Fall 2009

Real-time reconfiguration of programmable logic controller communication paths

Curtis Alan Parrott

Department:

Recommended Citation

Parrott, Curtis Alan, "Real-time reconfiguration of programmable logic controller communication paths" (2009). *Masters Theses*. 4738.
https://scholarsmine.mst.edu/masters_theses/4738

REAL-TIME RECONFIGURATION OF PROGRAMMABLE LOGIC CONTROLLER COMMUNICATION PATHS

by

CURTIS ALAN PARROTT

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

2009

Approved by

Kelvin T. Erickson, Advisor
Ann Miller
Steven Grant

ABSTRACT

This thesis explores the topics related to reconfiguration of Programmable Logic Controller's (PLC's) communications paths as it relates to network security and reliability. These paths are normally fixed, which creates a single fault point which can easily be disrupted by network failure or network based attack. With the ability for autonomous communications path reconfiguration these disruptions in communications can be avoided or bypassed. This work builds on these principles and a series of PLC programs are developed to facilitate several things: Scanning of the three different network types most common in PLC to PLC communications; a comprehensive network scan routine for locating multiple communications paths to available network enabled modules and devices; add-on functions for verifying and using these found communications paths; and MS Excel macros for documenting the found modules and devices along with their communications paths from the host processor.

ACKNOWLEDGMENTS

I would like to thank Dr. Erickson and my committee for their guidance during the pursuit of a Master of Science degree in Electrical Engineering at the Missouri University of Science and Technology. Without the help of Dr. Erickson and the rest of my committee it would not have been possible to complete. The guidance of Dr. Erickson during the entirety of my master’s work was valuable beyond words and I am deeply happy I was able to work with him.

My parents have also helped me greatly over my entire life in more ways that can be mentioned. I owe them a great deal for their help emotionally and financially while completing my master’s work.

TABLE OF CONTENTS

Page

ABSTRACT iii
ACKNOWLEDGMENTS iv
LIST OF ILLUSTRATIONS vii
LIST OF TABLES viii
SECTION
1. INTRODUCTION 1
2. NETWORK SCAN APPROACH AND ROUTINES 3
2.1. NETWORK SCAN APPROACH 3
2.2. MESSAGE SETUP 3
2.2.1. Communication Message for Device Who. 5
2.2.2. Communication Message for Serial Number. 5
2.2.3. MSG Communications Path Formulation. 7
2.3. NETWORK SCAN ROUTINES 9
2.3.1. ControlNet Scan Routine 10
2.3.2. Ethernet Scan Routine 12
2.3.3. DeviceNet Scan Routine. 13
2.4. BACKPLANE SCAN ROUTINE 15
2.5. MAIN SCAN ROUTINE 16
2.6. VERIFY PATHS ROUTINE 17
2.7. FIND_PATH ADD-ON INSTRUCTION 19
3. LIST AND DOCUMENT FOUND PATHS 20
3.1. MICROSOFT EXCEL PATH EXTRACTION MACRO 20
4. VERIFY AND TESTING 22
5. CONCLUSION AND FUTURE WORK 23

APPENDICES

A. CONTROLNET SCAN ROUTINE 25

B. ETHERNET SCAN ROUTINE 28

C. DEVICENET SCAN ROUTINE 32

D. BACKPLANE SCAN ROUTINE 36

E. MAIN SCAN ROUTINE 39

F. PATH VERIFICATION ROUTINE 43

G. FIND PATH ADD-ON INSTRUCTION 46

H. MICROSOFT EXCEL VISUAL BASIC MACRO 49

BIBLIOGRAPHY 53

VITA 54

LIST OF ILLUSTRATIONS

Page

Figure 2.1. Configuration dialog box setup for the DeviceWho MSG block. 6

Figure 2.2. Configuration dialog box setup for the Serial Number MSG block................ 6

Figure 2.3. Illustration of communications path example 9

Figure 2.4. Pseudo code for the ControlNet scan routine. 11

Figure 2.5. Pseudo code for the Ethernet scan routine. 13

Figure 2.6. Pseudo code for the Ethernet scan routine. 15

Figure 2.7. Pseudo code for the backplane scan routine. 16

Figure 2.8. Pseudo code for Verify Path routine. 18

Figure 4.1. MST PLC Laboratory layout with separated ControlLogix Ethernet subnet.22

LIST OF TABLES

Page

Table 2.1. Media Types for Path formation [5]. 7

Table 2.2. Node or Address types for different communications networks [5].................. 7

Table 2.3. Decoding of example message path 1,5,2,100.100.100.100,1,0,2,24,1,10 [5]. . 8

1. INTRODUCTION

The increasing dependence on industrial control systems in daily life requires the evaluation of the reliability and security of these systems. These systems become critical either for monetary or safety and security reasons. Down time in any industry can cost millions of dollars per hour and reliability of the highest degree is imperative [1]. Also, in critical infrastructure real-time response is necessary to safely manage temperature, pressure, water level and other critical factors. These Supervisory, Control and Data Acquisition (SCADA) systems are increasingly essential in energy, telecommunications and manufacturing systems. SCADA systems, as opposed to other systems and networks, require high reliability and long life expectancy, even under 24x7 operation.

Due to the increasing integration of Human Machine Interface (HMI) equipment and Programmable Logic Controllers (PLCs) with enterprise networks these SCADA networks are more susceptible to all of the vulnerabilities of Ethernet-based network attacks. This trend of moving from proprietary communications protocols to standard protocols, including Ethernet, also opens these networks to attack.

There are a few standard networks that connect PLC processors with other PLC processors or distributed I/O; the trend is to migrate to Ethernet-based communications systems often paired with a control-specific protocol layer, like CIP or Modbus. Typically, these networks pass individual messages between devices over fixed communications paths. Autonomous reconfiguration of these communications paths is needed to survive a cyber attack, physical attack or any possible communications failure which disrupts the original communications path. This autonomous reconfiguration system must be contained within the PLC processor for maximum protection.

Although it is not known if it has been attempted, it is possible to autonomously change the communications path within several brands of PLCs. This communication path reconfiguration is possible with Allen-Bradley ControlLogix, PLC-5 and SLC-500 processors. The PLC program running on the processor is able to access all necessary components required to perform this reconfiguration such that no additional components need to be added to the PLC processor firmware. This eliminates the need for collaboration with the manufacturer and the use of specialized firmware development tools. Allen-Bradley ControlLogix processors are the focus of this work. However the methods used here can be ported to other models in the Allen-Bradley family of PLCs.

2. NETWORK SCAN APPROACH AND ROUTINES

2.1. NETWORK SCAN APPROACH

The approach to scanning the available networks is to build a comprehensive table of all found modules and devices. This table of modules is separated into two groups: communications modules and other modules and devices. The separation allows for easy differentiation between scannable modules and other modules in the scanning routines. Each of the entries is identified by serial number so other addressing is not needed, as there is no other easy way to identify the modules. Under each serial number all of the found paths are recorded in the order in which they are discovered. If the module is a communication module the network type of the module is also recorded for later use in scanning. As new paths are found for a given module the new paths are appended to the path list.

In order to do a comprehensive scan of all of the available networks there are three main parts: two communication messages, scan routines for each of the three network modules covered in this work and the main scan routine that ties all parts together.   These parts of the program are elaborated upon in the following sections.

2.2. MESSAGE SETUP

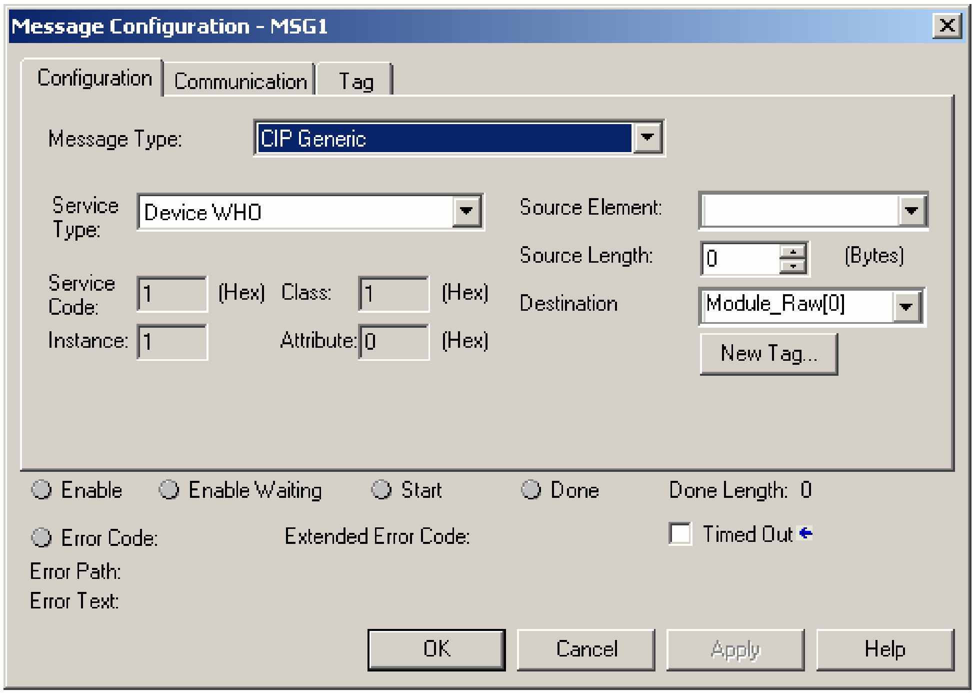
The message, or MSG, block provides a peer-to-peer communications function for Allen-Bradley PLCs [2]. This MSG block is mainly used for PLC-to-PLC communications [2]. This is done by reading or writing tags from the target PLC in any of the families of Allen-Bradley PLCs [2]. Although PLC-to-PLC communications is the main use of the MSG block, communications with other modules and devices over the Allen-Bradley networks is also possible. This allows the use of MSG blocks to discover modules and devices other than just PLC processors. This advantage is used in the network scanning routines in order to comprehensively scan the entire SCADA network.

The MSG instruction has an associated tag which holds all relevant data used in the MSG instruction execution: status flags, destination or received tag, communications path, timeout values, and other required variables. The main concern of this work is with the communications path contained within the tag. It is contained in an 86 element string which contains the path to the desired module or device. The components of this path are built from the host processor to the destination module or device and can take many hops between the two. The structure of these paths is explained in a later section.

Two different communication messages are used to scan for modules and devices on the networks. The first message is configured to return the “Device Who” information of the module and another which just returns the serial number of certain devices. The messages are both setup and saved ahead of time. These messages are described in detail the following sections.

2.2.1. Communication Message for Device Who. As stated before, a MSG is setup to retrieve the "Device Who" information from a module or device. If the module or device supports the Device Who query, the returned information includes the vender ID of the module, product type and code, firmware revisions, serial number and the module catalog name and description. This information is returned in the form of a single string and must be decoded into these individual parts. The decoding is done via the DECODE_DEVICE_WHO function. Instructions for decoding this information can be seen in [3]. This function places the decoded information in a User Defined Type (UDT) tag, which is similar to a structure in C programming, to make the information more easily accessible. This message is used to find all modules and devices on the ControlNet, Ethernet networks and most of the devices on the DeviceNet network. The setup for this message can be seen in [3]. The setup box for the "Device Who" MSG block can be seen in figure 2.1.

2.2.2. Communication Message for Serial Number. Not all of the devices on the network respond to the "Device Who" query and so for these devices it is needful to request the Serial Number in a separate MSG query. These devices which do not support the "Device Who" are standalone devices, like stack light bases, photo eyes and pushbutton stations, mostly on the DeviceNet network. This MSG is setup as a CIP Generic MSG with parameters found in [4]. The MSG returns the serial number of the queried device, if the device supports this query, as a DINT. The setup box for the Serial Number MSG block can be seen in figure 2.2.



Message Configuration - MSG1

Configuration | Communication | Tag

Message Type: CIP Generic

Service Type: Device WHO

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

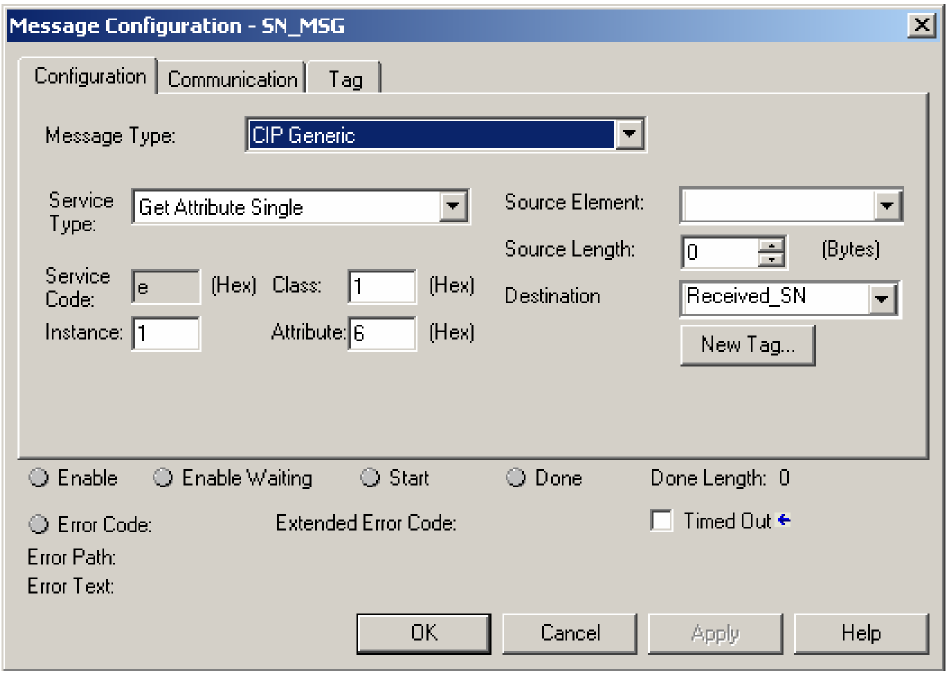
Instance: 1 Attribute: 0 (Hex)

Source Element:

Source Length: 0 (Bytes)

Destination: Module_Raw[0]

Figure 2.1. Configuration dialog box setup for the DeviceWho MSG block.



Message Configuration - SN_MSG

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:

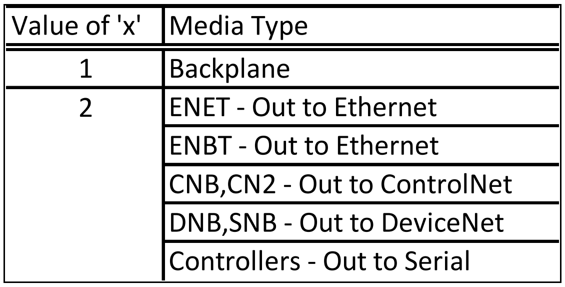
Source Length: 0 (Bytes)

Destination: Received_SN

Figure 2.2. Configuration dialog box setup for the Serial Number MSG block.

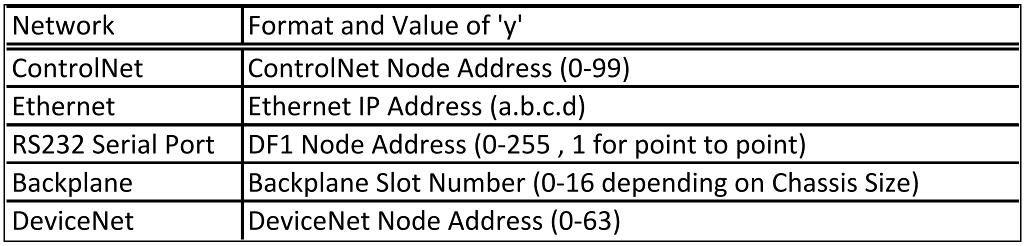
2.2.3. MSG Communications Path Formulation. The communications path to a particular module or device is encoded in a special format. This format is different whether input from the configuration menu or modified on the tag level. The paths are built from path pairs of the form x,y; the x component is the media type and y is the node or address for the media selected in x. The values which x can take can be seen below in table 2.1 and the values for y can be seen in table 2.2. These x,y pairs can be concatenated together, separated by commas, and the pairs can be concatenated together to create multi-hop paths over multiple modules and networks.

Table 2.1. Media Types for Path formation [5].



Value of 'x'	Media Type
1	Backplane
2	ENET - Out to Ethernet
	ENBT - Out to Ethernet
	CNB,CN2 - Out to ControlNet
	DNB,SNB - Out to DeviceNet
	Controllers - Out to Serial

Table 2.2. Node or Address types for different communications networks [5]



Network	Format and Value of 'y'
ControlNet	ControlNet Node Address (0-99)
Ethernet	Ethernet IP Address (a.b.c.d)
RS232 Serial Port	DF1 Node Address (0-255 , 1 for point to point)
Backplane	Backplane Slot Number (0-16 depending on Chassis Size)
DeviceNet	DeviceNet Node Address (0-63)

In order to understand the path statements an example is presented, taken from [5]. This example shows how to develop or decode a path statement. The given path is 1, 5, 2, 100.100.100.100, 1, 0, 2, 24, 1, 10. The decoded message can be seen in table 2.3, as well as an illustration of the decoded path in figure 2.3; in this illustration all paths segments are numbered with orange boxes and all paths are depicted in bold colors: red for backplane moves, gray for Ethernet moves and blue for ControlNet moves. This example shows how to communicate from one AB ControlLogix processor to another over Ethernet and ControlNet. This example does not include a path through a DeviceNet network but the setup is very similar to ControlNet with only less node addresses available.

Table 2.3. Decoding of example message path 1,5,2,100.100.100.100,1,0,2,24,1,10 [5].

Segment	Element of Path	Description of Path element
	Starting at ControlLogix Processor 1756-Lxx	
1	1	Out to the backplane
	5	To slot 5 (Slot 5 contains an Ethernet Module)
2	2	Out to Ethernet
	100.100.100.100	To Ethernet address 100.100.100.100
3	1	Out to the backplane
	0	To slot 0 (Slot 0 contains an ControlNet Module)
4	2	Out to ControlNet
	24	To ControlNet Node 24 (Another CNB Module)
5	1	Out to the backplane
	10	To slot 10 (Contains the Destintation Processor 1756-Lxx)

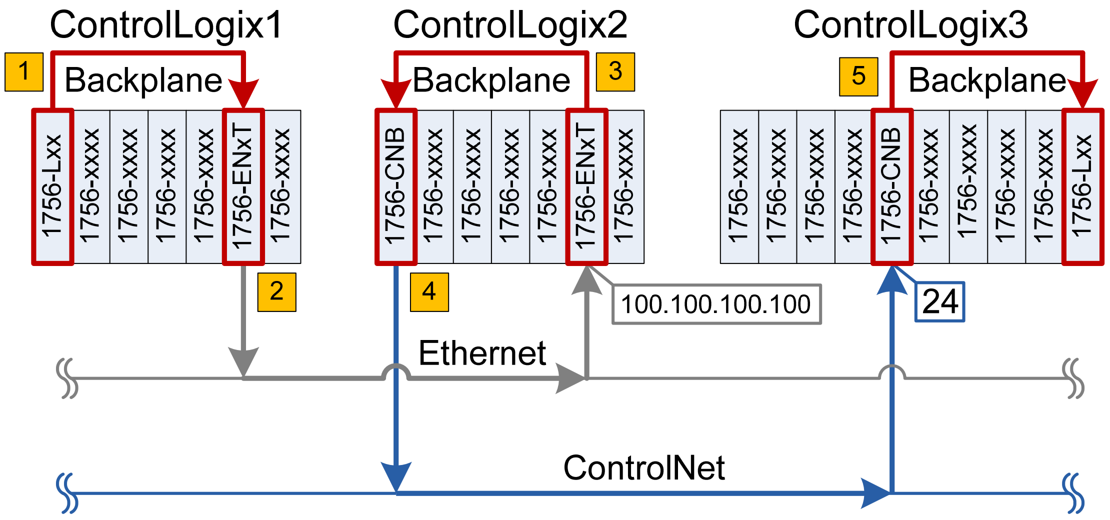


Figure 2.3. Illustration of communications path example

In order for a program to change the communications path the string tag holding the path must be modified. The path string is held in the MSG setup tag and is not encoded exactly as typed in the configuration box. These differences are not documented by Allen-Bradley and had to be discovered by entering different paths and examining the resulting string in the MSG setup tag. For all of the media types, as well as the DeviceNet and ControlNet, node addresses are not encoded as ASCII but are just encoded as plain byte numbers. The Ethernet IP address is, however, is encoded as an ASCII string in dotted-decimal notation. The two bytes in front of the ASCII encoded IP string identify that the proceeding information is an IP address and the length of the IP string. The IP address also contains the separation points, which are also encoded as ASCII. These findings make it possible to build a program which can change the path string and automatically change the destination module or device.

2.3. NETWORK SCAN ROUTINES

In order to construct a program for populating a table of all modules and devices on the various PLC networks, routines must be written for scanning the various PLC networks. The main networks considered in this work are Ethernet, ControlNet and

DeviceNet. However, there are many more networks used in SCADA systems but most do not allow Allen-Bradley modules to communicate through them. These three networks are the main networks in ControlLogix PLC systems.

All of the routines, which are expanded upon in the later sections, are based on a similar code structure. Each subroutine contains a state machine which governs the steps the programs take. These routines take advantage of the two pre-configured MSG functions to scan the designated network. Each routine is passed a base-path, which is the path to be scanned, scanning range and base-path serial number. The base-path serial number is the serial number of the module which will have its network scanned.

2.3.1. ControlNet Scan Routine. The first routine discussed is the scanning routine for ControlNet. This routine is passed the base-path, scan range and module serial number. This routine increments through all of the ControlNet node addresses within the supplied range through the supplied path and records all of the found modules or devices on that network.

The ControlNet routine has seven steps which facilitate the scanning of the ControlNet network. The first step to this routine is to concatenate the passed base-path with the media-type, which would be an ‘x’ value of 2 for ControlNet, and the node address to be scanned. The MSG block is then reset to prepare it to be used. The second step is then to activate the MSG block. This starts the communication with the destination node address. Once the MSG block is activated, the DN and ER bits are monitored. If the MSG block DN bit is activated then the message was successful and the next step is to decode the “Device Who” information received. If the MSG block ER bit is active then either the destination node is empty or the module does not support the “Device Who” query and the program skips to the next to last step. Once the “Device Who” information is decoded the found module table is searched for the found module. If the module or device is found in the table then the new path for that module is compared to the other found paths for that module or device. If the new path is different it is added to the list and if it is a repeated path then it is ignored. If the module is new then the module is appended to the table of modules and the path is added to it. The next step is to increment the node address and to check if it is still in the desired scan range. When the whole scan range is scanned the routine signals that it is done and stays in a final wait state. The pseudo code for the routine can be seen in figure 2.4, which illustrates a more complete explanation of the routine. Also, the code listing appears in Appendix A.

```
Step1: Concatenate the passed scan path with the ControlNet media type and node address
        Setup the Device Who MSG block
        GOTO Step 2
Step 2: Enable Device Who MSG block
        GOTO Step 3
Step 3: Wait for Done or Error bits to be set
        IF MSG Done THEN GOTO Step 4
        IF MSG Error THEN GOTO Step 6.
Step 4: Call decode_device_who subroutine
        GOTO Step 5
Step 5: Check for found module in table
        IF Found Module in Existing Module List THEN Add path to found paths if new
        IF Found Module NOT in Existing Module List THEN Add Module to Found Modules List
        GOTO Step 6
Step 6: Increment the node address.
        IF Node Address == Upper Scan Range THEN GOTO Step 7
        IF Node Address != Upper Scan Range THEN  GOTO Step 1
Step 7: Signal scan is done; Wait in step 7.
```

Figure 2.4. Pseudo code for the ControlNet scan routine.

2.3.2. Ethernet Scan Routine. The next routine discussed is the scanning routine for Ethernet. This routine is similar to the other routines in that it takes in the base-path, scan range and module serial number. This routine, however, does not scan all IP addresses but only scans the last octet of the address and is passed a base address of the first three octets. This least significant byte, or fourth octet, of the IP address is scanned within the supplied range, through the supplied path, and all modules or devices found are recorded in the table of found modules.

The Ethernet scan routine has seven steps which facilitate the scanning of the network. The first step to this routine is to convert the Ethernet IP address to a string in dotted-decimal notation, concatenate the passed base-path with the media-type, which would be an ‘x’ value of 2 for Ethernet, and the IP address string and saved to the DeviceWho MSG block’s communications path. The MSG block is then reset to prepare it to be used. The second step is then to activate the MSG block, which starts the communication with the destination IP address. Once the MSG block is activated, the DN and ER bits are monitored for this MSG block. If the MSG block DN bit is activated then the message is successful and the next step is to decode the “Device Who” information received. If the MSG block ER bit is active then either the destination node is empty or the module does not support the “Device Who” query and the program skips to the next to last step. Once the “Device Who” information is decoded the found module table is searched for the found module. If the module or device is found in the table then the new path is added for that module, if the path does not already exist. If the module has not been found before, the module is appended to the table of modules and the path is added to it. The next step is to increment counter, which is used to set the fourth octet, and is checked to see if it is still in the scan range. When the whole scan range is scanned the routine signals it is done and stays in a final wait state. The pseudo code for the routine can be seen in figure 2.5, which illustrates a more complete explanation of the Ethernet scan routine. Also, the code listing appears in Appendix B.

```
Step1: Convert the IP address octets into a string separated by points.  Concatenate the base path with the
           media type, Ethernet preface, IP string length and IP string.
           Prepare the Device Who MSG.
           GOTO Step 2
Step 2: Enable Device Who MSG blocks
           GOTO Step 3
Step 3: Wait for Done or Error bits to be set
           IF MSG Done THEN GOTO Step 4
           IF MSG Error THEN GOTO Step 6.
Step 4: Call decode_device_who subroutine
           GOTO Step 5
Step 5: Check for found module in table
           IF Found Module in Existing Module List THEN Add path to found paths if new
           IF Found Module NOT in Existing Module List THEN Add Module to Found Modules List
           GOTO Step 6
Step 6: Increment the IP address fourth octet.
           IF Fourth Octet == Upper Scan Range THEN GOTO Step 7
           IF Fourth Octet != Upper Scan Range THEN  GOTO Step 1
Step 7: Signal scan is done; Wait in step 7.
```

Figure 2.5. Pseudo code for the Ethernet scan routine.

2.3.3. DeviceNet Scan Routine. The final routine discussed is the scanning routine for DeviceNet. The routine is slightly different than the other two routines since some DeviceNet devices do not respond to the “Device Who” query. This facilitates the need for another MSG block, explained earlier, which only queries the serial number of a device. The routine increments through the supplied DeviceNet node address range, through the supplied path, and records all of the found modules or devices on the network.

The DeviceNet routine has eight steps which facilitate the scanning of the network. The first step to this routine is to concatenate the passed base-path with the media-type, which would be an ‘x’ value of 2 for DeviceNet, and the node address to be scanned and saved to the DeviceWho and SerialNumber MSG blocks’ communications paths. The MSG blocks are then reset in preparation for reuse. The second step is then to activate the MSG block and verify that they are enabled, which starts the communication with the destination node address. Once the MSG blocks are activated, the DN and ER bits are monitored for both MSG blocks. If the DeviceWho MSG block DN bit is activated then the message was successful and the next step is to decode the “Device Who” information received. Alternately, if the SerialNumber MSG block DN bit is activated then the decode_device_who subroutine is skipped.   If both the MSG block ER bits are active then either the destination node is empty or the module does not support either of the queries and the program skips to the next to last step. The found modules table is then searched for the found module or device and if it is an existing module the found path is added if new. This check avoids adding redundant paths. If the module has not been found beforehand the module and path are appended to the table. The next step is to increment the node address and verify if it is still in the scan range. When the whole scan range is scanned the routine signals it is done and stays in a final wait state. The pseudo code for the routine can be seen in figure 2.6, which illustrates a more complete explanation of the routine. Also, the code listing appears in Appendix C.

```
Step1: Concatenate the passed scan path with the DeviceNet media type and node address
        Setup the Device Who and Serial Number MSG block
        GOTO Step 2
Step 2: Enable Device Who and Serial Number MSG blocks
        GOTO Step 3
Step 3: Wait for Done or Error bits to be set
        IF either MSG Done THEN GOTO Step 4
        IF both MSG Error THEN GOTO Step 6.
Step 4: IF Device Who Done THEN Call decode_device_who subroutine
        GOTO Step 5
Step 5: Check for found module in table
        IF Found Module in Existing Module List THEN Add path to found paths if new
        IF Found Module NOT in Existing Module List THEN Add Module to Found Modules List
        GOTO Step 6
Step 6: Increment the node address.
        IF Node Address == Upper Scan Range THEN GOTO Step 7
        IF Node Address != Upper Scan Range THEN  GOTO Step 1
Step 7: Signal scan is done; Wait in step 7.
```

Figure 2.6. Pseudo code for the Ethernet scan routine.

2.4. BACKPLANE SCAN ROUTINE

In earlier sections the network scan routines were covered; however, the capability to scan the backplane of the found communications modules is still needed. This routine is similar to the network scan routines with a different media type. The routine is passed the path to a communication module and scans the backplane of that module. During the scanning process all new found modules are appended to the found modules table.

The first step of this routine is to concatenate the passed module path and the media type associated with the backplane and the slot to be scanned. The “Device Who” MSG block is also prepared for transmission. Next the flag bits are then monitored; if the MSG is done then the “Device Who” info is decoded; if the MSG is in error then the slot is assumed to be empty, or may contain an incompatible module, and the state machine skips to the last step. Once the “Device Who” information is decoded the found module table is searched for the found module. If the module or device is found in the table then the new path is added for that module, if it does not already exist. If the module is new, the module is appended to the table of modules and the path is added to it. The next step is to increment the slot number and is then checked to make sure it is less than seventeen; the largest ControlLogix rack available is a seventeen slot rack. When the entire rack is scanned the routine signals it is done and stays in a final wait state. The pseudo code for the routine can be seen in figure 2.7, which illustrates a more complete explanation of the backplane scan routine. Also, the code listing appears in Appendix D.

```
Step 1: Concatenate the passed module path with the media type for backplane and scanned slot.
Step 2: Enable Device Who block
        GOTO Step 3
Step 3: Wait for Done or Error bits to be set
        IF MSG Done THEN GOTO Step 4
        IF MSG Error THEN GOTO Step 6.
Step 4: IF Device Who Done THEN Call decode_device_who subroutine
        GOTO Step 5
Step 5: Check for found module in table
        IF Found Module in Existing Module List THEN Add path to found paths if new
        IF Found Module NOT in Existing Module List THEN Add Module to Found Modules List
        GOTO Step 6
Step 6: Increment the Slot Number.
        IF Slot Number == Max Slots THEN GOTO Step 7
        IF Slot Number <> Max Slots THEN  GOTO Step 1
Step 7: Signal scan is done; Wait in step 7.
```

Figure 2.7. Pseudo code for the backplane scan routine.

2.5. MAIN SCAN ROUTINE

The main scan routine combines all of the above subroutines to create a comprehensive scanning routine which scans the entire PLC networks. This routine is a state machine which steps the program through the six steps required to scan the entire network. The routine repeats itself three times in order to fill the array of path strings, but this repeat number can be changed to suit the PLC network.

The routine first scans the backplane of the host processors; this is done by passing the backplane scan routine a blank base path. This step will populate the first entry in the found modules table and find all the available communications modules. The next step is to scan the networks associated with these found communications modules by calling the appropriate routine for the found module. Once all of the communications modules have been scanned and paths updated, the backplanes of all of the found communications modules are scanned. This may find alternate paths to already found modules; this is the basis of the scan routine since alternate paths are found for all of the modules. Very quickly all of the modules accessible on the network are found and alternate paths are found. Once all of the path slots are filled, or the program has repeated a certain number of times, the program runs the verify paths routine. This routine checks all of the found paths and makes sure they are still valid. Once this routine is done it remains in a wait state and signals it is done. Once the routine is in this wait state it can be recalled to update the paths or the table generated can be used to find different path to specific modules. Also, the code listing can be seen in Appendix E.

2.6. VERIFY PATHS ROUTINE

This routine is used to verify the paths for all of the modules found in the main scan routine. The routine uses the appropriate MSG block to check that all the paths return the correct serial number. Since the topology of the network may have changed due to module failure or network attack may have occurred, some paths may no longer be valid and must be checked. Once the path is checked, the valid path flag gets cleared or set accordingly.

In order to process the steps needed to verify the paths two state machines are used. Since the modules are separated into communications and non-communications modules these two state machines are needed. Both state machines are identical but each checks the two different types of modules. The state machine steps through all of the found modules and checks all of the paths for each of the modules. In order to check all the modules both of the MSG blocks are needed. The state machine first sets the communications path for each MSG block to the path to be checked and enables the MSG blocks. Once the MSG blocks are enabled, the program checks the flag bits for the MSG blocks to set the valid bit for the checked path. If either MSG block is done, the serial number is checked to make sure the path still communicates with the correct module; if the module is correct then the valid path flag bit is set for the communications path. However, if the path is not to the correct module, or if both MSG blocks are in error, then the valid flag bit is cleared. The state machine is then repeated for the non-communications modules. The state machine for the verify paths routine can be seen in figure 2.8. Also, the code listing can be seen in Appendix F.

```
Step 1: Copy path to both DeviceWho and SerialNumber MSG block;
        GOTO Step 2
Step 2: Enable Device Who and Serial Number MSG blocks
        GOTO Step 3
Step 3: Wait for Done or Error bits to be set ; Verify_Path = 0 ;
        IF either MSG Done THEN GOTO Step 4
        IF both MSG Error THEN GOTO Step 6
Step 4: IF DeviceWho MSG Done THEN Call decode_device_who subroutine
        GOTO Step 5
Step 5: IF DeviceWho Serial Number == Path Serial Number THEN Verify_Path = 1
        GOTO Step 6
Step 6: Increment the Path counter
        IF Path Counter != Number Paths THEN  GOTO Step 1
        ELSE GOTO Step 7
Step 7: Increment the Module counter
        IF Module Counter != Number Modules THEN  GOTO Step 1
        ELSE GOTO Step 8
Step 8: Wait
```

Figure 2.8. Pseudo code for Verify Path routine.

2.7. FIND_PATH ADD-ON INSTRUCTION

The Find_Path function returns a valid path to a given module. This routine is passed a serial number and returns the first valid path, from the found modules table, which communicates with the module identified by the passed serial number. This program is different from the other routines in that it is built as an Add-On instruction. Since an add-on instruction cannot call an outside routine, the Verify_Paths routine must be called before using an instance of this function.

This routine has several inputs and outputs; the routine must be passed the FoundModules UDT, the serial number to be searched for and the string to hold the found path. The routine also has several output flag bits which signal the status of the add-on instruction: the DN bit signals the module serial number was found and a path has been returned; the ER bit signals that either the module serial number was not found or none of the paths for the module are valid. Once the DN bit is set, the string to hold the returned path will contain the communications path to the module identified by the serial number. If the ER bit is set then the communications path string will be empty and no communications path is available to the specified module. The add-on instruction setup and code listing can be seen in Appendix G.

3. LIST AND DOCUMENT FOUND PATHS

Once the main scan routine has completed and all modules and devices have been found it may be necessary to extract the information in the FoundModules UDT for easy decoding and documentation. There is no easy way to view and document with any single piece of software. For the ability to extract tag values from the PLC, Allen-Bradley RSLinx software is needed. RSLinx allows OPC and DDE transfers to and from a networked PLC to extract tag values. In order to use these communications links a program must be written. Excel contains DDE transfer functions accessible with the built-in Visual Basic macro language and provides a way to extract these paths.

3.1. MICROSOFT EXCEL PATH EXTRACTION MACRO

The Excel macro is written in VBA (Visual Basic for Applications) and uses the DDE transfer functions. These functions communicate with RSLinx, which in turn communicates with the networked PLC. In order for this process to work, a topic for the destination PLC must be configured in RSLinx. This topic name is then used with the DDE communications functions to specify the destination PLC. First, in order to use the DDE functions, a channel must be opened using the DDEInitiate function which states the destination program, which is RSLinx in this case, and the topic, which is associated with the destination PLC in RSLinx. Once this is done, the DDE functions use this created channel for the request function. This function, DDERequest, takes in the channel and the requested tag in an ASCII encoded string.

Due to the path strings not being totally ASCII encoded, the string values must be fetched a byte at a time instead of all at once as a single string. If the string is fetched as a single string the non-standard characters are not preserved. This requires a for-loop to be used to fetch a byte at a time. This loop then differentiates between the ASCII encoded Ethernet IP address and the other parts, then rebuilds the string as it would appear in the PLC.

This routine has two different loops which step through every found module and extracts every path for that module. The serial numbers are encoded in hexadecimal and then all paths are listed under each serial number. The three different flags associated with each path are also extracted: valid path, scanned network and backplane scanned flag bits. Each PLC topic has its own sheet in the Excel worksheet. The macro steps through each sheet and uses the sheet name as the DDE topic. This allows for easy addition of each PLC needed to be documented. The listing of the Excel macro VBA can be seen in Appendix H.

4. VERIFY AND TESTING

In order to verify the workings of the scan routines, the scan routine was adjusted for every v16 ControlLogix processors in the MST PLC lab. This includes 13 ControlLogix processors which vary in model and slot numbers. The layout of the scannable modules and devices can be seen in figure 4.1. First the ControlLogix subnet was connected to the department Ethernet and the scanning routines were run. After these routine had finished the path extraction MS Excel macros were run to extract the found paths from every processor running the routines.

Once all of these processors had completed scanning the network and found paths documented, the Found Modules UDT where cleared and the programs reset. Next the ControlLogix Ethernet subnet was disconnected from the department network. The modules were verified working and had their IP addresses set. After these steps the scan routines were run to completion. At the time the routines are done, the MS Excel macro is run in a new worksheet to extract the new paths. The extracted paths, as documented in the MS Excel worksheets, are decoded by hand to verify they point to the appropriate module or device.

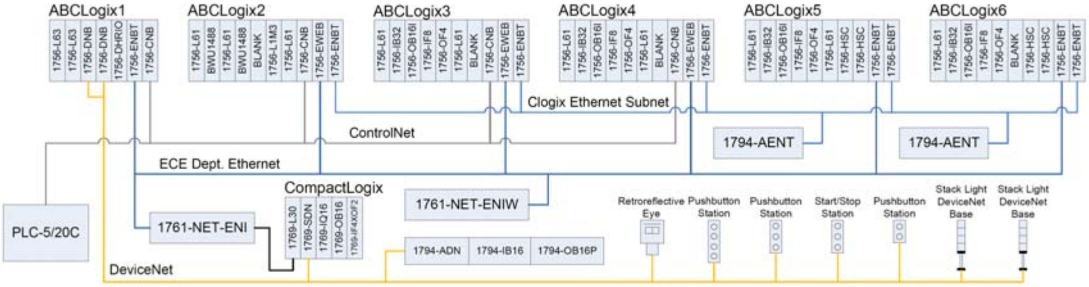


Figure 4.1. MST PLC Laboratory layout with separated ControlLogix Ethernet subnet.

5. CONCLUSION AND FUTURE WORK

This work has demonstrated a complete system for reconfiguring communications paths using Allen-Bradley ControlLogix PLC processors. The set of PLC structured text routines demonstrate scanning of three of the major communications networks: ControlNet, Ethernet and DeviceNet. These routines, when combined in a larger scanning routine, are used to create a comprehensive scanning routine for finding all modules and devices available on the network, as well as multiple valid paths to these found modules and devices. Since these paths may become invalid due to a number of reasons, including network attack or network failure, a routine is demonstrated to check all found paths for validity. Once these paths have been validated, it is desired to use the found valid paths. An add-on instruction is included which searches for the desired module or device in the found modules table and returns the first valid path found for this device. Along with the PLC based programs an MS Excel document is included which will fetch the found modules table from all specified PLCs.

Some improvements could be made to the existing work to make the network scan complete faster and scanning routines easier to deploy on a larger scale. First, multiple MSG blocks could be used to initiate multiple messages in parallel, instead of serially as they are now. This technique is thought to be used in Allen-Bradley’s RSLinx software and has potential for greatly increasing the speed of the scan routines. Also in an effort to increase the speed of the network scan the timeouts could be optimized. These could be initially adjusted globally but could be further refined. As a final improvement, these routines might be packaged as add-on instructions for ease of deployment. This process is complicated due to the direct manipulation of the MSG block tags and may be an impassable hurdle with the routines in their current form.

As this work provides a working example of one approach to the problem at hand, another approach could be explored in future work. The adopted approach of completing a comprehensive scan can be redundant if multiple PLCs on the network are performing the same scan. If this is the case, an AD-HOC approach could be explored. The host processor would only scan one level; i.e., the networks and racks directly connected to its host rack. When the processor wishes to communicate with a module or device more than one communications level away from it, the host processor would send a request to all other PLC processors running the scan routine for a path to the desired module. They would either relay the message to the other processors it can communicate within one level or return the path to the desired processor. This approach has the possibility for greatly speeding the scanning of the network by effectively parallelizing the scan over many modules.

APPENDIX A.
CONTROLNET SCAN ROUTINE

```
IF NOT ONS3 THEN
    i := ScanRange.Min;
    ONS3 := 1;
END_IF;

CASE Step OF
    1:  // Set message path to backplane to slot 'i'
        DELETE(MSG1.Path,82,1,MSG1.Path);
        INSERT(MSG1.Path,Path_Net,1,MSG1.Path);
        IF MSG1.path.len > 84 THEN
            Step := 7;
        ELSE
            MSG1.Path.Data[MSG1.path.len] := 2; // Go out on ControlNet
            MSG1.Path.Data[MSG1.path.len+1] := i; // to address 'i'
            MSG1.path.len := MSG1.path.len + 2;

            // Clear DN and ER bits to make sure ready for new scan.
            MSG1.DN := 0;
            MSG1.ER := 0;
            Step := Step + 1;
        END_IF;

    2:  //Send Message for "Module WHO" to CNet Node 'i'
        MSG(MSG1);

        //If MSG1 is enabled, goto next step (Should always be 1)
        IF MSG1.EN THEN
            Step := Step + 1;
        END_IF;

    3:  IF MSG1.DN THEN
            // Clear DN bit
            MSG1.DN := 0;

            // GoTo next step
            Step := Step + 1;
        ELSIF MSG1.ER THEN
            // Clear ER bit
            MSG1.ER := 0;
            // Skip to next slot
            Step := 6;
        END_IF;

    4:  // Decode RAW data from MSG
        Decode_Module_Who(DecodeModWho1,Module_Raw,Module);
        // GoTo next step
        Step := Step + 1;

    5:  // Check if current slot contains my processor
        IF Module.Product_Type = 12 AND Mod_SN <> Module.Serial_Num THEN
            NewMod := 1;

            FOR m := 0 TO FoundModules.NumComm DO
                IF FoundModules.Comm[m].SN = Module.Serial_Num THEN
                    NewPath := 1;
                    FOR h := 0 TO 24 DO
                        IF FoundModules.Comm[m].PATH[h] = MSG1.Path THEN
                            NewPath := 0;
                        END_IF;
                    END_FOR;
                    IF NewPath AND FoundModules.Comm[m].NumPaths < 24 THEN
                        NumPaths := FoundModules.Comm[m].NumPaths;
                        INSERT(FoundModules.Comm[m].Path[NumPaths],MSG1.Path,1, ▼
                                                                FoundModules.Comm[m].Path[NumPaths]);
                        FoundModules.Comm[m].NetScanned[NumPaths] := 1;
                        FoundModules.Comm[m].NumPaths := NumPaths + 1;
                    END_IF;
                    NewMod := 0;
                END_IF;
            END_FOR;
            IF NewMod AND FoundModules.NumComm < 100 THEN
                FoundModules.Comm[FoundModules.NumComm].SN := Module.Serial_Num;
                FoundModules.Comm[FoundModules.NumComm].Type := Module.Product_Code;
                // Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
                NumPaths := FoundModules.Comm[FoundModules.NumComm].NumPaths;
                INSERT(FoundModules.Comm[FoundModules.NumComm].Path[NumPaths],MSG1.Path,1,▼
                                                                FoundModules.Comm[FoundModules.
                NumComm].Path[NumPaths]);
                FoundModules.Comm[FoundModules.NumComm].BPScanned[0] := 0;
                FoundModules.Comm[FoundModules.NumComm].NetScanned[NumPaths] := 1;
                FoundModules.Comm[FoundModules.NumComm].NumPaths := NumPaths + 1;
                FoundModules.NumComm := FoundModules.NumComm + 1;
            END_IF;
        ELSIF Module.product_type <> 12 AND Mod_SN <> Module.Serial_Num THEN
            FOR m := 0 TO FoundModules.NumMod DO
                IF FoundModules.Mod[m].SN = Module.Serial_Num THEN
                    NewPath := 1;
                    FOR h := 0 TO 24 DO
                        IF FoundModules.Mod[m].PATH[h] = MSG1.Path THEN
                            NewPath := 0;
                        END_IF;
                    END_FOR;
                    IF NewPath AND FoundModules.Mod[m].NumPaths < 24 THEN
                        NumPaths := FoundModules.Mod[m].NumPaths;
                        INSERT(FoundModules.Mod[m].Path[NumPaths],MSG1.Path,1,
                            FoundModules.MOD[m].Path[NumPaths]);
                        FoundModules.Mod[m].NumPaths := NumPaths + 1;
                    END_IF;
                    NewMod := 0;
                END_IF;
            END_FOR;
            IF NewMod AND FoundModules.NumMod < 100 THEN
                FoundModules.Mod[FoundModules.NumMod].SN := Module.Serial_Num;
                // Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
                NumPaths := FoundModules.Mod[FoundModules.NumMod].NumPaths;
                INSERT(FoundModules.Mod[FoundModules.NumMod].Path[NumPaths],MSG1.Path,1,▼
                                                                FoundModules.Mod[FoundModules.
                NumMod].Path[NumPaths]);
                FoundModules.Mod[FoundModules.NumMod].NumPaths := NumPaths + 1;
                FoundModules.NumMod := FoundModules.NumMod + 1;
            END_IF;
        END_IF;
        // GoTo next step
        Step := Step + 1;

    6:  // Increment node counter 'i'
        i := i + 1;
        // Check 'i'
        IF i <= ScanRange.Max THEN
            Step := 1;
        ELSE
            i := 0;
            Step := 7;
        END_IF;

    7:  // DONE scanning CNet; STOP
        ONS3 := 0;
        Step := 7;

    ELSE
        // If in non-defined step, GoTo first step
        Step := 1;

END_CASE;
```

APPENDIX B.

ETHERNET SCAN ROUTINE

```
// !!!: DONT TRY TO SCAN IP ENDING IN 0 :!!!
IF i = 0 OR NOT ONS3 THEN
    i := ScanRange.Min;
    ONS3 := 1;
END_IF;
CASE Step OF
    1:  //Disable MSG1 (for good measure) while chaning path str
        MSG1.EN := 0;
        // Set message path to backplane to ENet addr 131.151.52.i
        DELETE(MSG1.Path,82,1,MSG1.Path);
        INSERT(MSG1.Path,Path_Net,1,MSG1.Path);

        IF MSG1.path.len > 69 THEN
            Step := 7;
        ELSE
            MSG1.Path.Data[MSG1.path.len] := 18; // ITS JUST WHAT MSG DIALOG DOES FOR ETHERNET
            // Make sure to clear old IP string (Not sure if needed)
            DELETE(IP.STR,82,1,IP.STR);
            IP.octet[3] := i;

            // Convert DINT octets of IP to single STRING (could be converted to ADD-ON for tiddyness)
            FOR h := 0 TO 3 DO
                // Convert each Octet in a sting
                DTOS(IP.octet[h],TempStr);
                // Concatinate this with the IP string
                CONCAT(IP.STR,TempStr,IP.STR);
                // Add '.' between OCTETs in IP (only between 0-1, 1-2 and 2-3 octets)
                IF h <> 3 THEN
                    IP.STR.DATA[IP.STR.LEN] := 46; // Add '.' after octet (46 = '.' in ASCII ; LEN points to last element+1)
                    IP.STR.LEN := IP.STR.LEN + 1; // INC String length since added '.'
                END_IF;
            END_FOR;

            MSG1.Path.Data[MSG1.path.len+1] := IP.STR.LEN; // Length of the IP address imbeded in the path string
            MSG1.path.len := MSG1.path.len + 2;
            // Concatinate destination IP to Path string (should take care of setting correct path string length)
            CONCAT(MSG1.Path,IP.STR,MSG1.path);
            // Clear DN and ER bits to make sure ready for new scan.
            MSG1.DN := 0;
            MSG1.ER := 0;
            Step := Step + 1;
        END_IF;

2:  //Send Message for "Module WHO" to ENet address in IP.str (131.151.52.i in this example)
    MSG(MSG1);
    Step := Step + 1;

3:  IF MSG1.DN THEN
        // Clear DN bit
        MSG1.DN := 0;
        // GoTo next step
        Step := Step + 1;
    ELSIF MSG1.ER THEN
        // Clear ER bit
        MSG1.ER := 0;
        // Skip to next slot
        Step := 6;
    END_IF;

4:  // Decode RAW data from MSG
    Decode_Module_Who(DecodeModWho1,Module_Raw,Module);
    // GoTo next step
    Step := Step + 1;

5:  // Copy new info for node 'i' to UDT
    IF Module.Product_Type = 12 AND Mod_SN <> Module.Serial_Num THEN
        NewMod := 1;
        FOR m := 0 TO FoundModules.NumComm DO
            IF FoundModules.Comm[m].SN = Module.Serial_Num THEN
                NewPath := 1;
                FOR h := 0 TO 24 DO
                    IF FoundModules.Comm[m].PATH[h] = MSG1.Path THEN
                        NewPath := 0;
                    END_IF;
                END_FOR;
                IF NewPath AND FoundModules.Comm[m].NumPaths < 24 THEN
                    NumPaths := FoundModules.Comm[m].NumPaths;
                    INSERT(FoundModules.Comm[m].Path[NumPaths],MSG1.Path,1,
                                                                FoundModules.Comm[m].Path[NumPaths]);
                    FoundModules.Comm[m].NetScanned[NumPaths] := 1;
                    FoundModules.Comm[m].NumPaths := NumPaths + 1;
                END_IF;
                NewMod := 0;
            END_IF;
        END_FOR;
        IF NewMod AND FoundModules.NumComm < 100 THEN
FoundModules.Comm[FoundModules.NumComm].SN := Module.Serial_Num;
FoundModules.Comm[FoundModules.NumComm].Type := Module.Product_Code;
// Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
NumPaths := FoundModules.Comm[FoundModules.NumComm].NumPaths;
INSERT(FoundModules.Comm[FoundModules.NumComm].Path[NumPaths],MSG1.Path,1,
                    FoundModules.Comm[FoundModules.NumComm].Path[NumPaths]);
FoundModules.Comm[FoundModules.NumComm].BPScanned[0] := 0;
FoundModules.Comm[FoundModules.NumComm].NetScanned[NumPaths] := 1;
FoundModules.Comm[FoundModules.NumComm].NumPaths := NumPaths + 1;
FoundModules.NumComm := FoundModules.NumComm + 1;
        END_IF;
    ELSIF Module.product_type <> 12 AND Mod_SN <> Module.Serial_Num THEN
        FOR m := 0 TO FoundModules.NumMod DO
            IF FoundModules.Mod[m].SN = Module.Serial_Num THEN
                NewPath := 1;
                FOR h := 0 TO 24 DO
                    IF FoundModules.Mod[m].PATH[h] = MSG1.Path THEN
                        NewPath := 0;
                    END_IF;
                END_FOR;
                IF NewPath AND FoundModules.Mod[m].NumPaths < 24 THEN
NumPaths := FoundModules.Mod[m].NumPaths;
INSERT(FoundModules.Mod[m].Path[NumPaths],MSG1.Path,1,
                                FoundModules.MOD[m].Path[NumPaths]);
FoundModules.Mod[m].NumPaths := NumPaths + 1;
                END_IF;
                NewMod := 0;
            END_IF;
        END_FOR;
        IF NewMod AND FoundModules.NumMod < 100 THEN
FoundModules.Mod[FoundModules.NumMod].SN := Module.Serial_Num;
// Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
NumPaths := FoundModules.Mod[FoundModules.NumMod].NumPaths;
INSERT(FoundModules.Mod[FoundModules.NumMod].Path[NumPaths],MSG1.Path,1,▼
                                        FoundModules.Mod[FoundModules.NumMod].Path[NumPaths]);
FoundModules.Mod[FoundModules.NumMod].NumPaths := NumPaths + 1;
FoundModules.NumMod := FoundModules.NumMod + 1;
        END_IF;
    END_IF;
    // GoTo next step
    Step := Step + 1;

6:  // Increment node counter 'i'
    i := i + 1;
    // Check 'i'
    IF i <= ScanRange.Max THEN
        Step := 1;
    ELSE
        i := 0;
        Step := 7;
    END_IF;

7:  // DONE scanning ENet; STOP
    ONS3 := 0;
    Step := 7;

ELSE
    // If in non-defined step, GoTo first step
    Step := 1;

END_CASE;
```

APPENDIX C.

DEVICENET SCAN ROUTINE

```
IF NOT ONS3 THEN
    i := ScanRange.Min;
    ONS3 := 1;
END_IF;

CASE Step OF
    1:  // Set message path to backplane to slot 'i'
        DELETE(SN_MSG.Path,82,1,SN_MSG.Path);
        INSERT(SN_MSG.Path,Path_Net,1,SN_MSG.Path);
        IF MSG1.path.len > 84 THEN
            Step := 7;
        ELSE
SN_MSG.Path.Data[SN_MSG.path.len] := 2; // Go out on DeviceNet
SN_MSG.Path.Data[SN_MSG.path.len+1] := i; // to address 'i'
SN_MSG.path.len := SN_MSG.path.len + 2;
// Clear DN and ER bits to make sure ready for new scan.
SN_MSG.DN := 0;
SN_MSG.ER := 0;
// Prepare "DeviceWho" MSG with same path as "SerialNumber" MSG
DELETE(MSG1.Path,82,1,MSG1.Path);
INSERT(MSG1.Path,SN_MSG.path,1,MSG1.Path);
// Clear DN and ER bits to make sure ready for new scan.
MSG1.DN := 0;
MSG1.ER := 0;
Step := Step + 1;
        END_IF;

    2:  //Send Message for "Module WHO" to CNet Node 'i'
        MSG(SN_MSG);
        MSG(MSG1);
        IF SN_MSG.EN AND MSG1.EN THEN
            Step := Step + 1;
        END_IF;

    3:  IF SN_MSG.DN AND MSG1.ER THEN
            // Clear DN and ER bits
            SN_MSG.DN := 0;
            MSG1.ER := 0;
            // GoTo step 4 (add NonComm module to UDT)
            Step := 4;
        ELSIF SN_MSG.DN and MSG1.DN THEN
            // Clear DN and ER bits
            SN_MSG.DN := 0;
            MSG1.DN := 0;
            // GoTo step 5 (add Comm module to UDT)
            Step := 5;
        ELSIF SN_MSG.ER THEN
            // Clear bits
            SN_MSG.ER := 0;
            MSG1.DN := 0;
            MSG1.ER := 0;
            // Skip to next address
            Step := 7;
        END_IF;

4:  // Add NonComm module to UDT
    FOR m := 0 TO FoundModules.NumMod DO
        IF FoundModules.Mod[m].SN = Received_SN THEN
            NewPath := 1;
            FOR h := 0 TO 24 DO
                IF FoundModules.Mod[m].PATH[h] = SN_MSG.Path THEN
                    NewPath := 0;
                END_IF;
            END_FOR;
            IF NewPath AND FoundModules.Mod[m].NumPaths < 24 THEN
                NumPaths := FoundModules.Mod[m].NumPaths;
                INSERT(FoundModules.Mod[m].Path[NumPaths],SN_MSG.Path,1,▼
                                                        FoundModules.MOD[m].Path[NumPaths]);
                FoundModules.Mod[m].NumPaths := NumPaths + 1;
            END_IF;
            NewMod := 0;
        END_IF;
    END_FOR;
    IF NewMod AND FoundModules.NumMod < 100 THEN
        FoundModules.Mod[FoundModules.NumMod].SN := Received_SN;
        // Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
        NumPaths := FoundModules.Mod[FoundModules.NumMod].NumPaths;
        INSERT(FoundModules.Mod[FoundModules.NumMod].Path[NumPaths],SN_MSG.Path,1,▼
                                                FoundModules.Mod[FoundModules.NumMod].Path[NumPaths]);
        FoundModules.Mod[FoundModules.NumMod].NumPaths := NumPaths + 1;
        FoundModules.NumMod := FoundModules.NumMod + 1;
    END_IF;
    // goto step 7 and skip CommMod add
    Step := 7;

    5:  // Decode RAW data from MSG
        Decode_Module_Who(DecodeModWho1,Module_Raw,Module);
        // GoTo next step
        Step := Step + 1;

    6:  // Check if current slot contains my processor
        IF Module.Product_Type = 12 AND Mod_SN <> Module.Serial_Num THEN
            NewMod := 1;
            FOR m := 0 TO FoundModules.NumComm DO
                IF FoundModules.Comm[m].SN = Module.Serial_Num THEN
                    NewPath := 1;
                    FOR h := 0 TO 24 DO
                        IF FoundModules.Comm[m].PATH[h] = MSG1.Path THEN
                            NewPath := 0;
                        END_IF;
                    END_FOR;
                    IF NewPath AND FoundModules.Comm[m].NumPaths < 24 THEN
                        NumPaths := FoundModules.Comm[m].NumPaths;
                        INSERT(FoundModules.Comm[m].Path[NumPaths],MSG1.Path,1,▼
                                                                FoundModules.Comm[m].Path[NumPaths]);
                        FoundModules.Comm[m].NetScanned[NumPaths] := 1;
                        FoundModules.Comm[m].NumPaths := NumPaths + 1;
                    END_IF;
                    NewMod := 0;
                END_IF;
            END_FOR;
            IF NewMod THEN
FoundModules.Comm[FoundModules.NumComm].SN := Module.Serial_Num;
FoundModules.Comm[FoundModules.NumComm].Type := Module.Product_Code;
// Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
NumPaths := FoundModules.Comm[FoundModules.NumComm].NumPaths;
INSERT(FoundModules.Comm[FoundModules.NumComm].Path[NumPaths],MSG1.Path,1,▼
                                                                FoundModules.Comm[FoundModules.
NumComm].Path[NumPaths]);
FoundModules.Comm[FoundModules.NumComm].BPScanned[0] := 0;
FoundModules.Comm[FoundModules.NumComm].NetScanned[NumPaths] := 1;
FoundModules.Comm[FoundModules.NumComm].NumPaths := NumPaths + 1;
FoundModules.NumComm := FoundModules.NumComm + 1;
            END_IF;
        ELSIF Module.product_type <> 12 AND Mod_SN <> Module.Serial_Num THEN
            FOR m := 0 TO FoundModules.NumMod DO
                IF FoundModules.Mod[m].SN = Module.Serial_Num THEN
                    NewPath := 1;
                    FOR h := 0 TO 24 DO
                        IF FoundModules.Mod[m].PATH[h] = MSG1.Path THEN
                            NewPath := 0;
                        END_IF;
                    END_FOR;
                    IF NewPath AND FoundModules.Mod[m].NumPaths < 24 THEN
                        NumPaths := FoundModules.Mod[m].NumPaths;
                        INSERT(FoundModules.Mod[m].Path[NumPaths],SN_MSG.Path,1,▼
                                                                FoundModules.MOD[m].Path[NumPaths]);
                        FoundModules.Mod[m].NumPaths := NumPaths + 1;
                    END_IF;
                    NewMod := 0;
                END_IF;
            END_FOR;
            IF NewMod THEN
                FoundModules.Mod[FoundModules.NumMod].SN := Module.Serial_Num;
                // Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
                NumPaths := FoundModules.Mod[FoundModules.NumMod].NumPaths;
                INSERT(FoundModules.Mod[FoundModules.NumMod].Path[NumPaths],SN_MSG.Path,1,▼
                                                                FoundModules.Mod[FoundModules.
                NumMod].Path[NumPaths]);
                FoundModules.Mod[FoundModules.NumMod].NumPaths := NumPaths + 1;
                FoundModules.NumMod := FoundModules.NumMod + 1;
            END_IF;
        END_IF;
        // GoTo next step
        Step := Step + 1;

    7:  // Increment node counter 'i'
        i := i + 1;
        // Check 'i'
        IF i <= ScanRange.Max THEN
            Step := 1;
        ELSE
            i := 0;
            Step := 8;
        END_IF;

8:  // DONE scanning CNet; STOP
    ONS3 := 0;
    Step := 8;

ELSE
    // If in non-defined step, GoTo first step
    Step := 1;

END_CASE;
```

APPENDIX D.

BACKPLANE SCAN ROUTINE

```
CASE Step OF
    1:  // Set message path to backplane to slot 'i'
        DELETE(MSG1.Path,82,1,MSG1.Path);
        INSERT(MSG1.Path,Path_BP,1,MSG1.Path);
        IF MSG1.path.len > 84 THEN
            Step := 7;
        ELSE
            MSG1.Path.Data[MSG1.path.len] := 1;
            MSG1.Path.Data[MSG1.path.len+1] := i;
            MSG1.path.len := MSG1.path.len + 2;
            // Clear DN and ER bits to make sure ready for new scan.
            MSG1.DN := 0;
            MSG1.ER := 0;
            Step := Step + 1;
        END_IF;

    2:  //Send Message for "Module WHO" to slot 'i'
        MSG(MSG1);
        //If MSG1 is enabled, goto next step (Should always be 1)
        IF MSG1.EN THEN
            Step := Step + 1;
        END_IF;

    3:  IF MSG1.DN THEN
            // Clear DN bit
            MSG1.DN := 0;
            // GoTo next step
            Step := Step + 1;
        ELSIF MSG1.ER THEN
            // Clear ER bit
            MSG1.ER := 0;
            // Skip to next slot
            Step := 7;
        END_IF;

    4:  // Decode RAW data from MSG
        Decode_Module_Who(DecodeModWho1,Module_Raw,Module);
        // GoTo next step
        Step := Step + 1;

    5:  // Check if current slot contains my processor
        NewMod := 1;
        IF FoundModules.NumMod >= FoundModules.NumComm THEN
            LEN := FoundModules.NumMod;
        ELSE
            LEN := FoundModules.NumComm;
        END_IF;
        FOR m := 0 TO LEN DO
          IF FoundModules.Comm[m].SN = Module.Serial_Num OR FoundModules.Mod[m].SN = Module.Serial_Num THEN
                NewPath := 1;
                FOR h := 0 TO 24 DO
                    IF Module.Product_Type = 12 AND FoundModules.Comm[m].PATH[h] = MSG1.Path THEN
                        NewPath := 0;
                    ELSIF Module.Product_Type <> 12 AND FoundModules.Mod[m].PATH[h] = MSG1.Path THEN
                        NewPath := 0;
                    END_IF;
                END_FOR;
                IF NewPath AND Module.Product_Type = 12 AND FoundModules.Comm[m].NumPaths < 24 THEN
                    NumPaths := FoundModules.Comm[m].NumPaths;
                    INSERT(FoundModules.Comm[m].Path[NumPaths],MSG1.Path,1,▼
                                                    FoundModules.Comm[m].Path[NumPaths]);
                    FoundModules.Comm[m].BPScanned[NumPaths] := 1;
                    FoundModules.Comm[m].NumPaths := NumPaths + 1;
                ELSIF NewPath AND Module.Product_Type <> 12 AND FoundModules.Mod[m].NumPaths < 24 THEN
                    NumPaths := FoundModules.Mod[m].NumPaths;
                    INSERT(FoundModules.Mod[m].Path[NumPaths],MSG1.Path,1,▼
                                                    FoundModules.MOD[m].Path[NumPaths]);
                    FoundModules.Mod[m].NumPaths := NumPaths + 1;
                END_IF;
                NewMod := 0;
            END_IF;
        END_FOR;
        IF NewMod AND Module.Product_Type = 12 AND FoundModules.NumComm < 100 THEN
            FoundModules.Comm[FoundModules.NumComm].SN := Module.Serial_Num;
            FoundModules.Comm[FoundModules.NumComm].Type := Module.Product_Code;
            // Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
            NumPaths := FoundModules.Comm[FoundModules.NumComm].NumPaths;
            INSERT(FoundModules.Comm[FoundModules.NumComm].Path[NumPaths],MSG1.Path,1,▼
                                    FoundModules.Comm[FoundModules.NumComm].Path[NumPaths]);
            FoundModules.Comm[FoundModules.NumComm].BPScanned[0] := 1;
            FoundModules.Comm[FoundModules.NumComm].NumPaths := NumPaths + 1;
            FoundModules.NumComm := FoundModules.NumComm + 1;
        ELSIF NewMod AND Module.Product_Type <> 12 AND FoundModules.NumMod < 100 THEN
            FoundModules.Mod[FoundModules.NumMod].SN := Module.Serial_Num;
            // Must pull out variable NumPaths; For some reason INSEART did not like it in the indices
            NumPaths := FoundModules.Mod[FoundModules.NumMod].NumPaths;
            INSERT(FoundModules.Mod[FoundModules.NumMod].Path[NumPaths],MSG1.Path,1,▼
                                        FoundModules.Mod[FoundModules.NumMod].Path[NumPaths]);
            FoundModules.Mod[FoundModules.NumMod].NumPaths := NumPaths + 1;
            FoundModules.NumMod := FoundModules.NumMod + 1;
        END_IF;
        // GoTo next step
        Step := Step + 1;

    6:  // Copy new info for slot 'i' to UDT
        Step := Step + 1;

    7:  // Increment slot counter 'i'
        i := i + 1;
        // Check 'i'
        IF i <= 16 THEN
            Step := 1;
        ELSE
            i := 0;
            Step := 8;
        END_IF;

    8:  // DONE scanning rack; STOP
        Step := 8;

    ELSE
        // If in non-defined step, GoTo first step
        Step := 1;
END_CASE;
```

APPENDIX E.

MAIN NETWORK SCAN ROUTINE

```
// On first startup, scan own Backplane
IF NOT FS THEN
    // Set path to scan backplane (if Path_BP is empty, routine will just scan own backplane)
    DELETE(Path_BP,82,1,Path_BP);
    // Jump to backplane scanning routine
    JSR(Scan_BP);
    // If in Step 8 then scanning complete
    IF Step = 8 THEN
        FS := 1;
        FoundMods := FoundModules.NumComm;
    END_IF;
ELSE
    CASE ScanStep OF
        0:  n := 0;
            A := 0;
            ScanStep := ScanStep + 1;

        1:  // Check for scannable network and scan unscanned paths
            CASE FoundModules.Comm[n].Type OF
                7:  //ControlNet type of network; SCANNABLE
                    IF NOT FoundModules.Comm[n].NetScanned[A] THEN
                        IF NOT ONS2 THEN
                            DELETE(Path_Net,82,1,Path_Net);
                            INSERT(Path_Net,FoundModules.Comm[n].Path[A],1,Path_Net);
                            Step := 1;
                            ONS2 := 1;
                            Mod_SN := FoundModules.Comm[n].SN;
                        END_IF;
                        // Scan ControlNet in the range [0,5]
                        ScanRange.Min := 0;
                        ScanRange.Max := 5;
                        JSR(Scan_CNet);
                    END_IF;
                    IF Step = 7 OR FoundModules.Comm[n].NetScanned[A] THEN
                        ONS2 := 0;
                        FoundModules.Comm[n].NetScanned[A] := 1;
                        A := A + 1;
                    END_IF;
                    IF A >= FoundModules.Comm[n].NumPaths THEN
                        ScanStep := ScanStep + 1;
                        A := 0;
                    END_IF;

                14: //Scan DeviceNet (Product_Code = 12)
                    IF NOT FoundModules.Comm[n].NetScanned[A] THEN
                        IF NOT ONS2 THEN
                            DELETE(Path_Net,82,1,Path_Net);
                            INSERT(Path_Net,FoundModules.Comm[n].Path[A],1,Path_Net);
                            Step := 1;
                            ONS2 := 1;
                            Mod_SN := FoundModules.Comm[n].SN;
                        END_IF;
                        // Scan DeviceNet in the range [0,63]
                        ScanRange.Min := 0;
                        ScanRange.Max := 63;
                        JSR(Scan_DNet);
                    END_IF;
                    IF Step = 8 OR FoundModules.Comm[n].NetScanned[A] THEN
                        ONS2 := 0;
                        FoundModules.Comm[n].NetScanned[A] := 1;
                        A := A + 1;
                    END_IF;
                    IF A >= FoundModules.Comm[n].NumPaths THEN
                        ScanStep := ScanStep + 1;
                        A := 0;
                    END_IF;

                58,125:  // Scan ENBT and EWEB modules.
                    IF NOT FoundModules.Comm[n].NetScanned[A] THEN
                        IF NOT ONS2 THEN
                    DELETE(Path_Net,82,1,Path_Net);
                    INSERT(Path_Net,FoundModules.Comm[n].Path[A],1,Path_Net);
                    Step := 1;
                    ONS2 := 1;
                    Mod_SN := FoundModules.Comm[n].SN;
                END_IF;
                // Set IP to scan (Octets [0,2] could also be varied for scanning network)
                IP.OCTET[0] := 131;
                IP.OCTET[1] := 151;
                IP.OCTET[2] := 52;
                // Scan Octet3 in the range [129,151]
                ScanRange.Min := 129;
                ScanRange.Max := 151;
                JSR(Scan_ENet);
            END_IF;
            IF Step = 7 OR FoundModules.Comm[n].NetScanned[A] THEN
                ONS2 := 0;
                FoundModules.Comm[n].NetScanned[A] := 1;
                A := A + 1;
            END_IF;
            IF A >= FoundModules.Comm[n].NumPaths THEN
                ScanStep := ScanStep + 1;
                A := 0;
            END_IF;

        ELSE
            // If unscannable network (i.e not in above list) then skip to next comm module.
            ScanStep := 3;

    END_CASE;

2:  //IF BPSCANNED[A] bit NOT set then need to scan that modules backplane.
    IF NOT FoundModules.Comm[n].BPScanned[A] THEN
        IF NOT ONS1 THEN
            DELETE(Path_BP,82,1,Path_BP);
            INSERT(Path_BP,FoundModules.Comm[n].Path[A],1,Path_BP);
            Step := 1;
            ONS1 := 1;
        END_IF;
        JSR(Scan_BP);
    END_IF;
    IF Step = 8 OR FoundModules.Comm[n].BPScanned[A] THEN
        ONS1 := 0;
        FoundModules.Comm[n].BPScanned[A] := 1;
        A := A + 1;
    END_IF;
    IF A >= FoundModules.Comm[n].NumPaths THEN
        ScanStep := ScanStep + 1;
        A := 0;
    END_IF;

3:  // Inc n, check limits and goto 1 if n ok
    n := n + 1;
    RestartFlag := 0;
    IF n = FoundModules.NumComm THEN
        FOR j := 0 TO (FoundModules.NumComm - 1) DO
            CASE FoundModules.Comm[j].type OF
                7,14,58,125:
                    FOR z := 0 TO (FoundModules.Comm[j].NumPaths - 1) DO
                        IF NOT FoundModules.Comm[j].Netscanned[z] OR NOT
                                FoundModules.Comm[j].BPscanned[z] THEN
                            RestartFlag := 1;
                        END_IF;
                    END_FOR;
            END_CASE;
        END_FOR;
        IF RestartFlag AND RestartCTR < 3 THEN
            ScanStep := 0;
            RestartCTR := RestartCTR + 1;
        ELSE
```

```
                    ScanStep := 4;
                END_IF;
            ELSE
                ScanStep := 1;
            END_IF;

        4:  // Validate all paths found.
            IF NOT ONS6 THEN
                V_DN := 0;
                V_STEP := 0;
                V_TOGGLE := 0;
                ONS6 := 1;
            END_IF;
            JSR(Validate_Paths);
            IF V_DN THEN
                ScanStep := 5;
                V_DN := 0;
                ONS6 := 0;
            END_IF;

        5:  // DONE.
            ScanStep := 5;

        ELSE
            ScanStep := 0;

    END_CASE;
END_IF;
```

APPENDIX F.

PATH VERIFICATION ROUTINE

```
IF NOT V_TOGGLE AND NOT V_DN THEN //Check Non-Comm module paths.
    CASE V_STEP OF
        0:  // Reset pointers
            V_IDX1 := 0;
            V_IDX2 := 0;
            V_STEP := 1;

        1:  // Set paths
            DELETE(MSG1.Path,82,1,MSG1.Path);
            INSERT(MSG1.Path,FoundModules.Mod[V_IDX1].Path[V_IDX2],1,MSG1.Path);
            DELETE(SN_MSG.Path,82,1,SN_MSG.Path);
            INSERT(SN_MSG.Path,FoundModules.Mod[V_IDX1].Path[V_IDX2],1,SN_MSG.Path);
            V_STEP := V_STEP + 1;

        2:  // Set message for 'Device_Who' and 'SerialNumber' to enable.
            MSG(MSG1);
            MSG(SN_MSG);
            IF MSG1.EN AND SN_MSG.EN THEN
                V_STEP := V_STEP + 1;
            END_IF;

        3:  //Check to see if messages are done or error
            IF MSG1.DN OR SN_MSG.DN THEN
                FoundModules.Mod[V_IDX1].ValidPath[V_IDX2] := 1;
                V_STEP := 4;
            ELSIF MSG1.ER AND SN_MSG.ER THEN
                FoundModules.Mod[V_IDX1].ValidPath[V_IDX2] := 0;
                V_STEP := 4;
            END_IF;

        4:  // Increment path pointer
            V_IDX2 := V_IDX2 + 1;
            IF V_IDX2 = FoundModules.Mod[V_IDX1].NumPaths THEN
                V_IDX2 := 0;
                V_STEP := 5;
            ELSE
                V_STEP := 1;
            END_IF;

        5:  // Increment module pointer
            V_IDX1 := V_IDX1 + 1;
            IF V_IDX1 = FoundModules.NumMod THEN
                V_IDX1 := 0;
                V_STEP := 6;
            ELSE
                V_STEP := 1;
            END_IF;

        6:  // DONE
            V_STEP := 0;
            V_TOGGLE := 1;

        ELSE
            V_STEP := 0;

    END_CASE;

ELSIF V_TOGGLE AND NOT V_DN THEN //Check comm module paths.

    CASE V_STEP OF
        0:  // Reset pointers
            V_IDX1 := 0;
            V_IDX2 := 0;
            V_STEP := 1;

        1:  // Set paths
            DELETE(MSG1.Path,82,1,MSG1.Path);
            INSERT(MSG1.Path,FoundModules.Comm[V_IDX1].Path[V_IDX2],1,MSG1.Path);
            DELETE(SN_MSG.Path,82,1,SN_MSG.Path);
            INSERT(SN_MSG.Path,FoundModules.Comm[V_IDX1].Path[V_IDX2],1,SN_MSG.Path);
            V_STEP := V_STEP + 1;
        2:   // Set message for 'Device_Who' and 'SerialNumber' to enable.
            MSG(MSG1);
            MSG(SN_MSG);
            IF MSG1.EN AND SN_MSG.EN THEN
                V_STEP := V_STEP + 1;
            END_IF;

        3:  //Check to see if messages are done or error
            IF MSG1.DN OR SN_MSG.DN THEN
                FoundModules.Comm[V_IDX1].ValidPath[V_IDX2] := 1;
                V_STEP := 4;
            ELSIF MSG1.ER AND SN_MSG.ER THEN
                FoundModules.Comm[V_IDX1].ValidPath[V_IDX2] := 0;
                V_STEP := 4;
            END_IF;

        4:  // Increment path pointer and set step
            V_IDX2 := V_IDX2 + 1;
            IF V_IDX2 = FoundModules.Comm[V_IDX1].NumPaths THEN
                V_IDX2 := 0;
                V_STEP := 5;
            ELSE
                V_STEP := 1;
            END_IF;

        5:  // Increment module pointer
            V_IDX1 := V_IDX1 + 1;
            IF V_IDX1 = FoundModules.NumComm THEN
                V_IDX1 := 0;
                V_STEP := 6;
            ELSE
                V_STEP := 1;
            END_IF;

        6:  // DONE
            V_IDX1 := 0;
            V_IDX2 := 0;
            V_STEP := 0;
            V_TOGGLE := 0;
            V_DN := 1;

        ELSE
            V_STEP := 0;

    END_CASE;
END_IF;
```

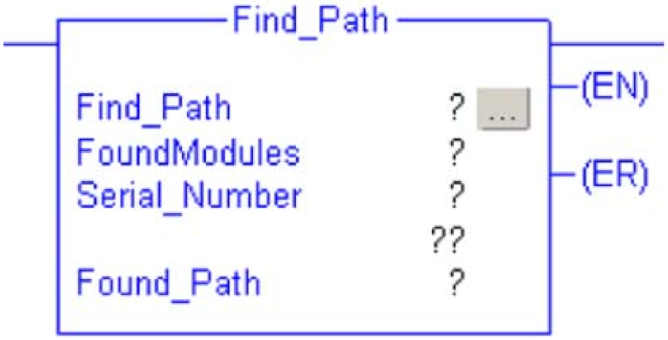
APPENDIX G.

FIND PATH ADD-ON INSTRUCTION

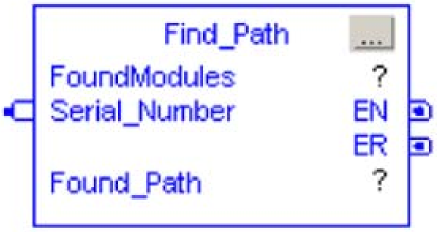
Find_Path v1.0

Available Languages

Relay Ladder



Function Block



Structured Text

Find_Path(FoundModules, Serial_Number, Found_Path);

Parameters

Required	Name	Data Type	Usage	Description
X	Find_Path	Find_Path	InOut	
	EnableIn	BOOL	Input	
	EnableOut	BOOL	Output	
X	FoundModules	Found_Modules	InOut	
X	Serial_Number	DINT	Input	
	EN	BOOL	Output	
	ER	BOOL	Output	
X	Found_Path	STRING86	InOut	
	DN	BOOL	Output	

CODE:

```
flag := 0;
FoundMod := 0;

FOR idx := 0 TO 99 DO
    IF FoundModules.Mod[idx].SN = Serial_Number THEN
        flag := 1;
        FoundMod := idx;
        idx := 100;
    END_IF;
END_FOR;

IF NOT flag THEN
    FOR idx := 0 TO 99 DO
        IF FoundModules.Comm[idx].SN = Serial_Number THEN
            flag := 1;
            FoundMod := idx;
            idx := 100;
        END_IF;
    END_FOR;
ELSE
    DN := 1;
END_IF;

IF DN AND flag THEN
    FOR idx := 0 TO (FoundModules.Mod[FoundMod].NumPaths - 1) DO
        IF FoundModules.Mod[FoundMod].ValidPath[idx] THEN
            Insert(FoundModules.Mod[FoundMod].Path[idx],Found_Path,1,Found_Path);
            idx := FoundModules.Mod[FoundMod].NumPaths;
        END_IF;
    END_FOR;
    DN := 1;
    ER := 0;
ELSIF NOT DN AND flag THEN
    FOR idx := 0 TO (FoundModules.Comm[FoundMod].NumPaths - 1) DO
        IF FoundModules.Comm[FoundMod].ValidPath[idx] THEN
            Insert(FoundModules.Comm[FoundMod].Path[idx],Found_Path,1,Found_Path);
            idx := FoundModules.Comm[FoundMod].NumPaths;
        END_IF;
    END_FOR;
    DN := 1;
    ER := 0;
ELSE
    DN := 0;
    ER := 1;
END_IF;
```

APPENDIX H.

MICROSOFT EXCEL VISUAL BASIC MACRO

```
Sub Get_Paths()
    Dim DDE_Topic As String
    Dim TempAry(90) As Byte
    Dim TestSTR As String
    Dim RowCtr As Long
    Dim H As Byte
    Dim M As Byte
    Dim DelALL
    Dim SerialNumber As Long
    Dim wksht As Worksheet
    Dim i As Long

    For i = 1 To ActiveWorkbook.Worksheets.Count Step 1
        Worksheets(i).Activate
        DDE_Topic = Worksheets(i).Name

        If i = 1 Then
            Set DelALL = Range("A3:C2500")
            DelALL.Delete
            Range("A3").Value = "Serial Number:"
            Range("B3").Value = "Paths:"
            Range("C3").Value = "Valid:"
            Range("D3").Value = "BP Scan:"
            Range("E3").Value = "Net Scan:"
            RowCtr = 4
        Else
            Set DelALL = Range("A1:C2500")
            DelALL.Delete
            Range("A1").Value = "Serial Number:"
            Range("B1").Value = "Paths:"
            Range("C1").Value = "Valid:"
            Range("D1").Value = "BP Scan:"
            Range("E1").Value = "Net Scan:"
            RowCtr = 2
        End If

        Channel = DDEInitiate("RSLinx", DDE_Topic)

        NumMod = DDERequest(Channel, "FoundModules.NumMod")
        NumComm = DDERequest(Channel, "FoundModules.NumComm")

        With WorksheetFunction

        RowCtr_STR = Trim(.Substitute(Str(RowCtr), " ", ""))
        Range("A" + RowCtr_STR).Value = "Non-Comm Modules:"
        RowCtr = RowCtr + 1

        For H = 0 To NumMod(1) - 1 Step 1
            H_STR = Trim(.Substitute(Str(H), " ", ""))
            SN = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].SN")
            RowCtr_STR = Trim(.Substitute(Str(RowCtr), " ", ""))
            SerialNumber = SN(1)
            Range("A" + RowCtr_STR).Value = "0h" + Hex8(SerialNumber)
            RowCtr = RowCtr + 1
            NumPaths = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].NumPaths")
            For M = 0 To NumPaths(1) - 1 Step 1
                M_STR = Trim(.Substitute(Str(M), " ", ""))
                ValidPath = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].ValidPath[" + M_STR + "]")
                BPScanned = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].BPScanned[" + M_STR + "]")
                NetScanned = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].NetScanned[" + M_STR + "]")
                RowCtr_STR = Trim(.Substitute(Str(RowCtr), " ", ""))

                Range("C" + RowCtr_STR).Value = ValidPath(1)
                Range("D" + RowCtr_STR).Value = "N/A"
                Range("E" + RowCtr_STR).Value = "N/A"

                TestSTR = ""

                StrLEN = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].Path[" + M_STR + "].Len")
        For idx = 0 To StrLEN(1) Step 1
            IDX_STR = Trim(.Substitute(Str(idx), " ", ""))
            temp = DDERequest(Channel, "FoundModules.Mod[" + H_STR + "].Path[" + M_STR + "].DATA["
                                                                                   + IDX_STR + "]")
            TempAry(idx) = temp(1)
        Next idx
        For idx = 0 To StrLEN(1) - 1 Step 1
            If idx <> 0 Then
                If TempAry(idx) = 18 And TempAry(idx - 1) <> 2 Then
                    TestSTR = TestSTR + "2, "
                    idx = idx + 2
                    temp = idx + TempAry(idx - 1)
                    While idx < temp
                        TestSTR = TestSTR + Chr(TempAry(idx))
                        idx = idx + 1
                    Wend
                    idx = idx - 1
                    TestSTR = TestSTR + ", "
                Else
                    TestSTR = TestSTR + Trim(.Substitute(Str(TempAry(idx)), " ", ""))
                    If idx <> (StrLEN(1) - 1) Then
                        TestSTR = TestSTR + ", "
                    End If
                End If
            Else
                TestSTR = TestSTR + Trim(.Substitute(Str(TempAry(idx)), " ", "")) + ", "
            End If
        Next idx

        Range("B" + Trim(.Substitute(Str(RowCtr), " ", ""))).Value2 = ""
        Range("B" + Trim(.Substitute(Str(RowCtr), " ", ""))).Value2 = TestSTR
        RowCtr = RowCtr + 1
    Next M
    RowCtr = RowCtr + 1
Next H

RowCtr_STR = Trim(.Substitute(Str(RowCtr), " ", ""))
Range("A" + RowCtr_STR).Value = "Comm Modules:"
RowCtr = RowCtr + 1

For H = 0 To NumComm(1) - 1 Step 1
    H_STR = Trim(.Substitute(Str(H), " ", ""))
    SN = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].SN")
    RowCtr_STR = Trim(.Substitute(Str(RowCtr), " ", ""))
    SerialNumber = SN(1)
    Range("A" + RowCtr_STR).Value = "0h" + Hex8(SerialNumber)
    RowCtr = RowCtr + 1
    NumPaths = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].NumPaths")

    For M = 0 To NumPaths(1) - 1 Step 1
        M_STR = Trim(.Substitute(Str(M), " ", ""))
        ValidPath = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].ValidPath[" + M_STR + "]")
        BPScanned = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].BPScanned[" + M_STR + "]")
        NetScanned = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].NetScanned[" + M_STR +"]")
        RowCtr_STR = Trim(.Substitute(Str(RowCtr), " ", ""))

        Range("C" + RowCtr_STR).Value = ValidPath(1)
        Range("D" + RowCtr_STR).Value = BPScanned(1)
        Range("E" + RowCtr_STR).Value = NetScanned(1)

        TestSTR = ""
        StrLEN = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].Path[" + M_STR + "].Len")

        For idx = 0 To StrLEN(1) Step 1
            IDX_STR = Trim(.Substitute(Str(idx), " ", ""))
            temp = DDERequest(Channel, "FoundModules.Comm[" + H_STR + "].Path["
                                                          + M_STR + "].DATA[" + IDX_STR + "]")
            TempAry(idx) = temp(1)
        Next idx
```

```
                For idx = 0 To StrLEN(1) - 1 Step 1
                    If idx <> 0 Then
                        If TempAry(idx) = 18 And TempAry(idx - 1) <> 2 Then
                            TestSTR = TestSTR + "2, "
                            idx = idx + 2
                            temp = idx + TempAry(idx - 1)

                            While idx < temp
                                TestSTR = TestSTR + Chr(TempAry(idx))
                                idx = idx + 1
                            Wend

                                idx = idx – 1
                            If idx <> (StrLEN(1) - 1) Then
                                TestSTR = TestSTR + ", "
                            End If
                        Else
                            TestSTR = TestSTR + Trim(.Substitute(Str(TempAry(idx)), " ", ""))
                            If idx <> (StrLEN(1) - 1) Then
                                TestSTR = TestSTR + ", "
                            End If
                        End If
                    Else
                        TestSTR = TestSTR + Trim(.Substitute(Str(TempAry(idx)), " ", "")) + ", "
                    End If
                Next idx

                Range("B" + Trim(.Substitute(Str(RowCtr), " ", ""))).Value2 = ""
                Range("B" + Trim(.Substitute(Str(RowCtr), " ", ""))).Value2 = TestSTR
                RowCtr = RowCtr + 1
            Next M
            RowCtr = RowCtr + 1
        Next H
        DDETerminate Channel
        End With
    Next i
End Sub
```

BIBLIOGRAPHY

[1] R. W. Atherton, “Secure Manufacturing Control Systems and Industrial Java,”, ARC Advisory Group Driving Operational Excellence in Manufacturing Forum 2003, February 12 – 13, 2003

[2] K. T. Erickson, Programmable Logic Controllers: An emphasis on design and applications, Missouri; Dogwood Valley Press LLC, 2005, pp. 1231-1244.

[3] “Obtaining a Logix Processors Firmware Level Using a CIP Generic Message,” Allen-Bradley Knowledge Base Document, no. 23386, February 2009.

[4] “How to Read SN From the Module?,” Allen-Bradley Knowledge Base Document, no. 52870, February 2009.

[5] “Brief Summary to Clarify Message MSG Instruction CIP Paths and Pathing,” Allen-Bradley Knowledge Base Document, no. 22562, February 2009.

VITA

Curtis Alan Parrott is the son of Chris and Jean Parrott. He was born on January 27, 1984 in Springfield, Missouri. He attended Hickory Hill Elementary and Middle school while later attending Glendale High School, all in Springfield, Missouri. Curtis graduated high school in May of 2002 and started college later that year. He earned his Bachelor of Science in Electrical Engineering from the University of Missouri – Rolla in 2007. Once he completed his undergraduate studies he remained in Rolla, Missouri, to attend graduate school. In 2009 he graduated with a Master of Science degree from Missouri University of Science and Technology.