Заметки о Modbus

20.01.2021
версия 1.0

https://oscat.ru/

Оглавление

Оглавление....2
Введение....3
1. Общие вопросы....5
1.1. Карты регистров и модели адресации....5
1.2. Модели памяти....16
1.3. Область памяти 2x....18
1.4. Ошибка ILLEGAL DATA VALUE (exception code 03)....18
1.5. Порядок байт/регистров....20
1.6. ПО для дешифровки Modbus-запросов....21
2. Modbus RTU....23
2.1. Паузы между символами и фреймами ($t_{1.5}$ и $t_{3.5}$)....23
2.2. Широковещательная рассылка (broadcast)....31
2.3. Надежность CRC....32
2.4. Порядок байт в CRC....34
2.5. Требования к линии связи (терминаторы, подтяжка и т.д.)....35
3. Modbus ASCII....36
3.1. Преимущества и недостатки....36
3.2. Размер символа (7 бит данных)....36
3.3. Расчет LRC....37
4. Шлюзы Modbus....39
4.1. Типы шлюзов....39
4.2. Unit ID....40
4.3. Специальные коды ошибок для шлюзов....41
5. Список литературы....42

Введение

Modbus является одним из самых распространенных промышленных протоколов. Согласно отчету HMS Industrial Networks за 2020 год он занимает 10% рынка (5% приходится на Modbus RTU и 5% – на Modbus TCP). При этом Modbus является одним из первых (а, возможно, и самым первым) промышленных протоколов – он был представлен в 1979 году разработчиками ПЛК линейки Modicon (с которой в конце 1960-х началось зарождение рынка ПЛК) – и, собственно, название протокола означает «Modicon Bus». Позже бренд Modicon был перепродан компании Schneider Electric, которая владеет им и в настоящее время. Вместе с брендом Schneider Electric долгое время владела правами на протокол Modbus, но в 2004 передала их некоммерческой организации Modbus IDA.

Таким образом, в настоящий момент за протоколом не стоит «основной» вендор, который занимается его контролем и развитием (например, для EtherCAT таким вендором является Beckhoff, для Profibus/Profinet – Siemens, для CC-Link – Mitsubishi Electric и т.д.). В принципе, это является редкостью – из других известных протоколов, не связанных с конкретными вендорами, можно вспомнить разве что CANopen и EtherNet/IP (из современных – еще OPC UA и MQTT). За счет этого Modbus стал одним из популярных способов интеграции устройств различных производителей – например, для ПЛК Siemens основным протоколом может быть PROFINET, а для Mitsubishi Electric – CC-Link, при этом каждый ПЛК не поддерживает протокол «конкурента», но они оба поддерживают Modbus TCP, что позволяет настроить между ними обмен данными.

На момент создания протокола – в качестве физического уровня использовался интерфейс RS-232, но в настоящий момент в основном используется RS-485 (для Modbus RTU/ASCII) и TCP/IP (для Modbus TCP).

Основные преимущества протокола:

- открытость и отсутствие лицензирования – спецификация является общедоступной и выложена на официальном сайте Modbus IDA;
- простота реализации – не требуются значительные аппаратные ресурсы и специальные микросхемы, программная реализация также обычно является простой и компактной;
- распространенность – протокол поддерживается значительным количеством устройств и практически всеми основными вендорами.

К недостаткам протокола можно отнести:

- отсутствие типизации – данные передаются в виде битов или регистров. Передача данных с плавающей точкой, строк и т.д. не специфицирована и может отличаться у разных производителей. Зачастую это требует от пользователя низкоуровневой работы с памятью в программе контроллера;
- отсутствие стандартизированного описания карт регистров в текстовом формате (например, .xml) – это затрудняет настройку обмена с устройствами, так как пользователю требуется вручную прописывать параметры каждого запроса. Некоторые производители делают для своего ПО шаблоны своих Modbus-устройств – но, очевидно, это не решает проблему при

использовании их устройств вместе с контроллерами или ПО другого производителя. Кроме того, в документации разных производителей часто используются разная терминология и модели адресации;

- отсутствие средств безопасности – данные передаются в открытом виде. Тем не менее, для Modbus TCP существует относительно недавно разработанное расширение Modbus Security (с поддержкой TLS), но оно пока не получило особого распространения;
- отсутствие режима multi-master для интерфейсов RS-232/RS-485 – с одной стороны, это следует из-за ограничений последовательного интерфейса, но другие протоколы, основанные на этих же интерфейсах, поддерживают такую возможность (например, CAN и Profibus).

Modbus – достаточно простой протокол, и в сети можно найти множество статей для начинающих пользователей, описывающих его основы. Кроме того, крупные производители оборудования и ПО обычно предоставляют инструкции по настройке обмена для своих устройств. Но в ряде конкретных ситуаций даже у опытных пользователей, настроивших обмен уже с десятками разных приборов, может возникнуть непонимание определенных специфических моментов. В этой статье мы постарались рассказать о таких моментах, с которыми сами в свое время столкнулись на практике. Подразумевается, что читатель уже знаком с протоколом, имеет хотя бы небольшой опыт его использования и, соответственно, понимает терминологию (master/slave, функции, регистры и т.д.).

Авторы: Евгений Кислов, Екатерина Чибисова

1. Общие вопросы

1.1. Карты регистров и модели адресации

Для настройки обмена со slave-устройством на стороне master-устройства требуется указать адрес slave'а и сконфигурировать запросы, которые будут ему отправляться. Запрос описывается тремя параметрами:

- используемый код функции Modbus (например 0x03 – чтение input-регистров, 0x06 – запись одного holding-регистра и т.д.);
- адрес начального бита/регистра;
- число бит/регистров, к которым будет применена выбранная функция.

Кроме того, в случае работы с регистрами также требуется понимать типы данных для параметров устройства – так как сам Modbus не специфицирует типы данных, то значение любого типа представляется в нем как один или несколько регистров (с точки зрения стандарта МЭК 61131-3 понятие «регистр» эквивалентно переменной типа WORD). Соответственно, пользователю нужно знать тип данных, чтобы правильно интерпретировать полученные значения.

С указанием адреса slave-устройства обычно не возникает проблем – он задается либо в ПО, с помощью которого проводится конфигурация slave'а, либо аппаратно (например, с помощью DIP-переключателей на корпусе прибора).

А вот с параметрами запросов могут возникнуть сложности, связанные с особенностями конкретного ПО и документации на устройства.

Код функции

Код функции может задаваться двумя основными способами – либо в явном виде, либо через область памяти. При указании в явном виде всё понятно – пользователь просто выбирает из списка нужную функцию. Такой способ, к примеру, используется в среде CODESYS V3.5:

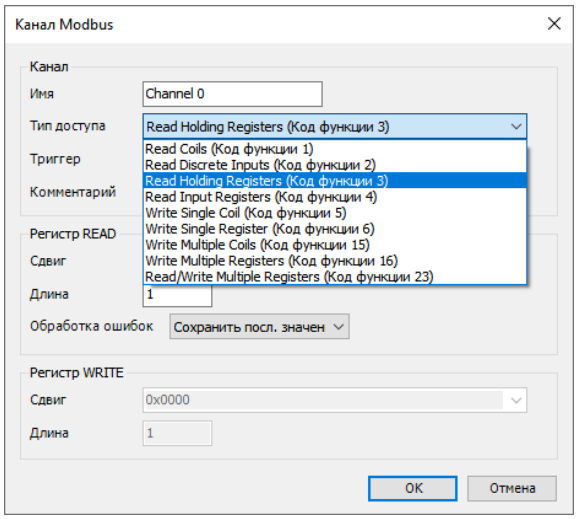


Рис. 1.1. Среда CODESYS V3.5 – код функции задается в явном виде

Но, например, в ПО EasyBuilder, с помощью которого конфигурируются панели оператора Weintek, нет возможности выбрать код функции в явном виде – вместо этого в настройках элемента визуализации пользователь видит вот такой список обозначений:

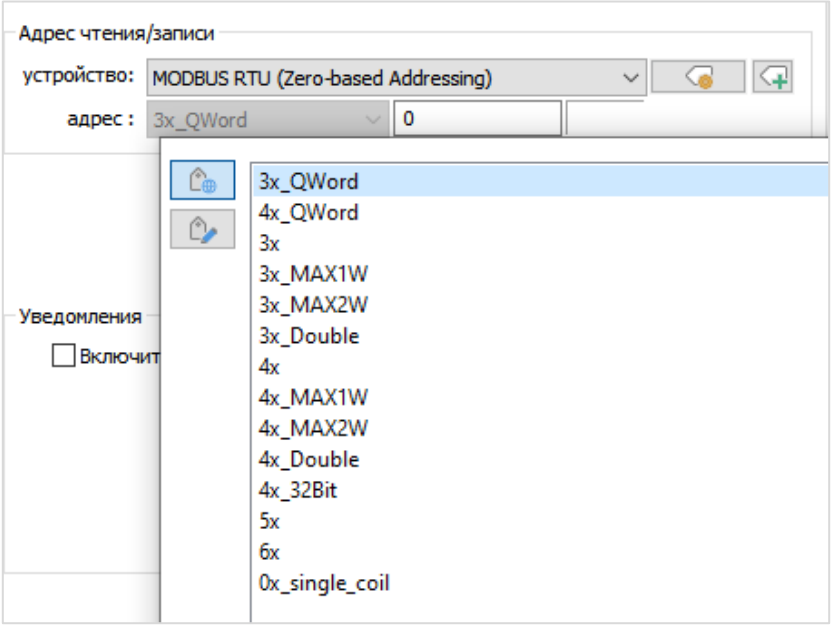


Рис. 1.2. Среда EasyBuilder (Weintek) – код функции задается через идентификатор области памяти

Начинающий пользователь может подумать, что **0x**, **1x** и т.д. – это и есть коды функций. Но это не так. В данном случае, **0x** … **6x** – это идентификаторы областей памяти Modbus. Эти идентификаторы не описаны в текущей версии спецификации[1], но по историческим причинам используются в некотором ПО и в настоящее время. В таблице ниже приведено соответствие идентификаторов областей памяти и кодов функций.

Табл. 1.1. Соответствие идентификаторов областей памяти и кодов функций Modbus

Идентификатор	Область памяти	Описание	Функции Modbus
0x	Coils	Биты, доступные для чтения и записи	01 (0x01) Read Coils 05 (0x05) Write Single Coil 15 (0x0F) Write Multiple Coils.
1x	Discrete Inputs	Биты, доступные только для чтения	02 (0x02) Read Discrete Inputs
3x	Input Registers	Регистры, доступные только для чтения	04 (0x04) Read Input Registers
4x	Holding Registers	Регистры, доступные для чтения и записи	03 (0x03) Read Holding Registers 06 (0x06) Write Single Register 16 (0x10) Write Multiple Registers

В этот момент у читателя должно возникнуть два вопроса: как вообще об этом можно догадаться и почему в таблице 1.1 не описаны идентификаторы 5x и 6x?

[1] Читатель с ником **SiMM** уточняет, что они упоминаются в документе Conformance Test Specification for Modbus TCP (принцип выбора идентификаторов там не поясняется)

Ответ на оба вопроса один и тот же: «это описано в документации». В документе PLC Connection Guide (он доступен на сайте Weintek и сайтах дистрибьюторов его продукции) можно найти следующую информацию:

Device Address:

Bit/Word	Device type	Format	Range	Memo
B	0x	DDDDD	1 ~ 65535	Output bit
B	1x	DDDDD	1 ~ 65535	Input bit (read only)
B	3x_Bit	DDDDDdd	100 ~ 6553515	Input Register bit (read only)
B	4x_Bit	DDDDDdd	100 ~ 6553515	Output Register bit
B	6x_Bit	DDDDDdd	100 ~ 6553515	Output Register bit
B	0x_multi_coils	DDDDD	1 ~ 65535	Write multiple coils
W	3x	DDDDD	1 ~ 65535	Input Register (read only)
W	4x	DDDDD	1 ~ 65535	Output Register
DW	5x	DDDDD	1 ~ 65535	4x double word swap
W	6x	DDDDD	1 ~ 65535	4x single word write
W	4x_32Bit*	DDDDD	1 ~ 65535	Output Register

4x_32Bit will only read / write 2 words for each package, for continuous addresses, it will be divided into several packages.

Modbus RTU function code:

0x	0x01 Read coil	0x05 write single coil
0x_multi_coils	0x01 Read coil	0x0f write multiple coils
1x	0x02 Read discrete input	N/A for write operation
3x	0x04 Read input register	N/A for write operation
4x	0x03 Read holding register	0x10 write multiple registers
5x	0x03 Read holding register	0x10 write multiple registers

(Note: reverse word order in double word format)

496

WEINTEK PLC Connection Guide

3xbit is equivalent to 3x

4xbit is equivalent to 4x

6x	0x03 Read holding register	0x06 write single register

(Note: 6x is limited to device of one word only)

Рис. 1.3. Фрагмент из документа PLC Connection Guide, в котором описывается соответствие идентификаторов и используемых областей памяти. Столбец **Format** описывает формат ввода адресов – мы вернемся к нему в разговоре об адресах регистров

Таким образом, даже если ранее вы не были знакомы с информацией из табл. 1.1, то вы смогли бы найти ее в документации на это конкретное устройство.

Итак, в данном случае код функции определяется неявно при выборе идентификатора области памяти. Это упрощает настройку, потому что значительная часть элементов визуализации используется и для чтения, и для записи (например, элемент типа *Numeric Input* отображает текущее значение параметра (чтение) и позволяет его изменить (запись)). Соответственно, при «ручной» настройке (см. рис. 1.1) потребовалось бы настраивать два Modbus-запроса – а в данном случае достаточно указать только один идентификатор.

Поскольку для областей Coils и Holding-регистров доступно по 2 функции записи (одного и нескольких бит/регистров соответственно – **0x05/0x0F** для Coils и **0x05/0x10** для Holding-регистров), то Weintek предоставляет специальные идентификаторы **0x_multi_coils** (для функции записи группы бит **0x0F**) и **6x** (для функции записи Holding-регистров **0x06**). Таким образом, это предоставляет пользователю все нужные комбинации для функций чтения/записи.

Нам доводилось встречать устройства, где используемая функция записи зависит от типа данных. Например, при выборе идентификатора **4x** и типа данных WORD используется **0x06** (Write Single Register), а типа DWORD – **0x10** (Write Multiple Registers). Это кажется логичным – для записи одного регистра применяется функция записи одного регистра, а для записи двух и более регистров – функция записи группы регистров. Но проблема в том, что некоторые slave-устройства не поддерживают функцию **0x06** и требуют использования функции **0x10** даже для записи одного регистра. Сложно сказать, зачем отказываться от поддержки такой общепринятой функции – в специфичных случаях, возможно, разработчикам не хватило памяти (для embedded-устройств она может измеряться килобайтами), но, вероятно, в основном это связано с ленью и недостатками ТЗ.

Кроме того, Weintek предоставляет идентификаторы **3x_Bit**, **4x_Bit** и **6x_Bits** для побитного доступа к Input- и Holding-регистрам. Это удобно в тех случаях, когда в slave-устройстве используются битовые маски (например, для хранения регистра статуса, слова состояния в ПЧВ и т.д.). Интересный момент, который вызывает вопрос и у нас – производит ли панель оптимизацию запросов при использовании побитового доступа? Иными словами, если добавить на экран две лампочки с идентификатором **3x_Bit** и адресами бит 0 и 1 – то будет ли панель отправлять два отдельных запроса или автоматически сформирует один групповой запрос? Мы не можем ответить на этот вопрос (если вам интересно – то вы можете проверить это опытным путем в симуляторе EasyBuilder), но отметим, что некоторое ПО способно выполнять такую оптимизацию.

Осталось рассказать об идентификаторе **5x** – как видно из документации, он соответствует области **4x** с «переставленным» порядком регистров – то есть если в памяти slave-устройства лежит значение 0xAABBCCDD, то при использовании идентификатора **4x** мы получим на панели именно 0xAABBCCDD, а при использовании **5x** – значение 0xDDCCBBAA (с инвертированным порядком регистров). Эта возможность полезна, так как в разных устройствах данные могут храниться с разным порядком регистров, а протокол Modbus, как мы уже говорили, не специфицирует типы данных (иначе, соответственно, в стандарте был бы жестко определен порядок регистров при передаче значений различных типов).

У внимательных читателей мог остаться вопрос – почему ни в табл. 1.1, ни на рис. 1.3 не используется идентификатор **2x**? Ответ на этот вопрос приведен в п. 1.3.

Отметим еще один момент – коды функций Modbus и идентификаторы областей памяти легко спутать, так для них используется один и тот же набор цифр. Поэтому довольно часто даже инженер, который запомнил концепцию идентификаторов, может допустить ошибку и интуитивно подумать, что области **3x** соответствует функция **0x03** – но это не так! Области **3x** (Input-регистры) соответствует функция чтения **0x04** (Read Input Registers), а области **4x** (Holding-регистры) – функция **0x03** (Read Holding-регистры). Эту особенность желательно запомнить раз и навсегда – она сбивает с толку даже опытных инженеров.

Рассмотрим еще один пример неявного выбора функции – OPC-сервер MasterOPC Universal Modbus Server от компании ИнСат. Мы регулярно используем это хорошо продуманное и удобное ПО для отладки обмена с другими устройствами. При добавлении тега (опрашиваемого параметра) для него указывается регион (то есть области памяти), а код функции выбирается автоматически. При этом код функции записи можно выбрать в явном виде (см. на рис. 1.4 параметр **Принудительная запись командой 6** – соответственно, несложно догадаться, что по умолчанию для записи используется функция **0x10**).

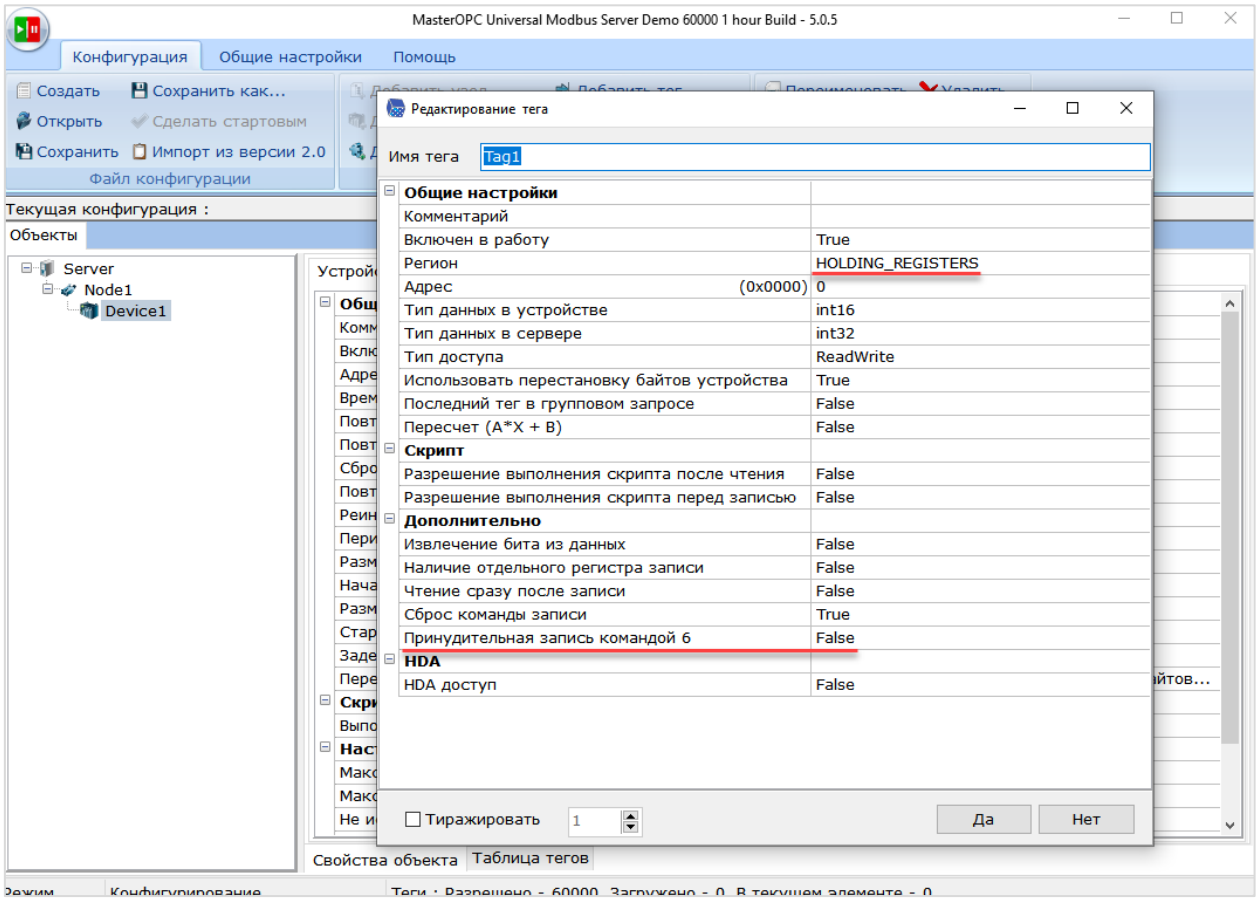


Рис. 1.4. Добавление тега в OPC-сервере MasterOPC Universal Modbus Server – пользователь указывает область памяти, а код функции будет выбран автоматически

Мы рассмотрели особенности выбора кода функции на стороне master-устройств. С точки зрения slave-устройства – пользователь должен знать, какие коды функций оно поддерживает. Обычно это написано в документации. Ниже приведен фрагмент из руководства на индикатор СМИ2-М компании ОВЕН:

Приложение А. Список регистров Modbus

Таблица А.1 – Чтение и запись параметров по протоколу Modbus

Операция	Функция
Чтение	3 (0x03) или 4 (0x04)
Запись	6 (0x06) или 16 (0x10)

Рис. 1.5. Фрагмент из руководства по эксплуатации индикатора ОВЕН СМИ2-М с перечислением поддержанных функций Modbus

Тут может вызвать интерес формулировка «0x03 **или** 0x04» – это приводит нас к разговору к моделям памяти Modbus, о которых мы поговорим в п. 1.2.

Мы уже достаточно рассказали о кодах функций, и пора бы перейти к адресам регистров – кстати, там нам снова придется вспомнить об идентификаторах областей памяти.

Адреса битов/регистров

С точки зрения спецификации Modbus ([1], п. 4.4) адрес регистра – это число в диапазоне 0…65535. Казалось бы, какие проблемы могут возникнуть при его определении?

Первая возможная проблема – система счисления, в которой записаны адреса. В повседневной жизни мы в основном используем десятичную систему (DEC), но в системах автоматизации часто используется шестнадцатеричная (HEX). Иногда ПО поддерживает ввод адреса в разных системах счисления – например, в CODESYS V3.5 адрес регистра можно указать как в DEC, так и в HEX, причем в HEX двумя способами – с «общепринятым» префиксом **0x** и «мэковским» (определенным в стандарте МЭК 61131-3) **16#:**

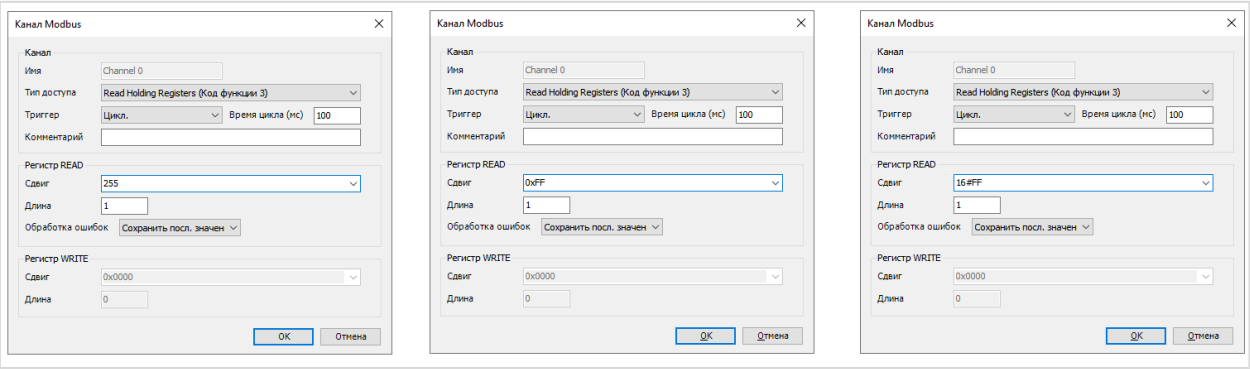


Рис. 1.6. Ввод адреса регистра в различных системах счисления в CODESYS V3.5 (*рисунок хорошо масштабируется*)

При этом независимо от способа ввода все добавленные адреса отображаются унифицировано – в формате HEX с префиксом **16#.**

В документации slave-устройств адреса могут быть записаны как в DEC, так и в HEX:

- хороший подход – в явном виде указывать, какая система счисления используется в документе, еще лучше – использовать сразу обе (см. рис. 1.7);
- приемлемый подход – использовать формат HEX и общепринятые префиксы/постфиксы (например, 0x0010 или 10h). Хорошим тоном является использование ведущих нулей для HEX-значений. Посчитать, что запись 0x10 соответствует «десятичному» 10 может только студент или совсем начинающий сотрудник;
- плохой подход – отсутствие каких-либо указаний на систему счисления. Самый худший вариант – когда все адреса slave'а записаны в HEX, но содержат только цифры. Догадаться, что «адрес регистра: 112» соответствует адресу 0x112 – наверное, сходу не сможет никто. К сожалению, нам доводилось наталкиваться на такое в документации – но, к счастью, это происходило лишь всего пару раз.

Таблица 6.3 – Регистры обмена по протоколу ModBus

Параметр	Значение (ед. изм.)	Адрес регистра		Тип доступа	Формат данных
		DEC	HEX		
Значение (float) на входе 1	—	4000	0xFA0	Только чтение	FLOAT 32
Циклическое время измерения входа 1	0...65535 (миллисекунд)	4002	0xFA2	Только чтение	UINT 16

Рис. 1.7. Фрагмент из руководства по эксплуатации модуля ввода ОВЕН МВ210-101 – адреса регистров указаны как в DEC, так и HEX

Еще одной проблемой при записи адресов может быть модель адресации. Существует две известные модели адресация:

- стандартная (ее также называют физической) – она описана в спецификации ([1], п. 4.4). В этой модели адрес регистра – это просто число из диапазона 0…65535. Если у вас «стандартное» master-устройство (т.е. master-устройство, которое ожидает ввода адреса в формате физической адресации) – то для опроса регистра slave-устройства с адресом 10 вам нужно будет в настройках master-устройства указать адрес регистра 10;
- логическая – она использовалась в старых версиях спецификации Modbus.

Давайте поговорим про логическую адресацию более подробно. Ее отличия от физической адресации заключаются в следующем:

- первым символом адреса является идентификатор области памяти, о котором мы говорили в прошлом разделе (см. табл. 1.1);
- следующие 4 символа содержат адрес регистра в DEC в диапазоне 0001…9999;
- как следует из предыдущего пункта – адресация регистров ведется с единицы (а не с нуля, как при физической адресации).

Т.е. «регистр с адресом **30001**» в данном случае будет означать «**Input**-регистр с адресом **0**».

Ниже приведен фрагмент карты регистров 8-канального модуля аналогового ввода ZT-2017 компании ICP DAS:

10129 ~ 10136	The under range status of channels 0 to 7 (supports types 0x7 and 0x1A only)	R
30001 ~ 30008	The analog input value of channels 0 to 7	R
30513 ~ 30520	The high latch value	R
30545 ~ 30552	The low latch value	R
40257 ~ 40264	The type code of channels 0 to 7	R/W

Рис. 1.8. Фрагмент карты регистров модуля ввода ICP DAS ZT-2017, в которой используется логическая адресация

Адреса 10129…10136 («under range status» – бит ошибки «полученное значение меньше нижней границы измерения») соответствуют битам 128…135 области **1x** (Discrete Inputs).

Адреса 30001…30008, в которых хранятся значения входов модуля, соответствует регистрам 0…7 области **3x** (Input Registers). Это подтверждается примером, приведенным в руководстве перед картой регистров:

Example :

A. Read the analog input value of the module 01, the following command should be sent

01 04 00 00 00 08 F1 CC

Рис. 1.9. Пример из руководства ICP DAS ZT-2017 – команда считывания значений аналоговых входов

Мы видим, что в примере используется функция **0x04** (Read Input Registers) с начальным адресом **0** и количеством считываемых регистров **8** – никаких адресов типа «30001» в запросе нет.

Соответственно, адреса 40257…40264 соответствуют регистрам 256…263 области 4x (Holding Registers).

В хорошей документации можно увидеть подробную информацию по используемой системе адресации с примерами конкретных запросов. Это значительно упрощает настройку обмена. Но в некоторых случаях документация может запутывать. Например, адрес 40001 может использоваться и при физической адресации – в этом случае он может соответствовать биту или регистру с адресом 40001 из любой области памяти.

Ниже приведен фрагмент руководства на электропривод ГЗ КС 15 компании ГЗ Электропривод. При беглом прочтении можно подумать, что в документе используется логическая адресация – на это намекают пятизначные адреса и нумерация с единицы. Но при внимательном прочтении становится ясно, что для чтения битов 00001…00005 используется функция **0x02** (Read Discrete Inputs). Мы помним, что идентификатор **0x** соответствует области Coils – и это приводит нас к выводу, что адресация в документе – физическая:

- адреса 00001…00005 соответствуют битам 1…5 области Discrete Inputs;
- адреса 00017…00018 соответствуют битам 17…18 области Coils;
- адрес 40001 соответствует регистру 40001 области Holding Registers и т.д.

В таблице 2 приведены адреса рабочих регистров необходимых для работы с модулем.
Для работы с дискретными входными сигналами используется функция №02 Read discrete inputs.
Для работы с дискретными выходными сигналами используются функции №01 Read Coils, №05 Write single coil, №15 Write multiple coils.
Для работы с аналоговым входом используется функция №04 Read input register.
Для работы с Вспомогательными регистрами используются функции №03 Read holding registers, №06 Write single register, №16 Write multiple registers.

4.4 Плата индикации

Плата индикации устанавливается в окно индикации электропривода (рис. 10). На плате расположены светодиоды индикации о текущем состоянии привода.

Также блок КС15 выдает во внешнюю систему управления следующие сигналы:
- открыто
- закрыто
- превышение рабочего момента
- готов
- авария

Выдача сигналов состояния организована по принципу «сухой контакт».

Максимальный рабочий ток через контакты составляет 6А при напряжении 250В переменного тока.

Блок КС15 осуществляет контроль питающей электропривод сети и защищает электропривод от обрыва питающей фазы, неправильного чередования фаз и падения питающего напряжения.

В случае исправной сети блок выдает сигнал «Готов» на плате ПРТ, плате индикации в окне индикации электропривода, в противном случае блок управления запрещает работу

Таблица 2. Адреса рабочих регистров необходимых для работы с модулем.

Адрес	Канал	Назначение	Атрибут	Примечание
Дискретные входные сигналы				
00001	0	Дискретный входной сигнал	R	Сигнал 13
00002	1	Дискретный входной сигнал	R	Сигнал 14
00003	2	Дискретный входной сигнал	R	Сигнал 15
00004	3	Дискретный входной сигнал	R	Сигнал 41
00005	4	Дискретный входной сигнал	R	Сигнал 42
Дискретные выходные сигналы				
00017	0	Дискретный выходной сигнал	R/W	Сигнал 5 «Открыть»
00018	1	Дискретный выходной сигнал	R/W	Сигнал 7 «Закрыть»
Аналоговые входные сигналы				
40001	0	Аналоговый входной сигнал	R	Сигнал 2 «Степень открытия»
Вспомогательные регистры				
40211		Версия программы	R	
40212		Статус сброса модуля	R	Сброс по СОР

Рис. 1.10. Фрагмент руководства ГЗ КС 15

Это было несколько неожиданно, не так ли?

В других случаях мы можем увидеть в документации смесь логической и физической адресации:

➢ 3xxxx: AI Address (Base 0)

Starting Address	Points	Description	Bits per Point	Range	Access Type
151 (0x97)	1	Firmware Version	16	"123" denotes that the version is 1.2.3	R
158 (0x9E)	1	Modbus Communication Status	16	0 = No Error 1 = Timeout	R
160 (0xA0)	1	Pair-Connection Status	16	0 = Normal 1 = Timeout 2 = Disconnected	R
Notes	"**R**": Read				

Рис. 1.11. Фрагмент из карты регистров модуля ввода-вывода ICP DAS ET-2260

Здесь обозначение **3xxxx** нужно только для понимания кода функции, который должен быть выбран в запросе (соответственно, **3x** – идентификатор области Input Registers, так что нужно использовать функцию **0x04** (Read Input Registers)). Адреса при этом указаны в явном виде – т.е. «151» означает «Input-регистр с адресом 151».

И последний пример – фрагмент карты регистров тепловычислителя ТСРВ-024 компании Взлет:

Регистры хранения типа целое значение 1 байт

MB адрес		Название параметра	Название в приборе	Пределы	Уровень доступа	
Логический	Физический				Видимость	Редактирование
400001	0x0000	Сетевой адрес устройства Вторичника	Адрес	1 ... 247	Работа Сервис Настройка Тестирование	Работа Сервис Настройка Тестирование
400002	0x0001	Задержка ответа прибора Вторичника, мс	Задержка	0 ... 125	Работа Сервис Настройка Тестирование	Работа Сервис Настройка Тестирование

Рис. 1.12. Фрагмент карты регистров тепловычислителя Взлет ТСРВ-024

В ней приведены как логические, так и физические адреса – это позволяет легко использовать документ инженерам, ранее встречавшимся лишь с одной из моделей адресации. Код нужной функции можно понять либо из логической адресации (**4x** – идентификатор области памяти Holding Registers, т.е. для чтения нужно использовать функцию **0x03**, а для записи – **0x06** или **0x10**), либо из заголовка «Регистры хранения» (это один из вариантов русскоязычного перевода термина «Holding Registers»).

Обратим внимание на интересный момент – в логической адресации данного прибора используются шестизначные адреса, хотя общепринятыми (и специфицированными в [4]) являются пятизначные. С другой стороны, это позволяет расширить диапазон доступных регистров – потому что при «классической» логической модели адресации размер каждой области памяти ограничен 9999 регистрами, хотя современная спецификация Modbus ([1], п. 4.4) устанавливает максимальный размер каждой области памяти в 65536 элементов.

Таким образом, в некоторых случаях понять систему адресации для конкретного прибора очень легко, а в других – сложно. В конечном итоге всё зависит от качества документации и опыта инженера.

Напоследок отметим, что для идентификаторов областей памяти в документации могут использоваться следующие термины:

0x (Coils)	Ячейки, Катушки, Обмотки, DO, Дискретные выходы
1x (Discrete Inputs)	DI, Дискретные входы
3x (Input Registers)	Входные регистры, Регистры ввода, AI, Аналоговые входы
4x (Holding Register)	Регистры хранения, AO, Аналоговые выходы

На наш взгляд, оптимальным является использование англоязычных вариантов (например, «Holding Registers» или «Holding-регистры»).

Мы закончили разговор об адресах регистров и теперь осталось поговорить о последнем параметре Modbus-запроса, задаваемом пользователем – количестве регистров в запросе.

Количество регистров

В данном разделе мы не будем говорить про биты (Coils и Discrete Inputs), потому что с ними всё ясно – каждый такой битовый параметр имеет размер 1 бит.

С регистрами всё сложнее – slave-устройство может содержать значения с плавающей точкой, строки, структуры и т.д. – каждый из таких параметров занимает больше одного регистра. Соответственно, пользователю нужно знать, сколько регистров занимает каждый параметр, чтобы корректно настроить опрос в master-устройстве. Помимо этого, требуется описание типа параметра (формата его представления), чтобы преобразовать набор регистров в осмысленное значение. Обычно по типу параметра можно сразу понять, сколько регистров он занимает.

Ниже приведен фрагмент карты регистров блока питания БП120К компании ОВЕН. В нем для каждого параметра указан формат данных, а в примечании – размер каждого типа данных в битах.

Параметры, доступные по протоколу Modbus

Приложение А. Параметры, доступные по протоколу Modbus

ПРИМЕЧАНИЕ
Используемые форматы данных:
- UINTx – 16-, 32- и 48-разрядное беззнаковое целое число;
- FLOAT32 – 32-разрядное число стандарта IEEE 754 (IEC 60559).

Заводские настройки выделены ***полужирным курсивом***.

Параметр	Значение (ед. изм)	Адрес регистра		Тип доступа	Формат данных	Комментарий
		DEC	HEX			
Контроль реле	0 – Выкл.; 1 – Вкл.; ***2 – Автомат***	1617	0x0651	Чтение и запись	UINT16	Настраивается возможность контролировать реле встроенным микроконтроллером в зависимости от режима работы блока (см. раздел 6): • Выкл. – контроллер не меняет состояние реле; • Вкл. – возможность удаленно включить подключенное к выходу реле (при работающем блоке); • Автомат – микроконтроллер переключает реле в зависимости от состояния блока
Настройка выходного тока	0,010…***5,5***…6 (А)	1618	0x0652	Чтение и запись	FLOAT32	По установленному в данном параметре значению блок переходит в режим ограничения тока и начинает снижать («заваливать») выходное напряжение

Рис. 1.12. Фрагмент карты регистров блока питания ОВЕН БП120К

Другой распространенный вариант – указание числа регистров, которое занимает каждый параметр. Ниже приведен фрагмент карты регистров модуля ввода-вывода ioLogic E1242 компании Moxa. Модуль имеет 4 аналоговых входа. Поэтому под «AI_burnoutValue» *(это значение, по которому определяется обрыв датчика – например, 2.0 мА*) скрывается сразу 4 параметра типа Float (значение с плавающей точкой), каждый из которых занимает 2 регистра («2 words»). Стоит заметить, что фактический тип данных (Float) в карте регистров не указан – что не очень корректно, хотя у опытного инженера это, конечно, не вызовет вопросов. Кстати, обратите внимание, что в документации приведены и физические, и логические адреса регистров, а область памяти указана в явном виде (HOLDING REGISTER) – это довольно удобно.

ioLogik E1242 Modbus Address and Register Map

I/O

Parameter Name	Description	Start Address (decimal)	Point Type	Start Register (decimal)	Length	Access	Type
AI_burnoutValue	high/low word	0560	03:HOLDING REGISTER	40561	8	R/W	2 words
AI_mode	0: 0-10 V 1: 4-20mA 2: 0-20mA 4: 4-20mA burnout	0544	03:HOLDING REGISTER	40545	4	R/W	word

Рис. 1.13. Фрагмент карты регистров модуля ввода-ввода Moxa ioLogic E1242

Иногда адрес регистра указывается для каждого регистра параметра. Ниже приведен фрагмент карты регистров модуля ввода ZT-2017 компании ICP DAS. Параметры «The Firmware version» и «The Module Name» фактически имеют тип UINT32 (т.е. занимают два регистра), при этом в карте регистров указан адрес каждого регистра параметра с указанием порядка их передачи в ответе. Это довольно удобно, потому что спецификация протокола Modbus этот порядок не уточняет.

40481	The firmware version (low word)	R
40482	The firmware version (high word)	R
40483	The module name (low word)	R
40484	The module name (high word)	R

Рис. 1.13. Фрагмент карты регистров модуля ввода ICP DAS ZT-2017

На этом мы заканчиваем разговор о картах регистрах и переходим к другой теме, которая близко с ними связана – к обсуждению моделей памяти Modbus.

1.2. Модели памяти

Когда мы говорили об областях памяти, то подразумевали, что они являются независимыми – то есть, например, области Input- и Holding-регистров имеют независимую адресацию (и адреса регистров в них совершенно спокойно могут быть совпадающими) и для работы с ними используются разные функции. Но это только один из примеров организации областей памяти Modbus.

Он приведен в спецификации ([1], п.4.3):

Example 1 : Device having 4 separate blocks

The example below shows data organization in a device having digital and analog, inputs and outputs. Each block is separate because data from different blocks have no correlation. Each block is thus accessible with different MODBUS functions.

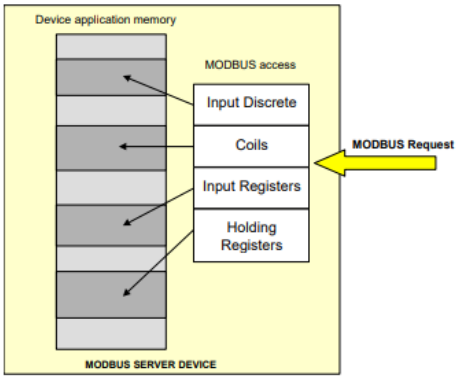


Figure 6 MODBUS Data Model with separate block

Рис. 1.14. Пример устройства с независимыми областями памяти Modbus

Другой пример организации памяти – наложение всех областей памяти друг на друга (т.е. использование общего адресного пространства для всех областей). В этом случае запрос input- и holding регистра с адресом 0 вернет одно и то же значение (потому что фактически мы обратились к одному и тому же регистру), а запрос discrete input и coil с адресом 0 вернет младший бит регистра 0. Такая «общая» модель памяти, например, используется в контроллерах ОВЕН ПЛК1хх, некоторых панелях оператора и других устройствах. Иногда производители в этом случае убирают поддержку функций **0x02** (Read Discrete Input) и **0x04** (Read Input Registers), потому что их использование теряет смысл (так как всё равно любой регистр доступен для записи).

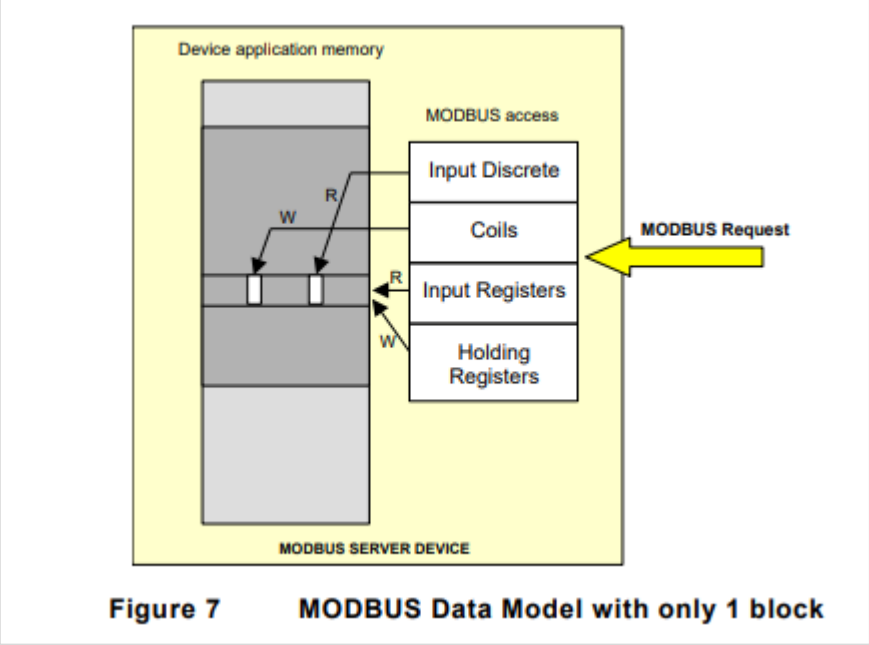


Рис. 1.15. Пример устройства с наложением всех областей памяти Modbus

Спецификация допускает использование и других моделей памяти. Например, в среде **CODESYS V3.5** для Modbus Slave используется две области памяти – одна из них представляет собой «наложенные» Coils и Holding Registers, а вторая – «наложенные» Discrete Inputs и Input Registers. Начиная с версии CODESYS V3.5 SP15 пользователь может с помощью специальной настройки изменить модель памяти на вариант с 4 независимыми областями (см. рис. 1.14).

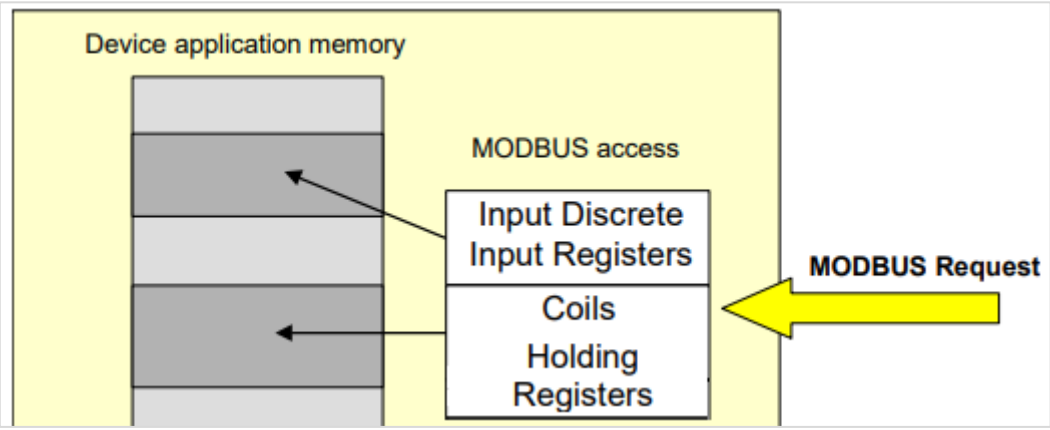


Рис. 1.16. Пример устройства с двумя областями памяти Modbus и наложением битов/регистров

1.3. Область памяти 2x

Еще раз напомним идентификаторы областей памяти Modbus:

- 0x – Coils;
- 1x – Discrete Inputs;
- 3x – Input Registers;
- 4x – Holding Registers.

Внимательный читатель, вероятно, спросит – «А почему после **1x** сразу следует **3x**? Что случилось с **2x**?»

Как вспоминают ветераны АСУ (здесь и здесь) – в минувшие времена при работе с ПЛК первых поколений этот идентификатор использовался для доступа к памяти барабанного командоаппарата. По мере развития средств автоматизации потребность в командоаппарате отпала, и вместе с ним канула в лету область **2x**.

1.4. Ошибка ILLEGAL DATA VALUE (exception code 03)

Среди ошибок, описанных в спецификации Modbus, есть ошибка **03** (ILLEGAL DATA VALUE). Кажется, что по названию всё понятно – slave-устройство вроде бы должно возвращать ее при попытке записи значения, выходящего за допустимый диапазон параметра (например, регистр задания частоты ПЧВ может иметь ограничение 0…5000, где 5000 соответствуют 50 Гц). В некоторых устройствах эта ошибка действительно возвращается slave'ом именно в таких случаях.

Но на самом деле – это неправильное понимание смысла ошибки и отступление от спецификации. Согласно спецификации ([1], п. 7):

«It specifically does NOT mean that a data item submitted for storage in a register has a value outside the expectation of the application program, since the MODBUS protocol is unaware of the significance of any particular value of any particular register»

(перевод) «Это не означает, что значение, записываемое в регистр, выходит за диапазон, определяемый программой устройства, так как протокол Modbus не специфицирует какие-либо ограничения для значений регистров»

Согласно спецификации, ошибка ILLEGAL DATA VALUE должна возвращаться в следующих случаях:

- при попытке чтения группы бит/регистров, длина которой превышает ограничения протокола (2000 бит и 125 регистров соответственно);
- при попытке записи группы бит/регистров, длина которой превышает ограничения протокола (1968 бит и 123 регистра соответственно);

- при записи функцией **0x05** (Write Single Coil) значения, отличного от **0x0000** (выключить бит) или **0xFF00** (включить бит);
- в нескольких других специфичных случаях (например, при некорректной длине запроса на чтение/запись файла для функций **0x14**/**0x15**).

Тем не менее, приборы некоторых производителей возвращают эту ошибку именно при попытке записи «некорректного» (с точки зрения прибора) значения. В их защиту можно сказать, что это действительно удобный способ сообщить пользователю, что его команда не была принята (иначе пользователь может подумать, что выбранное им значение записано в slave-устройство и ожидать от slave-устройства выполнения каких-либо операций – которых, очевидно, в этом случае не будет).

Правильный подход – документировать это отступление от спецификации для того, чтобы пользователь мог понять дополнительную возможную причину этой ошибки. Ниже приведен фрагмент из руководства модуля ввода-вывода МК210-301 компании ОВЕН с описанием этой ситуации.

6.5.2 Коды ошибок для протокола Modbus

Во время работы модуля по протоколу Modbus возможно возникновение ошибок, представленных в таблице 6.6. В случае возникновения ошибки модуль отправляет Мастеру сети ответ с кодом ошибки.

Таблица 6.6 – Список возможных ошибок

Название ошибки	Возвращаемый код	Описание ошибки
MODBUS_ILLEGAL_FUNCTION	01 (0x01)	Недопустимый код функции – ошибка возникает, если модуль не поддерживает функцию Modbus, указанную в запросе
MODBUS_ILLEGAL_DATA_ ADDRESS	02 (0x02)	Недопустимый адрес регистра – ошибка возникает, если в запросе указаны адреса регистров, отсутствующие в модуле
MODBUS_ILLEGAL_DATA_ VALUE	03 (0x03)	Недопустимое значение данных – ошибка возникает, если запрос содержит недопустимое значение для записи в регистр

Рис. 1.17. Фрагмент руководства модуля ввода-вывода ОВЕН МК210-301 с описанием возвращаемых кодов ошибок

1.5. Порядок байт/регистров

Спецификация Modbus ([1], п. 4.2) определяет, что порядок байт при передаче – старшим байтом вперед (big endian).

4.2 Data Encoding

- MODBUS uses a 'big-Endian' representation for addresses and data items. This means that when a numerical quantity larger than a single byte is transmitted, the most significant byte is sent first. So for example

Register size	value			
16 - bits	0x1234	the first byte sent is	0x12	then 0x34

Рис. 1.18. Фрагмент спецификации Modbus с пояснением порядка байт при передаче данных

Некоторые параметры занимают несколько регистров Modbus (например, 32-bit Float – 2 регистра). Порядок, в котором будут возвращены регистры при запросе такого параметра, спецификация Modbus не определяет (это связано с тем, что спецификация вообще не описывает типы данных, которые можно передавать по протоколу). Поэтому этот порядок зависит от производителя конкретного устройства.

Большинство master-устройств предоставляют готовый функционал для изменения порядка регистров в принимаемых данных (вспомним, например, специальную область **5x** в ПО EasyBuilder для панелей Weintek – см. рис. 1.2). Для программируемых устройств эту манипуляцию можно произвести «вручную» в пользовательском проекте с помощью кода.

Некоторые slave-устройства позволяют задать порядок регистров (а иногда даже байт – на тот случай, если порядок байт в master-устройстве отличается от slave'а), который будет использоваться в ответах, отправляемых ими на запрос master-устройства. Ниже приведен скриншот из конфигуратора индикатора СМИ2-М компании ОВЕН с соответствующей настройкой:

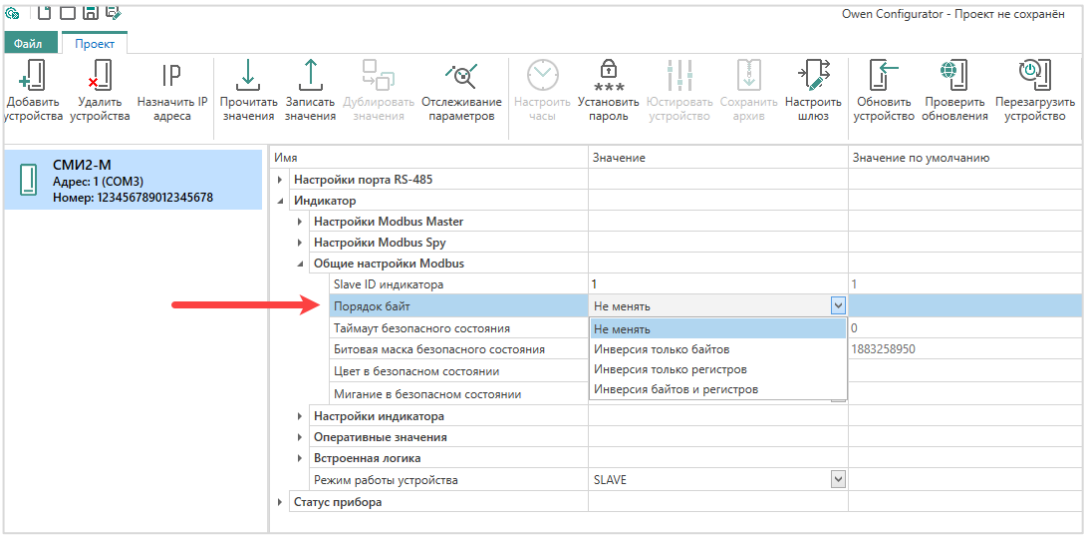


Рис. 1.19. Настройка порядка байт/регистров для Modbus в конфигураторе индикатора ОВЕН СМИ2-М

1.6. ПО для дешифровки Modbus-запросов

В некоторых случаях при отладке обмена возникает необходимость изучения конкретных запросов и ответов, посылаемых по линии связи (*мы говорим о ситуации, когда нельзя вместо одного из устройств использовать ПК со специальным ПО типа ОРС-серверов, которые показывают логи обмена в понятном человеку виде*). В этом случае удобно использовать ПО, которое помогает «разобрать» запрос и показать, какой в нем используется адрес slave-устройства, код функции и т.д. В принципе, опытный инженер в состоянии анализировать запросы относительно небольшой длины в уме, но если речь идет о групповой записи/чтении десятков регистров, и нужно вычленить из запроса значения параметров – то это становится слишком трудоемким.

При работе с Modbus TCP удобно использовать известную утилиту Wireshark, которая сразу показывает пакет в расшифрованном виде:

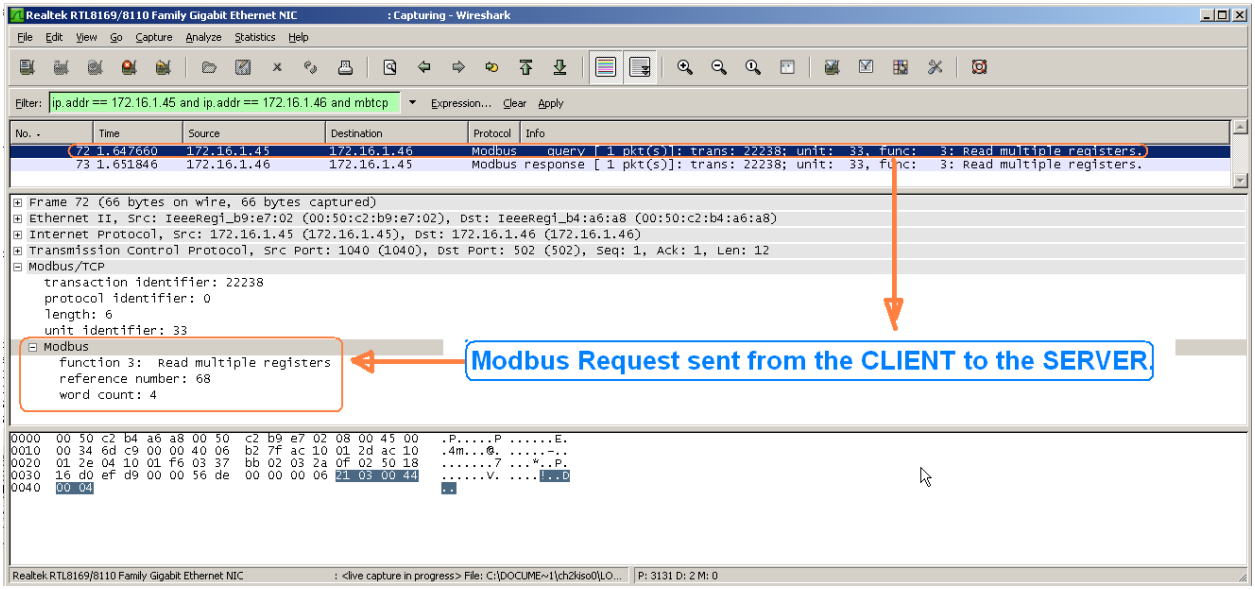


Рис. 1.20. Сниффинг и разбор Modbus TCP пакетов в Wireshark

При подключении к линии связи RS-232/RS-485 обычно используется какая-то терминальная программа (Putty, wTerm, Terminal 1.9b, Hercules Setup Utility – тысячи их). За годы работы в отрасли нам не встретилось ни одной программы, которая бы имела встроенный функционал по расшифровке пакетов Modbus (хотя, возможно, такие всё же есть). Вероятно, это связано с тем, что для Modbus RTU начало и конец кадра определяются интервалами тишины на линии связи и разбирать пакеты надо в реальном времени, работая с COM-портом на довольно низком уровне.

Но есть замечательный онлайн-парсер от разработчиков Rapid SCADA – мы часто используем его при отладке обмена.

Rapid SCADA Modbus Parser

Protocol:
- (●) Modbus RTU
- ( ) Modbus TCP

Data Direction:
- (●) Request
- ( ) Response

Data Package (Application Data Unit):

```
01 05 00 64 FF 00 CD E5
```

Parse

Part of Data Package	Description	Value
01	Slave address	0x01 (1)
05	Function code	0x05 (5) - Write Single Coil
00 64	Output address	Physical: 0x0064 (100) Logical: 0x0065 (101)
FF 00	Output value	On
CD E5	CRC	0xCDE5 (52709)

Рис. 1.21. Разбор Modbus RTU пакетов в Rapid SCADA Modbus Parser

2. Modbus RTU

2.1. Паузы между символами и фреймами ($t_{1.5}$ и $t_{3.5}$)

Одним из самых сложных аспектов реализации протокола Modbus RTU является контроль паузы между байтами и фреймами. В соответствии со спецификацией ([2], п. 2.5.1.1) передаваемые фреймы (пакеты) должны разделяться интервалом тишины определенной длительности на линии связи. В спецификации этот интервал называется «паузой между фреймами» (*inter-frame delay*) и обозначается как $t_{3.5}$ (о расчете его длительности мы поговорим ниже). Кроме того, фрейм должен передаваться в виде непрерывной последовательности символов. Если в течение интервала времени $t_{1.5}$ (в спецификации он назван «межсимвольным таймаутом» (*inter-character time-out*)) не приходит новых символов, то устройство может приступать к его проверке на корректность. Если же после истечения времени $t_{1.5}$ и до истечения времени $t_{3.5}$ приходит хотя бы один новый символ – то данный фрейм должен считаться некорректным и не обрабатываться устройством (при этом все символы, полученные после истечения межсимвольного таймаута, также отбрасываются, и началом нового фрейма считается первый символ, полученный после выдержки паузы между фреймами).

Под «символом» в данном случае подразумевается один байт фрейма с «обвязкой» – стартовым битом, стоповым битом и битом четности. Согласно спецификации ([2], п. 2.5.1) в протоколе Modbus RTU один байт данных передается в виде 11 бит (8 бит данных и уже упомянутые 3 служебных бита; при этом бит четности используется даже в случае отсутствия контроля четности).

Длительность паузы между фреймами должна составлять <u>не менее</u> интервала времени, который требуется для передачи 3.5 символов для используемой скорости обмена. Формула расчета выглядит следующим образом:

$$t_{3.5}(\text{мс}) = \frac{3.5 \cdot 11}{\text{скорость (бит/с)}} \cdot 1000$$

Например, для скорости 9600 бит/с минимальная длительность паузы между фреймами составит ≈ 4 мс.

Межсимвольный таймаут вычисляется аналогичным образом:

$$t_{1.5}(\text{мс}) = \frac{1.5 \cdot 11}{\text{скорость (бит/с)}} \cdot 1000$$

Если скорость обмена <u>превышает</u> 19200 бит/с, то спецификация рекомендует вместо формул использовать константы: **0.750 мс** для $t_{1.5}$ и **1.750 мс** для $t_{3.5}$. Это связано с тем, что на высоких скоростях обмена при расчете интервалов времени по формулам, прерывания аппаратных таймеров будут срабатывать слишком часто (например, для скорости 115200 бит/с интервал $t_{3.5}$ составлял бы ≈ 0.33 мс), что может мешать выполнению других задач.

Первый вопрос, который стоит обсудить – зачем вообще нужны два интервала времени ($t_{1.5}$ и $t_{3.5}$) и почему нельзя обойтись одним?

Как уже упоминалось, межсимвольный таймаут используется для определения корректности фрейма. По его истечении устройство может переходить к проверкам – проверке на четность, проверке CRC и проверке адреса (совпадения адреса устройства с адресом в запросе при проверке на стороне slave'а и проверке на совпадение адреса в ответе и адреса в запросе при проверке на стороне master'а). Если эти проверки прошли успешно и была выдержана пауза между фреймами – то считается, что фрейм получен полностью и устройство может переходить к его разбору (выделению из него данных). Кроме того, пауза между фреймами создает искусственную задержку, которая не позволяет slave'у «слишком быстро» ответить на запрос, а master'у – «мгновенно» отправить новый запрос после получения ответа на предыдущий. Необходимость этой задержки вызвана историческими причинами – в минувшие времена практически ни одно устройство не могло достаточно быстро переключиться из режима приема в режим передачи (или наоборот). Впрочем, такие устройства выпускаются и в наши дни.

Ниже приведена диаграмма состояний устройства при разборе пакета Modbus RTU, на которой и основывается приведенный выше текст ([2], п. 2.5.1.1, рис. 14):

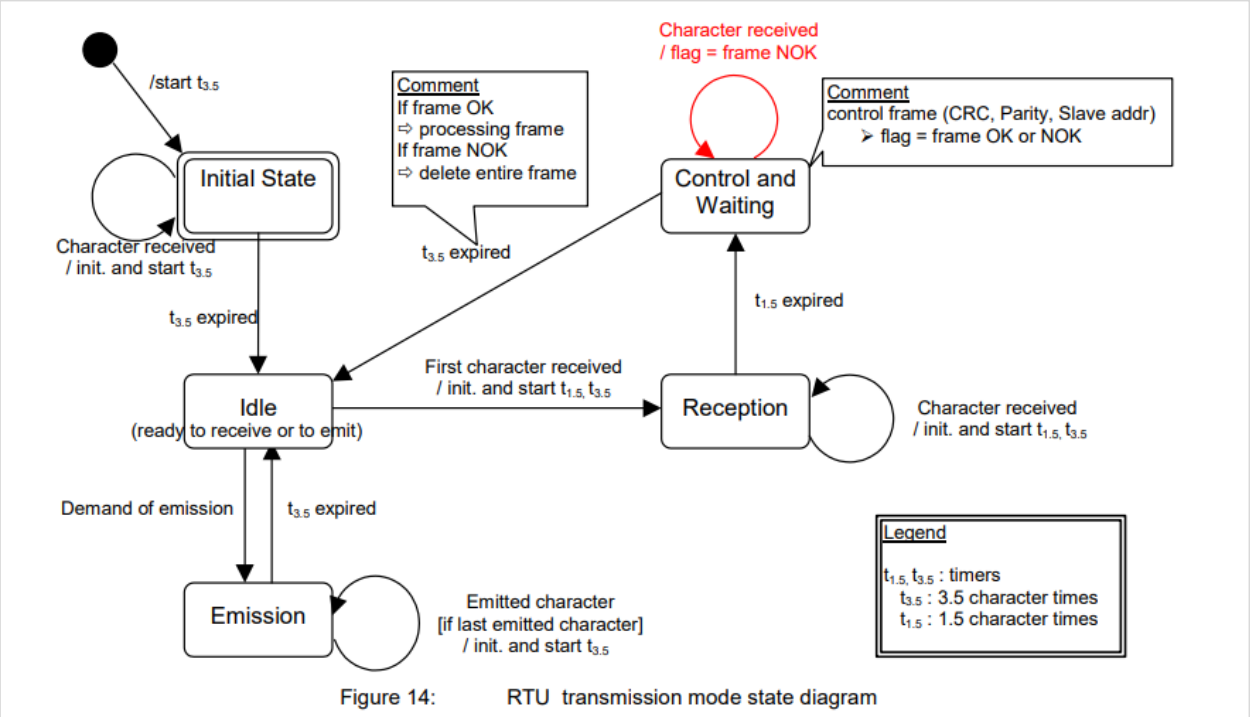


Рис. 2.1. Диаграмма состояний устройства при разборе пакета Modbus RTU

Как уже упоминалось – наличие в протоколе этих интервалов времени во многом вызвано историческими причинами. Надо отметить, что и в наши дни можно встретить устройства, которые передают символы с определенными задержками. К тому же, часто у бюджетных микроконтроллеров нет специальных прерываний для этих интервалов, что вынуждает программистов реализовывать «программные» таймеры (дискретность которых в зависимости от устройства может составлять единицы миллисекунд или даже больше). С другой стороны, развитие

аппаратных ресурсов позволяет анализировать фреймы «на лету», при получении каждого нового символа. В этом случае после получения первых нескольких символов уже становится ясна предполагаемая длина пакета (она зависит от кода функции и числа бит/регистров в запросе), и определить конец фрейма можно после получения последнего ожидаемого символа и проверки CRC.

Ниже приведены отдельные цитаты по этой теме.

Детальный комментарий от **Lynn August Linse** – инженера, разрабатывающего устройства с поддержкой Modbus с начала 90-х годов (источник):

«*When transmitting, make sure your device can send data with less than 1.5 character gap between bytes. This may mean you have the serial transmit priority very high - perhaps even higher than Ethernet etc. This is generally not hard to do.*

When receiving, generally you should NOT enforce a strict 3.5 character timeout - you'll find it causes more support problems than 'the better way'. Many wireless/radio modems and wide-area-network solutions will NOT work if you succeed in doing a perfect 3.5 character (or 3.5 millisecond at 9600 baud) end-of-message detection. This is because they 'packetize' data into fixed blocks of perhaps 32 or 128 bytes and send these with error correction information, retries, and token rotation between blocks. This means you may see a 5 or 25 msec gaps within your Modbus message. With wide-area-network (like satellite) this can be even more extreme with gaps up to 1-second! Plus it is nearly impossible to measure on a modern system with UART FIFOs, heavy OS overhead and stock serial drivers. Trying to do a 4 msec gap detect on a stock Windows serial driver will lead to long delays after every message.

The better way is to add some Modbus protocol knowledge into your driver. For example if you are a slave seeing a function code 3 request, then after the second byte (the func 3) you'll know you should receive a total of 8 bytes with the CRC. If after seeing 8 bytes the CRC is valid, then you know that you have a complete message and no 3.5 character pause or gap is required. If you are a masters seeing a function code 3 response, then after the third byte (the data length) you'll know you should receive a total of 5 bytes plus the data length value. If after seeing this number of bytes the CRC is valid, then you know that you have a complete message and no 3.5 character pause or gap is required.

Why is this smarter? Because it allows you to have more tolerant, customer-friendly end-of-message timeout of 50 msec. This will NOT slow down your Modbus - in fact it will be faster than trying to use a 3.5 char gap! With the combination of message-length estimation and a 50 millisecond timeout, once you see the expected number of bytes and a valid CRC you are done without the need for the 50 msec delay. You'll get by with NO added delay. The 50 msec delay only is used when a CRC error is being detected or you are seeing a Modbus function you do not support. Compare this to the situation with the 3.5 character (4-msec at 9600 baud). You will always add about 5msec of dead time to every message - and if you find yourself talking over a wireless or wide-area-network system that adds gaps and pauses you'll have either a large error rate or no communications at all.

Yes, this violates the SACRED ISO/OSI 7-layer model by making your "Link-Layer" application aware. Whoppity-do I say. In the real world - solve your problems as you must. This estimation method gives you the best throughput and performance, with the least support head-aches. It will work with hardware that demands a 3.5 character timeout, as well as hardware and media that cannot deliver such tight timing.

Just make sure you allow your users to adjust this time delay. The 50 msec I mention is a good default, but for wide-area-network systems delays up in the 1-2 second range are possible. I prefer a range of about 5 to 5000 msec.»

Перевод:

«*При передаче данных убедитесь, что ваше устройство способно отправлять данные с межсимвольным таймаутом менее $t_{1.5}$. Это может означать, что для задачи, в которой выполняется передача данных, будет установлен очень высокий приоритет – возможно, даже выше, чем у Ethernet и т.д. Обычно это несложно организовать.*

При приеме данных обычно НЕ СЛЕДУЕТ устанавливать таймаут для интервала времени $t_{3.5}$ – вы обнаружите, что это доставляет гораздо больше проблем, чем «правильный способ». Многие решения, основанные на использовании шлюзов и беспроводных/радиомодемов не будут работать, если вам удастся организовать идеально точное определение истечения времени $t_{3.5}$ (или 3.5 мс для скорости 9600). Это связано с тем, что в таких системах происходит фрагментация данных на блоки (например, размером 32 или 128 байт), которые включают в себя информацию для исправления ошибок, могут отправляться повторно при отсутствии получения о подтверждении, могут содержать периодически меняющиеся токены и т.д. Это может привести к тому, что пауза между символами будет составлять 5 или даже 25 мс. При передаче данных через интернет (например, через спутниковую связь) задержки будут еще значительнее и могут составлять даже секунду! Кроме того, точно измерять такие интервалы времени практически невозможно на современных системах с их реализацией FIFO на UART, накладными расходами ОС и стандартными драйверами последовательных портов. Попытка детектировать интервал длиной 4 мс на стандартном драйвере COM-порта Windows приведет к большим задержкам после получения каждого фрейма.

Правильный способ – добавить некоторые обработки фреймов Modbus прямо в ваш драйвер последовательного порта. Например, если вы реализуете slave и получаете запрос с кодом функции 0x03, то уже после получения 2-го байта (который как раз содержит код функции) вы знаете длину всего фрейма – она составит 8 байт (с учетом CRC). После получения 8 байт вы можете проверить CRC, и если она корректна – то вам не требуется ждать еще истечения времени $t_{3.5}$. Если вы реализуете master-устройство и получаете ответ с кодом функции 0x03, то после получения 3-го байта (в котором содержится число байт данных в фрейме) вы знаете длину фрейма – она будет равна значению этого байта плюс 5 (по одному байту на адрес устройства, код функции и число байт данных плюс два байта CRC). После получения этого числа байт вы можете проверить CRC, и если она корректна – то у вас в буфере полный фрейм, и вам опять же не следует ждать еще $t_{3.5}$.

Почему лучше делать именно так? Потому что это позволяет определять конец фрейма по более приемлемому и удобному для клиента таймауту длиной в 50 мс. Это НЕ ЗАМЕДЛИТ ваш обмен по Modbus – фактически, всё будет работать даже быстрее, чем при точном детектировании времени $t_{3.5}$! Так как вы «на лету» рассчитывает ожидаемую длину фрейма, то при получении нужного количества байт и подтверждении корректности CRC вам не потребуется ждать еще 50 мс. Вам НЕ ПОТРЕБУЕТСЯ дополнительная задержка. Таймаут в 50 мс нужен только тогда, когда вы получаете фрейм с некорректной CRC или, например, запрос с неподдерживаемым вами кодом функции. Сравните это с задержкой $t_{3.5}$ (напомню, 4 мс на

скорости 9600). Вы всегда будете ждать лишние 4-5 мс при получении каждого сообщения – и если протестируете это в системах, которые работают через интернет или беспроводную связь, в которой добавляются свои задержки и пропуски, то обнаружите, что у вас будет очень много ошибок или связь вообще будет отсутствовать.

Да, предложенный мной способ нарушает «священную» модель OSI/ISO, потому что часть функций уровня приложений переносится на канальный уровень. «Ну что же» – скажу я. – «В реальном мире нужно решать практические проблемы, а не концептуальные». Этот способ обеспечивает наилучшую пропускную способность и производительность с минимальной головной болью для разработчика. Он подойдет как для устройств, которым требуется пауза между фреймами, так и для тех приборов, которые не способны детектировать такие малые интервалы времени.

Только убедитесь, что вы даете пользователям возможность настройки таймаута. Упомянутое мной значение в 50 мс подходит в качестве значения по умолчанию, но для систем, работающих через интернет, задержки могут составлять 1-2 секунды. Я предпочитаю, чтобы таймаут можно было настраивать в диапазоне 5…5000 мс».

Комментарий от Jerry Miill, инженера компании Miille Applied Research Co., Inc, на форуме Control Automation (источник):

«*…You have the final numbers right, but I have to tell you that almost no one pays strict attention to them. You will find lots of applications that have inter-character delays of longer than 1.5 character times, even longer than 3.5 character time in some cases. You will need to build in a way to override the specification numbers.*

Think about this: if two devices are attempting to communicate using Modbus RTU and BOTH really use the 3.5 character time as a framing mark then the communication will never work because it is impossible to synchronize to EXACTLY 3.5 character time between two independent systems. That is why the framing time is specified to be 3.5 character time OR LONGER.

Or suppose your connection is over a modem or a satellite. There could be significant delays inserted in the bit stream that will exceed the specifications that are beyond your control. Build in a way to adjust this time.

Most modern day microcontrollers have very precise timers that can be used to set the delays but I repeat, build in a way to adjust the timing if you want a product that will work everywhere.»

Перевод:

«…Вы правильно рассчитали значения таймаутов, но я должен сказать, что почти никто не обращает на них серьезного внимания. Вы можете встретить множество систем, где межсимвольный таймаут превышает время $t_{1.5}$, а в некоторых случаях даже время $t_{3.5}$. Поэтому вам нужно хорошо подумать, какие значения использовать.

Учтите вот что: если два устройства используют протокол Modbus RTU и ОБА используют значение $t_{3.5}$ для определения конца фрейма, то связи между ними просто не будет,

потому что невозможно ТОЧНО синхронизировать таймеры между двумя независимыми приборами. Поэтому в спецификации указано, что пауза между фреймами должна составлять НЕ МЕНЕЕ $t_{3.5}$, а не ровно $t_{3.5}$.

Или, предположим, обмен между устройствами настроен через модемы или спутниковую связь. В этом случае при передаче данных возможны значительные задержки, которые превышают значения $t_{1.5}$/$t_{3.5}$ и на которые вы никак не можете повлиять. Поэтому создайте (для пользователя) механизм настройки значений таймаутов для конкретных ситуаций.

У большинства современных микроконтроллеров довольно точные таймеры, но, повторюсь, если вы разрабатываете устройства, которые будут применяться в разных системах – то дайте возможность пользователям вручную настраивать эти таймауты.»

Комментарий от пользователя **sawdust** из обсуждения на stackoverflow.com (источник):

«*UARTs do not have a capability to meter its output by inserting a delay between the transmission of character frames. Adding such a delay is an additional processor burden as well as the use of a timer. On the contrary UARTs have evolved to transmit characters as fast as the baudrate allows with the least processor intervention, e.g. hardware FIFO and DMA. <…> A microprocessor or microcontroller should be able to keep a UART busy and transmit without any intercharacter gaps. A UART that requires gaps during receiving is IMO an overloaded system and is broken. For reliable communication (without flow control) use a baudrate low enough so that metering the transmitted characters is not necessary.»*

Перевод:

«*UART не имеет возможности вставлять задержку между фреймами. Добавление этой задержки требует использование таймера и увеличивает нагрузку на процессор. С другой стороны, современные микросхемы UART позволяют передавать данные с максимально возможной скоростью с минимальным использованием ресурсов процессора – например, с помощью аппаратной реализации FIFO или DMA. <…> Микропроцессор или микроконтроллер не должен прерывать работу UART и обеспечить передачу данных без каких-либо пауз между символами. UART, который требует пауз во время приема, на мой взгляд, не является работоспособным решением. Для обеспечения надежной связи (при отсутствии управления потоком) используйте низкую скорость обмена, чтобы избежать необходимости контроля каких-либо интервалов времени.»*

Комментарий от пользователя **Timmy_A** из обсуждения на stackoverflow.com (источник):

«*You can't use timeouts. On higher baud rates 3.5 character timeout means a few milliseconds, or even hundreds of microseconds. Such timeouts can't be handled in the Linux user space.*

On the client side, it isn't a big deal since Modbus doesn't send asynchronous messages. So it's up to you not to send 2 consecutive messages within 3.5 character timeout.

On the server side, the problem is that if your clients have an extremely short response timeouts and Linux is too busy you can't write a bullet-proof framing solution. There is a chance that read() function will return more than one packet. Here is (a little contrived) example.

- *Client writes a packet to server. Timeout is let's say 20 ms.*
- *Let's say that Linux is at the moment very busy, so kernel doesn't wake up your thread within next 50 ms.*
- *After 20 ms client detects that it didn't receive any response so it sends another packet to server (maybe resent the previous one).*
- *If Linux wakes up your reading thread after 50 ms, read() function can get 2 packets or even 1 and half depending to how many bytes were received by the serial port driver.*

In my implementation I use a simple method that tries to parse bytes on-the-fly – first detecting the function code and then I try to read all remaining bytes for a specific function. If I get one and half packet I parse just the first one and remaining bytes are left in the buffer. If more bytes come within a short timeout I add them and try to parse, otherwise I discard them. It's not a perfect solution (for instance some sub-codes for function 8 doesn't have a fixed size) but since MODBUS RTU doesn't have any STX ETX characters, it's the best one I were able to figure out.»

Перевод:

«*Вы не можете использовать эти таймауты. На высоких скоростях обмена значение* $t_{3.5}$ *будет составлять единицы (или даже доли) миллисекунд. Такие интервалы времени невозможно обрабатывать в пользовательском пространстве Linux.*

На стороне master-устройства это не имеет большого значения, так как протокол Modbus является синхронным (запрос-ответ) и за время отсчета паузы между фреймами не может прийти более одного фрейма.

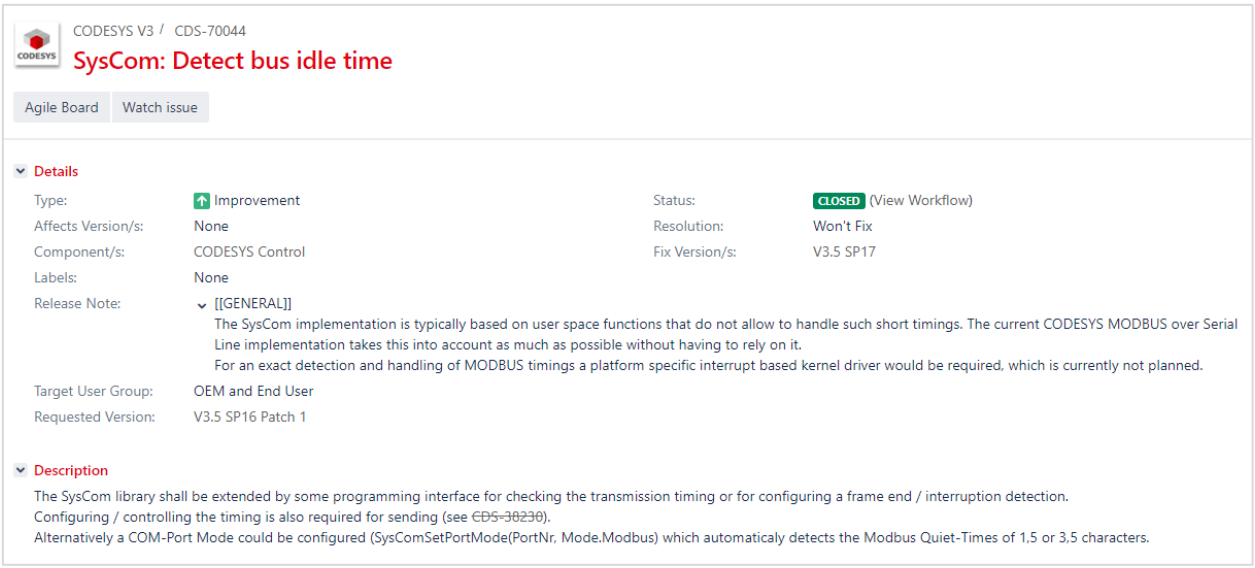
На стороне slave'а возможна проблема в том случае, если master-устройства имеют очень малые таймауты ответа, а Linux слишком загружен – и у вас нет вариантов для надежного определения конца фрейма. В этом случае есть вероятность, что вызов функции read() вернет содержимое буфера, в котором уже несколько фреймов. Вот синтетический пример:

- *Master-устройство отправляет запрос. Таймаут ожидания ответа – 20 мс;*
- *Предположим, Linux сейчас занят, и ядро не пробудит ваш поток в течение следующих 50 мс;*
- *Спустя 20 мс master-устройство детектирует отсутствие ответа и отправляет новый запрос (или повторяет предыдущий);*
- *Если Linux пробудит ваш поток спустя 50 мс, то вызов функции read() может вернуть 2 или даже 1.5 фрейма – в зависимости от того, сколько байт было получено драйвером последовательного порта.*

В своих устройствах я просто осуществляю разбор фрейма «на лету» – сначала определяю код функции, затем вычисляю предполагаемую длину фрейма и считываю из буфера именно столько байт, сколько нужно. Если я получаю 1.5 фрейма, то разбираю только первый и оставляю начальные байты второго в буфере. Если в течение короткого промежутка времени

придут еще байты – я пытаюсь разобрать второй фрейм; если они не придут – то очищаю буфер. Это не идеальное решение (например, для некоторых подфункций функции 0x08 размер фрейма невозможно определить по его начальным байтам), но так как протокол Modbus RTU не имеет стоп-символов – это наилучший из известных мне способов.»

Комментарий от разработчиков CODESYS V3.5 на запрос реализации обработки межсимвольного таймаута и времени между фреймами в полном соответствии со спецификацией Modbus (см. *Release Note* и *Resolution: Won't Fix* – не запланировано к исправлению).


CODESYS V3 / CDS-70044

SysCom: Detect bus idle time

Agile Board | Watch issue

Details

Type:	Improvement	Status:	CLOSED (View Workflow)
Affects Version/s:	None	Resolution:	Won't Fix
Component/s:	CODESYS Control	Fix Version/s:	V3.5 SP17
Labels:	None		

Release Note: [[GENERAL]]
The SysCom implementation is typically based on user space functions that do not allow to handle such short timings. The current CODESYS MODBUS over Serial Line implementation takes this into account as much as possible without having to rely on it.
For an exact detection and handling of MODBUS timings a platform specific interrupt based kernel driver would be required, which is currently not planned.

Target User Group: OEM and End User

Requested Version: V3.5 SP16 Patch 1

Description

The SysCom library shall be extended by some programming interface for checking the transmission timing or for configuring a frame end / interruption detection.
Configuring / controlling the timing is also required for sending (see ~~CDS-30230~~).
Alternatively a COM-Port Mode could be configured (SysComSetPortMode(PortNr, Mode.Modbus) which automaticaly detects the Modbus Quiet-Times of 1,5 or 3,5 characters.

Рис. 2.2. Комментарий от разработчиков CODESYS V3.5

Надо отметить, что многие master-устройства поддерживают задание длительности паузы между фреймами в явном виде:

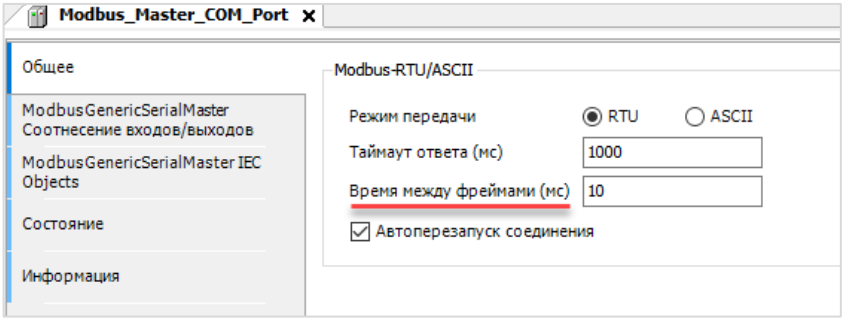


Рис. 2.3. Установка длительности паузы между фреймами в CODESYS V3.5

2.2. Широковещательная рассылка (broadcast)

Протокол Modbus RTU поддерживает широковещательную (broadcast) рассылку ([2], п. 2.1). Для широковещательных команд зарезервирован адрес **0**. Запрос, отправляемый master-устройством на адрес 0, получают все slave-устройства, подключенные к данной линии связи. Slave-устройства не отвечают на широковещательный запрос (в рамках последовательного интерфейса связи это привело бы к возникновению коллизий), поэтому в большинстве случаев в широковещательных запросах используют только функции записи (0x05, 0x06, 0x0F, 0x10). В качестве исключения можно привести пример (*не описанный в спецификации*), когда в ответ на широковещательный запрос с функцией чтения slave-устройство возвращает свой адрес (*подразумевается, что на этапе конфигурирования на линии связи находится только одно устройство, чтобы избежать коллизий*).

Спецификация уточняет ([2], п. 2.4.1), что хотя ответа от slave-устройства в случае широковещательной рассылки не ожидается, master-устройство должно выдержать паузу (*turnaround delay*) перед отправкой следующего запроса, чтобы дать slave'ам время на обработку полученных данных. Спецификация рекомендует для этой задержки значение в 100-200 мс.

Несмотря на то, что отсутствие ответа на широковещательный запрос является корректной ситуацией, некоторые master-устройства генерируют в этом случае ошибку таймаута ответа.

Обычно широковещательная рассылка используется для синхронного обновления данных на группе приборов – потому что при значительном числе устройств и особенностях линии связи (большая длина, низкая скорость обмена и т.д.) суммарное время на отправку «индивидуальных» команд будет довольно значительным. Протокол Modbus не специфицирует, как можно с помощью широковещательной рассылки записать в отдельные приборы разные данные. Некоторые устройства (например, индикатор ОВЕН СМИ2-М) имеют специальные настройки, позволяющие выделить из данных широковещательного запроса именно тот набор байт, который предназначается этому конкретному прибору.

В некоторых устройствах обработка широковещательной рассылки не реализована и адрес 0 является «обычным» - то есть, например, может быть присвоен slave'у (что является отступлением от спецификации). В качестве примера можно привести контроллеры Segnetics (см. здесь и здесь):

Описание элементов диалога "Свойства слейва":

1. Название. Имя слейва, которое отображается в дереве устройств и на сетевых переменных.

2. Адрес. Адрес Modbus слейва. Значение от 0 до 255.

24.09.2012, 10:46

Arsie
Сотрудник Segnetics

Регистрация: Jan 2006
Адрес: Russia, SPb
Сообщения: 13 288
Благодарил(а): 4 раз(а)
Поблагодарили: 239 раз(а) в 219 сообщениях

Ответ: СМИ2

Цитата

Сообщение от **LordN**
не могу понять как на смлоджике организовать такую широковещательную посылку.
это вообще возможно впринципе?

Для наших контроллеров адрес 0 является таким же обычным адресом как и любой другой. Т.е. контроллер отправляет запрос и ждёт ответ.

В принципе, вы можете создать слейва с нулевым адресом и поставить тайм-аут 1 мсек, этим вы обеспечите минимально возможные задержки в работе мастера.

Рис. 2.4. Скриншот из документации контроллеров Segnetics и комментарий сотрудника техподдержки

2.3. Надежность CRC

Для проверки целостности фрейма в протоколе Modbus используется CRC-16-IBM (также иногда называемая CRC-16-ANSI) с порождающим полиномом $x^{16} + x^{15} + x^2 + 1$. Алгоритм расчета контрольной суммы приведен в [2], п. 6.2.2.

Контрольная сумма является 16-битным значением и, таким образом, существует всего 65536 ее возможных вариантов. Очевидно, что число вариаций Modbus-запросов (с разными адресами slave'ов, кодами функций, начальными адресами, числом регистров и данными) значительно больше, поэтому для разных пакетов контрольная сумма может совпадать. Кроме того, в некоторых ситуациях корректная контрольная сумма не гарантирует корректности фрейма. Пользователь Хабра *Polaris99* описывает в своей статье Надежность контрольной суммы CRC16 следующую ситуацию (*речь идет не конкретно о Modbus, а о другом протоколе с использованием того же алгоритма расчета CRC; впрочем, такая ситуация могла произойти и при использовании Modbus*):

«*Время от времени в буфере, где собирались полученные данные, оказывался пакет, состоящий из конца предыдущего пакета и начала следующего, причем контрольная сумма у такого комбинированного пакета оказывалась верной. То есть, налицо коллизия контрольной суммы: пакет не имеет смысла, но дает верную контрольную сумму*.»

В статье описывается следующий эксперимент: 100.000.000 сгенерированных фреймов специальным образом повреждались (путем сдвига произвольного числа бит/байт, заполнением произвольного числа байт значениями 0x00 или 0xFF, и другими способами), после чего происходила проверка контрольной суммы. Если для такого поврежденного пакета контрольная сумма оставалось корректной, то это считалось коллизией. Результаты эксперименты приведены в таблице:

Обозначение	Number of collisions	Place
CMS	24099	8
CCITT	12728	3
CDMA2000	10662	2
DECT-X	15412	6
EN-13757	15000	5
Modbus	23845	7
T10-DIF	9812	1
TELEDISK	14031	4

Рис. 2.5. Результаты эксперимента проверки надежности для различных алгоритмов CRC

Как можно заметить, по надежности CRC Modbus показал один из худших результатов (7-е место из 8).

Схожую ситуацию описывает пользователь ЖЖ *Vostnod* в посте 2 нюанса использования Modbus RTU:

«Звёзды сошлись так, что проверка CRC в верификаторе посылки Modbus RTU протокола дала сбой.

Это сообщение проходит проверку CRC, но некорректно:

0x04, 0x03, 0x1C, 0x3F, 0xE9, 0x15, 0x18, 0x03, 0x02, 0x00, 0x01, 0x02, 0x02, 0x00

Вот корректное сообщение:

0x04, 0x03, 0x1C, 0x3F, 0xE9, 0x15, 0x18, 0x03, 0x02, 0x00, 0x01, 0x02, 0x02, 0x00, 0x03, 0x18, 0x02, 0x00, 0x03, 0x0B, 0x06, 0xC6, 0x39, 0x07, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0D, 0xEE

*Это не проблема протокола Modbus, т.к. он создавался для медной линии связи *дцать лет назад и в нем концом посылки является пауза между байтами длиннее, чем 1.5 байта. В наше время Modbus RTU часто используют через TCP\IP соединение (преобразователи MOXA и т.п.), а в этом случае временные промежутки становятся случайным.*

Вывод: надо проверять не только CRC, а еще и длину посылки в сообщении (она не всегда есть, надо смотреть тип сообщений).

И еще есть одна, уже не исправляемая, проблема Modbus: в ответной посылке не фигурирует номер регистра из запроса.

Может возникнуть следующая ситуация: спросили адрес 1000, ответа от устройства за N мс нет, спрашиваем адрес 1001, приходит ответ на предыдущий запрос (1000), который интерпретируется ответом на 1001й. Страховки от такой ситуации нет. Минимизация: установка таймаута больше, чем возможная задержка в TCP соединении. Однако при GPRS подключении (GSM модем) она может быть сколь угодно большой.

Вывод: не использовать Modbus RTU/TCP в сетях, где нет гарантированного времени ответа.»

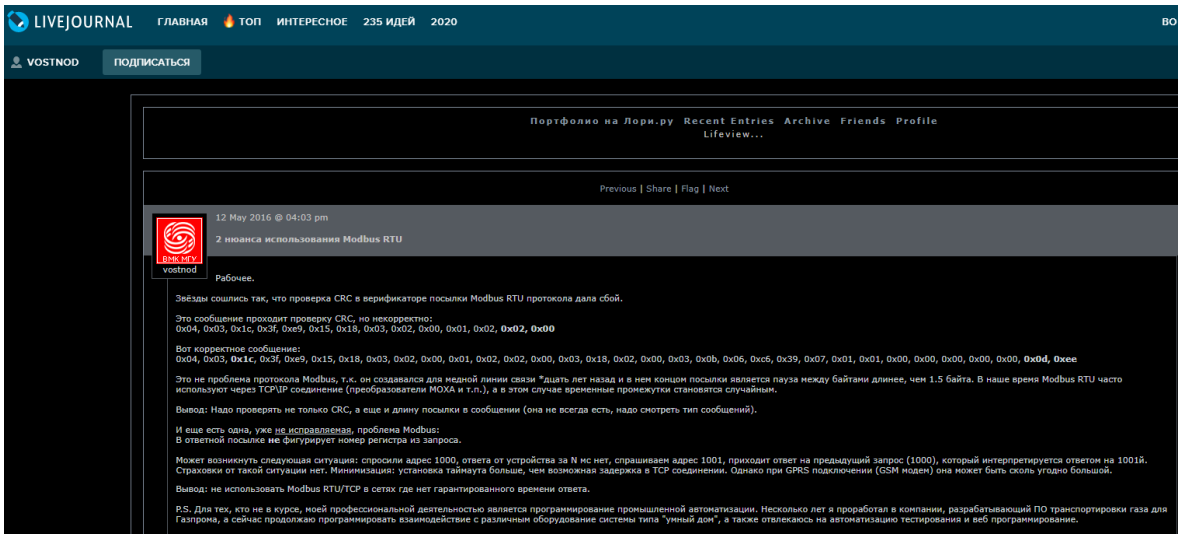


Рис. 2.5. Скриншот поста 2 нюанса использования Modbus RTU

2.4. Порядок байт в CRC

Согласно спецификации (см. рис. 12 и п. 2.5.1.2 в [2]) контрольная сумма в Modbus передается младшим байтом вперед. Иногда это становится неожиданностью для разработчиков, потому что значительное число инструментов расчета CRC возвращает контрольную сумму с другим порядком байт (старшим байтом вперед).

Например, в этом онлайн-калькуляторе для пакета **01 03 00 00 00 01** будет рассчитана следующая контрольная сумма:

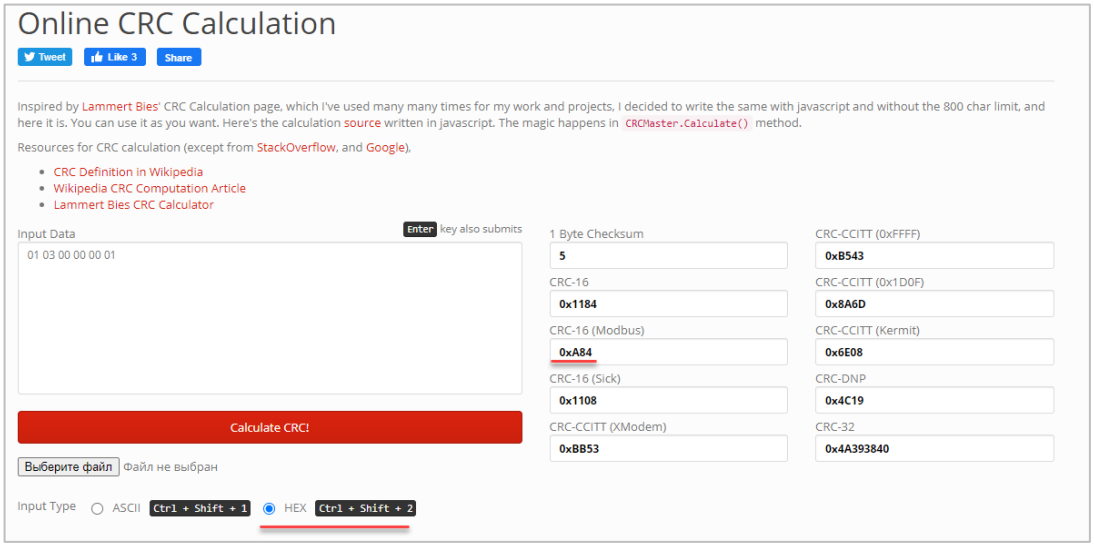


Рис. 2.6. Пример расчета контрольной суммы в онлайн-калькуляторе – значение возвращается с порядком байтом *Старшим байтом вперед*

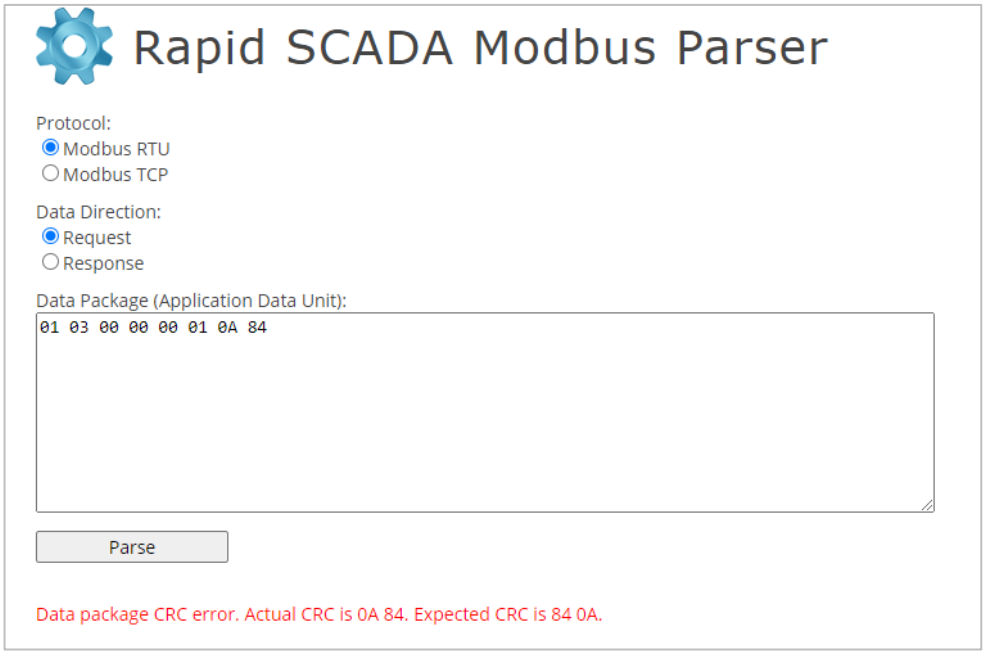


Рис. 2.7. Проверка фрейма в онлайн-парсере – видно, что контрольная сумма должна передаваться младшим байтом вперед

2.5. Требования к линии связи (терминаторы, подтяжка и т.д.)

Многие пользователи не обращают внимания (или даже не знают), что спецификация Modbus описывает требования к физическому уровню.

В частности, для 2-проводного интерфейса RS-485:

- на концах линии связи **должны** устанавливаться терминирующие резисторы ([2], п. 3.4.5). Это позволяет предотвратить отражение сигнала от конца линии связи. Спецификация рекомендует использовать резистор с сопротивлением 150 Ом (0.5 Вт). При наличии подтяжки линии рекомендуется использовать конденсатор[2] (с емкостью 1 нФ и напряжением не менее 10 В) и резистор с сопротивлением 120 Ом (0.25 Вт);
- в некоторых случаях может потребоваться использование резисторов подтяжки. Это связано с тем, что при отсутствии обмена линия связи находится в «подвешенном» состоянии (не управляется ни одним из устройств). Поэтому наличие помех на линии может быть воспринято устройствами как получение некорректных пакетов, что, в свою очередь, может повлиять на производительность (из-за частых прерываний на UART и т.д.). Для предотвращения этой ситуации используются резисторы подтяжки. Спецификация определяет ([2], п. 3.4.6), что на линии связи должна быть установлена только одна пара таких резисторов (обычно – на интерфейсе master-устройства), один из которых будет «подтягивать» линию A к земле, а второй – линию B к 5 В. Согласно спецификации, сопротивление этих резисторов должно выбираться из диапазона 450…650 Ом (чем больше устройств на линии связи – тем выше должно быть сопротивление). Некоторые устройства имеют встроенные резисторы подтяжки (иногда – включаемые аппаратно или программно) – информация об этом должна быть приведена в их документации;
- экран кабеля **должен быть** ([2], п. 3.4.4) соединен с защитной землей в одной точке (обычно соединение выполняется на интерфейсе master-устройства).

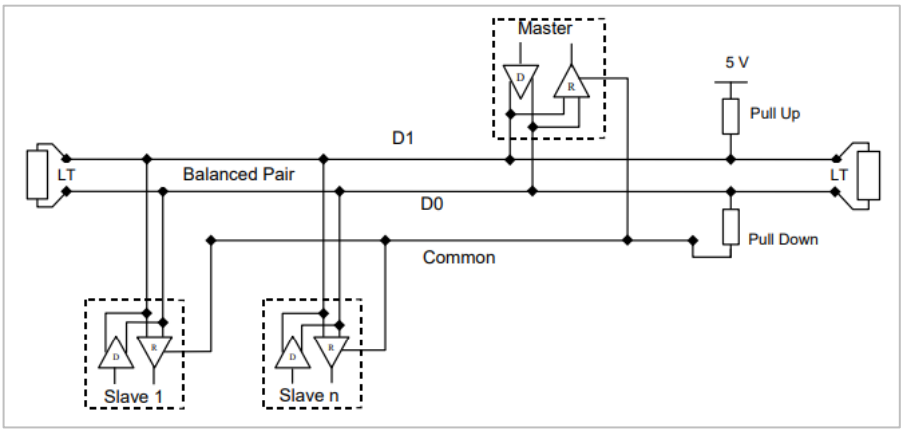


Рис. 2.8. Организация линии связи для 2-проводного интерфейса RS-485 (рис. 20 из [2])

[2] Рекомендация по установке конденсатора, на наш взгляд, носит полемический характер, поскольку в ряде случаев (низкие скорости, пакеты с большим числом бит в состоянии «1») это может привести к ухудшению качества связи из-за «загрубления» сигнала на конденсаторе.

3. Modbus ASCII

3.1. Преимущества и недостатки

Преимуществом протокола Modbus ASCII по сравнению с Modbus RTU является наличие стартового и стопового символа, что позволяет избежать проблем с определением конца фреймов, описанных в п. 2.1 – особенно если устройство не имеет точных аппаратных таймеров. Кроме того, это позволяет использовать в качестве среды передачи ненадежный канал связи – например, радиоканал, GPRS, спутниковую связь и т.д. – в которых во время передачи фрейма могут возникать задержки в несколько секунд.

Недостатком является больший размер фрейма – размер фрейма Modbus ASCII превышает размер аналогичного фрейма Modbus RTU приблизительно в 2 раза.

3.2. Размер символа (7 бит данных)

В соответствии со спецификацией ([2], п. 2.5.2) каждый символ Modbus ASCII содержит 7 бит данных и 3 служебных бита (стартовый бит, стоповый бит и бит контроля четности).

Тем не менее, некоторые устройства позволяют использовать 8 бит данных для Modbus ASCII, а некоторые – вообще позволяют использовать **только** такой вариант (что, безусловно, является отступлением от спецификации). В этом случае 7-битный ASCII-символ дополняется старшим незначащим битом. Такая реализация в основном встречается при использовании микроконтроллеров, где поддержка передачи 7-битных символов отсутствует или требует дополнительных усилий.

Ниже приведен скриншот из конфигуратора индикатора СМИ2-М компании ОВЕН, на котором видно, что устройство поддерживает только 8 бит данных для протокола Modbus ASCII.

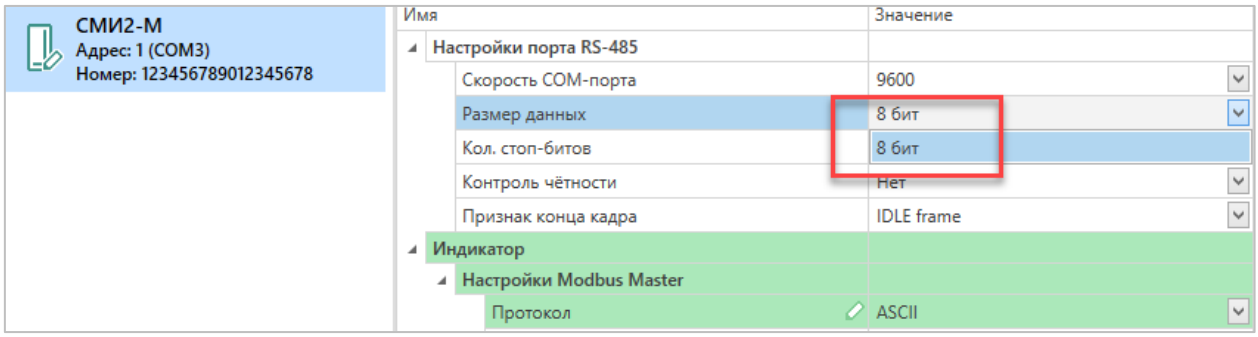


Рис. 2.9. Скриншот конфигурации индикатора ОВЕН СМИ2-М

3.3. Расчет LRC

В протоколе Modbus ASCII в качестве контрольной суммы используется 1-символьная LRC (Longitudinal Redundancy Checking). Принцип ее расчета описан в спецификации ([2], п. 2.5.2.2), но мы наблюдали ситуации, когда начинающие разработчики не могли корректно интерпретировать этот текст, поэтому хотели бы привести конкретный пример.

Пусть имеется фрейм без рассчитанной LRC и стоп-символа:

:0A03000C0001

(запрос к slave'у с адресом 10 с кодом функции 0x03 для чтения регистра 12 (0x000C), число считываемых регистров – 1 (0x0001)).

Расчет LRC происходит следующим образом:

- выполняется сложение всех чисел (не ASCII-кодов!), соответствующих байтам фрейма, за исключением стартового символа (*при этом результат занимает байт, так что необходимо учитывать переполнение*):
  0x0A + 0x03 + 0x00 + 0x0C + 0x01 = 0x1A;
- полученное на предыдущем шаге значение вычитается из 0xFF:
  0xFF – 0x1A = 0xE5;
- к полученному на предыдущем шаге значению прибавляется 0x01:
  0xE5 + 0x01 = 0xE6;

Таким образом, полный фрейм будет выглядеть так:

:0A03000C0001E6<CR><LF>

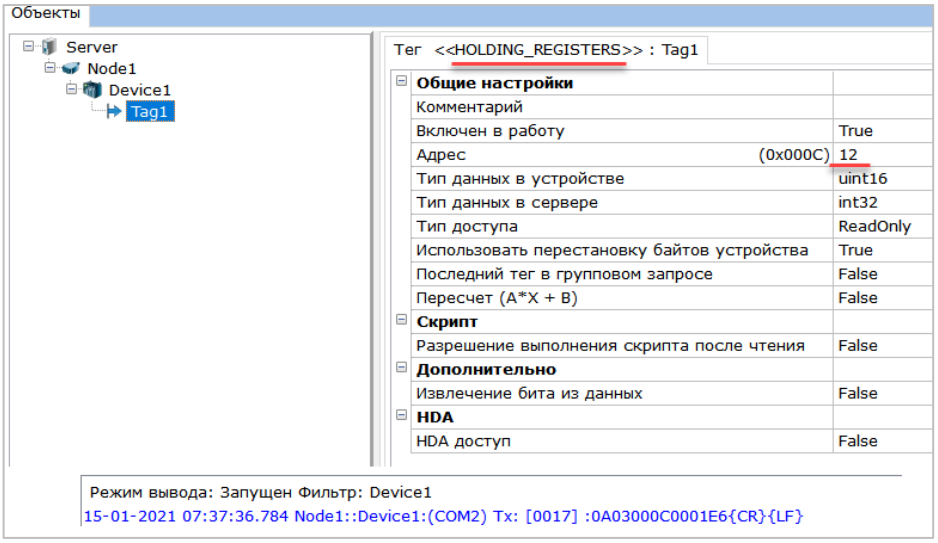


Рис. 2.10. Подтверждение корректности расчета LRC с помощью MasterOPC Universal Modbus Server

Пример кода функции расчета LRC для среды CODESYS V3.5 (используются библиотеки CAA Types, CAA Memory и OwenStringUtils):

```
MB_ASCII_LRC ✕ | Library Manager

1   FUNCTION MB_ASCII_LRC : STRING(2)
2   VAR_INPUT
3       pData:  CAA.PVOID;
4       szSize: CAA.SIZE;
5   END_VAR
6   VAR
7       i:          INT;
8       sHexChar:   STRING(2);
9       bySum:      BYTE;
10  END_VAR
11

1   FOR i := 1 TO TO_INT(szSize) BY 2 DO
2       MEM.MemMove(pData + TO_DWORD(i), ADR(sHexChar), 2);
3       bySum := bySum + TO_BYTE(OSU.HEX_STR_TO_WORD(sHexChar, '') );
4   END_FOR
5
6   bySum := 16#FF - bySum;
7   bySum := bySum + 1;
8
9   MB_ASCII_LRC := OSU.WORD_TO_HEX_STR(bySum, TRUE, '');
10
```

Рис. 2.11. Пример кода функции расчета LRC для среды CODESYS V3.5

PLC_PRG ✕

Device.Application.PLC_PRG

Expression	Type	Value
sRequest	STRING(513)	':0A03000C0001'
sLrc	STRING(2)	'E6'

```
1  sLrc 'E6' := MB_ASCII_LRC(ADR(sRequest ':0A03000C0 ▸ ), TO_UDINT(LEN(sRequest ':0A03000C0 ▸ )) ); RETURN
```

Рис. 2.12. Подтверждение корректности работы функции

4. Шлюзы Modbus

4.1. Типы шлюзов

Иногда при построении систем автоматизации возникает потребность в интеграции устройств с Modbus в сети с другими промышленными протоколами. Частным случаем этой задачи является интеграция сетей Modbus RTU и Modbus TCP (например, ПК со SCADA-системой и драйвером Modbus TCP не имеет последовательных портов, но должен выполнять опрос модулей ввода-вывода с интерфейсом RS-485 и протоколом Modbus RTU).

Для решения этих задач используются специальные устройства, называемыми шлюзами. Существует несколько типов шлюзов:

- преобразователи интерфейсов (например, Moxa NPort). Эти устройства относятся к так называемым «прозрачным» шлюзам – они пересылают данные из одного своего интерфейса в другой (обычно интерфейсами являются Ethernet и RS-232/485) без какого-либо преобразования. При работе с такими конвертерами часто используется специальная вариация протокола Modbus, не определенная в его спецификации и называемая *Modbus RTU over TCP*. При использовании этого протокола по TCP передаются фреймы Modbus RTU (отличающиеся от фреймов Modbus TCP отсутствием заголовка MBAP Header и наличием контрольной суммы). Это позволяет избежать необходимости в конвертации фрейма на уровне шлюза;
- преобразователи протоколов (например, ОВЕН МКОН). Эти устройства осуществляют конвертацию протоколов – например, шлюз получает по Ethernet запрос Modbus TCP, преобразует его в фрейм Modbus RTU и отправляет в свой последовательный порт. Соответственно, ответ преобразуется в фрейм Modbus TCP и отправляется по Ethernet обратно master-устройству. На рынке представлены шлюзы Modbus для всех основных промышленных протоколов – CAN, Profibus, Profinet, EtherNet/IP и т.д.;
- каплеры (например, Kyland KGW301x). Каплер представляет собой конфигурируемый коммуникационный контроллер, который использует некоторые из своих интерфейсов для работы в режиме master-устройства, а некоторые – в режиме slave'а. Упомянутый вышел каплер компании Kyland выполняет роль Modbus RTU Master на своих последовательных портах (их может быть от 1 до 4 в зависимости от модификации) и роль Modbus TCP Slave на Ethernet. Пользователь конфигурирует запросы, которые будет отправлять каплер, и адреса slave-регистров, в которых будут размещаться полученные данные. Преимуществом такого подхода является снижение времени, требуемого для передачи данных: каплер может опрашивать несколько шин RS-485 «параллельно» и поддерживать одновременное подключение нескольких master-устройств, которые будут «забирать» эти данные.

4.2. Unit ID

Спецификация Modbus TCP ([3], п. 3.1.2, 3.1.3, 4.4.1.2) определяет, что в каждом фрейме должен быть указан адрес устройства (Unit ID). Необходимость этого параметра часто вызывает вопросы у начинающих инженеров – ведь в сетях TCP/IP для идентификации устройства достаточно только IP-адреса и номера порта.

Наличие этого параметра в протоколе обусловлено как раз возможностью интеграции сетей Modbus RTU/Modbus TCP с помощью шлюзов. Получив фрейм Modbus TCP – шлюз должен понимать, какому из устройств на шине Modbus RTU он предназначен (то есть знать его Slave ID).

Если же master-устройство обращается к slave-устройству по протоколу Modbus TCP «напрямую» (без использования шлюзов), то согласно спецификации Unit ID должен иметь значение 0xFF (допускается также использовать значение 0x00). Тем не менее, некоторые производители slave-устройств задают им «значимые» адреса (обычно адрес 1). В качестве примеров таких устройств можно привести модули ввода-вывода ET-2200 компании ICP DAS, которые по умолчанию имеют адрес 1 и модули ввода-вывода Mx210 компании ОВЕН, которые в ранних версиях прошивок[3] имели адрес 1 без возможности его изменения.

Это отступление от спецификации фактически не позволяет использовать такие приборы вместе со шлюзами Modbus.

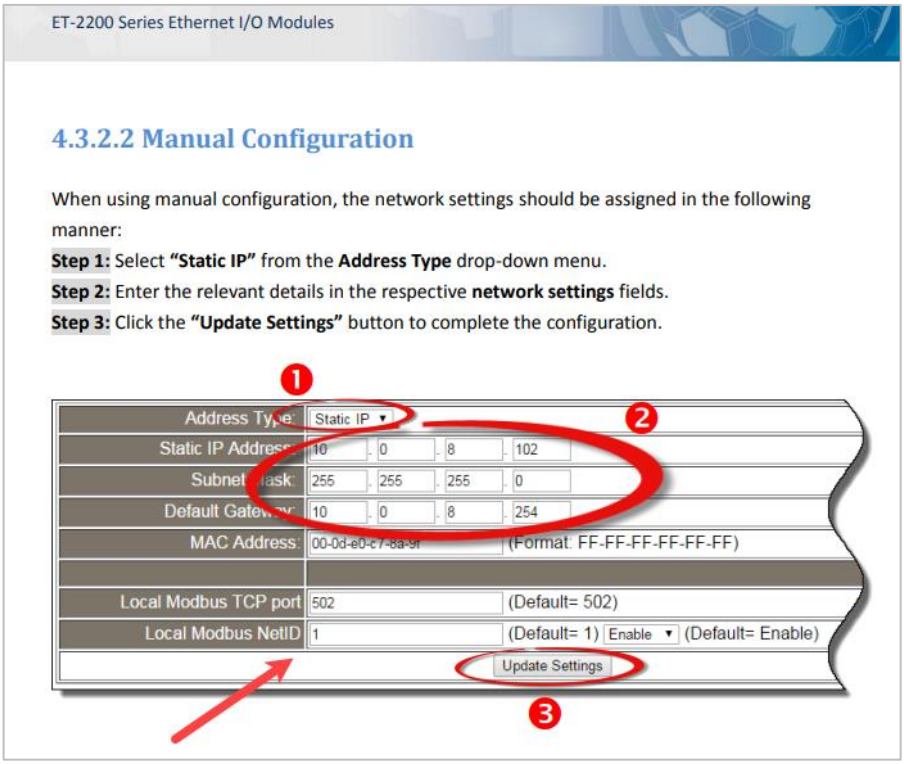


Рис. 4.1. Фрагмент руководства модулей ввода-вывода ICP DAS ET-2200 с указанием Unit ID по умолчанию

[3] До прошивки версии 1.0.

4.3. Специальные коды ошибок для шлюзов

Спецификация Modbus ([1], п. 7) описывает специальные коды ошибок, которые могут быть возвращены только шлюзами Modbus:

- 0x0A (GATEWAY PATH UNAVAILABLE) – не удалось построить маршрут из одного интерфейса в другой (это может быть связано с некорректной настройкой шлюза или нехваткой аппаратных ресурсов для обработки текущего числа запросов);
- 0x0B (GATEWAY TARGET DEVICE FAILED TO RESPOND) – отсутствие ответа от slave-устройства. Наличие этой ошибки позволяет master-устройству отличить выход из строя устройства, размещенного за шлюзом, от выхода из строя самого шлюза (во втором случае master-устройство вообще не получило бы никакого ответа).

MODBUS Application Protocol Specification V1.1b3 Modbus

		on the server device.
0A	GATEWAY PATH UNAVAILABLE	Specialized use in conjunction with gateways, indicates that the gateway was unable to allocate an internal communication path from the input port to the output port for processing the request. Usually means that the gateway is misconfigured or overloaded.
0B	GATEWAY TARGET DEVICE FAILED TO RESPOND	Specialized use in conjunction with gateways, indicates that no response was obtained from the target device. Usually means that the device is not present on the network.

Рис. 4.2. Фрагмент спецификации Modbus с описанием кодов ошибок шлюзов

5. Список литературы

1. MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3
2. MODBUS over Serial Line. Specification and Implementation Guide V1.02
3. MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b
4. Modicon Modbus Protocol Reference Guide. PI–MBUS–300 Rev. J