Dynamic networking for programmable logic controller communication

Richard Andrew Hardison

DYNAMIC NETWORKING FOR PROGRAMMABLE LOGIC CONTROLLER COMMUNICATION

by

RICHARD ANDREW HARDISON

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURIUNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

2010

Approved by

Kelvin T. Erickson, Advisor
Ann Miller
Jagannathan Sarangapani

ABSTRACT

This thesis confronts the issue of set path communication in Programmable Logic Controllers (PLCs). In order to increase reliability in a networked automated process, it is necessary to develop a dynamic method of communication between PLCs. Three methods for dynamic messaging in PLCs are explored in this work. These methods are based on the real-time reconfiguration of message paths when communication fails on the originally configured path. The simplest method involves replacing the original message block with an Add-On Instruction (AOI) which can accept multiple communication paths and has the ability to switch between these paths to maintain communication. A more complex method involving a complete network scan in order to automatically build communication paths is also explored. Finally, a direct scan method having the ability to build communication paths between two controllers which are located on multiple common networks is developed. For the purpose of this research the Allen-Bradley ControlLogix platform was used.

ACKNOWLEDGMENTS

I would like to thank Dr. Erickson for his guidance and direction during the pursuit of a Master of Science degree. He was an integral part of my success as a graduate student. It has truly been a privilege and an honor to work with Dr. Erickson.

I would like to thank my graduate committee for their support and willingness to work with me.

I would like to thank my dad Richard L. Hardison and my good friend Chris Smiley who helped steer me in the direction of a graduate degree. I would also like to thank Terry and Linda Glick, Keith and Brenda Athmer, Doug and Patty McClendon, Don and Judy Baker, and Tom and Connie Cromwell. Without their moral support, guidance, and love this degree would not have been possible.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

With the extensive use of Programmable Logic Controllers (PLCs) in manufacturing facilities, in order to control and optimize processes, there is a need for a high standard of reliability. While stand alone networks which cannot be compromised from outside sources may be safe from attack they may still cause equipment downtime due to network hardware failures and other system faults. When equipment downtime can cost in the thousands of dollars per minute [1] it becomes necessary to minimize it. In order to combat downtime due to network failures it is possible to use redundant networks for communication between PLCs. To obtain these redundant networks PLCs may be placed on an automation network as well the facility Ethernet network. The automation network is used as the primary means of communication between PLCs with the facility network being used as a back-up. Placing the PLCs on the facility network also adds the ability for real-time evaluation of processes by engineers without having to be on the plant floor.

The current method of communication between PLCs utilizes a preconfigured set path to send data to or retrieve data from another controller. The method is not dynamic in any way and communication fails if any part of the network path fails. To minimize the amount of time communication is lost, methods have been developed which allow communication to be rerouted to another network. By reconfiguring the message path in real-time a communication fault can be recovered from quite quickly.

For the purpose of this research the Allen-Bradley ControlLogix platform has been used, although it is possible to transfer these methods to both the SLC-500 and PLC-5 platforms with a few changes.

2. PROGRAMMABLE LOGIC CONTROLLER COMMUNICATION

In a PLC network of ControlLogix processors there are two ways in which the controllers are able to share data with each other. These methods are producer/consumer connections and message blocks [2].

Producer/consumer connections are created at the tag level by defining the tag to be produced in one controller and consumed in another. The tag which is defined as produced is configured with a maximum number of consumers, where the default is one. The tag which is defined as consumed is configured with a path to the controller containing the produced tag as well as the name of the produced tag. This path is chosen from a drop down list containing controllers that have been added to the I/O tree of the project. For this reason it is necessary to define paths in the I/O tree to all controllers to be communicated with.

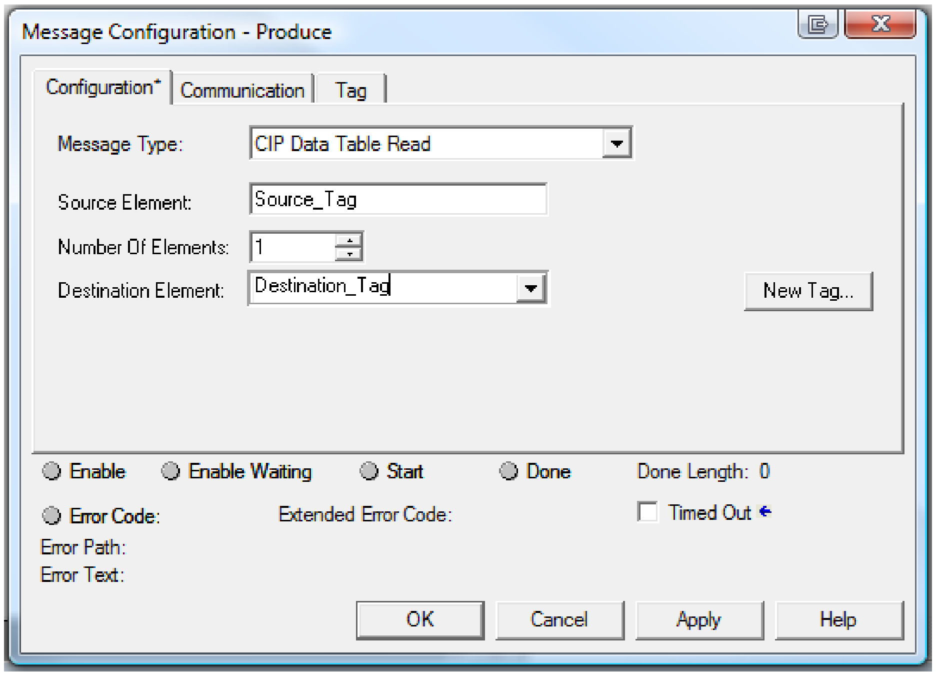
For the purpose of creating dynamic connections, producer/consumers tags do not lend themselves to the idea. This is due to the way RSLogix handles the consumer connection path making it inaccessible from the program. However, there is a work around for this. It is possible to define multiple consumed tags which draw from the same source, but use different paths to reach that source. This method loses feasibility though due to increased network traffic across non-preferred networks. Since all defined connections would be active at the same time, bandwidth is wasted on the connections that are not needed unless a network failure occurs on the preferred network. For this reason message blocks are the preferred method of communication with regards to dynamic connections.

2.1. MESSAGE CONFIGURATION

Message blocks provide for PLC to PLC communication and can be configured in a variety of ways depending on the intended communication and the type of PLC to be communicated with [2].

2.1.1. CIP Data Table Read/Write. The CIP Data Table Read is used to copy the value in the source tag located on another controller to the destination tag located on the local controller. It is important to note that the source and destination tag need to be of the same data type. Figure 2.1 on the following page shows the configuration tab of a CIP Data Table read message. The CIP Data Table Write differs from the CIP Data Table Read in only the location of the source tag and destination tag. For the CIP Data Table Write message, the source tag is located on the local controller and the destination tag is located on the remote controller. Otherwise, these two messages are configured the same. It is often preferred to use the read type messages whenever possible to avoid accidently writing over critical data in another controller.

2.1.2. Device Who Message. A module's identification information can be obtained directly from the module using a message type of CIP generic with a service type of "Device WHO". This identification information includes the vendor ID, product type, product code, major and minor firmware revisions, device status, serial number, and name of the module. Upon execution of the message, this data is stored in the destination tag which is anarray of SINT data type. The array must then be decoded into its appropriate parts. How these parts are stored in the array can be found in [3]. The configuration of a "Device WHO" message is shown in Figure 2.2.


Message Configuration - Produce

Configuration* | Communication | Tag

Message Type: CIP Data Table Read

Source Element: Source_Tag

Number Of Elements: 1

Destination Element: Destination_Tag

New Tag...

Enable | Enable Waiting | Start | Done | Done Length: 0

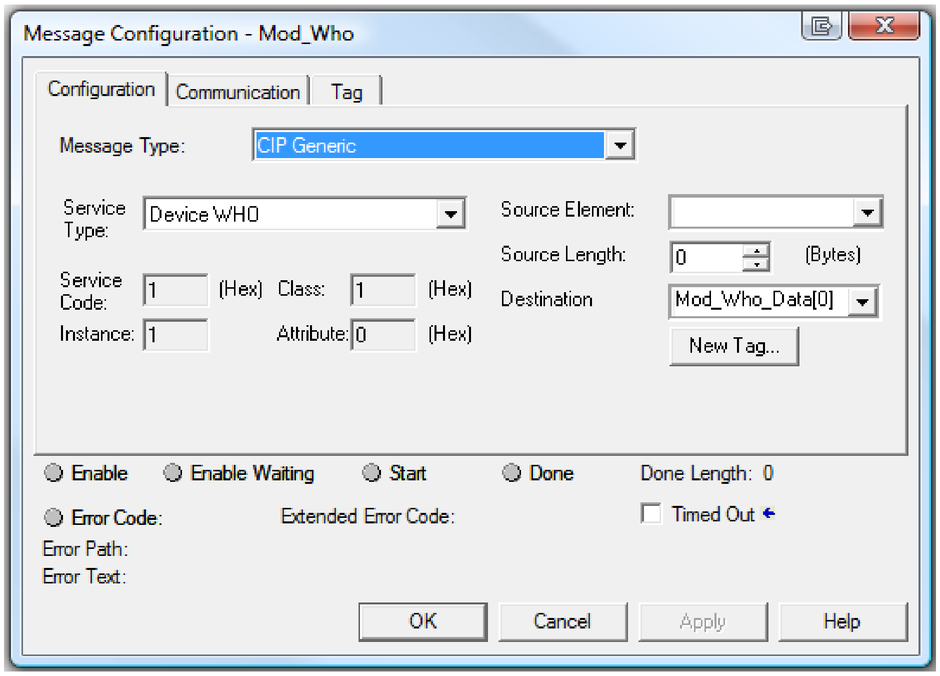
Error Code: | Extended Error Code: | Timed Out

Error Path:

Error Text:

OK | Cancel | Apply | Help

Figure 2.1. CIP Data Table Read Configuration Tab


Message Configuration - Mod_Who

Configuration | Communication | Tag

Message Type: CIP Generic

Service Type: Device WHO

Service Code: 1 (Hex) Class: 1 (Hex)

Instance: 1 Attribute: 0 (Hex)

Source Element:

Source Length: 0 (Bytes)

Destination: Mod_Who_Data[0]

New Tag...

Enable | Enable Waiting | Start | Done | Done Length: 0

Error Code: | Extended Error Code: | Timed Out

Error Path:

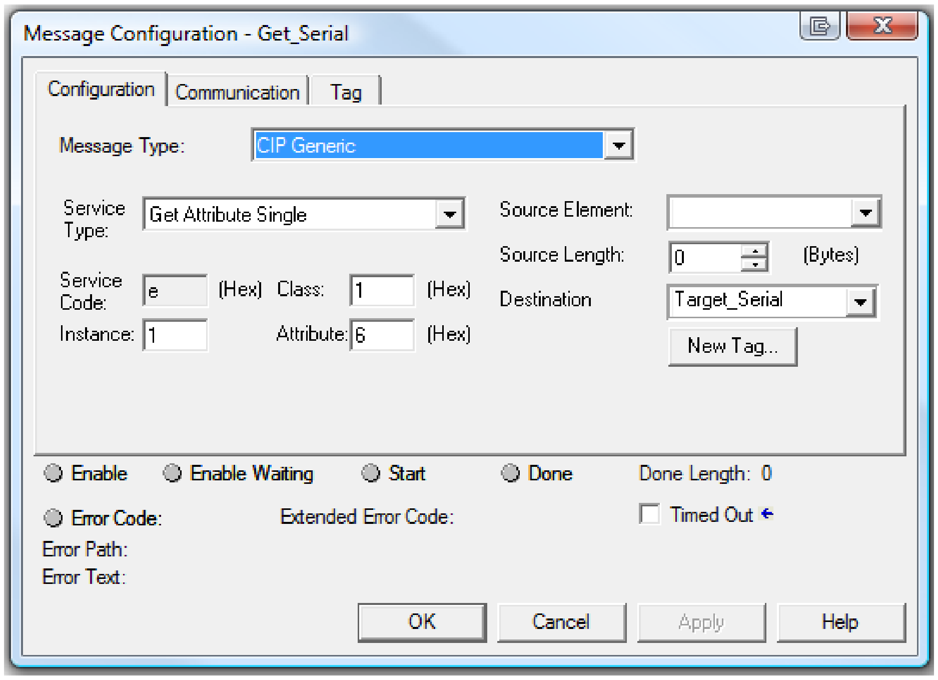
Error Text:

OK | Cancel | Apply | Help

Figure 2.2. “Device WHO” Message Configuration

2.1.3. Get Attribute Single. Get attribute single is a service type of the CIP generic message type. The data type returned by this type of message depends on the data being retrieved. Three instances of this were used in the process of this research. The first instance was used to obtain the serial number of a module, and when this is done the message returns a DINT. The configuration of a CIP generic message used to obtain the serial number of a module is shown in Figure 2.3 [4]. The second instance of this type of message was used to find the node address of a ControlNet module. This message also returns a DINT and the configuration is changed only slightly. These changes are in the Class and Attribute fields of the message configuration. The Class and Attribute values used to return the node address of a ControlNet module are F0 and 84 respectfully [5]. The third use of this message type was to retrieve the IP address from an Ethernet module. This message returns an array of SINT containing the IP address in the form of an ASCII string. The configuration for the message to retrieve the IP address uses a Class of F5 and an Attribute of 5 [6].

2.1.4. Message Path Configuration. There are multiple ways to define the path between two controllers. The simplest way is to build the path in the I/O Configuration tree of the project. Figure 2.4 shows a path built over Control Net from the main controller to a remote controller operating a conveyor system. An illustration of this path is shown in Figure 2.5. Though this method is simple it is limited by the types of communication modules that can be used (EWEB and ENET modules cannot be used this way) and does not show the full path in the communication configuration tab. This is shown in figure 2.6.


Message Configuration - Get_Serial

Configuration | Communication | Tag

Message Type: CIP Generic

Service Type: Get Attribute Single

Service Code: e (Hex) Class: 1 (Hex)

Instance: 1 Attribute: 6 (Hex)

Source Element:

Source Length: 0 (Bytes)

Destination: Target_Serial

New Tag...

Enable Enable Waiting Start Done Done Length: 0

Error Code: Extended Error Code: Timed Out

Error Path:

Error Text:

OK Cancel Apply Help

Figure 2.3. Get Attribute Single (Serial Number) Message Configuration

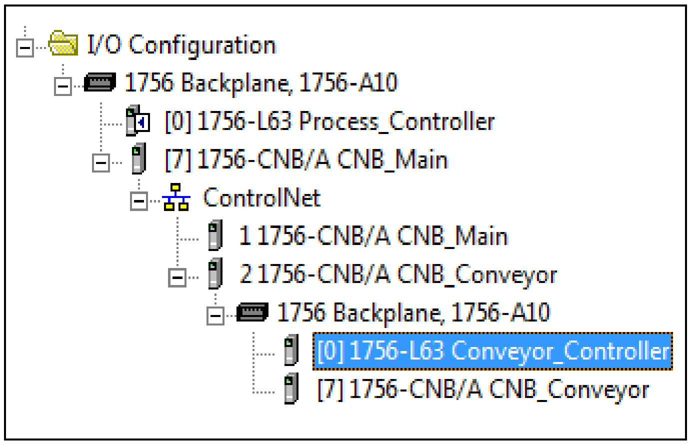

```
I/O Configuration
  1756 Backplane, 1756-A10
    [0] 1756-L63 Process_Controller
    [7] 1756-CNB/A CNB_Main
      ControlNet
        1 1756-CNB/A CNB_Main
        2 1756-CNB/A CNB_Conveyor
          1756 Backplane, 1756-A10
            [0] 1756-L63 Conveyor_Controller
            [7] 1756-CNB/A CNB_Conveyor
```

Figure 2.4. Using I/O Configuration to Build a Path

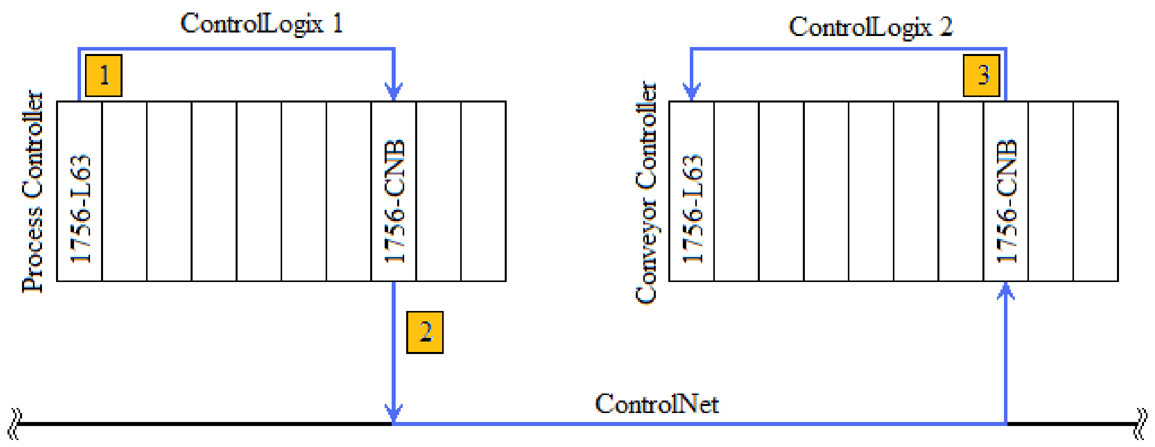
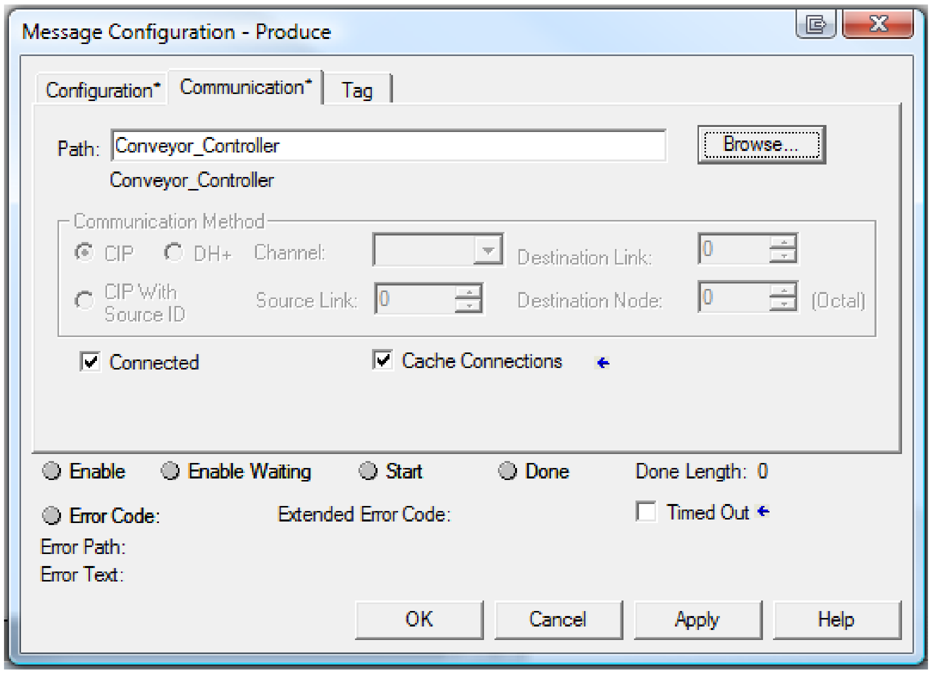


Figure 2.5. Network Path Illustration



Path: Conveyor_Controller
Conveyor_Controller

Communication Method
[✓ CIP] DH+ Channel: Destination Link: 0
CIP With Source ID Source Link: 0 Destination Node: 0 (Octal)

☑ Connected ☑ Cache Connections

Enable Enable Waiting Start Done Done Length: 0
Error Code: Extended Error Code: ☐ Timed Out
Error Path:
Error Text:

Figure 2.6. Communication Configuration Tab

The second method for configuring a path is the preferred method though it has a higher learning curve. This method uses port/address pairs to build the path for the message. Table 2.1 shows some of the possible port values. Table 2.2 shows some of the possible address values.

Table 2.1.Port Numbers of Certain Media Types [7]

Port #	Media Type
1	Backplane
2	DH485 - out channel A DHRIO - out channel A ENET - out onto Ethernet ENBT - out onto Ethernet CNB(R),CN2 - out onto ControlNet Controllers - out the serial port
3	DH485 - out channel B DHRIO - out channel B

Table 2.2. Address Format of Certain Network Types [7]

Network Type	Address Format and Value
Backplane	Backplane slot number (0-16 depending on chassis size)
Ethernet	Ethernet IP address (a.b.c.d)
ControlNet	ControlNet node address (0-99)
DeviceNet	DeviceNet node address (0-63)
DH+	Decimal equivalent of Octal DH+ node address (0-63 or 8#00 - 8#77)
RS232 Serial Port	DF1 node address (0-255, 1 for point to point)

When building paths using port/address pairs, the pairs are concatenated in the form port, address, port, address… For example: To go from the controller to a communication module in slot 7 of the backplane the path would be 1, 7. 1, the first number, to go out on the backplane and the second number, 7, to go to slot seven. The slightly more complicated path in Figure 2.4 would be 1, 7, 2, 2, 1, 0. This path is described as (1) go out on the backplane to slot 7. Then (2) go out over Control Net using port 2 to Node address 2. Then (3) go out on the backplane to slot 0.

If the path used Ethernet instead of ControlNet, then the path would be 1, 7, 2, 192.168.0.106, 1, 0. The port/address pairs would then be described as (1) go out on the backplane to slot 7. Then (2) go out over Ethernet using port 2 to IP address 192.168.0.106. Then (3) go out on the backplane to slot 0.

No matter which method is used to create the path between the two controllers the data is stored in the message structure in the same way. The full path is converted into a hexadecimal string. This portion is handled by the RSLogix software, but needs to be understood in order to develop a program which can create new paths. All paths with the exception of a path involving Ethernet are converted in the same way. First the commas are ignored, then each port number and address is converted in to a hexadecimal equivalent in the form of $00. Table 2.3 shows the conversion of 0-16. Note the difference in numbers 9, 10, 12, and 13 from standard hexadecimal.

Using Table 2.3 it can be shown that the path 1, 7, 2, 2, 1, 0 would be stored as $01$07$02$02$01$00.

Table 2.3. Number to Hexadecimal String conversion

Numbe r	HEX String	Numbe r	HEX String	Numbe r	HEX String	Numbe r	HEX String
0	$00	5	$05	10	$1	15	$0F
1	$01	6	$06	11	$0B	16	$10
2	$02	7	$07	12	$p		
3	$03	8	$08	13	$r		
4	$04	9	$t	14	$0E		

The difficulty comes in understanding what the software is doing when it sees an Ethernet IP address in the path. First of all port/address pairs not involving the IP address are converted in the same way previously mentioned. Only the port/IP pair is interpreted differently. Take the port/address pair 2, 192.168.0.106 for instance. The port will always be a 2 since that is what tells the controller to go out over Ethernet. This 2 however, is stored as $12 instead of $02. Then an Ethernet identifier is added ($0E) followed by the IP address. So the pair 2, 192.168.0.106 is stored as $12$0E192.168.0.106. The full path 1, 7, 2, 192.168.0.106, 1, 0 would be stored as $01$07$12$0E192.168.0.106$01$00.

Knowing how message paths are created, interpreted, and stored allows for the movement from the set path method used by the PLC into a more dynamic approach developed in the next chapter.

3. DYNAMIC MESSAGING

Dynamic messaging involves a single concept of real-time reconfiguration of the message path when communication fails on the current path. The method of doing this reconfiguration depends on the knowledge of the network and ranges from simple, when a network map is available, to complex, when the network is virtually unknown. There is also the stipulation that the PLCs in question be on redundant networks. This is usually not a problem since in highly automated environments networked PLCs will often be on both an automation network, usually ControlNet or Ethernet, and on the facility network, usually Ethernet. In this case it is possible to use the facility network temporarily until communication can be re-established across the automation network.

3.1. KNOWN NETWORK APPROACH

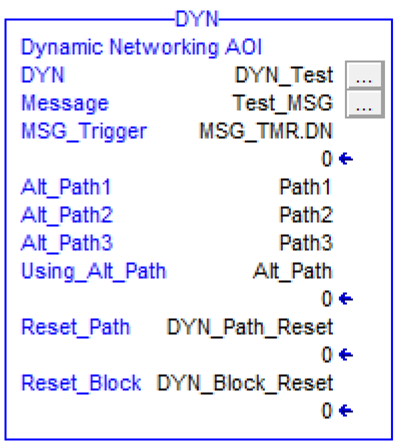
This method is the simplest approach for dynamic message path reconfiguration and relies on the process engineer having knowledge of the network. An Add-On Instruction (AOI) has been developed which allows for the definition of multiple paths to the target PLC. When the network is known and both PLCs exist on multiple common networks, the automation network and facility network for instance, then it is possible to build an alternate path for communication between the local and remote PLC.

3.1.1. AOI Operation. The operation of the AOI, shown in Figure 3.1, begins with creating a copy of the original message path to ensure it is not lost. This copy is made on the first scan of the program and a counter is incremented. This counter is used to prevent the copy instruction from executing again if the controller is switched to program mode then back to run mode. There is a block reset bit which resets this counter in case it is necessary to set a new original path.

Once a copy of the original message path has been made the message block can be executed. As long as the message is completed the AOI does nothing. If the message error bit is set then a counter is incremented causing the AOI to change the path stored in the message structure. If any of the variables associated with the counter values are empty the AOI will increment the counter in order to find a stored path. Once the AOI has incremented through the list of alternate paths the counter is reset and the AOI switches back to the original path.

To indicate there has been a fault on the given path a variable of type DINT is used to indicate the current path being used. When this variable is zero the original path is in operation. Values of 1, 2, or 3 indicate that one of the alternate paths is in operation. This variable is designed to give the process engineer a method of programming an HMI to tell process operators when there has been a communication fault so proper maintenance personnel can be notified. This way communication can be restored over the original message path with little to no downtime associated with the communication error.

Also included in the AOI is a path reset bit. This bit, when activated, restores the original message path allowing engineers and maintenance personnel to move communication back to the preferred network. Figure 3.1 shows the AOI as it would appear in Ladder Logic. Figure 3.2 shows the pseudo code for the AOI.



```
DYN
Dynamic Networking AOI
DYN                    DYN_Test  ...
Message                Test_MSG  ...
MSG_Trigger          MSG_TMR.DN
                              0 ←
Alt_Path1                 Path1
Alt_Path2                 Path2
Alt_Path3                 Path3
Using_Alt_Path         Alt_Path
                              0 ←
Reset_Path       DYN_Path_Reset
                              0 ←
Reset_Block     DYN_Block_Reset
                              0 ←
```

Figure 3.1. Dynamic Networking AOI Block

```
Step 1:  IF prevent counter not done THEN Copy current path to orig_path
         Increment prevent counter
         GOTO Step 2

Step 2:  Execute message block
         IF Error THEN GOTO Step 3 ELSE GOTO Step 9

Step 3:  Increment error counter
         IF error counter = 1 THEN Copy Alt_Path1 to Message.Path
         IF error counter = 2 THEN Copy Alt_Path2 to Message.Path
         IF error counter = 3 THEN Copy Alt_Path3 to Message.Path
         IF error counter = 4 THEN Copy Original Path to Message.Path
         IF error counter done THEN reset error counter
         GOTO Step 4

Step 4:  IF reset_path_bit THEN Copy Original Path to Message.Path
         IF reset_block_bit THEN reset prevent counter
         GOTO STEP 2
```

Figure 3.2. AOI Pseudo Code

3.1.2. AOI Implementation. To use the AOI it is placed in the ladder logic on a rung by itself directly under the rung containing the original message block. An example of the AOI used in ladder logic is shown in Figure 3.3. The original message block is then replaced by a Boolean which is to be the message trigger. Tags for each of the AOI parameters DYN, Message, MSG_Trigger, Alt_Path (1, 2, 3), Using_Alt_Path, Reset_Path, and Reset_Block are then defined. Table 3.1 explains the use of each of these parameters. To create alternate paths, full message paths are entered into the message configuration under the communication tab. Once the path is entered it is possible to copy its converted form from the message structure in the tag database into one of the alternate path variables. It is not necessary to define more than one alternate path, but the variables must exist. The AOI will simply skip over paths that have been left empty. Once the alternate paths have been defined the original path is re-entered into the message block.

Table 3.1. AOI Parameter Explanation

Parameter	Description
DYN	AOI Control Tag
Message	Original Message Control Tag
MSG_Trigger	Boolean used to Trigger Message
Alt_Path1	String used to Store Alternate Path 1
Alt_Path2	String used to Store Alternate Path 2
Alt_Path3	String used to Store Alternate Path 3
Using_Alt_Path	DINT used to Indicate Current Path (0=Original 1,2,3=Alternate)
Reset_Path	Boolean used to set Path to Original Path
Reset_Block	Boolean used to Reset AOI so a new Original Path can be set

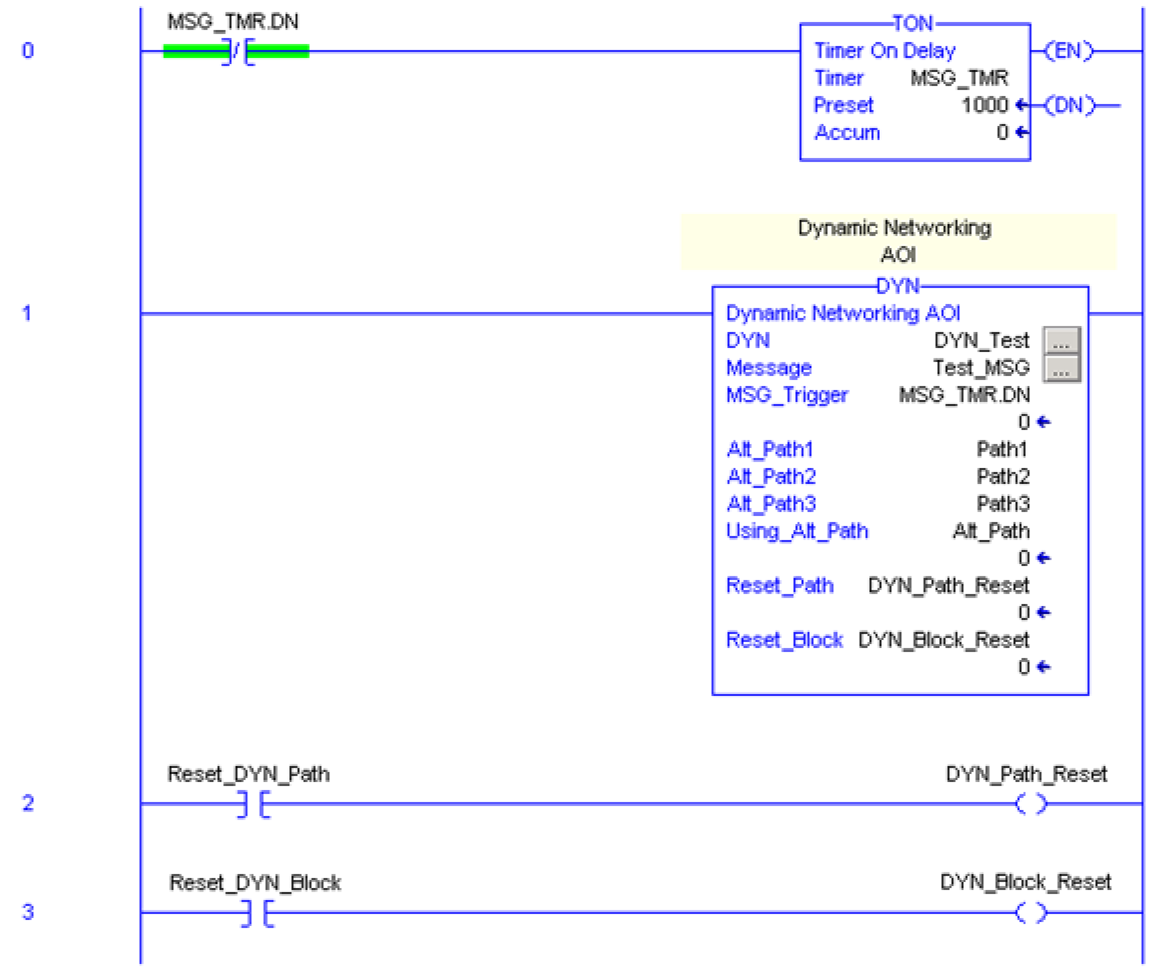


Figure 3.3. AOI Used in Ladder Logic

Once all AOI parameters have been configured, one last step remains. In the message configuration screen under the communications tab there is a box labeled Cache Connections. This check box determines how each connection is handled. When the message is executed a connection across the given path is opened and the data is transferred. If the box is checked when the message is finished the connection will remain open minimizing the time it takes to execute the message again. This creates a problem when attempting to change the path back to a previously faulted path other than

Ethernet. When the box is left unchecked then the connection is closed after the message is complete and this problem is alleviated. The increased time it takes to re-open the connection is less than 10 milliseconds. This problem is believed to stem from how the networks operate by scheduling communication. When a connection is faulted on one of these networks it must be closed and then reopened in order to reset it. For this reason the AOI forces the Cache Connections bit off prior to the execution of the message block.

3.2. FULL NETWORK SCAN APPROACH

The full network scan approach involves a complete scan of the entire automation network developed in [8] in addition to message error handling. The full network scan consists of several routines, each with its own state machine to prevent over running the maximum scan time set in the processor, as well as two AOIs. The purpose of each routine and how it operates is described in the following sections. For further explanation beyond what is described here see [8].

3.2.1. Scan Routine. [8] The Scan routine is the backbone of the full network scan and handles the transitions to the other routines. The networks paths are set here allowing the program to reach deeper and deeper into the network as the scan continues. The first step of this routine is to clear the current path, then call the Scan_BP routine. In order to use this routine on a system other than the lab where it was designed, the engineer must have some knowledge of the network (i.e. address ranges to be scanned by the other routines).

3.2.2. Scan_BP Routine. [8] The Scan_BP routine is used for scanning the backplane of a PLC. The first time this routine is run it is set to scan the local backplane looking for communication modules. Each time after the initial run this routine also checks for the target controller as well as communication modules. The communication modules are logged and the scan continues.

3.2.3. Scan_CNET Routine. [8] The Scan_CNET routine scans a ControlNet network for the range of addresses provided by the scan routine. When a module is found at an address the new backplane will be scanned and communication modules logged. When the ControlNet network has been scanned a bit is set to indicate its completion.

3.2.4. Scan_DNET Routine. [8] The Scan_DNET routine is similar to the ControlNet scanning routine but designed to operate over a DeviceNet network.

3.2.5. Scan_ENET Routine. [8] The Scan_ENET routine is also similar to the ControlNet scanning routine but is more complex in nature due to having to construct the Ethernet IP addresses. The range of addresses to be scanned must still be known and this routine only allows for the last octet to be changed.

3.2.6. Device_Module_Who Instruction. [8] Device_Module_Who is an AOI developed for the purpose of decoding raw data retrieved from a device who type message. The raw data is stored in the form of an array of SINT containing the vendor ID, product type, product code, major and minor firmware revisions, device status, serial number, and name of the module the message was sent to. The data is separated and stored in a User defined Data Type (UDT) for later use.

3.2.7. Validate_Paths Routine. [8] This routine is used to send messages across the constructed paths to validate their correctness. Once the paths have been checked, the valid paths are stored for use.

3.2.8. Find_Path Instruction. [8] The Find_Path AOI is an instruction that can be used to locate valid paths to a given module from the table of stored paths. This instruction is passed the target processor serial number, The name of the table containing the paths, and the destination for the retrieved path.

3.2.9. Message Error Handling. The error handling for this approach is done by first locating the array for the target processor in the table of stored paths. Once the array is located the error handling section waits for the error bit on the message to be set. When the error bit is set a counter is incremented and a new path is copied to the message structure. The next time the message bock is executed it will use the new path in an attempt to re-established communication. Each time communication fails the next path is used. Once all possible paths has been attempted, the counter is reset and list is gone through again.

3.2.10. Problems. There were issues involved with this approach that prevent it from being a practical solution for actual implementation. The first of these issues is time. The time it takes to complete the full network scan is extremely high. When scanning the lab network consisting of four individual networks with between 4 and 9 nodes each, the scan was still not complete after 78 hours of run time. The program was then forced out of the scanning process so the error handling could be tested. The error handling did prove successful, but this is not enough. The level of complexity in the program combined with the amount of code which needs to be adapted for every installation, severely hinders this approach from being implemented.

3.3. DIRECT PATH SCAN APPROACH

Direct path scanning is a redesigned version of the full network scan. This approach also involves several routines in order to define multiple paths to the target controller. This method utilizes the original known path to establish a connection with the remote backplane. The remote backplane is scanned and the communication modules a logged. A message is then sent to each communication module to retrieve its network address. This process eliminates the need to scan a range of addresses looking for modules. Provided an original path has been configured and operates correctly, no other knowledge of the network is needed since the appropriate network addresses are retrieved from the communication modules involved. By not sending messages to a range of addresses in an attempt to locate the communication modules, the time needed to complete the scan is reduced to less than 1 second. The time necessary to validate possible paths however, remains the same. For paths which return a value, correct or not, only need a few milliseconds, while paths which “error out” require 30 seconds each. The 30 seconds required by failed paths is due to the internal time-out of the message when a connection cannot be established. The sections below describe in further detail the operation of each routine.

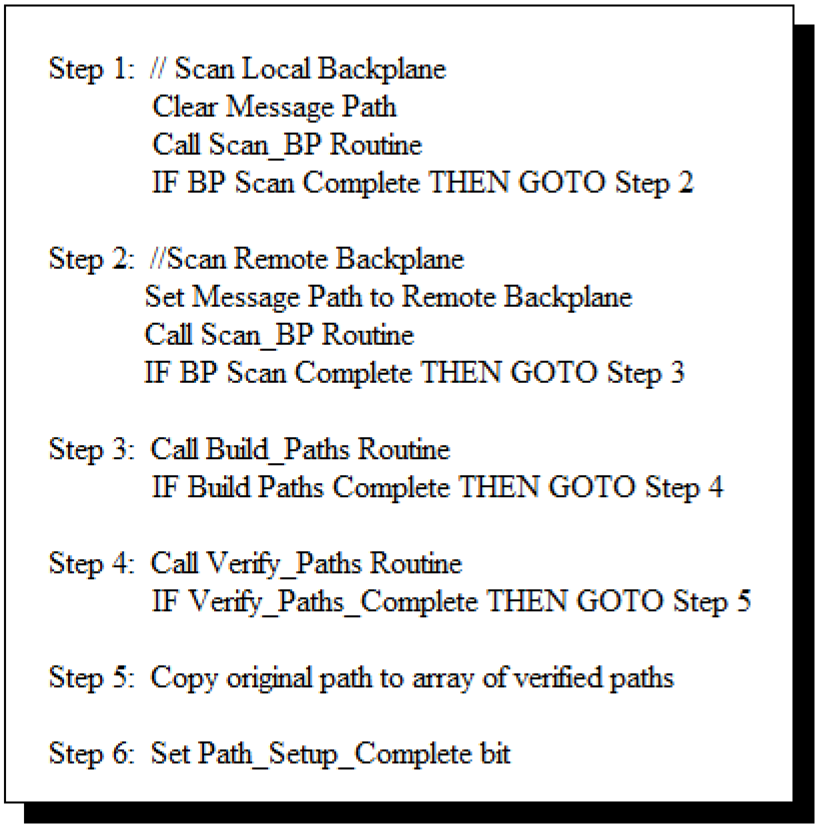
3.3.1. Control. The control routine is a short ladder routine used to switch from scanning to fault handling. While path setup is not complete the program flows through the scanning routines. Once the paths have been setup and verified the program flow switches to fault handling and no longer flows through the scanning routines. Figure 3.4 shows the pseudo code for the control routine.

```
Step 1:  IF NOT Target_Found THEN GOTO Routine Get_Target
         IF Target_Found THEN GOTO Step 2

Step 2:  IF Scan not complete THEN GOTO Scan Routine
         IF Scan complete THEN GOTO Step 3

Step 3:  GOTO Fault_Handling Routine
```

Figure 3.4 Pseudo Code for Control Routine

3.3.2. Scan. The scan routine is the main state machine for the scanning process. When the routine is first run a message is sent to the target controller to retrieve its serial number. Once this is done the main scan state machine is activated.

The first step is to scan the local backplane. This is done by clearing the path the backplane scan routine will use and then calling the backplane scan routine. When the program returns to the scan routine the main state machine is incremented. On the next program scan the path for the backplane is cleared again and set to reach the remote backplane. The backplane scan routine is called and the remote backplane is scanned. Once both backplanes have been scanned, paths between the two can be built. This is done by a call of the build paths routine. After completion of the possible message paths, the routine used to verify the paths is called to check each path. When each path has been verified they are stored in an array and the original path is concatenated to the end of the array. A path setup complete bit is then set so the control routine will know to switch to fault handling. Figure 3.5 shows the pseudo code for the scan routine.

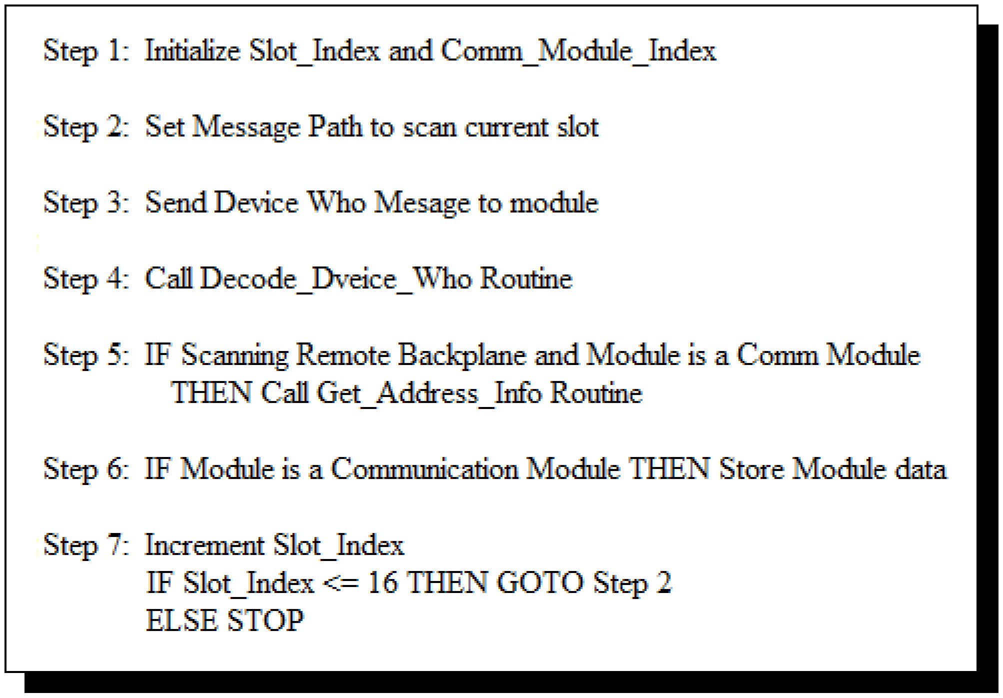


```
Step 1:  // Scan Local Backplane
         Clear Message Path
         Call Scan_BP Routine
         IF BP Scan Complete THEN GOTO Step 2

Step 2:  //Scan Remote Backplane
        Set Message Path to Remote Backplane
        Call Scan_BP Routine
        IF BP Scan Complete THEN GOTO Step 3

Step 3:  Call Build_Paths Routine
         IF Build Paths Complete THEN GOTO Step 4

Step 4:  Call Verify_Paths Routine
         IF Verify_Paths_Complete THEN GOTO Step 5

Step 5:  Copy original path to array of verified paths

Step 6:  Set Path_Setup_Complete bit
```

Figure 3.5 Pseudo Code for Scan Routine

3.3.3. Scan_BP. The backplane scan routine is designed with its own state machine to scan a ControlLogix backplane located by the path passed to the routine and catalog any communication modules found. If the routine is passed an empty path the local backplane will be scanned.

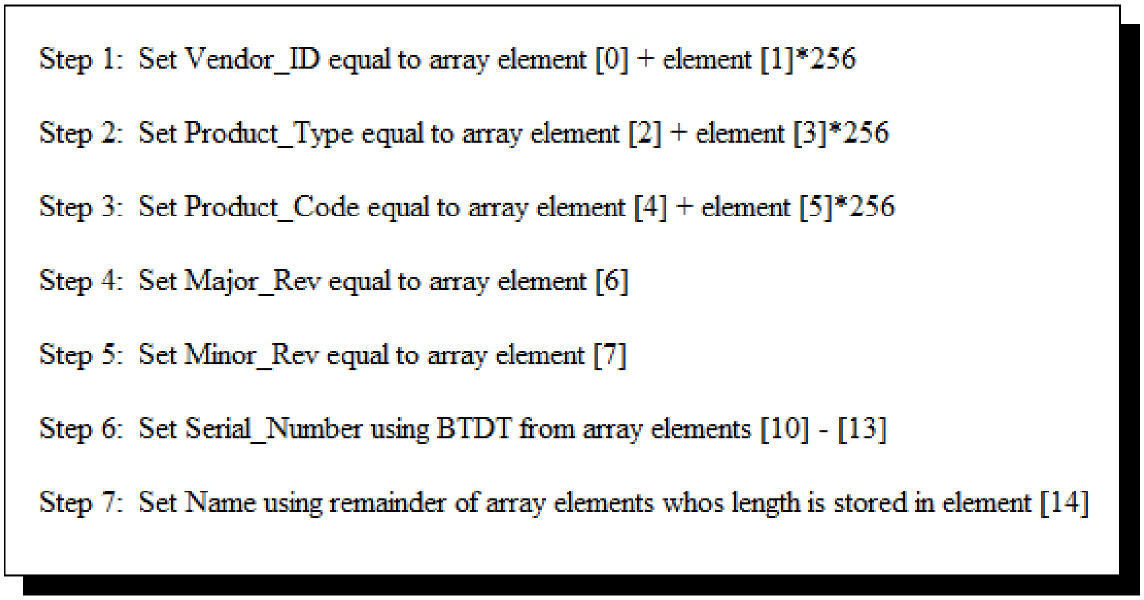
In order to scan a backplane the path to that backplane is first set and a slot counter is initialized. A "Device WHO" type message is then sent to the current slot to identify the module which may be located there. The Decode_Device_Who routine is then called to decode the data retrieved by the message. This decoded data is then checked in order to determine if the module is a communication module. If it is a communication module and the module is located on the remote backplane then another message is sent to the module to obtain its network address and the data is stored in an array for use by the Build_Paths routine. Local communication modules are stored without network addresses because it is not necessary to know them. The slot counter is then incremented. This slot counter will increment from 0 to 16 which is the maximum size of a ControlLogix chassis. Once slot 16 has been checked the program returns to the scan routine. Figure 3.6 shows the pseudo code for the Scan_BP routine.



```
Step 1:  Initialize Slot_Index and Comm_Module_Index

Step 2:  Set Message Path to scan current slot

Step 3:  Send Device Who Mesage to module

Step 4:  Call Decode_Dveice_Who Routine

Step 5:  IF Scanning Remote Backplane and Module is a Comm Module
           THEN Call Get_Address_Info Routine

Step 6:  IF Module is a Communication Module THEN Store Module data

Step 7:  Increment Slot_Index
         IF Slot_Index <= 16 THEN GOTO Step 2
         ELSE STOP
```

Figure 3.6 Pseudo Code for Scan_BP Routine

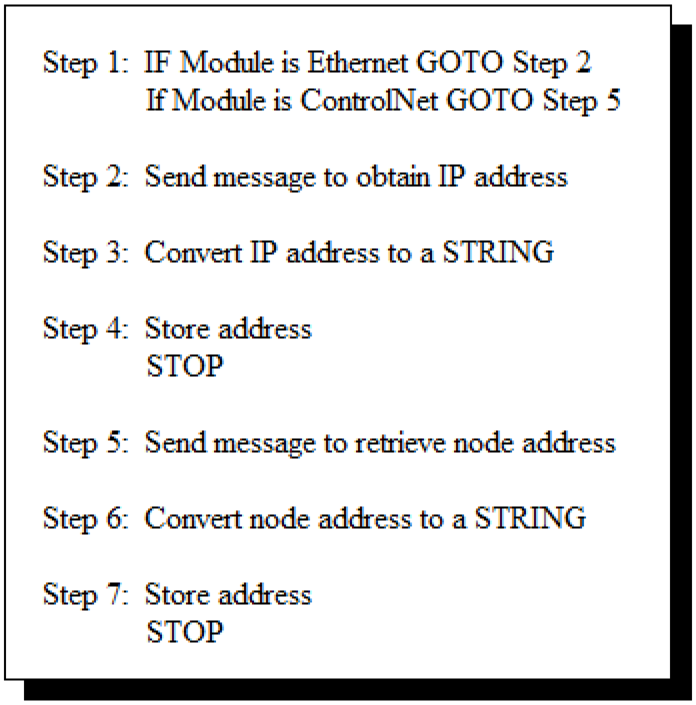
3.3.4. Decode_Device_Who. The Decode_Device_Who routine is a rework of an AOI found in [8]. This routine takes an SINT array returned from a “Device WHO” type message and separates it into parts and stores the information in a UDT of type Module. The UDT is defined as a structure which contains the vendor ID, product type, product code, major and minor firmware revisions, module serial number, module name, slot number module was found in, and the module’s network address (either node or IP). Figure 3.7 shows the pseudo code for the Decode_Device_Who routine.



```
Step 1:  Set Vendor_ID equal to array element [0] + element [1]*256
Step 2:  Set Product_Type equal to array element [2] + element [3]*256
Step 3:  Set Product_Code equal to array element [4] + element [5]*256
Step 4:  Set Major_Rev equal to array element [6]
Step 5:  Set Minor_Rev equal to array element [7]
Step 6:  Set Serial_Number using BTDT from array elements [10] - [13]
Step 7:  Set Name using remainder of array elements whos length is stored in element [14]
```

Figure 3.7 Pseudo Code for Decode_Device_Who Routine

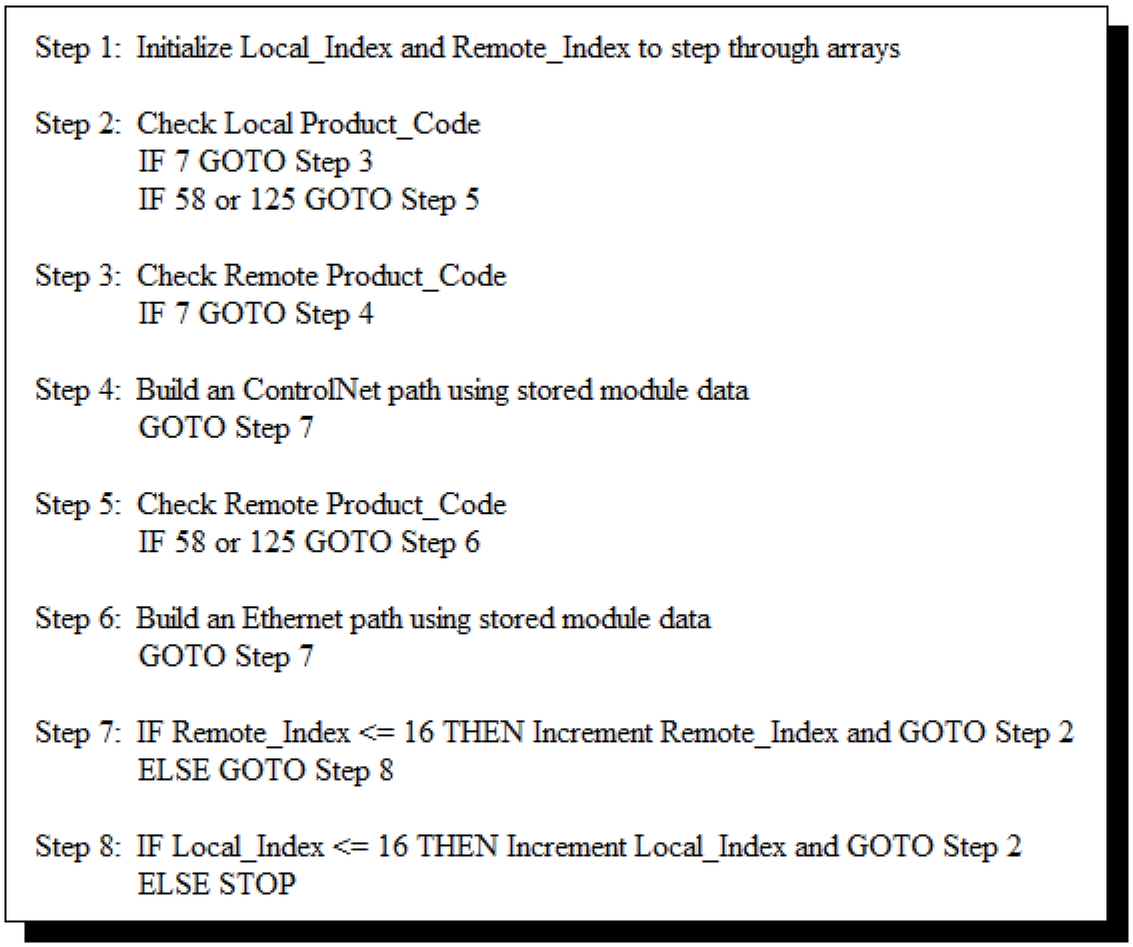
3.3.5. Get_Address_Info. The Get_Address_Info routine is used to send a message to a communication module located on the remote backplane in order to retrieve its network address. This is done by first checking the type of network the module communicates over, then sending the appropriate type of message. The raw data from the message is converted into a hexadecimal string and stored in the current module structure. Figure 3.8 shows the pseudo code for the Get_Address_Info routine.



```
Step 1:  IF Module is Ethernet GOTO Step 2
         If Module is ControlNet GOTO Step 5

Step 2:  Send message to obtain IP address

Step 3:  Convert IP address to a STRING

Step 4:  Store address
         STOP

Step 5:  Send message to retrieve node address

Step 6:  Convert node address to a STRING

Step 7:  Store address
         STOP
```

Figure 3.8 Pseudo Code for Get_Address_Info Routine

3.3.6. Build_Paths. The Build_Paths routine constructs possible message paths based on comparison of network module product codes. Two indices are initialized to step through the arrays of local and remote communication modules. When modules are found to be of the same network type, a path is built using the information stored in their structure. The path is then stored for verification and the process continues until all modules have been compared. Figure 3.9 shows the pseudo code for the Build_Paths routine.



```
Step 1:  Initialize Local_Index and Remote_Index to step through arrays

Step 2:  Check Local Product_Code
         IF 7 GOTO Step 3
         IF 58 or 125 GOTO Step 5

Step 3:  Check Remote Product_Code
         IF 7 GOTO Step 4

Step 4:  Build an ControlNet path using stored module data
         GOTO Step 7

Step 5:  Check Remote Product_Code
         IF 58 or 125 GOTO Step 6

Step 6:  Build an Ethernet path using stored module data
         GOTO Step 7

Step 7:  IF Remote_Index <= 16 THEN Increment Remote_Index and GOTO Step 2
         ELSE GOTO Step 8

Step 8:  IF Local_Index <= 16 THEN Increment Local_Index and GOTO Step 2
         ELSE STOP
```

Figure 3.9 Pseudo Code for Build_Paths Routine

3.3.7. Verify_Paths. The Verify_Paths routine sends messages across each path built by the Build_Paths routine. This is done by initializing an index used to step through the array of possible paths. Each path in turn is copied to a message structure and a message is sent. The verification message is configured to retrieve the serial number of a module. If the done bit of the message is set, the serial number retrieved is compared with the target processor serial number. If they are the same, the path is a good path and it is stored in the verified paths array. If the serial numbers are different or if the message error bit is set, the index is incremented and the next path is checked. When all paths have been checked, the program returns to the scan routine. Figure 3.10 shows the pseudo code for the Verify_Paths routine.

```
Step 1:  Initialize Path_Index and Verified_Index

Step 2:  Copy path to message structure

Step 3:  Send verify message

Step 4:  Compare returned serial number to target processor serial number
         IF = THEN Copy path to Verified array and Increment Verified_Index
         GOTO Step 5

Step 5:  IF Path_Index <=99 THEN Increment Path_Index and GOTO Step 2
         ELSE STOP
```

Figure 3.10 Pseudo Code for Verify_Paths Routine

3.3.8. Fault_Handling. The Fault_Handling routine is used to handle communication errors. When the original message is executed this routine looks for the error bit of that message. If it is set a counter is incremented and the next path stored in the verified paths array is copied to the message structure. The next time the message is executed, communication will travel across the new path attempting to re-establish communication with the remote controller. Each time an error occurs a different path will be copied to the message structure. Once the last path in the array is used the counter is reset and the process starts over at the beginning of the array. Figure 3.11 shows the pseudo code for the Fault_Handling routine.

```
IF MSG.ER THEN Copy Next Path in array to message structure

IF end of array is reached THEN reset to beginning of array

IF MSG.Path ≠ Original Path THEN Set Using_Alt_Path

IF Rest_Path THEN Copy original path to message structure
```

Figure 3.11 Pseudo Code for Fault_Handling Routine

4. CONCLUSION AND FUTURE WORK

In this work three separate methods for dynamic messaging, ranging from simple to complex, have been explored. These methods all use real-time reconfiguration of the message path in order move away from the set path method to a more dynamic communication approach.

The simplest method, the AOI, is by far the most feasible and deployable. The AOI can be imported into a project and used in place of a standard message block, which affords the user the ability to use dynamic messaging with the least amount of hassle.

As it turns out the most complex approach is the least feasible. Due to the nature of how the full network scan works, this method requires a great deal of time and effort to implement. To deploy this on any scale would require major revisions to the method used to scan the available networks. As future work it may be possible to decrease the time the network scan takes to complete as well as refine the code for easier deployment.

The third method utilizing a direct scan approach is somewhat feasible on a small scale. The scan executes very fast making the time necessary to build alternate paths almost negligible. However, in order for this to be implemented on a large scale the code would need to be adapted to handle multiple message blocks.

As new revisions of PLC programming software become available it may be possible to revisit the issues concerning producer/consumer connections. If future revisions allow for access to the communication path stored in the consumer tag it may be possible to develop methods for monitoring their communication and altering the connection path when a failure occurs. For this to become reality, the manufacturer would have to realize the need for such a capability and alter their software accordingly.

As processes and systems become more automated the need for reliability in that automation will also increase making the need for dynamic networking between controllers be of greater necessity. Also, as manufacturing and other facilities move further toward the use of redundant networks to increase efficiency and better monitor processes, the ability to easily implement a dynamic networking control scheme is strengthened. This work is just a small step in that direction.

APPENDIX A.

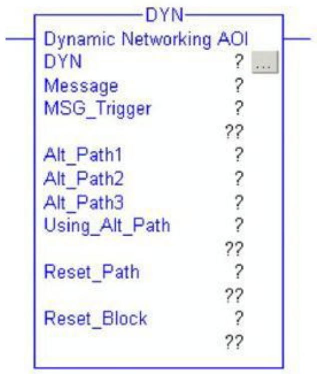
AOI DEFINITION

DYN v1.0

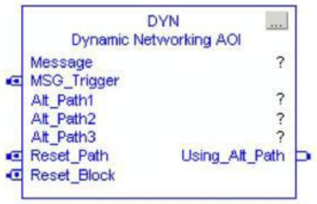
Dynamic Networking AOI

Available Languages

Relay Ladder



Function Block



Structured Text
DYN(Message, MSG_Trigger, Alt_Path1, Alt_Path2, Alt_Path3, Using_Alt_Path, Reset_Path, Reset_Block);

Parameters

Required	Name	Data Type	Usage	Description
X	DYN	DYN	InOut	Dynamic Networking AOI
	EnableIn	BOOL	Input	
	EnableOut	BOOL	Output	
X	Message	MESSAGE	InOut	
X	MSG_Trigger	BOOL	Input	
X	Alt_Path1	STRING	InOut	
X	Alt_Path2	STRING	InOut	
X	Alt_Path3	STRING	InOut	
X	Using_Alt_Path	DINT	Output	
X	Reset_Path	BOOL	Input	
X	Reset_Block	BOOL	Input	

Extended Description

DYN: AOI Control Tag□

Message: Message Control tag

MSG_Trigger: Bit used to activate message

Alt_Path1,2,3: Alternate Paths for communication
Should be entered in $01$07$12$0E131.151.52.139$01$00 format
Conversion to this format may be obtained by entering the path in a message
block and then copying from the tag under Monitor Tags
May be empty if not enough alternate paths

Using_Alt_Path: DINT used to indicate which path is being used (0 = Original Path)

Reset_Path: Bit used to set message path back to the original path

Reset_Block: Bit used to reset internal AOI logic which stores current path as original path. If the bit is set then the next transition of the controller into RUN mode will make the current path the original.
*** Caution should be used when using this bit, as the first path set as original may be lost. ***

Note1* - Faults not related to network or communication module failure are not likely to be recovered from by this AOI. For this block to function properly the original MSG should work correctly and the alternate paths entered will need to be valid communication paths.

Note2* - An EWEB module may be used in a MSG path if it is defined manually. RSLogix 5000 software will not allow anything to be hung off of it in the I/O tree, making autogeneration of a complete path not possible. To manually use an EWEB in a path, it should be treated as if it were an ENBT.

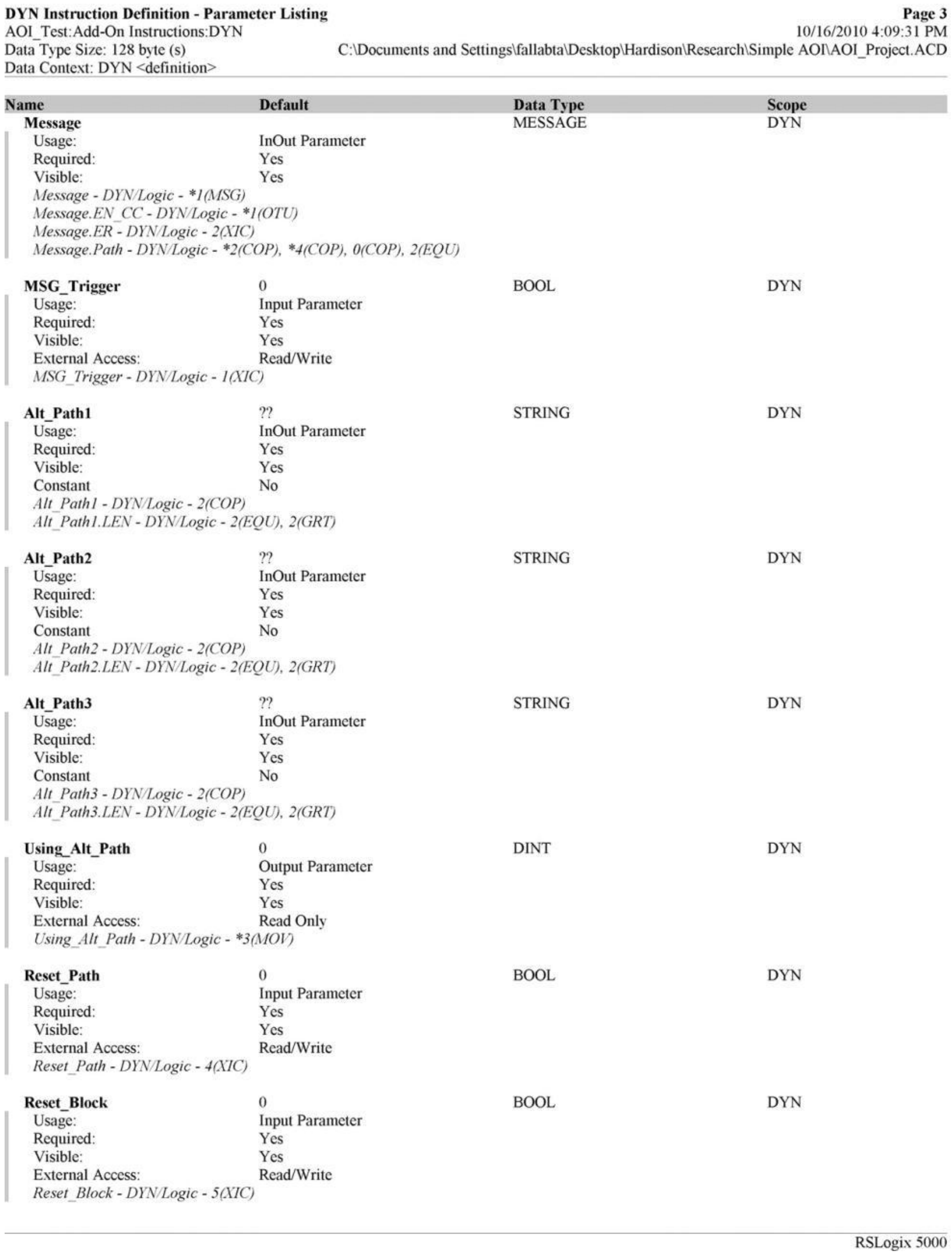
Note3* - This AOI will switch communication paths on any MSG error; therefore correct MSG operation should be varified prior to its use.

Note4* - This AOI forces "Cache Connections" off due to a conflict when thrying to reconnect to some networks.

Execution

Condition	Description
EnableIn is true	

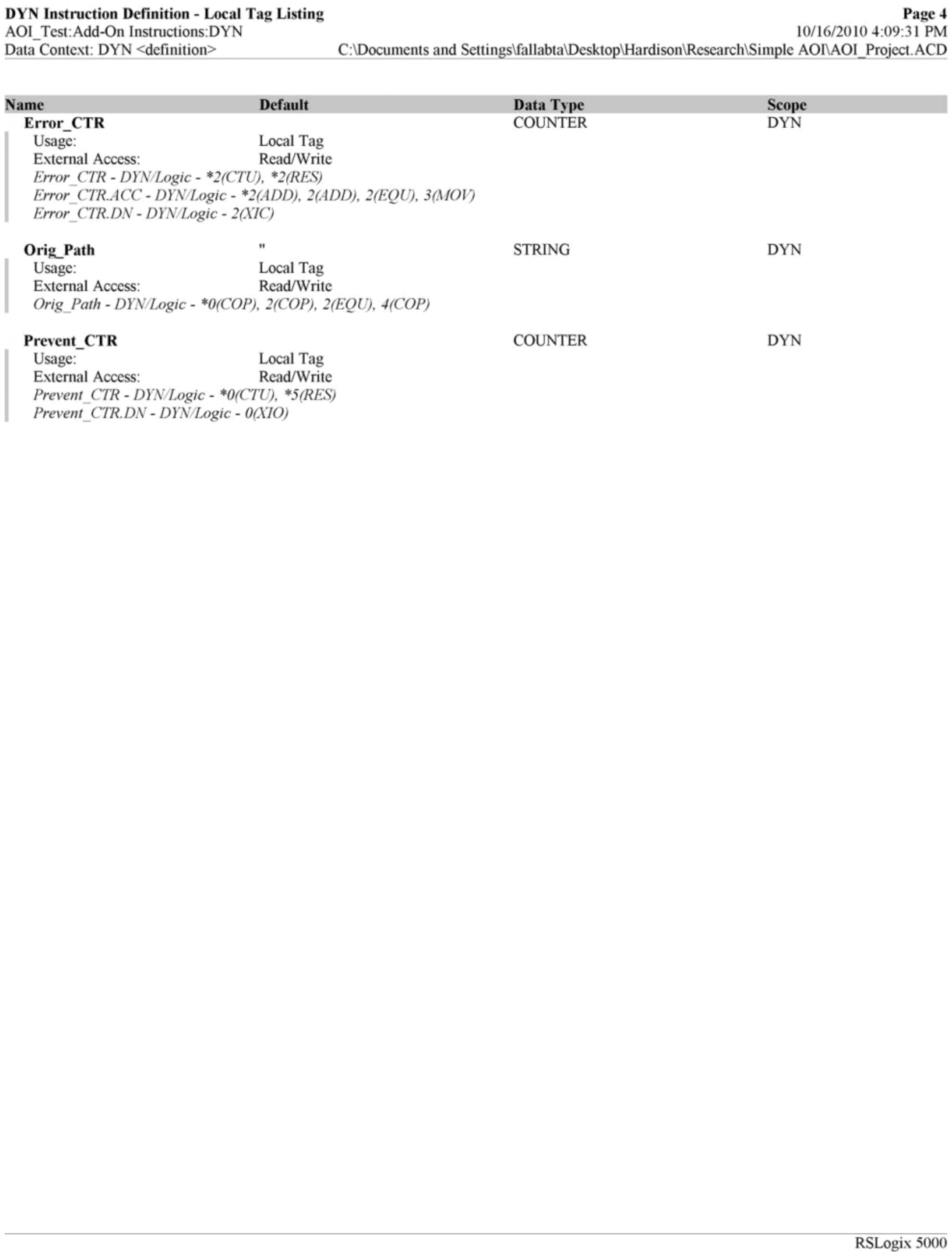
Revision v1.0 Notes



DYN Instruction Definition - Parameter Listing **Page 3**
AOI_Test:Add-On Instructions:DYN 10/16/2010 4:09:31 PM
Data Type Size: 128 byte (s) C:\Documents and Settings\fallabta\Desktop\Hardison\Research\Simple AOI\AOI_Project.ACD
Data Context: DYN <definition>

Name	Default	Data Type	Scope
Message		MESSAGE	DYN
Usage:	InOut Parameter		
Required:	Yes		
Visible:	Yes		
*Message - DYN/Logic - *1(MSG)*			
*Message.EN_CC - DYN/Logic - *1(OTU)*			
Message.ER - DYN/Logic - 2(XIC)			
*Message.Path - DYN/Logic - *2(COP), *4(COP), 0(COP), 2(EQU)*			
MSG_Trigger	0	BOOL	DYN
Usage:	Input Parameter		
Required:	Yes		
Visible:	Yes		
External Access:	Read/Write		
MSG_Trigger - DYN/Logic - 1(XIC)			
Alt_Path1	??	STRING	DYN
Usage:	InOut Parameter		
Required:	Yes		
Visible:	Yes		
Constant	No		
Alt_Path1 - DYN/Logic - 2(COP)			
Alt_Path1.LEN - DYN/Logic - 2(EQU), 2(GRT)			
Alt_Path2	??	STRING	DYN
Usage:	InOut Parameter		
Required:	Yes		
Visible:	Yes		
Constant	No		
Alt_Path2 - DYN/Logic - 2(COP)			
Alt_Path2.LEN - DYN/Logic - 2(EQU), 2(GRT)			
Alt_Path3	??	STRING	DYN
Usage:	InOut Parameter		
Required:	Yes		
Visible:	Yes		
Constant	No		
Alt_Path3 - DYN/Logic - 2(COP)			
Alt_Path3.LEN - DYN/Logic - 2(EQU), 2(GRT)			
Using_Alt_Path	0	DINT	DYN
Usage:	Output Parameter		
Required:	Yes		
Visible:	Yes		
External Access:	Read Only		
*Using_Alt_Path - DYN/Logic - *3(MOV)*			
Reset_Path	0	BOOL	DYN
Usage:	Input Parameter		
Required:	Yes		
Visible:	Yes		
External Access:	Read/Write		
Reset_Path - DYN/Logic - 4(XIC)			
Reset_Block	0	BOOL	DYN
Usage:	Input Parameter		
Required:	Yes		
Visible:	Yes		
External Access:	Read/Write		
Reset_Block - DYN/Logic - 5(XIC)			

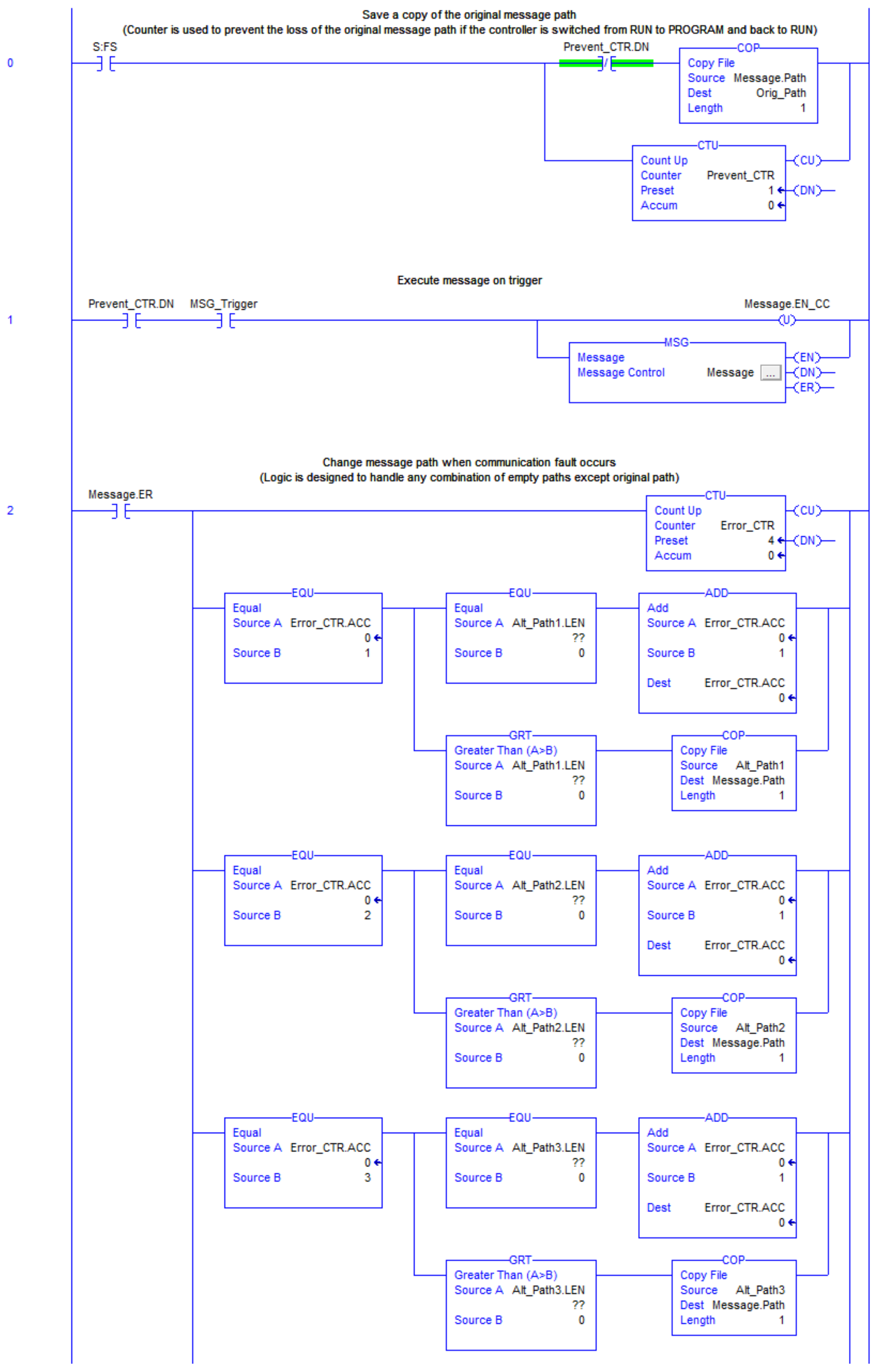
RSLogix 5000



DYN Instruction Definition - Local Tag Listing **Page 4**
AOI_Test:Add-On Instructions:DYN 10/16/2010 4:09:31 PM
Data Context: DYN <definition> C:\Documents and Settings\fallabta\Desktop\Hardison\Research\Simple AOI\AOI_Project.ACD

Name	Default	Data Type	Scope
Error_CTR		COUNTER	DYN
Usage:	Local Tag		
External Access:	Read/Write		
*Error_CTR - DYN/Logic - *2(CTU), *2(RES)*			
*Error_CTR.ACC - DYN/Logic - *2(ADD), 2(ADD), 2(EQU), 3(MOV)*			
Error_CTR.DN - DYN/Logic - 2(XIC)			
Orig_Path	''	STRING	DYN
Usage:	Local Tag		
External Access:	Read/Write		
*Orig_Path - DYN/Logic - *0(COP), 2(COP), 2(EQU), 4(COP)*			
Prevent_CTR		COUNTER	DYN
Usage:	Local Tag		
External Access:	Read/Write		
*Prevent_CTR - DYN/Logic - *0(CTU), *5(RES)*			
Prevent_CTR.DN - DYN/Logic - 0(XIO)			

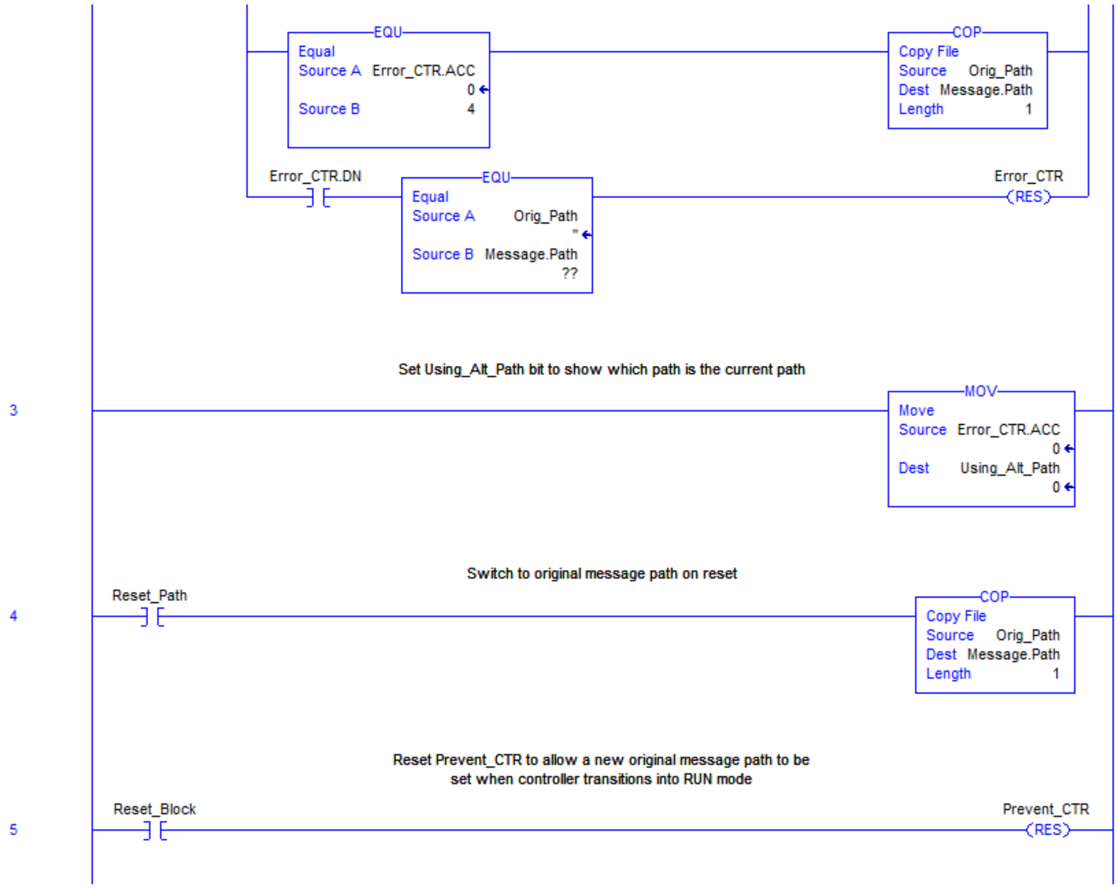
RSLogix 5000



Save a copy of the original message path
(Counter is used to prevent the loss of the original message path if the controller is switched from RUN to PROGRAM and back to RUN)

Execute message on trigger

Change message path when communication fault occurs
(Logic is designed to handle any combination of empty paths except original path)



EQU
Equal
Source A Error_CTR.ACC
0
Source B 4
COP
Copy File
Source Orig_Path
Dest Message.Path
Length 1
Error_CTR.DN
EQU
Equal
Source A Orig_Path
''
Source B Message.Path
??
Error_CTR
RES
Set Using_Alt_Path bit to show which path is the current path
3
MOV
Move
Source Error_CTR.ACC
0
Dest Using_Alt_Path
0
Switch to original message path on reset
Reset_Path
4
COP
Copy File
Source Orig_Path
Dest Message.Path
Length 1
Reset Prevent_CTR to allow a new original message path to be set when controller transitions into RUN mode
Reset_Block
5
Prevent_CTR
RES

APPENDIX B.

DIRECT PATH SCAN ROUTINE

```
// Main Scan Routine

// On First run Get Serial Number from Target Processor
IF NOT First THEN
   COP(Get_Count.Path, Orig_Path, 1);
   COP(Get_Count.Path, Get_Serial.Path, 1);

   CASE Get_Target OF
     0: // Send Message to Target Processor
        MSG(Get_Serial);

        // Goto Step 1 if Message in Enabled
        IF Get_Serial.EN THEN
          Get_Target := 1;
        END_IF;

     1: // Wait for Message to Finish or Error
        IF Get_Serial.DN THEN
          Get_Target := 2;
        ELSIF Get_Serial.ER THEN
          Get_Target := 0;
        END_IF;

     2: // Set Target Found
        Target_Found := 1;

        // Goto Step 3
        Get_Target := 3;

     3: // Set First
        First := 1;

     END_CASE;
END_IF;
```

```
IF Target_Found THEN
    // Main Scan State Machine
    CASE Scan_Step OF
      0: // Scan Local Backplane
         // Set Path to Local Backplane
         // (Scan_BP Will Scan the Local Backplane if MSG Path is empty)
         DELETE(Path_BP,82,1,Path_BP);

         // Jump to Scan_BP Routine
         JSR(Scan_BP);

         // Goto Step 1 if Local Backplane Scan is Complete
         IF BP_Scan_Step = 8 THEN
            BP_Scan_Step := 0;
            Scan_Step := 1;
         END_IF;

      1: // Scan Remote Backplane
         // Set Path to Remote Backplane
         Start_Pos := Get_Count.Path.Len-1;
         DELETE(Get_Count.Path,82,Start_Pos,Path_BP);

         // Jump to Scan_BP Routine
         JSR(Scan_BP);

         // Goto Step 2 if Remote Backplane Scan is Complete
         IF BP_Scan_Step = 8 THEN
            Scan_Step := 2;
         END_IF;

      2: // Build Communication Paths
         JSR(Build_Paths);
```

```
            // Goto Step 3 if Build Paths is Complete
            IF Build_Path_Step = 8 THEN
                Scan_Step := 3;
            END_IF;

        3: // Varify Paths
            JSR(Verify_Paths);

            // Goto Step 4 When Path Verification is Complete
            IF Verify_Step = 6 THEN
                Scan_Step := 4;
            END_IF;

        4: // Copy Original Path in to Verified Paths
            COP(Orig_Path, Verified_Paths[V_Path_Index], 1);

            // Goto Step 5
            Scan_Step := 5;

        5: // Set Path_Setup_Complete
            Path_setup_Complete := 1;

    END_CASE;
END_IF;
```

APPENDIX C.

DIRECT PATH SCAN_BP ROUTINE

```
// Scan Backplane Routine

// Scan Backplane State Machine
Case BP_Scan_Step OF
   0: // Set Slot and Index to 0
     Slot := 0;
     Index := 0;

     // Goto Step 1
     BP_Scan_Step := 1;

   1: // Append Message Path to Scan Slot
     DELETE(Mod_Who.Path,82,1,Mod_Who.Path);
     INSERT(Mod_Who.Path,Path_BP,1,Mod_Who.Path);

     // Append Message Path to go out on Backplane to Slot
     Mod_Who.Path.Data[Mod_Who.path.len] := 1;
     Mod_Who.Path.Data[Mod_Who.path.len+1] := Slot;
     Mod_Who.path.len := Mod_Who.path.len + 2; // Two is added Here to Allow Room for Slot
        Number

     // Goto Step 2
     BP_Scan_Step := 2;

   2: // Send Device Who Message to Module
     MSG(Mod_Who);

     // Goto next step if message is enabled
     IF Mod_Who.EN THEN
        BP_Scan_Step := 3;
     END_IF;

   3: // Wait for Message to Finish or Error
     IF Mod_Who.DN THEN
```

```
        BP_Scan_Step := 4;
    ELSIF Mod_Who.ER THEN
        BP_Scan_Step := 7;
    END_IF;

4: // Decode Retrieved Data
    JSR(Decode_Device_Who);

    // Goto Step 5
    BP_Scan_Step := 5;

5: // Get Address Information if Remote Module is a Communication Module
    IF Current_Module.Product_Type = 12 AND Scan_Step = 1 THEN
        JSR(Get_Address_Info);

        IF Set AND (Add_Step = 9) THEN
            // Reset Set bit and Add_Step Then Goto Step 6
            Set := 0;
            Add_Step := 0;
            BP_Scan_Step := 6;
        END_IF;
    ELSE
        // Goto Step 6
        BP_Scan_Step := 6;
    END_IF;

6: // Check if Communication Module and Store Information if so
    // If Type is 12 and Code is 7,17,58,125
    IF Current_Module.Product_Type = 12 THEN
        CASE Current_Module.Product_Code OF
            7,17,58,125:   // Store Data and increment index to next storage spot
                IF Index < 17 THEN
                    IF Scan_Step = 0 THEN
                    COP(Current_Module, Scan_Result.Local[Index],1);
                    Scan_Result.Local[Index].Slot := Slot;
                    Index := Index +1;

                  ELSIF Scan_Step = 1 THEN

                    COP(Current_Module, Scan_Result.Remote[Index],1);
                    Scan_Result.Remote[Index].Slot := Slot;
                    Index := Index +1;

                  END_IF;
                END_IF;
              END_CASE;
        END_IF;
        // Goto Step 7
        BP_Scan_Step := 7;

      7: // Increment slot counter 'Slot'
        Slot := Slot + 1;

        // Check 'Slot'
        IF Slot <= 16 THEN
          BP_Scan_Step := 1;
        ELSE
          BP_Scan_Step := 8;
        END_IF;

      8: // DONE scanning Backplane; Stop

ELSE
        // If Not in Steps 0-6, Goto Step 1
        BP_Scan_Step := 1;

END_CASE;
```

APPENDIX D.

DIRECT PATH DECODE_DEVICE_WHO ROUTINE

```
// Decode Device Who Message Routine

// Extract Vendor ID
Current_Module.Vendor_ID := 256*Mod_Who_Data[1] + Mod_Who_Data[0];

// Extract Product Type
Current_Module.Product_Type := 256*Mod_Who_Data[3] + Mod_Who_Data[2];

// Extract Product Code
Current_Module.Product_Code := 256*Mod_Who_Data[5] + Mod_Who_Data[4];

// Extract Major Revision
Current_Module.Major_Rev := Mod_Who_Data[6];

// Extract Minor Revision
Current_Module.Minor_Rev := Mod_Who_Data[7];

// Extract Serial Number (Use BTDT so sign is not considered)
Serial_BTDT.Source    := Mod_Who_Data[10];
Serial_BTDT.SourceBit := 0;
Serial_BTDT.Length    := 8;
Serial_BTDT.Target    := Current_Module.Serial_Num;
Serial_BTDT.DestBit   := 0;
BTDT(Serial_BTDT);
Current_Module.Serial_Num := Serial_BTDT.Dest;
Serial_BTDT.Source    := Mod_Who_Data[11];
Serial_BTDT.Target    := Current_Module.Serial_Num;
Serial_BTDT.DestBit   := 8;
BTDT(Serial_BTDT);
Current_Module.Serial_Num := Serial_BTDT.Dest;
Serial_BTDT.Source    := Mod_Who_Data[12];
Serial_BTDT.Target    := Current_Module.Serial_Num;
Serial_BTDT.DestBit   := 16;
BTDT(Serial_BTDT);
```

```
Current_Module.Serial_Num := Serial_BTDT.Dest;
Serial_BTDT.Source    := Mod_Who_Data[13];
Serial_BTDT.Target    := Current_Module.Serial_Num;
Serial_BTDT.DestBit   := 24;
BTDT(Serial_BTDT);
Current_Module.Serial_Num := Serial_BTDT.Dest;

// Extract Name
Current_Module.Name.LEN := Mod_Who_Data[14];
FOR i:= 1 TO Mod_Who_Data[14] DO
Current_Module.Name.DATA[i-1] := Mod_Who_Data[i+14];
END_FOR;
```

APPENDIX E.

DIRECT PATH GET_ADDRESS_INFO ROUTINE

```
// Get Address Information Routine

IF NOT Set THEN
    // Set Case Step to Zero if Getting IP Address
    IF (Current_Module.Product_Code = 58) OR (Current_Module.Product_Code = 125) THEN

        // Set Path for Get IP Message
        COP(Mod_Who.Path, Get_IP.Path,1);

        // Set to Step 1
        Add_Step := 1;
        Set := 1;

    END_IF;

    // Set Case Step to Zero if Getting Node Address
    IF Current_Module.Product_Code = 7 THEN

    // Set Path for Get Node Message
    COP(Mod_Who.Path, Get_Node.Path, 1);

    // Set Step to 5
    Add_Step := 5;
    Set := 1;

    END_IF;
END_IF;

// Address Information State Machine
CASE Add_Step OF

    1: // Send Message to Get IP Address
        MSG(Get_IP);
```

```
   // Goto Next Step if Message is Enabled
   IF Get_IP.EN THEN
       Add_Step := 2;
   END_IF;

2: // Wait for Message to Finish or Error
   IF Get_IP.DN THEN
       Add_Step := 3;
   ELSIF Get_IP.ER THEN
       Add_Step := 9;
   END_IF;

3: // Convert IP Address to STRING

   // Extract IP Address Parts to DINT (Use BTDT so sign is not considered)
   IP_BTDT.Source    := Current_IP_Info[0];
   IP_BTDT.SourceBit := 0;
   IP_BTDT.Length    := 8;
   IP_BTDT.Target    := Current_IP_Temp_DINT[0];
   IP_BTDT.DestBit   := 0;
   BTDT(IP_BTDT);
   Current_IP_Temp_DINT[0] := IP_BTDT.Dest;
   IP_BTDT.Source    := Current_IP_Info[1];
   IP_BTDT.Target    := Current_IP_Temp_DINT[1];
   BTDT(IP_BTDT);
   Current_IP_Temp_DINT[1] := IP_BTDT.Dest;
   IP_BTDT.Source    := Current_IP_Info[2];
   IP_BTDT.Target    := Current_IP_Temp_DINT[2];
   BTDT(IP_BTDT);
   Current_IP_Temp_DINT[2] := IP_BTDT.Dest;
   IP_BTDT.Source    := Current_IP_Info[3];
   IP_BTDT.Target    := Current_IP_Temp_DINT[3];
   BTDT(IP_BTDT);
   Current_IP_Temp_DINT[3] := IP_BTDT.Dest;
   // Convert IP Address Parts to ASCII
   DTOS(Current_IP_Temp_DINT[3], Current_IP_Temp_ASCII[0]);
   DTOS(Current_IP_Temp_DINT[2], Current_IP_Temp_ASCII[1]);
   DTOS(Current_IP_Temp_DINT[1], Current_IP_Temp_ASCII[2]);
   DTOS(Current_IP_Temp_DINT[0], Current_IP_Temp_ASCII[3]);

   // Concatinate IP Address Parts
   INSERT(Current_IP_Add, Current_IP_Temp_ASCII[0], 1,Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;
   INSERT(Current_IP_Add, Period, Position, Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;
   INSERT(Current_IP_Add, Current_IP_Temp_ASCII[1], Position, Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;
   INSERT(Current_IP_Add, Period, Position, Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;
   INSERT(Current_IP_Add, Current_IP_Temp_ASCII[2], Position, Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;
   INSERT(Current_IP_Add, Period, Position, Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;
   INSERT(Current_IP_Add, Current_IP_Temp_ASCII[3], Position, Current_IP_Add);
   Position := Current_IP_Add.LEN + 1;

   // Goto Step 4
   Add_Step := 4;

4: // Store Address in Current.Address
   COP(Current_IP_Add, Current_Module.Address, 1);

   // Goto Step 9
   Add_Step := 9;

5: // Send Message to Get Node Address
   MSG(Get_Node);
```

```
  // Goto Next Step if Message is Enabled
  IF Get_Node.EN THEN
     Add_Step := 6;
  END_IF;

6: // Wait for Message to Finish or Error
  IF Get_Node.DN THEN
     Add_Step := 7;
  ELSIF Get_Node.ER THEN
     Add_Step := 9;
  END_IF;

7: // Convert Node Address to STRING
  //Copy First 8 Bits of DINT
  Current_Node_Temp.0 := Current_Node_Info.0;
  Current_Node_Temp.1 := Current_Node_Info.1;
  Current_Node_Temp.2 := Current_Node_Info.2;
  Current_Node_Temp.3 := Current_Node_Info.3;
  Current_Node_Temp.4 := Current_Node_Info.4;
  Current_Node_Temp.5 := Current_Node_Info.5;
  Current_Node_Temp.6 := Current_Node_Info.6;
  Current_Node_Temp.7 := Current_Node_Info.7;

  // Store SINT as STRING
  Current_Node_Add.Data[0].0 := Current_Node_Temp.0;
  Current_Node_Add.Data[0].1 := Current_Node_Temp.1;
  Current_Node_Add.Data[0].2 := Current_Node_Temp.2;
  Current_Node_Add.Data[0].3 := Current_Node_Temp.3;
  Current_Node_Add.Data[0].4 := Current_Node_Temp.4;
  Current_Node_Add.Data[0].5 := Current_Node_Temp.5;
  Current_Node_Add.Data[0].6 := Current_Node_Temp.6;
  Current_Node_Add.Data[0].7 := Current_Node_Temp.7;
  Current_Node_Add.LEN := 1;
```

```
        // Goto Step 8
        Add_Step := 8;

     8: // Store Address in Current.Address
        COP(Current_Node_Add, Current_Module.Address, 1);

        // Goto Step 9
        Add_Step := 9;

     9: // Stop, Set Done Bit, and Reset Set bit

END_CASE;
```

APPENDIX F.

DIRECT PATH BUILD_PATHS ROUTINE

```
// Build Paths Routine

// Build Path State Machine
CASE Build_Path_Step OF

   0: // Select First Local and Remote Communication Modules and Set Path Index to 0
     Local_Mod_Index := 0;
     Remote_Mod_Index := 0;
     Path_Index := 0;

     // Goto Step 1
     Build_Path_Step := 1;

   1: // Check Local Product Code
     IF Scan_Result.Local[Local_Mod_Index].Product_Code = 7 THEN

        // Goto Step 2
        Build_Path_Step := 2;

     ELSIF (Scan_Result.Local[Local_Mod_Index].Product_Code = 58) OR
              (Scan_Result.Local[Local_Mod_Index].Product_Code = 125) THEN

        // Goto Step 4
        Build_Path_Step := 4;

     ELSE

        // Goto Step 7
        Build_Path_Step := 7;

     END_IF;

   2: // Check Remote Product Code
     IF Scan_Result.Remote[Remote_Mod_Index].Product_Code = 7 THEN
```

```
        // Goto Step 3
        Build_Path_Step := 3;

    ELSE

        // Goto Step 6
        Build_Path_Step := 6;

    END_IF;

3: // Build Path for Control Net
    // Delete Contents of Path_Temp
    DELETE(Path_Temp,82,1,Path_Temp);

    // Go out on Backplane
    Path_Temp.Data[0].0 := 1;
    Path_Temp.LEN := Path_Temp.LEN + 1;

    // To Local Slot
    Path_Temp.Data[1].0 := Scan_Result.Local[Local_Mod_Index].Slot.0;
    Path_Temp.Data[1].1 := Scan_Result.Local[Local_Mod_Index].Slot.1;
    Path_Temp.Data[1].2 := Scan_Result.Local[Local_Mod_Index].Slot.2;
    Path_Temp.Data[1].3 := Scan_Result.Local[Local_Mod_Index].Slot.3;
    Path_Temp.Data[1].4 := Scan_Result.Local[Local_Mod_Index].Slot.4;
    Path_Temp.Data[1].5 := Scan_Result.Local[Local_Mod_Index].Slot.5;
    Path_Temp.Data[1].6 := Scan_Result.Local[Local_Mod_Index].Slot.6;
    Path_Temp.Data[1].7 := Scan_Result.Local[Local_Mod_Index].Slot.7;
    Path_Temp.Len := Path_Temp.LEN + 1;

    // Go Out Over Control Net
    Path_Temp.Data[2].1 := 1;
    Path_Temp.Len := Path_Temp.LEN + 1;

    // To Remote Node Address
```

```
  CONCAT(Path_Temp, Scan_Result.Remote[Remote_Mod_Index].Address, Path_Temp);

   // Go Out on Backplane
  Path_Temp.Data[Path_Temp.LEN].0 := 1;
  Path_Temp.LEN := Path_Temp.LEN + 1;

  // To Remote Controller
  Position :=Get_Count.Path.LEN - 1;
  Path_Temp.Data[Path_Temp.LEN].0 := Get_Count.Path.Data[Position].0;
  Path_Temp.Data[Path_Temp.LEN].1 := Get_Count.Path.Data[Position].1;
  Path_Temp.Data[Path_Temp.LEN].2 := Get_Count.Path.Data[Position].2;
  Path_Temp.Data[Path_Temp.LEN].3 := Get_Count.Path.Data[Position].3;
  Path_Temp.Data[Path_Temp.LEN].4 := Get_Count.Path.Data[Position].4;
  Path_Temp.Data[Path_Temp.LEN].5 := Get_Count.Path.Data[Position].5;
  Path_Temp.Data[Path_Temp.LEN].6 := Get_Count.Path.Data[Position].6;
  Path_Temp.Data[Path_Temp.LEN].7 := Get_Count.Path.Data[Position].7;
  Path_Temp.LEN := Path_Temp.LEN + 1;

  // Copy Finished Path to Storage
  COP(Path_Temp, Unverified_Paths[Path_Index], 1);

  // Increment Path Index
  Path_Index := Path_Index + 1;

  // Goto Step 6
  Build_Path_Step := 6;

4: // Check Remote Product Code
  IF (Scan_Result.Remote[Remote_Mod_Index].Product_Code = 58) OR
     (Scan_Result.Remote[Remote_Mod_Index].Product_Code = 125) THEN

     // Goto Step 5
     Build_Path_Step := 5;
  ELSE
     // Goto Step 6
     Build_Path_Step := 6;

  END_IF;

5: // Build Path for Ethernet
  // 1, Local_Slot_Num, $12, $0E131.151.52.140, 1, Remote_Controller

  // Delete Contents of Path_Temp
  DELETE(Path_Temp,82,1,Path_Temp);

  // Go out on Backplane
  Path_Temp.Data[0].0 := 1;
  Path_Temp.LEN := Path_Temp.LEN + 1;

  // To Local Slot
  Path_Temp.Data[1].0 := Scan_Result.Local[Local_Mod_Index].Slot.0;
  Path_Temp.Data[1].1 := Scan_Result.Local[Local_Mod_Index].Slot.1;
  Path_Temp.Data[1].2 := Scan_Result.Local[Local_Mod_Index].Slot.2;
  Path_Temp.Data[1].3 := Scan_Result.Local[Local_Mod_Index].Slot.3;
  Path_Temp.Data[1].4 := Scan_Result.Local[Local_Mod_Index].Slot.4;
  Path_Temp.Data[1].5 := Scan_Result.Local[Local_Mod_Index].Slot.5;
  Path_Temp.Data[1].6 := Scan_Result.Local[Local_Mod_Index].Slot.6;
  Path_Temp.Data[1].7 := Scan_Result.Local[Local_Mod_Index].Slot.7;
  Path_Temp.Len := Path_Temp.LEN + 1;

  // Go Out Over Ethernet Net
  Path_Temp.Data[2].1 := 1;
  Path_Temp.Data[2].4 := 1;
  Path_Temp.Len := Path_Temp.LEN + 1;

  // Add Ethernet Designation
  Path_Temp.Data[3].0 := 0;
```

```
    Path_Temp.Data[3].1 := 1;
    Path_Temp.Data[3].2 := 1;
    Path_Temp.Data[3].3 := 1;
    Path_Temp.Len := Path_Temp.LEN + 1;

    // To Remote Node Address
    CONCAT(Path_Temp, Scan_Result.Remote[Remote_Mod_Index].Address, Path_Temp);

    // Go Out on Backplane
    Path_Temp.Data[Path_Temp.LEN].0 := 1;
    Path_Temp.LEN := Path_Temp.LEN + 1;

    // To Remote Controller
    Position :=Get_Count.Path.LEN - 1;
    Path_Temp.Data[Path_Temp.LEN].0 := Get_Count.Path.Data[Position].0;
    Path_Temp.Data[Path_Temp.LEN].1 := Get_Count.Path.Data[Position].1;
    Path_Temp.Data[Path_Temp.LEN].2 := Get_Count.Path.Data[Position].2;
    Path_Temp.Data[Path_Temp.LEN].3 := Get_Count.Path.Data[Position].3;
    Path_Temp.Data[Path_Temp.LEN].4 := Get_Count.Path.Data[Position].4;
    Path_Temp.Data[Path_Temp.LEN].5 := Get_Count.Path.Data[Position].5;
    Path_Temp.Data[Path_Temp.LEN].6 := Get_Count.Path.Data[Position].6;
    Path_Temp.Data[Path_Temp.LEN].7 := Get_Count.Path.Data[Position].7;
    Path_Temp.LEN := Path_Temp.LEN + 1;

    // Copy Finished Path to Storage
    COP(Path_Temp, Unverified_Paths[Path_Index], 1);

    // Increment Path Index
    Path_Index := Path_Index + 1;

    // Goto Step 6
    Build_Path_Step := 6;

6: // Increment Remote Index
```

```
        Remote_Mod_Index := Remote_Mod_Index + 1;

        IF Remote_Mod_Index<= 16 THEN

            // Goto Step 1
            Build_Path_Step := 1;

        ELSE

            // Goto Step 7
            Build_Path_Step := 7;

        END_IF;

    7: // Increment Local Index and Reset Remote Index

        Local_Mod_Index := Local_Mod_Index + 1;
        Remote_Mod_Index := 0;

        IF Local_Mod_Index<= 16 THEN

             // Goto Step 1
             Build_Path_Step := 1;

        ELSE

            // Goto Step 8
            Build_Path_Step := 8;

        END_IF;

    8: // Stop

END_CASE;
```

APPENDIX G.

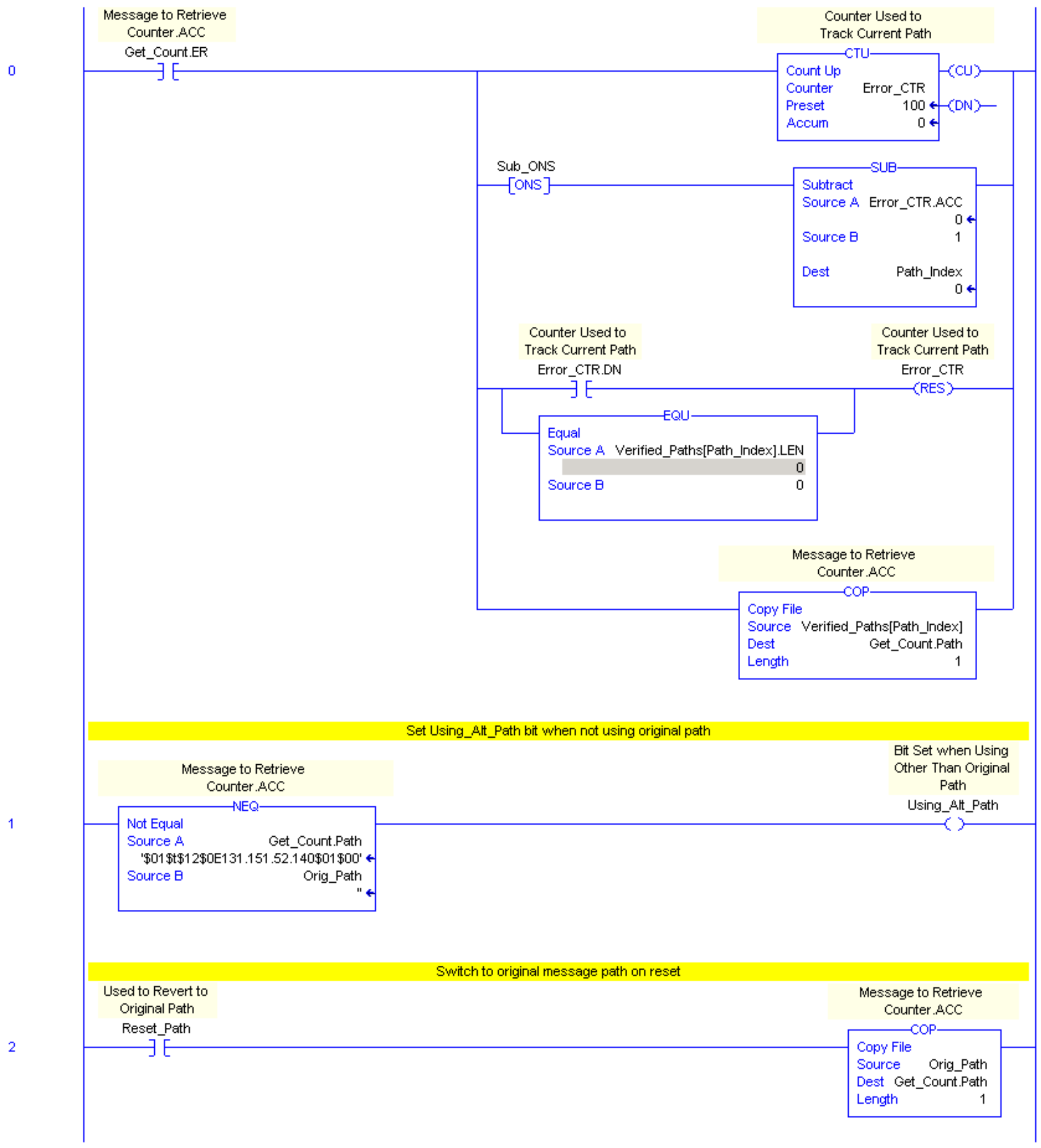
DIRECT PATH VERIFY_PATHS ROUTINE

```
// Verify Paths Routine

// Verify Path State Machine
CASE Verify_Step OF

   0: // Set Path Indices to 0 and Check to Ensure Target has been Identified
     Path_Index := 0;
     V_Path_Index := 0;

     IF Target_Serial<> 0 THEN

        // Goto Step 1
        Verify_Step := 1;

     END_IF;

   1: // Copy Message Path and Send Message
     COP(Unverified_Paths[Path_Index], Verify_MSG.Path, 1);

     // Goto Step 2
     Verify_Step := 2;

   2: // Send Message
     MSG(Verify_MSG);

     // Goto Next Step if Message is Enabled
     IF Verify_MSG.EN THEN
         Verify_Step := 3;
     END_IF;

   3: // Wait for Message to Finish or Error
     IF Verify_MSG.DN THEN
        Verify_Step := 4;
     ELSIF Verify_MSG.ER THEN
```

```
            Verify_Step := 5;
        END_IF;

    4: // Compare Serial Numbers and Store Path if Good
        IF Verify_Serial = Target_Serial THEN

            COP(Unverified_Paths[Path_Index], Verified_Paths[V_Path_Index], 1);
            V_Path_Index := V_Path_Index + 1;

        END_IF;

        // Goto Step 5
        Verify_Step := 5;

    5: // Increment Path Index
        Path_Index := Path_Index + 1;

        IF (Path_Index<= 99) AND (Unverified_Paths[Path_Index].LEN > 0) THEN

            // Goto Step 1
            Verify_Step := 1;

        ELSE

            // Goto Step 6
            Verify_Step := 6;

        END_IF;

    6: // Stop

END_CASE;
```

APPENDIX H.

DIRECT PATH FAULT_HANDLING ROUTINE


0
Message to Retrieve Counter.ACC
Get_Count.ER
Counter Used to Track Current Path
CTU
Count Up
Counter Error_CTR
Preset 100
Accum 0
CU
DN
Sub_ONS
ONS
SUB
Subtract
Source A Error_CTR.ACC
0
Source B 1
Dest Path_Index
0
Counter Used to Track Current Path
Error_CTR.DN
Counter Used to Track Current Path
Error_CTR
RES
EQU
Equal
Source A Verified_Paths[Path_Index].LEN
0
Source B 0
Message to Retrieve Counter.ACC
COP
Copy File
Source Verified_Paths[Path_Index]
Dest Get_Count.Path
Length 1
Set Using_Alt_Path bit when not using original path
1
Message to Retrieve Counter.ACC
NEQ
Not Equal
Source A Get_Count.Path
'$01$t$12$0E131.151.52.140$01$00'
Source B Orig_Path
''
Bit Set when Using Other Than Original Path
Using_Alt_Path
Switch to original message path on reset
2
Used to Revert to Original Path
Reset_Path
Message to Retrieve Counter.ACC
COP
Copy File
Source Orig_Path
Dest Get_Count.Path
Length 1

BIBLIOGRAPHY

[1] M.R. Muller, et al., “Industrial Productivity Training Manual”, Annual IAC Director’s Meeting, 1996, Rutgers, the State University of New Jersey

[2] K. T. Erickson, Programmable Logic Controllers: An Emphasis on Design and Application, Missouri: Dogwood Valley Press LLC, 2005, pp. 1231-1238.

[3] “Obtaining a Logix Processors Firmware Level Using a CIP Generic Message,” Allen-Bradley Knowledge Base Document, no. 23386, October 2008.

[4] “How Read SN From The Module?,”Allen-Bradley Knowledge Base Document, no. 52870, October 2008.

[5] “Obtaining CNB node number,”Allen-Bradley Knowledge Base Document, no. 25592, August 2009.

[6] “Accessing IP Information on the 1756-ENBT 1788ENBT Module via Ladder Logic,” Allen-Bradley Knowledge Base Document, no. 26779, August 2009.

[7] “Brief Summary to Clarify Message MSG Instruction CIP Paths and Pathing,” Allen-Bradley Knowledge Base Document, no. 22562, December 2008.

[8] C.A. Parrott, “Real-Time Reconfiguration of Programmable Logic Controller Communication Paths,” M.S. Thesis, Dept. Elect. Comp. Eng., Missouri S&T, Rolla, 2009.

VITA

Richard Andrew Hardison was born April 13, 1978 in Ashland, Kentucky to Richard L. and Pauletta A. Hardison. He joined the U.S. Army Reserves in 1996 as a junior in high school and graduated the following year. In 2001 his reserve unit was mobilized to Fort Leonard Wood Missouri as part of Operation Noble Eagle. After the mobilization ended, Richard relocated to Missouri and worked for Briggs and Stratton. In 2004 he was deployed to Iraq for a one year tour in Operation Iraqi Freedom. In January of 2007 he transferred to The University of Missouri – Rolla then earned a Bachelor of Science degree in Electrical Engineering in 2009. Upon completion of his undergraduate work he continued on to graduate school, completing a Master of Science degree in Electrical Engineering from Missouri University of Science and Technology in 2010.