ModMaster Manual v3.11

Table of Contents

Introduction

Note: This manual is also available on the Wingpath website in HTML and PDF formats.

In this manual the traditional Modbus terms "master" and "slave" are used. If you are from a networking background, you might find the manual easier to read if you substitute "client" for "master" and "server" for "slave".

Overview

ModMaster is a versatile Modbus diagnostic program that can be used to test almost any device or system that uses the Modbus protocol.

ModMaster is a Modbus master that provides two ways of sending commands to a Modbus slave:

- You can define some Modbus registers and use the **Read**, **Write** and **Poll** buttons to send requests to the slave. See Setup for using defined registers for how to do this.
- Alternatively, you can define and send Modbus commands using the **Define Command** page - this method does not use register definitions. See Setup for sending commands only for how to do this.

Running ModMaster

To run ModMaster under Windows, you can use the Windows Run dialog or double-click on the icon or filename.

To run ModMaster under other operating systems (or from a Windows command window) use the command:

```
java -jar modmaster3.11.jar
```

You will, of course, need to use the full pathname for the `modmaster3.11.jar` file if it's not in your current directory. You will probably want to create a shortcut, launcher, shell script, or whatever to save you having to keep typing the above command.

Instructions for setting up ModMaster are given in the section Setup overview .

To exit from ModMaster, select the **File** → **Exit** menu item or click the close button of the main window.

The GUI

When you start ModMaster the main window is displayed.

At the top of the window are the usual menu bar and tool bar.

On the left side of the window there is a side panel showing a list of pages. When you select an item in this list, the page is displayed in the main panel on the right side of the window.

At the bottom of the window is a status bar, which is used to display error and information messages.

Side panel

The pages listed in the side panel are in four groups:

- Settings: These options are described in the Settings section.
- Registers: These options are described in the Registers section.
- File Registers: These options are described in the File Registers section.
- Commands: These options are described in the Commands section.

Entries in the list of pages are colour-coded to indicate the state of the corresponding page:

- Blue/grey: The page is selected (i.e. displayed in the main panel).
- Yellow: The page has unapplied changes. The changes themselves are also highlighted in yellow within the page.
- Red: The page has an error message in its status bar.

Main panel

Each page that may be displayed in the main panel has buttons for various actions. These typically include an **Apply** button (which saves the changes you have made), a **Reset** button (which restores the settings to their last saved state), and a **Help** button (which provides help on the page - you can also get help by pressing the F1 key).

Each page also has a status bar below its buttons, which is used to display error and information messages that are specific to the page.

Status bars

At the bottom of the main window is a "status bar", which is used to display error and information messages. Each page displayed in the main panel also has its own status bar, which is used to display messages that are specific to that page.

Many of the error messages displayed in the status bars have an **Error Help** button, which provides help on the error message (you can also get this help by pressing the F4 key).

Most of the messages that are displayed in the status bars are removed automatically when they are no longer applicable. This is not possible for some messages, and a **Clear** button is provided for these so that they can be manually removed.

Entering data

You can select a data field in which to enter data either by clicking on it with the mouse or by moving to it using the Tab or Shift-Tab key.

When you select a field, all the text within the field is initially selected, so what you type will replace the original value. If you want to edit the value, rather than replace it, click on the value again or press the right-arrow or left-arrow key.

If you place the mouse cursor over a data field (or its label), a tooltip is displayed, which provides a brief description of the field's purpose. You can also display the tooltip of the currently selected data field by typing Ctrl-F1. The display of tooltips can be disabled or enabled using the **View** → **Tool tips** menu option or by typing Shift-F1. For a more detailed description of a field's purpose, click the **Help** button or by press the F1 key.

If you start entering or editing a field value and change your mind, you can restore the original value by pressing the Esc key. To restore the values of *all* the fields in a page, click the **Reset** button.

When you have finished entering or editing a field value, you can click on another field or a button, or press Tab or Enter to move to the next field.

Some of the pages have a "mirror" field, situated below the buttons and above the status bar, which can be used for easier viewing and editing of long values. When a data field is selected, its value is displayed in the mirror field and the label of the mirror field is changed to indicate which field is being mirrored. You can enter or edit the field value as usual, or you can click on the mirror field and enter/edit the value there. The F2 key can also be used to switch between the normal field and the mirror field.

Setup overview

This section gives an overview of how to set up ModMaster.

The pages used for setup are from **Settings** group of pages listed on left of the main window. Each page has a **Help** button, which provides context-sensitive help on the page.

ModMaster provides two ways of sending commands to a Modbus slave:

- You can define some Modbus registers and use the **Read**, **Write** and **Poll** buttons to send requests to the slave. See Setup for using defined registers for how to do this.
- Alternatively, you can define and send Modbus commands using the **Define Command** page - this method does not use register definitions. See Setup for sending commands only for how to do this.

Setup for sending commands only

If you simply want to send Modbus commands to the slave (i.e. you don't want to use polling or the register read/write buttons) you can set up ModMaster using the following steps:

- Use the **Interface to Slave** page (see Interface settings) to configure the interface to the Modbus slave.
- If the slave uses 32-bit or 64-bit values, configure the handling of these using the **32/64-bit Values** page (see 32/64-bit Value Settings).
- You may also want to configure the **Number of Retries** and **Response Timeout** in the **General** page (see General settings).
- Click the **Run** button to open the connection to the Modbus slave.

You may then send individual Modbus commands to the slave using the **Define Command** page (see Commands).

You may close the connection to the slave by clicking the **Run** button, and re-open it by clicking the **Run** button again.

Setup for using defined registers

If you want to use polling or the register read/write buttons, the steps for setting up are a bit more complicated, since you will need to define some registers:

- In the **General** page (see General settings), set the **Slave ID** to the slave identifier that you want ModMaster to send in requests.
- Use the **Interface to Slave** page (see Interface settings) to configure the interface to the Modbus slave.
- If you want to use coils or discrete inputs, or you want to use register addresses that are different from the addresses sent in messages, configure the address mapping using the **Address Mapping** page (see Address mapping).
- If the slave uses 32-bit or 64-bit values, configure the handling of these using the **32/64-bit Values** page (see 32/64-bit Value Settings).
- Define the registers that you want to access (see Defining registers).
- If you want to use polling, you may want to configure how ModMaster polls registers in the **Polling** page (see Polling Settings).
- You may also want to configure the **Number of Retries** and **Response Timeout** in the **General** page (see General settings).
- Click the **Run** button to open the connection to the Modbus slave.

You may then use polling and the register read/write buttons.

You may close the connection to the slave by clicking the **Run** button, and re-open it by clicking the **Run** button again.

Saving and restoring the configuration

ModMaster's configuration can be saved to a disk file by selecting the **File** → **Save Configuration** menu item. This displays a dialog that enables you to enter or select the name of the file to save the configuration to. All of ModMaster's configuration (including register and command definitions) is saved to the file in an XML format.

To restore the configuration from a file, use the **File** → **Load Configuration** menu item. This displays a dialog that enables you to enter or select the name of the file to restore the configuration from. If ModMaster is "running" (i.e. the **Run** button is depressed), you will have to click the **Run** button before you can load the configuration.

Loading the configuration from a file will add new register definitions and replace existing register definitions, but will not delete any register definitions. If you want to totally replace the the set of register definitions you should delete all the register definitions (using the **Delete All** buttons in the Register Table and File Register Table pages) before loading the new configuration.

Similarly, loading the configuration will add new command definitions and replace existing command definitions, but will not delete any command definitions. If you want to totally replace the the set of register definitions you should delete all the command definitions (using the **Delete All** button in the Command Table page) before loading the new configuration.

You can also load the configuration from a file when you start ModMaster, by passing the name of the file as a command-line argument, e.g.:

```
java -jar modmaster3.11.jar config.xml
```

If you pass a folder/directory name on the command line instead of a file name, it will be used to set the current directory of the **Load Configuration** dialog.

There is, currently, no formal DTD (Document Type Definition) or XML schema for the format of the XML files in which the configuration is saved. However, the format should be fairly evident to a human reader, and can easily be edited, if necessary, using a text editor.

Settings

General settings

The **General** settings page enables you to set the following items:

- **Slave ID.** The **Slave ID** is what is called the "slave address" in the original Modbus specification, and is called the "unit identifier" in the Modbus/TCP specification. It should be in the range 0 to 255.

  A zero slave identifier is used in write requests as a broadcast identifier (each slave that receives the message should store the values supplied in the request, but should not normally send a response). The serial specification limits the range of slave identifiers to 0 to 247, for no apparent reason. The Modbus/TCP implementation guide suggests that Modbus/TCP slaves that can be uniquely identified by their IP address should use slave identifier 255.

- **Number of Retries.** The **Number of Retries** field specifies how many times ModMaster should retry a request that it has not received a response to.

- **Response Timeout.** The **Response Timeout** field specifies how long (in milliseconds) ModMaster should wait for a response before re-trying the request or reporting a failure.

- **Request Delay.** The **Request Delay** field is used to insert a time delay before each read or write request when polling and when sending a sequence of commands. This delay may be necessary in a 2-wire RS485 RTU setup, to enable slaves to separate master request messages from response messages sent by other other slaves. The delay may also be used to reduce the load on the slave or on the computer running ModMaster. The delay is measured in milliseconds. (Also see Pass Delay.)

- **Max PDU size.** This field specifies the maximum allowable number of bytes in the PDU (Protocol Data Unit) of Modbus messages. You may need to reduce the maximum PDU size when communicating over some networks to avoid fragmentation of messages.

  ModMaster will not send any request that exceeds the PDU limit, or whose response would exceed the limit. See the note on message size limits for more information.

- **Check Count Limits.** The current Modbus specification specifies arbitrary limits on the counts in some Modbus requests (e.g. a maximum of 2000 coils in a Read Coils request). Select this checkbox if you want ModMaster to check these limits.

  Since ModMaster can handle non-standard register sizes, count limits for holding and input registers are multiplied by 2 and then interpreted as byte count limits. ModMaster will warn you if you send a request that exceeds a count limit or whose response would exceed a limit. See the note on message size limits for more information.

- **Strict Checking.** Normally ModMaster checks that received Modbus messages conform to the current Modbus specification, and rejects them if they do not. Some of these checks are not essential: messages that fail them can still be interpreted correctly. Deselect **Strict Checking** if you want ModMaster to be more tolerant of non-conforming messages.

  The checks enabled by the **Strict Checking** setting are:

  - The byte count in a response message must match the count in the request message (commands 1 Read Coils, 2 Read Discrete Inputs, 3 Read Holding Registers, 4 Read Input Registers, 20 Read File Record, 23 Read/Write Multiple Holding Registers).
  - A response message must correctly echo data that was sent in the request (commands 5 Write Single Coil, 6 Write Single Holding Register, 8 Diagnostics, 15 Write Multiple Coils, 16 Write Multiple Holding Registers, 21 Write File Record, 22 Mask Write Holding Register, 43/14 Read Device Identification).
  - Unused fields in messages must be correctly set (command 43/14 Read Device Identification).
  - A message must not contain excess data.

- **Allow Long Messages.** Some Modbus messages contain a count of the number of data bytes in the message. This byte-count is in a one-byte field, and so has a maximum value of 255, which limits the quantity of data that can be sent in a single message. The byte-count is actually redundant: it is possible to determine the number of data bytes in a message from the overall size of the message.

  If you select **Allow Long Messages**, ModMaster will ignore the byte-count field and determine the number of data bytes from the overall message size, and will set the byte-count to 0 when sending messages containing more than 255 data bytes. If you also increase the Max PDU Size, ModMaster will be able to send and receive messages containing more than 255 data bytes.

  Sending messages containing more than 255 data bytes is non-standard and not widely supported. It may affect the detection of the end of messages when using the RTU packet type, and may affect the ability of the CRC/LRC check to detect errors. However, it should be reliable when using a socket (TCP or UDP) connection. Note that UDP datagrams are limited in size, so this will limit the size of Modbus messages that can be sent using UDP.

- **Use Write-Single Functions.** Normally ModMaster uses function code 15 (Write Multiple Coils) and function code 16 (Write Multiple Registers) for register writes. You can select this checkbox to use function code 5 (Write Single Coil) and function code 6 (Write Single Register) instead.

Interface settings

Use the **Interface to Slave** page to configure the interface to the slave.

The first four items in the page must always be configured:

- **Interface Type.** Select **Serial** for a RS232/485 serial interface. Select **TCP Socket** for a TCP/IP network interface. Select **UDP Socket** for a UDP/IP interface (use of UDP for Modbus is non-standard). Note that the **Serial** option is only available on Windows and Linux operating systems (the **Help** → **About** window tells you whether the serial library is loaded).

- **Packet Type.** You would normally select **RTU** if you are using a serial interface, and **TCP** if you are using a socket interface. You may have to use **ASCII** for some legacy serial interfaces or devices. It is possible to use RTU or ASCII packets over a socket interface - this is non-standard, but it is used by some serial-TCP converters and can also be useful for software testing. It is also possible to use TCP packets over a serial interface, but there is no good reason to do so (and there would be no error-checking of the packets).

- **EOM Timeout.** Enter the maximum delay (in milliseconds) expected within a message. The default value works fine in most situations, but you may need to increase the value if you are using a serial interface (or an operating system) that introduces large delays in the middle of messages.

- **Idle timeout.** Period in seconds after which a connection will be closed if idle. A value of 0 will disable the timeout.

  A connection to a slave device is considered to be idle if no valid responses have been received from the device within the timeout period.

  Setting an idle timeout can be used to avoid holding on to resources that are not currently in use. It can also be useful for recovering from some error conditions.

- **Responses.** When using Modbus RTU or ASCII over a serial interface, a slave should not respond to a broadcast request or to a request for a different slave ID (this is necessary for RS485 multi-drop to work). When using Modbus TCP over a TCP socket interface, the slave/bridge should respond to all requests, including broadcasts and requests to unknown/unreachable slaves (responding to the latter with an exception 10 or 11 response). When using non-standard protocol variants (such as RTU encapsulated in TCP), the behaviour in these situations is not defined. You should select **Always responds** if you want the Modbus TCP behaviour, or deselect it if you want the serial behaviour.

The **Interface Type** and **Packet Type** together determine the variant of the Modbus protocol that will be used. The table below lists the possible combinations:

Table 1. Modbus protocol variants

Interface type	Packet type	Protocol variant
Serial	RTU	Modbus RTU
Serial	ASCII	Modbus ASCII
Serial	TCP	Not recommended since no error checking (non-standard)
TCP Socket	RTU	Modbus RTU encapsulated in TCP (non-standard)
TCP Socket	ASCII	Modbus ASCII encapsulated in TCP (non-standard)
TCP Socket	TCP	Modbus TCP
UDP Socket	RTU	Modbus RTU encapsulated in UDP (non-standard)
UDP Socket	ASCII	Modbus ASCII encapsulated in UDP (non-standard)
UDP Socket	TCP	Modbus UDP (also known as Modbus TCP encapsulated in UDP) (non-standard)

The following items must be configured for a socket interface:

- **Host.** Enter the host name or IP address of the slave (or Modbus bridge/gateway).
- **Port.** Enter the port number that ModMaster should connect to on the slave host (or Modbus bridge/gateway).
- **Local host.** You would normally leave this field empty. However, if you are using a computer that has multiple network interfaces, you can enter the host name or IP address of one of the network interfaces to force that interface to be used for the connection to the slave.
- **Local port.** You would normally leave this field set to 0, which allows dynamic allocation of the local port to be used for the connection. However, some firewalls or NAT (Network Address Translation) systems may require you to set a specific port.

The following items must be configured for a serial interface:

- **Port.** Select or enter the name of the serial port that ModMaster should use to talk to the slave. If (during testing) the connection between the master and slave is between two serial ports that are both on the same machine, then you must obviously use different port names for each. Don't forget to connect the two ports with a cable!
- **Speed.** Select or enter the speed in bits/second. Note that serial interfaces and drivers vary in the speeds that they support.
- **Parity.** It's usual to use no parity, since the CRC in RTU packets is a good error check on its own, but some devices may require it. The serial specification requires even parity to be the default.
- **Data Bits.** Must be 8 for RTU. Some ASCII devices may require 7 (the serial specification actually specifies 7 for ASCII).
- **Stop Bits.** Usually 1, but some devices may require 2. The serial specification actually specifies 2 stop bits if parity is not being used, but this requirement is normally ignored.
- **RTS Control.** The default setting of **High** will be OK for most purposes.

  Selecting **Flow Control** will enable hardware flow control ("handshaking") using RTS and CTS. Since Modbus messages are short, flow control is not normally necessary.

Selecting **RS485** will use RTS to control a RS232 to RS485 converter: ModMaster will raise RTS when it is transmitting data, and lower RTS to receive data. Note that the lowering of RTS may be delayed by the operating system, and this may cause loss of data. For this reason, we recommend the use of RS232-RS485 converters that do not require RTS control.

Address mapping

As explained in section 4.3 of the current Modbus specification, the Modbus protocol uses four different address ranges (called "tables" in the specification): discrete inputs, coils, input registers, and holding registers. The function code of a message determines to which address range the address in the message belongs. The address ranges may be separate, or they may be overlaid so that a register may appear in more than one address range. Note that the specification says nothing about how the overlaying is done - in particular, there is no mention of the byte-ordering issues involved.

Types of address

In this documentation, we use the term "message address" to refer to the address that is actually sent in a message. We use the term "model address" to refer to the address that you supply when you define a register in ModMaster (see Defining registers).

Note that message addresses are restricted to the range 0..65535 since they are transmitted in a 16-bit field in Modbus messages, whereas model addesses are only used internally by ModMaster and may be in the range 0..2147483647.

Defining address ranges

The **Address mapping** page enables you to set up the mapping between message addresses and model addresses.

For each address range, you have to specify a "base address" and a "number of addresses".

The number of addresses is simply the number of message addresses in the range, i.e. for each message address:

message_address >= 0 and message_address < number_of_addresses

The base address is the model address that corresponds to message address 0. For holding registers and input registers this means that for each model address:

model_address = base_address + message_address

This also applies to coils and discrete-inputs, provided that you use 1-bit "discrete" registers. If the coil or discrete input address range does not overlay the holding register or input register address range, then it is simplest to use 1-bit registers for coils and discrete inputs.

Overlaying address ranges

If the same register appears in more than one address range, then the address ranges are said to be "overlaid". Overlaying of holding registers with input registers presents no particular problems, nor does overlaying of coils with discrete inputs. However, overlaying of coils/discrete-inputs with holding/input registers is tricky.

If you want to overlay the discrete input or coil address range with the input register or holding register address range, then you will probably want to use 16-bit or larger registers.

If you use 8-bit or larger registers for coils or discrete-inputs, the address mapping is more complicated because message addresses refer to individual bits, whereas model addresses refer to registers. In this case, ModMaster uses the size of the registers in the address range to determine the number of bits in a register, and then uses the mapping:

model_address = base_address + message_address / bits_in_register

The remainder after dividing by bits_in_register (i.e. message_address modulo bits_in_register) is used to determine which bit in the register the address refers to. The bits are numbered "from right to left" (i.e. bit 0 is the least significant

bit), unless you select the **Reverse Bits** checkbox, in which case they will be numbered "from left to right" (i.e. bit 0 will be the most significant bit).

Note that all the registers in the discrete input or coil address range must be the same size for this method of mapping to work.

If you use 16-bit or larger registers for discrete inputs or coils, then you will have to decide (or find out) what byte-ordering you are using and configure ModMaster accordingly. Discrete inputs and coils are sent in messages in increasing address order; for example, if you are transferring coils 24..39 then coils 24..31 would be in the first byte and coils 32..39 would be in the second byte. However, 16-bit registers are transferred in MSB first (big-endian) order, so the first byte contains bits 8..15 of the value and the second byte contains bits 0..7. A straightforward mapping of the two bytes of coils to the two bytes of the 16-bit register therefore results in coils 24..31 mapping to bits 8..15 of the register, and coils 32..39 mapping to bits 0..7 of the register. If you want the bytes mapped the other (more intuitive?) way round, you should select the **Swap Bytes for Coils/Discrete Inputs** checkbox. If you use registers larger than 16-bits, the word-ordering configuration described in 32/64-bit Value Settings will also apply.

If the mapping of overlaid discrete inputs and coils seems complicated, that's because it is! If you are designing a new Modbus device, we strongly recommend that you avoid overlaying discrete inputs or coils onto input registers or holding registers. Better still, you could not use discrete inputs and coils at all, and instead use the Mask Write Register command for bit manipulation.

Example address mappings

- The default address mapping used by ModMaster is shown below:

Table 2. Default address mapping

	Base Address	Number of Addresses
Holding Register	0	65536
Input Register	0	65536
Coil	0	0
Discrete Input	0	0

This mapping overlays input registers and holding registers, and makes message addresses and model addresses identical. It effectively disables discrete inputs and coils by setting the sizes of these address ranges to zero.

- Another example mapping is the traditional non-overlaid setup used by Modicon (also known as 5-digit addressing):

Table 3. Traditional Modicon 5-digit address mapping

	Base Address	Number of Addresses
Holding Register	40001	9999
Input Register	30001	9999
Coil	1	9999
Discrete Input	10001	9999

- A more recent version of the traditional Modicon scheme uses 6-digit addresses:

Table 4. Modicon 6-digit address mapping

	Base Address	Number of Addresses
Holding Register	400001	65536
Input Register	300001	65536
Coil	1	65536
Discrete Input	100001	65536

- The following table shows a "simple" example of overlaying coils with holding registers:

Table 5. Overlaid address mapping

	Base Address	Number of Addresses
Holding Register	0	65536
Coil	100	96

With this mapping, registers 100..105 would be accessible as holding registers 100..105 and also as coils 0..95 (assuming the use of 16-bit registers).

For example, using the overlaying rules coil 37 would map to bit 5 (37 modulo 16) of register 102 (100 + 37 / 16). Which bit of the register is "bit 5" would, of course, depend on the **Swap Bytes** and **Reverse Bits** settings.

32/64-bit Value Settings

The official Modbus protocol (in both the original Modbus specification and the current Modbus specification) only allows 1-bit and 16-bit integer values to be transferred. Many manufacturers have extended the protocol to allow 32-bit and 64-bit values, and also to allow floating-point values. Fortunately, everyone seems to have used the IEEE format for floating-point numbers, but that is where the agreement ends. The **32/64-bit Value Settings** page enables you to configure ModMaster to handle all implementations of 32/64-bit values that we know of.

- **Little Endian.** Modbus is a "big-endian" protocol: that is, the more significant byte of a 16-bit value is sent before the less significant byte. It seems obvious that 32-bit and 64-bit values should also be transferred using big-endian order. However, some manufacturers have chosen to treat 32-bit and 64-bit values as being composed of 16-bit words, and to transfer the words in little-endian order. For example, the 32-bit value 0x12345678 would be transferred as 0x56 0x78 0x12 0x34. You should select the **Little Endian** checkbox to use this mixed ordering.

- **Word Registers.** Each register in the Modbus protocol holds a single (16-bit) value. The simplest way to extend the protocol to handle 32-bit and 64-bit values is to allow registers that contain these larger values. However, some manufacturers have chosen to keep to the 16-bit register size, and use 2 registers to hold a 32-bit value, and 4 registers to hold a 64-bit value. For example, if a 32-bit value is stored at address 100, then register 100 would hold one half of the value and register 101 would hold the other half of the value. Some devices will actually allow you to access the halves of the value independently; others will only allow accesses that transfer the complete value (thus making address 101 in the example an invalid address).

  You should select the **Word Registers** checkbox to use multiple registers to store large values, and leave the checkbox unselected to use a single register to hold each value. Note that when you use the **Word Registers** option, ModMaster will only allow access to complete values, and you should add definitions only for the first register used for each value (in the above example, you should define register 100, but not register 101).

- **Word Count.** Some Modbus requests (e.g function 3 Read Holding Registers, and function 16 Write Multiple Registers) include a count of how many registers/values are to be transferred. There are three possible interpretations for this count:

  - The number of values to be transferred.
  - The number of 16-bit words to be transferred.
  - The number of registers to be transferred.

  In the official Modbus protocol, these three interpretations are equivalent, since all values and all registers are 16-bit. When 32-bit and 64-bit values and/or registers are allowed, the three interpretations become distinct. However, the third interpretation will be equivalent to one or other of the first two, depending on the **Word Registers** setting, so ModMaster only deals directly with the first two interpretations.

  You should select the **Word Count** checkbox if the count is to be interpreted as the number of 16-bit words to be transferred. You should leave the **Word Count** checkbox unselected if the count is to be interpreted as the number of values to be transferred.

These three options (Little-endian, Word-registers, and Word-count) allow 8 possible variations of 32/64-bit value handling. At least 5 of these variations have actually been used in devices, so you may need to carefully read documentation, or to experiment, to determine which variation a particular device uses. The default settings (all options deselected) should work with Enron/Daniel devices. For Modicon/Schneider devices you will probably need to select all the options.

Polling Settings

The **Polling** settings page is used to configure how ModMaster polls registers (see Polling).

- **Grouping: Single Values.** Normally, ModMaster will try to transfer as many values as possible in each read or write request that it sends to the slave. If you select the **Single Values** checkbox, then it will transfer only one value in each request.
- **Error Handling: Continue Polling.** Normally, ModMaster will stop polling if it receives an error response from the slave. If you select the **Continue Polling**, then it will continue polling if it receives an error response.
- **Pass Delay.** The **Pass Delay** field is used to insert a time delay into the polling sequence to reduce the load on the slave (and maybe also on the computer running ModMaster). The delay is inserted after each complete pass through all the registers. The delay is measured in milliseconds. (Also see Request Delay).

Registers

You can send register read and write requests using the Define Command page, without needing to define any registers. Alternatively, you can define some registers, as described in the following sections, and use the Register Table buttons to request register reads and writes.

Defining registers

Use the **Add Registers** page to add definitions of registers. If ModMaster is "running" (i.e. the **Run** button is depressed), you will have to click the **Run** button before you can add register definitions.

This page enables you to enter the following details:

- **Quantity.** How many registers to add. The **Address** will be incremented for each register added. The other fields will have the same values in each register.
- **Address.** The address of the first register. This is a "model address", and may be different from the address used in messages, depending on how the address mapping is configured (see Address mapping).
- **Name.** An optional name for the register. ModMaster displays the name, but does not use it in any other way. It is purely for your benefit, to remind you of what the register is used for.
- **Unit.** An optional unit for the register. ModMaster displays the unit, but does not use it in any other way. It is purely for your benefit, to remind you of what unit is used for the register value.
- **Type.** The type (signed integer, unsigned integer, or floating-point) and size (1, 8, 16, 32 or 64 bit) of the value to be stored in the register. You would normally use **Int 16** or **Uint 16** for holding registers and input registers, and **Discrete** (i.e. 1-bit integer) for coils and discrete inputs - the other sizes are non-standard.
- **Radix.** The radix to be used to display the register value. The radix only affects the display of unsigned integer values: floating-point and signed integer values are always displayed in decimal.
- **Offset** and **Scale.** These allow you to scale the register value when displaying it in decimal (the scale and offset are not used when displaying in other radixes). The scaling uses the formula:

  displayed_value = (register_value + offset) / scale

- **Minimum** and **Maximum.** These allow you to specify limits for the register value. If the register value is less than the specified minimum or greater than the specified maximum, the value is displayed with a red background.
- **Value.** The initial value to be stored in the register.
- **Write.** When polling, ModMaster will write this register to the slave if this option is selected, otherwise it will read this register from the slave (see Polling).

When you have entered/selected values for the above fields, click the **Add** button to add the register definition(s). You may then edit the values and click the **Add** button again to add further register definitions. If you click the **Reset** button, all the values will be reset to their defaults.

Register Table

Details of all defined registers are displayed in the **Register Table**.

See the section Defining registers for explanations of the values displayed in each column.

You can select which columns are displayed in the table using the **View** → **Columns** menu item.

Editing register definitions

A field of a register can be edited in the register table by clicking on the field (or moving to it with the keyboard) and then entering/selecting the new value. The **Type** of a register may only be changed when ModMaster is stopped (i.e. click the **Run** button if necessary).

Alternatively, a register may edited using the **Edit Register** page. Use the **Address** drop-down list to select the register to be edited, change the values as required, and click **Apply** to save the changes.

Deleting register definitions

To delete a register, select it by clicking on its address in the Register Table (or moving to it using the keyboard), and then click the **Delete** button.

To delete several registers, select them by clicking on their addresses while holding down the control key, and then click the **Delete** button.

To delete *all* the registers click the **Delete All** button.

You can delete registers only when ModMaster is stopped (i.e. click the **Run** button if necessary).

Reading and writing registers

You can read from or write to defined registers by selecting them in the Register Table and clicking the **Read** or **Write** button. ModMaster will read values from the slave and display them in the register table, or write the values displayed in the register table to the slave.

ModMaster uses the address range of the register to choose which Modbus command to use, as shown in the following table:

Table 6. Register read/write implementation

Address range	Read command	Write command
Holding register	3 Read Holding Registers	16 Write Multiple Registers*
Input register	4 Read Input Registers	-
Coil	1 Read Coils	15 Write Multiple Coils*
Discrete input	2 Read Discrete Inputs	-

* If the **Use Single-Write Functions** option has been selected, 6 Write Single Register will be used instead of 16 Write Multiple Registers and 5 Write Single Coil will be used instead of 15 Write Multiple Coils.

If a register appears in more than one address range (i.e. is overlaid), then ModMaster will use the first (in the above table) address range that it appears in.

If more than one Modbus request is needed to perform the read or write, ModMaster will delay each request except the first by the configured **Request Delay** .

Polling

If you want to read or write *all* the defined registers, you can use the **Poll Once** button on the Register Table page. This causes ModMaster to make a pass through all the register definitions, writing to the slave those registers that have the write flag set, and reading the other registers. The pass is done in model address order (i.e. in the order that the registers are displayed in the register table).

If you click the **Poll** button on the toolbar, ModMaster will continually poll all registers when it is running. While ModMaster is continually polling, you may also request a poll manually (e.g. you may have configured ModMaster to poll every 5 minutes, and you want to update the registers without waiting several minutes for the next automatic poll).

See Reading and writing registers for a description of how the reading and writing is implemented.

The Polling Settings control several aspects of how the polling is done.

Saving and restoring register definitions

Register definitions may saved to, and restored from, disk files as part of ModMaster's configuration (see Saving and restoring the configuration).

Register definitions can also be saved separately by selecting the **File** → **Export Registers** menu item. This displays a dialog that enables you to enter or select the name of the file to save the registers to.

To restore the registers from a file, use the **File** → **Import Registers** menu item. This displays a dialog that enables you to enter or select the name of the file to restore the registers from. If ModMaster is "running" (i.e. the **Run** button is depressed), you will have to click the **Run** button again before you can load the registers.

Loading from a file will add new register definitions and replace existing register definitions, but will not delete any register definitions. If you want to totally replace the the set of register definitions you should delete all the register definitions (using the **Delete All** button in the Register Table page) before importing the new definitions.

Export Registers saves the register definitions in CSV format, and **Import Registers** expects the file it reads to be in CSV format. The CSV format is described in the CSV file format appendix.

File Registers

File registers are used to store the values read and written by the Read File Record (function code 20) and Write File Record (function code 21) commands.

You can send Read File Record and Write File Record commands using the Define Command page, without needing to define any file registers. Alternatively, you can define some file registers, as described in the following sections, and use the File Register Table buttons to request file register reads and writes.

Note that file registers are referred to by several different names in the Modbus specifications. They are variously called "file records", "registers" and "references" in the current Modbus specification, and they are called "General References" or "Extended Memory file (6x) references" in the original Modbus specification and the Modbus/TCP specification (with the corresponding commands being called "Read General Reference" and "Write General Reference").

Defining file registers

Use the **Add File Registers** page to add definitions of file registers. If ModMaster is "running" (i.e. the **Run** button is depressed), you will have to click the **Run** button before you can add file register definitions.

This page enables you to enter the following details:

- **Quantity.** How many registers to add. The **Address** will be incremented for each register added. The other fields will have the same values in each register.
- **File.** The number of the file containing the register(s). The file number must be in the range 1 to 65535.
- **Address.** The address of the first register to be added. File register addresses are also called "record numbers" in the current Modbus specification.

  The current Modbus specification restricts file register addresses to the range 0 to 9999, for no apparent reason. ModMaster will accept addresses in the range 0 to 65535 if you turn off **Strict Checking**.

  Note that there is no address mapping done for file registers: the address that you enter here is the address that is used in Modbus messages.
- **Name.** An optional name for the register. ModMaster displays the name, but does not use it in any other way. It is purely for your benefit, to remind you of what the register is used for.
- **Unit.** An optional unit for the register. ModMaster displays the unit, but does not use it in any other way. It is purely for your benefit, to remind you of what unit is used for the register value.
- **Type.** The type (signed integer, unsigned integer, or floating-point) and size (8, 16, 32 or 64 bit) of the value to be stored in the register. You would normally use **Int 16** or **Uint 16** for file registers - the other sizes are non-standard.
- **Radix.** The radix to be used to display the register value. The radix only affects the display of unsigned integer values; floating-point and signed integer values are always displayed in decimal.
- **Offset** and **Scale.** These allow you to scale the register value when displaying it in decimal (the scale and offset are not used when displaying in other radixes). The scaling uses the formula:

  displayed_value = (register_value + offset) / scale
- **Minimum** and **Maximum.** These allow you to specify limits for the register value. If the register value is less than the specified minimum or greater than the specified maximum, the value is displayed with a red background.
- **Value.** The initial value to be stored in the register.

When you have entered/selected values for the above fields, click the **Add** button to add the register definition(s). You may then edit the values and click the **Add** button again to add further register definitions. If you click the **Reset** button, all the values will be reset to their defaults.

File Register Table

Details of all defined file registers are displayed in the **File Register Table**.

See the section Defining file registers for explanations of the values displayed in each column.

You can select which columns are displayed in the table using the **View** → **File Columns** menu item.

Editing file register definitions

A field of a file register can be edited in the File Register Table by clicking on the field (or moving to it with the keyboard) and then entering/selecting the new value. The **Type** of a register may only be changed when ModMaster is stopped (i.e. click the **Run** button if necessary).

Alternatively, a register may edited using the **Edit File Register** page. Use the **File** and **Register** drop-down lists to select the register to be edited, change the values as required, and click **Apply** to save the changes.

Deleting file register definitions

To delete a file register, select it by clicking on its address in the File Register Table (or moving to it using the keyboard), and then click the **Delete** button.

To delete several registers, select them by clicking on their addresses while holding down the control key, and then click the **Delete** button.

To delete *all* the file registers in the displayed file, click the **Delete File** button.

To delete *all* the file registers in *all* files, click the **Delete Files** button.

You can delete file registers only when ModMaster is stopped (i.e. click the **Run** button if necessary).

Reading and writing file registers

You can read from or write to defined file registers by selecting them in the file register table and clicking the **Read** or **Write** button. ModMaster will read values from the slave and display them in the file register table, or write the values displayed in the file register table to the slave.

If more than one Modbus request is needed to perform the read or write, ModMaster will delay each request except the first by the configured **Request Delay** .

Saving and restoring file register definitions

File register definitions may saved to, and restored from, disk files as part of ModMaster's configuration (see Saving and restoring the configuration).

File register definitions can also be saved separately by selecting the **File** → **Export File Registers** menu item. This displays a dialog that enables you to enter or select the name of the file to save the registers to.

To restore the registers from a file, use the **File** → **Import File Registers** menu item. This displays a dialog that enables you to enter or select the name of the file to restore the registers from. If ModMaster is "running" (i.e. the **Run** button is depressed), you will have to click the **Run** button again before you can load the registers.

Loading from a file will add new file register definitions and replace existing file register definitions, but will not delete any file register definitions. If you want to totally replace the the set of file register definitions you should delete all the file register definitions (using the **Delete Files** button in the File Register Table page) before importing the new definitions.

Export File Registers saves the register definitions in CSV format, and **Import File Registers** expects the file it reads to be in CSV format. The CSV format is described in the CSV file format appendix.

Commands

ModMaster provides two ways of sending commands to a Modbus slave.

If you have defined some Modbus registers, you can use the **Read**, **Write** and **Poll** buttons to send requests to the slave.

Alternatively, you can define and send Modbus commands using the **Define Command** page, as described in the next section. These commands do not use register definitions.

Defining commands

You can use the **Define Command** page to define and send Modbus commands.

Select the required type of command from the **Command Type** drop-down list, and enter/select values as necessary.

When you have entered a command definition, you can click the **Send** button to send the command to the slave. The result of sending the command will be displayed in the page.

If you want to save the command definition, enter a unique name for the command in the **Name** field and click the **Save** button. All saved command definitions are listed in the Command table.

You can provide a description of the command in the **Description** field. ModMaster displays the description, but does not use it in any other way. It is purely for your benefit, to remind you of what the command is used for.

The **Reset** button will restore the values to their last saved state.

There are three kinds of command types in the **Command Type** drop-down list:

- **Custom command.** This option enables you to define and send non-standard Modbus commands, and is described in the Custom commands section.
- **Send raw data.** This option enables you to send arbitrary data to the slave, and is described in the Sending raw data section.
- **Standard commands.** These are standard Modbus commands, which are described in the Standard commands section.

Standard commands

The **Standard commands** in the **Command Type** drop-down list provide a low-level interface for sending Modbus commands to a slave.

These standard command pages make no use of any register definitions you may have entered, nor do they use or affect the register display area. All transferred values are displayed in the page itself.

The following fields are common to several of the pages:

- **Slave ID.** The **Slave ID** is what is called the "slave address" in the original Modbus specification, and is called the "unit identifier" in the Modbus/TCP specification. It should be in the range 0 to 255.

  A zero slave identifier is used in write requests as a broadcast identifier (each slave that receives the message should store the values supplied in the request, but should not normally send a response). The serial specification limits the range of slave identifiers to 0 to 247, for no apparent reason. The Modbus/TCP implementation guide suggests that Modbus/TCP slaves that can be uniquely identified by their IP address should use slave identifier 255.

  You may enter a slave identifier that is different from the slave identifier configured in General settings. In particular, you may enter 0 for the slave identifier in order to send broadcast messages.

- **Address.** Address of the first register to be transferred, in the range 0 to 65535. You should enter *message addresses* into these pages - the Address mapping settings are not used.

  The manuals for many Modbus devices adopt the confusing convention of adding an offset to Modbus addresses to indicate the type of variable:

Variable type	Offset
Coil	1
Discrete Input	10001 or 100001
Input Register	30001 or 300001
Holding Register	40001 or 400001

  The manuals for some Modbus devices simply add an offset of 1 to all addresses, so that, for example, "Holding Register 123" would have an address of 122.

  ModMaster requires actual Modbus addresses (as sent in Modbus messages), so you may have to subtract the appropriate offset from the addresses in device manuals.

- **Count.** Enter the number of *values* to be transferred - for example, if you want to transfer five 32-bit floating-point values you should enter 5. Note that this count may be different from the number of registers and may also be different from the count that is actually sent in the Modbus message (the settings for 32/64-bit Value Settings are used to convert the count if necessary).

- **Type.** The type (signed integer, unsigned integer, or floating-point) and size (1, 8, 16, 32 or 64 bit) of the value(s) to be transferred. You would normally use **Int 16** or **Uint 16** for holding registers and input registers, and **Discrete** (i.e. 1-bit integer) for coils and discrete inputs - the other sizes are non-standard.

- **Radix.** The radix to be used to display the value(s). The radix only affects the display of unsigned integer values; floating-point and signed integer values are always displayed in decimal.

For most commands, the other fields that need to entered should be obvious if you read the appropriate description in the current Modbus specification. The following sections describe the few cases that may not be obvious.

Read File Record (20)

The Read File Record command reads multiple groups of file registers from the Modbus slave.

Note that file registers are referred to by several different names in the Modbus specifications. They are variously called "file records", "registers" and "references" in the current Modbus specification, and they are called "General References" or "Extended Memory file (6x) references" in the original Modbus specification and the Modbus/TCP specification (with this command being called "Read General Reference").

To add a register group to the request, click the **Add** button and then enter the group details in the **File Register Groups** table:

- **File.** Enter the number of the file to read from. The file number must be in the range 1 to 65535.

- **Address.** Enter the address of the first register to read. File register addresses are also called "record numbers" in the current Modbus specification.

  The current Modbus specification restricts file register addresses to the range 0 to 9999, for no apparent reason. ModMaster will accept addresses in the range 0 to 65535.

- **Count.** Enter the number of *values* to be read - for example, if you want to read five 32-bit floating-point values you should enter 5. Note that this count may be different from the number of registers and may also be different from the count that is actually sent in the Modbus message (the settings for 32/64-bit Value Settings are used to convert the count if necessary).

To remove a register group from the request, select the group in the table and click the **Delete** button.

Write File Record (21)

The Write File Record command writes multiple groups of file registers to the Modbus slave.

Note that file registers are referred to by several different names in the Modbus specifications. They are variously called "file records", "registers" and "references" in the current Modbus specification, and they are called "General References" or "Extended Memory file (6x) references" in the original Modbus specification and the Modbus/TCP specification (with this command being called "Write General Reference").

To add a register group to the request, click the **Add** button and then enter the group details in the **File Register Groups** table:

- **File.** Enter the number of the file to write to. The file number must be in the range 1 to 65535.
- **Address.** Enter the address of the first register to write to. File register addresses are also called "record numbers" in the current Modbus specification.

  The current Modbus specification restricts file register addresses to the range 0 to 9999, for no apparent reason. ModMaster will accept addresses in the range 0 to 65535.
- **Count.** Enter the number of *values* to be written - for example, if you want to write five 32-bit floating-point values you should enter 5. Note that this count may be different from the number of registers and may also be different from the count that is actually sent in the Modbus message (the settings for 32/64-bit Value Settings are used to convert the count if necessary).

To remove a register group from the request, select the group in the table and click the **Delete** button.

The register values to be written should be entered in the **Value** column of the **Data to be written** table.

Read Device Identification (43/14)

The Read Device Identification command reads identification "objects" from the slave.

- **Code.** Select the required access type: **Basic**, **Regular**, **Extended** or **Specific**. **Specific** is used to read a single identification object. The other access types limit the range of objects returned.
- **First Object ID.** Enter the ID of the first object to be requested. If you are using the **Specific** access type, this will be the only object read.

  The ID should be in the range 0 to 255. Although not required by the Modbus specification, the ID would normally be restricted to the range 0 to 2 for **Basic** access, and to the range 0 to 127 for **Regular** access.

Custom commands

The **Custom Command** page enables you to define and send non-standard Modbus commands.

The following values must be entered:

- **Slave ID.** See Slave ID in the previous section.
- **Function.** Enter the Modbus function code for the command, in decimal.
- **Count.** Enter the number of bytes of data that you want to send.
- **Send data.** Enter the data bytes that you want to send, in hex.

If you want to enter all the data bytes as a group using copy-paste or drag-and-drop (e.g. from the trace output), you should paste/drop on to a grey cell in the table. The **Count** value will be set automatically if you enter the data this way.

The header/CRC/LRC will be added to the command as appropriate to the packet type (TCP/RTU/ASCII) that you have configured.

When you send the data to the slave, any data received in response from the slave will be displayed in the **Received Data** table. This data does not include the slave ID, function code and header/CRC/LRC bytes.

Sending raw data

The **Send Raw Data** page enables you to send arbitrary data to the slave. This is particularly useful for testing the error-handling capabilities of the slave.

The following values must be entered:

- **Count.** Enter the number of bytes of data that you want to send.
- **Send data.** Enter the data bytes that you want to send, in hex.

If you want to enter all the data bytes as a group using copy-paste or drag-and-drop (e.g. from the trace output), you should paste/drop on to a grey cell in the table. The **Count** value will be set automatically if you enter the data this way.

The data that you enter will be sent to the slave without adding any slave ID, function code or header/CRC/LRC bytes. If you want to send slave/function/header/CRC/LRC bytes, you will have to include these in the data that you enter.

When you send the data to the slave, any data received in response from the slave will be displayed in the **Received Data** table. This is also *raw* data, and will include any slave ID, function code or header/CRC/LRC bytes.

Command table

All the commands you have defined are listed in the **Command Table**.

The **Name** and **Command Type** columns show the name you gave the command and the command type. To see further details of a command, select the command and then select the View/Edit Command page.

Sending commands

You can send a command to the slave by selecting it and then clicking the **Send** button. To view the results (if any) of the command, select the View/Edit Command page.

You can send *all* the defined commands by clicking the **Send All** button (click the **Send All** button again if you want to interrupt the sending). The commands will be sent in the order that they are listed in the table - you can change the order if required. You can also insert a delay after sending each command, if you need to reduce the load on the slave.

To view the results (if any) of a command, select the command in the table and then select the View/Edit Command page.

Deleting commands

To delete a command, select it and the click the **Delete** button.

To delete several commands, select them by clicking on them while holding down the control key, and then click the **Delete** button.

To delete *all* commands, click the **Delete All** button.

Re-ordering commands

To move one or more commands, select them and then drag and drop them in the required position using the mouse.

To move commands using the keyboard, select them and type Ctrl-X. Use the arrow keys to move to the required position and then type Ctrl-V.

If you are moving more than one command, the commands must be next to each other.

Viewing and editing command definitions

You can view or edit a defined command using the **View/Edit Command** page.

Use the **Command** drop-down list to select the command to be viewed/edited.

If you change any values, click **Apply** to save the changes.

To restore the values to their last saved state, click the **Reset** button.

You can make a copy of the command by clicking the **Copy** button. The copy of the command will be displayed in the **Define Command** page, where you can modify it (if required), name it, and save it.

You can send the command to the slave (before or after saving it) by clicking the **Send** button.

Logging

The Logging page is used to configure the logging of error and information messages.

The messages generated by ModMaster are categorized into levels:

- **Fatal**: Messages about errors that are so serious that ModMaster cannot continue running.
- **Error**: Messages about errors that may be recoverable, i.e. ModMaster will attempt to continue running.
- **Warn**: Messages about potential problems.
- **Info**: Messages about the normal operation of ModMaster.
- **Trace**: Tracing of Modbus messages.

ModMaster can log messages to two different destinations: **Terminal** and **File**. You may log messages to more than one destination, with a different level for each destination.

For each destination, you should select the level of messages to be logged to that destination. When you select a particular level, messages of that level and higher levels will be logged. For example, if you select **Warn** then messages from the levels **Warn**, **Error** and **Fatal** will be logged. Select **None** if no messages are to be logged to that destination.

The destinations are:

- **Terminal**: This is a window, which can be displayed using the **View** → **Log** menu option.
- **File**: This is a text file. You will have to enter the name of the file in the **File Name** field, or use the **Browse** button to open a dialog to browse for the file.

  ModMaster will append messages to the file, so output from previous runs of the program will not get lost.

  You may limit the size of the log file by entering a non-zero value in the **Maximum file size (MB)** field. Enter 0 if you do not want to limit the file size. If the log file grows bigger than the size limit, ModMaster will rename the log file by adding the suffix ".bak", and start a new log file with the specified name.

Messages of level **Info** and above are also displayed in ModMaster's status bar.

Tracing

ModMaster can trace all raw data that is sent or received, each Modbus message that is sent or received, and each register value read or written. The Tracing page is used to configure the categories of tracing.

The trace output is logged at **Trace** level (and so can normally be viewed using the **View** → **Log** menu option). Trace output can be turned on and off using the **Trace** button on the toolbar.

Raw data tracing

To enable tracing of raw data, select the **Tracing** → **Trace raw data** checkbox.

If raw tracing is enabled, all data sent or received is displayed as lines in the log. Each line begins with the time that the data was sent or received. This is followed by the character '<' for incoming data or the character '>' for outgoing data, and then the sent or received data in hex.

Message tracing

To enable message tracing, select the **Tracing** → **Trace messages** checkbox.

If message tracing is enabled, each Modbus message sent or received is logged as two lines.

The first line begins with the time that the message was sent or received. This is followed by the character '<' for an incoming message or the character '>' for an outgoing message. The '<' or '>' is followed by the transaction identifier (only for TCP packets), slave identifier, PDU length (in bytes), function code, and then the first few bytes of the body of the message (in hex).

The second line of a message trace provides an interpretation of the main fields in the message. This line also begins with the time. After the time is the word "Request" for a request message, or "Response" for a normal response message. "Request" or "Response" is followed by the function code (expressed in words) and the main fields of the message (address, count, etc.).

Here is an example of an outgoing and an incoming message:

```
16:11:20.324: > transid 5 slave 1 pdulen 5 func 3: 00 c8 00 01
16:11:20.325:    Request: Read Holding Registers: address 200, count 1
16:11:20.326: < transid 5 slave 1 pdulen 4 func 3: 02 00 00
16:11:20.327:    Response: Read Holding Registers: byte count 2, data bytes 2
```

If the message is an error response, the second line will contain "Error response" followed by the error code (expressed in words). For example:

```
16:23:43.186: > transid 10 slave 1 pdulen 5 func 3: 00 64 00 01
16:23:43.188:    Request: Read Holding Registers: address 100, count 1
16:23:43.192: < transid 10 slave 1 pdulen 2 func 131: 02
16:23:43.193:    Error response: Illegal data address
```

If ModMaster receives any data that is not correctly formatted as a Modbus message, it will also trace this data. The data is displayed in hex on a separate line, preceded by the time and the character '<'. This line is followed by a line with the word "Discarded" followed by the number of bytes discarded and an explanation of what is wrong with the format. For example:

```
15:18:13.554: < 00 01 00 00 00 06 01 03 00 64 02 01
15:18:13.555:    Discarded 12 bytes: CRC failed
```

Troubleshooting help is available for the "Error response" and "Discarded" lines. Click on the line in the log window (or move to it with the keyboard and press space) to display the help.

Register value tracing

To enable tracing of register reads, select the **Tracing** → **Trace values read** checkbox.

To enable tracing of register writes, select the **Tracing** → **Trace value written** checkbox.

If tracing is enabled, each read or write of a defined register (see Registers and File Registers) is traced as a single line in the log. Each line begins with the time of the read or write. The time is followed by the character 'R' for a register read or 'W' for a register write. This is followed by the slave ID, the register address and the value read or written. For example (with message tracing also enabled):

```
17:41:12.272: < transid 8 slave 1 pdulen 5 func 3: 00 11 00 03
17:41:12.273:       Request: Read Holding Registers: address 17, count 3
17:41:12.276: > transid 8 slave 1 pdulen 14 func 3: 0c 00 00 00 1d 00 00 01 55 41 ...
17:41:12.277:       Response: Read Holding Registers: byte count 12, data bytes 12
17:41:12.281:         R  1  17  29
17:41:12.282:         R  1  18  341
17:41:12.284:         R  1  19  29.230000
```

The register address for ordinary registers is a "model" address (see Address mapping). The register address for a file register consists of the file number and register number separated by ':'. For example (with message tracing also enabled):

```
17:37:08.552: < transid 6 slave 1 pdulen 16 func 20: 0e 06 00 03 00 03 00 03 06 27 ...
17:37:08.553:       Request: Read File Record
17:37:08.557: > transid 6 slave 1 pdulen 16 func 20: 0e 07 06 85 2b 00 24 03 75 05 ...
17:37:08.558:       Response: Read File Record
17:37:08.565:         R  1  3:3  34091
17:37:08.567:         R  1  3:4  36
17:37:08.568:         R  1  3:5  885
17:37:08.569:         R  1  10000:102  4486
17:37:08.570:         R  1  10000:103  20371
```

A. Troubleshooting

This section provides information on various errors that can occur when using ModMaster, and suggests ways of tackling them.

Entry headings in quotes are messages that are displayed in ModMaster's status bar or log output. For other entries, the heading is a description of the symptoms.

The easiest way to find a message in this section is to click on the **Error Help** button in the status bar (or press the F4 key), or to click on the message in the log window (or move to the message and press the space bar).

Communications

General

"Connection closed"

- Click the **Run** button to open the connection.
- If you have lost the connection to the slave check that the slave is still running and listening.

"Timed out"

A request was sent to the slave, but no response was received.

- Turn on tracing to get more information. Check whether the response was received late, and increase the **Response Timeout** if necessary.
- Check that the **Packet Type** is consistent with the slave. The packet type is usually **TCP** for a socket connection, and **RTU** or **ASCII** for a serial connection.
- Check that the **Slave ID** is consistent with the slave.
- If using a socket connection, Check that **Host** and **Port** are consistent with the slave. You may have sent the request to the wrong slave/server!
- If using UDP, check that the slave is running/listening.
- If you sent a broadcast request using a non-standard protocol variant, check that the **Always responds** setting is consistent with the slave.
- Check that the **Slave ID** is consistent with the slave.
- If you are using a serial connection and the master and slave are on same machine, check that the **ports** are different.
- If you are using a serial connection, check that the **Speed**, **Parity**, **Data bits**, **Stop bits** and **RTS control** settings are consistent with the slave.
- If you are using a serial connection, check that a cable is connecting master and slave interfaces!
- If you are using a serial connection with **RTS control: Flow control** check that the cable is appropriately wired.
- If you are using a serial connection, check that no other software is using the same **port**.

"Error response: ..."

ModMaster received an error response from the slave.

- The rest of the message says what kind of error response. See the troubleshooting entry for the rest of the message for further explanation.

- If tracing is turned on, you can view the received message in the log window.

"Invalid data received: ..."

Data was received by ModMaster that was unexpected or not correctly formatted as a Modbus message. The received data is discarded without further processing.

- The rest of the message says what was wrong with the received data. See the troubleshooting entry for the rest of the message for further explanation.
- If tracing is turned on, you can view the received data in the log window.

"Discarded: ..."

This message is displayed in the log window when data is received by ModMaster that is unexpected or not correctly formatted as a Modbus message. The received data is discarded without further processing.

- The rest of the message says what was wrong with the received data. See the troubleshooting entry for the rest of the message for further explanation.
- The received data is displayed in the log window preceding the "Discarded" message.

"PDU size (XXX bytes) exceeds maximum (XXX bytes)"

A received message exceeded the arbitrary limit on PDU size. The message is discarded without further processing.

- Turn on tracing to get more information.
- Increase the **Max PDU size** if you want to be able to receive longer messages and don't mind not conforming to the current Modbus specification.

Socket Communications

"Unknown host: XXX"

The host name that you entered could not be mapped to an IP address.

- Check that the **Host** is consistent with the slave.
- Check that your DNS server is working and reachable.

"Unknown local host: XXX"

The local host name that you entered could not be mapped to an IP address.

- Check that the local host name is correct. Leave the local host field empty if you are not sure.
- Check that your DNS server is working and reachable.

"Can't bind to port XXX"

The socket could not be bound to the local host and port that you entered.

- Check that the **Local host** corresponds to a network interface on the computer on which ModMaster is running.
- Check that you are permitted to use the **Local port**, and that it is not already being used.

"Can't connect to host 'XXX': No route to host"
"Can't connect to host 'XXX': Connect timed out"

The host did not respond to the TCP connection request

- Check that the **Host** is consistent with the slave.

- Check that the network is working.

"Can't connect to host 'XXX' port XXX: Connection refused"

The host appears to be reachable and responding, but would not accept the TCP connection request.

- Check that the **Host** and **Port** are consistent with the slave.
- Check that slave is listening.

"UDP Port Unreachable"

It was not possible to deliver a UDP message.

- Check that the **Host** and **Port** are consistent with the slave.
- Check that slave is listening.

"Error response: No path available to target"

The server/bridge sent a response indicating that the required slave was not reachable.

- Check that the **Slave ID** is consistent with the slave.
- Check that your server/bridge is correctly configured for the slave you are trying to reach.

"Error response: Target device failed to respond"

The server/bridge sent a response indicating that the required slave did not respond.

- Check that the **Slave ID** is consistent with the slave.
- Check that your server/bridge is correctly configured for the slave you are trying to reach.

Serial Communications

"Serial comms not available"

Serial communications is not supported on the system you are using.

"No serial port specified"

- You must select or enter a port name in the interface page.

"Can't open serial port"

The serial port could not be opened and configured correctly.

- Check that you have selected/entered the correct serial port name.
- On Unix/Linux systems, check that you have read and write access permission to the device file ('/dev/...').
- Check that the port is not in use by another program.

"Can't set serial speed"
"Can't set serial data bits"
"Can't set serial stop bits"
"Can't set serial parity"
"Can't set serial RTS control"

The serial port parameter could not be set.

- Check that the serial device supports the parameter value that you are trying to set.

"Parity error"

A parity error occurred when receiving data via the serial port.

- Check that the **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.

"Framing error"

A framing error occurred when receiving data via the serial port.

- Check that the **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.

"Overrun error"

An overrun error occurred when receiving data via the serial port. In other words, data was sent faster than it could be received by the machine running ModMaster.

- Try using hardware flow control, if it is supported.
- Try using a slower speed.

RTU Packet Type

"CRC failed"

Data received by ModMaster failed the Cyclic Redundancy Check (CRC).

- Turn on tracing to get more information.
- If using serial communications, check that **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.
- Try increasing the **EOM Timeout** setting - messages may be getting fragmented by long delays introduced by the comms link or the operating system.

"Message too short (XXX bytes)"

An RTU message must be at least 4 bytes long (1 byte for the slave ID, 1 byte for the function code, and 2 bytes for the CRC).

- Turn on tracing to get more information.
- If using serial communications, check that **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.
- Try increasing the **EOM Timeout** setting - messages may be getting fragmented by long delays introduced by the comms link or the operating system.

"Message too long (> XXX bytes)"

A received message was too long to fit in ModMaster's input buffer.

- Turn on tracing to get more information.
- If using serial communications, check that **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.

ASCII Packet Type

"No ':' at start of message"
"':' in middle of message"
"LF terminator missing"
"Invalid message length (XXX bytes) - should be odd"
"Non-hex character in message"
"Invalid LRC"

Data received by ModMaster was not correctly formatted as an ASCII message.

- Turn on tracing to get more information.
- If using serial communications, check that **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.

"Message too short (XXX bytes)"

An ASCII message must be at least 9 bytes long.

- Turn on tracing to get more information.
- If using serial communications, check that **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.
- Try increasing the **EOM Timeout** setting - messages may be getting fragmented by long delays introduced by the comms link or the operating system.

"Message too long (> XXX bytes)"

A received message was too long to fit in ModMaster's input buffer.

- Turn on tracing to get more information.
- If using serial communications, check that **Speed**, **Parity**, **Data bits** and **Stop bits** are all set correctly.

TCP Packet Type

"Incomplete message: only XXX bytes"

A TCP message must be at least 8 bytes long (7 bytes for the header and 1 byte for the function code).

- Turn on tracing to get more information.
- Try increasing the **EOM Timeout** setting - messages may be getting fragmented by long delays introduced by the comms link or the operating system.

"Incorrect protocol identifier: XXX"

The protocol identifier in the received message header was not zero.

"Length (XXX) too small"

The value in the length field of the received message header must be at least 2 (1 byte for the slave ID and 1 byte for the function code).

"Incomplete message: only XXX bytes when expecting XXX"

The received message was shorter than indicated by the length field of the header.

- Turn on tracing to get more information.
- Try increasing the **EOM Timeout** setting - messages may be getting fragmented by long delays introduced by the comms link or the operating system.

Sending commands

Value is read from or written to address in slave that is wrong but near to address in command

- Check that the address mapping is consistent with the slave.
- If you are using 32 or 64 bit registers, check that **Word Registers** setting is consistent with the slave.
- Turn on tracing to get more information.

- If you are using polling or a register read/write button, try a defined command (e.g. **Define Command** → **Read Holding Registers**) to get more information.

Wrong value is displayed when reading from slave, or wrong value is written to slave

- Check that the register **Type** is consistent with the slave.
- If you are using 32 or 64 bit registers, check that **Little-Endian** and **Word Registers** settings are consistent with the slave.
- Turn on tracing to get more information.
- If you are using polling or register read/write button, try a defined command (e.g. **Define Command** → **Read Holding Registers**) to get more information.

"Unexpected"

An unexpected message was received by ModMaster.

- The message may be a delayed response to an earlier request sent by ModMaster (this should be apparent in the trace output). You may need to increase the response timeout setting if the slave or the comms link is slow.

"Late response"

A late response was received by ModMaster.

- A response to a request sent by ModMaster was received after the timeout period. You may need to increase the response timeout setting if the slave or the comms link is slow.

"Invalid read count (XXX) - response byte count (XXX) would not fit in a byte"

The request that you are attempting to send contains a read count that is too large. The response would have to contain more than 255 bytes of data, and the byte count would then not fit into the single byte reserved for it in the response message.

- If you think you should be able to send the request and you are using 32 or 64 bit registers, check that the **Word Registers** and **Word Count** settings are consistent with the slave.
- If your slave can handle messages containing more than 255 data bytes, and you don't mind not conforming to the current Modbus specification, you could select Allow Long Messages to enable the request to be sent.

"Invalid read count (XXX) - response would exceed max PDU size (XXX bytes)"

The request that you are attempting to send contains a read count that is too large. The response would exceed the arbitrary limit on the PDU size.

- If you want to send the request and don't mind not conforming to the current Modbus specification, increase the **Max PDU Size** setting.

"Invalid read count (XXX) - exceeds count limit (XXX bytes)"

The request that you are sending contains a read count that exceeds the arbitrary count limit (interpreted as a limit on the number of data bytes) imposed by the current Modbus specification.

- You can suppress this warning by deselecting the **Check Count Limits** setting.

"Invalid read count (XXX) - will not fit in 16-bits"

The request that you are attempting to send contains a read count that is too large to fit in the 16-bit field of the request message.

"Invalid read count (0)"

The request that you are sending contains a zero read count, which is not allowed by the current Modbus specification.

- You can suppress this warning by deselecting the **Strict Checking** setting.

"Invalid write count (XXX) - byte count (XXX) would not fit in a byte"

The request that you are attempting to send contains a write count that is too large. The request would have to contain more than 255 bytes of data, and the byte count would then not fit into the single byte reserved for it in the request message.

- If you think you should be able to send the request and you are using 32 or 64 bit registers, check that the **Word Registers** and **Word Count** settings are consistent with the slave.
- If your slave can handle messages containing more than 255 data bytes, and you don't mind not conforming to the current Modbus specification, you could select Allow Long Messages to enable the request to be sent.

"Invalid write count (XXX) - request would exceed max PDU size (XXX bytes)"

The request that you are attempting to send contains a write count that is too large. The request would exceed the arbitrary limit on the PDU size.

- If you want to send the request and don't mind not conforming to the current Modbus specification, increase the **Max PDU Size** setting.

"Invalid write count (XXX) - exceeds count limit (XXX bytes)"

The request that you are sending contains a write count that exceeds the arbitrary count limit (interpreted as a limit on the number of data bytes) imposed by the current Modbus specification.

- You can suppress this warning by deselecting the **Check Count Limits** setting.

"Invalid write count (0)"

The request that you are sending contains a zero write count, which is not allowed by the current Modbus specification.

- You can suppress this warning by deselecting the **Strict Checking** setting.

"Invalid write count (XXX) - will not fit in 16-bits"

The request that you are attempting to send contains a write count that is too large to fit in the 16-bit field of the request message.

"Error response: Illegal data address"

The slave has sent an exception response indicating that the address in the command sent by ModMaster was incorrect.

If you are using polling or register read/write button:

- Check that the slave has registers corresponding to the registers defined in ModMaster.
- Check that the addresses of the registers defined in ModMaster are consistent with the slave.
- Check that the address mapping is consistent with the slave.
- Check that the slave allows the registers to be written, if you are trying to write.
- Turn on tracing to see if the correct command is being sent. If not, check that the registers defined in ModMaster are in the correct address ranges.
- Try a defined command (e.g. **Define Command** → **Read Holding Registers**) to get more information.

If you are using a defined command:

- Check that the address you have entered is a message address and not a model address.
- Check that the slave has a Holding-register/Input-register/Coil/Discrete-input at the **Address** you have entered.
- Check that the slave has enough consecutive addresses for the **Count** you have entered.
- Check that the slave allows the registers/coils to be written, if you are trying to write.

"Error response: Illegal data value"

The slave has sent an exception response indicating that data in the command sent by ModMaster was incorrect.

- Turn on tracing to get more information.
- If you are using 32 or 64 bit values check that the **Word Count** setting is consistent with the slave. If ModMaster and the slave disagree over how to interpret the count in the Modbus message, then the number of bytes written by ModMaster may be different from what the slave is expecting.

If you are using polling:

- Check that the **Type**s of the registers defined in ModMaster are consistent with the slave.
- Try a defined command (e.g. **Define Command** → **Write Multiple Holding Registers**) to get more information.

If you are using a defined command:

- Check that the **Type** is consistent with the slave.
- Check that the **Count** you have entered does not exceed the maximum supported by the slave.

"Error response: Illegal function"

The slave has sent an exception response indicating that the function code in the command sent by ModMaster is not supported.

- If you are using polling or a register read/write button, turn on tracing to see if the correct command is being sent. If not, check that the registers defined in ModMaster are in the correct address ranges.

"Error response: Server failure"
"Error response: Acknowledge"
"Error response: Server busy"
"Error response: Memory parity error"
"Error response: Error code XXX"

The slave/bridge/server sent an unexpected error response.

- Check that your slave/bridge/server is functioning correctly.

"Wrong transaction ID in response: XXX instead of XXX"

The slave sent a response containing a transaction ID that was different from the transaction ID that ModMaster sent in the request. This may have occurred because the slave responded late to an earlier request.

- Turn on tracing to get more information.
- Increase the **Response Timeout** in General settings if necessary.
- Check that your slave is functioning correctly.

"Wrong function code in response: XXX instead of XXX"

The slave sent a response containing a function code that was different from the function code that ModMaster sent in the request. This may have occurred because the slave responded late to an earlier request.

- Turn on tracing to get more information.
- Increase the **Response Timeout** in General settings if necessary.
- Check that your slave is functioning correctly.

"Response is not echo of request"
"Wrong slave ID in response: XXX instead of XXX"
"Wrong address in response: XXX instead of XXX"
"Wrong count in response: XXX instead of XXX"
"Wrong value in response"
"Wrong masks in response"
"Wrong subfunction in response: XXX instead of XXX"
"Wrong 'MEI Type' in response: XXX instead of 14"
"Wrong 'Read Device ID Code' in response: XXX instead of XXX"

The slave sent a response that did not correctly echo the data that ModMaster sent in the request.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.
- If you want ModMaster to accept the response even though it is incorrect, you could deselect **Strict Checking**.

"Invalid 'Conformity Level' in response: XXX"

The slave sent a 'Read Device Identification' response containing an invalid 'Conformity Level' value. The only valid values are 01, 02, 03, 81, 82 and 83 (in hex).

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.
- If you want ModMaster to accept the response even though it is incorrect, you could deselect **Strict Checking**.

"Invalid 'More Follows' in response: XXX"

The slave sent a 'Read Device Identification' response containing an invalid 'More Follows' value. The only valid values are 00 and FF (in hex). 'More Follows' must be 00 if the 'Read Device ID Code' is 04 (individual access).

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.
- If you want ModMaster to accept the response even though it is incorrect, you could deselect **Strict Checking**.

"Invalid 'Next Object ID' in response: XXX instead of 0"

The slave sent a 'Read Device Identification' response containing an invalid 'Next Object ID' value. The 'Next Object ID' value must be zero if 'More Follows' is zero.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.
- If you want ModMaster to accept the response even though it is incorrect, you could deselect **Strict Checking**.

"Invalid 'Number of Objects' in response: XXX instead of 1"

The slave sent a 'Read Device Identification' response containing an invalid 'Number of Objects' value. The 'Number of Objects' value must be 1 if the 'Read Device ID Code' is 04 (individual access).

- Turn on tracing to get more information.

- Check that your slave is functioning correctly.

"Response PDU size incorrect: XXX instead of XXX"

The PDU size of a received response was not correct for the function code specified in the response. If the response included a byte count (e.g. a Read Holding Registers response) then the PDU size is inconsistent with the byte count. If the response did not include a byte count (e.g. a Write Multiple Holding Registers response), then the PDU size was not the expected fixed size.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.

"Response PDU too short: XXX when it should be at least XXX"

The PDU size of a received response was not big enough to contain all the fields required for the function code specified in the response.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.

"Wrong byte count in response: XXX when expecting XXX"

The byte count in the response sent by the slave is not what was expected for the count that ModMaster sent in the request.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.
- If you want ModMaster to accept the response even though it is incorrect, you could deselect **Strict Checking**.

"Byte count too small: XXX when expecting at least XXX"

The byte count in the response sent by the slave too small for a valid response.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.

"Byte count too big: XXX when expecting at most XXX"

The byte count in the response sent by the slave too big for a valid response.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.

"Incorrect reference type: XXX instead of 6"

The slave sent a Read File Record (function code 20) response containing an incorrect reference type.

- Turn on tracing to get more information.
- Check that your slave is functioning correctly.

"Wrong sub-response length: XXX instead of XXX"

The slave sent a Read File Record (function code 20) response containing a sub-response length that does not match the length in the request.

- Turn on tracing to get more information.

- Check that your slave is functioning correctly.

"Queue count exceeds 31: XXX"

The slave sent a Read FIFO Queue (function code 24) response containing a queue count greater than 31. A FIFO queue size must be in the range 0 to 31.

- Turn on tracing to get more information.
- Check that your master is functioning correctly.

"Invalid command: broadcasting a read request"

You are sending a read request with a zero slave ID, which is used to indicate a broadcast request. The Modbus specification only allows write requests to be broadcast.

Miscellaneous

"Inconsistent sizes for coils"

Only one size of register (e.g. 1-bit, 32-bit) is allowed within the address range defined for **Coils** (see Address mapping). The **Coil** address range that you are attempting to define (or load from a file) includes registers with different sizes.

- If the **Coil** address range overlays the **Input** or **Holding Registers** address range, then registers in the overlayed part are also restricted to one size.
- If your slave device uses Coils, check that all are defined in ModMaster with the same size.
- If your slave device does not use Coils, set **Number of Addresses** to 0 in the address map.

"Inconsistent size for coil"

Only one size of register (e.g. 1-bit, 32-bit) is allowed within the address range defined for **Coils** (see Address mapping). The register that you are attempting to add (or load from a file) has a different size from those already in the address range.

- If the **Coil** address range overlays the **Input** or **Holding Registers** address range, then registers in the overlayed part are also restricted to one size.
- If your slave device uses Coils, check that all are defined in ModMaster with the same size.
- If your slave device does not use Coils, set **Number of Addresses** to 0 in the address map.

"Inconsistent sizes for discrete inputs"

Only one size of register (e.g. 1-bit, 32-bit) is allowed within the address range defined for **Discrete Inputs** (see Address mapping). The **Discrete Inputs** address range that you are attempting to define (or load from a file) includes registers with different sizes.

- If the **Discrete Input** address range overlays the **Input** or **Holding Registers** address range, then registers in the overlayed part are also restricted to one size.
- If your slave device uses Discrete Inputs, check that all are defined in ModMaster with the same size.
- If your slave device does not use Discrete Inputs, set **Number of Addresses** to 0 in the address map.

"Inconsistent size for discrete input"

Only one size of register (e.g. 1-bit, 32-bit) is allowed within the address range defined for **Discrete Inputs** (see Address mapping). The register that you are attempting to add (or load from a file) has a different size from those already in the address range.

- If the **Discrete Input** address range overlays the **Input** or **Holding Registers** address range, then registers in the overlayed part are also restricted to one size.
- If your slave device uses Discrete Inputs, check that all are defined in ModMaster with the same size.
- If your slave device does not use Discrete Inputs, set **Number of Addresses** to 0 in the address map.

"Address XXX is not in any address range"

The address of the register that you are attempting to add (or load from a file) is not within any address range (see Address mapping).

- If the register address is correct, check that the appropriate address range includes the address.
- You must use a "model address" when defining a register, not a "message addess".

"Addresses XXX are not in any address range"

The addresses of the registers that you are attempting to add (or load from a file) are not within any address range (see Address mapping).

- If the register addresses are correct, check that the appropriate address range includes the addresses.
- You must use "model addresses" when defining registers, not "message addesses".

"Registers XXX and XXX overlap"

When you select **Word Registers** each 32-bit and 64-bit register requires more than one address. This error occurs when an extra address required by a 32-bit or 64-bit register is used by another register.

Coils: The value appears not to have been written or has been written to the wrong byte

- Select/deselect **Swap Bytes** to make the order of bytes of a 16-bit word consistent with the slave.

Coils/Discrete Inputs: bit values are in reverse order within each byte

- Select/deselect **Reverse Bits** to reverse the order and make it consistent with the slave.

Coils/Discrete Inputs: byte values are in reverse order in 16-bit words

- Select/deselect **Swap Bytes** to reverse the order and make it consistent with the slave.

The 16-bit words of 32 or 64-bit values are in reverse order

- Select/deselect **Little Endian** to reverse the order and make it consistent with the slave.

"Not enough data in message: XXX bytes when expecting XXX"

A received message contains too little data for the count and register sizes specified in the request.

- Turn on tracing to get more information.
- If you are using 32 or 64 bit registers, check that the **Word Registers** and **Word Count** settings are correct.
- Check that the **Type**s of the registers being transferred are specified correctly.

"Excess data in message: XXX bytes when expecting XXX"

A received message contains too much data for the count and register sizes specified in the request.

- Turn on tracing to get more information.
- If you are using 32 or 64 bit registers, check that the **Word Registers** and **Word Count** settings are correct.

- Check that the **Types** of the registers being transferred are specified correctly.
- If you want ModMaster to accept the message even though it is incorrect, you could deselect **Strict Checking**.

B. Release Notes

3.11 - 2 Nov 2018

Resolved issues arising from changes in recent releases of Java JRE.

3.10 - 17 Jan 2017

Added support for serial interfaces on ARM processor on Raspberry Pi.

Minor documentation changes.

3.09 - 21 Oct 2016

Fix for problem loading serial library.

3.08 - 1 Dec 2015

Fixed a problem with broadcasting custom commands.

3.07 - 30 Oct 2015

Fixed a GUI problem with defining a Write Holding Register command.

3.06 - 4 Feb 2015

Fixed a problem with handling of zero counts in commands.

3.05 - 29 Jan 2015

Fixed some problems with handling of invalid counts in commands.

3.04 - 20 Jan 2015

Fixed a timeout problem when reading raw data from serial interfaces.

3.03 - 6 Jan 2015

Added a "mirror" field for viewing and entering lengthy values.

Workaround for reading CSV files that have been edited by Microsoft utilities.

Minor documentation changes.

3.02 - 15 Dec 2014

Fix for a bug that could cause write delays on serial connections.

3.01 - 18 Sep 2014

Support for additional Modbus commands:

- 12 Get Comm Event Log
- 20 Read File Record
- 21 Write File Record

- 24 Read FIFO Queue

All official Modbus commands are now supported.

Logging/tracing to a file.

Changes to handling of unsigned values.

Configurable limits for register values, with highlighting of values out of range.

Better diagnostics for late responses.

Various GUI changes.

2.25 - 29 Jul 2013

Under Windows, scan for serial ports up to COM99 instead of COM20.

2.24 - 10 Jun 2013

Minor changes to registration dialogs.

2.23 - 15 May 2013

Added support for higher and custom serial speeds.

Minor GUI changes.

2.22 - 9 Apr 2013

Minor GUI changes.

2.21 - 9 Jan 2013

Minor changes to trace and error messages.

2.20 - 25 Sep 2012

Report conformity level in Read Device ID response.

Report opening/closing of connections in status bar.

2.19 - 21 Sep 2012

Allow configuration of the local host and local port for socket connections to slaves.

2.18 - 15 Aug 2012

Fixed a problem with loading of serial library under Linux.

2.16 - 14 Jul 2012

Minor documentation changes.

2.15 - 8 Jun 2012

Minor changes to registration dialogs.

2.14 - 2 May 2012

New serial library to replace the Java Comms API. Provides 32-bit and 64-bit support under Windows and Linux.

2.12-2.13 - 11 Jan 2012

Minor changes.

2.11 - 11 Oct 2011

Minor serial comms changes: don't use customer's copy of Java Comms API, and work around a bug in a Windows device driver.

C. Modbus protocol documents

The Modbus protocol was invented by Gould Modicon (now a division of Schneider Electric), and is now administered by modbus.org. Gould-Modicon/Schneider/modbus.org have published five documents describing the Modbus protocol:

Modbus Protocol Reference Guide. PI-MBUS-300. Rev J. June 1996. This is the original Modbus specification, which has been superceded by the current Modbus specification and the serial specification listed below. This document is referred to as the "original Modbus specification" in this manual.

Open Modbus/TCP Specification. Andy Swales. Release 1.0. March 1999. This document specifies how Modbus messages are sent over a network using the TCP/IP protocols. It also contains some intelligent commentary on the Modbus protocol, and an appendix on client and server implementation, which may be useful if this area is new to you. Surprisingly, this document does not appear to be available from the official Modbus web-site. This document is referred to as the "Modbus/TCP specification" in this manual.

Modbus Application Protocol. V1.1b3. April 2012 This is the current definition of Modbus function codes and the format of the corresponding request and response messages. It does *not* cover how these messages are packaged in the RTU, ASCII or TCP variants of the protocol. This specification is referred to as the "current Modbus specification" in this manual.

Modbus Messaging on TCP/IP Implementation Guide. Rev 1.0b. December 2006 This consists mainly of what appear to be design notes for Schneider Automation's implementation of Modbus/TCP. You will probably find it hard to read if you don't already have a good understanding of the area, and not very useful if you do. It is, however, the only document on the official Modbus web-site that contains (in section 3.1) a description of the Modbus/TCP protocol. This document is referred to as the "Modbus/TCP implementation guide" in this manual.

Modbus over Serial Line - Specification & Implementation Guide. V1.02. December 2006 This is the current definition of the RTU and ASCII variants of the Modbus protocol. It also contains physical layer (electrical) specifications. This document is referred to as the "serial specification" in this manual.

D. CSV file format

This appendix describes the CSV file format used by the **Export Registers**, **Import Registers**, **Export File Registers** and **Import File Registers** items on the **File** menu (see Saving and restoring register definitions and Saving and restoring file register definitions).

Files written by **Export Registers** and **Export File Registers** follow these rules:

- Each line of the file consists of comma-separated fields.
- Each line contains the same number of fields.
- Each field is enclosed in double-quotes.
- A double-quote within a field is represented by two consecutive double-quotes. For example, the string '"Real" pressure' would be written as:

  """"Real"" pressure"

- The first line of the file is a header line containing the names of the fields:

  "Address","Name","Type","Value","Unit","Radix","Offset","Scale","Min","Max","Write"

  for **Export Registers** and

  "File","Address","Name","Type","Value","Unit","Radix","Offset","Scale","Min","Max","Write"

  for **Export File Registers**.

- Each subsequent line contains the values for a register, e.g.:

  "3049","Alarm 4","int16","0","","10","0.0","1.0","false"
  "3058","New data flag","int16","0","","10","0.0","1.0","true"

- Each Type field contains "int1" (for **Export Registers** only), "int8", "uint8", "int16", "uint16", "int32", "uint32", "int64", "uint64", "float32" or "float64".
- The Value, Scale, Offset, Min and Max fields contain values in decimal.
- Each Write field contains "true" or "false".
- There are no space characters in the file, except within double-quotes when the field value is a string that contains spaces.
- UTF-8 character encoding is used.

Import Registers and **Import File Registers** are fairly tolerant of the format of the files they will read:

- The first line of the file must be a header line containing the names of the fields.
- The field names do not have to be in the same order that **Export Registers** and **Export File registers** use.
- The field names do not all have to be present. The only required field names are "Address" for **Import Registers**, and "File" and "Address" for **Import File Registers**. Default values will be provided for any missing fields.
- The field names "File", "Address", "Name", "Type", "Value", "Unit", "Radix", "Offset", "Scale", "Min", "Max" and "Write" are recognized. "RW" is accepted as an alternative to "Write". Unrecognized field names, and their corresponding values, are ignored.
- "discrete" and "disc/coil" are accepted as alternatives to "int1" in the Type field.
- Case is not significant in field names.
- Subsequent lines in the file must contain register values corresponding to the fields named on the first line.
- Double-quotes are only necessary around fields that contain commas or double-quotes, or that have leading or trailing spaces.
- Blank lines and lines that start with '#' are ignored.
- Leading and trailing space characters in fields are ignored (except within double-quotes).
- The Unix style of escaping using backslash is accepted in fields that are not enclosed in double-quotes. For example, the field:

```
Pressure\, \"Real\"
```

would be read as 'Pressure, "Real"'.

- The Value, Scale, Offset, Min and Max fields must contain values in decimal.
- Default values are provided for empty fields, except for the File and Address fields (which must contain values).
- UTF-8 character encoding is assumed. ASCII is accepted, since it is a subset of UTF-8.

E. Limits on Modbus message sizes

Limits on the size of Modbus messages are confusing and have a chequered history. Take the Write Multiple Coils request (function code 15) as an example:

- The Write Multiple Coils request contains a count of the number of bytes of data to be written. This byte count is stored in a single byte of the request, and so cannot exceed 255. Each byte of data can contain 8 bits (coils), so at most **2040** (255 * 8) coils can be written.
- The original Modbus specification imposed no other limits on the size of a Write Multiple Coils request, and so implicitly allowed 2040 coils to be written. However, an appendix listed implementation limits for Modicon controllers, which varied from 32 to 800 coils.
- The Modbus/TCP specification introduced arbitrary limits on the counts of some Modbus requests. It specified a limit of **800** coils in a Write Multiple Coils request.
- Version 1 of the current Modbus specification introduced the term "PDU", and imposed arbitrary limits on the PDU size. The PDU for serial communications was limited to 253 bytes, and the PDU for TCP communications was limited to 249 bytes. The PDU for a Write Multiple Coils request requires 6 bytes for the function code, starting address, etc. For serial comms, this leaves 247 (253 - 6) bytes of coil data, which allows **1976** (247 * 8) coils to be written. For TCP comms, there could be 242 (249 - 6) bytes of coil data, which allows **1936** (242 * 8) coils to be written.

  The arbitrary count limit on the Write Multiple Coils request was increased from 800 to **1968** coils.
- Version 1.1b of the current Modbus specification changed the TCP PDU limit to 253 bytes, making it the same as the serial PDU limit. This would allow 1976 coils to be written using TCP, if it were not for the arbitrary count limit (still at 1968).

Some recommendations for dealing with this confusion:

- If you want ModMaster to conform to the current Modbus specification, set the **Max PDU Size** to 253, and select the **Check Count Limits** checkbox.
- If you are designing a Modbus slave, ignore the arbitrary limits on counts and the PDU size (for example, you should allow 2040 coils to be written in a Write Multiple Coils request - if the device has that many coils). This will allow your slave to work with the maximum number of Modbus masters.

  In order to test the slave, set ModMaster's **Max PDU Size** to 265 or greater, and deselect the **Check Count Limits** checkbox.
- If you are designing a Modbus master, you should, by default, conform to the arbitrary count and PDU limits of the current Modbus specification. You will probably also need to provide an option or options for reducing request sizes, since many slave devices cannot cope with large request sizes.