INDUSTRIAL CONTROL COMMUNICATIONS, INC.

Modbus/TCP Server Driver Manual

October 30, 2014

TABLE OF CONTENTS

1 Modbus/TCP Server

1.1 Overview

The Modbus/TCP server driver can be used to allow internal database access from any device which supports the Modbus/TCP client protocol. This driver also supports the 32-bit extension to the Modbus standard (commonly referred to as the “Enron/Daniel” extension).

Other notes of interest are:

- Database data can be accessed as either holding registers (4X references) or input registers (3X references).
- Specific bits within the database can be accessed as either coils (0X references) or discrete inputs (1X references).
- The driver is conformance class 0 and partial class 1 and class 2 compliant. Supported Modbus/TCP server functions are indicated in Table 1.

Table 1: Supported Modbus/TCP Server Functions

Function Code	Function	Modbus/TCP Class
01	Read coils	1
02	Read discrete inputs	1
03	Read holding registers	0
04	Read input registers	1
05	Write single coil	1
06	Write single register	1
15	Write multiple coils	2
16	Write multiple registers	0

- Supports up to 8 simultaneous Modbus/TCP server connections (sockets).
- 32-bit register accesses are supported in a variety of options and formats.
- The “unit identifier” (UI) field of the request packets is ignored.
- Configuration tip: Improved network utilization may be obtained by appropriately grouping contiguous register assignments in the database. In this way, the “read multiple registers”, “read input registers” and “write multiple registers” functions can be used to perform transfers of larger blocks of registers using fewer Modbus transactions compared to a situation where the read/write registers were arranged in an alternating or scattered fashion.

1.2 Server Settings

Timeout Time

Defines the maximum number of milliseconds for a break in network communications before a timeout event will be triggered. To disable timeout processing, set this field to 0.

- If a particular open socket experiences no activity for more than the timeout time setting, then the driver assumes that the client or network has experienced some sort of unexpected problem, and will close that socket.
- Because the timeout determination is performed on a per-socket basis, note that a certain degree of caution must be exercised when using the network timeout feature to avoid "nuisance" timeouts from occurring. Specifically, do not perform inadvisable behavior such as sending a request from the client device, and then closing the socket prior to successfully receiving the server's response. The reason for this is because the server will experience an error when attempting to respond via the now-closed socket. Always be sure to manage socket life cycles "gracefully", and do not abandon outstanding requests.
- If a socket error occurs (regardless of whether the error was due to a communication lapse or abnormal socket error), the driver will trigger a timeout event.

Enable Supervisory Timer

The Connection Timeout Options group contains a selection to enable the driver's "supervisory timer" function. This timer provides the ability for the driver to monitor timeout occurrences between successive Modbus/TCP socket connections, as opposed to the standard timeout functionality which monitors timeout occurrences only within the scope of each client socket connection. While this feature provides an additional level of fail-safe functionality for those applications that require it, there are several ramifications that must be understood prior to enabling this capability. Before enabling this timer, therefore, it is suggested that users read the ICC whitepaper titled "*A Discussion of Modbus/TCP Server-Side Timeout Processing*", which can be found in the documents section at http://www.iccdesigns.com.

1.3 Node Settings

There are no node settings. A node is simply a container for objects.

1.4 Modbus Object Settings

1.4.1 Default Modbus Mapping

When the "Default Modbus Mapping" item is added to the node, predefined Modbus objects are automatically mapped to the internal database to provide convenient, configuration-free accessibility. User-defined Modbus Mapping objects can simultaneously co-exist with the default Modbus mapping.

1.4.1.1 Holding & Input Register Default Mapping

The default mapping provides read/write support for holding registers (4X references) and read-only support for input registers (3X references). Both holding registers and input registers access the same data. For example, reading Holding Register 4 returns the same data as reading Input Register 4. By default, registers are mapped into the database using the following scheme:

Register 1 is mapped to address 0,
Register 2 is mapped to address 2,
Register 3 is mapped to address 4,
:

Arithmetically, the default mapping register-to-address relationship can be described via Equation 1:

$$address = 2 \times (register - 1) \qquad \textbf{Equation 1}$$

For clarity, let's use Equation 1 in a calculation example. Let's say that we wish to read registers 24 and 25 when the default mapping is in place. Using Equation 1, we can calculate that register 24 is mapped to database address 46 and register 25 is mapped to database address 48. Therefore, reading registers 24 and 25 will return two 16-bit words of data from addresses 46 and 48 in the database, respectively.

1.4.1.2 Coil & Discrete Input Default Mapping

The default mapping provides read/write support for coils (0X references) and read-only support for discrete inputs (1X references). These will collectively be referred to from here on out as simply "discretes". Accessing discretes does not reference any new physical data: discretes are simply indexes into various bits of existing registers (specifically, coils index into holding registers and inputs index into status registers.) What this means is that when a discrete is accessed, that discrete is resolved by the driver into a specific register, and a specific bit within that register. The pattern of discrete-to-register/bit relationships can be described as follows:

Discrete 1...16 map to register #1, bit0...bit15 (bit0=LSB, bit15=MSB)
Discrete 17...32 map to register #2, bit0...bit15,
:

Arithmetically, the discrete-to-register/bit relationship can be described as follows: For any given discrete, the register in which that discrete resides can be determined by Equation 2:

$$register = \left\lfloor \frac{discrete + 15}{16} \right\rfloor \qquad \textbf{Equation 2}$$

Where the bracket symbols "$\lfloor\ \rfloor$" indicate the "floor" function, which means that any fractional result (or "remainder") is to be discarded, with only the integer value being retained.

Also, for any given discrete, the targeted bit in the register in which that discrete resides can be determined by Equation 3:

$$bit = (discrete - 1)\ \%\ 16 \qquad \textbf{Equation 3}$$

Where "discrete" $\in$[1…65535], "bit" $\in$[0…15], and "%" is the modulus operator, which means that any fractional result (or "remainder") is to be retained, with the integer value being discarded (i.e. it is the opposite of the "floor" function).

Conversely, for any bit in a register, the targeted discrete corresponding to that bit can be calculated by Equation 4:

$$discrete = 16 \times (register - 1) + bit + 1 \qquad \textbf{Equation 4}$$

For clarity, let's use Equation 2 and Equation 3 in a calculation example. Say, for instance, that we are going to read coil #34. Using Equation 2, we can determine that coil #34 resides in holding register #3, as $\lfloor 3.0625 \rfloor = \lfloor 3\ r1 \rfloor = 3$. Then, using Equation 3, we can determine that the bit within holding register #3 that coil #34 targets is (34-1)%16 = 1, as 33%16 = mod(2 r1) = 1. Therefore, reading coil #34 will return the value of holding register #3, bit #1. This example would unfold in the same manner should we choose instead to access discrete input #34, except that the value returned would be from input register #3, bit #1.

Note from the discussion above that discretes are mapped to registers, not database addresses. The location of a given register in the database determines what physical address the discrete will access. Because of this, it is possible to indirectly map discretes when the default mapping is in place by using register mapping objects. If a given register has been mapped to an alternate database address, then the discretes that map to that register will also be mapped to that alternate address.

1.4.2 Register Mapping Object Settings

Register mapping objects can be created to map holding and/or input registers to a different address in the database, or to map a register that is outside of the default mapping range into the database. There are three different types of register mapping objects:

- Holding register mapping objects, which map holding (4X) registers only.
- Input register mapping objects, which map input (3X) registers only.
- Holding/input register mapping objects, which simultaneously map both holding (4x) and input (3x) registers.

Because all three of these register mapping object types differ only in the register types on which they operate, their settings will be addressed together.

Register mapping objects can simultaneously co-exist with the default mapping described in section 1.4.1. When both items are included in a configuration and a register mapping object exists which maps a register to a database location that would otherwise be mapped to a different database location via the default mapping, then the register mapping object has priority. Said another way, register mapping objects are always evaluated first by the driver in order to ascertain the targeted database address. If no register mapping object exists for the targeted register, then the request is passed to the default mapping (if it is included in the configuration).

For further clarity regarding the simultaneous use of register mapping objects with the default mapping, let's assume we have defined a holding register mapping object to map holding register 25 to database address 62. This means that instead of holding register 25 mapping to database address 48 (as it would with the default mapping as described by Equation 1), it will now map to address 62. Now say we wish to read holding registers 24 and 25. We already know that holding register 25 maps to database address 62 (due to the holding register mapping object), so we must use Equation 1 to calculate the address that holding register 24 is mapped

to. Using the equation, we can determine that holding register 24 is mapped to database address 46. Therefore, reading holding registers 24 and 25 will return values from database addresses 46 and 62, respectively.

Note that in the above example we mapped only holding register 25 by using a “holding register mapping object”, versus a “holding/input register mapping object” that would map both holding register 25 and input register 25. This means that if we read input registers 24 and 25 using the above configuration, then both of these registers would be resolved by the default Modbus mapping to database addresses 46 and 48, respectively.

Also note that because discretes are mapped by the driver to registers (not database addresses), the creation of mapping objects indirectly impacts the physical database address that a given discrete accesses. Holding register mapping objects will impact the address resolution for coils, input register mapping objects will impact the address resolution for discrete inputs, and holding/input register mapping objects will impact the address resolution for both types of discretes.

Description

This 32-character (max) field is strictly for user reference: it is not used at any time by the driver.

Start Register

Defines the starting register number (1…65535) for a range of registers to be mapped.

Number of Registers

Defines the number of registers (1…2048) to map.

Database Address

Defines the database address where the mapping begins. The configuration studio will not allow entry of a starting database address that will cause the mapping object to run past the end of the database. The highest valid database address will therefore depend on the data type, as well as the number of registers to map.

Multiplier

The amount that associated network values are scaled by prior to being stored into the database or after being retrieved from the database. Upon retrieval from the database, raw data is multiplied by the multiplier to produce a network value. Similarly, network values are divided by the multiplier before being stored into the database.

Data Type

Available only when the “32-Bit Registers” checkbox is checked. Defaults to standard Modbus “16-Bit Unsigned” when the “32-Bit Registers” checkbox is unchecked. Specifies how the value will be stored in the database for each register (or register pair) in this mapping object. This defines how many bytes will be allocated, whether the value should be treated as signed or unsigned, and whether the value should be interpreted as an integer or a floating point number. Select the desired data type from this dropdown menu.

1.4.2.1 32-Bit Options Settings

32-Bit Registers

Check this checkbox if the target registers are associated with the Enron/Daniel extension to the Modbus specification, or are represented by 32-bit values.

Floating Point

Available only when the "32-Bit Registers" checkbox is checked. Check this checkbox if the register values are to be encoded in IEEE 754 floating point format.

Big Endian

Available only when the "32-Bit Registers" checkbox is checked. Check this checkbox if the register values are to be encoded in big-endian 16-bit word order, i.e. the most significant 16-bit word is before the least significant 16-bit word.

Word-Size Register

Available only when the "32-Bit Registers" checkbox is checked. Check this checkbox if target registers are only 16-bits wide, but two 16-bit registers are to comprise one 32-bit value. If unchecked, target registers are assumed to be 32-bits wide.

Note that when this checkbox is checked, the "Number of Registers" field indicates the number of 16-bit register pairs to be mapped. Each register pair will use two register addresses and the selected Data Type will be applicable for the register pair, not each individual register. For example, if Start Register is set to 100, Number of Registers is set to 2, and Data Type is set to 32-bit Unsigned, then register numbers 100…103 will be mapped, with registers 100 and 101 representing the first 32-bit unsigned value and registers 102 and 103 representing the next 32-bit unsigned value in the internal database.

Word Count

Available only when the "32-Bit Registers" checkbox is checked. Check this checkbox to interpret the Modbus "quantity of registers" field as the number of 16-bit words to be transferred. If unchecked, the Modbus "quantity of registers" field is interpreted as the number of 32-bit registers.

ICC

INDUSTRIAL CONTROL COMMUNICATIONS, INC.

1600 Aspen Commons, Suite 210
Middleton, WI USA 53562-4720
Tel: [608] 831-1255 Fax: [608] 831-2045

http://www.iccdesigns.com

Printed in U.S.A