2 高字节	01
数据 2 低字节	02
CRC 校验高字节	C6
CRC 校验低字节	F0

地址	0001H 高字节	0001H 低字节	0002H 高字节	0003H 低字节
数值	00	0A	01	12

●响应：

	Hex
从机地址	11
功能码	10
寄存器起始地址高字节	00
寄存器起始地址低字节	01
寄存器数量高字节	00
寄存器数量低字节	02
CRC 校验高字节	12
CRC 校验低字节	98

2.3.8 写多个线圈寄存器 0FH

●说明：

写多个线圈寄存器。若数据区的某位值为“1”表示被请求的相应线圈状态为 ON，若某位值为“0”，则为状态为 OFF。

●查询：

从机地址为 11H，线圈寄存器的起始地址为 0013H，线圈寄存器的结束地址为 001CH。总共访问 10 个寄存器。寄存器内容如下表所示。

001AH	0019H	0018H	0017H	0016H	0015H	0014H	0013H
1	1	0	0	1	1	0	1
0022H	0021H	0020H	001FH	001EH	001DH	001CH	001BH
0	0	0	0	0	0	0	1

传输的第一个字节 CDH 对应线圈为 0013H 到 001AH，LSB（最低位）对应线圈 0013H，传输第二个字节为 01H，对应的线圈为 001BH 到 001CH，LSB 对应线圈 001CH，其余未使用位使用 0 填充。

	Hex
从机地址	11
功能码	0F
寄存器地址高字节	00
寄存器地址低字节	13
寄存器数量高字节	00
寄存器数量低字节	0A
字节数	02
数据 1(0013H-001AH)	CD
数据 2(001BH-001CH)	01
CRC 校验高字节	BF
CRC 校验低字节	0B

●响应：

	Hex
从机地址	11
功能码	0F
寄存器地址高字节	00
寄存器地址低字节	13
寄存器数量高字节	00
寄存器数量低字节	0A
字节数	02
CRC 校验高字节	99
CRC 校验低字节	1B

课后作业

第 3 章 Modbus 移植实验

现如今有很多的开源 Modbus 协议代码，例如 FreeModbus、LibModbus 等。但移植Modbus 也不是一件简单的事情，FreeModbus 是一个优秀的应用层协议，它很简洁也相对完善。本章就以FreeModbus 为例，将其移植到伍陆电子 STM32 开发板上使用。本章分为如下几部分内容：

3.1 FreeModbus 源码获取及介绍

3.2 FreeModbus 移植到 STM32

3.3 实验现象

3.1 FreeModbus 源码获取及介绍

3.1.1 FreeModbus 简介

FreeModbus 是一个奥地利人写的Modbus 协议。它是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。

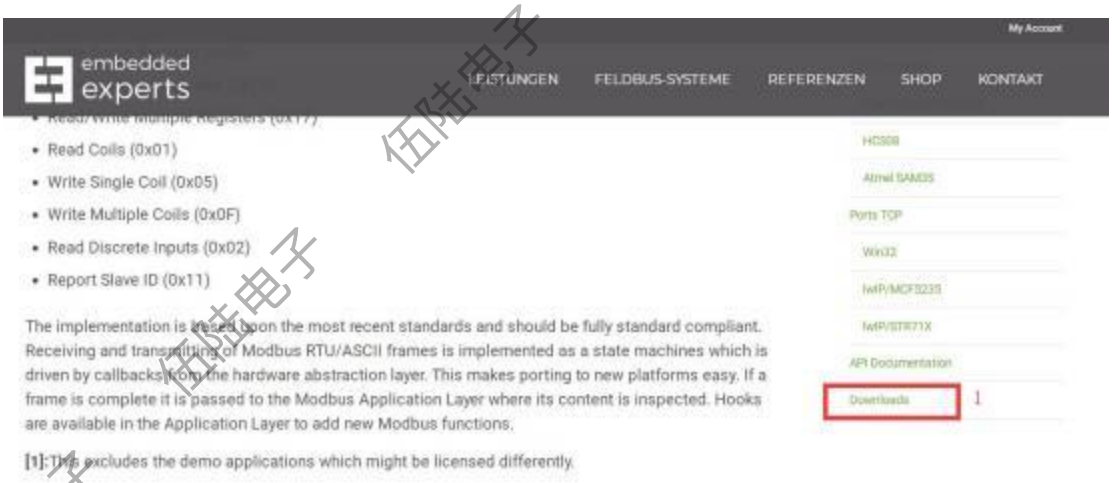
FreeMODBUS 提供了 RTU/ASCII 传输模式及 TCP 协议支持。FreeModbus 遵循 BSD 许可证，这意味着用户可以将FreeModbus应用于商业环境中。版本 FreeModbus-V1.5 提供如下的功能支持：

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通用请求与应答	否	
0x2B/0x0E	读设备身份表示	否	

从上图知道，**本章移植的 FreeModbus 将作为从机使用，暂不支持主机功能**。简单了解了FreeModbus 后，下面介绍如何获取该源码。

3.1.2 FreeModbus 源码的获取

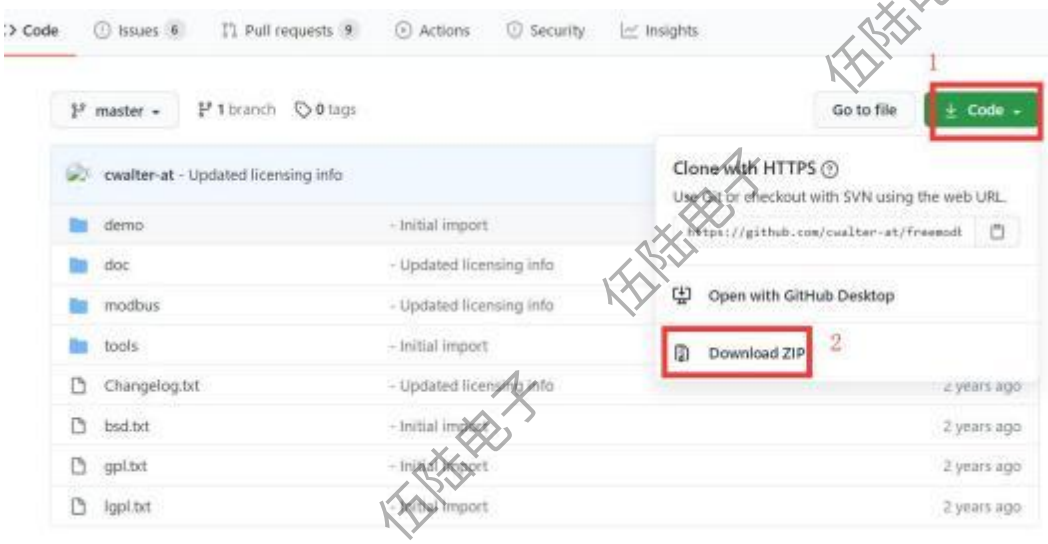
要获取FreeModbus源码，我们可以直接从他们的官网：https://www.embedded-solutions.at/en/freemodbus/获取，点击链接，等待几秒后会自动进入该网页，如下所示：（**注意：为了节省大家的宝贵时间，建议直接使用我们下载好的文件**）



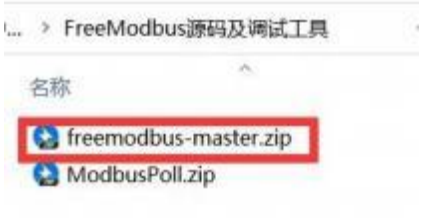
点击“Downloads”，进入如下界面：



如果选择直接下载压缩包会很慢，建议直接进入 GIT 仓库，选择 ZIP 文件下载。如下所示：



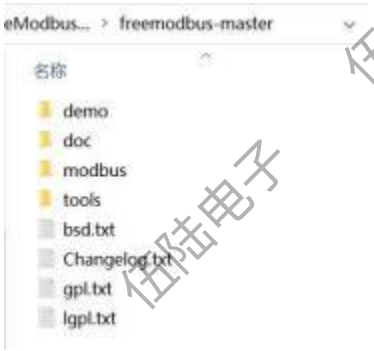
下载速度慢的用户可直接使用我们下载好的文件，如下：



3.1.3 FreeModbus 源码文件的介绍

前面我们已经获取了FreeModbus源码，下面介绍下里面的文件都有什么作用。

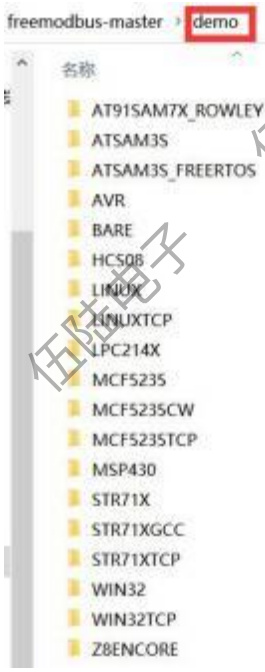
首先将下载好的源码进行解压，打开如下所示：



从上图可以看到，FreeModbus 源码主要有 demo、modbus、doc、tools 文件夹构成。

Demo 文件夹中主要是 freemodbus 官方新建好的各种平台的测试例程，以加快用户的开发进度，其中包括 Win32 平台、Linux 平台、ARM 平台等。我们需要

移植到STM32上，它也属于 ARM 平台的范畴，因此，在移植的过程中可以参考 ARM 平台已经新建好的测试工程。Demo 文件夹下具体测试平台工程如下所示：



Modbus 文件夹下，主要放一些关于Modbus 自身协议的源码，其中包括 Modbus-Rtu、Modbus-Ascii、Modbus-Tcp 等，具体如下图所示，移植的过程中，可根据实际情况的需要对该文件夹进行适当的裁剪。



至于 **doc 和 tools 文件夹**就不再赘述，doc 主要放一些帮助和说明文件，tools 是放置一些需要的工具。

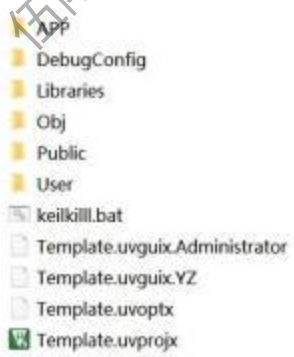
FreeModbus 文件就介绍这么多，下面开始从 0 到 1 完成在 stm32 平台上的移植。

3.2 FreeModbus 移植到 STM32

3.2.1 基础工程准备

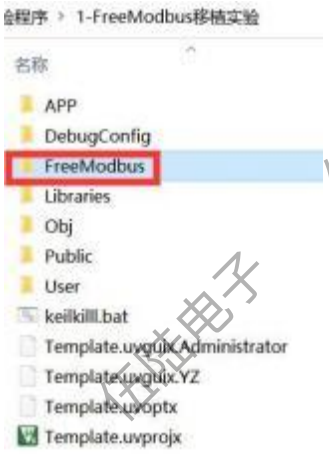
在将 FreeModbus 移植到 STM32 上之前，先要准备一个 STM32 基础工程。本章我们可从开发板配套例程中的基础实验中选择“RS485 通信实验”作为本次移植的基础工程，将其拷贝一份，重命名为“FreeModbus 移植实验”。

基础工程结构如下：



3.2.2 添加 FreeModbus 至基础工程

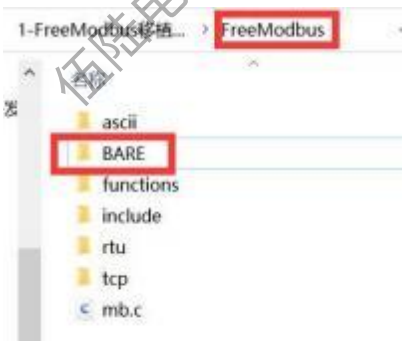
（1）在基础工程中我们新建一个文件夹命名为“FreeModbus”，如下：



（2）然后将前面下载好的 FreeModbus 源码的“..\modbus”文件夹下的所有文件及文件夹拷贝到我们新建的FreeModbus文件夹下。如下：



接着还需要将FreeModbus 源码的“..\demo”文件夹下的 BARE 文件夹拷贝到我们新建的 FreeModbus 文件夹下。如下：



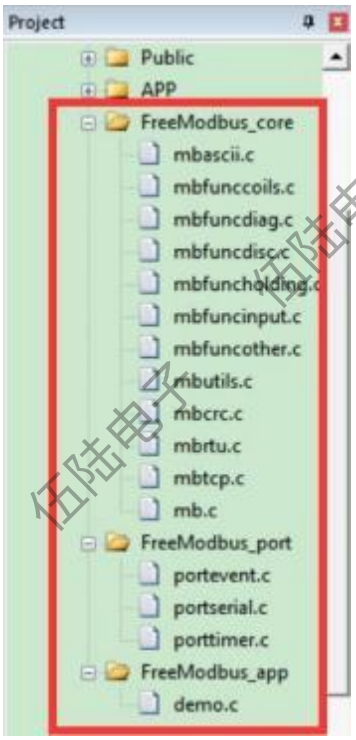
（3）使用 KEIL 打开基础工程，向工程中添加 FreeModbus_core、FreeModbus_port 和 FreeModbus_app 分组，然后将 FreeModbus 文件夹下对应的文件添加到对应的分组中。

①添加到 FreeModbus_core 分组中，主要包括：ascii 文件夹的 mbascii.c 文件；function 文件夹下的所有.c 文件；rtu 和 tcp 文件夹下的.c 文件；mb.c 文件。

②添加到 FreeModbus_port 分组中，主要包括：“\FreeModbus\BARE\port”下的.c 文件。

③添加到 FreeModbus_app 分组中，主要包括：“\FreeModbus\BARE”下的 demo.c 文件。

添加后的效果如下：



（4）添加对应的头文件路径，工程需要添加以下文件夹：ascii文件夹、BARE/port 文件夹、include 文件夹、rtu 文件夹、tcp 文件夹；总之有.h 的文件夹统统包含，具体如下：



至此，我们就将FreeModbus添加到工程中，接下来还需要对部分文件修改，以完成 Modbus 对串口和定时器的需要。

3.2.3 修改 FreeModbus 中 portserial.c 文件

前面添加进来的 FreeModbus 源码是没有任何串口和定时器的驱动，接下来

我们需要对 FreeModbus 的 portserial.c 文件添加 Modbus 串口发送中断和接收中断使能函数。

portserial.c 修改后的具体代码如下：（红色标记为添加的代码）

```
/*
 * FreeModbus Libary: BARE Port
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id$
 */

#include "port.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"


#include "system.h"

/* ----------------------- static functions ---------------------------------*/
static void prvvUARTTxReadyISR( void );
static void prvvUARTRxISR( void );

/* ----------------------- Start implementation -----------------------------*/
void
vMBPortSerialEnable( BOOL xRxEnable, BOOL xTxEnable )
{
    /* If xRXEnable enable serial receive interrupts. If xTxENable enable
     * transmitter empty interrupts.
```

```
41.         */
42.        //STM32 串口 接收中断使能
43.        if(xRxEnable==TRUE)
44.        {
45.            //使能接收和接收中断
46.            USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);
47.            GPIO_ResetBits(GPIOG,GPIO_Pin_8);//RS485 驱动芯片，接收使能
48.        }
49.        else
50.        {
51.            //禁止接收和接收中断
52.            USART_ITConfig(USART2, USART_IT_RXNE, DISABLE);
53.            GPIO_SetBits(GPIOG,GPIO_Pin_8);//RS485 驱动芯片，发送使能
54.        }
55.        //STM32 串口 发送中断使能
56.        if(xTxEnable==TRUE)
57.        {
58.            //使能发送完成中断
59.            USART_ITConfig(USART2, USART_IT_TXE, ENABLE);
60.        }
61.        else
62.        {
63.            //禁止发送完成中断
64.            USART_ITConfig(USART2, USART_IT_TXE, DISABLE);
65.        }
66.    }
67.
68.    BOOL
69.    xMBPortSerialInit( UCHAR ucPORT, ULONG ulBaudRate, UCHAR ucDataBits, eMBParity eParity )
70.    {
71.        GPIO_InitTypeDef GPIO_InitStructure;
72.        USART_InitTypeDef USART_InitStructure;
73.        NVIC_InitTypeDef NVIC_InitStructure;
74.
75.        (void)ucPORT;       //用串口 2,不修改串口
76.        (void)ucDataBits; //不修改数据位长度
77.
78.        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOG,ENABLE); //使能 GPIOA\G
    时钟
79.        RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE);//使能 USART2 时钟
80.
81.        //串口 2 引脚复用映射
82.        GPIO_PinAFConfig(GPIOA,GPIO_PinSource2,GPIO_AF_USART2); //GPIOA2 复用为 USART2
83.        GPIO_PinAFConfig(GPIOA,GPIO_PinSource3,GPIO_AF_USART2); //GPIOA3 复用为 USART2
```

```
84.
85.     //USART2
86.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3; //GPIOA2 与 GPIOA3
87.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
88.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;  //速度 100MHz
89.     GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
90.     GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
91.     GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA2，PA3
92.
93.     //PG8 推挽输出，485 模式控制
94.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8; //GPIOG8
95.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;//输出
96.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;  //速度 100MHz
97.     GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
98.     GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; //上拉
99.     GPIO_Init(GPIOG,&GPIO_InitStructure); //初始化 PG8
100.
101.    //USART2 初始化设置
102.    USART_InitStructure.USART_BaudRate = ulBaudRate;//波特率设置
103.    USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
104.     USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无硬件
数据流控制
105.     USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式
106.    switch(eParity)
107.    {
108.        case MB_PAR_NONE:
109.        {
110.            USART_InitStructure.USART_Parity =USART_Parity_No;
111.            USART_InitStructure.USART_WordLength = USART_WordLength_8b;
112.            USART_ITConfig(USART2, USART_IT_PE, DISABLE);//关闭奇偶校验错中断
113.            break;
114.        }
115.        case MB_PAR_ODD:
116.        {
117.            USART_InitStructure.USART_Parity =USART_Parity_Odd;
118.            USART_InitStructure.USART_WordLength = USART_WordLength_9b;
119.            USART_ITConfig(USART2, USART_IT_PE, ENABLE);//使能奇偶校验错中断
120.            break;
121.        }
122.        case MB_PAR_EVEN:
123.        {
124.            USART_InitStructure.USART_Parity =USART_Parity_Even;
125.            USART_InitStructure.USART_WordLength = USART_WordLength_9b;
126.            USART_ITConfig(USART2, USART_IT_PE, ENABLE);//使能奇偶校验错中断
```

```
            break;
        }
        default:break;
    }
    USART_Init(USART2, &USART_InitStructure); //初始化串口 2
    USART_Cmd(USART2, ENABLE);  //使能串口 2

    //Usart2 NVIC 配置
    NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;//抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority =0;        //子优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;          //IRQ 通道使能
    NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化 VIC 寄存器、

    return TRUE;
}

BOOL
xMBPortSerialPutByte( CHAR ucByte )
{
    /* Put a byte in the UARTs transmit buffer. This function is called
     * by the protocol stack if pxMBFrameCBTransmitterEmpty( ) has been
     * called. */
    USART_SendData(USART2, ucByte);
    while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
    return TRUE;
}

BOOL
xMBPortSerialGetByte( CHAR * pucByte )
{
    /* Return the byte in the UARTs receive buffer. This function is called
     * by the protocol stack after pxMBFrameCBByteReceived( ) has been called.
     */
    *pucByte = USART_ReceiveData(USART2);
    return TRUE;
}

/* Create an interrupt handler for the transmit buffer empty interrupt
 * (or an equivalent) for your target processor. This function should then
 * call pxMBFrameCBTransmitterEmpty( ) which tells the protocol stack that
 * a new character can be sent. The protocol stack will then call
 * xMBPortSerialPutByte( ) to send the character.
 */
```

```
171.  static void prvvUARTTxReadyISR( void )
172. {
173.     pxMBFrameCBTransmitterEmpty(  );
174. }
175.
176. /* Create an interrupt handler for the receive interrupt for your target
177.  * processor. This function should then call pxMBFrameCBByteReceived( ). The
178.  * protocol stack will then call xMBPortSerialGetByte( ) to retrieve the
179.  * character.
180.  */
181. static void prvvUARTRxISR( void )
182. {
183.     pxMBFrameCBByteReceived(  );
184. }
185.
186. void USART2_IRQHandler(void)
187. {
188.     if(USART_GetITStatus(USART2, USART_IT_PE) == SET)
189.     {
190.         eMBEnable(); //初始化接收
191.         USART_ClearITPendingBit(USART2, USART_IT_PE);
192.     }
193.     else
194.     {
195.         if(USART_GetITStatus(USART2, USART_IT_RXNE) == SET)
196.         {
197.             prvvUARTRxISR();//接受中断
198.             USART_ClearITPendingBit(USART2, USART_IT_RXNE);
199.         }
200.         if(USART_GetITStatus(USART2, USART_IT_TXE) == SET)
201.         {
202.             prvvUARTTxReadyISR();//发送完成中断
203.             USART_ClearITPendingBit(USART2, USART_IT_TXE);
204.         }
205.     }
206.     //溢出-如果发生溢出需要先读 SR,再读 DR 寄存器 则可清除不断进入中断的问题
207.     if(USART_GetFlagStatus(USART2,USART_FLAG_ORE)==SET)
208.     {
209.         USART_ClearFlag(USART2,USART_FLAG_ORE); //读 SR
210.         USART_ReceiveData(USART2);              //读 DR
211.     }
212. }
```

首先在 portserial.c 文件开头添加#include "system.h"语句，因为该头文

件内已经包含了STM32 对应的头文件，如果不添加的话，那么再添加串口等驱动时会提示错误。

所添加的代码都很简单，在学习STM32 基础实验时都详细介绍过，不清楚的回头看下我们STM32 对应的开发攻略。

3.2.4 修改 FreeModbus 中 porttimer.c 文件

接下来需要修改的是 porttimer.c 文件,需要补充的就是 Modbus 定时器初始化函数、Modbus 定时器使能和失能函数，以及Modbus 定时器中断函数。这个也是非常简单的。具体代码如下：（红色标记为添加代码）

```
/*
 * FreeModbus Libary: BARE Port
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id$
 */

/* ----------------------- Platform includes --------------------------------*/
#include "port.h"

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"


#include "system.h"
```

```
/* ----------------------- static functions ---------------------------------*/
 static void prvvTIMERExpiredISR( void );

/* ----------------------- Start implementation -----------------------------*/
BOOL
xMBPortTimersInit( USHORT usTim1Timerout50us )
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

     uint16_t PrescalerValue = 0;

      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE);//使能 TIM4 时钟

    /* Compute the prescaler value */
    PrescalerValue = (uint16_t) (SystemCoreClock / 20000) - 1; // 1/20000=50us

    TIM_TimeBaseInitStructure.TIM_Period=usTim1Timerout50us;    //自动装载值
    TIM_TimeBaseInitStructure.TIM_Prescaler=PrescalerValue; //分频系数
    TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;
    TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up; //设置向上计数模式
    TIM_TimeBaseInit(TIM4,&TIM_TimeBaseInitStructure);

     TIM_ARRPreloadConfig(TIM4, ENABLE);

    NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;//定时器中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;//抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority =3;       //子优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;         //IRQ 通道使能
    NVIC_Init(&NVIC_InitStructure);

    TIM_ClearITPendingBit(TIM4,TIM_IT_Update);
    TIM_ITConfig(TIM4, TIM_IT_Update,DISABLE);
    TIM_Cmd(TIM4,DISABLE);
    return TRUE;
}


void
vMBPortTimersEnable(   )
{
    /* Enable the timer with the timeout passed to xMBPortTimersInit( ) */
    TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
    TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
```

```
75.      TIM_SetCounter(TIM4,0x0000);
76.      TIM_Cmd(TIM4, ENABLE);
77.  }
78.
79.  void
80.  vMBPortTimersDisable(  )
81.  {
82.      /* Disable any pending timers. */
83.      TIM_Cmd(TIM4, DISABLE);
84.      TIM_SetCounter(TIM4,0x0000);
85.      TIM_ITConfig(TIM4, TIM_IT_Update, DISABLE);
86.      TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
87.  }
88.
89.  /* Create an ISR which is called whenever the timer has expired. This function
90.   * must then call pxMBPortCBTimerExpired( ) to notify the protocol stack that
91.   * the timer has expired.
92.   */
93.  static void prvvTIMERExpiredISR( void )
94.  {
95.      ( void )pxMBPortCBTimerExpired(  );
96.  }
97.
98.
99.  void TIM4_IRQHandler(void)
100. {
101.     if(TIM_GetITStatus(TIM4,TIM_IT_Update))
102.     {
103.         prvvTIMERExpiredISR();
104.     }
105.     TIM_ClearITPendingBit(TIM4,TIM_IT_Update);
106. }
```

3.2.5 修改 FreeModbus 中 port.h 文件

接下来需要修改的文件是 port.h 文件，port.h 文件中主要补充进入临界区和退出临界区的宏定义。要修改 port.h 文件，首先要编译工程，才能找到该文件，此次编译会报很多错误，先不用管它。修改代码如下：（红色标记为添加代码）

```
1.  /*
2.   * FreeModbus Libary: BARE Port
3.   * Copyright (C) 2006 Christian Walter <wolti@sil.at>
```

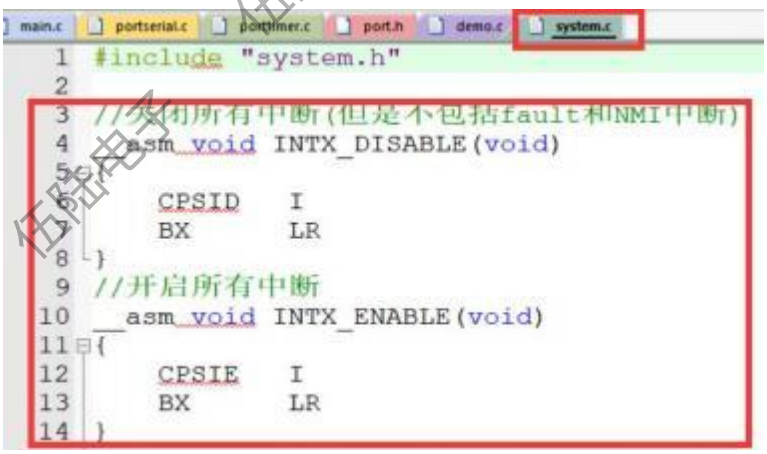
```
4.  *
5.  * This library is free software; you can redistribute it and/or
6.  * modify it under the terms of the GNU Lesser General Public
7.  * License as published by the Free Software Foundation; either
8.  * version 2.1 of the License, or (at your option) any later version.
9.  *
10. * This library is distributed in the hope that it will be useful,
11. * but WITHOUT ANY WARRANTY; without even the implied warranty of
12. * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
13. * Lesser General Public License for more details.
14. *
15. * You should have received a copy of the GNU Lesser General Public
16. * License along with this library; if not, write to the Free Software
17. * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
18. *
19. * File: $Id$
20. */
21.
22. #ifndef _PORT_H
23. #define _PORT_H
24.
25. #include <assert.h>
26. #include <inttypes.h>
27.
28. #include "system.h"
29.
30. #define INLINE                      inline
31. #define PR_BEGIN_EXTERN_C           extern "C" {
32. #define PR_END_EXTERN_C             }
33.
34. #define ENTER_CRITICAL_SECTION( )   INTX_DISABLE() //关闭中断
35. #define EXIT_CRITICAL_SECTION( )    INTX_ENABLE() //开启中断
36.
37. typedef uint8_t BOOL;
38.
39. typedef unsigned char UCHAR;
40. typedef char CHAR;
41.
42. typedef uint16_t USHORT;
43. typedef int16_t SHORT;
44.
45. typedef uint32_t ULONG;
46. typedef int32_t LONG;
47.
```

```
#ifndef TRUE
#define TRUE            1
#endif

#ifndef FALSE
#define FALSE           0
#endif

#endif
```

添加代码中 INTX_ENABLE()和 INTX_DISABLE()函数为中断开关功能，其函数实现在 system.c 中。如下：



当然我们在 port.h 文件中还可以定义自己的宏，比如串口、定时器需要修改部分等等，这样做的好处是方便任意串口定时器的移植，只需修改 port.h 文件即可实现全部修改。这种编程思想在学习STM32 基础实验时已经讲解过。这里为了简单，我们就不定义。

至此，FreeModbus 所需的移植代码就修改完成，接下来编译下，会发现依然有报错，我们直接定位错误之处，发现是在 porttimer.c 文件的定时器使能和失能函数位置，我们尝试删除 vMBPortTimersEnable 函数和 vMBPortTimersDisable 函数前的 inline 字样，结果编译之后没有该错误。

但编译后依然还存在关于__aeabi_assert 的错误，这是因为工程中缺少该函数的定义，可以在main.c 文件最后添加如下代码解决：

```
#ifdef  USE_FULL_ASSERT
/*******************************************************************************
  * @函数名称    assert_failed
  * @函数说明    报告在检查参数发生错误时的源文件名和错误行数
  * @输入参数    file：源文件名
                 line：错误所在行数
```

```
 * @输出参数    无
 * @返回参数    无
 *******************************************************************************/
void assert_failed(uint8_t* file, uint32_t line)
{
    /* 用户可以增加自己的代码用于报告错误的文件名和所在行数,
       例如：printf("错误参数值： 文件名 %s 在 %d 行\r\n", file, line) */

    /* 无限循环 */
    while (1)
    {
    }
}

#else
void __aeabi_assert(const char * x1, const char * x2, int x3)
{
}
#endif
```

通过上述一系列修改，最终编译工程没有错误，存在警告是因为定义的某些变量或函数没有使用导致，这个不用管它。

至此，FreeModbus移植基本上已经完成，我们还需要添加相应的测试代码来验证 FreeModbus 是否移植成功。

3.2.6 修改 FreeModbus 中 demo.c 文件

前面移植时，我们在 FreeModbus_app 中添加了一个 demo.c 文件，该文件是从官方源码案列中拷贝而来。也就是说，该文件本身就是一个Modbus测试程序，其内含有 Modbus 输入寄存器功能测试。为了使Modbus测试功能更全，我们还添加了输入寄存器、保持寄存器、线圈的起始地址和定义的内容，以及输入寄存器、保持寄存器、线圈的处理函数。代码如下：

```
/*
 * FreeModbus Libary: BARE Demo Application
 * Copyright (C) 2006 Christian Walter <wolti@sil.at>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
```

```
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 * File: $Id$
 */

/* ----------------------- Modbus includes ----------------------------------*/
#include "mb.h"
#include "mbport.h"


/*---------------------------------------------------------------------
名称:           线圈
存储区标识:     0XXXX
类型:           位
读写:           读/写
存储单元地址:  (00001~0XXXX).XXXX:与设备相关
功能码:         读取线圈状态 (0x01)，写单个线圈 (0x05)，写多个线圈 (0x0F)
用途:           DO 继电器
//---------------------------------------------------------------------*/
#define REG_COILS_START     0x0001
#define REG_COILS_SIZE      16
unsigned char        ucRegCoilsBuf[REG_COILS_SIZE / 8]={0X00,0X00};

/*---------------------------------------------------------------------
名称:           输入线圈
存储区标识:     1XXXX
类型:           位
读写:           只读
存储单元地址:  (10001~1XXXX).XXXX:与设备相关
功能码:         读输入状态 (0x02)
用途:           DI 继电器
//---------------------------------------------------------------------*/
/* ----------------------- Defines ------------------------------------------*/
#define REG_DISC_START     0x0001
#define REG_DISC_SIZE      16
```

```
53.  unsigned char ucRegDiscBuf[REG_DISC_SIZE / 8] = { 0x0F, 0XF0 };
54.
55.  /*-----------------------------------------------------------------
56.  名称：          输入寄存器
57.  存储区标识：    3XXXX
58.  类型：          字
59.  读写：          只读
60.  存储单元地址：(30001~3XXXX).XXXX:与设备相关
61.  功能码：        读输入寄存器 (0x04)
62.  用途：          模拟量输入
63.  //-----------------------------------------------------------------*/
64.  #define REG_INPUT_START 0x0001
65.  #define REG_INPUT_NREGS 8
66.  static USHORT   usRegInputStart = REG_INPUT_START;
67.  static USHORT   usRegInputBuf[REG_INPUT_NREGS]=
68.  {0x1111,0x2222,0x3333,0x4444,0x5555,0x6666,0x7777,0x8888};
69.
70.
71.  /*-----------------------------------------------------------------
72.  名称：          保持/输出寄存器
73.  存储区标识：    4XXXX
74.  类型：          字
75.  读写：          读/写
76.  存储单元地址：(40001~4XXXX).XXXX:与设备相关
77.  功能码：        读保持寄存器 (0x03)，写单个寄存器 (0x06)，写多个寄存器 (0x10)
78.  用途：          模拟量输出
79.  //-----------------------------------------------------------------*/
80.  #define REG_HOLDING_START 0x0001 //保持寄存器起始地址
81.  #define REG_HOLDING_NREGS 8    //保持寄存器数量
82.  USHORT   usRegHoldingStart = REG_HOLDING_START;
83.  USHORT   usRegHoldingBuf[REG_HOLDING_NREGS]=
84.  {0X1100,0X2200,0X3300,0X4400,0X5500,0X6600,0X7700,0X8800};
85.
86.
87.  /*=================================================================
88.  function name: eMBRegInputCB()
89.  Description:   0x04 输入寄存器处理函数，将数据放入 BUF
90.  Attention:
91.  Input Parameters:
92.  Returncode:
93.  =================================================================*/
94.  eMBErrorCode
95.  eMBRegInputCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNRegs )
96.  {
```

```
    eMBErrorCode    eStatus = MB_ENOERR;
    int             iRegIndex;

    if( ( usAddress >= REG_INPUT_START )
        && ( usAddress + usNRegs <= REG_INPUT_START + REG_INPUT_NREGS ) )
    {
        iRegIndex = ( int )( usAddress - usRegInputStart );
        while( usNRegs > 0 )
        {
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] >> 8 );
            *pucRegBuffer++ =
                ( unsigned char )( usRegInputBuf[iRegIndex] & 0xFF );
            iRegIndex++;
            usNRegs--;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }

    return eStatus;
}

/*============================================================================
 function name: eMBRegHoldingCB()
Description:   0x03，0x06.0x10 保持寄存器处理函数，将数据放入 BUF
Attention:
 Input Parameters:
Returncode:
============================================================================*/
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 )
```

```
        {
        case MB_REG_READ:
            while( usNRegs > 0 )
            {
                 *pucRegBuffer++ = ( unsigned char )( usRegHoldingBuf[iRegIndex] >> 8 );
                 *pucRegBuffer++ = ( unsigned char )( usRegHoldingBuf[iRegIndex] & 0xFF );
                iRegIndex++;

                usNRegs--;
            }
            break;

        case MB_REG_WRITE:
            while( usNRegs > 0 )
            {
                usRegHoldingBuf[iRegIndex] = *pucRegBuffer++ << 8;
                usRegHoldingBuf[iRegIndex] |= *pucRegBuffer++;
                iRegIndex++;
                usNRegs--;
            }
            break;
        }
    }
    else
    {
        eStatus = MB_ENOREG;
    }
    return eStatus;
}

/*============================================================================
function name: eMBRegCoilsCB()
Description:  0x01 ,0x05,0x0F 线圈处理函数，将数据放入 BUF
Attention: 0x01 读输出（DO），0x05,0x0f 强制单个，多个线圈（DO）
Input Parameters:
Returncode:
============================================================================*/
eMBErrorCode
eMBRegCoilsCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNCoils,
               eMBRegisterMode eMode )
{
    eMBErrorCode    eStatus = MB_ENOERR;

    int             iNCoils = ( int )usNCoils;
```

```
184.     unsigned short   usBitOffset;
185. 
186.     /* Check if we have registers mapped at this block. */
187.     if( ( usAddress >= REG_COILS_START ) &&( usAddress + usNCoils <= REG_COILS_START + REG_COILS_SIZE ) )
188.     {
189.         usBitOffset = ( unsigned short )( usAddress - REG_COILS_START );
190.         switch ( eMode )
191.         {
192.                 /* Read current values and pass to protocol stack. */
193.             case MB_REG_READ:
194.                 while( iNCoils > 0 )
195.                 {
196.                     *pucRegBuffer++ = xMBUtilGetBits( ucRegCoilsBuf, usBitOffset, ( unsigned char )( iNCoils > 8 ? 8 : iNCoils ) );
197.                     iNCoils -= 8;
198.                     usBitOffset += 8;
199.                 }
200.                 break;
201. 
202.                 /* Update current register values. */
203.             case MB_REG_WRITE:
204.                 while( iNCoils > 0 )
205.                 {
206.                     xMBUtilSetBits( ucRegCoilsBuf, usBitOffset,( unsigned char )( iNCoils > 8 ? 8 : iNCoils ), *pucRegBuffer++ );
207.                     iNCoils -= 8;
208.                     usBitOffset += 8;
209.                 }
210.                 break;
211.         }
212. 
213.     }
214.     else
215.     {
216.         eStatus = MB_ENOREG;
217.     }
218.     return eStatus;
219. }
220. 
221. /*============================================================================
222. function name: eMBRegDiscreteCB()
223. Description:  0x02 读输入状态/读取输入状态处理函数，将数据放入 BUF
224. Attention:
```

```
225.  Input Parameters:
226.  Returncode:
227.  ==========================================================================*/
228.  eMBErrorCode
229.  eMBRegDiscreteCB( UCHAR * pucRegBuffer, USHORT usAddress, USHORT usNDiscrete )
230.  {
231.      eMBErrorCode    eStatus = MB_ENOERR;
232.      short           iNDiscrete = ( short )usNDiscrete;
233.      unsigned short  usBitOffset;
234.
235.      /* Check if we have registers mapped at this block. */
236.      if( (usAddress >= REG_DISC_START) &&( usAddress + usNDiscrete<= REG_DISC_START + REG_DI
    SC_SIZE ))
237.      {
238.          usBitOffset = (unsigned short)( usAddress - REG_DISC_START );
239.          while( iNDiscrete > 0 )
240.          {
241.              *pucRegBuffer++ =xMBUtilGetBits(ucRegDiscBuf, usBitOffset, (unsigned char)( iND
    iscrete >8?8: iNDiscrete ) );
242.              iNDiscrete -= 8;
243.              usBitOffset+=8;
244.          }
245.      }
246.      else
247.      {
248.          eStatus = MB_ENOREG;//illegal register address
249.      }
250.      return eStatus;
251.  }
```

3.2.7 修改 main.c 文件

最后我们要让 STM32 运行FreeModbus，还需要在main 中调用相应的初始化函数，main.c 代码如下：（红色标记为新添加代码）

```
1.  #include "system.h"
2.  #include "SysTick.h"
3.  #include "led.h"
4.  #include "usart.h"
5.  #include "key.h"
6.  #include "mb.h"
7.
8.
9.  /**************************************************************************
```

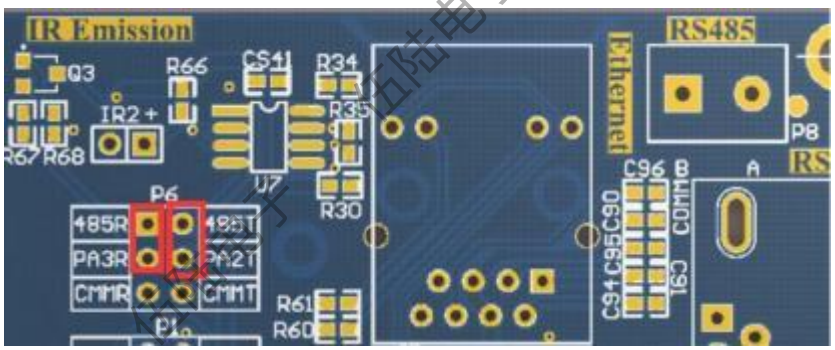
```
10.  * 函 数 名          : main
11.  * 函数功能          : 主函数
12.  * 输    入          : 无
13.  * 输    出          : 无
14.  
15.  int main()
16.  {
17.      u8 i=0;
18.  
19.      SysTick_Init(168);
20.      NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);   //中断优先级分组 分 2 组
21.      LED_Init();
22.      USART1_Init(115200);
23.      KEY_Init();
24.  
25.      //初始化 FreeModbus
26.      //参数 1：Modbus 模式：这里可以选择 RTU、ASCII、TCP，我们选择 RTU 模式(MB_RTU)
27.      //参数 2：设备地址：这里可以直接写死(0x01)，也可以通过拨码开关来设置
28.      //参数 3：Modbus 选择的串口号：这里默认是串口 2(0x02)
29.      //参数 4：波特率：默认写 115200，我们这里配置为 9600
30.      //参数 5：校验位：默认不效验(MB_PAR_NONE)
31.      eMBInit(MB_RTU,0x01,0x02,9600,MB_PAR_NONE);
32.      eMBEnable();//启动 FreeModbus
33.      while(1)
34.      {
35.          //FreeMODBUS 不断查询
36.          eMBPoll();
37.  
38.          i++;
39.          if(i%20==0)
40.          {
41.              LED1=!LED1;
42.          }
43.          delay_ms(10);
44.      }
45.  }
46.  
47.  
48.  #ifdef  USE_FULL_ASSERT
49.  /*****************************************************************************
50.    * @函数名称    assert_failed
51.    * @函数说明    报告在检查参数发生错误时的源文件名和错误行数
52.    * @输入参数    file：源文件名
53.                   line：错误所在行数
```

```
54.     * @输出参数    无
55.     * @返回参数    无
56.     *******************************************************************************/
57.   void assert_failed(uint8_t* file, uint32_t line)
58.   {
59.       /* 用户可以增加自己的代码用于报告错误的文件名和所在行数,
60.          例如：printf("错误参数值 : 文件名 %s 在 %d 行\r\n", file, line) */
61.
62.       /* 无限循环 */
63.       while (1)
64.       {
65.       }
66.   }
67.
68.   #else
69.   void __aeabi_assert(const char * x1, const char * x2, int x3)
70.   {
71.   }
72.   #endif
```

将工程编译无错误。至此，FreeModbus移植就完成，我们需要将其下载到STM32 开发板上运行，验证 FreeModbus 是否运行正常。

3.3 实验现象

程序编译无错误后，将其下载到开发板内运行。因为本实验的Modbus 是基于 RS485 的，所以要将开发板相应的黄色跳线帽调到RS485 串口有效。这里以麒麟为例：（伍陆电子 STM32 开发板在对应的开发攻略内 RS485 实验章节中实验现象
都有说明）



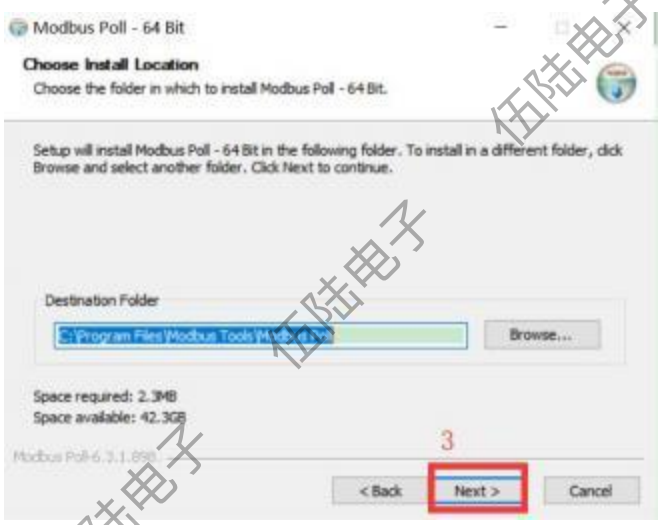
3.3.1 ModbusPoll 工具的安装和使用

因为移植的FreeModbus 只能作为从机使用，还需要借助一个非常好用的工具 ModbusPoll 来调试。本实验使用的是 Modbus RTU 模式。ModbusPoll 工具我们已经提供，如下：

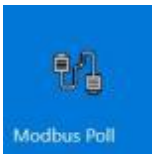


将其解压后，选择对应的系统版本双击安装即可，安装过程非常简单，直接下一步即可。如下：





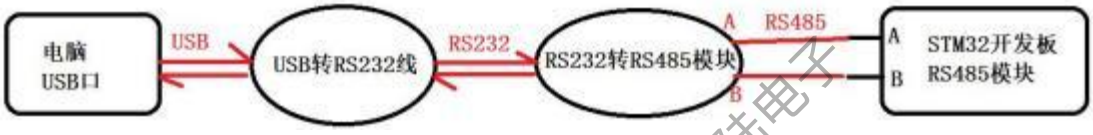
安装成功后，桌面会有一个如下图标：



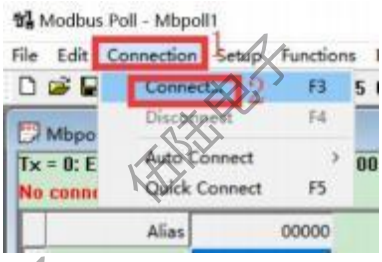
双击该图标，打开 ModbusPo11 工具。如下：



现在我们要将从机与 PC 机连接，即使用一个 RS232 转 RS485 模块和一条 USB 转 RS232 线。RS232 转 RS485 模块的 RS485 端与开发板的 485 的 A、B 一一对应连接，不能交叉。RS232 转 RS485 模块的 RS232 端与 USB 转 RS232 线连接，该线另一端与电脑 USB 口连接，连接示意图如下：



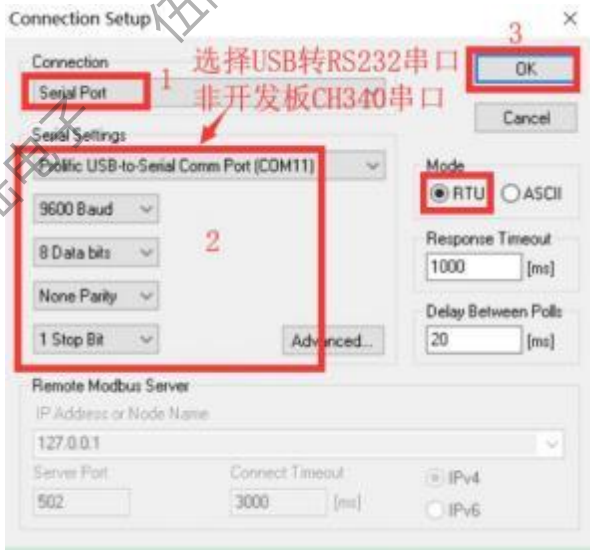
硬件线路连接好后，开始对ModbusPo11 软件设置。操作如下：



因为软件没有破解，所以只有 30 天试用，点击 OK 即可。



弹出如下对话框，开始设置串口参数，没有选择的保持默认。如下：

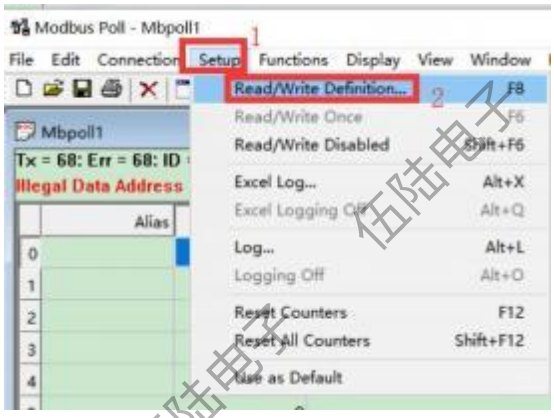


点击 OK 后，画面显示无效的数据地址，这是因为我们还没有配置对应的寄存器地址和数据量。如下：



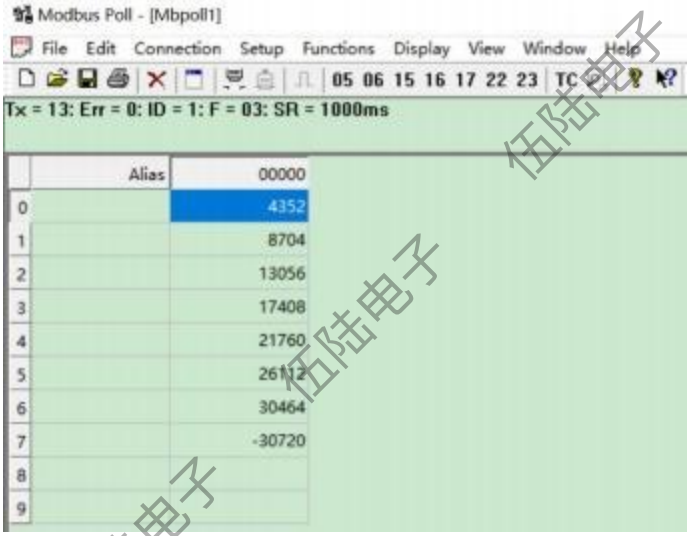
3.3.2 Modbus 功能测试

（1）首先来测试下读保持寄存器 03H 功能，我们点击 Setup，操作如下：

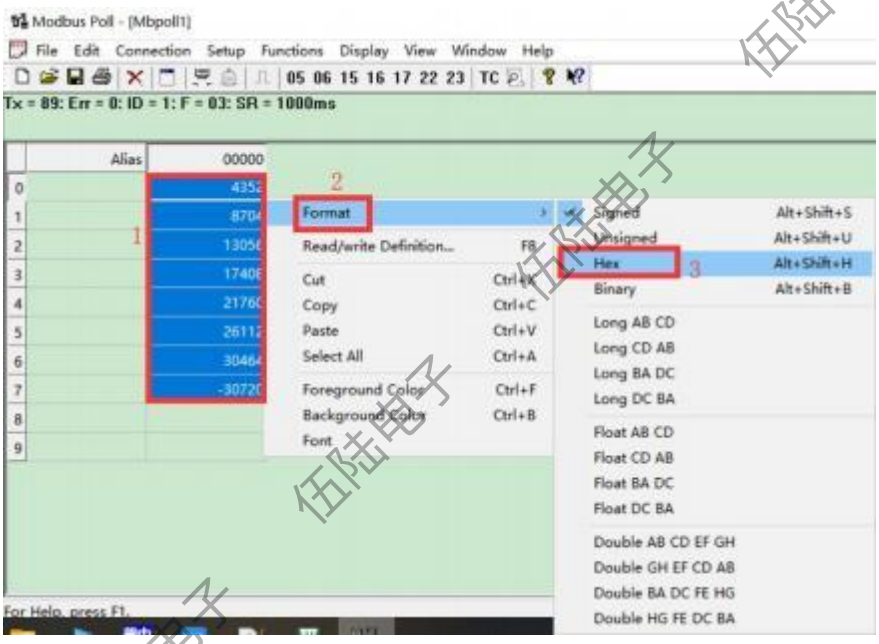


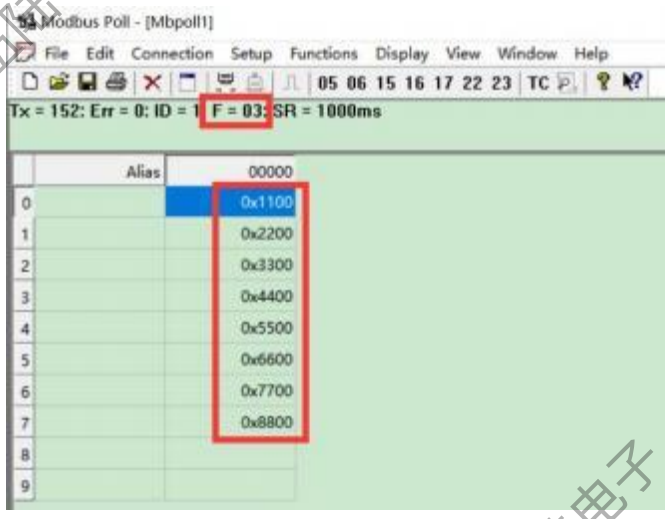


因为程序中配置的 03H 保持寄存器数量为8，所以要对应修改。点击 OK 后，软件现象如下：



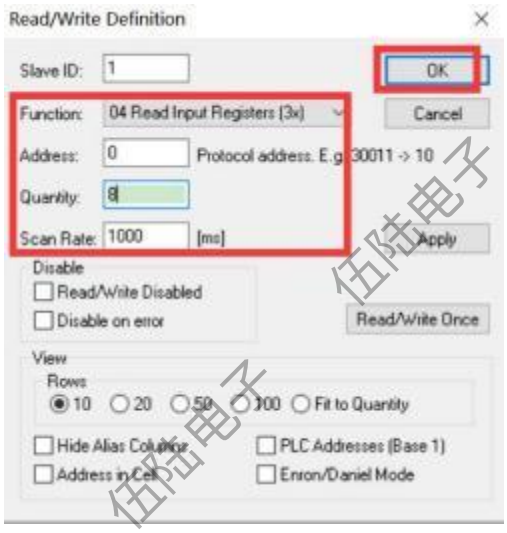
咋一看，这些数据和我们写入寄存器的数据不一样，因为默认该软件是以 10 进制数显示，而我们写入的是 16 进制数，所以在软件中设置下显示方式即可。选中要查看的数据，鼠标右键，选中格式 HEX 即可。如下：



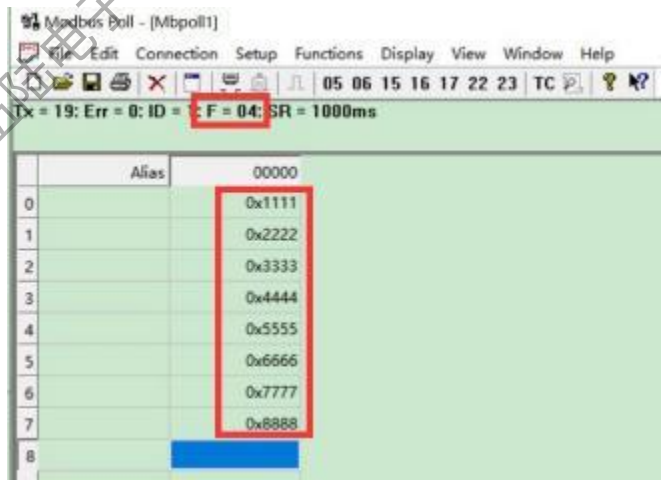


读取与我们写入是一致的，说明FreeModbus 移植正确。

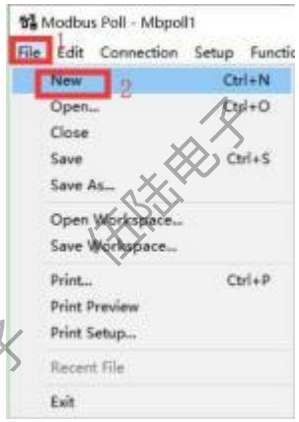
（2）然后还可以对输入寄存器 04H 功能测试，方法与03H 操作类似，如下：



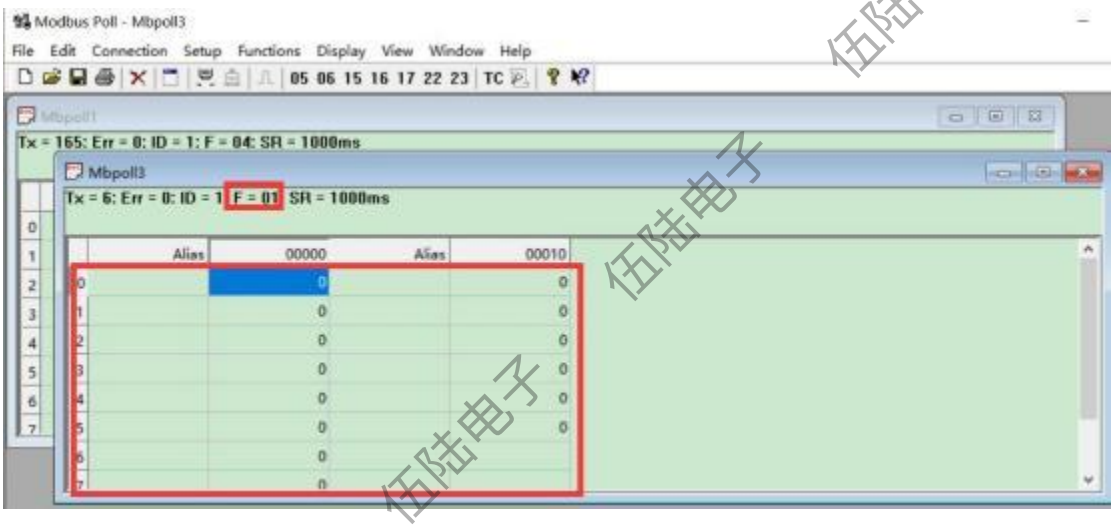
点击 OK 后显示现象如下：



（3）还可以新建一个工作台对读线圈寄存器 01H 功能测试，方法与前面操作类似，如下：



此时会弹出一个新的监视窗口，按照前面同样的方法，设置好功能指令和数据量大小，最终显示效果如下：



至此，我们对移植的FreeModbus就验证完成，大家可以根据移植的测试功能，测试其他功能。

课后作业