**ICC**

**INDUSTRIAL CONTROL COMMUNICATIONS, INC.**

# Modbus RTU Slave Driver Manual

# TABLE OF CONTENTS

# 1 Modbus RTU Slave

## 1.1 Overview

This driver supports the Modbus RTU slave protocol. Some notes of interest are:

- Supported Modbus slave functions are indicated in Table 1.

**Table 1: Supported Modbus RTU Slave Functions**

| Function Code | Function |
|:---:|:---|
| 01 | Read Coils |
| 02 | Read Discrete Inputs |
| 03 | Read Holding Registers |
| 04 | Read Input Registers |
| 05 | Write Single Coil |
| 06 | Write Single Register |
| 08 | Diagnostics (Subfunction 0 only) |
| 15 | Write Multiple Coils |
| 16 | Write Multiple Registers |

- 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).

- 32-bit register accesses are supported in a variety of options and formats.

- 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 Slave Settings

**Baud Rate**
Selects the baud rate of the network.

**Parity**
Selects the parity and number of stop bits.

**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.

## Response Delay

Defines the time in milliseconds that the driver waits before responding to master requests. This is a useful feature for certain master devices or infrastructure components (such as radio modems) that may require a given amount of time to place themselves into a "receiving mode" where they are capable of listening for slave responses. If no delay is required, setting this field to 0 instructs the driver to send its responses as soon as possible.

## 1.3 Node Settings

### Address

Defines the station address (1…247) for this node.

## 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). Both coils and discrete inputs access the same data. For example, reading Coil 124 returns the same data as reading Discrete Input 124. For simplicity, coils and discrete inputs will collectively be referred to from here on out as simply "discretes".

Discretes are mapped to individual bits in the device's internal database. What this means is that when a discrete is accessed, that discrete is resolved by the driver into a specific database address, and a specific bit within that database location. Each database location is 8 bits wide, resulting in 8 discretes being mapped to each database address. The pattern of discrete-to-address/bit relationships can be described as follows:

*Discrete 1...8 map to address 0, bit0...bit7 (bit0=LSB, bit7=MSB),*
*Discrete 9...16 map to address 1, bit0...bit7,*
*:*

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

$$address = \left\lfloor \frac{(discrete - 1)}{8} \right\rfloor \qquad \textbf{Equation 2}$$

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

Additionally, for any given discrete, the targeted bit in the database location at which that discrete resides can be determined by Equation 3:

$$bit = (discrete - 1)\% \, 8 \qquad \textbf{Equation 3}$$

Where "discrete" $\in[1…65535]$, "bit" $\in[0…7]$, 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 the database, the targeted discrete corresponding to that bit can be calculated by Equation 4:

$$discrete = 8 \times address + bit + 1 \qquad \textbf{Equation 4}$$

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 at database address 4, as ⌊(34-1)/8⌋ = ⌊33/8⌋ = ⌊4 r1⌋ = 4. Then, using Equation 3, we can determine that the bit within database address 4 that coil 34 targets is 1, as (34-1)%8 = 33%8 mod(4 r1) = 1. Therefore, reading coil 34 will return the value of database address 4, bit 1. This example would unfold in the same manner should we choose instead to access discrete input 34.

## 1.4.2 Register Mapping Object Settings

Register mapping objects can be created to map holding and/or input registers to a specific address in the database. There are two different types of register mapping objects:

- <u>Holding register mapping objects</u>, which map holding (4X) registers.

- <u>Input register mapping objects</u>, which map input (3X) registers.

Because 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", and not input register 25 since we did not add an "input register mapping object". 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.

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

## <u>Start Register</u>
Defines the starting register number for a range of registers to be mapped.

## <u>Number of Registers</u>
Defines the number of registers to map.

## <u>Database Address</u>
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.

### 1.4.3 Discrete Mapping Object Settings

Discrete mapping objects can be created to map coils and/or discrete inputs to a specific address/bit in the database. There are two different types of discrete mapping objects:

- Coil mapping objects, which map coils (0X).

- Discrete input mapping objects, which map discrete inputs (1X).

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

#### 1.4.3.1 Coil Mapping Object Settings

**Description**

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

**Start Coil**

Defines the starting coil number for a range of coils to be mapped.

**Number of Coils**

Defines the number of coils 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 coils to map.

**Data Type**

Specifies how each coil will be stored in the database. This determines how many bits or bytes will be allocated for each coil.

The "1 Bit" data type is used to map each coil to a single bit in the database. Consecutive coils will be "bit packed" such that up to 8 coils can be mapped to a single database address.

The "8-Bit Unsigned", "16-Bit Unsigned", and "32-Bit Unsigned" data types are used to map each coil to multiple bits, specified by the bitmask, at a database location. Consecutive coils will map to consecutive 8, 16, or 32-bit integers in the database, with the same bitmask applied to each database value.

## Starting Bit

*Available only when the "1 Bit" data type is selected.* This specifies the starting bit in the byte designated by the database address where the coils' values are mapped. Each subsequent value will be packed into sequential bits starting with the selected starting bit.

## Bitmask

*Available only when the "8-Bit Unsigned", "16-Bit Unsigned", or "32-Bit Unsigned" data type is selected.* Specifies which bit(s) in the location designated by the database address and data type that the coils' values will map to. Each subsequent value will be mapped into sequential locations determined by the selected data type.

Note that, by default, this value is represented in hexadecimal using the "0x" prefix.

### 1.4.3.2   Discrete Input Mapping Object Settings

## Description

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

## Start Input

Defines the starting discrete input number for a range of discrete inputs to be mapped.

## Number of Inputs

Defines the number of discrete inputs 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 discrete inputs to map.

## Data Type

Specifies how each discrete input will be stored in the database. This determines how many bits or bytes will be allocated for each discrete input.

The "1 Bit" data type is used to map each discrete input to a single bit in the database. Consecutive discrete inputs will be "bit packed" such that up to 8 discrete inputs can be mapped to a single database address.

The "8-Bit Unsigned", "16-Bit Unsigned", and "32-Bit Unsigned" data types are used to map each discrete input to multiple bits, specified by the bitmask, at a database location. Consecutive discrete inputs will map to consecutive 8, 16, or 32-bit integers in the database, with the same bitmask applied to each database value.

## Starting Bit

*Available only when the "1 Bit" data type is selected.* This specifies the starting bit in the byte designated by the database address where the discrete inputs' values are mapped. Each subsequent value will be packed into sequential bits starting with the selected starting bit.

**Bitmask**

*Available only when the "8-Bit Unsigned", "16-Bit Unsigned", or "32-Bit Unsigned" data type is selected.* Specifies which bit(s) in the location designated by the database address and data type that the discrete inputs' values will map to. Each subsequent value will be mapped into sequential locations determined by the selected data type.

Note that, by default, this value is represented in hexadecimal using the "0x" prefix.

**ICC**

**INDUSTRIAL CONTROL COMMUNICATIONS, INC.**

230 Horizon Drive, Suite 100
Verona, WI USA 53593
Tel: [608] 831-1255   Fax: [608] 831-2045

http://www.iccdesigns.com                                                 Printed in U.S.A