October 22, 2018

CP260 Computer Real Time Interfacing

Devlin Paddock

**Multiple Robotic Arm Product Delivery System**

**1).Overview**

**2). Code Documentation**

**3). Two Highlights**

**4).Most Challenging Section**

**5).Summary**

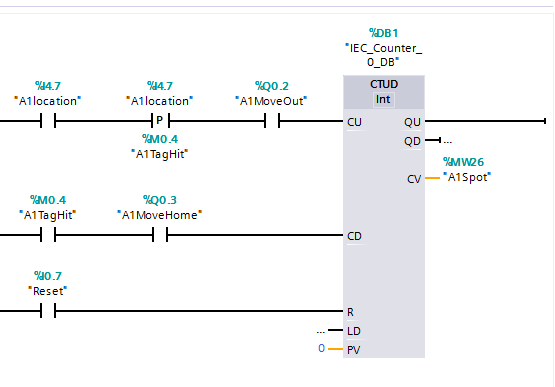
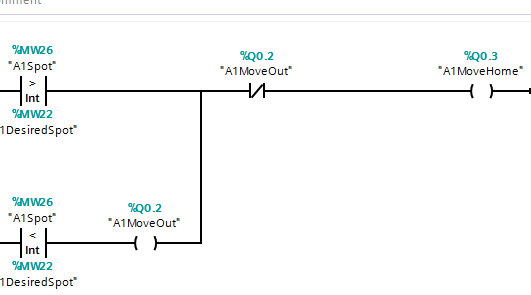
**6).PLC Tag Table**

**// error checking sentence is large consider cutting it down.**

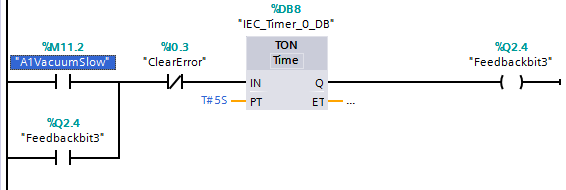
**Overview:**

The program uses a queue to hold up to 4 instructions to be used by two asynchronous arms for pickup up different product then returning to the home position and placing it on the bed. This program supports up to 3 different products; however, it can be expanded to support more by increasing the binary input switches and increasing the mask applied to the input bank inside the queue logic. There are error checks in place for if either arm on either pickup or delivery is taking too long to go down or initialize a vacuum pointing to either something in the way or a failure of the hand pump. If you overload the queue it will ignore any inputs given past the first four. If both arms arrive to a spot simultaneously arm 1 will extend prior to arm2 however if one is there even 1 scan before that arm will extend prior to the other and the other will wait until the other has returned to the in position. The elevator queue and arm movement are all asynchronous from each other and communicate through bits set and reset by the Arm Controllers.

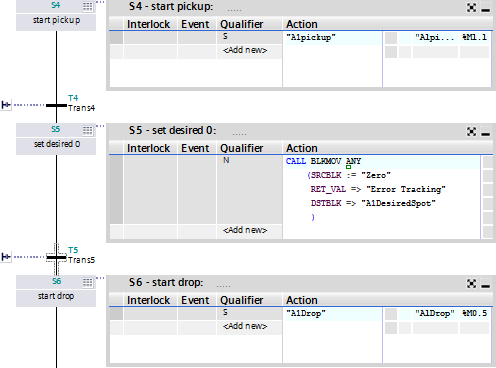
**Code Documentation:** A brief description of how each main process works in your program. Include snapshots of code if necessary.



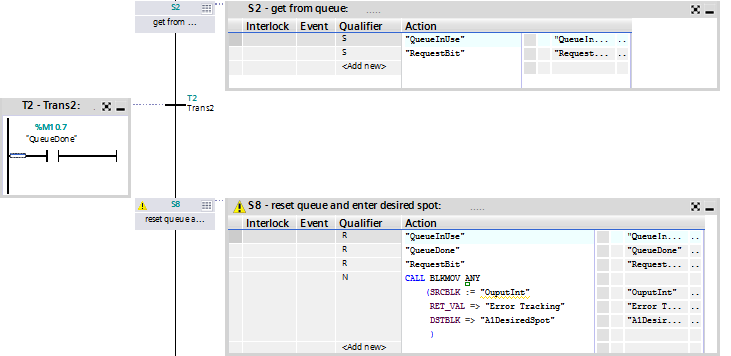
The two rungs of logic above makup the asynchronous elevator section. These will check wether the desired spot is greater then or less than the current spot before setting the move out or move home bits and counting each stop they pass untilthe desired spot and current spot are equal.



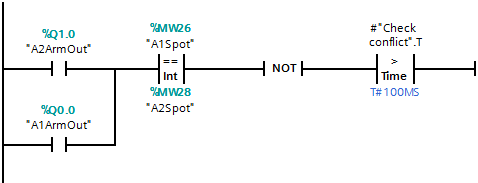
The rung above is the A1 Vacuum error logic. All error outputs follow a very similar setup where one bit is set on vacuum start it runs through an IEC delay timer before feed backing to keep the light on once the step has passed. The clear error will clear all error outputs and set bits.



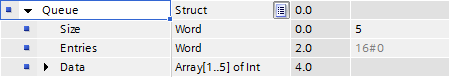
The screenshot above shows how the controller has mostly simple abstracted commands being used to interact with the other functions. The BLKMOV just sets the DesiredSpotInt to 0 , triggering the elevator to move it back while the bits A1Drop and A1Pickup trigger the pickup and drop methods in the A1Movement function.



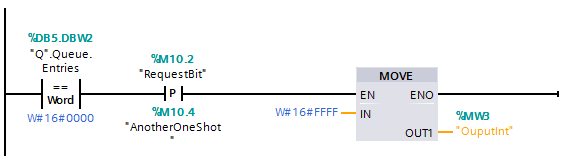
The Screenshot above is the queue request part of the A1Controler. This sets the queue in use and request bit preventing the other controller from using it while it retrieves data before taking the output data and inserting it into the desired spot triggering theA1Elevator method to move the device to the next location.



The above screenshot is an example of one of the conflict checking rungs used by the controllers. It requires a slight delay to allow the program to scan through and complete the logic before allowing the program to continue. Before the timer was placed it would skip this step due to the NOT instruction at the end defaulting it to true. While this timer can be set much lower I felt leaving it a bit longer gave more wiggle room for the controllers to avoid any collisions.



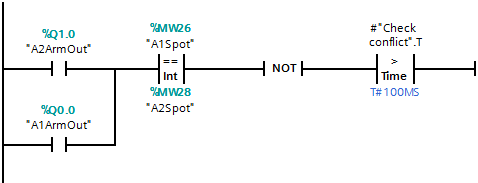
Above is the queue data struct. This is simply an array wrapped into a struct containing how many entries there are and the max size of the struct.



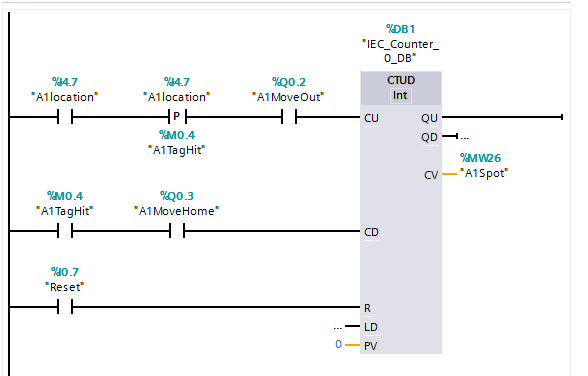
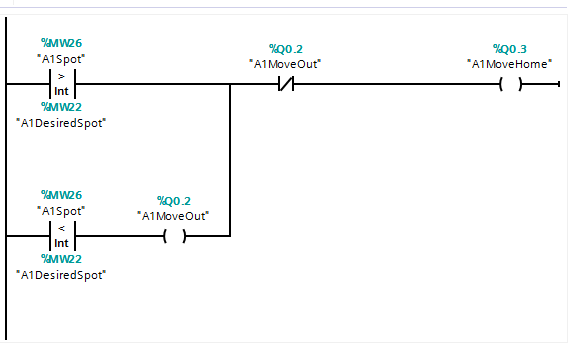


The two rungs above are the no data error rung and masking rung of the queue logic. The top rung detects if there is no data in the queue and outputs an error code (-1) if requested while the next rung masks the two bits in IB1 that is used for binary input into the queue. If more bits were to be added IN2 would be increased to a larger hex number.

**Two Highlights:** A description of two interesting solutions to programming problems presented by the lab. Include screen shots of code.

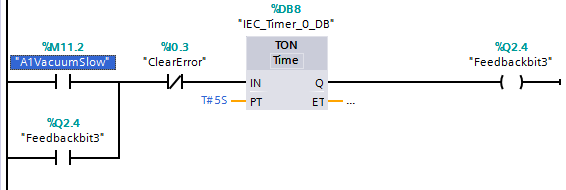


This rung had been extremely buggy without the timer as on half of the scans it seemed to skip the logic and just see the NOT as a true signal to continue. The timer allows it a second to breath and execute the logic prior to the NOT before passing onto the next step. I found this a really interesting problem as in most languages I used it executed before checking the value however here you can see it tries to execute and check simultaneously making the step an invalid state without the delay.



The above logic is the elevator method. This section I thought was interesting because it compressed really nicely allowing a self-checking A1MoveOut and Move Home preventing both from being hi at the same time. As well as having a really condense counter to keep track what location they were at.

**Most Challenging Section:** A description of the most challenging programming problem in the project.



# The logic above seems rather simple however at first I had the clear error past the timer and it threw an error that “[This use of IEC timers / IEC counters is not possible in this target system.](https://support.industry.siemens.com/tf/ww/en/posts/this-use-of-iec-timers-iec-counters-is-not-possible-in-this-target-system/161393/?page=0&pageSize=10)” This error I had mistakenly thought meant it could not be used in this plc leading me down a very long path with semantic timers. Semantic timers were way more complicated and required a lot more hands on control, eventually I accidentally replaced one of these in a different context and it fixed the error leading me to understand that you can only have a coil assignment after any timer.

**Summary:**

Overall the project works as intended and uses a very abstracted design method to give ease of debugging. Paring that with very thorough step, rung, function, and Tag Table names means that understanding the code from an outside view should be much easier than earlier projects. I would estimate my grade as 90.

**Simulation Display:** 