TRIANGLE RESEARCH INTERNATIONAL

# USING THE E10+ PLC AS REMOTE I/O FOR THE TRI 'SUPER' PLC

Revision 1

## APPLICATION NOTE

## Copyright Notice and Disclaimer



## Revision Sheet

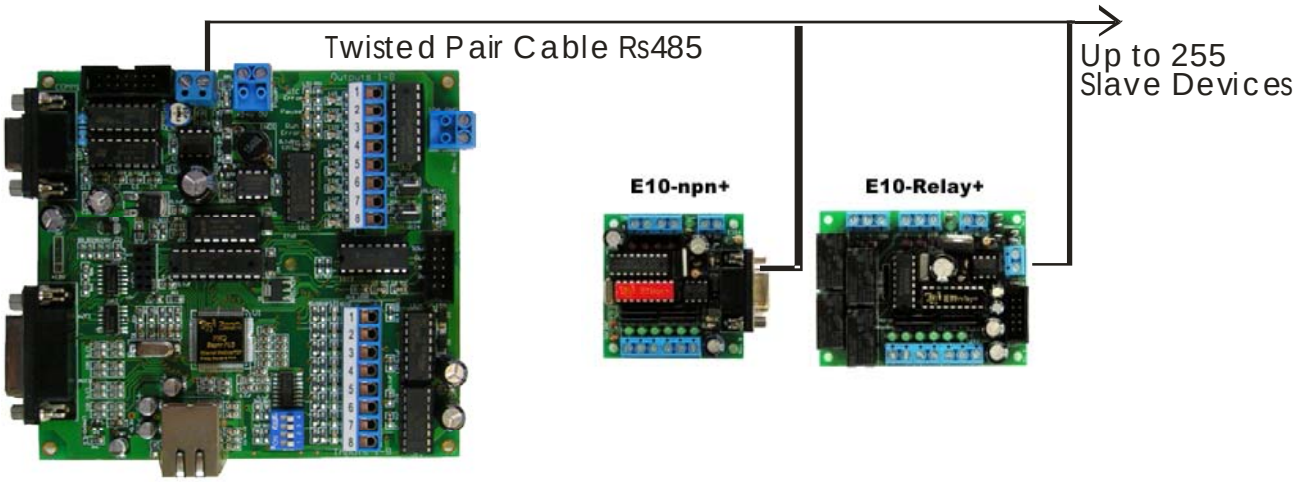
| Release No. | Date | Revision Description |
|---|---|---|
| Rev. 1 | 11/20/2012 | Updated to reflect current TRi PLC models. |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# TABLE OF CONTENTS

# 1 OVERVIEW

This application note explains how to go about using the E10+ as Remote I/O for the **TRi 'Super' PLC (Nano-10, FMD88-10, FMD1616-10, F1616-BA, F2424).** 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.

It will cover mapping I/O from one PLC to another, as well as serial communications using an RS485 connected network. 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 will discuss the network configuration as using one E10+ PLC as a slave; however, the sample program is programmed to have two slave E10+ PLCs and is expandable to up to 4 E10+ 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 an E10+ (npn) 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 E10+ PLC's 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. For even more information, see the "Links" section.

## 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.

### 2.3.2 Mapping Inputs from the E10+

The following line of code is used to map inputs from the slave PLCs to the master PLC.

```
RELAY[I] = (RELAY[I] & &HFFC0) | (HEXVAL(I$) & &H003F)
```

**Code taken from sample program: "RemoteIO_E10 basic", 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 and 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 is 2. Each 16 bit/contact RELAY[] array is used to both map inputs of a slave device to the Master PLC and to map outputs of a slave device to the Master PLC. RELAY[2] - RELAY[5] are designated for the slave devices.

RELAY[I] & &HFFC0

Since both the inputs and outputs of each slave device are mapped to a single RELAY[] location, it is necessary to mask the RELAY[] variable. The mask &HFFC0 is used to preserve the non input bits/contacts of RELAY[I] since only the inputs are being updated inside the “Update_IN” function.

HEXVAL(I$) & &H003F
I$ is a string that contains the input status of the current slave device. Its contents come from the NETCMD$ command to read the inputs of the current slave device. HEXVAL(I$) just converts the string to a number in HEX form. It is masked with the opposite HEX number as the above mask so that only the first 6 bits, which correspond to all 6 inputs, are extracted.

RELAY[I] = (RELAY[I] & &HFFC0) | (HEXVAL(I$) & &H003F)
This whole command will preserve all the bits except the inputs in RELAY[I] and then store only the inputs from the HEX value of I$.

### 2.3.3 Mapping Memory from the Master

The following 3 lines of code are used to map outputs from the master PLC to the slave PLCs.

```
DM[3995 +I] = RELAY[I] & &H0F00

DM[3900] = RELAY[I] & &H0F00

FOR X = 1 TO 8
        RSHIFT DM[3900],1
NEXT
```

**Code taken from sample program: “RemoteIO_E10 basic”, Custom function #3: “Update_OUT”**

The code to map outputs, 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.

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.

RELAY[I] & &H0F00
This code refers to the outputs of the slave devices. The mask &H0F00 is to ensure that only the 4 outputs are extracted from the RELAY[I] variable.

DM[3995 +I] = RELAY[I] & &H0F00
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] & &H0F00) 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].

DM[3900] = RELAY[I] & &H0F00
DM[3900] is used as a temporary variable where the output status of the current slave device is stored again. Since the output status of the slave devices is not stored at the beginning of the RELAY[I] variable, it is necessary to shift the bits over so that they are in the position of the first 4 bits (there are 4 ouputs). DM[3900] is used as the location for the bits to be shifted so that RELAY[I] isn't effected.

```
FOR X = 1 TO 8
        RSHIFT DM[3900],1
NEXT
```

This is the code that does the bit shifting mentioned above. This code will shift DM[3900] 8 bits to the right. Since the outputs start at bit 8, they need to be shifted 8 bits to the right so that the first output is the first bit of DM[3900]. Therefore, the code “RSHIFT DM[3900],1” will be repeated 8 times to complete the bit shift.

For example, if the value of the outputs for a slave device was 3 decimal (11 binary and 3 HEX), then the value of RELAY[I] & &H0F00 would be 768 decimal or 0000 0011 0000 0000 binary or 0300 HEX. Once the value of “RELAY[I] & &H0F00” has been placed in DM[3900], the value of DM[3900] will be the

same 768 decimal. After doing "RSHIFT DM[3900],1" eight times using the FOR loop, the value of DM[3900] will be 3 decimal, or 11 binary (14 leading zeros), or 0003 HEX.

### 2.3.4 Network communication

There are a couple of different ways to send data or commands from the master to the slave PLC. The function that I used in the sample program is the function that I will describe here, which is the "Netcmd$" function. The "Netcmd$" function makes communication easy in this case since we are talking in the native host link protocol.

The "Netcmd$" command takes 2 parameters; the COM port that will be used to send out the command string, and the command string itself. Netcmd$ will send out the command and store the response to a string.
The command will look like this:

A$ = Netcmd$(3, I$)

Where I$ is the command, A$ is the response, and 3 is the COM port used for data transfer between the devices.

The "Netcmd$" command will automatically append the "*" + "CR" (Asterisk and Carriage Return) components to the end of the command string that you are sending. Also, it will automatically calculate the correct FCS (Frame Check Sequence) and append it to the end of the command string, right after the "CR". All that is left to do is send the "@" + the Header + the Data in a string and Netcmd$ will take care of the rest. For more information on formatting strings for Host Link commands see the Host Link Commands section of Serial Communications.

There are 2 different kinds of command strings that will be sent to the slave PLC using "Netcmd$":

1. A command to read the inputs of the slave (RI) – "@02RI00"
2. A command to write the outputs of the slave (WO) – "@02WO00xxxx"

The first command, read inputs, will read the input status of the first 16 inputs of device #2. The "00" 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 E10+ only has 6 inputs total, only one input channel can be read – channel 0. An example of the code to read inputs is listed below.

```
G$ = STR$(I, 2)          'Build id part of netcmd string
I$ = "@" + G$ + "RI00"   'Build rest of netcmd string to read inputs of device I
I$ = NETCMD$(3, I$)      'Send command to com3 (RS485)
```

**Code taken from sample program: "RemoteIO_E10", Custom function #3: "Update_IN"**

The second command, write outputs, works exactly the same as read inputs except that data is being written to output channel 0. The data is "xxxx", which is the value of DM[3900] converted into a 16 bit Hex string. However, only the least significant 4 bits are important, as mentioned in section "1.3.3 Mapping Memory from the MASTER", because there are only 4 outputs on the E10+. An example of the code to write outputs is listed below.

```
R$ = HEX$(DM[3900], 2)   'convert dm[3900] (RELAY[I] & &H0F00) location into a string
O$ = "@" + G$ + "WO00" + R$   'build hostlink command
O$ = NETCMD$(3, O$)      'send host link cmd to write location DM[3900] (RELAY[I] &
                         &H0F00) of 'TRi 'Super' PLC to outputs 1-4
```

**Code taken from sample program: "RemoteIO_E10", Custom function #4: "Update_OUT"**

# 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 E10+ PLC's only have physical connections for RS232, but RS485 is jumper selectable. The +ve wire for RS485 is pin 5 of the DB9 connector and the –ve wire is pin 2. An example of this is shown below in Figure 1 – RS485 Network. A more detailed explanation of the connection is shown below Figure 1.

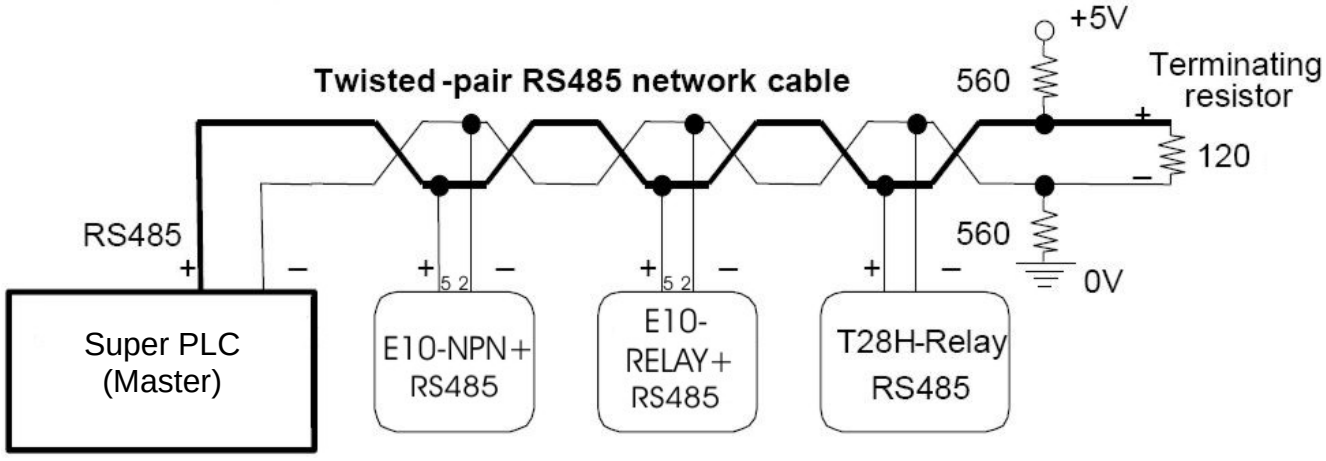



**Figure 1 - RS485 Network**

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 32 controllers (including the host computer node) to be networked together. When fitted with 1/8 power RS485 driver such as the 75HVD3082, up to 5 devices can be connected together. The twisted-pair cable goes from node to node in a daisy chain fashion and should be terminated by a 120ohm resistor as shown below.

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 3-2. 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 Host Link Commands

This is a protocol used to communicate between devices. The TRi 'Super' supports other protocols such as MODBUS, but the E10+ only supports Host Link Commands. Therefore, only Host Link Commands will be discussed here. There are two formats for host link commands; point to point, and multi point. The following explanation of point-to-point and multi-point only goes into a little detail. If more detail is required, the information can be found through the "Links" section.

### 3.4.1 Point to Point

Point to point communication is meant for a network of 2 devices because commands are sent out with no device id. For an example of the point-to-point communication process, see Figure 2 – Point to Point below.

The format for sending data is as follows:

1. The master sends character "ctrl 5" (ASCII 05) to signal slave that a command is coming.
2. The slave receives character "ctrl 5", sends back a "ctrl 5" character to the master, and prepares itself for an incoming command.
3. The master receives the "ctrl 5" response and sends the command.
4. The slave receives the command and sends back a response specific to that command.
5. The master receives the response and confirms that it is the correct response.

### 3.4.2 Multi Point

Multipoint communication is for a network of 2 or more devices, where each device has it's own id. For an example of multi-point communication, see Figure 3 – Multi Point below. The format for multipoint communication is as follows:

1. The master sends a command string containing "@", a 2 character id "__", the 2 character header, the data, the 2 character FCS (frame check sequence), and the "*" character. A carriage return is automatically appended to the end of all multipoint commands.
2. All of the slaves receive the command and check if it contains their specific id.
3. The slave device with the correct id checks that a valid command was sent.
4. The slave device responds with a response string containing "@", its 2 character id, the response, the 2 character FCS (frame check sequence), and the "*" character. A carriage return is automatically appended to the end of all multipoint responses.
5. The master checks that the correct response has been sent.

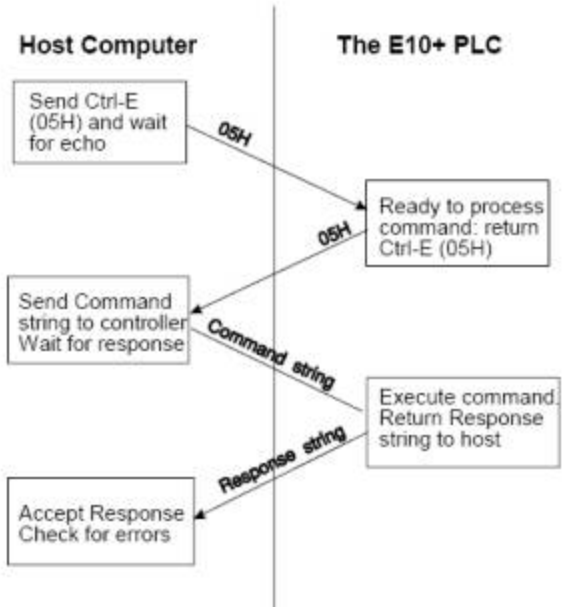



**Figure 2 - Point to Point**

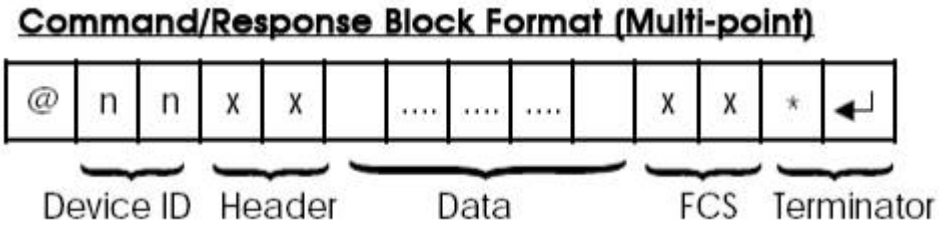



**Figure 3 - Multi Point**

# 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 TRI native protocol (Host Link commands). In this case the "control" actually means mapping I/O to and from the master and slave PLCs. In this application note the device of focus is the E10+; however, any TRI 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.

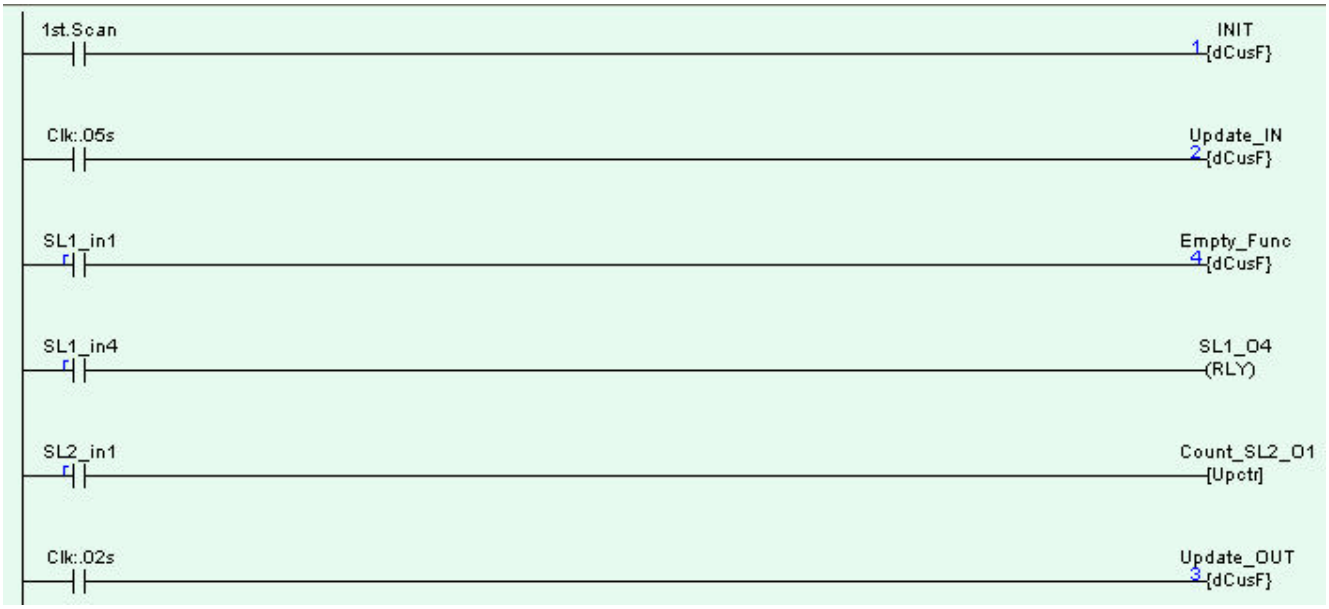



**Figure 4 – Ladder Logic**

```
'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 3,3
        'Set baud rate on com3 to 9600

I = 2
        'Initialize id # to 2 (device id)
D = 3
        'Total number of devices (including
master)
```

**Code taken from sample program: "RemoteIO_E10", Custom function #1: "INIT"**

### 4.1.2 Custom function #2: "Update_IN"

Update_IN will read the input status and store the status in the first 6 bits of RELAY[I]. The details of how this is done are shown below.

1. The current device number ( I ) is checked against the maximum allowed ( D )
2. If "I" exceeds "D" then "I" is reset else "I" stays the same.
3. The ID string is built
4. The command string with the ID string is built
5. The command is sent using "Netcmd$"
6. Error checking – check if the response string is empty
7. Error checking – check if there is a framing error
8. Error checking – check if there is a command error
9. Extract input status from response string
10. Store input status in first 6 bits of RELAY[I].

An example of the code is shown below.

```
'This function sends a request to read slave (e10 # I) inputs
'The input status is stored in RELAY[1]

IF I > D                        'when I (device#) is larger than D (# of devices), reset I to 2 (1st slave)
        I = 2
ENDIF

G$ = STR$(I, 2)                         'build id part of netcmd string
I$ = "@" + G$ + "RI00"                  'build rest of netcmd string to read inputs of device I
I$ = NETCMD$(3, I$)                     'send command to com3 (RS485)

IF STRCMP(I$,"") = 0                    'check if string is empty
        '*** add error handling code here ***
ENDIF

IF STRCMP(I$, "@" + G$ + "FE") = 0      'check for framing error
        '*** add error handling code here ***
ENDIF

IF STRCMP(I$, "@" + G$ + "EE") = 0      'check for command error
        '*** add error handling code here ***
ENDIF

I$ = MID$(I$, 6, 2)                     'extract hex input value from response string

RELAY[I] = (RELAY[I] & &HFFC0) |        'write inputs 1-6 of e10 device# I to relay bits 1-6 of FMD relay[I].
(HEXVAL(I$) & &H003F)       'preserve relay[I] (only 6 out of 16 bits are used here and 4 bits 'are for o/ps)
                            'mask relay[I] with binary 00111111 to represent 6 out of 8 relays used
```

**Code taken from sample program: "RemoteIO_E10", Custom function #5: "Update_IN"**

### 4.1.3 Custom function #4: "Empty_Func"

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[2] bit #12. 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[2] bit #12 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. When I say RELAY[I] bit #12, it would actually be indexed as RELAY[I], 11 in code even though it is the 12$^{th}$ bit of RELAY[I]. 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 11 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] with new status
4. Store new status of outputs in temp variable for bit shifting.
5. Shift outputs in temp variable 8 bits to the right so output 1 is bit 1.
6. Convert the output status into a string
7. Build the command string with the output status string
8. Send the command using "Netcmd$"
9. Error checking – check if the response string is empty
10. Error checking – check if there is a framing error
11. Error checking – check if there is a command error

(The error-checking portion has no action if an error were found. That would have to be added.)

The code for custom function Update_OUT is shown below.

```
'This function writes RELAY[I] locations of FMD to outputs 1-4 of e10 device I

'-------------------------------------------------------------------------------------------------------------
IF DM[3995 + I] <> (RELAY[I] & &H0F00)                  'compare old output status (DM[3995 +i]) to new
                                                        'output status (RELAY[I] & &H0F00)

DM[3995 +I] = RELAY[I] & &H0F00                 'assign new output status to old if changed
DM[3900] = RELAY[I] & &H0F00                    'store slave output status in dm for bit manipulation

FOR X = 1 TO 8
RSHIFT DM[3900],1                               'shift bits so that slave output bits are in the lower
NEXT                                            'nibble of the data being sent

R$ = HEX$(DM[3900], 2)                  'convert dm[3900] (RELAY[I] & &H0F00) location into a string

O$ = "@" + G$ + "WO00" + R$                     'build hostlink command

O$ = NETCMD$(3, O$)                             'send host link cmd to write location "RELAY[I] &
                                                '&H0F00" of FMD to outputs 1-4
'-------------------------------------------------------------------------------------------------------------
'Error Checking

IF STRCMP(O$,"") = 0                            'check if string is empty
        '*** add error response code here ***
ENDIF

IF STRCMP(O$, "@" + G$ + "FE") = 0                      'check for framing error
        '*** add error response code here ***
ENDIF

IF STRCMP(O$, "@" + G$ + "EE") = 0              'check for command error
        '*** add error response code here ***
ENDIF
ENDIF
I = I + 1                                       'next device
```

**Code taken from sample program: “RemoteIO_E10”, 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 WinTrilogi, via RS232 and changing the ID with the WinTrilogi software. The new ID will be permanent in the PLC until it's reset again. The master PLC can have an ID of 00, 01, or anything higher than 5.

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, the ladder logic may need to be modified depending on the use of the sample program.

4. 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 Remote I/O

### 5.1.1 The Physical Network

For more information on wiring the RS485 network, click here.

## 5.2 SERIAL COMMUNICATIONS

### 5.2.1 Auto485

For more information on the Auto485, click here.

### 5.2.2 Host Link Commands

For more information on Host Link Commands, click here.
The above link downloads the FMD88-10 user manual. You can reference chapter 14 (SERIAL COMMUNICATIONS), chapter 15 (HOST LINK PROTOCOL INTRODUCTION), and chapter 16 (HOST LINK PROTOCOL FORMAT) of the operation manual for the TRi 'Super' PLC. These chapters will provide detailed information on serial communication, the host link protocol, and the available host link commands, which is applicable to all 'Super' PLCs.