

安富莱 STM32 开发板 MODBUS 教程







版本：V0.5

日期：2016-01-11



适用的开发板型号：

- STM32-V4 (STM32F103ZET6)
- STM32-V5 (STM32F407IGT6)
- STM32-V6 (STM32F429BIT6)

配套的例程：

-  V4-RS485 MODBUS从站例程(V1.0).rar
-  V4-RS485 MODBUS主站例程(V1.0).rar
-  V5-RS485 MODBUS从站例程(V1.4).rar
-  V5-RS485 MODBUS主站例程(V1.4).rar
-  V6-RS485 MODBUS从站例程(V1.0).rar
-  V6-RS485 MODBUS主站例程(V1.0).rar

硬件环境：

- V4，V5，V6 开发板任意一款，使用 RS485 接口。
- 使用两个 STM32 主板，一个运行主站程序，一个运行从站程序。通过 RS232 串口观察运行结果。
 - 如果只有一个板子，可以使用我们开发的 PC 机调试软件进行试验。需要 PC 机配一个 USB 转 RS485 转换器。
- PC 软件下载地址：<http://bbs.armfly.com/read.php?tid=14967>
 -  MODBUS调试助手（V1.1）.rar (1128 K) 下载次数:344
 -  MODBUS虚拟设备2015_09_10（V1.1）.rar (1246 K) 下载次数:416
- USB-RS485 转换器
购买地址：https://item.taobao.com/item.htm?_u=b8teau726a7&id=522606958735

第1章 MODBUS 协议介绍

1.1 MODBUS 标准简介

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》

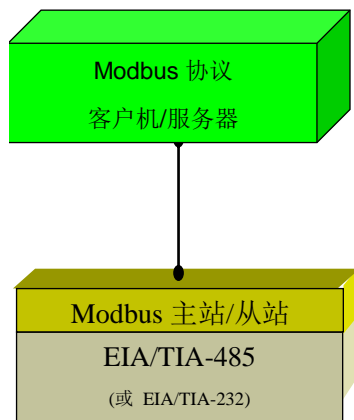
本教程仅涉及第 1 部分和第 2 部分, 串行链路仅介绍 RS485 网络。

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

1.2 MODBUS 协议概述

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。

1.3 Modbus 主站/从站协议原理

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。

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。

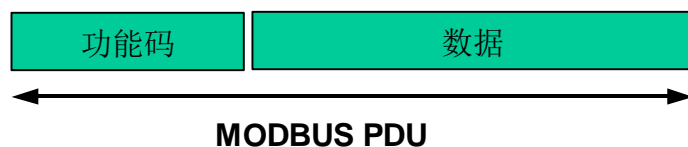
0	1 ~ 247	248 ~ 55
广播地址	子节点单独地址	保留

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。

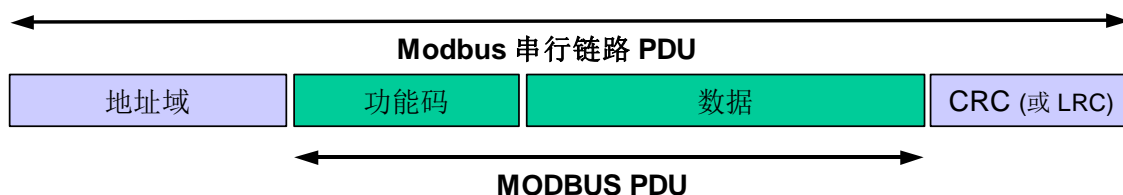
1.4 Modbus 帧描述

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



Modbus 协议数据单元

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。



串行链路上的 Modbus 帧

- 在 Modbus 串行链路, 地址域只含有子节点地址。
如前文所述, 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。

1.5 RTU 传输模式

有两种串行传输模式被定义: **RTU** 模式和 **ASCII** 模式。

它定义了报文域的位内容在线路上串行的传送。它确定了信息如何打包为报文和解码。

Modbus 串行链路上所有设备的传输模式 (和串行口参数) 必须相同。

尽管在特定的领域 ASCII 模式是要求的，但达到 Modbus 设备之间的互操作性只有每个设备都有相同的模式：所有设备必须实现 RTU 模式。**ASCII 传输模式是选项。**

当设备使用 RTU (Remote Terminal Unit) 模式在 Modbus 串行链路通信，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。**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)



RTU 模式位序列

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：



RTU 模式位序列 (无校验的特殊情况)

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

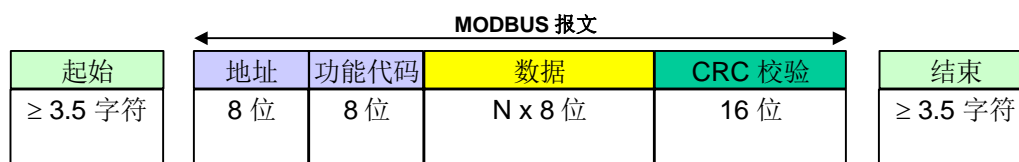
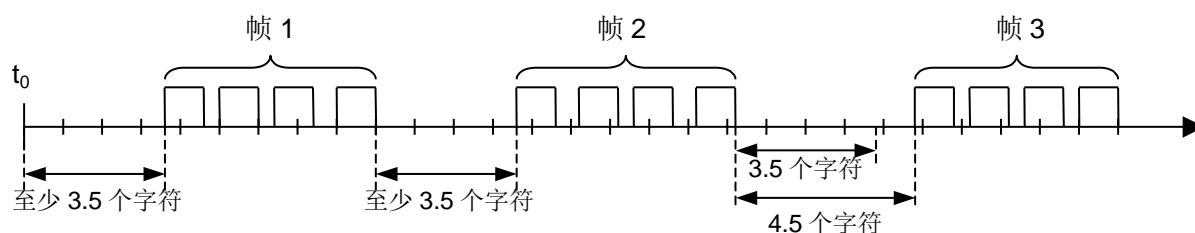
RTU 报文帧

Modbus RTU 帧总长度最大为 256 字节。

1.5.1 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。

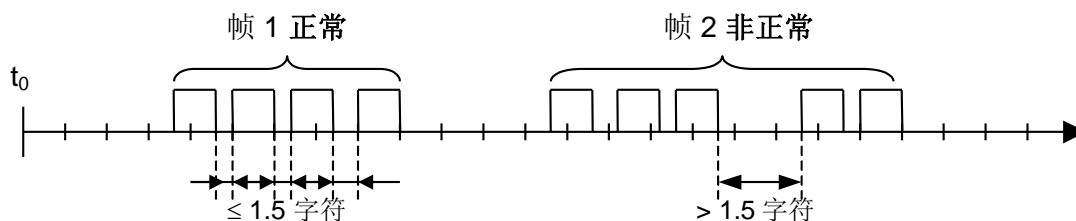
在 RTU 模式，**报文帧由时长至少为 3.5 个字符时间的空闲间隔区分**。在后续的部分，这个时间区间被称作 t3.5。



RTU 报文帧

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。



注：

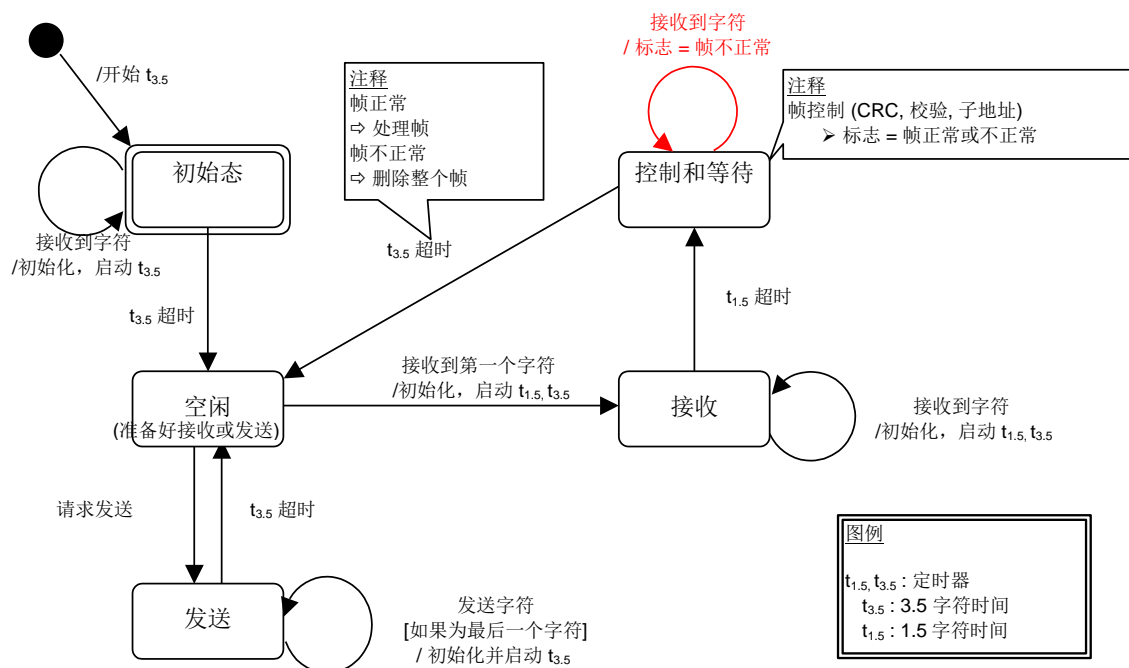
RTU 接收驱动程序的实现，由于 t1.5 和 t3.5 的定时，隐含着大量的对中断的管理。在高通信速

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($t_{1.5}$)为 750 μ s，

帧间的超时时间 ($t_{3.5}$) 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 $t_{3.5}$ 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隔达到 $t_{3.5}$ 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。

1.5.2 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 Redundancy

Checking) 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 位子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。

1.6 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 位子节以两个 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

Bits per Byte:

- 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

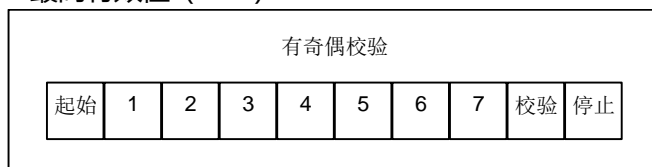
偶校验是要求的，其它模式（奇校验，无校验）也可以使用。为了保证与其它产品的最大兼容性，同时支持无校验模式是建议的。默认校验模式必须为偶校验。

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)



ASCII 模式位序列

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：



ASCII 模式模式位序列(无校验的特殊情况)

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

1.6.1 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 0D 和 0A) 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。

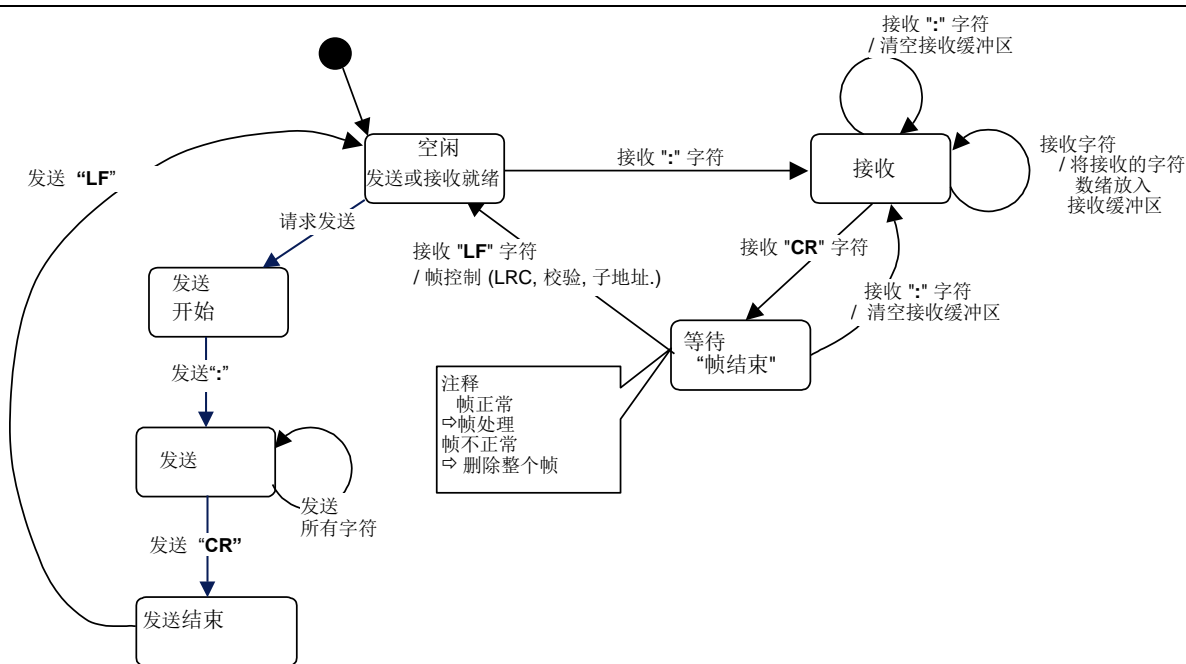
起始	地址	功能	数据	LRC	结束
1 字符 :	2 字符	2 字符	0 到 to 2x252 字符	2 字符	2 字符 CR,LF

图 1: ASCII 报文帧

注：每个字符子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。

1.6.2 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。



1.7 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。

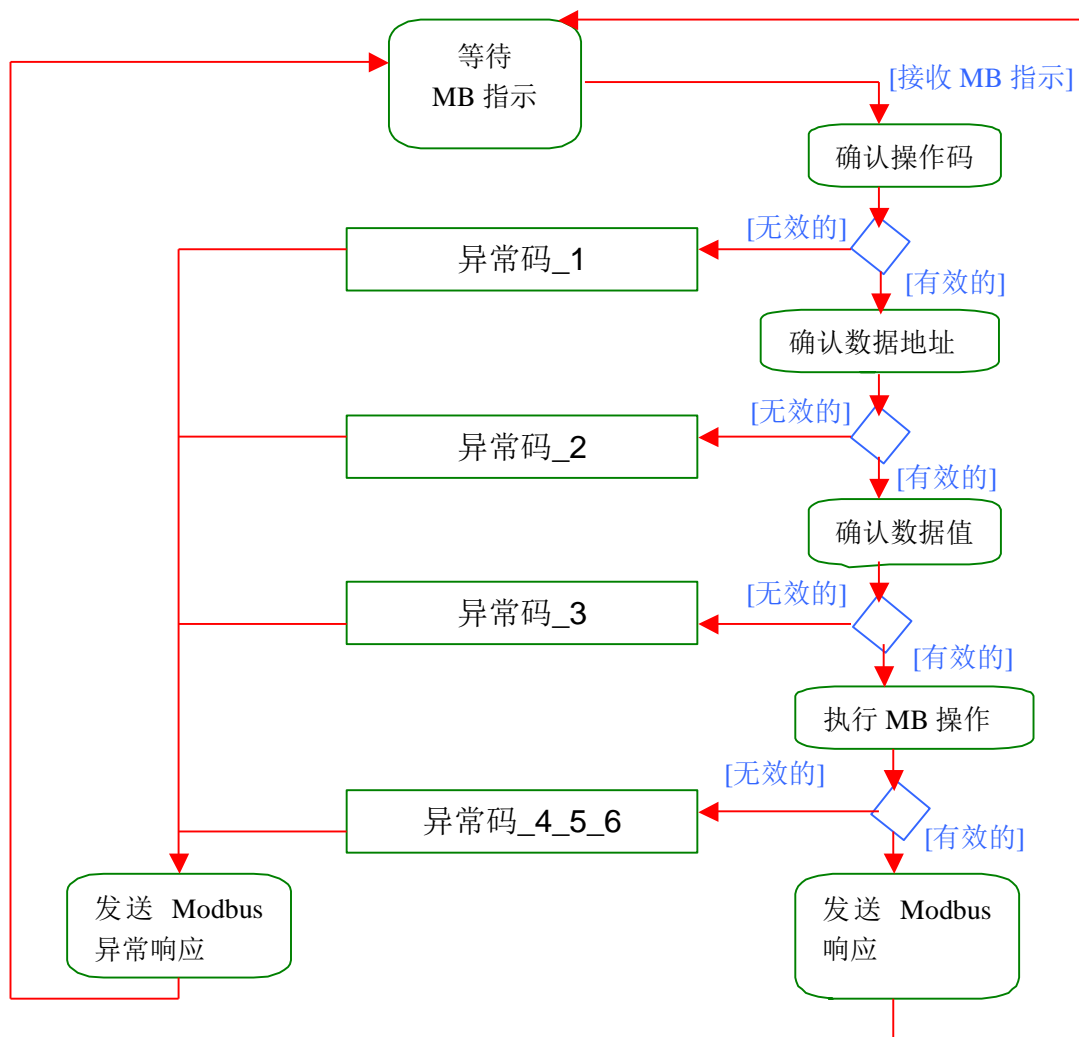


图 8：MODBUS 事务处理的状态图

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：

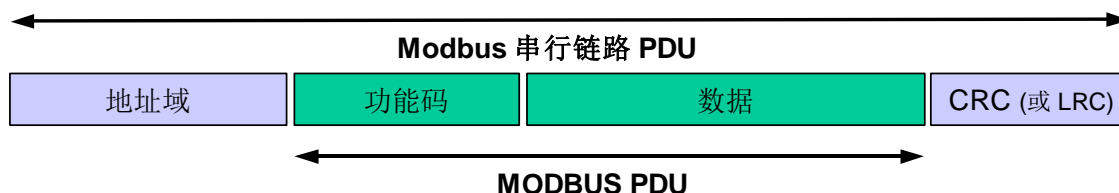
- 一个正确的 MODBUS 响应：响应功能码 = 请求功能码
- 一个 MODBUS 异常响应
 - 用来为客户机提供处理过程中与被发现的差错相关的信息；
 - 响应功能码 = 请求功能码 + 0x80；
 - 提供一个异常码来指示差错原因。

1.8 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，象输出域(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。

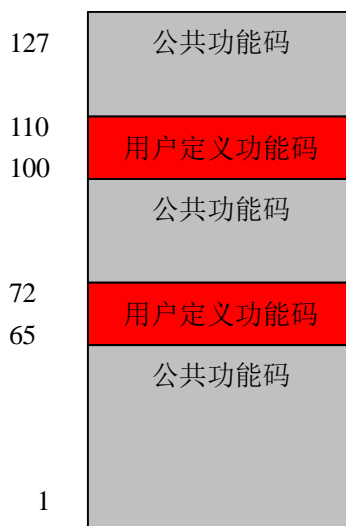
下面的不常用		
05	确认	与编程命令一起使用。服务器(或从站)已经接受请求,并切正在处理这个请求,但是需要长的持续时间进行这些操作。返回这个响应防止在客户机(或主站)中发生超时错误。客户机(或主站)可以继续发送轮询程序完成报文来确定是否完成处理。
06	从属设备忙	与编程命令一起使用。服务器(或从站)正在处理长持续时间的程序命令。当服务器(或从站)空闲时,用户(或主站)应该稍后重新传输报文。
08	存储奇偶性差错	与功能码 20 和 21 以及参考类型 6 一起使用,指示扩展文件区不能通过一致性校验。 服务器(或从站)设法读取记录文件,但是在存储器中发现一个奇偶校验错误。客户机(或主站)可以重新发送请求,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。

1.9 MODBUS 数据模型

第2章 功能码定义

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：

				功能码			
				码	子码	(十六进制)	页
数据访问	比特访问	物理离散量输入	读输入离散量	02		02	11
		内部比特或物理线圈	读线圈	01		01	10
			写单个线圈	05		05	16
			写多个线圈	15		0F	37
	16 比特访问	输入存储器	读输入寄存器	04		04	14
		内部存储器或物理输出存储器	读多个寄存器	03		03	13
			写单个寄存器	06		06	17
			写多个寄存器	16		10	39
			读/写多个寄存器	23		17	47
			屏蔽写寄存器	22		16	46
		文件记录访问	读文件记录	20	6	14	42
	写文件记录		21	6	15	44	
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 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。

表 1.1 读线圈寄存器—查询

	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，则在最后数据字节末尾使用 0 代替。

表 1.2 读线圈寄存器—响应

	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 11001101，该字节的最高字节为线圈 001AH，最低字节为线圈 0013H。线圈 001AH 到线圈 0013H 的状态分别为 ON-ON-OFF-OFF-ON-ON-OFF-ON

表 1.3 线圈 0013H 到 001A 状态

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 00011011)，线圈 0037H

是左数第 4 位，线圈 0033H 为该字节的最低字节，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。

表 1.4 线圈 0033H 到线圈 0037 状态

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

2.4 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。

表 2.1 读离散输入寄存器——查询

	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。

表 2.2 读输入寄存器——响应

	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 00110101)。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。

表 2.3 离散输入寄存器 00C4H 到 00DBH 状态

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

2.5 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。

表 3.1 读保持寄存器-查询

	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。

表 3.2 读保持寄存器-响应

	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

表 3.3 保持寄存器 006BH 到 006DH 结果

--	--	--	--	--	--

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

2.6 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。本次访问访问 2 个输入寄存器。

表 4.1 读输入寄存器-查询

	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。

表 4.2 读寄存器-响应

	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

表 4.3 输入寄存器 0008H 到 0009H 结果

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

2.7 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。

表 5.1 写单个线圈-查询

--	--

	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

5.2 强制单个线圈——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

2.8 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。

表 6.1 写单个保持寄存器——查询

	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

表 6.2 写单个保持寄存器——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

2.9 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。保持寄存器的其实地址为 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。

表 7.1 写多个保持寄存器——请求

	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

表 7.2 保持寄存器 0001H 到 0002H 内容

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

表 7.3 写多个保持寄存器——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

2.10 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。

表 8.1 线圈寄存器 0013H 到 001CH

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。

表 8.2 写多个线圈寄存器——查询

	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

表 8.3 写多个线圈寄存器——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

第3章 MODBUS 主站例程

3.1 例程简介

本例程为 RS485 MODBUS 主站例程。可以通过按键发送相应的 MODBUS 命令，然后等待从站应答（或超时）退出等待。

程序使用串口 3（USART1）与从机通讯，执行结果通过串口 1（USART1）送到计算机的串口。可以通过 PC 机的串口终端软件观察程序执行结果。

3.2 实验说明

本例程只介绍常用的 modbus 命令：01H 02H 03H 04H 05H 06H 10H。

RS485 MODBUS 主站例程（使用的是串口 3）。

1. 本例子为主站例程，需要通过按键发送相应的命令，等待从站应答（或超时）。
2. 另外所有开发板的 485-A 端子连接到一起，485-B 端子连接到一起，具体连接看工程 Doc 文件夹中的截图。

3.3 实验内容

3.3.1 操作方法

本机按键发送相应的命令，通过 RS485 发送到从机，然后等待应答（或超时）。串口 1 看到发送和接收的命令。本实验可以主从站联机操作，也可以使用 PC 软件 MODBUS 虚拟设备作为从机。按键命令如下图：

读单个或多个寄存器

按键	从机地址	功能码	寄存器首地址	寄存器数量	校验码
KEY_DOWN_K1	01	01	01 01	00 04	6D F5
KEY_DOWN_K2	01	03	03 01	00 02	95 8F
JOY_DOWN_OK	01	02	02 01	00 03	68 73

JOY_UP_OK	01	04	04 01	00 01	61 3A
-----------	----	----	-------	-------	-------

图 1-3-1

写单个寄存器

按键	从机地址	功能码	寄存器首地址	写入值	校验码
JOY_DOWN_U	01	06	03 01	00 01	19 8E
JOY_DOWN_D	01	06	03 01	00 00	D8 4E
JOY_DOWN_L	01	05	01 01	00 01	5C 36
JOY_DOWN_R	01	05	01 01	00 00	9D F6

图 1-3-2

写多个寄存器

按键	从地址	功能码	地址	数量	字节数	写入值 1	写入值 2	校验码
KEY_DOWN_K3	01	10	03 01	00 02	04	00 01	02 03	36 32

图 1-3-3

3.3.2 执行结果

请用 USB 转串口线连接 PC 机和开发板。PC 机上运行 SecureCRT 软件 ,波特率设置为 115200bps ,无硬件流控。按下 KEY_DOWN_K1 ,发送 01H 命令。从 PC 机的软件界面观察程序执行结果 ,结果如下 :

```

* 例程名称   : V5-RS485 MODBUS 主站例程
* 例程版本   : 1.4
* 发布日期   : 2015-11-28
* 固件库版本 : V1.3.0 (STM32F4xx_StdPeriph_Driver)
*
* QQ        : 1295744630
* 旺旺       : armfly
* Email      : armfly@qq.com
    
```

* 淘宝店: armfly.taobao.com

* Copyright www.armfly.com 安富莱电子

发送的命令 : 0x 01 01 01 01 00 04 6D F5

接收的命令 : 0x 01 01 01 01 90 48

3.4 例程讲解

3.4.1 主函数

```
/*
*****
*   函 数 名: main
*   功能说明: c 程序入口
*   形    参: 无
*   返 回 值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;           /* 按键代码 */
    MSG_T ucMsg;                 /* 消息代码 */

    bsp_Init();                  /* 硬件初始化 */
    PrintfLogo();                /* 打印例程信息到串口 1 */
    DispMenu();                  /* 打印寄存器的值 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();              /* 调用 MODH_Poll() */

        if (bsp_GetMsg(&ucMsg)) /* 读取消息代码 */
        {
            switch (ucMsg.MsgCode)
            {
                case MSG_MODS:
                    DispMenu();    /* 打印实验结果 */
                    break;

                default:

```



```

        break;
    }
}

/* 按键滤波和检测由后台 systick 中断服务程序实现，我们只需要调用 bsp_GetKey 读取键值即可。 */
ucKeyCode = bsp_GetKey();          /* 读取键值，无键按下时返回 KEY_NONE = 0 */
if (ucKeyCode != KEY_NONE)
{
    bsp_PutMsg(MSG_MODS, 0);

    switch (ucKeyCode)
    {
        case KEY_DOWN_K1:          /* K1 键按下 */
            if (MODH_ReadParam_01H(REG_D01, 4) == 1) ;else ;
            break;

        case KEY_DOWN_K2:          /* K2 键按下 */
            if (MODH_ReadParam_03H(REG_P01, 2) == 1) ;else ;
            break;

        case KEY_DOWN_K3:          /* K3 键按下 */
            {
                uint8_t buf[4];

                buf[0] = 0x01;
                buf[1] = 0x02;
                buf[2] = 0x03;
                buf[3] = 0x04;
                if (MODH_WriteParam_10H(REG_P01, 2, buf) == 1) ;else ;
            }
            break;

        case JOY_DOWN_U:           /* 摇杆 UP 键弹起 */
            if (MODH_WriteParam_06H(REG_P01, 1) == 1) ;else ;
            break;

        case JOY_DOWN_D:           /* 摇杆 DOWN 键按下 */
            if (MODH_WriteParam_06H(REG_P01, 0) == 1) ;else ;
            break;

        case JOY_DOWN_L:           /* 摇杆 LEFT 键弹起 */
            if (MODH_WriteParam_05H(REG_D01, 1) == 1) ;else ;
            break;
    }
}

```

```

        case JOY_DOWN_R:                /* 摇杆 RIGHT 键弹起 */
            if (MODH_WriteParam_05H(REG_D01, 0) == 1) ;else ;
            break;

        case JOY_DOWN_OK:                /* 摇杆 OK 键按下 */
            if (MODH_ReadParam_02H(REG_T01, 3) == 1) ;else ;
            break;

        case JOY_UP_OK:                  /* 摇杆 OK 键弹起 */
            if (MODH_ReadParam_04H(REG_A01, 1) == 1) ;else ;
            break;

        default:
            /* 其它的键值不处理 */
            break;
    }

```

主函数其实非常简单，主要就是按键发送命令。前面说到的等待从站应答（或超时）退出等待，其实是在按键发送命令后，程序就停在那里等待应答或超时退出，如：if (MODH_ReadParam_01H(REG_D01, 4) == 1) ;else ;。实际上是让主机处理完一条命令才能处理下一条命令，这样做的目的是为了防止串口同时处理多条命令，造成数据混乱。下面会详细介绍 MODBUS 相关的函数。

3.4.2 Modbus_host.c 文件

下面我将会对 MODBUS 主要函数进行讲解。

```

/*
*****
*   函 数 名: MODH_Send01H
*   功能说明: 发送 01H 指令，查询 1 个或多个保持寄存器
*   形    参: _addr : 从站地址
*             _reg : 寄存器编号
*             _num : 寄存器个数
*   返 回 值: 无
*****
*/
void MODH_Send01H(uint8_t _addr, uint16_t _reg, uint16_t _num)
{
    g_tModH.TxCount = 0;
    g_tModH.TxBuf[g_tModH.TxCount++] = _addr;        /* 从站地址 */
    g_tModH.TxBuf[g_tModH.TxCount++] = 0x01;         /* 功能码 */
    g_tModH.TxBuf[g_tModH.TxCount++] = _reg >> 8;    /* 寄存器编号 高字节 */

```

```
g_tModH.TxBuf[g_tModH.TxCount++] = _reg;      /* 寄存器编号 低字节 */
g_tModH.TxBuf[g_tModH.TxCount++] = _num >> 8; /* 寄存器个数 高字节 */
g_tModH.TxBuf[g_tModH.TxCount++] = _num;      /* 寄存器个数 低字节 */

MODH_SendAckWithCRC();      /* 发送数据，自动加 CRC */
g_tModH.fAck01H = 0;        /* 清接收标志 */
g_tModH.RegNum = _num;      /* 寄存器个数 */
g_tModH.Reg01H = _reg;      /* 保存 03H 指令中的寄存器地址，方便对应答数据进行分类 */
}
```

MODH_Send01H() 为发送命令函数，需要注意的是，g_tModH.fAck01H=0 和 g_tModH.Reg01H=reg 这两个变量非常重要，下面进行讲解。

```
/*
*****
* 函数名: MODH_ReadParam_01H
* 功能说明: 单个参数. 通过发送 01H 指令实现, 发送之后, 等待从机应答.
* 形参: 无
* 返回值: 1 表示成功. 0 表示失败 (通信超时或被拒绝)
*****
*/
uint8_t MODH_ReadParam_01H(uint16_t _reg, uint16_t _num)
{
    int32_t time1;
    uint8_t i;

    for (i = 0; i < NUM; i++)      /* 循环处理, 在实际运用中可能需要多次发送命令. 默认 NUM=1 */
    {
        MODH_Send01H (SlaveAddr, _reg, _num);      /* 发送命令 */
        time1 = bsp_GetRunTime();      /* 记录命令发送的时刻 */

        while (1)      /* 等待应答, 超时或接收到应答则 break */
        {
            bsp_Idle();      /* 调用 MODH_Poll() 解析 MODBUS 命令函数 */

            if (bsp_CheckRunTime(time1) > TIMEOUT)
            {
                break;      /* 通信超时了 */
            }

            if (g_tModH.fAck01H > 0)
            {
                break;      /* 接收到应答 */
            }
        }
    }
}
```

```

    }

    if (g_tModH.fAck01H > 0)
    {
        break;                /* 循环 NUM 次，如果接收到命令则 break 循环 */
    }
}

if (g_tModH.fAck01H == 0)
{
    return 0;                /* 01H 读失败 */
}
else
{
    return 1;                /* 01H 读成功 */
}
}

```

函数每一步的功能注释里已经讲的非常详细。g_tModH.fAck01H = 1 表示成功接收到应答，g_tModH.fAck01H = 0 表示接收失败。该标志实际上是在 MODH_Read_01H()函数设置的。这就是 MODH_Poll()函数的功能了，MODH_Poll()为主站处理 MODBUS 命令的函数，所用命令都是在该函数中解析。实际上本例程中 bsp_Idle()函数唯一的工作就是调用 MODH_Poll()函数。

```

/*
*****
*   函 数 名: bsp_Idle
*   功能说明: 空闲时执行的函数。一般主程序在 for 和 while 循环程序体中需要插入 CPU_IDLE() 宏来调用本函数。
*           本函数缺省为空操作。用户可以添加喂狗、设置 CPU 进入休眠模式的功能。
*   形    参: 无
*   返 回 值: 无
*****
*/
void bsp_Idle(void)
{
    MODH_Poll();            /* 485 modbus 主机 */
}

```

```

/*
*****
*   函 数 名: MODH_Poll
*   功能说明: 接收控制器指令。1ms 响应时间。

```

```
* 形 参: 无
* 返 回 值: 0 表示无数据 1 表示收到正确命令
*****
*/
void MODH_Poll(void)
{
    uint16_t crc1;

    if (g_modh_timeout == 0) /* 超过 3.5 个字符时间后执行 MODH_RxTimeOut() 函数。全局变量 g_rtu_timeout = 1 */
    {
        /* 没有超时，继续接收。不要清零 g_tModH.RxCount */
        return ;
    }

    g_modh_timeout = 0; /* 清标志 */

    if (g_tModH.RxCount < 4) /* 接收到的数据小于 4 个字节就认为错误 */
    {
        goto err_ret;
    }

    /* 计算 CRC 校验和 */
    crc1 = CRC16_Modbus(g_tModH.RxBuf, g_tModH.RxCount);
    if (crc1 != 0)
    {
        goto err_ret;
    }

    /* 分析应用层协议 */
    MODH_AnalyzeApp();

err_ret:
#ifdef 1 /* 此部分为了串口打印结果, 实际运用中可不要 */
    g_tPrint.Rxlen = g_tModH.RxCount;
    memcpy(g_tPrint.RxBuf, g_tModH.RxBuf, g_tModH.RxCount);
#endif

    g_tModH.RxCount = 0; /* 必须清零计数器, 方便下次帧同步 */
}
```

当接收到应答命令。函数 MODH_Poll()判断完数据长度和校验位后,就执行 MODH_AnalyzeApp()分析应用层协议。

```
/*
*****
*   函 数 名: MODH_AnalyzeApp
*   功能说明: 分析应用层协议。处理应答。
*   形    参: 无
*   返 回 值: 无
*****
*/
static void MODH_AnalyzeApp(void)
{
    switch (g_tModH.RxBuf[1])                /* 第 2 个字节 功能码 */
    {
        case 0x01:                          /* 读取线圈状态 */
            MODH_Read_01H();
            break;

        case 0x02:                          /* 读取输入状态 */
            MODH_Read_02H();
            break;

        case 0x03:                          /* 读取保持寄存器 在一个或多个保持寄存器中取得当前的二进制值 */
            MODH_Read_03H();
            break;

        case 0x04:                          /* 读取输入寄存器 */
            MODH_Read_04H();
            break;

        case 0x05:                          /* 强制单线圈 */
            MODH_Read_05H();
            break;

        case 0x06:                          /* 写单个保持寄存器 */
            MODH_Read_06H();
            break;

        case 0x10:                          /* 写多个保持寄存器 */
            MODH_Read_10H();
            break;

        default:
            break;
    }
}
```

MODH_AnalyzeApp()函数将应答命令分类，然后再分开处理，比如处理 01H 命令，下面是实现的代码：

```
/*
*****
*   函 数 名: MODH_Read_01H
*   功能说明: 分析 01H 指令的应答数据
*   形    参: 无
*   返 回 值: 无
*****
*/
static void MODH_Read_01H(void)
{
    uint8_t bytes;
    uint8_t *p;

    if (g_tModH.RxCount > 0)
    {
        bytes = g_tModH.RxBuf[2];          /* 数据长度 字节数 */
        switch (g_tModH.Reg01H)           /* g_tModH.Reg01H 为寄存器首地址，发送命令时将其赋值 */
        {
            case REG_D01:
                if (bytes == 8)
                {
                    p = &g_tModH.RxBuf[3]; /* 数据起始地址 */

                    g_tVar.D01 = BEBufToUint16(p); p += 2; /* 寄存器 D01 */
                    g_tVar.D02 = BEBufToUint16(p); p += 2; /* 寄存器 D02 */
                    g_tVar.D03 = BEBufToUint16(p); p += 2; /* 寄存器 D03 */
                    g_tVar.D04 = BEBufToUint16(p); p += 2; /* 寄存器 D04 */

                    g_tModH.fAck01H = 1;
                }
                break;
        }
    }
}
```

MODH_Read_01H()函数就将接收到的应答命令的值存在寄存器中，并设置 g_tModH.fAck01H = 1。至此 01H 指令从发送，到接收应答，再到解析应答分析完毕。其他命令也都一样。

第4章 MODBUS 从站例程

4.1 例程简介

从站在接收到主站发来的 MODBUS 指令后会做出响应,并将接收命令和应答命令通过串口打印出来。程序执行结果通过串口 1 (USART1) 送到计算机的串口。可以通过 PC 机的串口终端软件观察程序执行结果。

4.2 实验说明

本例程支持的 MODBUS 命令： 01H 02H 03H 04H 05H 06H 10H。

RS485 MODBUS 从站例程 (使用的是串口 3)。

- 本例子为从站例程,需要通过主站或者 MODBUS 调试助手发送命令,本机则响应命令。
- 另外所有开发板的 485-A 端子连接到一起,485-B 端子连接到一起,具体连接看工程 Doc 文件夹中的截图。

4.3 实验内容

4.3.1 操作方法

通过 MODBUS 调试助手或者 MODBUS 主站发送命令到本机。串口 1 会将接收到的命令和应答命令打印到 SecureCRT 软件上。

4.3.2 执行结果

请用 USB 转串口线连接 PC 机和开发板。PC 机上运行 SecureCRT 软件,波特率设置为 115200bps,无硬件流控。通过 RS485 向从站发送 03 命令,从 PC 机的软件界面观察程序执行结果,结果如下:

* 例程名称 : V5-RS485 MODBUS从站例程
* 例程版本 : 1.4
* 发布日期 : 2015-11-28
* 固件库版本 : V1.3.0 (STM32F4xx_StdPeriph_Driver)


```
*
* QQ    : 1295744630
* 旺旺  : armfly
* Email : armfly@qq.com
* 淘宝店: armfly.taobao.com
* Copyright www.armfly.com 安富莱电子
*****

P01 = 65535
P02 = 11111
A01 = 604

接收的命令 : 0x 01 03 03 01 00 02 95 8F
发送的命令 : 0x 01 03 04 FF FF 2B 67 A5 0D
```

4.4 例程讲解

4.4.1 主函数

主函数非常精简，就是处理消息，并打印输出结果。需要注意的是，05H 命令单独处理，是因为 05H 为强制单线圈，例程中将 05H 相关的 4 个寄存器分别与开发板的 4 个 LED 相关联。也就是说，05H 命令可以看到 LED 的亮灭。而从站 MODBUS 函数是在 bsp_Idle()中调用的。

```
/*
*****
*   函 数 名: main
*   功能说明: c 程序入口
*   形    参: 无
*   返 回 值: 错误代码(无需处理)
*****
*/
int main(void)
{
    MSG_T ucMsg;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo();         /* 打印例程信息到串口 1 */
    DispMenu();           /* 打印寄存器的值 */
}
```

```
/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();                /* 调用 MODS_Poll() */

    if (bsp_GetMsg(&ucMsg))
    {
        switch (ucMsg.MsgCode)
        {
            case MSG_MODS_05H:    /* 打印 发送的命令 和 应答的命令 刷新 LED 状态 */
                DispMenu();
                SetLed();          /* 设置 LED 亮灭(处理 05H 指令) */
                break;

            default:
                DispMenu();
                break;
        }
    }
}
```

```
/*
*****
*   函 数 名: bsp_Idle
*   功能说明: 空闲时执行的函数。一般主程序在 for 和 while 循环程序体中需要插入 CPU_IDLE() 宏来调用本函数。
*               本函数缺省为空操作。用户可以添加喂狗、设置 CPU 进入休眠模式的功能。
*   形    参: 无
*   返 回 值: 无
*****
*/
void bsp_Idle(void)
{
    MODS_Poll();                /* 从站 MODBUS 函数 */
}
```

4.4.2 Modbus_slave.c 文件

```
/*
*****
*   函 数 名: MODS_Poll
```

```

* 功能说明：解析数据包，在主程序中轮流调用。
* 形 参：无
* 返回值：无
*****
*/
void MODS_Poll(void)
{
    uint16_t addr;
    uint16_t crc1;
    /* 超过 3.5 个字符时间后执行 MODH_RxTimeOut() 函数。全局变量 g_rtu_timeout = 1; 通知主程序开始解码 */
    if (g_mods_timeout == 0)
    {
        return; /* 没有超时，继续接收。不要清零 g_tModS.RxCount */
    }

    g_mods_timeout = 0; /* 清标志 */

    if (g_tModS.RxCount < 4) /* 接收到的数据小于 4 个字节就认为错误 */
    {
        goto err_ret;
    }

    /* 计算 CRC 校验和 */
    crc1 = CRC16_Modbus(g_tModS.RxBuf, g_tModS.RxCount);
    if (crc1 != 0)
    {
        goto err_ret;
    }

    /* 站地址 (1 字节) */
    addr = g_tModS.RxBuf[0]; /* 第 1 字节 站号 */
    if (addr != SADDR485) /* 判断主机发送的命令地址是否符合 */
    {
        goto err_ret;
    }

    /* 分析应用层协议 */
    MODS_AnalyzeApp();

err_ret:
#ifdef 1 /* 此部分为了串口打印结果,实际运用中可不要 */
    g_tPrint.Rxlen = g_tModS.RxCount;
    memcpy(g_tPrint.RxBuf, g_tModS.RxBuf, g_tModS.RxCount);
#endif
#endif

```

```
g_tModS.RxCount = 0;          /* 必须清零计数器，方便下次帧同步 */
}
```

MODS_Poll()函数和主站的 MODH_Poll()函数基本一样，只是多了地址门限判断。引文有了地址的判断，才能支持 1 个主机与多个从机之间通讯。

```
/*
*****
*   函 数 名: MODS_AnalyzeApp
*   功能说明: 分析应用层协议
*   形    参: 无
*   返 回 值: 无
*****
*/
static void MODS_AnalyzeApp(void)
{
    switch (g_tModS.RxBuf[1])          /* 第 2 个字节 功能码 */
    {
        case 0x01:                    /* 读取线圈状态（此例程用 led 代替）*/
            MODS_01H();
            bsp_PutMsg(MSG_MODS_01H, 0); /* 发送消息,主程序处理 */
            break;

        case 0x02:                    /* 读取输入状态（按键状态）*/
            MODS_02H();
            bsp_PutMsg(MSG_MODS_02H, 0);
            break;

        case 0x03:                    /* 读取保持寄存器（此例程存在 g_tVar 中）*/
            MODS_03H();
            bsp_PutMsg(MSG_MODS_03H, 0);
            break;

        case 0x04:                    /* 读取输入寄存器（ADC 的值）*/
            MODS_04H();
            bsp_PutMsg(MSG_MODS_04H, 0);
            break;

        case 0x05:                    /* 强制单线圈（设置 led）*/
            MODS_05H();
            bsp_PutMsg(MSG_MODS_05H, 0);
            break;
    }
}
```

```

case 0x06:                                /* 写单个保存寄存器（此例程改写 g_tVar 中的参数）*/
    MODS_06H();
    bsp_PutMsg(MSG_MODS_06H, 0);
    break;

case 0x10:                                /* 写多个保存寄存器（此例程存在 g_tVar 中的参数）*/
    MODS_10H();
    bsp_PutMsg(MSG_MODS_10H, 0);
    break;

default:
    g_tModS.RspCode = RSP_ERR_CMD;
    MODS_SendAckErr(g_tModS.RspCode);    /* 告诉主机命令错误 */
    break;
}
}

```

MODS_AnalyzeApp()函数也是命令分类。每个分支多了发送消息函数是为了通知主机打印结果。实际运用中可不需要该部分。

命令应答函数都非常相似，这里选 MODS_01H()函数来讲解。函数如下：

```

/*
*****
*   函 数 名: MODS_01H
*   功能说明: 读取线圈状态（对应远程开关 D01/D02/D03）
*   形    参: 无
*   返 回 值: 无
*****
*/
/* 说明:这里用 LED 代替继电器, 便于观察现象 */
static void MODS_01H(void)
{
    /*
    举例:
        主机发送:
            11 从机地址
            01 功能码
            00 寄存器起始地址高字节
            13 寄存器起始地址低字节
            00 寄存器数量高字节
            25 寄存器数量低字节
            0E CRC 校验高字节
    */

```

84 CRC 校验低字节

从机应答： 1 代表 ON，0 代表 OFF。若返回的线圈数不为 8 的倍数，则在最后数据字节末尾使用 0 代替。BIT0 对应第 1 个

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

例子:

01 01 10 01 00 03 29 0B --- 查询 D01 开始的 3 个继电器状态
01 01 10 03 00 01 09 0A --- 查询 D03 继电器的状态

*/

```
uint16_t reg;
uint16_t num;
uint16_t i;
uint16_t m;
uint8_t status[10];
```

```
g_tModS.RspCode = RSP_OK;
```

```
if (g_tModS.RxCount != 8) /* 01H 命令必须为 8 字节 */
{
    g_tModS.RspCode = RSP_ERR_VALUE; /* 数据值域错误 */
    return;
}
```

```
reg = BEBufToUint16(&g_tModS.RxBuf[2]); /* 寄存器号 */
num = BEBufToUint16(&g_tModS.RxBuf[4]); /* 寄存器个数 */
```

```
m = (num + 7) / 8; /* 1 个字节的 8 位表示 8 个寄存器, 不满时高位置 0 */
```

```
if ((reg >= REG_D01) && (num > 0) && (reg + num <= REG_DXX + 1))
{
    for (i = 0; i < m; i++)
    {
        status[i] = 0;
    }
}
```

```

    for (i = 0; i < num; i++)
    {
        if (bsp_IsLedOn(i + 1 + reg - REG_D01))          /* 读 LED 的状态，写入状态寄存器的每一位 */
        {
            status[i / 8] |= (1 << (i % 8));
        }
    }
}

else
{
    g_tModS.RspCode = RSP_ERR_REG_ADDR;                  /* 寄存器地址错误 */
}

if (g_tModS.RspCode == RSP_OK)                          /* 正确应答 */
{
    g_tModS.TxCount = 0;
    g_tModS.TxBuf[g_tModS.TxCount++] = g_tModS.RxBuf[0];
    g_tModS.TxBuf[g_tModS.TxCount++] = g_tModS.RxBuf[1];
    g_tModS.TxBuf[g_tModS.TxCount++] = m;               /* 返回字节数 */

    for (i = 0; i < m; i++)
    {
        g_tModS.TxBuf[g_tModS.TxCount++] = status[i];   /* 继电器状态 */
    }

    MODS_SendWithCRC(g_tModS.TxBuf, g_tModS.TxCount);    /* 发送正确应答 */
}

else
{
    MODS_SendAckErr(g_tModS.RspCode);                   /* 告诉主机命令错误 */
}
}

```

函数中每一步的注释都非常详细。这里是读 LED 的状态，直接调用 bsp_IsLedOn() 函数。如果是读写保存寄存器的值，如下：

```

/*
*****
*   函 数 名: MODS_03H
*   功能说明: 读取保持寄存器 在一个或多个保持寄存器中取得当前的二进制值
*   形    参: 无
*   返 回 值: 无
*****
*/
static void MODS_03H(void)

```

```
{
    uint16_t reg;
    uint16_t num;
    uint16_t i;
    uint8_t reg_value[64];

    g_tModS.RspCode = RSP_OK;

    if (g_tModS.RxCount != 8)                /* 03H 命令必须是 8 个字节 */
    {
        g_tModS.RspCode = RSP_ERR_VALUE;      /* 数据值域错误 */
        goto err_ret;
    }

    reg = BEBufToUint16(&g_tModS.RxBuf[2]);    /* 寄存器号 */
    num = BEBufToUint16(&g_tModS.RxBuf[4]);    /* 寄存器个数 */
    if (num > sizeof(reg_value) / 2)
    {
        g_tModS.RspCode = RSP_ERR_VALUE;      /* 数据值域错误 */
        goto err_ret;
    }

    for (i = 0; i < num; i++)
    {
        if (MODS_ReadRegValue(reg, &reg_value[2 * i]) == 0) /* 读出寄存器值放入 reg_value */
        {
            g_tModS.RspCode = RSP_ERR_REG_ADDR; /* 寄存器地址错误 */
            break;
        }
        reg++;
    }

err_ret:
    if (g_tModS.RspCode == RSP_OK)            /* 正确应答 */
    {
        g_tModS.TxCount = 0;
        g_tModS.TxBuf[g_tModS.TxCount++] = g_tModS.RxBuf[0];
        g_tModS.TxBuf[g_tModS.TxCount++] = g_tModS.RxBuf[1];
        g_tModS.TxBuf[g_tModS.TxCount++] = num * 2; /* 返回字节数 */

        for (i = 0; i < num; i++)
        {
            g_tModS.TxBuf[g_tModS.TxCount++] = reg_value[2*i];
            g_tModS.TxBuf[g_tModS.TxCount++] = reg_value[2*i+1];
        }
    }
}
```



```

    }
    MODS_SendWithCRC(g_tModS.TxBuf, g_tModS.TxCount);    /* 发送正确应答 */
}
else
{
    MODS_SendAckErr(g_tModS.RspCode);                    /* 发送错误应答 */
}
}

```

```

/*
*****
*   函 数 名: MODS_06H
*   功能说明: 写单个寄存器
*   形    参: 无
*   返 回 值: 无
*****
*/
static void MODS_06H(void)
{
    uint16_t reg;
    uint16_t value;

    g_tModS.RspCode = RSP_OK;

    if (g_tModS.RxCount != 8)
    {
        g_tModS.RspCode = RSP_ERR_VALUE;    /* 数据值域错误 */
        goto err_ret;
    }

    reg = BEBufToUint16(&g_tModS.RxBuf[2]);    /* 寄存器号 */
    value = BEBufToUint16(&g_tModS.RxBuf[4]);    /* 寄存器值 */

    if (MODS_WriteRegValue(reg, value) == 1)    /* 该函数会把写入的值存入寄存器 */
    {
        ;
    }
    else
    {
        g_tModS.RspCode = RSP_ERR_REG_ADDR;    /* 寄存器地址错误 */
    }

err_ret:
    if (g_tModS.RspCode == RSP_OK)    /* 正确应答 */
    {
        MODS_SendAckOk();
    }
    else
    {
        MODS_SendAckErr(g_tModS.RspCode);    /* 告诉主机命令错误 */
    }
}

```

03H 和 06H 分别为读写保持寄存器。03H 指令调用了 MODS_ReadRegValue()函数,将保持寄存器中的值存入 reg_value 缓冲区,将 reg_value 中的值、地址、功能码等,重新组合成应答指令发送给主站。06H 指令调用了 MODS_WriteRegValue()函数,将接收到的主站的指令写入保持寄存器,应答指令跟接收指令相同。代码如下:

```
/*
*****
*   函 数 名: MODS_ReadRegValue
*   功能说明: 读取保持寄存器的值
*   形    参: reg_addr 寄存器地址
*             reg_value 存放寄存器结果
*   返 回 值: 1 表示 OK 0 表示错误
*****
*/
static uint8_t MODS_ReadRegValue(uint16_t reg_addr, uint8_t *reg_value)
{
    uint16_t value;

    switch (reg_addr)                                /* 判断寄存器地址 */
    {
        case SLAVE_REG_P01:
            value = g_tVar.P01;
            break;

        case SLAVE_REG_P02:
            value = g_tVar.P02;                        /* 将寄存器值读出 */
            break;

        default:
            return 0;                                /* 参数异常, 返回 0 */
    }

    reg_value[0] = value >> 8;
    reg_value[1] = value;

    return 1;                                        /* 读取成功 */
}
```

```
/*
*****
*   函 数 名: MODS_WriteRegValue
*   功能说明: 写入保持寄存器
*   形    参: reg_addr 寄存器地址
*             reg_value 寄存器值
*   返 回 值: 1 表示 OK 0 表示错误
*****
*/
static uint8_t MODS_WriteRegValue(uint16_t reg_addr, uint16_t reg_value)
{
    switch (reg_addr)                                /* 判断寄存器地址 */
    {
        case SLAVE_REG_P01:
            g_tVar.P01 = reg_value;                    /* 将值写入保存寄存器 */
            return 1;
    }
    return 0;
}
```

```

        break;

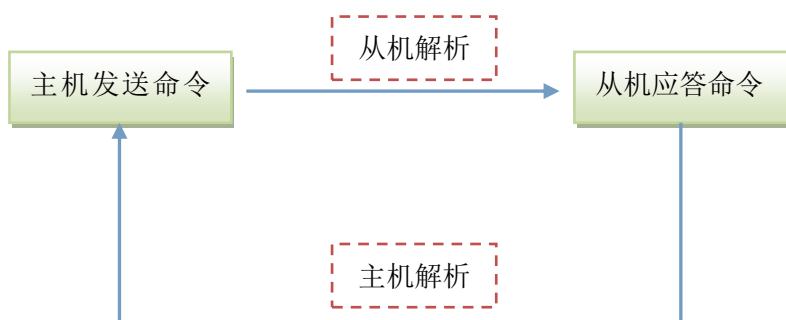
    case SLAVE_REG_P02:
        g_tVar.P02 = reg_value;          /* 将值写入保存寄存器 */
        break;

    default:
        return 0;                        /* 参数异常，返回 0 */
    }

    return 1;                            /* 读取成功 */
}

```

上面就讲解了 MODBUS 从站接收到命令后如何解析命令，并如何执行命令的。
特别注意 RS485 MODBUS 协议步骤如图：



第5章 文档更新记录

版本	更改说明	作者	发布日期
V0.1	初版。	xd	2015-12-02
V0.5	完善 MODBUS 协议介绍章节。补充必要的知识点。	Armfly	2016-01-11