



USING A TRi SUPER PLC AS REMOTE I/O FOR ANOTHER TRi SUPER PLC (MODBUS command)

Revision 1

**APPLICATION
NOTE**



Copyright Notice and Disclaimer

All rights reserved. No parts of this application note may be reproduced in any form without the express written permission of TRi.

Triangle Research International, Inc. (TRi) makes no representations or warranties with respect to the contents hereof. In addition, information contained herein is subject to change without notice. Every precaution has been taken in the preparation of this document. Nevertheless, TRi assumes no responsibility for errors or omissions or any damages resulting from the use of the information contained in this publication.

MS-DOS and Windows are trademarks of Microsoft Inc.
MODBUS is a trademark of Modbus.org
All other trademarks belong to their respective owners.

Revision Sheet

Release No.	Date	Revision Description
Rev. 1	11/20/2012	Updated to reflect current TRi PLC models.

TABLE OF CONTENTS

	<u>Page #</u>
1 OVERVIEW	ERROR! BOOKMARK NOT DEFINED.
2 REMOTE I/O	ERROR! BOOKMARK NOT DEFINED.
2.1 Introduction	Error! Bookmark not defined.
2.2 The Physical Network	2-1
2.3 Mapping I/O	2-1
2.3.1 Introduction	2-1
2.3.2 Mapping Inputs from the 'SUPER' PLC	2-2
2.3.3 Mapping Memory from the Master PLC	2-2
2.3.4 Network communication	2-3
3 SERIAL COMMUNICATIONS	3-1
3.1 Introduction	3-1
3.2 RS485	3-1
3.2.1 Background	3-1
3.2.2 Connections	3-1
3.3 Auto485	3-2
3.4 MODBUS	3-3
3.4.1 Introduction	3-3
3.4.2 MODBUS RTU	3-4
3.4.3 MODBUS ASCII	3-4
3.4.4 MODBUS ASCII VS. RTU	3-4
4 THE SAMPLE PROGRAM	4-1
4.1 How It Works	4-1
4.1.1 Basics	4-1
4.1.2 Custom function #2: "Update_IN"	4-2
4.1.3 Custom function #4: "Empty"	4-3
4.1.4 Relay Coil: "SL1_O4"	4-3
4.1.5 Up Counter: "Count_SL2_O1"	4-3
4.1.6 Custom function #3: "Update_OUT"	4-3
4.2 How To Use It	4-4
5 LINKS	5-1
5.1 SERIAL COMMUNICATIONS	5-1
5.1.1 Auto485	5-1
5.1.2 MODBUS	5-1

1 OVERVIEW

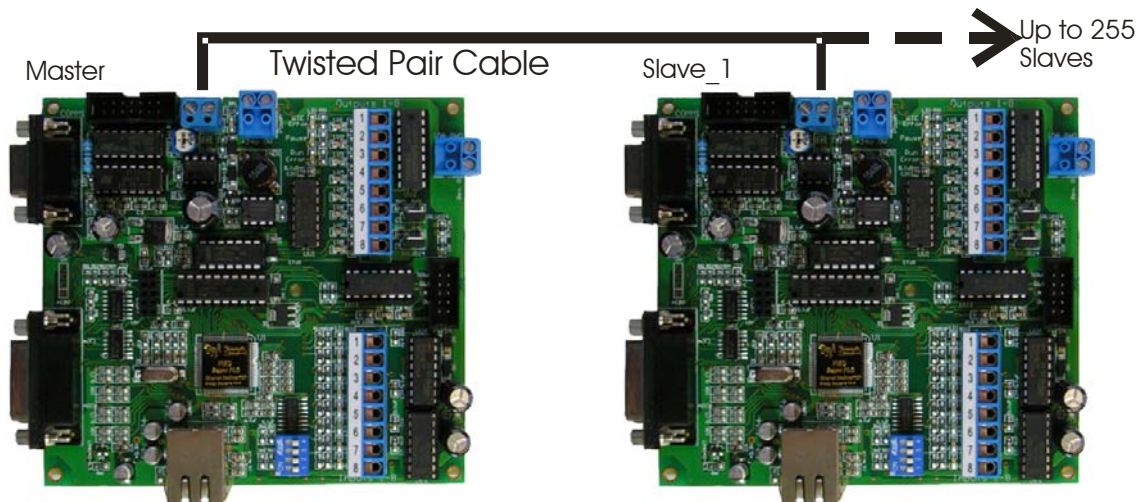
This application note explains how to go about using the **TRi 'Super' PLC (Nano-10, FMD88-10, FMD1616-10, F1616-BA, F2424)** as Remote I/O for other 'Super' PLCs using the MODBUS protocol to communicate. This could be useful if the system needs to have its I/O separate from the user interface, or if the systems I/O is spread out with one user access point or one control brain.

Using the MODBUS communication protocol will allow the PLCs to talk to other devices with MODBUS capability such as touch screens, printers, barcode scanners, and many other devices. Also, it will allow the system (master PLC, slave PLCs, and MODBUS devices) to be controlled and monitored with SCADA software and other software similar to SCADA.

This application note covers mapping I/O from one PLC to another, as well as serial communications using an RS485 connected network and the MODBUS protocol. A sample program that demonstrates these topics will be included. The sample program will be generic, with the intention that it can be modified for different applications.

NOTE:

This application note will specifically describe a remote I/O system where the master PLC is a FMD1616-10 and the slave PLCs are FMD888-10's. However, please note that the master PLC could be any 'Super' PLC depending on personal requirements and the slave PLCs could be either FMD888-10's or FMD1616-10's without changing the sample program. With some minor changes, 'Super' PLCs with more than 16 I/O could be used as slaves. Also, this application note will discuss the network configuration as using one FMD888-10 PLC as a slave; however, the sample program is programmed to have two slave FMD SERIES PLCs and is expandable to up to 4 FMD SERIES slaves (5 PLCs including the master).



2 REMOTE I/O

2.1 Introduction

Using remote I/O is a good way to collect data from multiple areas and send the data back to a central station through an RS485 network for interpretation and manipulation. Using remote I/O, data can be sent long distances over an RS485 network (1200m) without risking data loss.

The main parts to a remote I/O setup are:

1. The physical network
2. I/O mapping
3. Network communication

2.2 The Physical Network

The network involved here contains a FMD888-10 as the remote I/O (slave) and a FMD1616-10 as the central brain (master). This is a simple 2 device network to show how remote I/O works; however, this can be expanded to include multiple FMD SERIES PLCs as remote I/O (up to 5 PLC's can be connected on one network). The [Serial Communications](#) section shows how to wire the RS485 network.

2.3 Mapping I/O

2.3.1 Introduction

Mapping I/O is useful for building a gateway that will allow data to move through a network. I/O mapping is always programmed into the master PLC; the slave(s) have no idea where the data they collect is going. The slave(s) only respond to commands sent from the master, which is either to send input status or to update output status.

There are many possible ways to map I/O from one device to another. The way I am about to describe is the method that I used in the sample program included in this application note. It is just one possibility and may not be the best way in every situation.

There are 256 internal Relay contacts that are grouped into 16 chunks named RELAY[1] - RELAY[16]. These chunks are system variables and are equivalent to arrays. Each RELAY variable is a 16 bit word, each bit represents a contact. If one of these contacts is activated, either through ladder logic or code, then the bit in the RELAY variable that the contact represents will become a 1.

For example, if Relay contact #1 was activated from it's normally deactivated state and all other Relay contacts were deactivated, then the value of RELAY[1] would change from 0 to 0000000000000001 binary, 1 decimal, and 0001 hex. Relay contact #1 corresponds to the first bit of the RELAY[1] array (system variable). The reason for that explanation was to help with the following explanation of I/O mapping.

The relay contacts that are used in the I/O mapping can be used as relay contacts in ladder logic and in code to resemble the inputs on the slave PLC that they were mapped from. This allows the inputs of the

slave PLCs to affect the program in the master PLC, which is one reason for mapping I/O. This will be discussed in more detail in sub-sections 3.1.3-3.1.5 of section 3. [The Sample Program](#).

2.3.2 Mapping Inputs from the 'SUPER' PLC

Mapping inputs is quite simple using MODBUS because entire registers can be mapped in a single command. In this case, an input register (8-16 inputs) is being mapped to a relay register (16 relay contacts) using one MODBUS command. The following line of code is used to map inputs from the slave PLCs to the master PLC.

```
RELAY[I] = READMODBUS(3, I, 0)
```

Code from sample program: "RemotelO_FMD", Custom function #2: "Update_IN"

The code to map inputs, from above, will be split into pieces and each piece will be explained individually. Then the code will be put back together and explained as a whole.

RELAY[I]

This code refers to the inputs of the slave devices. The variable I represents the ID of the current slave device. Since the id of the first slave device is 2, the minimum value of I is 2. Each 16 bit/contact RELAY[] array is either used to map inputs of a slave device to the Master PLC or to map outputs of a slave device to the Master PLC. RELAY[2] - RELAY[5] are designated for the slave devices inputs.

For example, slave device #1 has ID #2 and the value of I would be 2. So its inputs would be mapped to RELAY[2]. Therefore when I = 2, the code RELAY[I] is equivalent to the 16 inputs (8 for FMD88-10) of the first slave device.

READMODBUS(3, I, 0)

This command is sent from the master to device "I" (slave device I - 1) via com3 and it tells the slave device to send its first input register status (status of inputs 1 - 16) to the master device. The READMODBUS() command and the MODBUS protocol will be explained in more detail in sections 1.3.4 [Network communication](#) and 2.4 [MODBUS](#) respectively.

```
RELAY[I] = READMODBUS(3, I, 0)
```

This line of code will map the inputs of device I (slave device I - 1) directly to relay register I.

2.3.3 Mapping Memory from the Master PLC

Mapping memory from the master FMD1616-10 PLC to the outputs of the slave FMD888-10 PLC is very similar to mapping inputs from the slave FMD888-10 PLC to the memory of the master FMD1616-10 PLC. Again, there are many different ways to go about this but I am using the same method I used in the sample program provided with this application note.

The following 2 lines of code are used to map outputs from the master PLC to the slave PLCs.

```
DM [3995 + I] = RELAY[I + 4]
WRITEMODBUS 3, I, 16, RELAY[I + 4]
```

Code from sample program: "RemotelO_FMD", Custom function #3: "Update_OUT"

RELAY[I + 4]

This code refers to the outputs of the slave devices. The variable I represents the ID of the current slave device. Since the id of the first slave device is 2, the minimum value of I + 4 is 6. Each 16 bit/contact RELAY[] array is either used to map inputs of a slave device to the Master PLC or to map outputs of a slave device to the Master PLC. RELAY[6] - RELAY[9] are designated for the slave devices outputs.

For example, slave device #1 has ID #2 and the value of $I + 4$ would be 6. So its outputs would be mapped to RELAY[6]. Therefore when $I = 2$, the code RELAY[$I + 4$] is equivalent to the 16 outputs (8 for FMD88-10) of the first slave device.

DM[3995 + I]

This code also refers to the outputs of the slave devices. Each DM[] location is 16 bits just like each RELAY[] location. Each bit represents an output of a slave device just like each relay contact represents an input/output of a slave device. The point of this code is to save a copy of each slave devices output status so that it may be used in a comparison. This is used in the custom function "Update_OUT" as a way to save time. What I mean is that it takes time to send commands to through the serial port such as the command to update the outputs of the slave devices. If the output status of a device hasn't changed then there is no point in sending the command to update its outputs. So everytime Update_OUT is executed the current value of the output status (RELAY[$I + 4$]) for the current slave device is compared to the previous value of the output status (DM[3995 + I]) for the current slave device. If these values are different then the devices output status is updated and the value in DM[3995 + I] is set to the new output status that is in RELAY[$I + 4$].

DM[3995 + I] <> RELAY[$I + 4$]

The point of this code is to save a copy of each slave devices output status so that it may be used in a comparison. This is used in the custom function "Update_OUT" as a way to save time. What I mean is that it takes time to send commands through the serial port such as the command to update the outputs of the slave devices. If the output status of a device hasn't changed then there is no point in sending the command to update its outputs. So everytime Update_OUT is executed, the current value of the output status (RELAY[$I + 4$]) for the current slave device is compared to the previous value of the output status (DM[3995 + I]) for the current slave device. If these values are different then the devices output status is updated and the value in DM[3995 + I] is set to the new output status that is in RELAY[$I + 4$].

WRITEMODBUS 3, I, 16, RELAY[$I + 4$]

This command is sent from the master PLC to device "I" (slave PLC $I - 1$) via com3 and it tells the slave device to update its first output register status (status of outputs 1 – 16) to the values of relay register " $I + 4$ ". The WRITEMODBUS command and the MODBUS protocol will be explained in more detail in sections 1.3.4 [Network communication](#) and 2.4 [MODBUS](#) respectively.

2.3.4 Network communication

'Super' PLCs support a few different protocols, but the most common and most powerful protocol supported is the MODBUS protocol. Both MODBUS ASCII and RTU are supported; however, this application note focuses on MODBUS ASCII. For more information on MODBUS see the MODBUS section of [Serial Communications](#). There are some BASIC functions that come with the I-Trilogi software that allow the PLC to talk to other devices in MODBUS quite easily.

There are 2 functions that are used in the sample program:

1. A function to read the data from the slave – READMODBUS(ch, deviceID, address)
2. A function to write data to the slave – WRITEMODBUS ch, deviceID, address, data

The first command, READMODBUS, is used read the input status of the first 16 inputs of device #I. The "ch" part of the command is the input channel to be read. Each input channel includes 16 inputs. Channel 0 ("00") includes inputs 1 to 16 and channel 1 ("01") includes inputs 17 to 32 and so on. Since the FMD888-10 only has 8 inputs total, only one input channel can be read – channel 0. An example of the code to read inputs is listed below. Figure 1 shows a more detailed structure of the READMODBUS function.

RELAY[1] = READMODBUS(3, I, 0)	'send readmodbus() command to com3 (RS485) to 'read inputs of device I
--------------------------------	---

Code taken from sample program: "RemotelO_FMD", Custom function #3: "Update_IN"

Command	READMODBUS (<i>ch, DeviceID, address</i>)
	{* Applicable only to M+ PLC models}
Purpose	<p>Automatically query a MODBUS ASCII device and return the 16-bit register data using the MODBUS ASCII protocol. The communication baud rate is the default baud rate of that COMM unless it has been changed by the SETBAUD command.</p> <p><i>ch</i> - PLC COMM port number (1-8) <i>DeviceID</i> - device ID of the MODBUS device (1 to 255) <i>address</i> - zero-offset address of the holding register in the MODBUS device.</p>
Examples	relay [3] = READMODBUS (3, 5, 101)
Comments:	<p><i>The relay will contain the 16-bit data obtained from the MODBUS device with ID = 05 and from register offset address 101 (in MODBUS term this refer to the #40102 holding register) . Reading it into the relay[] channel allows bit level manipulation by ladder logic. It can of course also be read into any data memory. The command automatically checks the response string received from the slave device for the correct LRC and the slave address. The status of the operation can be checked in the user program by executing the STATUS(2) function, which will return a '0' if there is any error or if the slave device is not present.</i></p>
See Also	WRITEMODBUS , STATUS(2) , NETCMD\$()

Figure 1: ReadModbus Command

The second command, WRITEMODBUS, is very much like READMODBUS except that data is being written to output channel 0. The data is the value of DM[3743 + I]. An example of the code to write outputs is listed below. Figure 2 shows a more detailed structure of the WRITEMODBUS function.

WRITEMODBUS 3, I, 16, RELAY[2]	'send writemodbus()command to com3 (RS485) to 'write contents of dm[3743 + I] to outputs of M-Series device I
--------------------------------	---

Code taken from sample program: "RemotelO_FMD", Custom function #4: "Update_OUT"

Command	WRITEMODBUS <i>ch, DeviceID, address, data</i>
---------	---

	{* Applicable only to M+ PLC models}
Purpose	<p>Automatically write the 16-bit <i>data</i> to a MODBUS ASCII device using the MODBUS ASCII protocol. The communication baud rate is the default baud rate of that COMM port unless it has been changed by the SETBAUD command.</p> <p><i>ch</i> - PLC COMM port number (1-8) <i>DeviceID</i> - Device ID of the MODBUS device (1 to 255) <i>address</i> - Zero-offset address of the holding register in the MODBUS device. <i>data</i> - the 16-bit data to be written to the MODBUS device</p>
Examples	WRITEMODBUS 3, 8, 1000, 1234
Comments:	<i>The data 1234 will be written to the MODBUS device with ID=08 at the holding register offset address 1000 (in MODBUS convention this refer to holding register #41001). The command automatically checks the response string received from the slave device for the correct LRC and the slave address. The status of the operation can be checked in the user program by executing the STATUS(2) function, which will return a '0' if there is any error or if the slave device is not present.</i>
See Also	READMODBUS(), STATUS(2), NETCMD\$()

Figure 2: WriteModbus Command

3 SERIAL COMMUNICATIONS

3.1 Introduction

There are two types of physical serial communication on TRi [‘Super’](#) PLCs; one is RS232, and the other is RS485. The only exception is that the Nano-10 only has an RS485 port. RS232 is used for one-to-one communication, and RS485 is used for one-to-many communication; however, it can be used for one-to-one communication as well. RS485 will be discussed in this application note.

Communication will be discussed in terms of physical connections, programming, and protocols.

3.2 RS485

3.2.1 Background

RS485 is an electrical standard, and it is not a communication protocol. RS485 is used for one-to-many communication; however, it can be used for one-to-one communication as well. Due to its method of signal transfer, devices can send data distances of up to 1200m.

RS485 has two different wire systems: full duplex and half duplex. Full duplex uses 4 wires and half duplex uses 2 wires. All Tri PLC's use the more common half duplex system. Half duplex has a +ve wire and -ve wire. The logic state of a signal is determined by the voltage of the +ve wire with respect to the voltage of the -ve wire.

- A logic 1 is when the +ve wire is > 200mV wrt the -ve wire
- A logic 0 is when the -ve wire is > 200mV wrt the +ve wire

3.2.2 Connections

All [‘Super’](#) PLC's have a blue screw terminal for RS485 connections. The terminal has connections for the +ve wire and the -ve wire. Between devices the +ve wire goes to the +ve wire and the -ve wire goes to the -ve wire.

All [‘Super’](#) PLCs have physical connections for both RS232 and RS485 (except the Nano-10). The +ve wire for RS485 is pin 5 of the DB9 connector and the -ve wire is pin 2. An example of an RS485 network with TRi [‘Super’](#) PLCs is shown below in Figure 3 – RS485 Network. A more detailed explanation of the connection is shown below Figure 3.

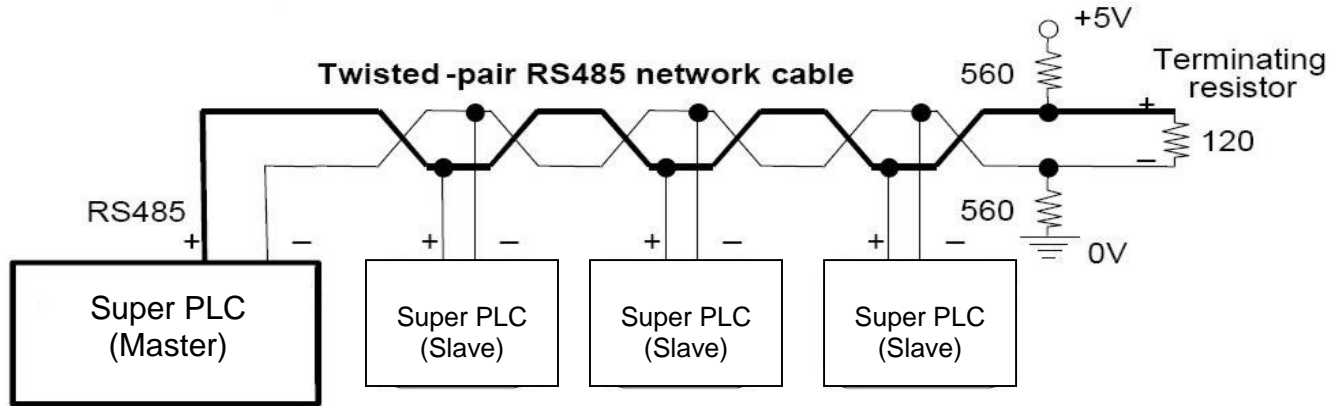


Figure 3 - RS485 Network

NOTE:

If the devices being connected are using power supplies with different commons then the commons will have to be connected to avoid signal Interference. This can be done by connecting a 3rd wire to power supply common of every device or if twisted pair wiring is used, then the shielding can be unraveled and connected to power supply common at every device.

The built-in RS-485 interface allows the TRi 'Super' controllers to be networked together using very low cost twisted-pair cables. Standard RS-485 allows up to 256 devices (including the host computer node) to be connected together. This works using a 1/8-power RS485 driver such as the 75HVD3082. The twisted-pair cable goes from node to node in a daisy chain fashion and should be terminated by a 120ohm resistor as shown above.

Note that the two wires are not interchangeable so they must be wired the same way to each controller. The maximum wire length should not be more than 1200 meters (4000 feet). RS-485 uses balanced or differential drivers and receivers, this means that the logic state of the transmitted signal depends on the differential voltage between the two wires and not on the voltage with respect to a common ground. As there will be times when no transmitters are active (which leaves the wires in "floating" state), it is a good practice to ensure that the RS-485 receivers will indicate to the CPUs that there is no data to receive. In order to do this, we should hold the twisted pair in the logic '1' state by applying a differential bias to the lines using a pair of 560W to 1KW biasing resistors connected to a +9V (at least +5V) and 0V supply as shown in Figure 3. Otherwise, random noise on the pair could be falsely interpreted as data. The two biasing resistors are necessary to ensure robust data communication in actual applications. Some RS485 converters may already have biasing built-in so the biasing resistors may not be needed. However, if the master is a TRi 'Super' PLC then you should use the biasing resistor to fix the logic states to a known state. Although in lab environment the PLCs may be able to communicate without the biasing resistors, their use is strongly recommended for industrial applications.

3.3 Auto485

The Auto485 is a device that will convert RS232 to RS485 and vice versa. For this application note, the Auto485 is used to connect a pc to the RS485 network through the pc's RS232 serial port.

For more information on the Auto485, go to the ["Links"](#) section.

3.4 MODBUS

3.4.1 Introduction

This is a protocol that many devices use to communicate in industry. It is used for all kinds of communication including Serial and Ethernet. It is a powerful protocol that is open source and free to be used by anyone as long as minimum requirements are met so that there is always one basic standard and not 1000's.

MODBUS employs a series of functions that allow devices to read and write data to and from other

Modbus Function Code	Current Terminology	Classic Terminology
Conformance Class 0		
3 (03 hex)	Read Multiple Registers	Read Holding Registers
16 (10 hex)	Write Multiple Registers	Preset Multiple Registers
Conformance Class 1		
1 (01 hex)	Read Coils	Read Coil Status
2 (02 hex)	Read Inputs Discretes	Read Input Status
4 (04 hex)	Read Input Registers	Read Input Registers
5 (05 hex)	Write Coil	Force Single Coil
6 (06 hex)	Write Single Register	Preset Single Register
7 (07 hex)	Read Exception Status	Read Exception Status
Conformance Class 2		
15 (0F hex)	Force Multiple Coils	Force Multiple Coils
22 (16 hex)	Mask Write Register	Mask Write Register
23 (17 hex)	Read/Write Registers	Read/Write Registers

devices through various locations such as I/O, memory, and various internal registers. MODBUS devices can be configured as a MODBUS slave or as a MODBUS master and slave depending on the device. In order to be a MODBUS master, the device must be able to send MODBUS commands, using at least the basic set of functions, to other devices. Figure 4 shows a table of MODBUS functions, where conformance class 0 is the base functions that any device must support to be MODBUS capable. The TRi [‘Super’](#) PLCs can be used as a MODBUS master and they support the highlighted functions in Figure 4: Figure 4 – Modbus Functions

The general Modbus frame consists of 4 different parts.

1. The address block – PLC id
2. The function code – HEX code from Figure 4
3. Data – location of data to read/write + how many registers to read/write if a multi read/write function is used. If a write function is used then the data to write is included.
4. Error check – either CRC or LRC

Figure 5 below shows a general Modbus frame.



Figure 5 – Modbus FrameThe TRi ‘[Super](#)’ PLCs support other protocols such as the native Host Link Commands, but only the ‘[Super](#)’ PLCs support MODBUS. There are two formats for MODBUS: MODBUS RTU, and MODBUS ASCII. The following explanation of MODBUS RTU, and MODBUS ASCII only goes into a little detail. If more detail is required, the information can be found through the “[Links](#)” section.

3.4.2 MODBUS RTU

Modbus RTU is the same as Modbus ASCII except for a few things.

1. Binary data is sent out directly rather than being converted into characters. Same amount of data is sent out in half the bytes.
2. The start of a Modbus frame is a 3.5 character gap in transmission
3. The end of a Modbus frame is a 3.5 character gap in transmission
4. Errors are checked using a 16 bit CRC (Cyclic Redundancy Check)

A Modbus RTU command/response block is shown below in Figure 6.

Start	Address	Function	Data	CRC 16	END
Silence of 3.5 char times	1 byte	1 byte	# byte	2 bytes	Silence of 3.5 char times

Figure 6 – Modbus RTU Command/Response Block

3.4.3 MODBUS ASCII

Modbus ASCII is the same as Modbus RTU except for a few things.

1. Each byte is sent out in two ASCII characters
2. The start of a Modbus frame is the “:” character (colon)
3. The end of a Modbus frame is a CR + LF (Carriage Return + Line Feed)
4. Errors are checked using an LRC (Longitudinal Redundancy Check)

A Modbus ASCII command/response block is shown below in Figure 7.

3.4.4 MODBUS ASCII VS. RTU

Depending on the application and the specific devices that are communicating, it may be better to use MODBUS ASCII or it may be better to use MODBUS RTU.

If you are trying to interpret the data communication between devices using some sort of serial listening hardware/software, then it will be better to use MODBUS ASCII if all devices communicating support it. This is because it is easier to interpret ASCII data than binary data.

If you transferring a lot of data, it may be better to use the MODBUS RTU protocol because it sends the data into smaller packets as explained in section 2.4.2 MODBUS RTU.

START	Address	Function	Data	LRC Check	CRLF
:	2 chars	2 chars	# chars	2 chars	2 chars

Figure 7 – Modbus ASCII Command/Response Block

4 THE SAMPLE PROGRAM

4.1 How It Works

4.1.1 Basics

The sample program can be programmed into and operated from any TRi [‘Super’](#) PLC. It uses ladder logic and basic programming combined to control any device that has an RS485 port and is capable of communicating in the MODBUS ASCII protocol. In this case the “control” actually means mapping I/O to and from the master and slave PLCs. In this application note the slave device of focus is the FMD888-10; however, any [‘Super’](#) PLC can be controlled using the sample program.

The ladder logic portion of the program is actually quite simple. It involves 6 contacts: 1st Scan, Clk:0.05s, SL1_in1, SL1_in4, SL2_in1, and Clk:0.02s and 4 custom functions: INIT, Update_IN, Empty_Function, and Update_OUT. Listed below, in Figure 4 – Ladder Logic, is a picture of the ladder logic screen. In the program there are comments integrated around the ladder circuits.

The contact 1st Scan is a special bit that is activated once on every power up or restart of the PLC. It is intended to activate initialization functions and coils. In this case, 1st Scan activates the INIT custom function that is used for initializing the baud rate in the master PLC, initializing variables used in the custom functions, and for initializing slave configurations. An example of the code for the INIT function is below.

The second contact, Clk:0.05s, is another special bit that is activated once every 0.05 seconds. This activates the Update_IN custom function that runs the code to update the inputs and map the slave inputs to the master relay[] bits. An example of the code for the Update_IN function is shown in the next section.

The next 3 contacts: SL1_in1, SL1_in4, and SL2_in1 are relay contacts that are inputs mapped from the slave PLC. For example, SL1_in1 corresponds to slave #1 (device#2), input#1. These 3 relay contacts are not part of the I/O mapping process, they are simple examples of how the mapped inputs can be used as contacts in ladder logic.

The last contact, Clk:0.02s is another clock signal that has a rising edge every 0.02 seconds. This activates the Update_OUT custom function that runs the code to update the outputs and map the master relay bits to the slave outputs. An example of the code for the Update_IN function is shown in a later section.

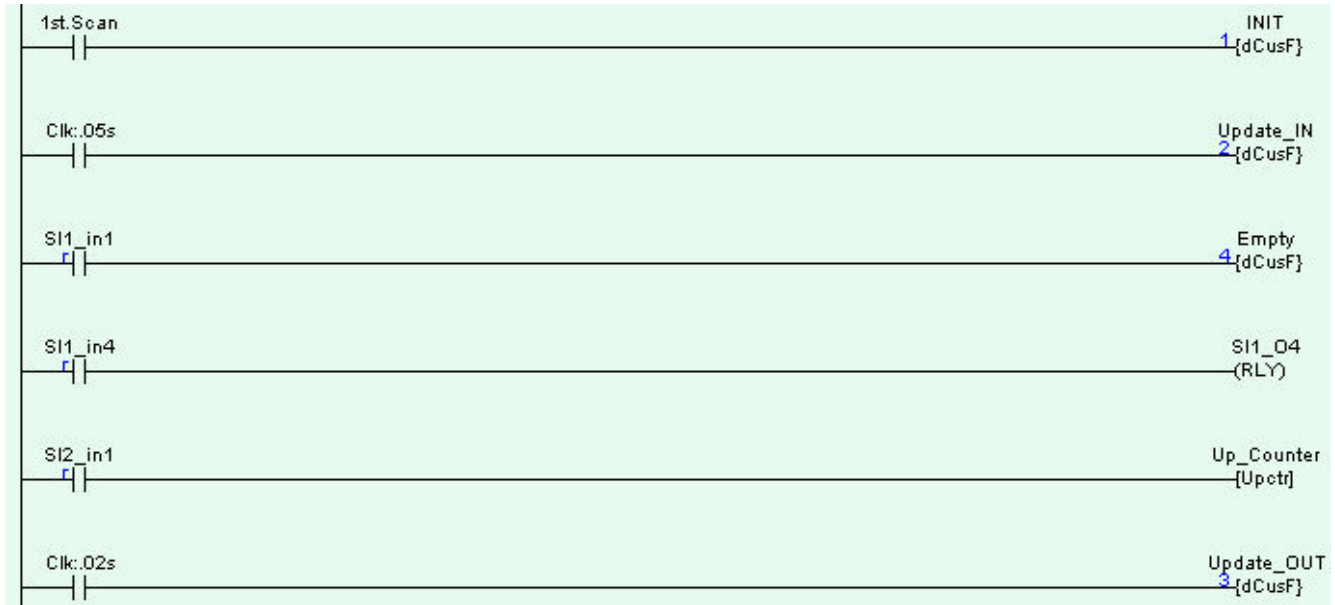


Figure 8 – Ladder Logic

'This function initializes the master and slave device settings.

'In order to change the number of devices in the network from the default of 3,

'D must be set to the new number of devices.

setbaud 3,6

'Set baud rate on com3 to 38400

I = 2

'Initialize id # to 2 (device id)

D = 3

'Total number of devices (including master)

Code taken from sample program: "RemotelO_FMD", Custom function #1: "INIT"

4.1.2 Custom function #2: "Update_IN"

Update_IN will read the input status and store the status in RELAY[I]. The details of how this is done are shown below.

1. Device # (I) is reset to 2 if it exceeds the maximum # of devices (D).
2. READMODBUS command is sent with 3 parameters: com port, device #, and MODBUS address
3. The value returned by READMODBUS is stored in system variable RELAY[I].
4. Error checking – check for a READMODBUS error using the STATUS(2) command

An example of the code is shown below.

'This function cycles through all of the remote devices from id 2 to D.

'It updates the input status of each device by:

' sending a request to read slave ('Super' # I) inputs

' storing the input status in RELAY[I]

IF I > D

'reset device id when it reaches last

device

I = 2

```
ENDIF
```

```
RELAY[I] = READMODBUS(3, I, 0)
```

'send readmodbus() command to com3 (RS485) to read 'inputs of device I

```
IF STATUS(2) = 0
```

'check for readmodbus error

*** add error response code here ***

```
ENDIF
```

Code taken from sample program: “RemotelO_FMD”, Custom function #2: “Update_IN”

4.1.3 Custom function #4: “Empty”

This function is not part of the Remote I/O process, it is there to show how the mapped inputs from the slave devices (specifically, slave device #1) can be used in ladder logic. In this case, the first input on the first slave device that was mapped to RELAY[2] bit #1 is used to activate a custom function. The custom function is empty because it just an example of what can be done with the mapped inputs. It could be filled with any code to do any number of things.

4.1.4 Relay Coil: “SL1_O4”

Again, this is not part of the remote I/O process, it is just to show how mapped inputs can be used in ladder logic. In this case, the fourth input on the first slave device, that was mapped to RELAY[2] bit #4, is used to activate a relay coil (SL1_O4). The relay coil SL1_O4 happens to be the coil that controls RELAY[6] bit #1. This relay bit (SL1_O4) represents output #4 on slave device #1. That means that when input #4 on slave device #1 is activated, it gets mapped to RELAY[2] bit #1, which activates relay coil SL1_O4. Relay coil SL1_O4 activates RELAY[6] bit #1 which gets mapped to output #4 on slave device #1. Essentially, input #4 on slave device #1 activates output #4 on slave device #1 indirectly through I/O mapping.

4.1.5 Up Counter: “Count_SL2_O1”

Again, this is not part of the remote I/O process, it is just to show how mapped inputs can be used in ladder logic. In this case, the first input of the second slave device, that was mapped to RELAY[3] bit #1, is used to activate an up counter (Count_SL2_O1). All this does is count up by 1 every time the first input on the second slave device is activated.

NOTE: in the previous sections, bit #x of RELAY[I] was referred to many times. When I say bit #1, I mean the first bit of RELAY[I] which is actually indexed as “RELAY[I], 0” in code due to zero indexing. However, the “I” in RELAY[I] is indexed starting from 1.

4.1.6 Custom function #3: “Update_OUT”

This function is called every 0.02 seconds and it involves 5 steps.

1. Compare previous value of relay bit to current value.
2. If different continue update process, else leave function without updating.
3. Update old output status in RELAY[I + 4] with new status.
4. Use WRITEMODBUS to send the new output status to the current device via COM3 (RS485)
5. Error checking – check if there is a WRITEMODBUS error using STATUS(2)

The code for custom function Update_OUT is shown below.

```
'This function writes RELAY[I + 4] location of Master to outputs 1-16 of slave 'Super' device I
'-----
IF (DM[3995 + I] <> RELAY[I + 4])           'check if the outputs of device I (slave#I - 1) have
been updated
DM [3995 + I] = RELAY[I + 4]               'save new output status in specific dm location

WRITEMODBUS 3, I, 16, RELAY[I + 4]         'send writemodbus()command to com3 (RS485) to
write 'contents of RELAY[I + 4] to outputs of 'Super'
device I

IF STATUS(2) = 0                           'check for writemodbus error
*** add error response code here ***
ENDIF
ENDIF

I = I + 1                                  'next device
```

Code taken from sample program: “RemotelO_FMD”, Custom function #4: “Update_OUT”

4.2 How To Use It

In order to use this sample program, a number of steps must be taken:

1. The ID of each slave device must be set to the correct number, starting with 2 and increasing by 1 up to a maximum of 5. This can be done by connecting each slave device to a pc, with I-Trilogi, via RS232 and changing the ID with TLServer. The new ID will be permanent in the PLC until it's reset again. The master PLC can have an ID of 00 or 01.
2. Next the physical network must be set up. The network should include the master PLC connected to the slaves through RS485; also, each device will need power and if separate power supplies are being used then all of the commons must be connected together. Specific instructions for wiring can be found in the “[Serial Communications](#)” section under “[RS485](#)”.
3. Now the program needs to be modified if it is not already. Modifications should to be made to the INIT function so that the variable D is set to the correct value (total number of devices including master). Also, Main may need to be modified depending on the use of the sample program.

The program can now be downloaded into the master PLC and then run after a restart.

5 LINKS

The links will be organized by section according to the table of contents so that they are easy to find.

5.1 SERIAL COMMUNICATIONS

5.1.1 Auto485

[For more information on the Auto485, click here.](#)

5.1.2 MODBUS

[For more information on MODBUS communication related to Tri PLCs, click here.](#)

You will be brought to chapter 5 (MODBUS /OMRON Protocols Support) of the operation manual for the FMD Series. In chapter 5 you will find a detailed explanation of the MODBUS format along with mapping tables.

[To view the MODBUS spec sheet from modbus.org, click here.](#)

You will be brought to the MODBUS spec sheet that contains information on all aspects of MODBUS from the MODBUS website: www.modbus.org