INDUSTRIAL CONTROL COMMUNICATIONS, INC.

Generic Serial Slave Driver Manual

April 7, 2020

TABLE OF CONTENTS

1 Generic Serial Slave

1.1 Overview

The Generic Serial Slave driver can be used to communicate as a slave with any serial device using configurable ASCII and/or binary data packets. This includes devices such as barcode scanners, weigh scales, ASCII serial devices, and devices using custom or proprietary serial protocols.

Some notes of interest are:

- Supports communication with almost any serial device.
- Supports both master-slave transactions and producer-consumer transactions.
- Transactions are defined at a packet level by adding configurable packet data objects.
- Versatile packet matching options allow handling a variety of different packets with a single transaction definition.
- Requests are matched to transactions based on the order that the transactions are defined. This allows defining a “default” or “exception” transaction last to be initiated for undefined requests.
- Transactions support using database logic to manipulate received data before placing the result into the response.
- Supports variable-sized data fields and packets containing a variable number of data elements.
- Supports binary, ASCII hexadecimal, ASCII decimal, and ACII text data encodings.
- Supports unsigned integer, signed integer, and IEEE-754 floating point number formats.
- Full support for 8-bit, 16-bit, and 32-bit checksum and CRC fields.

1.2 Slave Settings

Data Bits

Select between 7 or 8 data bits per character.

Baud Rate

Selects the baud rate of the network.

Parity

Selects the parity and number of stop bits.

Packet Gap Interval

Defines the number of character times of silence on the network that indicates the end of a packet.

Timeout

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 Transactions

Communications is established by defining configurable transactions which will be matched to a received packet based on the order they are defined. Each transaction consists of a packet expected to be received from a device on the network and, optionally, a response packet to transmit back to that device.

1.3.1 Slave Transaction

A slave transaction is a transaction in which a master device sends a request to the driver and expects a response. When a slave transaction is added to the configuration, an expected request and response is automatically added below it. The expected request defines a packet that is expected to be received from the master. The response defines the packet to send to the master after receiving the request.

1.3.2 Consumer Transaction

A consumer transaction is a transaction in which a producer device sends a data packet to the driver and does not expect a response. When a consumer transaction is added to the configuration, a consumed data packet is automatically added below it. The consumed data packet defines a packet that is expected to be received from producers on the network.

1.3.3 Transaction Settings

Description

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

Uses Database Logic

Check this option if the transaction requires database logic to run between receiving the expected request and building the response.

1.3.4 Received Event (PicoPort/Mirius Only)

Each transaction can optionally include a received event. This adds the ability to detect when a packet matching the associated transaction has been received. The event is assigned to a byte in the database. When a packet matching the transaction has been received, the event database location is set to a value of 1.

Received Event Database Address

Specifies the database address to use for detecting that a packet matching the associated transaction has been received.

1.4 Packet Data Objects

Packet data objects are used to build a packet. Every character or field that is present in a packet is defined by one or more packet data objects. The order of the fields in the packet dictates the order that the corresponding packet data objects must be added. There are various types of packet data objects available in order to facilitate the definition of a packet.

1.4.1 Constant Data

Adds constant data characters to the packet.

Name

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

Constant Data Characters

Defines the constant data that is encoded in the packet. Enter up to 16 hexadecimal bytes or ASCII characters.

1.4.2 Database Data

Adds data that is mapped to the device's database. For transmitted packets, the data is read from the database and put into the packet. For received packets, the data from the packet is written to the database.

Name

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

Element Encoding

Selects the encoding used for a data element in the packet.

Number Format

Selects the format for interpreting the value of a number.

Fixed Element Size

Check this option if the size of a data element is fixed. If this option is unchecked, the driver will automatically determine the size of a variable-sized data element.

Element Size

Enter the number of characters reserved for each data element in the packet.

Byte Order

Selects the byte ordering used for multi-byte data elements in the packet.

Number of Elements

Defines the number of data elements in the packet to map to the device's database.

Database Address

Defines the starting address in the database where the packet's data elements are mapped.

Database Data Type

Specifies how each data element in the packet will be stored in the database. 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.

Multiplier

The amount that data values are scaled by prior to being stored into the database or after being retrieved from the database. Prior to storage into the database, data values are divided by the multiplier to produce database values. Upon retrieval from the database, database values are multiplied by the multiplier to produce data values.

1.4.3 Variable Database Data

Adds data that is mapped to the device's database which consists of a variable number of elements. The data from a received packet is written to the database. If the actual number of elements in the packet is less than the defined maximum, the remaining database values will be set to 0.

Name

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

Element Encoding

Selects the encoding used for a data element in the packet.

Number Format

Selects the format for interpreting the value of a number.

Fixed Element Size

Check this option if the size of a data element is fixed. If this option is unchecked, the driver will automatically determine the size of a variable-sized data element.

Element Size

Enter the number of characters reserved for each data element in the packet.

Byte Order

Selects the byte ordering used for multi-byte data elements in the packet.

Max Number of Elements

Defines the maximum number of data elements in the packet to map to the device's database. If the actual number of elements in the packet is less than this, the remaining database values will be 0.

Database Address

Defines the starting address in the database where the packet's data elements are mapped.

Database Data Type

Specifies how each data element in the packet will be stored in the database. 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.

Multiplier

The amount that data values are scaled by prior to being stored into the database or after being retrieved from the database. Prior to storage into the database, data values are divided by the multiplier to produce database values. Upon retrieval from the database, database values are multiplied by the multiplier to produce data values.

1.4.4 Ignored Characters

Adds "Don't Care" characters whose values are ignored.

Name

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

Number of Characters

Defines the number of "Don't Care" characters in the packet that will be ignored.

1.4.5 Variable Ignored Characters

Adds a variable number of "Don't Care" characters whose values are ignored.

Name

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

Max Number of Characters

Defines the maximum number of "Don't Care" characters in the packet that will be ignored.

1.4.6 Ranged Byte

Adds a single byte that must be within a defined range.

Name

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

Min Value

Defines the minimum value the byte can have to be considered a match.

Max Value

Defines the maximum value the byte can have to be considered a match.

1.4.7 Bitmasked Byte

Adds a single byte that must match a defined value after a bitmask is applied.

Name

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

Bitmask

Specifies the bit(s) in the byte that are relevant for detecting a match.

Value

Defines the value the byte must have to be considered a match after applying the bitmask.

1.4.8 Database Matched Byte

Adds a single byte that must match the current value stored at a location in the device's database.

Name

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

Database Address

Defines the database address containing the value the byte must have to be considered a match.

1.4.9 List Matched Character

Adds a single character that must match a value from a list of values.

Name

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

Value List

Defines a list of possible values the character can have to be considered a match. Enter up to 15 hexadecimal bytes or ASCII characters.

1.4.10 Checksum

Adds a checksum field that is calculated from a defined start offset in the packet up to, but not including, the location of the checksum field itself.

Name

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

Checksum Data Type

Selects the data type and defines the width of the checksum.

Encoding

Selects the encoding used for the checksum in the packet.

Checksum Size

The number of characters reserved for the checksum in the packet.

Byte Order

Selects the byte ordering for how the checksum is encoded in the packet.

1.4.10.1 Checksum Calculation Parameters

Start Offset

Defines the starting offset in the packet where the checksum calculation will begin.

Use 2 Chars/Byte

Check this option to first convert 2 ASCII characters from the packet into a byte value which will be used in the checksum calculation. If this option is unchecked, the raw bytes from the packet will be used directly in the checksum calculation.

Final Operation

Selects an optional final operation to apply to the checksum value at the end of the calculation.

1.4.11 CRC

Adds a cyclic redundancy check field that is calculated from a defined start offset in the packet up to, but not including, the location of the CRC field itself.

Name

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

CRC Data Type

Selects the data type and defines the width of the CRC.

Encoding

Selects the encoding used for the CRC in the packet.

CRC Size

The number of characters reserved for the CRC in the packet.

Byte Order

Selects the byte ordering for how the CRC is encoded in the packet.

1.4.11.1 CRC Calculation Parameters

Start Offset

Defines the starting offset in the packet where the CRC calculation will begin.

Use 2 Chars/Byte

Check this option to first convert 2 ASCII characters from the packet into a byte value which will be used in the CRC calculation. If this option is unchecked, the raw bytes from the packet will be used directly in the CRC calculation.

Bit Order (Shift Direction)

Selects which bit in each byte is considered first (and by correlation, the direction bits are shifted) when calculating the CRC.

Polynomial

Defines the generator polynomial used in the CRC calculation. Enter the hexadecimal representation of the polynomial's bit sequence, most-significant bit first, omitting the highest-order bit.

Initial Value

Enter the value that the CRC will be initialized to at the beginning of the CRC calculation.

Final XOR Value

Enter the value that the CRC will be XORed with at the end of the CRC calculation.

2 Examples

2.1 Modbus ASCII Example

This example shows how to build a Modbus ASCII expected request and response packet for a slave transaction using packet data objects. This example demonstrates a remote device reading data from the device's internal database. The specific Modbus request used in this example is function code 3, Read Holding Registers.

2.1.1 Building the Expected Request

The first step is to identify what packet data objects are required to build the expected request packet. Figure 1 below shows the Modbus ASCII frame structure for the function code 3 request, grouped into packet data objects. As seen below, we will need three packet data objects to define the packet: Constant Data, Checksum, and Constant Data.

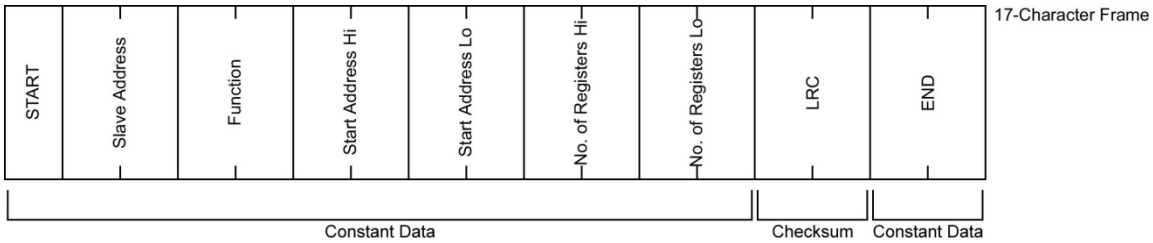

Figure 1: Modbus ASCII Read Holding Register Request

Now that we know the frame structure for the request, let's try a specific example. Figure 2 below shows the request to read register addresses 0 through 9 from a device at address 1.

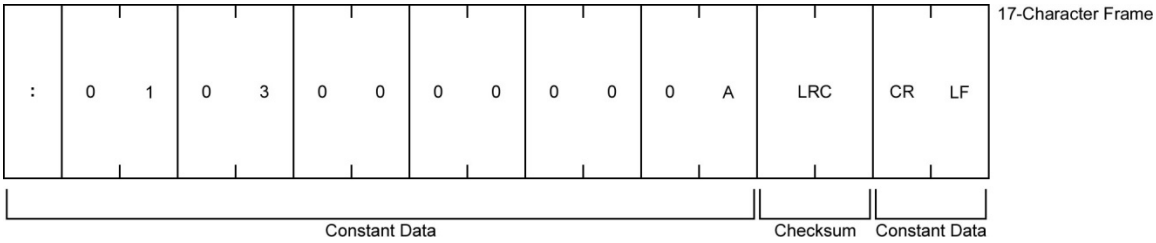

Figure 2: Read Holding Registers ASCII Request

Because the Constant Data packet data objects are defined in hexadecimal, the last step we need to do is convert the ASCII characters in our example request to hexadecimal characters. Figure 3 below shows the request converted to hexadecimal.

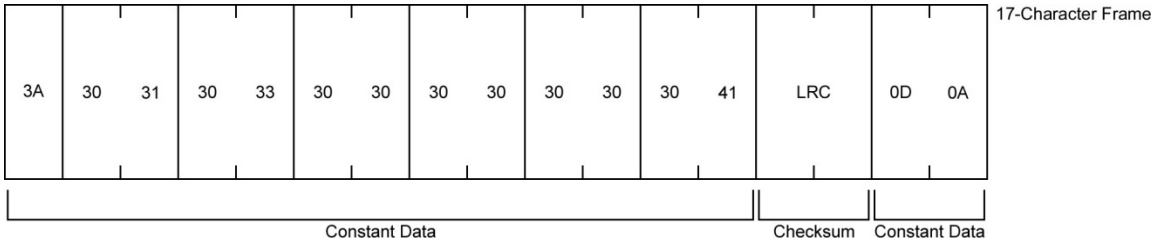

Figure 3: Read Holding Registers Hex Request

2.1.2 Packet Data Object Configuration for the Expected Request

2.1.2.1 Constant Data

The settings for the first Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: “3A 30 31 30 33 30 30 30 30 30 30 30 41”.

2.1.2.2 Checksum

The LRC is calculated by adding together successive 8-bit bytes of the message, discarding any carries, and then two’s complementing the result. It is performed on the ASCII message field contents excluding the ‘colon’ character that begins the message, and excluding the CRLF pair at the end of the message.

The settings for the Checksum object are as follows:

- Select *8-Bit Unsigned* for the **Checksum Data Type**.
- Select *ASCII* for the **Encoding**.
- Enter a value of “1” in the **Start Offset** field.
- Check the **Use 2 Chars/Byte** option.
- Select *2’s Complement* for the **Final Operation**.

2.1.2.3 Constant Data

The settings for the last Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: “0D 0A”.

2.1.3 Building the Response

Figure 4 below shows the Modbus ASCII frame structure for the function code 3 response, grouped into packet data objects. As seen below, we will need four packet data objects to define the packet: Constant Data, Database Data, Checksum, and Constant Data.

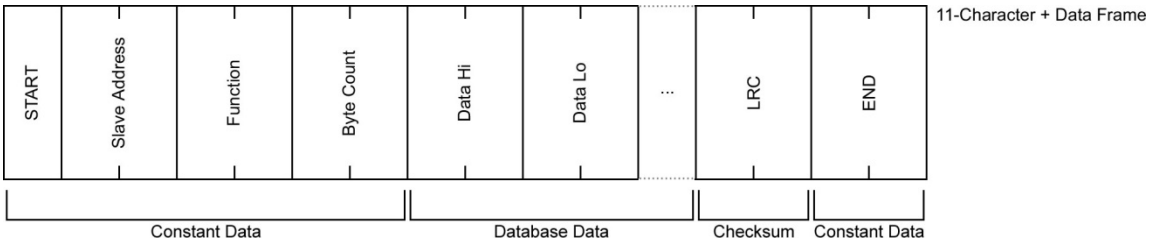

Figure 4: Modbus ASCII Read Holding Register Response

Using the example request from above, Figure 2 below shows the expected response to the request to read register addresses 0 through 9 from a device at address 1.

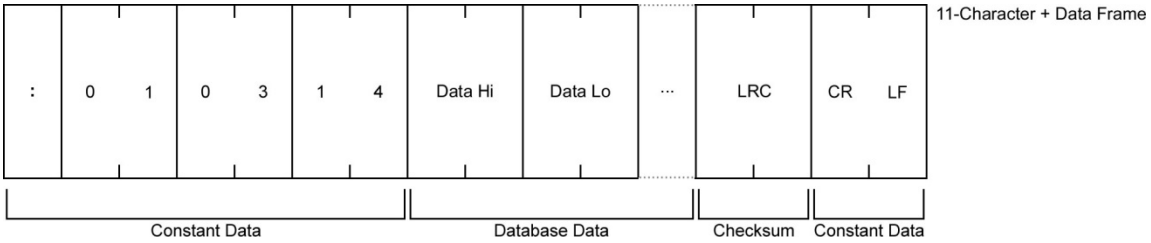

Figure 5: Read Holding Registers ASCII Response

Again, because the Constant Data packet data objects are defined in hexadecimal, the last step we need to do is convert the ASCII characters in our example response to hexadecimal characters. Figure 3 below shows the request converted to hexadecimal.

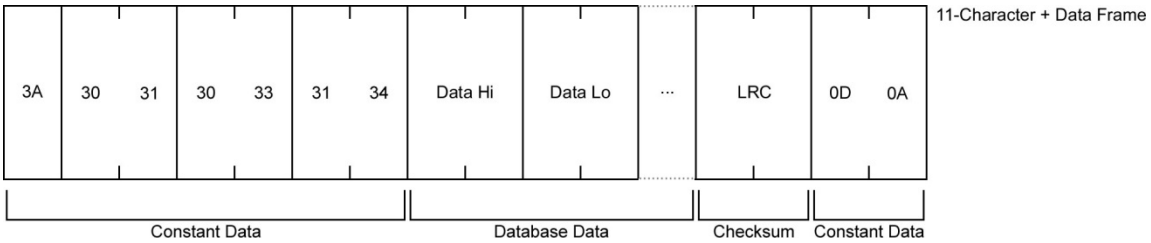

Figure 6: Read Holding Registers Hex Response

2.1.4 Packet Data Object Configuration for the Response

2.1.4.1 Constant Data

The settings for the first Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: “3A 30 31 30 33 31 34”.

2.1.4.2 Database Data

The format for each data character in a Modbus ASCII frame is hexadecimal, ASCII characters. One hexadecimal character, i.e. one nibble or half of a byte, is contained in each ASCII character of the frame. Therefore, two ASCII characters equate to one byte of the Modbus data.

The settings for the Database Data object are as follows:

- Select *ASCII Encoded Hexadecimal Number* for the **Element Encoding**.
- Select *Unsigned Integer* for the **Number Format**.
- Check the **Fixed Element Size** option.
- Enter a value of “4” in the **Element Size** field.
- Select *High Byte First* for the **Byte Order**.
- Enter a value of “10” in the **Number of Elements** field.

2.1.4.3 Checksum

The LRC is calculated by adding together successive 8-bit bytes of the message, discarding any carries, and then two’s complementing the result. It is performed on the ASCII message field contents excluding the ‘colon’ character that begins the message, and excluding the CRLF pair at the end of the message.

The settings for the Checksum object are as follows:

- Select *8-Bit Unsigned* for the **Checksum Data Type**.
- Select *ASCII* for the **Encoding**.
- Enter a value of “1” in the **Start Offset** field.
- Check the **Use 2 Chars/Byte** option.
- Select *2’s Complement* for the **Final Operation**.

2.1.4.4 Constant Data

The settings for the last Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: “0D 0A”.

2.2 Modbus RTU Example

This example shows how to build a Modbus RTU expected request and response packet for a slave transaction using packet data objects. This example demonstrates a remote device writing data to the device's internal database. The specific Modbus request used in this example is function code 6, Preset Single Register.

2.2.1 Building the Expected Request

First, we need to identify what packet data objects are required to build the expected request packet. Figure 7 below shows the Modbus RTU frame structure for the function code 6 request, grouped into packet data objects. As seen below, we will need three packet data objects to define the packet: Constant Data, Database Data, and CRC.

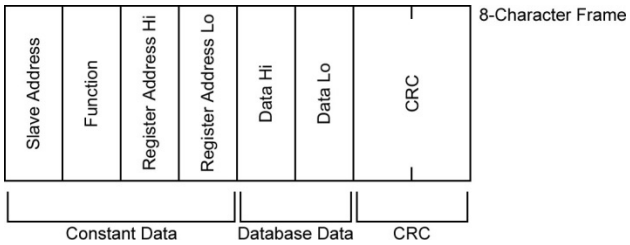

Figure 7: Modbus RTU Preset Single Register Request

Given the frame structure for the request, let's define a specific example. Figure 8 below shows the request to preset register address 4 on a device at address 1. Note that because Modbus RTU uses binary encoding, the data is shown in hexadecimal notation.

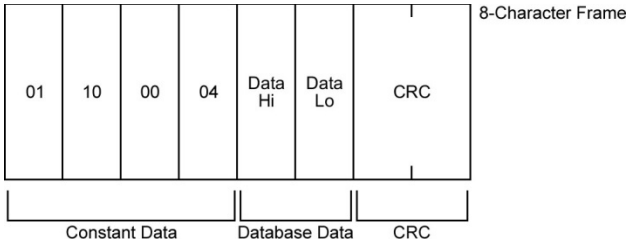

Figure 8: Preset Single Register Hex Request

2.2.2 Packet Data Object Configuration for the Expected Request

2.2.2.1 Constant Data

The settings for the Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: “01 10 00 04”.

2.2.2.2 Database Data

The format for each data character in a Modbus RTU frame is 8-bit binary, hexadecimal characters.

The settings for the Database Data object are as follows:

- Select *Binary Data* for the **Element Encoding**.
- Select *Unsigned Integer* for the **Number Format**.
- Check the **Fixed Element Size** option.
- Enter a value of “2” in the **Element Size** field.
- Select *High Byte First* for the **Byte Order**.
- Enter a value of “1” in the **Number of Elements** field.

2.2.2.3 CRC

The CRC field is two bytes, containing a 16-bit binary value. The CRC is started by first preloading a 16-bit register to all 1’s. During generation of the CRC, each 8-bit character is exclusive ORed with the register contents, starting with the first character of the message. Then the result is shifted in the direction of the least significant bit. The final contents of the register, after all the bytes of the message have been applied, is the CRC value. When the 16-bit CRC is transmitted in the message, the low-order byte will be transmitted first, followed by the high-order byte.

The settings for the CRC object are as follows:

- Select *16-Bit Unsigned* for the **CRC Data Type**.
- Select *Binary* for the **Encoding**.
- Enter a value of “0” in the **Start Offset** field.
- Select *Low-Order Bit First* for the **Bit Order**.
- Enter a value of “0xA001” in the **Polynomial** field.
- Enter a value of “0xFFFF” in the **Initial Value** field.
- Enter a value of “0x0” in the **Final XOR Value** field.

2.2.3 Building the Response

Figure 9 below shows the Modbus RTU frame structure for the function code 6 response, grouped into packet data objects. Note that the response includes an echo of the data sent in the request. As seen below, we will need three packet data objects to define the packet: Constant Data, Database Data, and CRC.

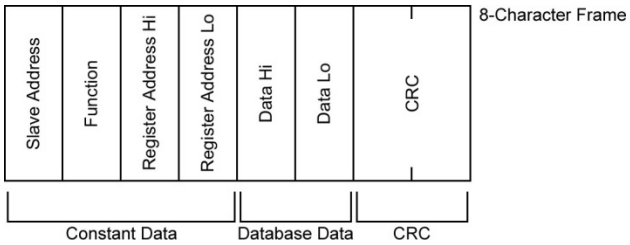

Figure 9: Modbus RTU Preset Single Register Response

Using the example request from above, Figure 10 below shows the expected response to the request to preset register addresses 4 on a device at address 1.

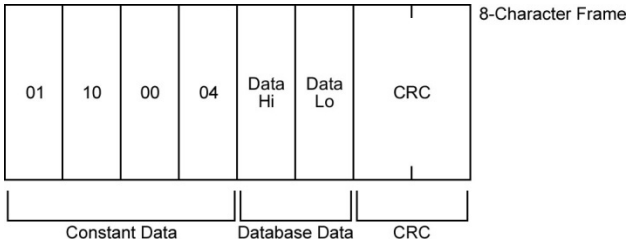

Figure 10: Preset Single Register Hex Response

2.2.4 Packet Data Object Configuration for the Response

2.2.4.1 Constant Data

The settings for the Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: “01 10 00 04”.

2.2.4.2 Database Data

The format for each data character in a Modbus RTU frame is 8-bit binary, hexadecimal characters.

The settings for the Database Data object are as follows:

- Select *Binary Data* for the **Element Encoding**.
- Select *Unsigned Integer* for the **Number Format**.
- Check the **Fixed Element Size** option.
- Enter a value of “2” in the **Element Size** field.
- Select *High Byte First* for the **Byte Order**.
- Enter a value of “1” in the **Number of Elements** field.

Note that this Database Data object must be mapped to the same database location as the Database Data object in the expected request so the written data can be correctly echoed back to the master.

2.2.4.3 CRC

The CRC field is two bytes, containing a 16-bit binary value. The CRC is started by first preloading a 16-bit register to all 1’s. During generation of the CRC, each 8-bit character is exclusive ORed with the register contents, starting with the first character of the message. Then the result is shifted in the direction of the least significant bit. The final contents of the register, after all the bytes of the message have been applied, is the CRC value. When the 16-bit CRC is transmitted in the message, the low-order byte will be transmitted first, followed by the high-order byte.

The settings for the CRC object are as follows:

- Select *16-Bit Unsigned* for the **CRC Data Type**.
- Select *Binary* for the **Encoding**.
- Enter a value of “0” in the **Start Offset** field.
- Select *Low-Order Bit First* for the **Bit Order**.
- Enter a value of “0xA001” in the **Polynomial** field.
- Enter a value of “0xFFFF” in the **Initial Value** field.
- Enter a value of “0x0” in the **Final XOR Value** field.

2.3 ASCII Barcode Scanner Example

This example shows how to consume ASCII data sent by a serial barcode scanner. This example demonstrates receiving variable-length data from a remote device and storing that data in the device's internal database.

2.3.1 Building the Expected Consumed Data

We first need to identify what packet data objects are required to build the expected consumed data packet. The data sent by the barcode scanner consists of nothing more than the ASCII data followed by a carriage-return character and a line-feed character. Figure 11 below shows the ASCII data frame structure, grouped into packet data objects. As seen below, we will need two packet data objects to define the packet: Variable Database Data and Constant Data.

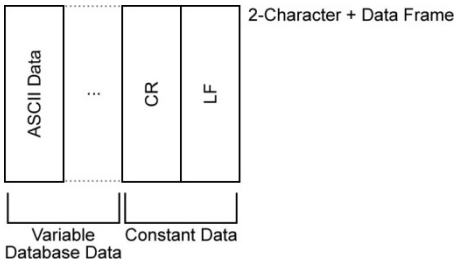

Figure 11: Variable-length ASCII Data

Because the Constant Data packet data objects are defined in hexadecimal, we need to convert the ASCII characters in our example frame to hexadecimal characters. Figure 12 below shows the frame converted to hexadecimal.

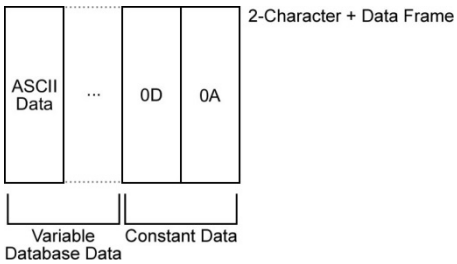

Figure 12: Variable-length ASCII Data in Hex

2.3.2 Packet Data Object Configuration for the Expected Consumed Data

2.3.2.1 Variable Database Data

For example purposes, we'll assume that the maximum length of the ASCII data is 80 characters and that the data that we wish to store in the database is ASCII text.

The settings for the Database Data object are as follows:

- Select *ASCII Text Character* for the **Element Encoding**.
- Enter a value of "80" in the **Max Number of Elements** field.

Note that if the actual length of ASCII data received is less than the defined maximum, the remaining database values will be set to 0.

2.3.2.2 Constant Data

The settings for the Constant Data object are as follows:

- Enter the following into the **Constant Data Bytes** field: "0D 0A".

ICC

INDUSTRIAL CONTROL COMMUNICATIONS, INC.

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

http://www.iccdesigns.com

Printed in U.S.A