TRIANGLE
RESEARCH
INTERNATIONAL

USING THE E10+ PLC AS REMOTE I/O FOR THE TRI 'SUPER' PLC (ADVANCED)

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.

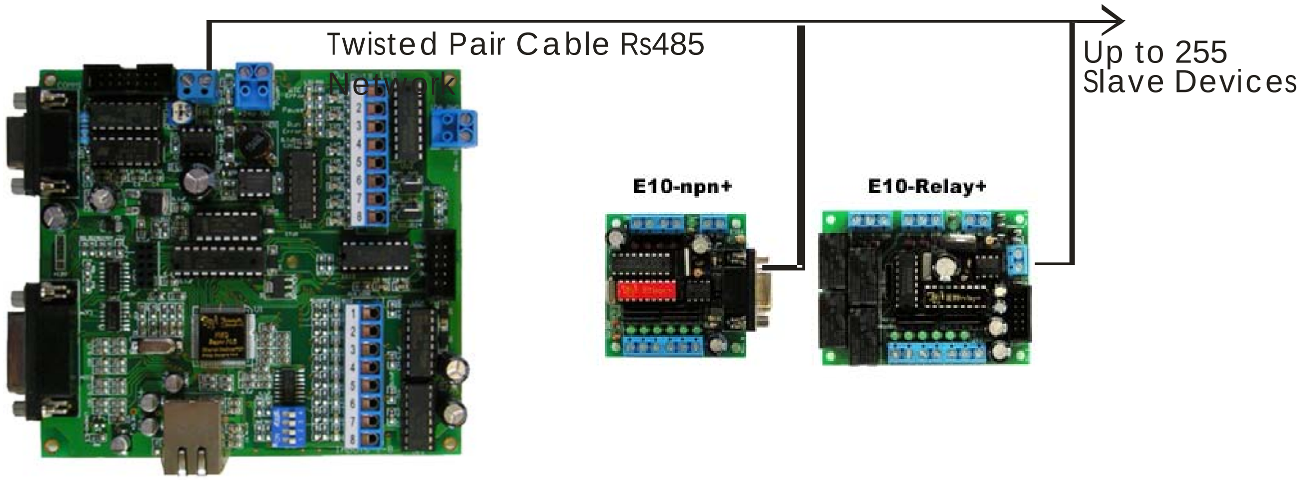
TABLE OF CONTENTS

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 255 E10+ slaves (256 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 256 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.

2.3.2 Mapping Inputs from the E10+

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.

Inputs from the E10+ PLC's can be mapped to the internal DM[] memory locations in the TRi 'Super' PLC. Each E10+ PLC has 6 inputs and the TRi 'Super' has 4000 16-bit DM[] memory locations. First the inputs from the E10+ our mapped to the first 6 relays of the FMD1616-10, they are then compared to the value in the devices specific DM[] location. If the value is different then the value in the DM[] location is updated with the new input status. The first 6 relays can be accessed as a group from the system variable RELAY[1], which will actually contain the states of the first 16 relays. RELAY[1] will be masked so that only the first 6 relay bits will be read.

For example, if device #2 (actually the first slave, id = 02) had all its inputs off the last time its status was checked and this time all of its inputs were on, then the value of the masked system variable RELAY[1] will be decimal 63 (binary 0000000000111111) and the value of DM[3488 + 2] will be decimal 0 (binary 0000000000000000). The value in memory location DM[3488 + 2] will then be updated to decimal 63 (binary 0000000000111111). An example of the code that does this is shown below.

```
IF DM[3488 + I] <> RELAY[1] & &H003F
        DM[3488 +I] = RELAY[1] & &H003F      'assign new output status to old if changed
        CALL Main                                    'function to control individual devices based on
                                                     'input status
ENDIF
```

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

The sample program will cycle through the number of devices that was selected in the program initialization. For each device, the same system variable RELAY[1] will be used as temporary storage for the current input status.

2.3.3 Mapping Memory from the Master

Mapping memory from the TRI '[Super]' PLC to the outputs of the slave E10+ PLC is very similar to mapping inputs from the E10+ PLC to the memory of the TRi '[Super]' 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.

In the previous section "Mapping Inputs from the E10+", the system variable RELAY[1] was used as a temporary variable to hold the current devices input status. The system variable RELAY[2] is used the same way, as a temporary variable for the current devices output status. From the temporary variable, RELAY[2], the output status is transferred to DM[] memory location DM[3743 + I], which is specific to the current device. The data is then transferred to the current slave PLC in a command string. Just as RELAY[1] was masked, RELAY[2] will be masked as well to preserve the rest of the variable. Each E10+ PLC has 4 digital outputs; so only 4 bits out of 16 are used from RELAY[2].

For example, if device #2 (actually the first slave, id = 02) had all its outputs off the last time its status was checked and this time all of its outputs were on, then the value of the masked system variable RELAY[2] will be decimal 15 (binary 0000000000001111) and the value of DM[3743 + 2] will be decimal 0 (binary 0000000000000000). The value in memory location DM[3743 + 2] will then be updated to decimal 15 (binary 0000000000001111). An example of the code that does this is shown below.

```
IF DM[3743 + I] <> RELAY[2] & &H000F
        DM[3743 +I] = RELAY[2] & &H000F      'assign new output status to old if changed
        CALL Update_OUT                              'function to write outputs of current device
ENDIF
```

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

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$" command will automatically append the "*" + "CR" (Asterisk and Carriage Return) components to the end of the command string that you are sending. Also, it will automatically calculate the correct FCS (Frame Check Sequence) and append to the end of the command string, right after the "CR". All that is left to do is send the "@" + the Header + the Data in a string and Netcmd$ 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[3743 + I] converted into a 16 bit Hex string. However, only the least significant 4 bits are important because there are only 4 outputs on the E10+. An example of the code to write outputs is listed below.

```
R$ = HEX$((DM[3743 + I]), 2)  'Convert DM[3743 + I] locations into a string
O$ = "@" + G$ + "WO00" + R$   'Build host link command
O$ = NETCMD$(3, O$)                    'Send host link cmd to write location DM[3743 + I] of TRi ‘Super’
                                       to ‘outputs 1-4 of E10+
```

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 the M-Series PLC; one is RS232, and the other is RS485. 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. More information can be found on serial communications in the Links section.

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 M-Series 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.

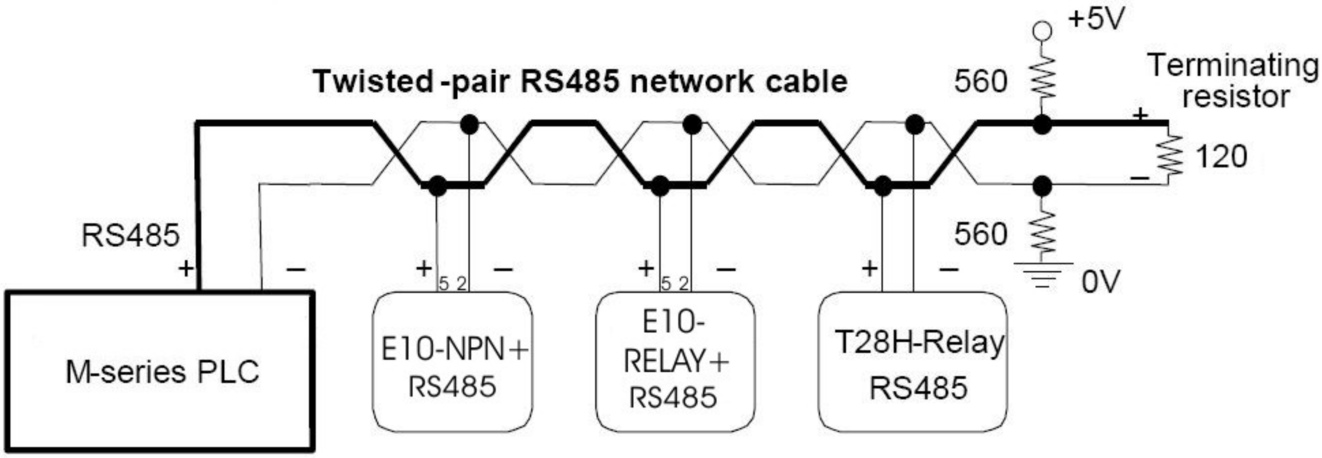


Figure 1 - RS485 Network

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 an M-series 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 M-Series 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.

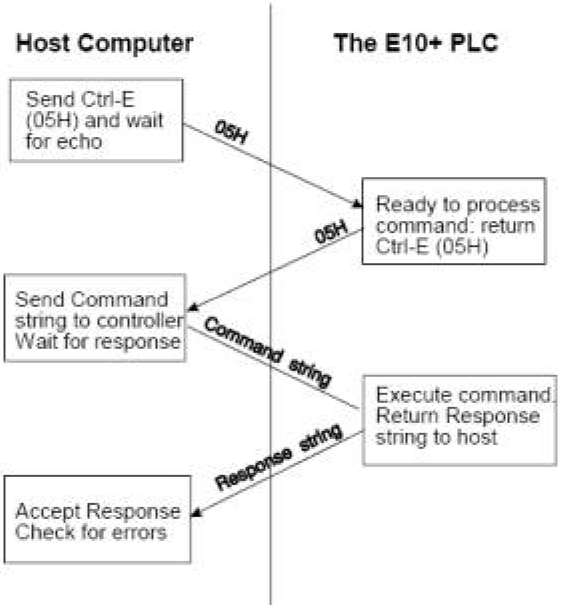

Host Computer
The E10+ PLC
Send Ctrl-E (05H) and wait for echo
05H
Ready to process command: return Ctrl-E (05H)
05H
Send Command string to controller Wait for response
Command string
Execute command. Return Response string to host
Response string
Accept Response Check for errors

Figure 2 - Point to Point

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.

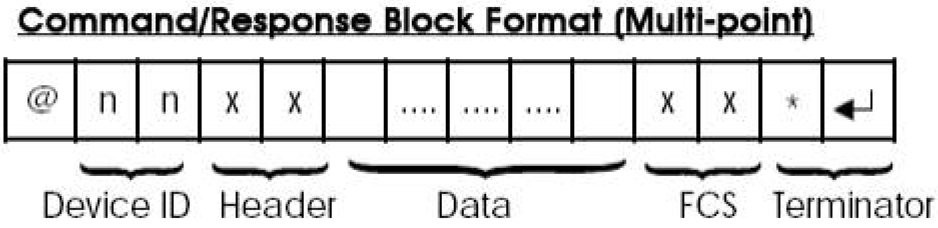


Figure 3 - Multi Point

4 THE SAMPLE PROGRAM

4.1 How It Works

4.1.1 Basics

The sample program can be programmed into and operated from any M-Series 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 2 contacts: 1st Scan and Clk:0.1s and 2 custom functions: INIT and I_O_Map. Listed below, in Figure 4 – Ladder Logic, is a picture of the ladder logic screen. In the program there are comments included above 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 and the LCD display. An example of the code for the INIT function is below. The second contact, Clk:0.1s, is another special bit that is activated once every 0.1 seconds. This activates the I_O_Map custom function that runs the code to everything, including: Mapping I/O, communicating with the slaves, displaying to the LCD. An example of the code for the I_O_Map function is below.

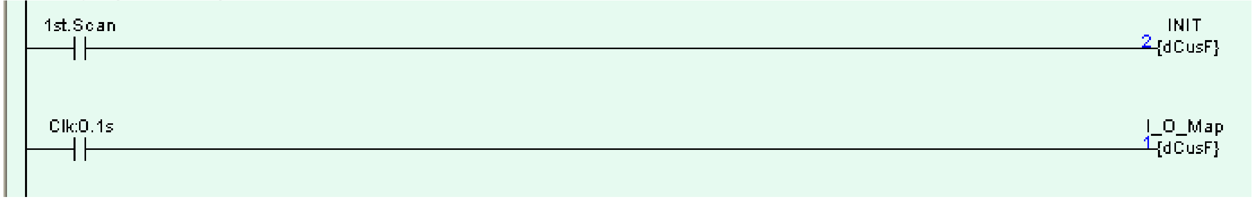


Figure 4 – Ladder Logic

```
'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)

'----------------------------------------------------------------------------------------------------------------
'Slave Device 1 Initialization

SETLCD 1,1, "INPUT 1 OF DEV 2 IS:"      'Initialize display for device 2
SETLCD 2,17, "OFF"

'----------------------------------------------------------------------------------------------------------------
'Slave Device 2 Initialization

SETLCD 3,1, "INPUT 1 OF DEV 3 IS:"      'Initialize display for device 3
SETLCD 4,17, "OFF"
```

Code taken from sample program: “RemoteIO_E10”, Custom function #2: “INIT”

4.1.2 Custom function #1: “I_O_Map”

The custom function I_O_Map is the program; everything is run through this function in one program cycle. The order of events of I_O_Map is shown below in numerical order.

1. It starts by checking the current device ID (variable I) and resetting it if it is past the ID limit, which is the number of devices connected (variable D) defined by the user in custom function INIT.
2. It will then call the custom function Update_IN, which will read the input status of the current slave device (variable I) and store it into system variable RELAY[1]. (Update_IN is covered in the next section.)
3. It will then compare the new input status stored in RELAY[1] to the old input status stored in DM[3488 + I]. (The number 3488 is an offset used so that the last 510 spots of DM memory are used for I/O mapping, which frees up the rest of DM memory for other program use)
4. If the new status is different from the old status then steps 4. a) and 4. b) will be executed before moving to step 5, otherwise 4. a) and 4. b) are skipped.
    a) DM[3488 + I] will be updated with the new input status.
    b) The custom function Main will be called. (More detail on Main is in a later section)
5. Next, the new output status stored in RELAY[2] is compared to the old output status stored in DM[3743 + I]. If the new and old output status is different then steps 5. a) and 5. b) are executed before moving on to step 6, otherwise steps 5. a) and 5. b) are skipped.
    a) DM[3743 + I] will be updated with the new output status.
    b) The custom function Update_OUT will be called. (More detail on Update_OUT is in a later section)
6. The old output status of the next device is loaded into system variable RELAY[2] to prepare for the next cycle. If the current ID is the value of D (last slave device) the old output status of the first slave device is stored in RELAY[2].
7. The device ID (variable I) is incremented by 1.

```
'This function cycles through all of the remote devices from id 2 to D.
'It reads the input status of each device and runs the code that is specific to that device

IF I > D                                'Reset device id when it reaches last device
        I = 2
ENDIF
CALL Update_IN                                  'Function to read inputs of current device
IF DM[3488 + I] <> RELAY[1] & &H003F
        DM[3488 + I] = RELAY[1] & &H003F    'Assign new input status to old if changed
        CALL Main                               'Function to control individual devices based on
ENDIF                                           'input status

IF DM[3743 + I] <> RELAY[2] & &H000F
        DM[3743 +I] = RELAY[2] & &H000F     'Assign new output status to old if changed
        CALL Update_OUT                         'Function to write outputs of current device
ENDIF
IF I = D
        RELAY[2] = DM[3743 + 2]                 'Reset system variable for first device
ELSE
        RELAY[2] = DM[3743 + I + 1]         'Reset system variable for next device
ENDIF
I = I + 1                               'Next device
```

Code taken from sample program: “RemoteIO_E10”, Custom function #1: “I_O_Map”

4.1.3 Custom function #4: "Update_IN"

Update_IN will read the input status and store the status in RELAY[1]. The details of how this is done are shown below.

1. The ID string is built
2. The command string with the ID string is built
3. The command is sent using "Netcmd$"
4. Error checking – check if the response string is empty
5. Error checking – check if there is a framing error
6. Error checking – check if there is a command error
7. Extract input status from response string
8. Store input status in RELAY[1].

(If any steps of the error-checking portion are true, the function is exited.)

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]

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
        RETURN
ENDIF

IF STRCMP(I$, "@" + G$ + "FE") = 0         'check for framing error
        RETURN
ENDIF

IF STRCMP(I$, "@" + G$ + "EE") = 0         'check for command error
        RETURN
ENDIF

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

RELAY[1] = (RELAY[1] & &HFFC0)     'write inputs 1-6 of e10 to relays 1-6 of the TRi 'Super'
PLC.
| (HEXVAL(I$) & &H003F)            'Preserve relay[1] (only 6 out of 16 bits are used here)
                                   'Mask relay[1] with binary 00111111 to represent 6 out of 8
relays used
```

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

4.1.4 Custom function #5: "Main"

Main is the function that controls actions and events based on the input status. Each device can have its own section of code for its own actions. Below is the code from the Main function.

The code below is just an example of what can be done. It uses the LCD to display whether Input 1 from device 2 (first slave device) is on or off. Also, if input 4 is on for device 2 then all 4 outputs are turned on for device 2. The same code is repeated for device 3; however, it is not necessary to have the same code for each device. If you had the same code for each device it could be combined to save code space.

```
'This function performs tasks based on the input conditions of the remote I/O
'This is as simple example of how an output or multiple outputs can be manipulated based on a devices
input status
'------------------------------------------------------------------------------------
'Device 2
IF I = 2
        IF TESTIO (E10_in1)                                  'Input 1 of device 2 is on
                SETLCD 2,17, "ON "
        ELSE                                                         'Input 1 of device 2 is off
                SETLCD 2,17, "OFF"
        ENDIF
        IF TESTIO (E10_in4)                                          'Input 4 of device 2 is on
                RELAY[2] = &HF                                       'Turn on all 4 outputs of device 2
        ELSE                                                         'Input 1 of device 2 is on
                RELAY[2] = 0                                 'Turn off all 4 outputs of device 2
        ENDIF
ENDIF
'------------------------------------------------------------------------------------
'Device 3
IF I = 3
        IF TESTIO (E10_in1)                                  'Input 1 of device 3 is on
                SETLCD 4,17, "ON "
        ELSE                                                         'Input 1 of device 3 is off
                SETLCD 4,17, "OFF"
        ENDIF
        IF TESTIO (E10_in4)                                          'Input 4 of device 3 is on
                RELAY[2] = &HF                                       'Turn on all 4 outputs of device 3
        ELSE                                                         'Input 1 of device 3 is on
                RELAY[2] = 0                                 'Turn off all 4 outputs of device 3
        ENDIF
ENDIF
```

Code taken from sample program: “RemoteIO_E10”, Custom function #5: “Main”

4.1.5 Custom function #4: “Update_OUT”

This function is only called if the current output status is different from the previous output status. The function itself is quite simple. It involves 6 steps.

1. Convert the output status into a string
2. Build the command string with the output status string
3. Send the command using “Netcmd$”
4. Error checking – check if the response string is empty
5. Error checking – check if there is a framing error
6. 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 DM[3743 + I] locations of TRi ‘Super’ PLC to outputs 1-4 of e10 device I

'-----------------------------------------------------------------------------------------------------------------

R$ = HEX$((DM[3743 + I]), 2)            'convert DM[3743 + I] locations into a string

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

O$ = NETCMD$(3, O$)                     'send host link cmd to write location DM[3743 + I] of
```

```
                                                  ‘master 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
```

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 256. 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 256.

2. Next the physical network must be set up. The network should include the master PLC connected to the slaves

3. 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”.

4. 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.

5. 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.