ARCUS DMX-ETH INTEGRATED STEPPER INTERFACE WITH ALLEN-BRADLEY COMPACTLOGIX

REVISION 1.01

This paper describes the integration of Arcus Technologies' DMX-ETH stepper motor/controller into an Allen-Bradley CompactLogix control system. At the time of development, the only method of communicating with the steppers over Ethernet was through the 1769-EWEB card. The EWEB card supports socket services necessary to manage the Ethernet connection. I've been told the newer AB processors L1 and L2 have socket management built in without having to use the EWEB card.

This author fully acknowledges there may be more than one way to implement this setup, but this has served me well and has been reliable.

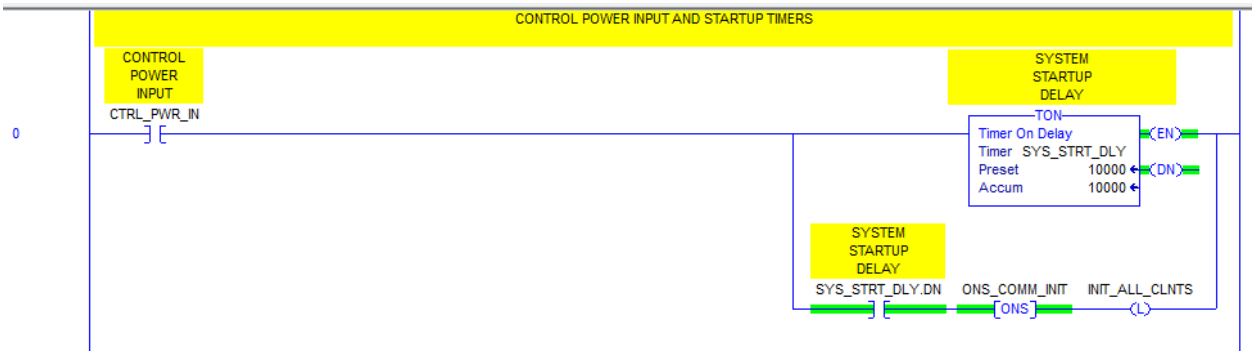
PROJECT OVERVIEW (THE BIG PICTURE)

This application establishes an Ethernet socket connection between a PLC (client) and five Arcus DMX-ETH's motor/controllers (servers) and maintains that connection. Rather than opening the connection, messaging, and then closing the connection; I chose to use a periodic message to keep the connection from timing out. Every 100 mS the PLC sends a message to each motor and gets a reply. The client subroutine establishes a hierarchy for these messages. If the main PLC routine needs to send a message to the motor, it queues the message and the client sends it out as the next command. If no messages are in the queue, the client will alternate between sending an 'SLS' and 'MST' which gives the PLC a continual status of both for each motor. The author's setup has the PLC and all DMX-ETH's Ethernet connections running through a managed switch.

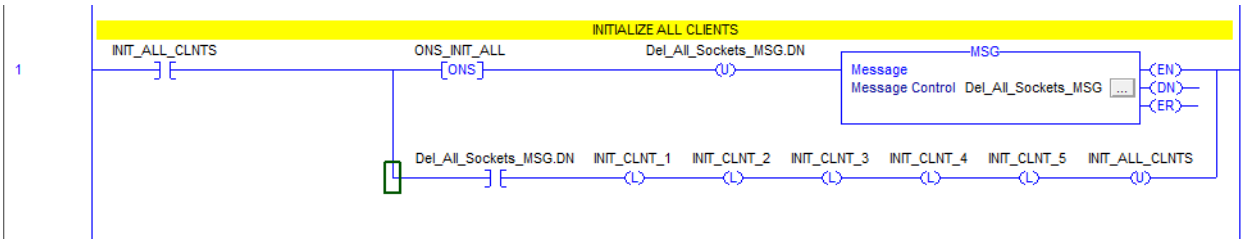
Within the MainTask of the project, the MainProgram folder contains a 'MAIN', five 'CLIENT', and five 'MOTOR' subroutines. The 'CLIENT' subroutines are called by main and handle all the Ethernet messaging. The 'MOTOR' subroutines contain an initialization section. I will only describe the details of CLIENT_1 and MOTOR_1. The other clients and motor subroutines are similar.

MAIN PROGRAM DETAILS

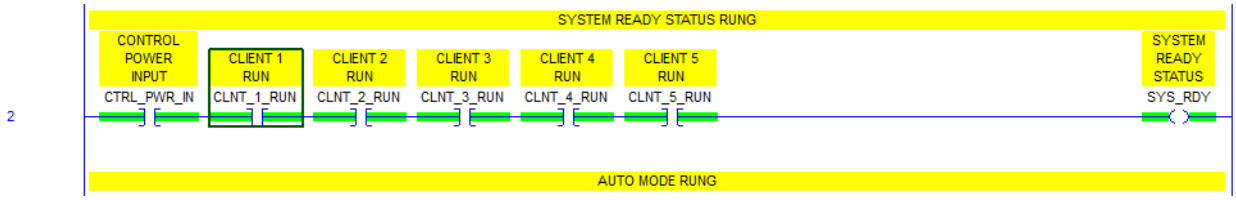
First, let it be known that the motor power inputs are fed from this systems control power. It takes several seconds after power-up for the motor controllers to initialize and ready to communicate. Rung 0, in the 'MAIN' routine, looks for CTRL_PWR_IN and delays 10 seconds before a one-shot latches INIT_ALL_CLNTS (Initialize All Clients).



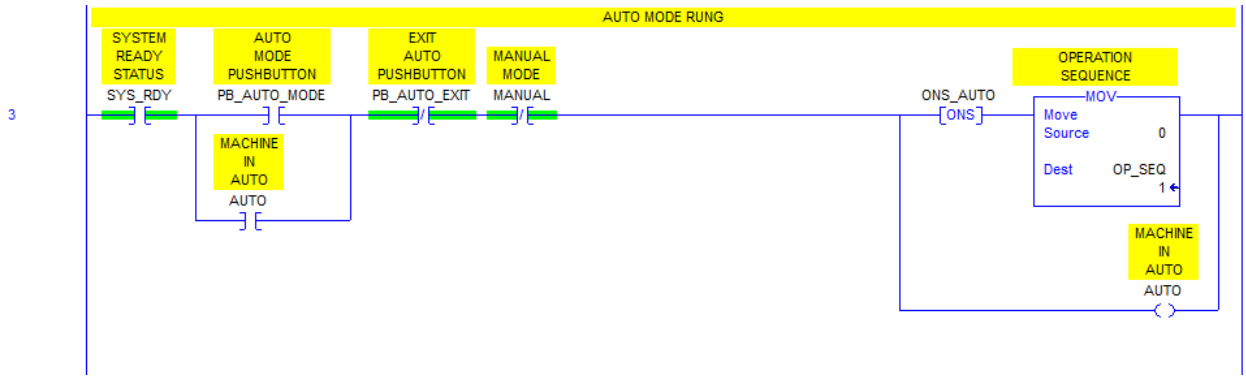
Rung 1 sees the INIT_ALL_CLNTS sets and broadcasts a message to delete all open sockets. It also sets five latches (INIT_CLNT_1, … INIT_CLNT_5) which will be picked up by their respective subroutines later in the scan. Details of client initialization will be described in a later section.



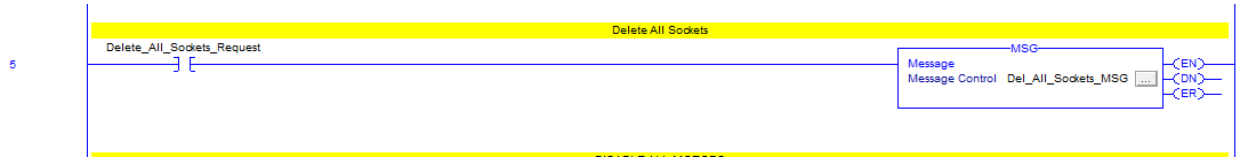
Rung 2 waits to see that all five clients are up and running before the 'system ready' status goes high.



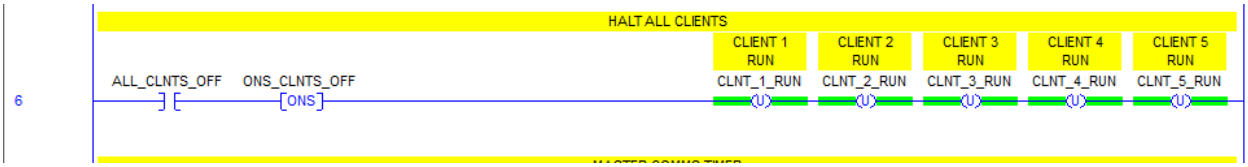
Rung 3 establishes this machines 'AUTO', as well as, initializing the AUTO sequence located in the SEQUENCE subroutine.



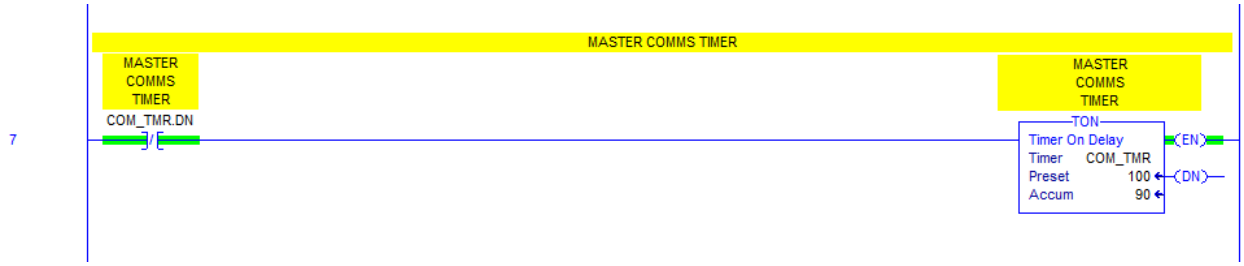
Rung 5 is a contact you can force which uses a MSG instruction to delete all open sockets. Details of the MSG block can be seen when opening in RSLogix5000.



Rung 6's input contact can be forced to turn off each client.



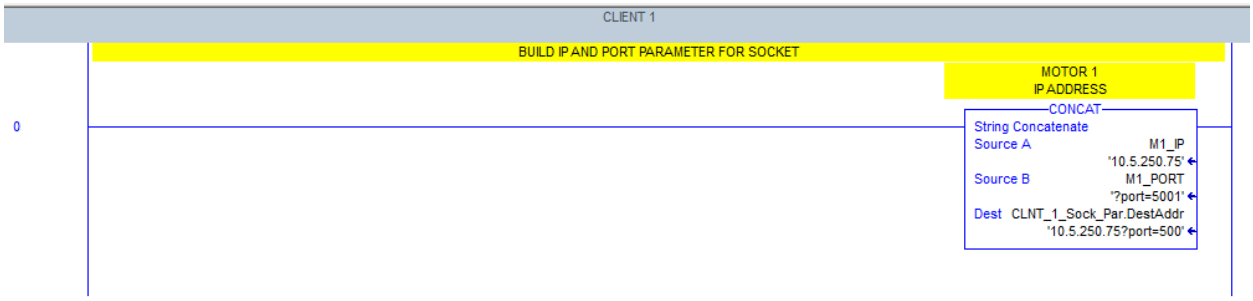
Rung 7 is the heartbeat for all messaging. It generates a COM_TMR.DN every 100 mS that gets picked up by active clients.



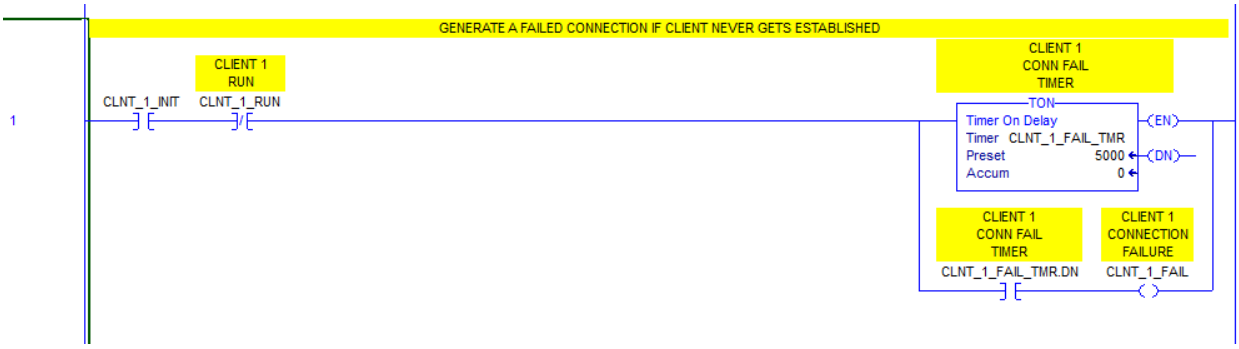
Rungs 8 through 18 call all CLIENT, MOTOR, and SEQUENCE subroutines sequentially.

CLINET_x PROGRAM DETAILS

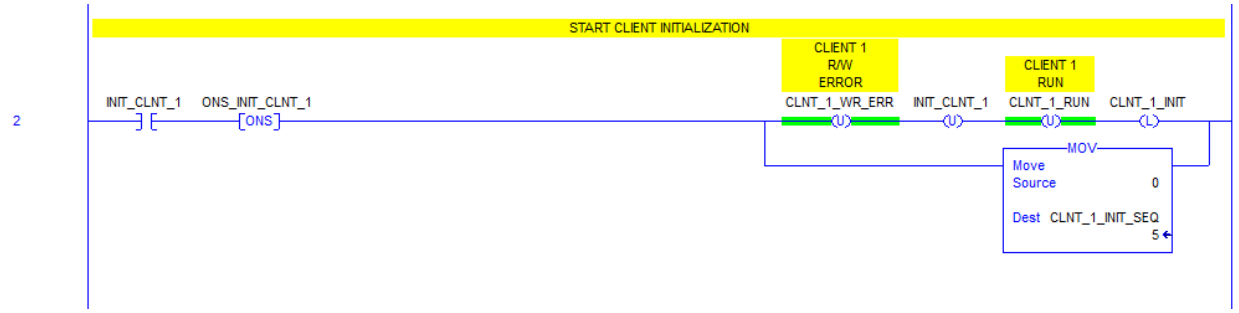
Enter the belly of the beast. The first half of this subroutine is for initialization only. When the 'MAIN' routine sets a 'CLNT_x_INIT', this section goes through the task of establishing this clients Ethernet connection. The second half is a sequencer that actually does the periodic messaging and handles the replies.
Rung 0 concatenates a string used for opening the socket. It takes a sting containing the IP address and adds it to a sting containing the ARCUS port number 5001.



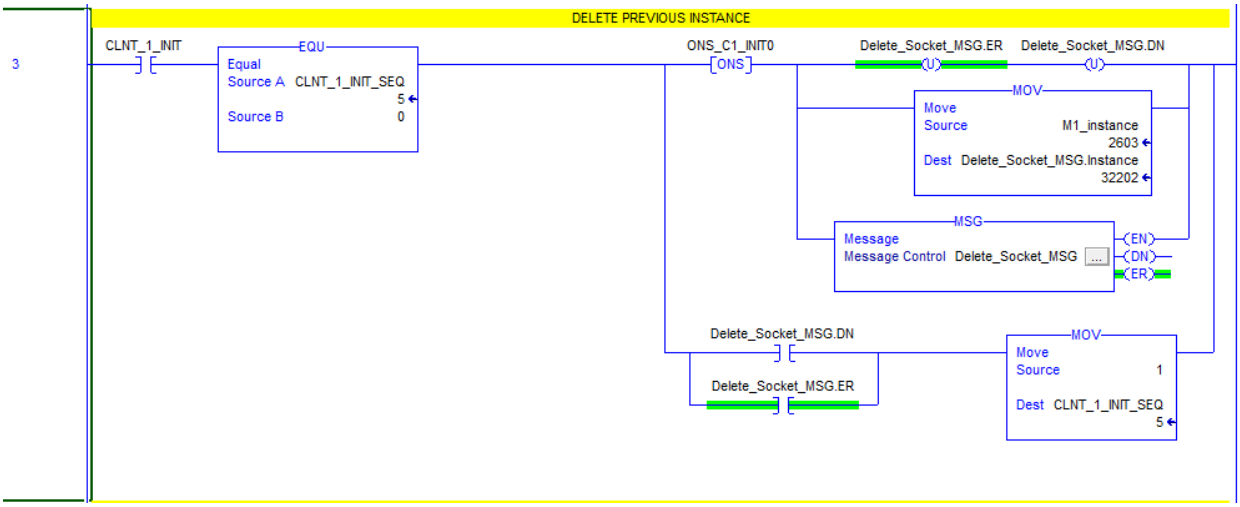
Rung 1 monitors for a failed connection. If the client is initializing and hasn't been established in 5 seconds, the CLNT_x_FAIL bit goes high.



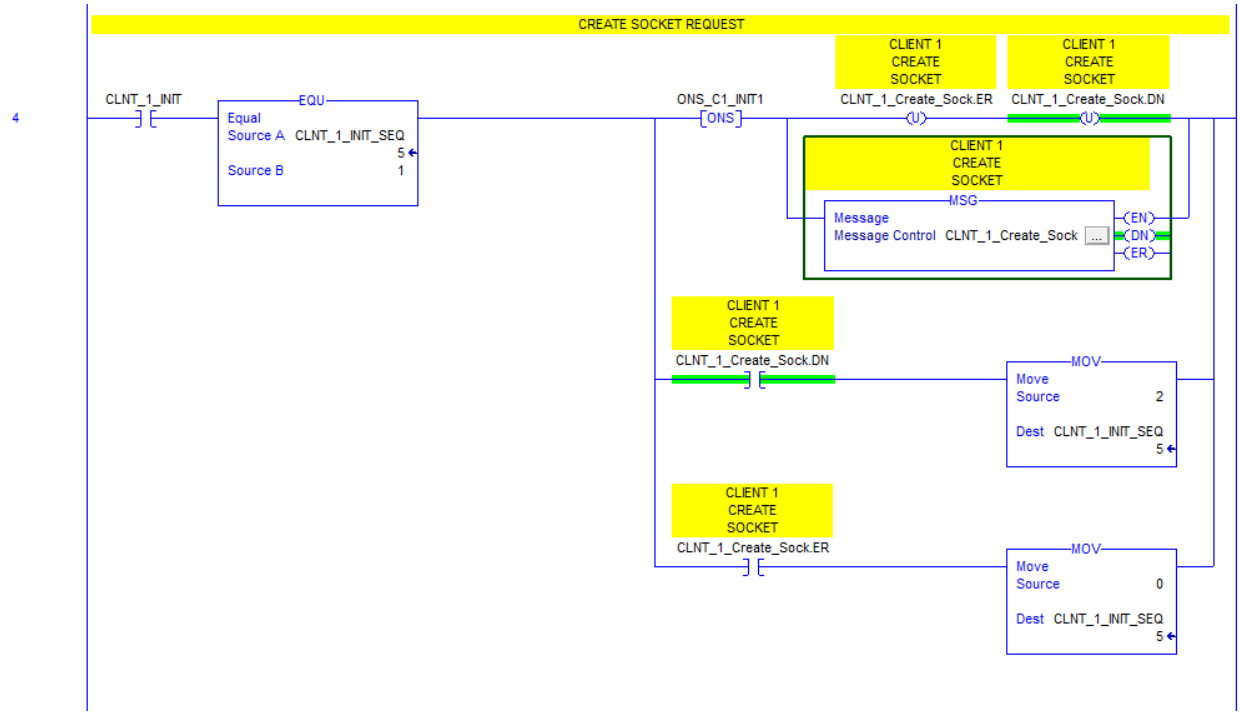
Rung 2 starts the client initialization sequence by turning on CLNT_x_INIT and setting the sequence number to 0.



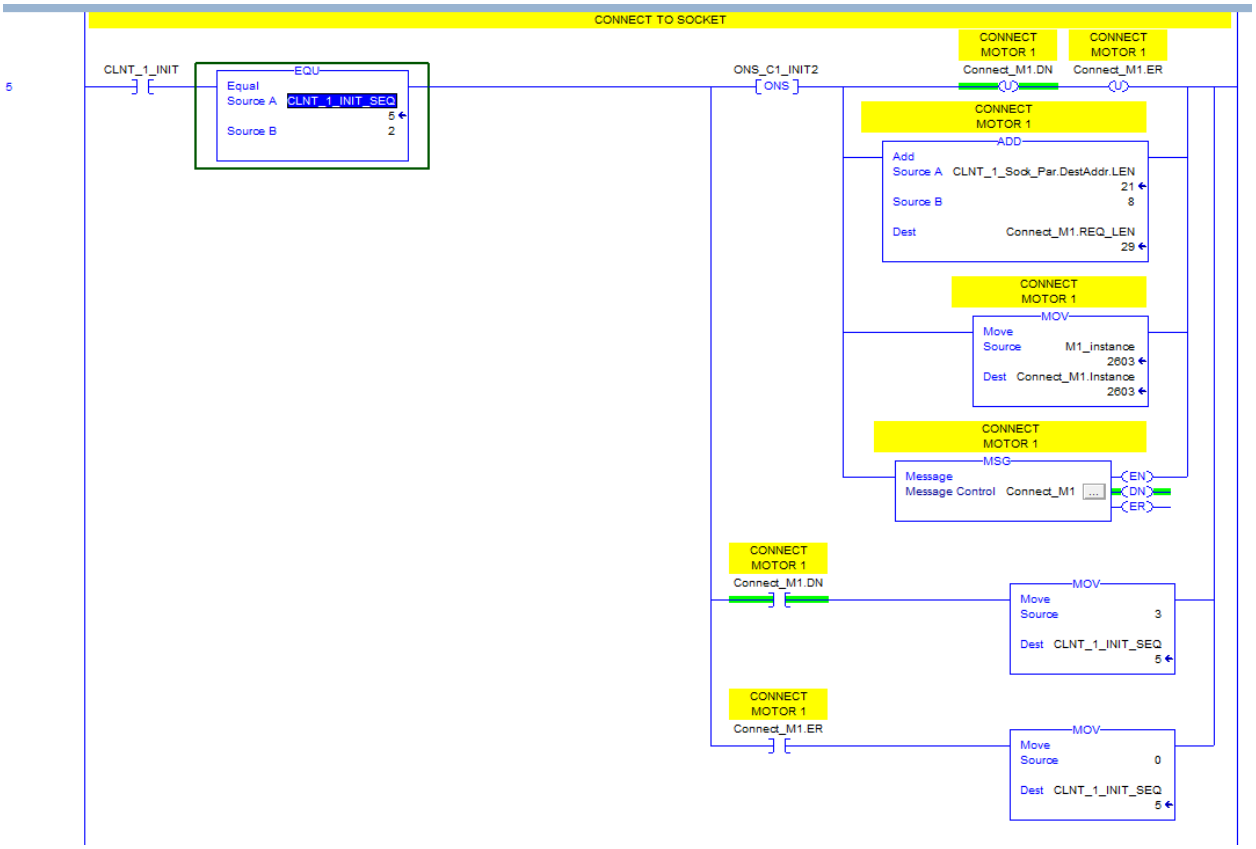
Rung 3; step 0 of the sequencer deletes any previous socket and advances to the next step.



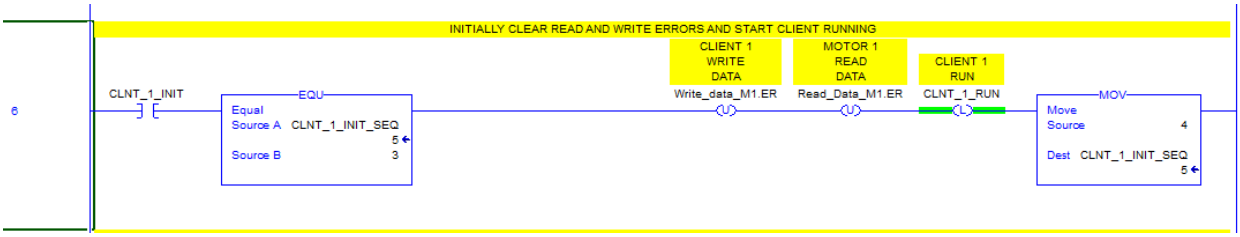
Rung 4; step 1 of the initialization, uses a MSG instruction to create a new socket. If it is successful, it advances to step 2. If it fails it jumps back to step 0.



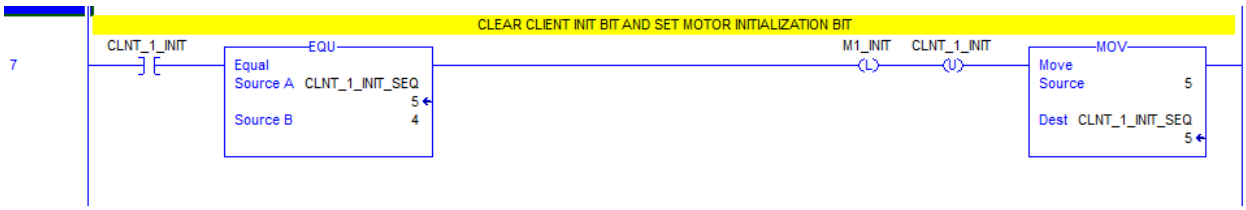
Rung 5; step 2 of the initialization, takes the newly opened socket and connects to it. If it is successful, it advances to the next step. If it errors, the sequence is reset to zero again.



Rung 6; step 3 of init, clears any erroneous read or write errors that may be present. It also latches the clients 'RUN' bit on before advancing to the next step.

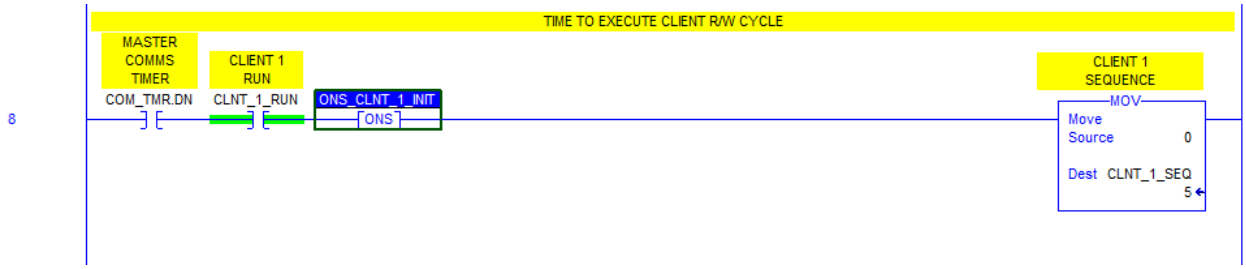


Rung 7; step 4 of init, sets the M1_INIT bit used in MOTOR_1 subroutine to initialize the ARCUS motor. It also turns off the CLNT_1_INIT at this point.

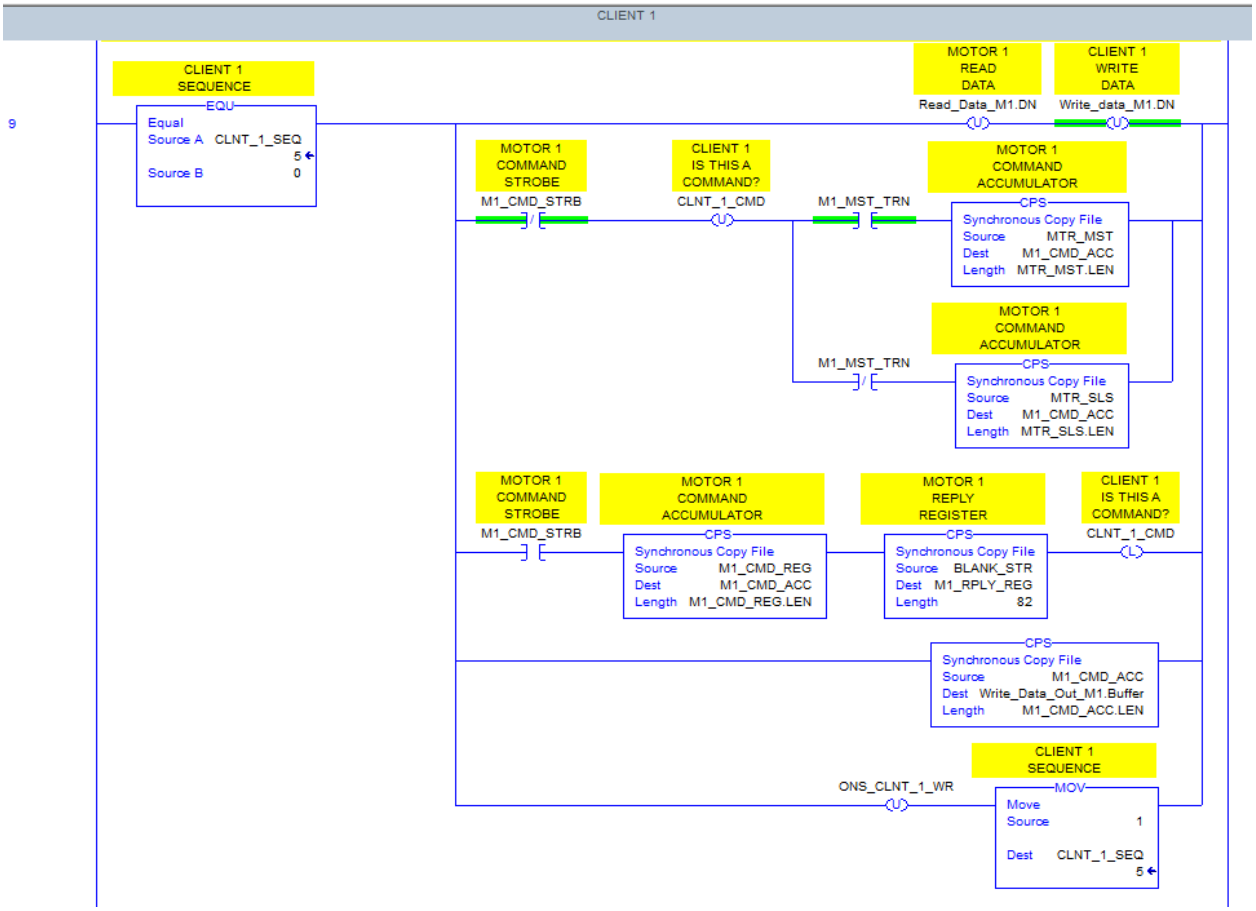


Rung 8 begins the last part of the CLIENT routine and is the heart of the Ethernet messaging. Whenever COM_TMR.DN is set, from the MAIN routine, and the client is

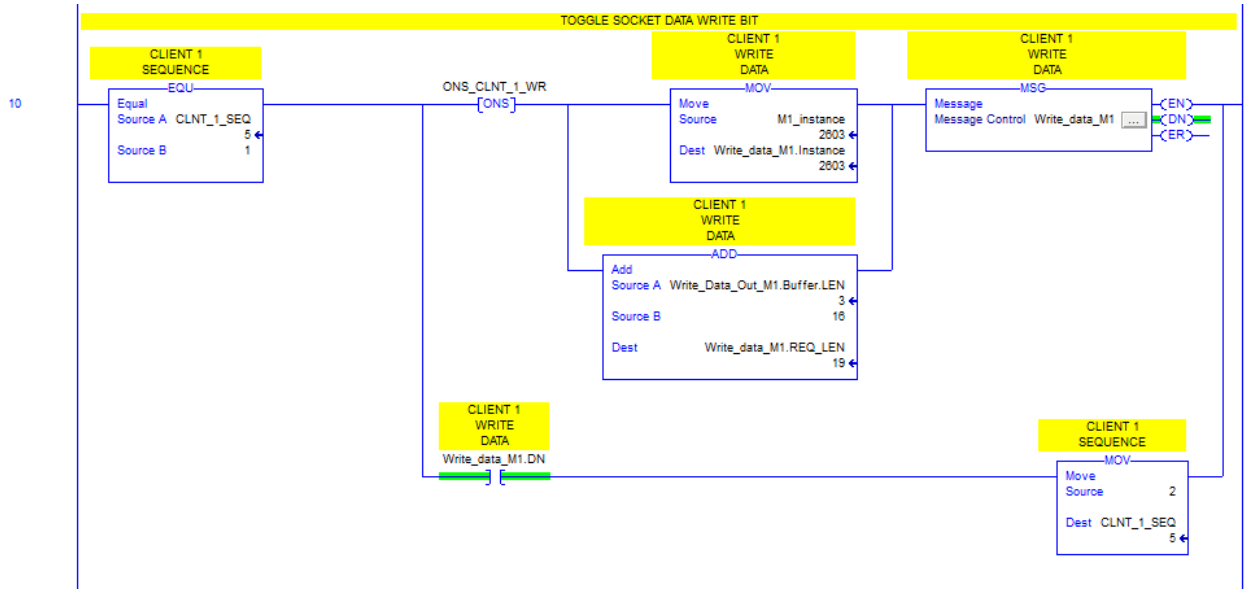
running, it's time to message. A zero gets moved into the client sequencer and proceeds.



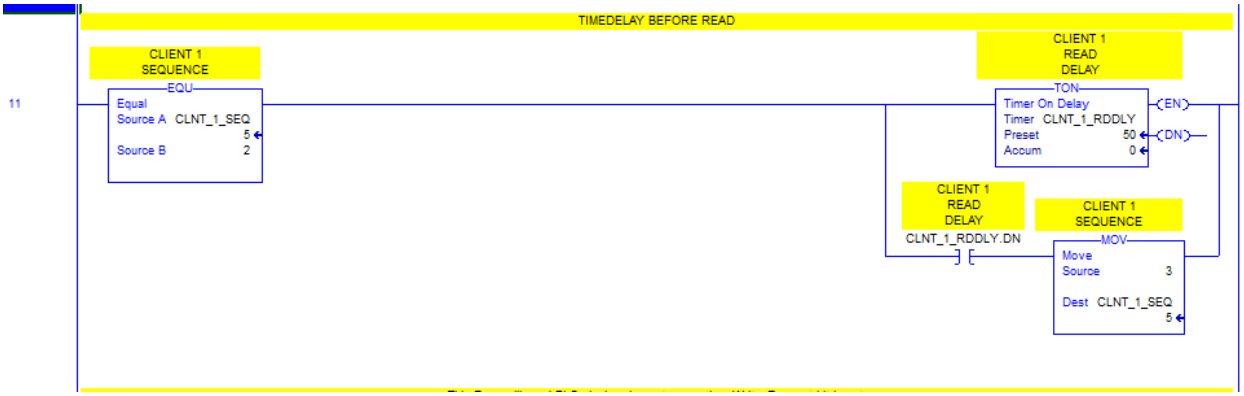
Rung 9; step 0 of messaging sequence, determines whether a command has been queued or whether it needs to process either MST or SLS. If the CMD_STRB is not set, it looks at whether it's time to message MST, or SLS, and moves that string to the command accumulator. If CMD_STRB is set, it moves the queued string to the command accumulator. Next the command accumulator gets moved to the control register of the MSG that is about to take place.



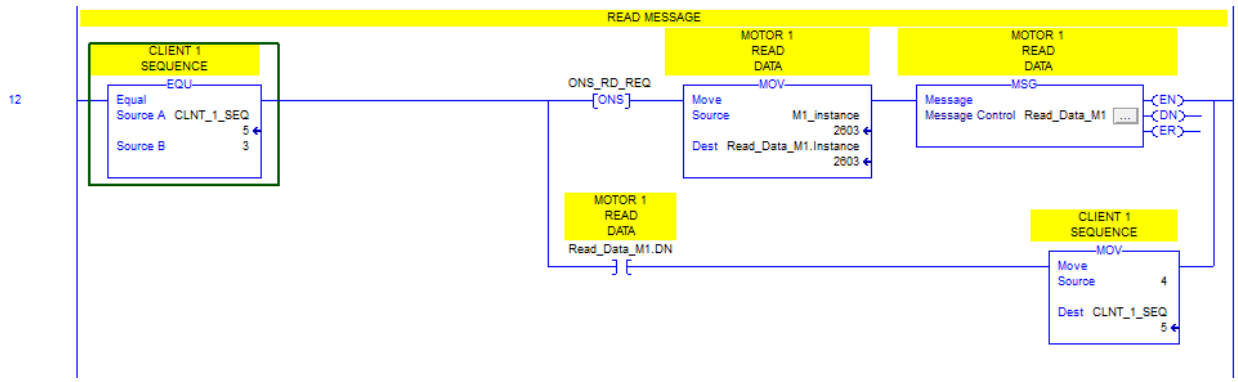
Rung 10; step 1 in the sequence, moves the instance number and buffer length into the MSG statement before actually sending the Ethernet message.



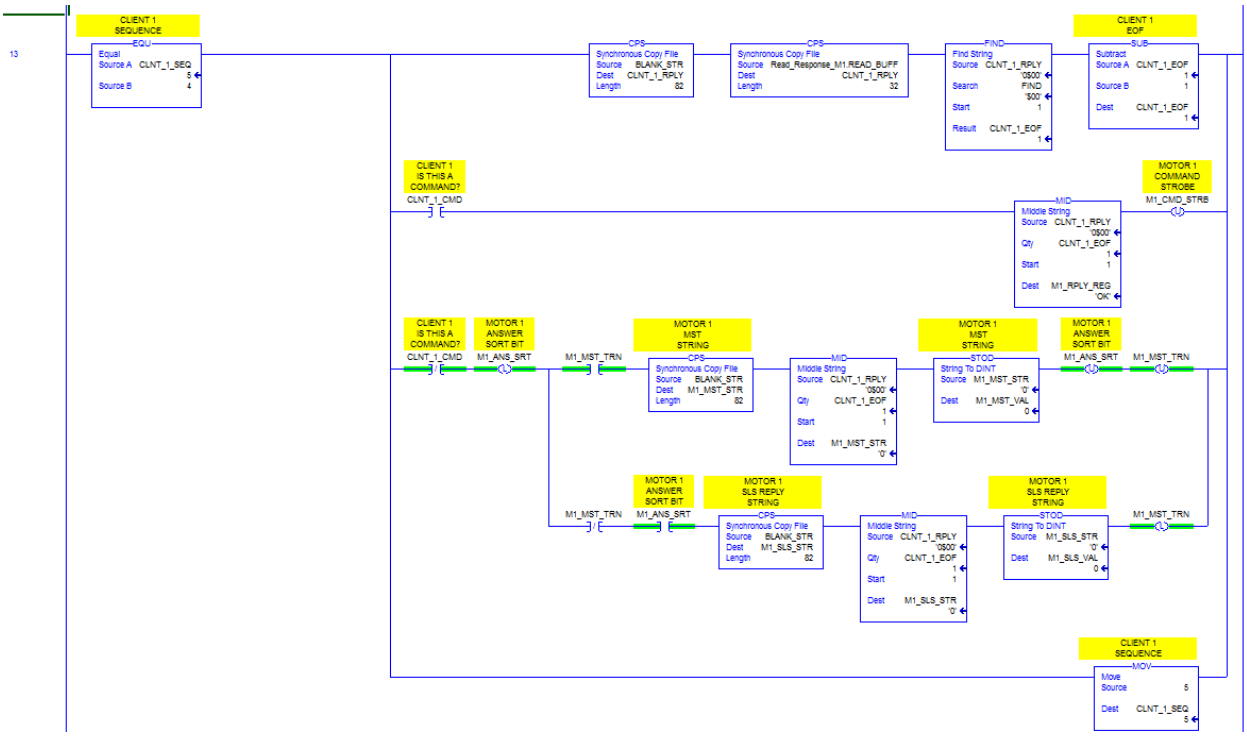
Rung 11 is a 50 mS time delay needed before a READ is performed.



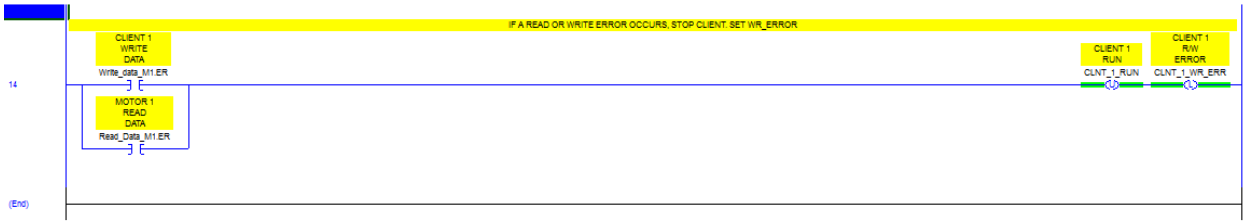
Rung 3 reads the Ethernet reply.



Rung 13 starts by clearing the reply string then copies the reply and parses it for EOF terminator. Next, it looks to see if this was a reply to a que'd command, or if it was a reply to either MST or SLS. The motor reply gets copied to its appropriate destination.



Rung 14 watches for any error in the READ or WRITE. If it occurs, the client is turned off and an error gets set.



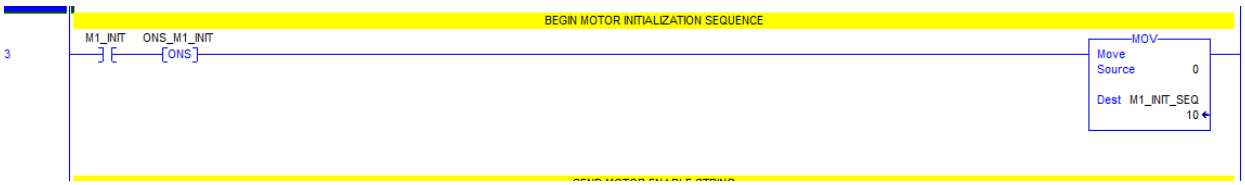
Now you know how the periodic messaging happens in the background. Next we will move onto how the DMX-ETH get's initialized for operation.

MOTOR_x PROGRAM DETAILS

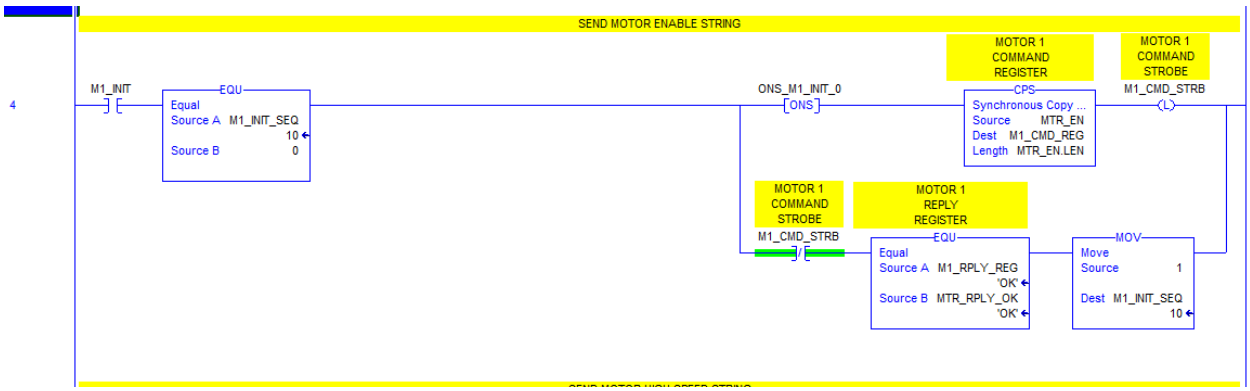
This program will vary depending on your application, or maybe omitted all together. I used it to determine motion status AND setup the default parameters of the motor such as speeds, ABS/INC, SLS mode, etc.

Rung 0 doesn't do anything, hence the NOP. It does parse the MST reply and allows you to see individual status of byte. Rungs 1 and 2 breaks out a general status based on the MST byte for 'moving' and 'limit error'.

Rung 3 begins the motor initialization sequence. The sequence gets set to zero to begin with.



Rung 4 moves MTR_EN string, which contains the ARCUS command 'EO=1', to the motor command register and the command strobe is set. The next time the client executes, it will see the command strobe set and send this string. It waits to see that CLIENT_x processed the command by clearing the stobe, and compares the motor's reply as 'OK' before continuing. This is the basic process from here on out; Move a string to the command register, set the strobe, look for strobe cleared (by the CLIENT), then act upon the reply.

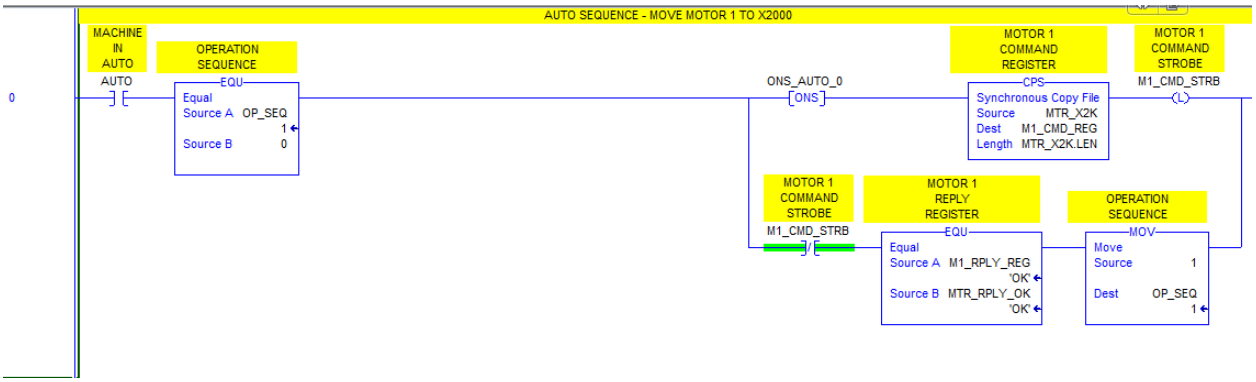


For simplicity, I will only describe what happens in the remaining rungs. Now you know the mechanics. Rung 5 sets the high speed parameter 'HSPD=2000' in the ARCUS motor. Rung 6 sends 'LSPD=800'. Next: acceleration, deceleration, ABS, SLS, etc. Once all the initialization parameters have been sent to the motor, the Mx_INIT bit gets cleared in rung 13.

SEQUENCE PROGRAM DETAILS

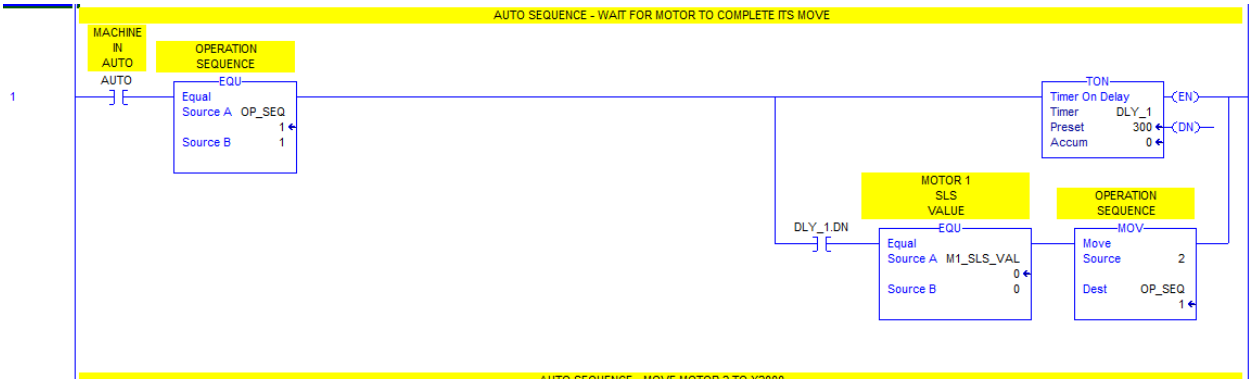
This program is provided as an example. When the system is placed in AUTO, motor 1 will move to X2000 and stop. Motor 2 will then move to X2000 and stop. Motor one will return to X0 and stop. Then motor 2 will follow to X0 and stop. The sequence will continue indefinitely. Use these rungs as a building block for your own motion needs.

Rung 0 moves MTR_X2K (string containing 'X2000') to M1_CMD_REG and sets M1's command strobe. When the strobe clears, we know the client has processed the command. We ensure M1 replied 'OK' and move on to the next step.



[!!!! IMPORTANT NOTE!!!]

Rung 1 is a delay before trying to determine if motor 1 has completed its move. The reason for the delay is due to any latency through the client processing a command, SLS, or MST. This 300mS delay has worked reliably for me. When M1_SLS_VAL is equal to '0', we know the motion has completed.



Rung 2 sends the string 'X2000' to motor 2. It looks for the strobe to be cleared and the reply is 'OK'. Another delay follows in rung 3 for the same reason as above. Rung 4 moves a 'X0' string to motor 1 and waits for an 'OK' reply. Rung 5, associated delay for SLS to be valid. Rung 6 moves motor 2 back to 'X0'. The sequence repeats as long as AUTO mode if valid.

NOTES, IDEAS, ADVICE

1. I'm sure many of you have reached a point of being out of system/PLC I/O. If you get in a pinch and need a few more I/O, you could always utilize the I/O of the ARCUS motor and read them through the Ethernet message. There will be a longer delay, but…… sometime you just need a few extra inputs.

2. I've performed homing both through the ARCUS internal homing inputs and command, as well as, through PLC inputs and move commands.

3. Any, and all, of the ARCUS commands can be executed with the preceding example. Often times you may want to read or even reset the encoder counter.

4. Bear in mind you may start multiple axes simultaneously, but in reality there is a very slight skew in starting/stopping. In other words, this is independent motion, not coordinated. In practice, starting and stopping multiple motors has worked as though they are synchronized.

5. Don't overlook the internal programming of the ARCUS motor! One could have canned routines in the ARCUS that could be fired off via Ethernet message. It could free up a lot of overhead of the PLC.

6. There are endless possibilities here. I hope this has been a stepping stone to success in your project.

My thanks and gratitude goes to the staff at ARCUS. Their technical support and coordinated efforts allowed me to achieve this particular motion integration.