

Ladder Logic
Programming Techniques
By Duane Snider
Version 1.0

Ladder logic Programming Techniques By Duane Snider

COPYRIGHT © 2005 by Duane Snider

All Rights Reserved

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Request for permission or further information should be addressed to Duane Snider 8921 Cedar Mills Cv Cordova Tn 38016

Due to the nature of this publication and because of the different applications of programmable controllers, the readers and users and those responsible for applying the information herein contained must satisfy themselves as to the acceptability of each application and the use of equipment therein mentioned. In no event shall the publisher, editors, contributors and others involved in this publication be liable for direct, indirect or consequential damages resulting from the use of any techniques or equipment mentioned.

The illustrations, tables, and examples in this book are intended solely to illustrate the methods used in each application. The publisher, editors, or contributors and others involved in this publication cannot assume responsibility or liability for actual use based on the illustrative uses and applications.

No patent liability is assumed with respect to use of information, circuits, illustrations, equipment, or software described in this text.

Table of Contents

Chapter 1 The Basics	1-9
Hello World	1-9
Stating the logic	1-10
Rung Scan Order	1-11
Program Scan	1-11
I/O Scan	1-12
Variables and Data Types	1-12
Array Access and Pointers	1-13
Direct Access	1-14
Indirect Array Access using Pointers.....	1-14
Aliasing	1-15
Program Structure	1-16
Chapter 2 Documentation	2-17
Machine and Conveyor Documentation	2-17
ISA Process Standard.....	2-22
ISA Batch Standard.....	2-24
Chapter 3 Controllers.....	3-25
Discrete Motor Controller.....	3-28
Variable Frequency Drive.....	3-33
Single Solenoid Valve Controller	3-36
Double Solenoid Valve Controller.....	3-40
Analog Indicator with Alarms	3-42
PID Controller.....	3-49
Ratio Controller	3-57
Chapter 4 State Logic.....	4-60
State Logic – First Form	4-61
State Logic – Second Form.....	4-63
Gain in Weight Feeder	4-65
State Logic – Third Form.....	4-74
Chapter 5 Batch Control	5-79
Batch Logic.....	5-80
Recipe Organization.....	5-82
Recipe Management.....	5-85
Batch Control.....	5-86
Phase Logic.....	5-92
Defining the phases.....	5-92
Start.....	5-95
End	5-96
Feed Material 1	5-97
Feed Material 2	5-100
Tare Scale.....	5-100
Start Heat	5-101
End Heat.....	5-101
Start Cooling	5-102

Ladder logic Programming Techniques By Duane Snider

End Cooling 5-102
Start Agitator..... 5-103
Stop Agitator..... 5-103
If Process Variable..... 5-104
Delay 5-106
If Ok Operator Prompt 5-107
If Yes Operator Prompt..... 5-108
Wait for Phase Complete 5-111
Go to Step 5-112
Creating a Recipe..... 5-113
Chapter 6 Sequential Machine Logic..... 6-114
The Sequence Diagram 6-114
Programming the steps..... 6-115
Programming the end of cycle 6-118
Programming a timed step 6-118
Maintaining the step with power loss 6-119
Programming the Outputs..... 6-119
 Bumpless transfer to manual..... 6-119
 Transitioning outputs to the off state 6-121
Programming faults and cycle hold 6-121
Programming sequence paths 6-125
 Initiating the sequence 6-125
 Selection branch diverge..... 6-127
 Parallel branch diverge 6-129
Chapter 7 Determining Priority 7-131
 First Logic for 2 stations 7-131
 First Logic for 3 or more stations 7-133
 Priority Stack 7-135
Chapter 8 Sortation 8-138
 Tracking with an encoder..... 8-138
 Tracking with timers 8-144
Chapter 9 Zone Control 9-148
 Power and Free, Stop to Stop, no accumulation 9-148
 Power and Free, Choke Zone..... 9-151
 Power and Free, Count Zone 9-153
 Power and Free, Merge into a count zone..... 9-154
 Power and Free, Track Switch..... 9-157
Chapter 10 Tips and Tricks..... 10-161
 Toggle Push Button 10-161
 Cascading Start Stop..... 10-161
 A simple message display..... 10-164
 Buffering Transactions..... 10-165
Appendix A Binary Numbers A-171
Appendix B Useful Calculations B-173
 Unit Conversions B-173
 Scaling..... B-174

Appendix C Instruction Set.....	C-175
Bit Instructions.....	C-175
Timers and Counters.....	C-176
Compare instructions.....	C-177
Compute/Math Instructions.....	C-178
Move/Logical Instructions.....	C-179
File Misc. Instructions.....	C-180
File/Shift Instructions.....	C-181
Sequencer Instructions.....	C-182
Program Control Instructions.....	C-183
For/Break Instructions.....	C-183
Special Instructions.....	C-184
Trig Functions.....	C-185
Advance Math Instructions.....	C-185
Math Conversions.....	C-186
ASCII Serial Port.....	C-187
ASCII String Instructions.....	C-188
ASCII Conversion.....	C-189
Appendix D Table of Figures.....	D-190

About The Author

I was born in Winters California in 1961. My father worked for PG&E as an electrician installing substations around the state. My mother was a homemaker with 4 children. Because of my fathers job we moved around a lot, mostly around central California. I remember that when I was in the first grade, this nation had begun the journey that would take us to the moon. Young engineers were on television, and Walter Cronkite was telling us how smart they were. The eyes of the nation were watching this new form of hero carry us into the 21st Century on the top of a Saturn five Rocket. They were the Rock Stars. They were the Sports heros. They were ordinary looking men who were doing extraordinary things. .And one first grade class room and one little boy in Oriville California began dreaming about the future. My mother told me years later that my first grade teacher thought I was really smart. She said I was always asking questions that she couldn't answer. And of course this was one of those stories that my mother told over and over.

When I was in the second grade we moved to Atkins Arkansas. My grandparents and parents had bought about 90 acres of pasture and woodland at the foothills of the Ozark Mountains. My grandparents had moved to Arkansas some years earlier. Each year we would make the annual pilgrimage to see them. I can still remember a few images of deserts and tall cactuses and Indians. All of this came pouring in through the back windows of a brown two tone Chevy station wagon. At first Arkansas seemed like one big camping trip. We drew our water from a well. We had outdoor plumbing as we liked to say. Others would call it a 2 hole outhouse. We heated the house with a wood stove and there was no air conditioner. We did however have electric fans. And if it got too hot and the bugs weren't too bad, and of course if it wasn't raining, we would move the beds outside and sleep. Of course it wasn't long until we were adding rooms onto the small house and running indoor plumbing. At first we had chickens and pigs. We eventually made it through just about all of the farm animals. They included rabbits, goats, turkeys, ducks, guinnes, dogs, and cats. At one time my grandmother even had a milk cow. But, I never did get used to drinking 100% milk fat from a cow that had been grazing on wild onions. We eventually put fencing around most of the acreage and we bought beef cattle. Our farm sat between two small mountains, Pea Ridge to the north and Crow Mountain to the south. When I would get home from school I would go outside and play. I could tell when it was time to do my chores, by gaging the length of the shadow that was cast by the south mountain onto the north. The nearest town where we went to school was 4 miles away.

By the time I was in high school I was taking college prep courses and participating in most of the sports programs. When I was seventeen I took a summer job at the local Pickle factory. My first paying job came with a title. I was lead man on the onion line. Responsibilities included taking a forklift down to the loading docks. Get a crate of onions. Proceed to open a sack of such onions and pour those onions into the onion peeler. Run the peeled onions onto a conveyor and then help cut the tops with a knife. Repeat these steps until the crate is empty, then, go get another crate. Over time I managed to do almost every job in the factory. These included pickle netting, truck unloading, heading pickle tanks, running slicers, packers, and labelers. By the time I was in college I was third shift supervisor over production.

I attended my first two years of college at North Arkansas Community College in Harrison Arkansas. I had received a basketball scholarship. I also did work study. My first work study job

was doing the laundry for the basketball team. Our colors were red and white and for my freshmen year we wore pink socks and pink jocks for practice. By my sophomore year I had improved my laundry skills. I also worked as a laboratory assistant for the freshmen chemistry class. My grades were pretty good in my freshmen year so in my sophomore year I was asked to do tutoring. I tutored basic math, college algebra, chemistry and American history. At one time I was tutoring nine students one or two hours a week. I received an associate of applied science in the field of engineering. I also received an award for the outstanding graduate in the field of engineering. After Junior college, I attended Arkansas Tech University in Russellville Arkansas where I received a bachelor of science in general engineering. This degree included mechanical and electrical course work. I took my electives in electrical.

I took my first degreed job in Little Rock Arkansas at a small company called Engram Systems. There, I programmed laboratory data collection system for the National Center for Toxicological Research in Pine Bluff Arkansas and other government laboratories around the country. Mostly they did rat, mouse, or monkey studies to determine if certain chemicals would cause cancer or other diseases. The famous Sacrin study was done in Pine Bluff. The biggest system that I worked on was written in Z80 assembly. This was before Big Blue had come out with their PC. I also worked in Basic and "C". With the Reagan budget cuts, I found myself looking for a job after two and half years. I did a bit of contract work for a few months mostly doing small business database systems. Then, I went to work for a small engineering firm Koehler Engineers in Little Rock. We were doing PLC systems for a small conveyor company and other manufacturers. But the conveyor work ran out pretty quick and I found myself out of a job again. So, I started doing contract work again. I did an instrument design for a company in Houston, I also did some PLC and process work for the engineering firm Garver and Garver in Little Rock. During this time a fellow from Memphis had been calling me and asking if I would go to work for a conveyor company there. I didn't pay much attention at first because I did not want to move. But, he called me about six months later when I was sitting at home with no work in site and a stack of unpaid bills. So, after contracting for a little over two years I made the move to Memphis. I had worked at Southern Systems about six months when they put me in charge of the electrical engineering department. In that position I hired and trained the electrical engineering staff. Our customers were primarily automotive and appliance manufacturers. Southern makes several types of heavy duty conveyor systems including, Power and Free, Tow Line, Chain Driven Live Roller, Chain on Edge, Chain on Flat, Slat conveyors, Lifts, and Transfers. We would also purchased conveyor components from other manufacturers. We designed sortation systems and package and pallet handling system. Of course all of this work required travel and I found we had a fairly high turnover among my staff. I usually had between three to six full time engineers. Most of these engineers would stay one to two years and then take there experience to a job requiring less travel and fewer hours. I did manage however to keep two really good engineers for several years. Of course with the high turnover I found myself playing the role of teacher and drawing upon my experience as a tutor in college. At the time and still today I found that there are few books, if any, that would provide a young engineer with a solid background in the most common techniques of ladder logic programming. I left Southern systems after having worked there for a little over six years. The work load, the stress, and the travel had taken its toll and I was ready for new challenges. I also had a family with one two year old boy and a baby girl that had just arrived. I decided to take a job with a local integrator. This integrator serves mainly local industries based in the Memphis area. There I broadened my experience with projects in the

Ladder logic Programming Techniques By Duane Snider

process world. These included the food and beverage, chemical, pharmaceutical, and pulp and paper industries. Projects included batching systems, continuous processes, instrumentation, data collection, and inventory tracking. I also began working with several DCS (Distributed Control Systems) controllers.

I still find myself playing the role of teacher. And so, with the encouragement from my coworkers I began the process of pulling together some of the most common programming techniques that I have come across. Some of these techniques I have taken from programming examples that others have done. In chapter 2, I discuss techniques for sequential machine control. I found this method in a Whirlpool electrical specification while I was working at Southern.

Chapter 1 The Basics

Ladder logic is an unstructured programming language. In order to make sense of complex logic we need to apply some structure to it. This book will look at several programming techniques that apply structure to your program. These techniques will allow you to program many common applications resulting in a program that is both easy for you to troubleshoot and easy for others to understand.

In this chapter, we will introduce you to ladder logic programming. We will also look at some guidelines on how to document your program.

Hello World

The first “Basic” language program that I learned was “Hello World”. The code is listed below

```
Print “Hello World”;
```

When I ran this program, “Hello World” was displayed on the screen.

In ladder logic we will also consider a simple program to illustrate how the language works. A pushbutton labeled “PRINT” is wired to a PLC input. A light labeled “HELLO_WORLD” is wired to a PLC output. In Figure 1-1 the “PRINT” input is assigned to a normally open contact instruction. The output “HELLO_WORLD” is assigned to an output instruction. The two instructions are connected together to form a ladder logic rung. When the pushbutton is pressed, the state of the PLC input changes from off to on. When the normally open contact instruction sees the state of the input change from off to on it passes that true state to the output instruction. The output instruction then changes the state of the output to on. The light “HELLO_WORLD” is then turned on. The output is turned off when the pushbutton is released.

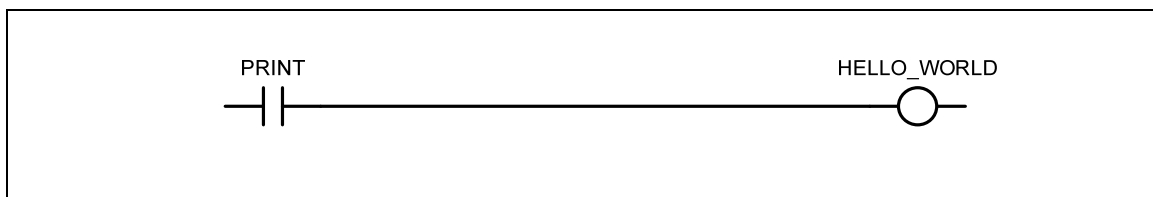


Figure 1-1 Normally Open Logic

The rung is read from left to right. If we translate the logic into English we would say, when PRINT is on, then HELLO_WORLD is turned on.

In Figure 1-2 the normally closed contact works in the opposite way as the normally open contact. The English translation for the rung is, when “A” is off, “B” is turned on. When “A” is on, then “B” is turned off. “A” determines the state of “B” and “B” does not affect the state of “A”.



Figure 1-2 Normally Closed Logic

Stating the logic

We can interpret the rung in Figure 1-3 to English. If “A” is on and “B” is on and “C” is not on then turn on “D”. We can also say, if “A” is not on or “B” is not on or “C” is on then “D” is not on. A, B and (not C) are in series. We can say they are “anded” together.

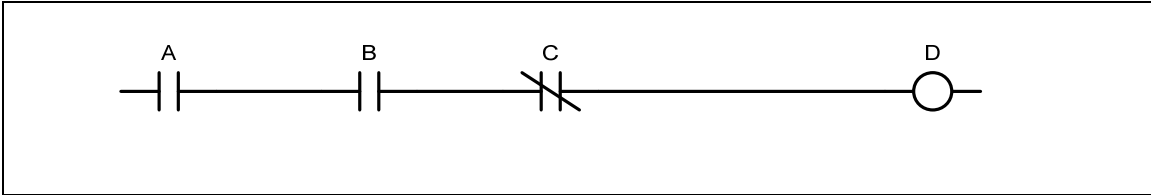


Figure 1-3 And Logic

In Figure 1-4 the English translation is, If “A” is on or “B” is on or “C” is off then “D” is on. We can also say that “D” is off, if “A” is off and “B” is off and “C” is on. A, B and (not C) are in parallel. We can say they are “or-ed” together.

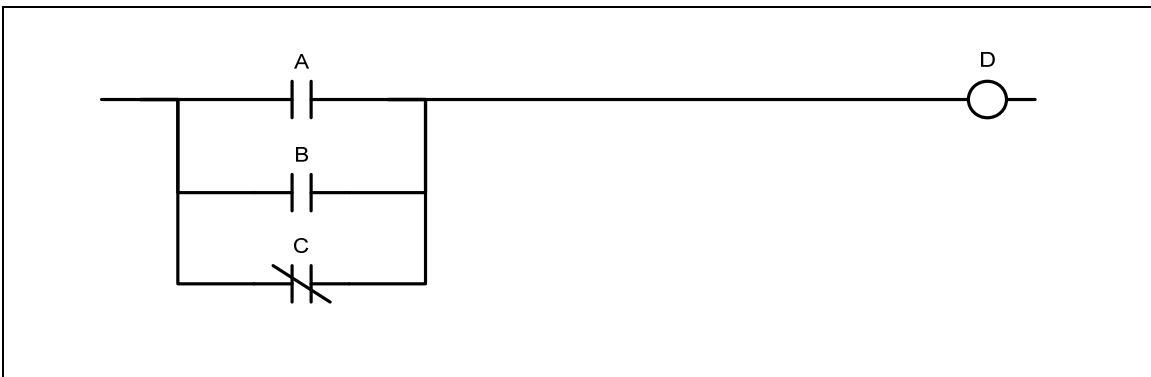


Figure 1-4 Or Logic

Figure 1-5 shows how a latch circuit is implemented. We can say that “C” is on when “A” is on. “C” will remain on until “B” is on. Because “C” is in a branch around “A”, “C” will remain on even when “A” is turned off. We would say that “C” is latched on.

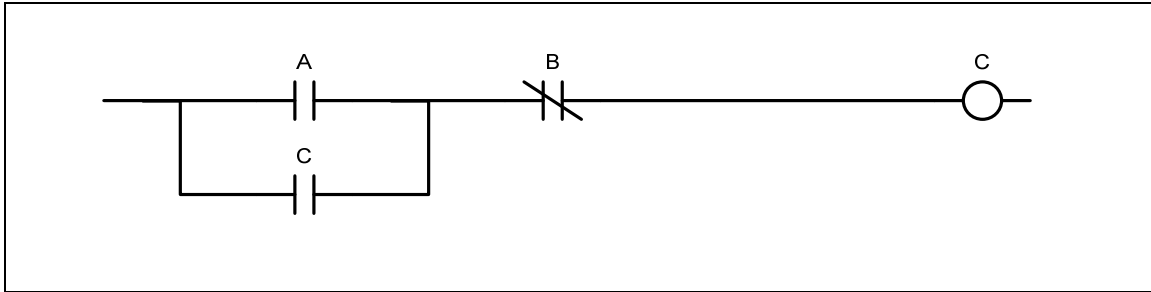


Figure 1-5 Latch Logic

In this example we used “C” as an output and as an input. In ladder logic, any bit can be examined with an input instruction such as a normally open contact or a normally closed contact. There is not a set limit to the number of times this bit can be used.

Rung Scan Order

In Figure 1-6 the order in which the instructions in a ladder logic rung are solved is shown. The logic is solved from left to right. When a branch occurs the top branch is solved first. The next lower branch is then solved.

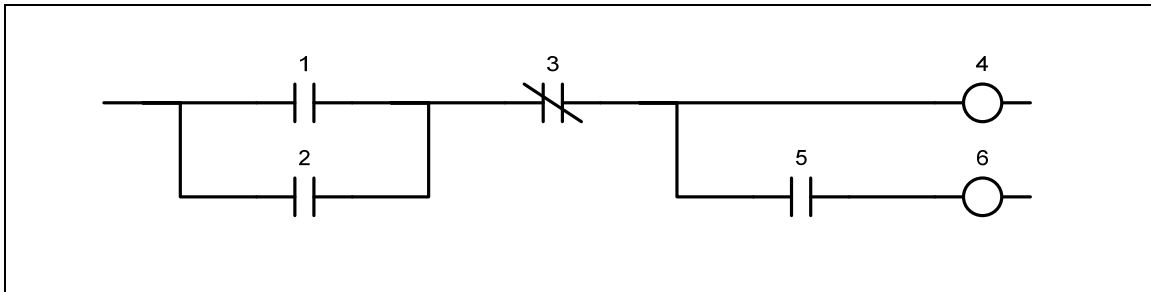


Figure 1-6 Scan Order

Program Scan

A ladder logic rung is scanned from left to right in order to determine the state of the outputs. Multiple rungs make a ladder logic program. The PLC scans these rungs beginning with the first rung and then ending with the last. The PLC then begins the scan again at the first rung. The time that it takes the PLC to complete one scan of the program is called the scan time. There are some variations on this theme also. In the Allen Bradley Control Logix processor there are continuous, periodic, or event driven programs. You can also have routines that can be called from a main program. These routines can also be executed in a for-next loop. Different PLC makers may scan their rungs in different manners. This may have some affect on the way some logic is evaluated.

I/O Scan

In the Allen Bradley PLC5, and SLC500 PLC's the I/O (Inputs/Outputs) is updated at the beginning or the end of the program scan. When an input changes state, the entire program sees that input during the next program scan. This is important if two or more rungs are examining the same input. In the Allen Bradley Control Logix processor, the I/O cards update the I/O tags in the PLC asynchronously from the program scan. This means that when an input changes state, the first part of the program could see the input off while the second part sees the input on. In some logic this difference could cause the logic to behave in unexpected ways. Even though the outputs are also updated asynchronously, usually this will have no effect on the way the logic is solved. If your program requires that all of the logic see the inputs change state at the same time, then you can program each input to change the state of an internal tag in the PLC. In Figure 1-7 the I/O address is used to update an internal tag. This logic would be placed at the beginning of the program for each input, in order to synchronize the inputs with the rest of the program.

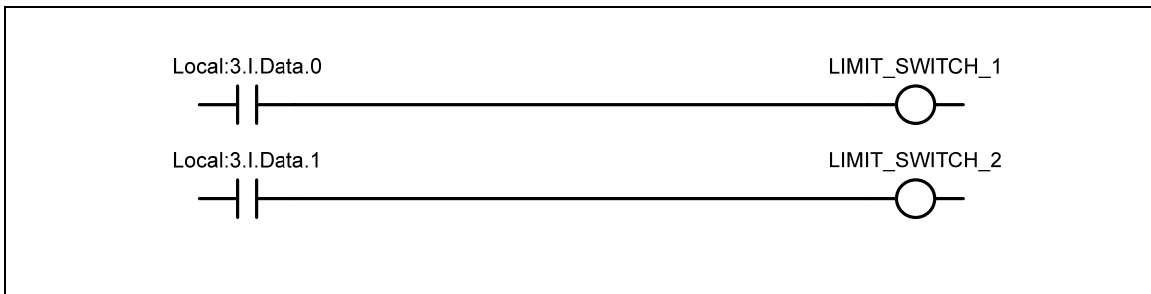


Figure 1-7 I/O Synchronization to Program Scan

In the first rung, the Control Logix format for the input address specifies that the input is in local slot three bit zero. It is tempting to use a MOV instruction to move a block of inputs into an array of discrete elements. But, by doing this you will not be able to force inputs from the ladder logic screen. And, then it will be harder to find these forces in the data table than it would be to find them in the ladder logic.

Variables and Data Types

Variables of different types can be created in the PLC. In Table 1-1, the primary data types are listed.

Type	Description	Range	Memory
BOOL	A binary digit, Boolean.	0 or 1	1 bit
INT	A 16 bit integer that expresses whole numbers	-32768 to 32767	1 16 bit word
SINT	A 8 bit integer that expresses whole numbers	-128 to 127	8 bits
DINT	A Double integer, 32 bit integer	-2,147,483,648 to +2,147,483,647	1 32 bit word
Real	Expresses a 32 bit IEEE floating point number		2 16 bit words
String	Holds character data	Up to 82 characters	2 characters per word

Table 1-1 Primary Data Types

When a variable is created, it is given a tag and a data type. For example, the tag MIXER_WEIGHT would be given a Real type.

An array of variables can also be created. The array SILO_WEIGHT[21] contains the weights of silos 1 to 20. The array contains elements SILO_WEIGHT[0] to SILO_WEIGHT[20]. The data type for the array would be Real.

The Control Logix processor also has predefined data types. These data types are collections of other data types (structures). For example, in Table 1-2, a timer tag is composed of the following elements.

<i>Mnemonic</i>	<i>Description</i>	<i>Type</i>
PRE	Preset Value	Dint
ACC	Accumulated Value	Dint
EN	Enable Bit	Boolean
TT	Timing Bit	Boolean
DN	Done Bit	Boolean

Table 1-2 Predefined Timer Data Type

If we create a timer tag called TIMER_1, then we can examine the done bit in a normally open contact by using the TIMER_1.DN tag.

In addition to the pre-defined tags, you can create user defined data types. These data types are also a collection of other data types. In Table 1-3, you can create a silo user defined data type.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	Silo description	String
PRODUCT	Product description in silo	String
WT	Actual weight in the silo	Dint
CAP	Capacity of the silo	Dint
LOAD_EN	Loading into silo enabled	Boolean
DISCH_EN	Discharging from silo enabled	Boolean

Table 1-3 User Defined Data Type SILO

You can then create an array of the silo data type. For ten silos, the tags would be SILO[0] to SILO[10]. If the silos are labeled 1 to 10, I would not use element zero in the program. Now, if we want to access the weight in silo one, the tag is SILO[1].WT.

Array Access and Pointers

When an array is created in the PLC, this array can be accessed in the ladder logic in two ways, directly and indirectly.

Direct Access

You are already familiar with direct access of an array. Let's assume we want to add up all of the weights in group of silos. If we use the example of the SILO[] array, then the weights of five silos can be added by using ADD instructions.

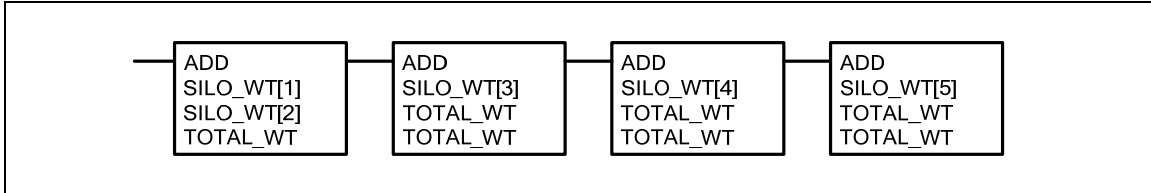


Figure 1-8 Direct Array Access

Indirect Array Access using Pointers

If we create a double integer variable called POINTER, then SILO_WT[POINTER] can be used to access the silo weight array. By setting POINTER to 2 we can access element 2 of the silo weight array, i.e. SILO_WT[2], with the expression SILO_WT[POINTER].

In Figure 1-9, the program scan is used as the looping mechanism to determine the total weight of all of the silos. The TOTAL_WT is updated on every fifth program scan. The temporary variable TEMP_WT is used to hold the summed weight until all of the silo weights have been added.

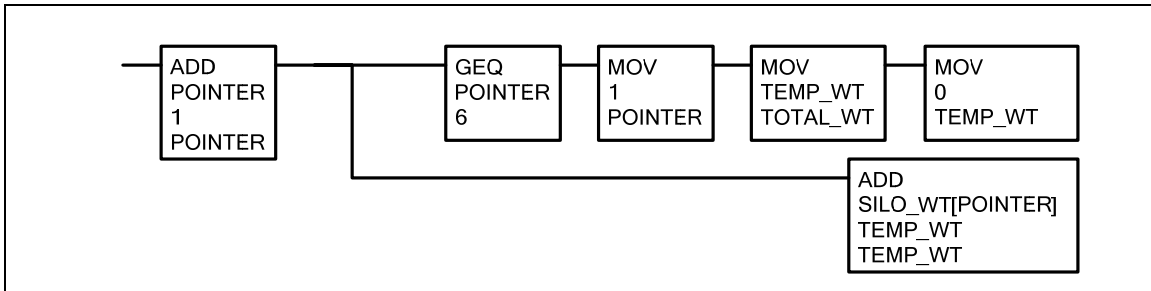


Figure 1-9 Using the Program Scan to Loop thru an Array

The FOR instruction can also be used to loop through logic using a pointer. In Figure 1-10 the total weight variable is initialized. The FOR instruction calls the WT_LOOP routine to add the silo weights. The FOR instruction is placed in the main body of the program.

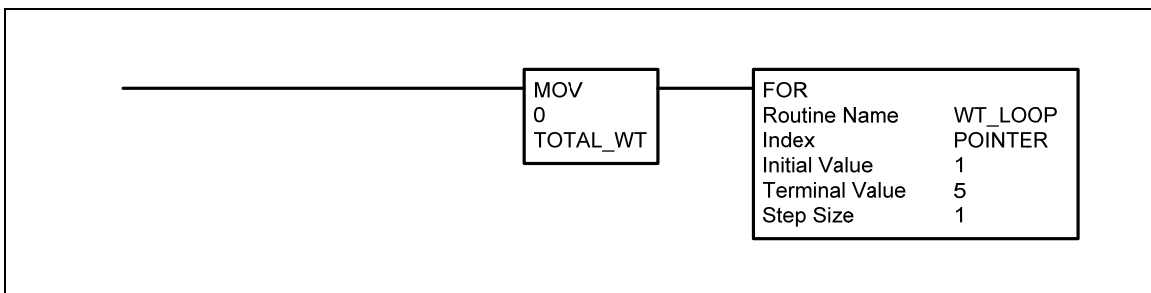


Figure 1-10 Using the FOR Instruction to Loop thru an Array

In Figure 1-11 the routine WT_LOOP adds the weight of each silo when it is called with the FOR instruction.

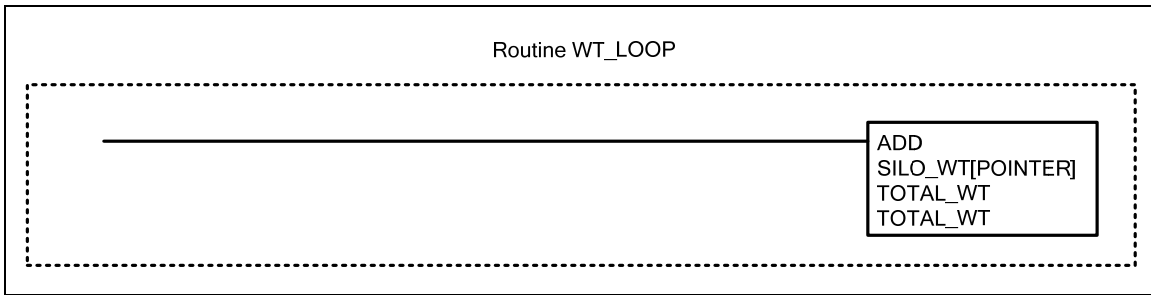


Figure 1-11 Routine WT_LOOP

In Figure 1-12 the FAL instruction is used to add the silo weights. The FAL instruction loops through the expression to determine the total weight.

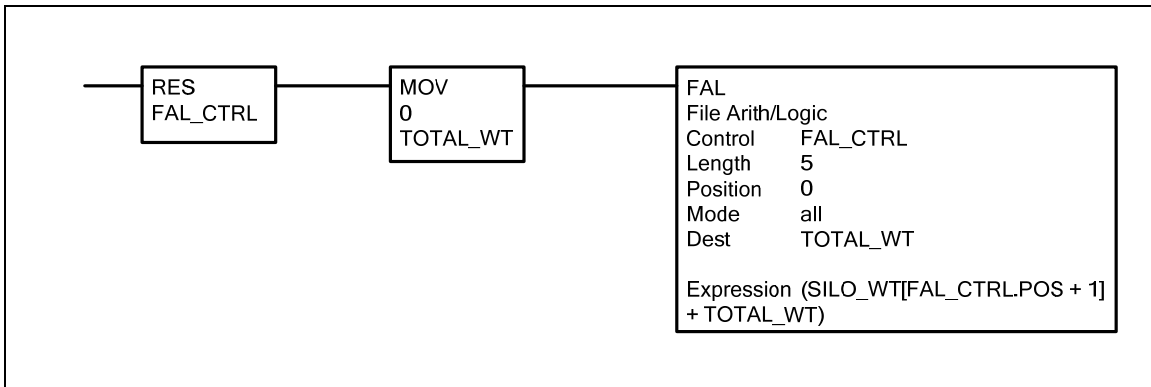


Figure 1-12 FAL instruction Loop

Aliasing

An Alias name allows you to assign a new tag name to an existing tag. An alias can be used to assign a tag name to an I/O point for example. Another use for an alias would be to assign a tag name to an array element. For example in our SILO[] array we could assign the alias tag UNLOAD_SILO to array element 1, SILO[1]. Now in your program you can use the more descriptive tag UNLOAD_SILO in place of SILO[1]. This new alias tag does not allocate a new storage location for your data. It only allows you to assign a new name to it. This new alias tag points to the original tag. Another common use assigns an alias to an I/O point. The alias name MIX_1_VLV1_ZSO could be assigned to the discrete I/O point Local:3.I.Data.0.

Program Structure

Structure your program. Group logical portions of your program together. The beginning of the process, conveyor or machine should usually be toward the beginning of the program with the end of process etc. toward the end.

The Control Logix processor uses tasks to organize and prioritize programs. The MainTask is scanned continuously. Periodic tasks are scanned at a specific period. Within a task programs contain the routines. The order in which the programs are scanned within the task is configurable. Routines contain the ladder logic within a program. Only the MainRoutine can be configured to execute within a program. Any other routines within the program must be called from the MainRoutine. Figure 1-13 shows the programming structure of the Control Logix processor.

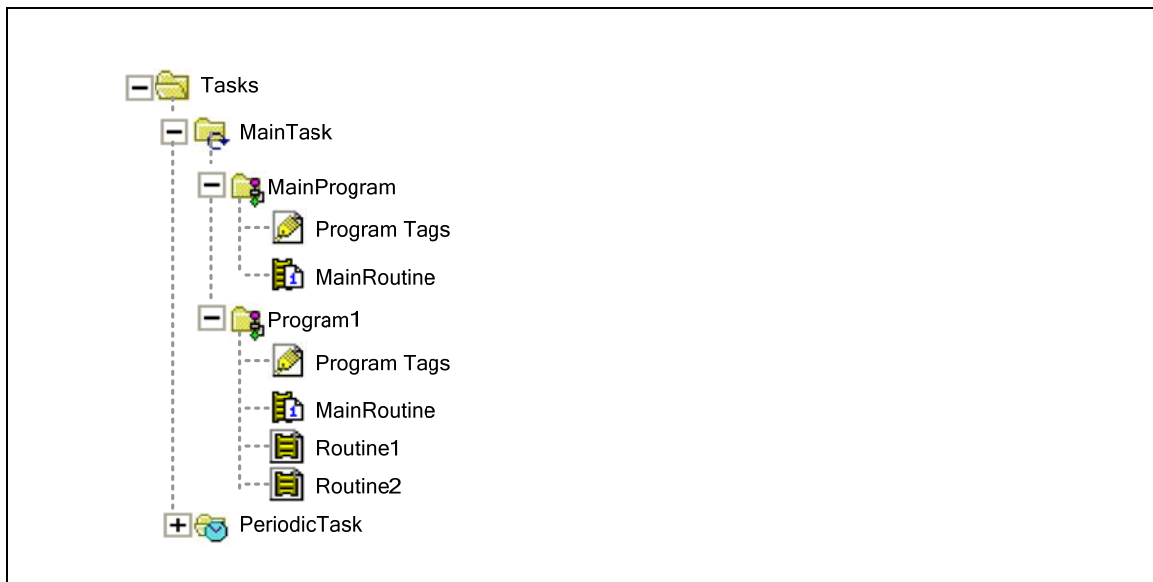


Figure 1-13 Control Logix Program Structure

Chapter 2 Documentation

Maintaining good documentation in any programming language is essential. I believe this is especially true in ladder logic and other control languages. You are rarely the only person who will be viewing your code. Most PLC programs will require that some person do troubleshooting using the online view of the ladder logic documentation. Good documentation will also allow you to navigate the program efficiently.

Symbols or tags should be used on all I/O, internal bits, counters, integers, floats etc. that are used in your program. **These same tags should also be used for the HMI application!** If an external simulator is used then the simulation should also use the same tag names as the HMI and the PLC. Use a spreadsheet to enter your tags and then import those tags into your programming documentation and HMI.

There are three primary camps when it comes to documentation.

- In **machines and conveyors** there are industry accepted documentation standards.
- In process, there is the **ISA process standard** which has been used for years prior to the development of the ISA batch standard.
- The **ISA batch standard** includes documentation standards which complement the batching methodology.

Machine and Conveyor Documentation

In machine and conveyor applications, you want to create symbol formats with an equipment abbreviation as the first character(s) of your symbol. This will then group your symbols by equipment when they are sorted alphabetically.

Create a symbol format based on your application. A typical symbol should be capable of expressing a description associated with that symbol. For example, the symbol ST1SVOP could represent the description “OPEN STOP ST1”. The symbol ST1SV could also represent the same description if it understood that the solenoid will open the stop. A typical symbol format is represented by the components EQUIPMENT, EQUIPMENT NUMBER, DEVICE, and FUNCTION.

[EQUIPMENT][EQUIPMENT NUMBER][DEVICE][FUNCTION]

Since the DEVICE such as a push button or solenoid is associated with I/O it is left out for internal tags.

[EQUIPMENT][EQUIPMENT NUMBER][FUNCTION]

Also, notice that in ST1SVOP the equipment number visually divides the tag making it easier to read. If the symbol abbreviations you are using get too long, or there is no number to divide the symbol, you may want to include underscores to separate each abbreviation. For example, you could use ST_1_SV_OP or ST1_SV_OP. What ever method you choose, be consistent.

Ladder logic Programming Techniques By Duane Snider

The same equipment number should be used if the equipment is physically grouped together. For example, if a work area contains a physical stop to stop the product, a clamp to secure it, and a lift to raise it then the equipment should be labeled ST3, CMP3 and LFT3.

The tags that you create should be used on the IO drawings. They can also be engraved into a plastic tag to identify the device in the field. For this reason, you should limit the size of your abbreviations.

Table 2-1 contains a list of equipment abbreviations and their associated descriptions that could be used on a conveyor or machine system.

<i>Equipment</i>	<i>Description</i>
AIR	AIR COMPRESSOR
BAN	BANDER
BAR	BELT DRIVEN ACCUMULATION ROLLER(ROLLER)
BC	BELT CONVEYOR
BDR	BELT DRIVEN LIME ROLLER(BELT)
BOR	BELT OVER ROLLER(BELT)
CAR	CAROUSEL
CC	CHAIN CONVEYOR
CDR	CHAIN DRIVEN LIVE ROLLER(CHAIN)
CMP	CLAMP
COE	CHAIN ON EDGE(CHAIN)
CP	CONTROL PANEL
CRN	CRANE
DIV	DIVERTER
DPAL	DEPALLETIZER
DR	DRIVE
DSP	DISPENSOR
HST	HOIST
LFT	LIFT
LN	LANE
LUB	LUBRICATOR
LVR	LIFTVEYOR
LWR	LOWERATOR
MRG	MERGE
OV	OVEN
PAL	PALLETIZER
PFD	POWER FACE DIVERTER
PLR	PULLER
PMP	PUMP
POS	POSITIONER
PSR	PUSHER
RC	ROLLER CONVEYOR
RES	RESERVOIR
SBD	SLIDER BED (BELT)

Chapter 2 Documentation

SLT	SLAT CONVEYOR
ST	STOP
STA	STATION
STK	STACKER
SW	TRACK SWITCH
TLT	TILT TABLE
TT	TURNTABLE
TTS	TILT TRAY SORTER
TU	TAKE-UP
WRP	WRAPPER
XFR	TRANSFER OR XFER

Table 2-1

Table 2-2 is a list of device abbreviations that are commonly used in a conveyor or machine application. I would not normally include the device description in tag description. It should be known by the abbreviation what the device is. For example for LFT1LSUP the description would be “LIFT 1 RAISED” and not “LIFT 1 RAISED LIMIT SWITCH”. The device type is either analog input, analog output, discrete input, discrete output, ASCII, or field.

<i>Device</i>	<i>Type</i>	<i>Description</i>
AM	ai	AMP METER
AMT	do	AIR MOTOR
CB	di	CIRCUIT BREAKER
CBC	do	CLUTCH BRAKE CONTROLLER
CR	do	CONTROL RELAY
CRX	di	CONTROL RELAY AUX
CRT	ascii	CRT DISPLAY
CT	f	CURRENT TRANSFORMER
DEC	ai	DECODER (SCANNER)
DISP	ascii	DISPLAY
DSC	di	DISCONNECT
ENC	ai	ENCODER
FL	do	FIELD LIGHT
FS	di	FOOT SWITCH
HC	do	HOLDING CONTACTOR
HCX	di	HOLDING CONTACTOR AUX.
HN	do	HORN
IK	di,do	INTERLOCK
INPT	f	INPUT STATION
ISB	f	INTRINSIC SAFE BARRIER
ITR	di	INSTANTANEOUS TRIP RELAY
KPD	ascii	KEYPAD
KS	di	KEYSWITCH
LS	di	LIMIT SWITCH
M	f	MOTOR
MS	do	MOTOR STARTER
MSX	di	MOTOR STARTER AUX.

Ladder logic Programming Techniques By Duane Snider

PB	di	PUSH BUTTON
PE	di	PHOTO EYE
PL	do	PILOT LIGHT
PS	di	PRESSURE SWITCH
PV	f	PANEL VIEW
PX	di	PROXIMITY SWITCH
SCA	ascii	SCANNER
SCL	ai	SCALE
SS	di	SELECTOR SWITCH
SV	do	SOLENOID VALVE
TR	di,do	TIMER
VFD	f	VARIABLE FREQ. DRIVE

Table 2-2

Table 2-3 contains a list of devices and functions. I have included the devices and functions together in one table to illustrate how the same abbreviation for the function can have a different description depending on whether the device is an input or an output. For example, the description for LSOP is “opened”. The description for SVOP is “open”. Use the past tense for the input and the present tense or command tense for the output. A push button will also use the command tense, open stop.

<i>Device & Function</i>	<i>IN/OUT</i>	<i>Description</i>
CR_RUN	I	RUNNING
HN	O	WARNING HORN
LS_AP	I	CARRIER APPROACHING
LS_CCW	I	COUNTER CLOCKWISE
LS_CL	I	CLOSED
LS_CLR	I	CARRIER CLEAR
LS_CLRL	I	CLEAR LEFT
LS_CLRR	I	CLEAR RIGHT
LS_CW	I	CLOCKWISE
LS_DEC	I	DECEL
LS_DECDN	I	DECEL DOWN
LS_DECUP	I	DECEL UP
LS_DN	I	LOWERED
LS_EOL	I	END OF LINE
LS_EXT	I	EXTENDED
LS_FULL	I	FULL
LS_L	I	LEFT
LS_OL	I	TORQUE OVERLOAD
LS_OP	I	OPENED
LS_POS1	I	POSITION 1
LS_POS2	I	POSITION 2
LS_POS3	I	POSITION 3
LS_PR	I	PRESENT
LS_R	I	RIGHT

Chapter 2 Documentation

LS_RET	I	RETRACTED
LS_UP	I	RAISED
MCR	I	MASTER CONTROL RELAY
MS	O	MOTOR STARTER
MS_OL	I	OVERLOAD
MSX	I	MOTOR STARTER AUXILIARY
PB_DN	I	LOWER
PB_EXT	I	EXTEND
PB_OP	I	OPEN
PB_REL	I	RELEASE CARRIER
PB_RET	I	RETRACT
PB_UP	I	RAISE
PE_PR	I	PRODUCT PRESENT
PL_DN	O	LOWERED
PL_FLT	O	FAULT
PL_OL	O	TORQUE OVERLOAD
PL_RUN	O	RUNNING
PL_UP	O	RAISED
PX_EXT	I	EXTENDED
PX_RET	I	RETRACTED
SSA	I	AUTO SELECT
SSM	I	MANUAL SELECT
SV_CCW	O	ROTATE COUNTER CLOCKWISE
SV_CL	O	CLOSE
SV_CW	O	ROTATE CLOCKWISE
SV_DN	O	LOWER
SV_EXT	O	EXTEND
SV_L	O	LEFT
SV_OP	O	OPEN
SV_R	O	RIGHT
SV_RET	O	RETRACT
SV_UP	O	RAISE

Table 2-3

ISA Process Standard

The original ISA documentation format consists of the DEVICE and the EQUIPMENT NUMBER.

[DEVICE][EQUIPMENT NUMBER]

The device is broken into components. The abbreviations for the device components are in Table 2-4

<i>Abbr.</i>	<i>Description</i>
T	Temperature
P	Pressure
F	Flow
L	Level
C	Conductivity
Z	Position
O	Open
C	Closed
H	Hand
C	Control
C	Controller
I	Indicator
V	Valve
T	Transmitter
E	Element
S	Switch
S	Solenoid
M	Motor
H	High
L	Low
X	Discrete

Table 2-4

The documentation for a typical flow control loop on vessel 100 is described in Table 2-5. These devices would be indicated on the engineering drawing for the vessel.

<i>Tag</i>	<i>Description</i>
FE_100	Flow Element
FT_100	Flow Transmitter
FIC_100	Flow Indicating Controller
FCV_100	Flow Control Valve

Table 2-5

Chapter 2 Documentation

If the PLC is used to control the flow then the flow transmitter is an analog input to the PLC. And, the flow control valve position is set by an analog output from the PLC. The flow controller then exists in the PLC. In Table 2-6 the PLC documentation associated with the control loop is described.

<i>Tag</i>	<i>Description</i>
FT_100	Flow input
FIC_100	PID controller
FIC_100_PV	Process Variable
FIC_100_CV	Control Variable
FIC_100_SP	Setpoint
FIC_100_DV	Deviation
FIC_100_SSA	Auto selector switch
FIC_100_SSR	Remote selector switch
FIC_100_DV_H	Deviation alarm high
FIC_100_DV_L	Deviation alarm low
FIC_100_PV_H	Process variable high alarm
FIC_100_PV_HH	Process variable high high alarm
FIC_100_PV_L	Process variable low alarm
FIC_100_PV_LL	Process variable low low alarm
FCV_100	Valve position output

Table 2-6

If your control system includes multiple flow control loops then sorting the PLC tags in alphabetic order puts all of the tags associated with the flow controllers together. The flow control valves would appear next, then the flow transmitter tags. This is why I do not like this method of documentation. In a large system it can be difficult to determine all of the devices that are associated with a loop. This becomes even more apparent when we consider the devices in a discrete valve as shown in Table 2-7.

<i>Tag</i>	<i>Description</i>
XCV_100	Discrete control valve
SVO_100	Open solenoid valve
SVC_100	Close solenoid valve
ZSO_100	Open limit switch
ZSC_100	Close limit switch
HS_100	Hand switch

Table 2-7

ISA Batch Standard

When the ISA batch standard came out, it included documentation standards that conformed to the batch structure. Field devices are now associated with a unit. A unit can be a tank, a mixer, or a reactor for example. Instead of using a three digit equipment number we can create an abbreviation that corresponds to the unit. The device is then tagged with the unit followed by the device.

[UNIT][DEVICE]

<i>Tag</i>	<i>Description</i>
R1_FE	Reactor 1 Flow Element
R1_FT	Reactor 1 Flow Transmitter
R1_FIC	Reactor 1 Flow Indicating Controller
R1_FCV	Reactor 1 Flow Control Valve

Table 2-8

If there are multiple flow transmitters on the reactor then they are suffixed with a number. For example, R1_FT_1 and R1_FT_2.

When the tags are sorted alphabetically, all of the tags associated with the unit are kept together. This makes it a preferable method to the original ISA standard.

Chapter 3 Controllers

In most process systems each physical device such as a motor or valve will have logic that controls the operation of that device. This controller logic handles the mode of the device such as automatic, remote or manual. The mode is usually set by an operator through an HMI. The controller also has setpoints which can then be set by an automatic sequence or some other source outside of the controller logic. An HMI screen will have a graphic representation of the physical process. The operator can display a faceplate for the device by selecting it on the HMI graphic screen. This faceplate will then allow the operator to change modes on the controller and enter setpoints. In this chapter we will look at several common types of controllers and their associated faceplates, the discrete motor, the variable frequency drive, the single solenoid valve, the double solenoid valve, the PID controller, and the Ratio controller.

Figure 3-1 shows a process mixer.

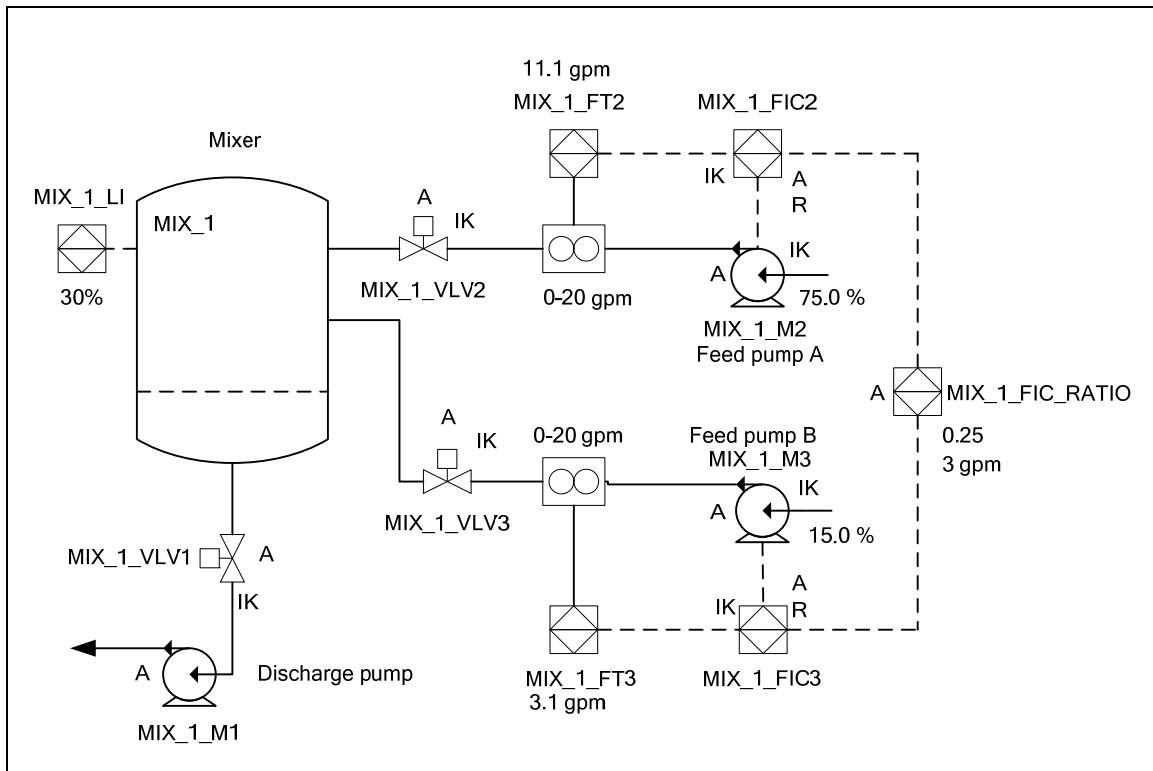


Figure 3-1 HMI Mixer process

The typical animations for a valve are listed in Figure 3-1. The discrete motor animations would be similar. The valve body will change colors based upon the position of the open and closed limit switches. If the valve is in transition the color will change to yellow. Red is closed and green is open. I have seen some customers who have reversed these colors. The thinking being that green is the safe off state and red is the unsafe on state. The valve will flash if the alarm bit is on. Of course we have to choose two colors when flashing. I prefer to flash between the current animated color and a neutral color like black or white. This is usually the kind of thing

Ladder logic Programming Techniques By Duane Snider

that you have everyone to agree upon at the beginning of the job and then you still end up changing it twice on one hundred plus valves before the end of the job.

<i>Device</i>	<i>Animations</i>	<i>Description</i>	<i>Logic</i>
MIX_1_VLV1	Red	Closed	MIX_1_VLV1_ZSC
	Green	Open	MIX_1_VLV1_ZSO
	Yellow	In transition	NOT (MIX_1_VLV1_ZSC OR MIX_1_VLV1_ZSO)
	Flashing	Alarm	MIX_1_VLV1.ALM
A	Black	Auto mode	MIX_1_VLV1.SSA
M	Red	Manual mode	NOT MIX_1_VLV1.SSA
IK	Red visible	Interlocked	NOT MIX_1_VLV1.IK

Table 3-1 HMI Animations

Figure 3-2 shows how the mixer plc program can be organized. The logic for each controller is in a separate routine. The MainProgram is used for common logic. The MainRoutine in the MIX_1 program contains the jump to subroutine instructions (JSR) for each routine. Separate program files contain the controllers for mixer 2 and 3. The ANALOG_ALARM routine is called by the level indicator MIX_1_LI and the flow controllers MIX_1_FIC1 and MIX_1_FIC2. This routine would have to be duplicated under the MIX_2 and MIX_3 program files because a subroutine can not be called from one program file to another.

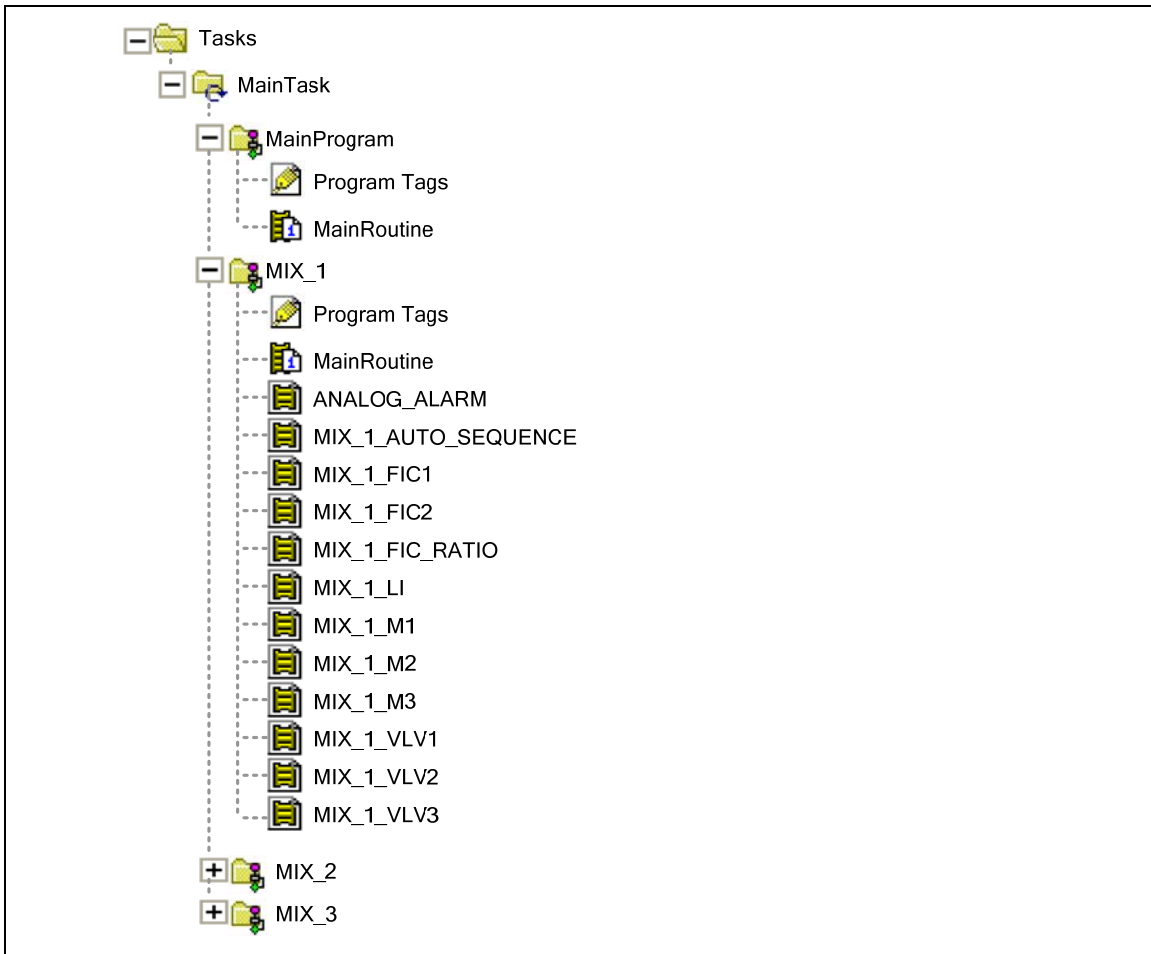


Figure 3-2 Controller Program Organization

The logic for each controller type is described in the sections that follow. Sequence logic is described in Chapter 4.

Discrete Motor Controller

The faceplate for a discrete motor controller is shown in Figure 3-3. This faceplate is shown when the operator selects the motor on the overview graphic screen. The interlocks faceplate is displayed directly adjacent to the motor faceplate when the Interlocks pushbutton is pressed at the bottom of the motor faceplate.

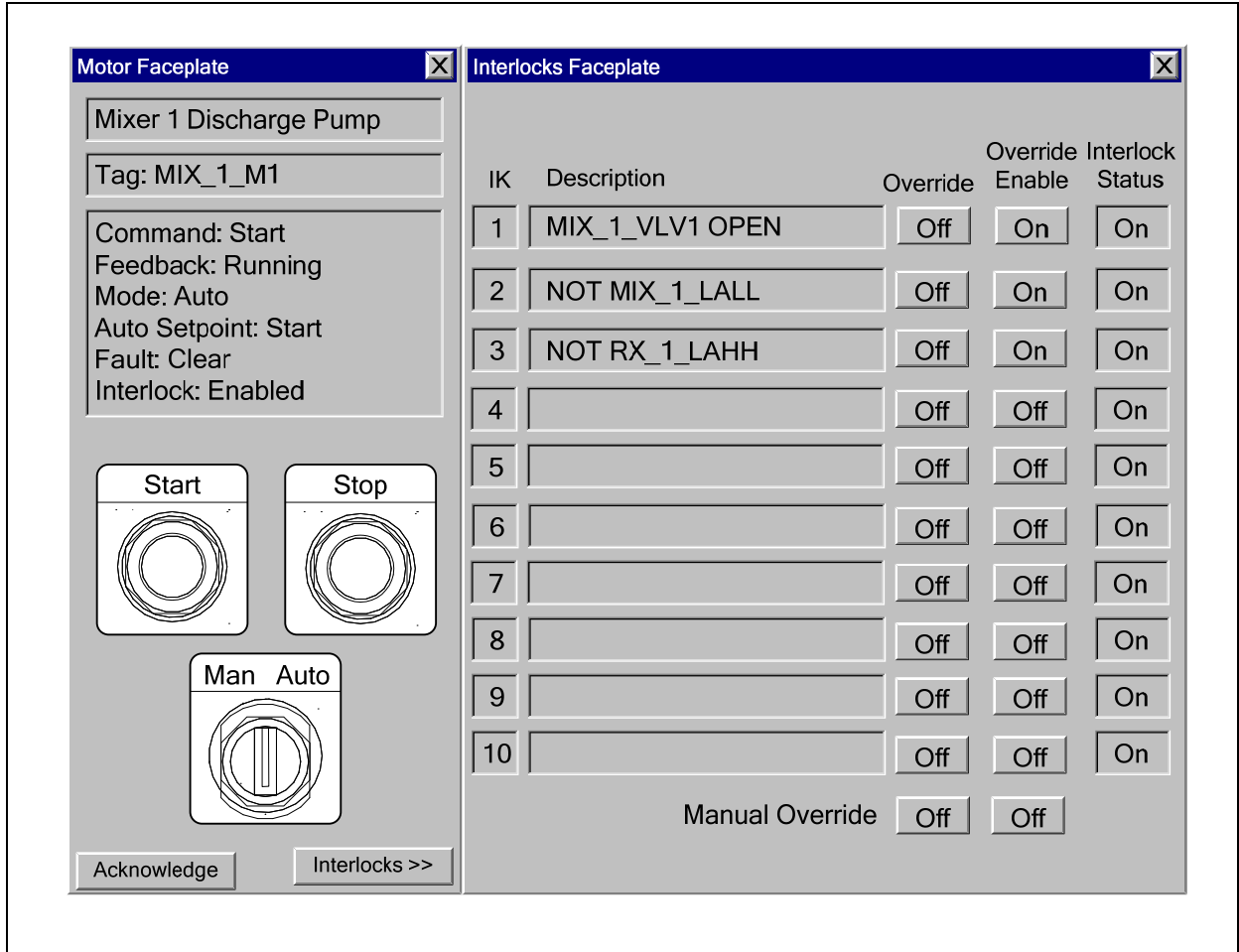


Figure 3-3 Discrete Motor Controller Faceplate

The HMI you are using may have some built in faceplates that you can use or adapt to your application. However, in most cases you will have to build your own faceplates. Most HMIs support the use of some sort of indirect tags or parameters that will allow you to create only one faceplate for all of the controllers of that same type in your system. If you have designed your tags right, you may then only have to pass the faceplate the prefix for all of the tags that will be displayed. In the case of this motor faceplate we could pass to the faceplate the prefix MIX_1_M1. All of the tags and information could then be derived by appending the tag suffix to this common tag prefix. Some information on the faceplate may not exist in a tag in the PLC, for example, the description of the motor. For this type of descriptive information there are a few options to consider. We can create a tag in the HMI that does not reference a tag in the PLC. This string tag would then be set when the faceplate is selected on the overview screen. We could also

set the description of a tag that we are already using to the description of the motor that will be displayed. This will reduce the number of HMI tags which is an important consideration if you are paying for your HMI on a per tag basis. Another option is to include this string tag in the PLC user defined data type for the motor. A few years ago I would have rejected this option because it would waste PLC memory, and it would require communication resources to the PLC. There are however a few good reasons why this may be a preferable option. If there are multiple HMIs, any change to the description of the motor would only have to be made one time in the PLC. You may also find it more convenient to make the change in the PLC versus the HMI software.

The interlock faceplate also requires a description for each interlock. We can use one of the methods described for the motor description. Or, if you have an application that requires many interlocks, some of which are used on multiple valves or motors, you may want to put the interlock descriptions in an external database. An interlock number could be included in the PLC user defined data type for each interlock. This number would be set by you, the programmer. The HMI could then look up this number in the interlocks database and extract the description for that interlock. These interlock descriptions could also exist in the PLC, either in the motor user defined data type or in a separate interlocks description array.

In Table 3-2 we define some string types that we can then use in our user defined data type. These strings will be displayed on the HMI faceplate.

<i>String</i>	<i>Description</i>	<i>Size</i>
DESC_STR	HMI controller name	30 characters
TAG_STR	HMI tag name	20 characters
UNITS_STR	HMI unit abbreviation	10 characters
IK_DESCR	HMI interlock description	25 characters

Table 3-2 Controller String Definitions

A user defined data type MOTOR is defined for the motor controller in Table 3-3. In our motor example, the variable MIX_1_M1 is defined as this variable type MOTOR.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	HMI description	DESC_STR
TAG	HMI tag name	TAG_STR
SSA	HMI Auto select	Boolean
PB_START	HMI start push button	Boolean
PB_STOP	HMI stop push button	Boolean
AUTO	Auto bit set by remote sequence	Boolean
ALM	Alarm	Boolean
ACK	HMI Alarm Acknowledge	Boolean
MAN_OVR	Interlock manual override	Boolean
MAN_OVR_EN	Interlock manual override enable	Boolean
IK	Interlock sum	Boolean
IKS	Interlock bits	Dint
IK_OVR	Interlock override bits	Dint

IK_OVR_EN	HMI Interlock override enable bits	Dint
IK_VISIBLE	HMI Interlock visible bits	Dint
IK_DESC	HMI Interlock description	IK_DESC[11]
ALM_TM1	Alarm timer 1	Timer
ALM_TM2	Alarm timer 2	Timer

Table 3-3 User Defined Data Type MOTOR

In Table 3-4 an alias is assigned to the I/O for the motor.

<i>I/O Alias</i>	<i>Description</i>	<i>Type</i>
MIX_1_M1_HC	Motor Holding Contactor Output	Boolean
MIX_1_M1_HCX	Motor Holding Contactor Auxiliary contact	Boolean

Table 3-4 Discrete Motor I/O Definitions

In Figure 3-4, each motor interlock is programmed. The variable MIX_1_M1.IKS.1 is bit 1 of the double integer (Dint) for sub-element IKS in the MIX_1_M1 variable. Bit 0 exists but I have elected not to use it so that my interlocks will be numbered starting with 1.



Figure 3-4 Discrete Motor Controller Interlocks

These IKS bits exist so that they can be easily referenced on the motor interlock faceplate. Multiple conditions can also be programmed into each interlock.

In Figure 3-5 the interlocks are summed into MIX_1M1.IK. This variable is displayed on the HMI process overview screen Figure 3-1 indicating that all interlocks are clear.

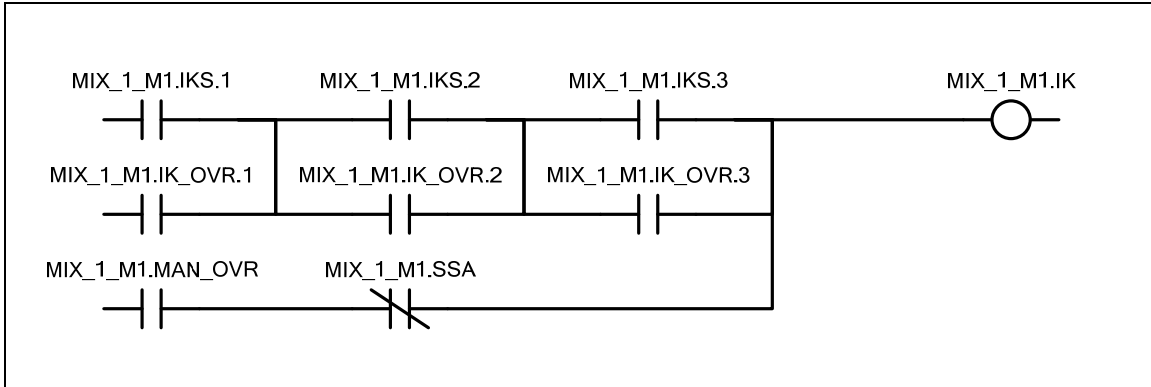


Figure 3-5 Discrete Motor Controller Interlock summation

The operator can override each interlock with MIX_1_M1.IK_OVR.1 for example. This button is disabled on the HMI if the override enable bit is not set, MIX_1_M1.IK_OVR_EN.1. The override enable bit would usually be set by an engineer or a supervisor as part of the system definition. Either the PLC programming software or the HMI could be used to set the interlock override enable. The variable MIX_1_M1.MAN_OVR determines if the interlocks are bypassed when the mode of the motor is changed by the operator to manual. This button also has an enable button that would be set from the HMI by an engineer or supervisor.

In Figure 3-6 the motor contactor MIX_1_M1_HC is programmed. This variable is defined as an alias to the physical output connected to the coil on the motor contactor. The motor will operate in the auto mode i.e. MIX_1_M1.SSA is on. Or, it will operate in manual with MIX_1_M1.SSA off. The variable MIX_1_M1.AUTO would be set by some other part of the program which would cause the motor to start in the automatic mode. The start and stop push buttons allow the motor to be started in manual mode.

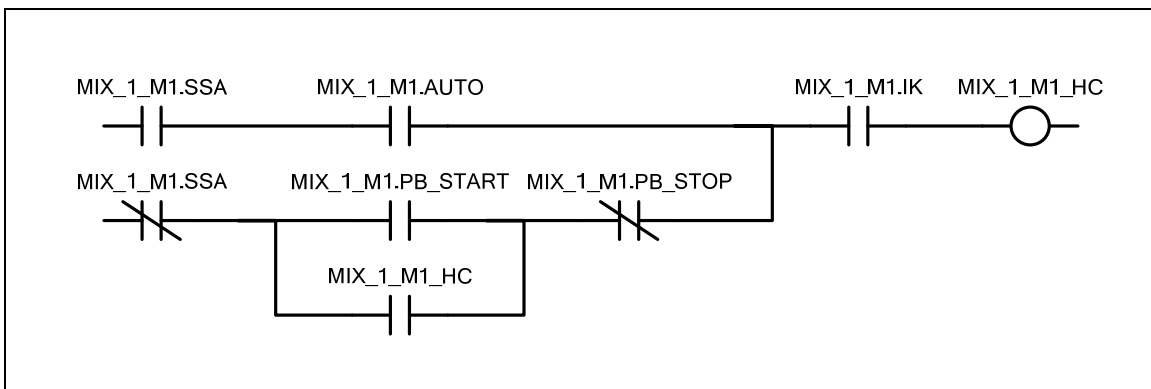


Figure 3-6 Discrete Motor Controller Contactor Circuit

In Figure 3-7 the motor alarm is programmed. An acknowledge from the HMI will reset the alarm. The variable MIX_1_HCX is an alias to the physical input connected the motor contactor auxiliary.

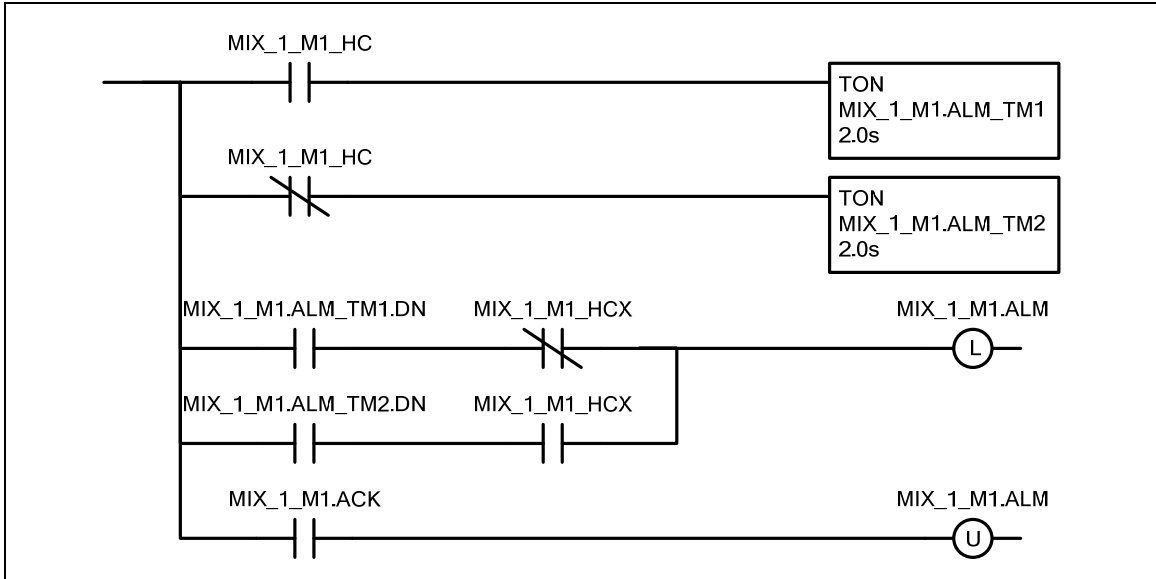


Figure 3-7 Discrete Motor Controller Alarm Logic

Variable Frequency Drive

The faceplate for a variable frequency drive controller is shown in Figure 3-8. This faceplate is shown when the operator selects the motor on the overview graphic screen. This faceplate allows the operator to change the mode of the controller to auto or manual. In manual, the start and stop pushbuttons are used to control the motor. The remote/local mode determines the source of the setpoint. In the local mode the operator can change the setpoint from the faceplate. In the remote mode the setpoint would come from the remote setpoint. The remote setpoint could be set by some other controller or an automatic sequence for example.

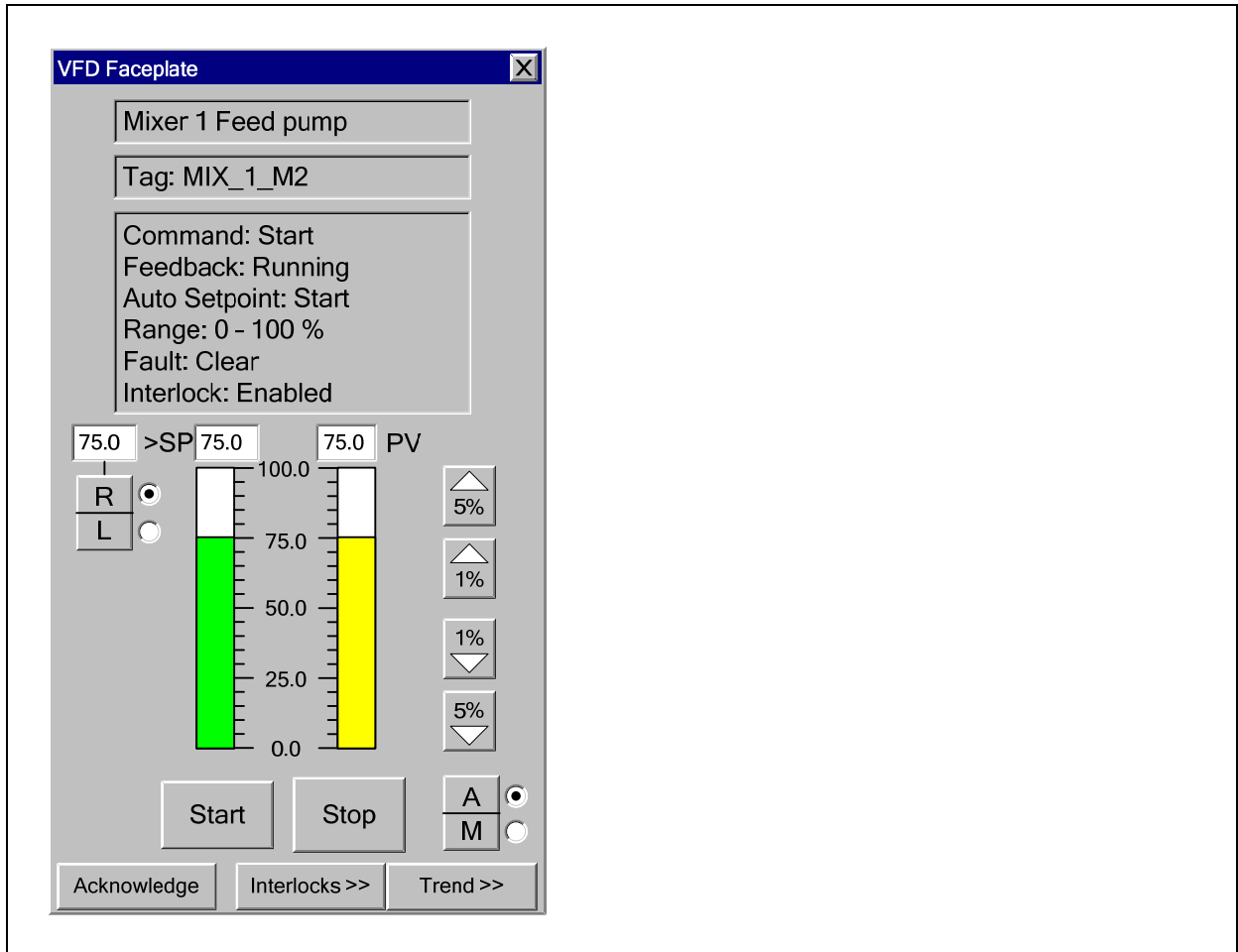


Figure 3-8 Variable Frequency Drive Controller Faceplate

The acknowledge button will reset a fault on the drive. The interlocks button displays the interlock faceplate. The trend button displays a trend of the setpoint and the process variable. The process variable is the speed feedback signal from the drive. In the manual mode the setpoint can be adjusted in increments of 1 and 5 percent.

The user defined data type for the variable frequency drive is shown in Table 3-5. You would create this variable type with the name VFD. The variable MIX_1_M2 would then be defined as type VFD.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	HMI description	DESC_STR
TAG	HMI tag name	TAG_STR
SSA	HMI Auto select	Boolean
SSR	HMI Remote select	Boolean
PB_START	HMI start push button	Boolean
PB_STOP	HMI stop push button	Boolean
AUTO	Auto bit set by remote sequence	Boolean
SP	Setpoint	Real
SP_REM	Remote Setpoint	Real
PV	Process Variable	Real
ALM	Alarm	Boolean
ACK	HMI Alarm Acknowledge	Boolean
MAN_OVR	Interlock manual override	Boolean
MAN_OVR_EN	Interlock manual override enable	Boolean
IK	Interlock sum	Boolean
IKS	Interlock bits	Dint
IK_OVR	Interlock override bits	Dint
IK_OVR_EN	HMI Interlock override enable bits	Dint
IK_VISIBLE	HMI Interlock visible bits	Dint
IK_DESC	HMI Interlock description	IK_DESC[11]
ALM_TM1	Alarm timer 1	Timer
ALM_TM2	Alarm timer 2	Timer

Table 3-5 User Defined Data Type VFD

As you can see, the VFD variable type is similar to the MOTOR variable type. Variables associated with the speed of the drive are also included.

The program for the VFD is the same as the discrete motor controller. The VFD controller however, also includes code associated with the speed of the VFD. In Figure 3-9 the remote setpoint of the controller is moved into the setpoint. If the controller is in the local mode, then the operator can change the setpoint from the faceplate.

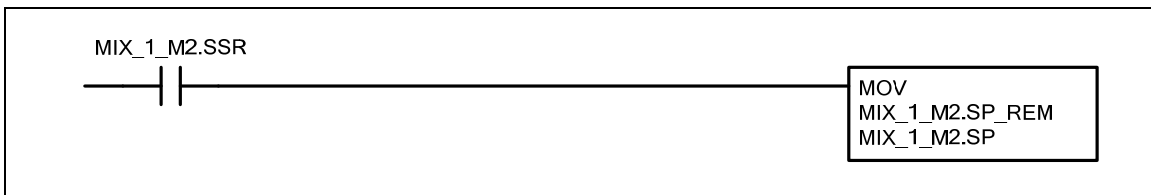


Figure 3-9 Variable Frequency Drive Controller Using the Remote Setpoint

Chapter 3 Controllers

The setpoint MIX_1_M2.SP still has to get scaled to the physical output. If the output is on a standard Control Logix analog card, then the scaling can be done by changing parameters on the card configuration. MIX_1_M2.SP would just be moved into MIX_1_M2_OUT_RAW, which is aliased to the output. On other I/O systems such as the Flex I/O, there are no provisions to scale the output on the card. In this case the output must be scaled in the PLC logic. Table 3-6 shows the user defined data type SCALING. The variable MIX_1_M2_OUT_SCL is then defined as the SCALING data type.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
RAW_MIN	Raw unscaled minimum	Real
RAW_MAX	Raw unscaled maximum	Real
EGU_MIN	Engineering unit minimum	Real
EGU_MAX	Engineering unit maximum	Real

Table 3-6 User Defined Data Type SCALING

In Figure 3-10 the speed setpoint for the VFD is scaled into the physical output connected to the drive speed reference.

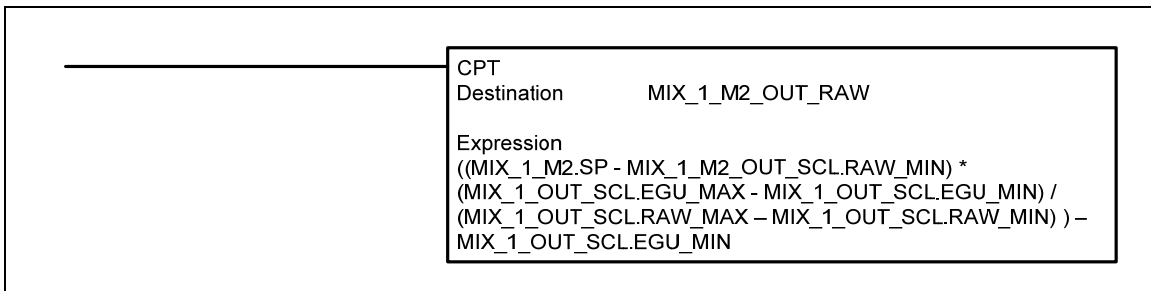


Figure 3-10 Variable Frequency Drive Controller Scaling the Output

The PLC can also communicate to a drive via some network such as ControlNet, DeviceNet, Ethernet, Profibus etc. Each of these networks would require different logic to pass information back and forth.

Single Solenoid Valve Controller

The faceplate for a single solenoid valve is shown in Figure 3-11. The operator can open and close the valve in the manual mode.

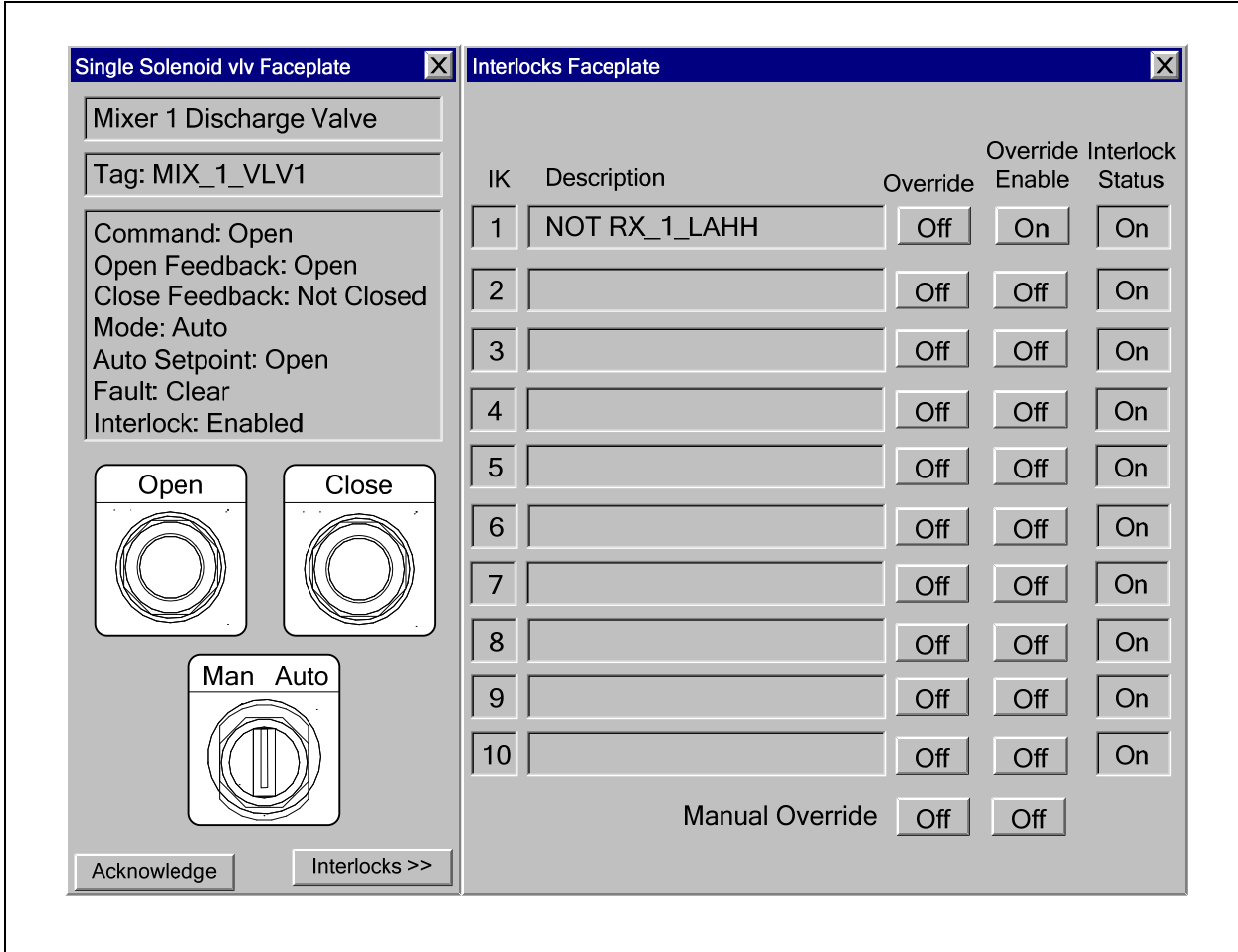


Figure 3-11 Single Solenoid Valve Controller Faceplate

If the open feedback or the closed feedback does not exist for the valve, then an HMI variable can be created which could toggle the visibility for these feedbacks on the faceplate. This variable could be set when the faceplate is displayed. This will keep you from having to create a separate faceplate for those valves where the feedbacks do not exist. This visibility variable could also be put in the PLC user defined data type for the valve.

Chapter 3 Controllers

Table 3-7 contains each variable in the user defined type SINGLE_SV. A variable of this type would be created for each single solenoid valve in your application. The variable MIX_1_VLV1 would be defined as type SINGLE_SV.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	HMI description	DESC_STR
TAG	HMI tag name	TAG_STR
SSA	HMI Auto select	Boolean
PB_OPEN	HMI open push button	Boolean
PB_CLOSE	HMI close push button	Boolean
AUTO	Auto bit set by remote sequence	Boolean
ALM	Alarm	Boolean
ACK	HMI Alarm Acknowledge	Boolean
MAN_OVR	Interlock manual override	Boolean
MAN_OVR_EN	Interlock manual override enable	Boolean
ZSO_VISIBLE	Open feedback visible on HMI	Boolean
ZSC_VISIBLE	Closed feedback visible on HMI	Boolean
IK	Interlock sum	Boolean
IKS	Interlock bits	Dint
IK_OVR	Interlock override bits	Dint
IK_OVR_EN	HMI Interlock override enable bits	Dint
IK_VISIBLE	HMI Interlock visible bits	Dint
IK_DESC	HMI Interlock description	IK_DESC[11]
ALM_TM	Alarm timer	Timer

Table 3-7 User Defined Data Type SINGLE_SV

In Table 3-8 an alias is assigned to the I/O for the valve.

<i>I/O Alias</i>	<i>Description</i>	<i>Type</i>
MIX_1_VLV1_SV	Valve open output	Boolean
MIX_1_VLV1_ZSO	Valve open limit switch	Boolean
MIX_1_VLV1_ZSC	Valve close limit Switch	Boolean

Table 3-8 Single Solenoid Valve Controller I/O Aliases

Each interlock is programmed in Figure 3-12.

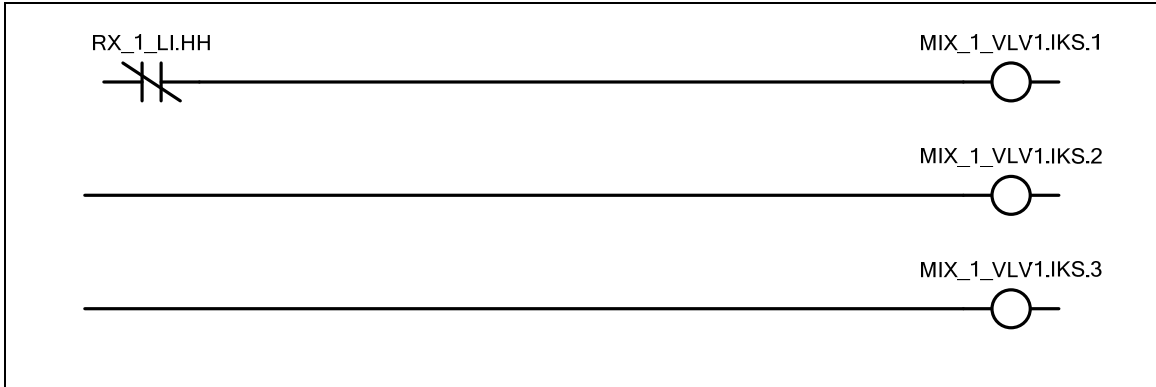


Figure 3-12 Single Solenoid Controller Interlocks

Like the discrete motor controller, the interlocks are summed in Figure 3-13 for the single solenoid valve controller.

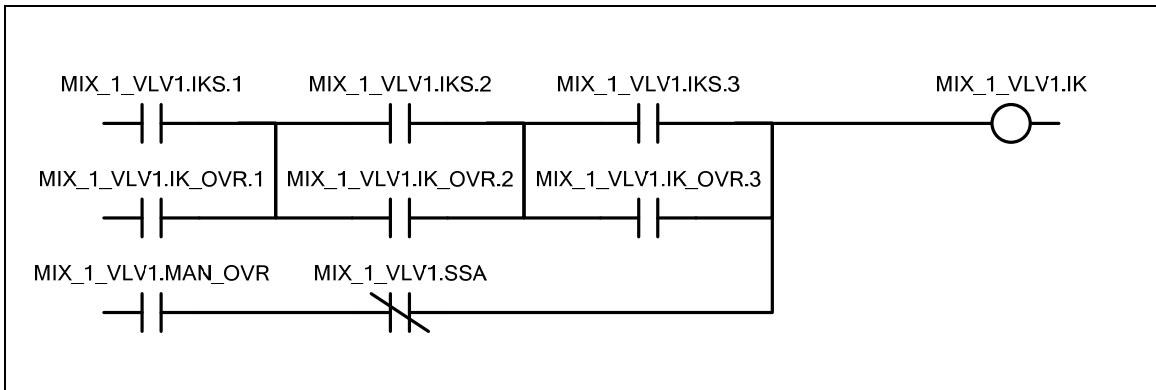


Figure 3-13 Single Solenoid Valve Controller Interlock Summation

Figure 3-14 shows the output logic for the single solenoid. MIX_1_VLV1_SV would be aliased to the physical discrete output which energizes the solenoid.

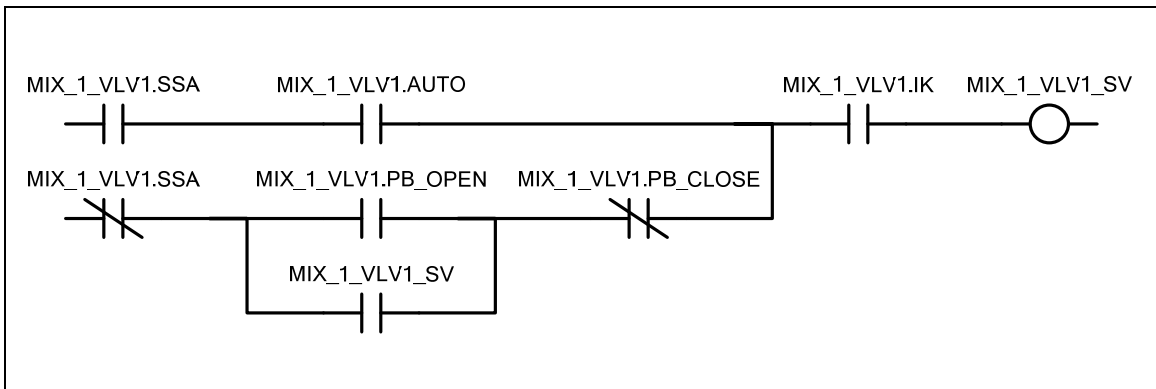


Figure 3-14 Single Solenoid Valve Controller Output Logic

Figure 3-15 shows the alarm logic for the single solenoid valve controller. MIX_1_VLV1_ZSO and MIX_1_VLV1_ZSC are the open and closed position switches for the valve. The alarm will occur if the valve does not reach the open or closed position within three seconds after the solenoid output has changed states.

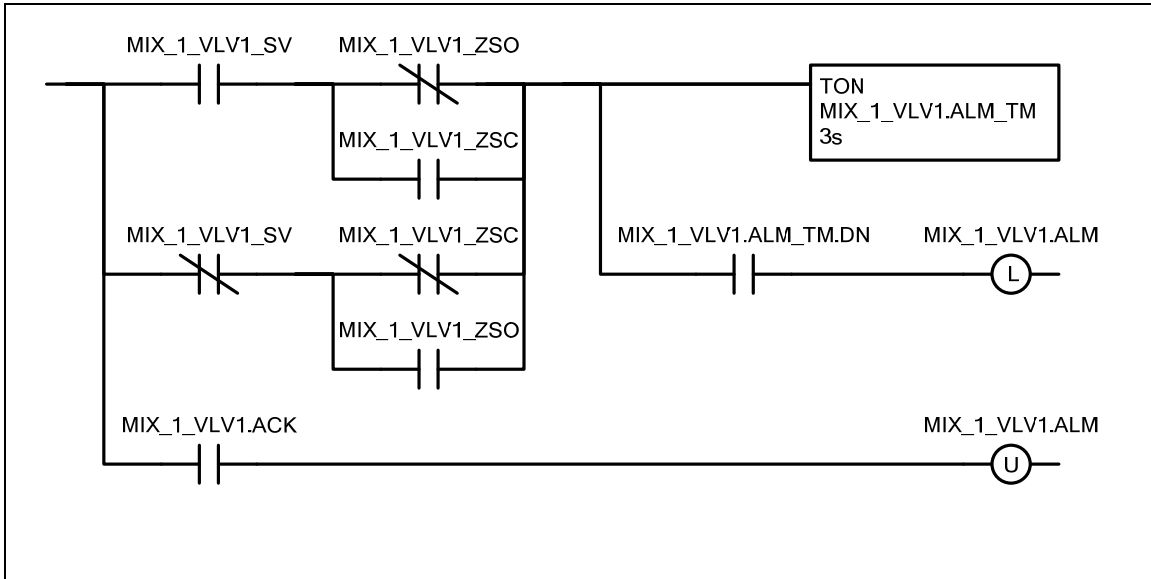


Figure 3-15 Single Solenoid Valve Controller Alarm Logic

Double Solenoid Valve Controller

The faceplate and the user defined data type for a double solenoid valve controller are the same as the single solenoid valve faceplate. The ladder logic is different to account for the additional output.

In Figure 3-16 the open output for the double solenoid is programmed.

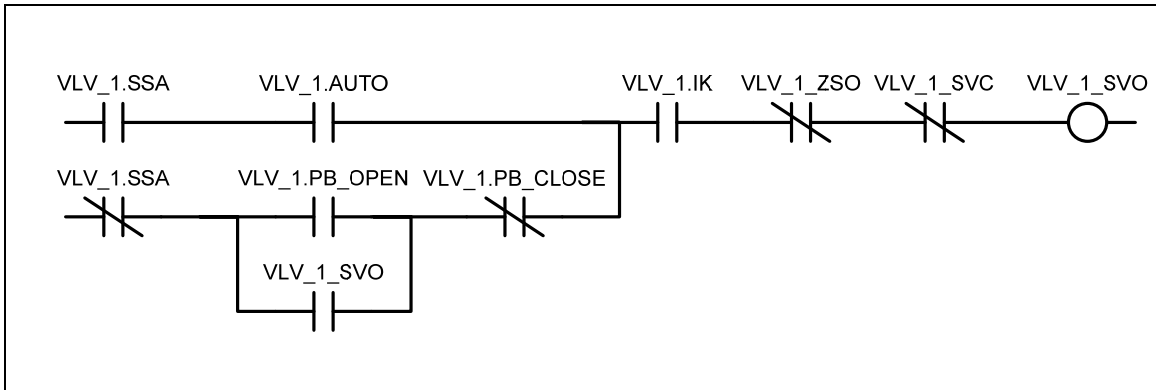


Figure 3-16 Double Solenoid Valve Controller Open Output Logic

In Figure 3-17 the closed output is programmed for the double solenoid. In this case, if the interlock permissive is lost then the valve is forced closed.

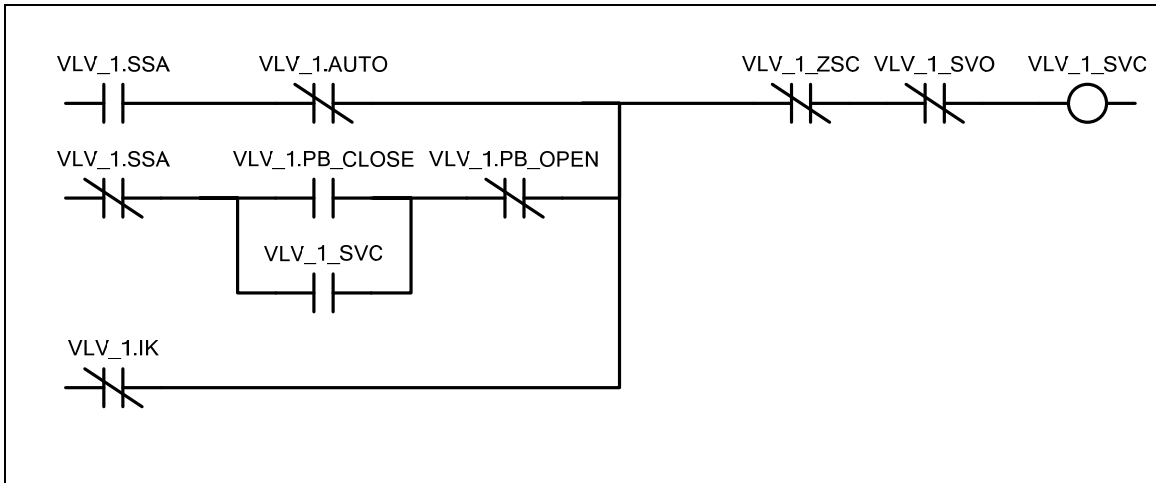


Figure 3-17 Double Solenoid Valve Controller Close Output Logic

The alarm for a double solenoid valve is shown in Figure 3-18. Because either the outputs do not have to remain on, the logic checks that one or the other of the position switches are made.

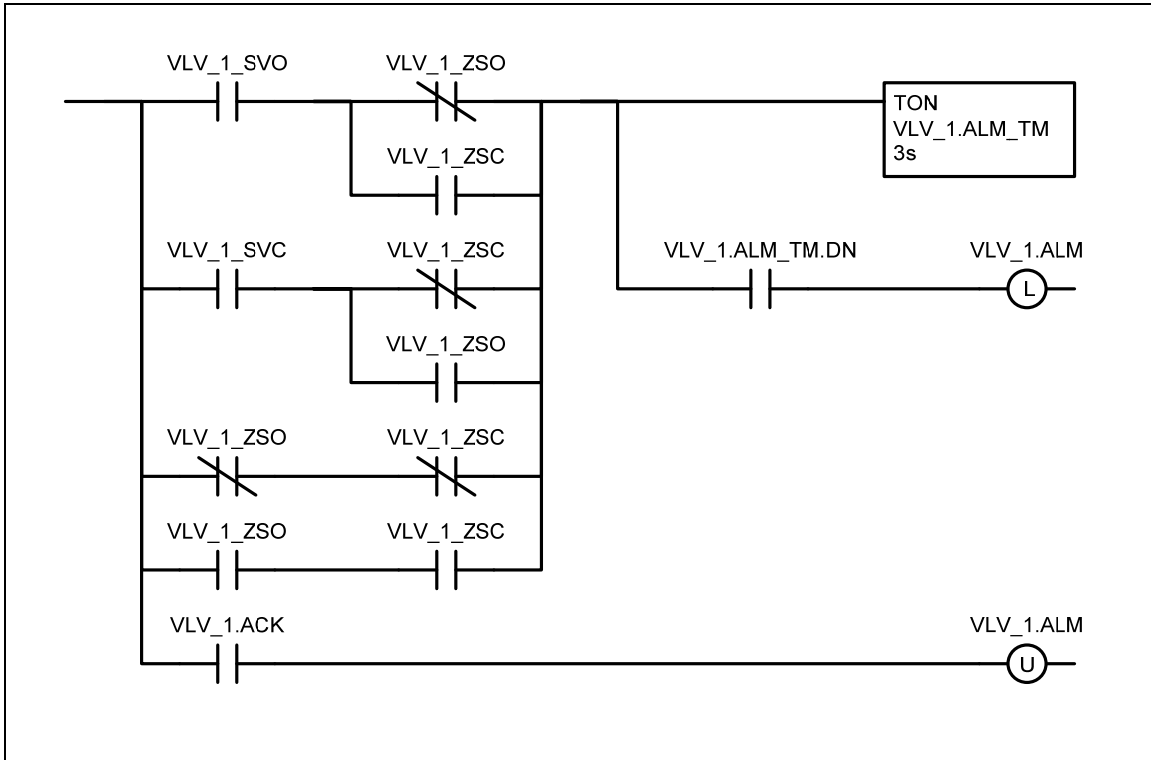


Figure 3-18 Double Solenoid Valve Controller Open Alarm Logic

Analog Indicator with Alarms

The faceplate for an analog indicator with alarms is shown in Figure 3-19. Each alarm that is associated with the process variable is displayed as a triangle to the right of the process variable bar graph. The position of these indicators along the bar graph are determined by the setpoint of the alarm. The triangles are visible if the alarm has been enabled. The state of the alarm does not determine the visibility of the triangle.

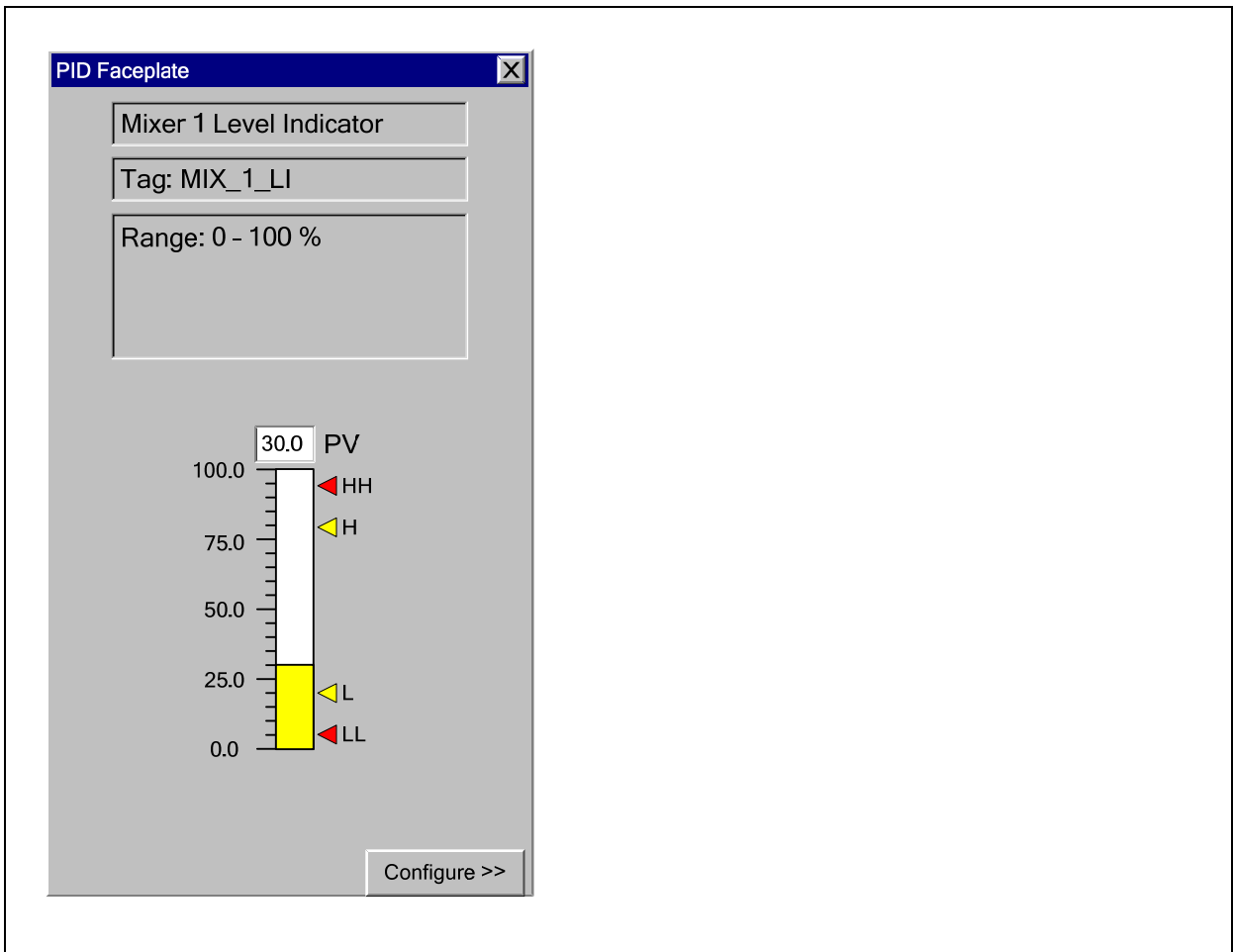


Figure 3-19 Analog Indicator Faceplate

The alarm setpoints for the indicator are set on the faceplate in Figure 3-20.

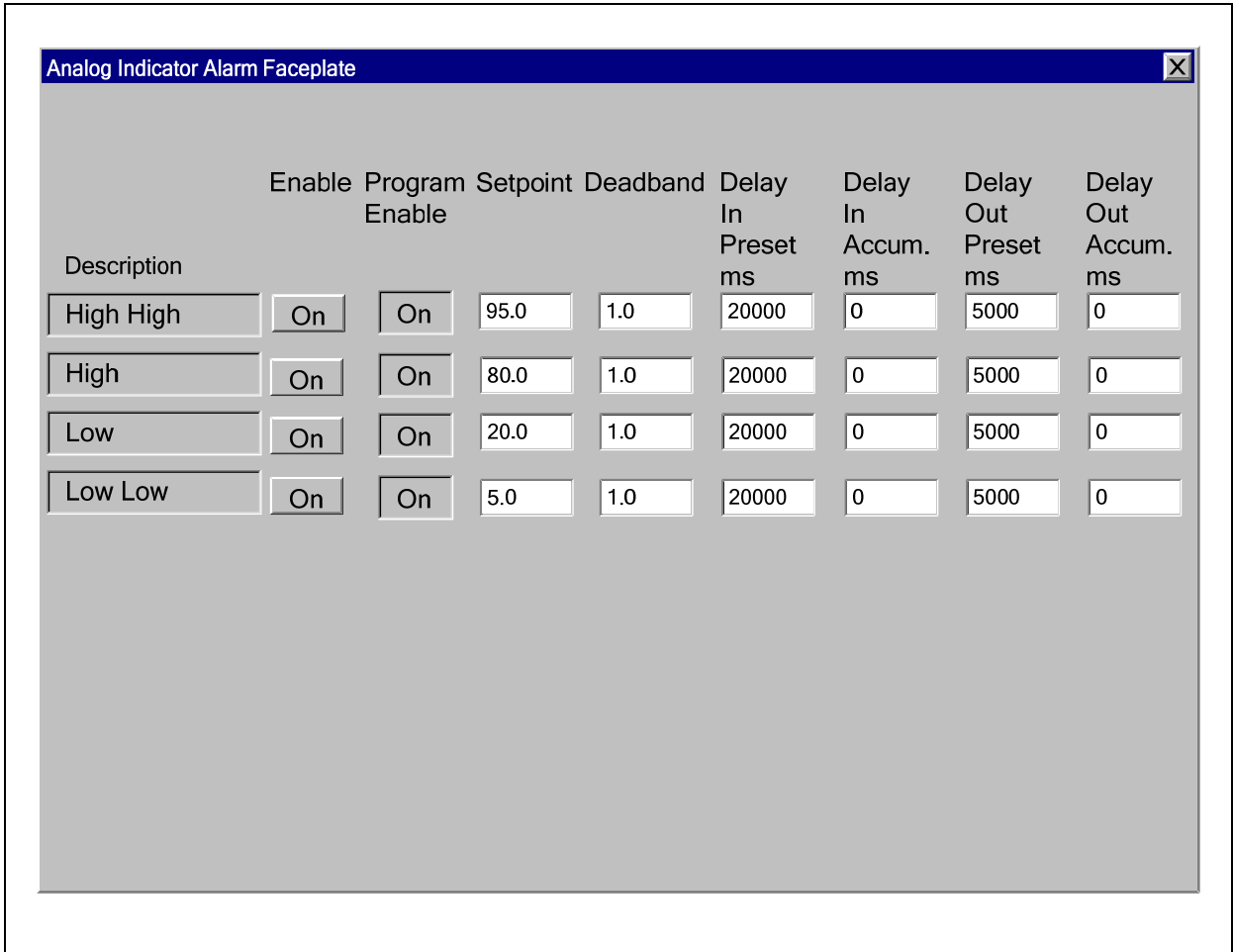


Figure 3-20 Analog Alarm Faceplate

The user defined data type for the analog indicator is in Table 3-9. The data type would be called ANA_ALM. The variable MIX_1_LI would be defined using this type.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	HMI description	DESC_STR
TAG	HMI tag name	TAG_STR
UNITS	HMI units	UNITS_STR
EGUH	High engineering unit	Real
EGUL	Low engineering unit	Real
IN	Input	Real
H_SP	High alarm setpoint	Real
H	High alarm	Boolean
H_EN	High alarm enable	Boolean
H_PROG_EN	High alarm program enable	Boolean
H_ACK	High alarm acknowledge	Boolean

H_ACK_EN	High alarm acknowledge enable	Boolean
H_TM_ON	High alarm delay on	Timer
H_TM_OFF	High alarm delay off	Timer
H_DB	High alarm deadband	Real
HH_SP	High High alarm setpoint	Real
HH	High High alarm	Boolean
HH_EN	High High alarm enable	Boolean
HH_PROG_EN	High High alarm program enable	Boolean
HH_ACK	High High alarm acknowledge	Boolean
HH_ACK_EN	High High alarm acknowledge enable	Boolean
HH_TM_ON	High High alarm delay on	Timer
HH_TM_OFF	High High alarm delay off	Timer
HH_DB	High High alarm deadband	Real
L_SP	Low alarm setpoint	Real
L	Low alarm	Boolean
L_EN	Low alarm enable	Boolean
L_PROG_EN	Low alarm program enable	Boolean
L_ACK	Low alarm acknowledge	Boolean
L_ACK_EN	Low alarm acknowledge enable	Boolean
L_TM_ON	Low alarm delay on	Timer
L_TM_OFF	Low alarm delay off	Timer
L_DB	Low alarm deadband	Real
LL_SP	Low Low alarm setpoint	Real
LL	Low Low alarm	Boolean
LL_EN	Low Low alarm enable	Boolean
LL_PROG_EN	Low Low alarm program enable	Boolean
LL_ACK	Low Low alarm acknowledge	Boolean
LL_ACK_EN	Low alarm acknowledge enable	Boolean
LL_TM_ON	Low Low alarm delay on	Timer
LL_TM_OFF	Low Low alarm delay off	Timer
LL_DB	Low Low alarm deadband	Real
ALM_ONS	Alarm one shot storage bits	DINT
ALM_OS	Alarm one shot	Boolean

Table 3-9 User Defined Data Type ANA_ALM

The subroutine ANALOG_ALARM contains the alarm logic. The routine acts upon the variable PV which is defined as type ANA_ALM. The Control Logix processor also has a built in function block to do analog alarming. I have chosen to provide my own logic for this function so that I can maintain the ability to customize it for my own needs and to provide functionality that is not built into the standard block.

Chapter 3 Controllers

In Figure 3-21, the first rung of the ANALOG_ALARM subroutine defines the input parameter that the subroutine acts upon.



Figure 3-21 Analog Alarm Subroutine ANALOG_ALARM

The logic for the high alarm, PV.H is shown in Figure 3-22. The alarm is enabled from the faceplate via the variable PV.H_EN. The variable PV.H_PROG_EN is used by the program to enable the alarm. The alarm utilizes a delay in PV.H_TM_ON, a delay out PV.H_TM_OFF, and a deadband PV.H_DB. The operator must acknowledge the alarm to reset it, if the acknowledge enable has been configured by an engineer or supervisor with the variable PV.H_ACK.EN.

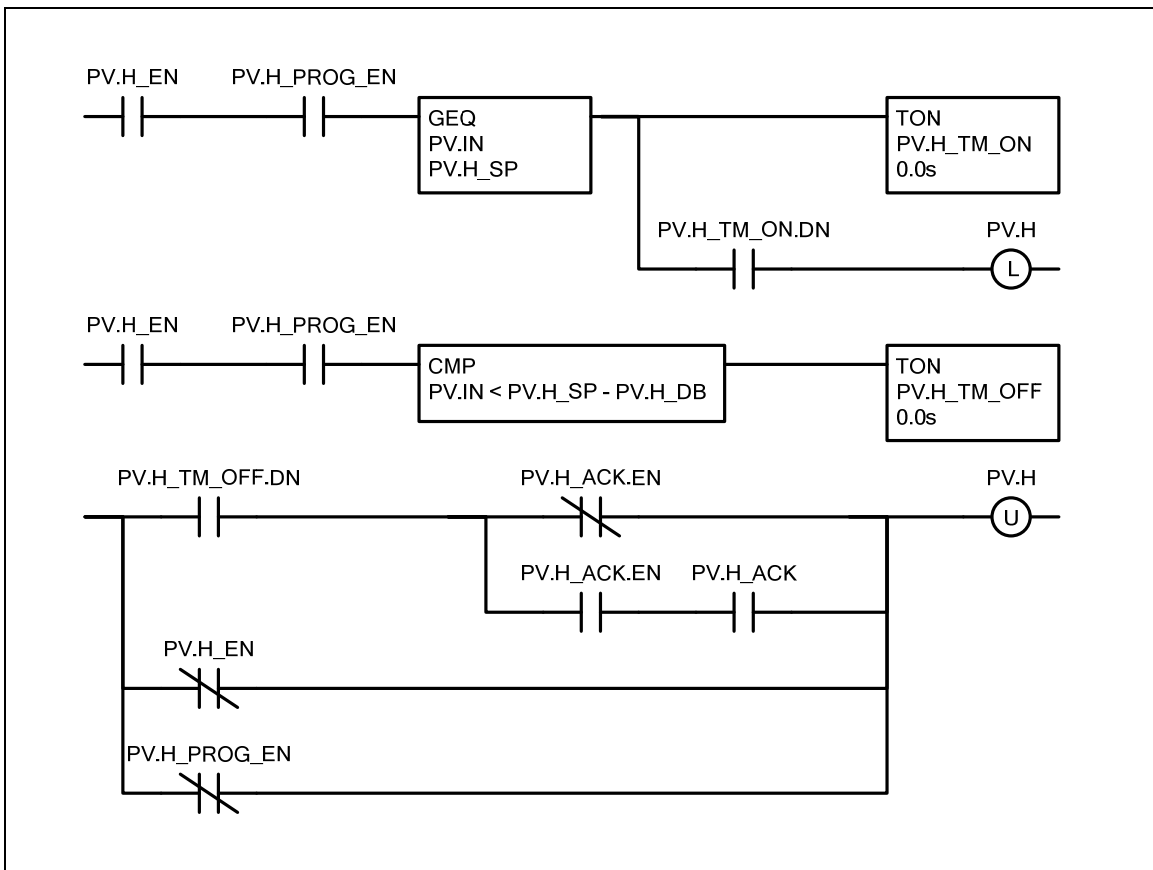


Figure 3-22 Subroutine ANALOG_ALARM High Alarm Logic

The logic for the High High alarm is the same as the High alarm in Figure 3-22. The HH variable is used instead of the H variable, for example PV.HH.

The logic for the Low alarm is shown in Figure 3-23. The Low Low alarm would be programmed in the same way.

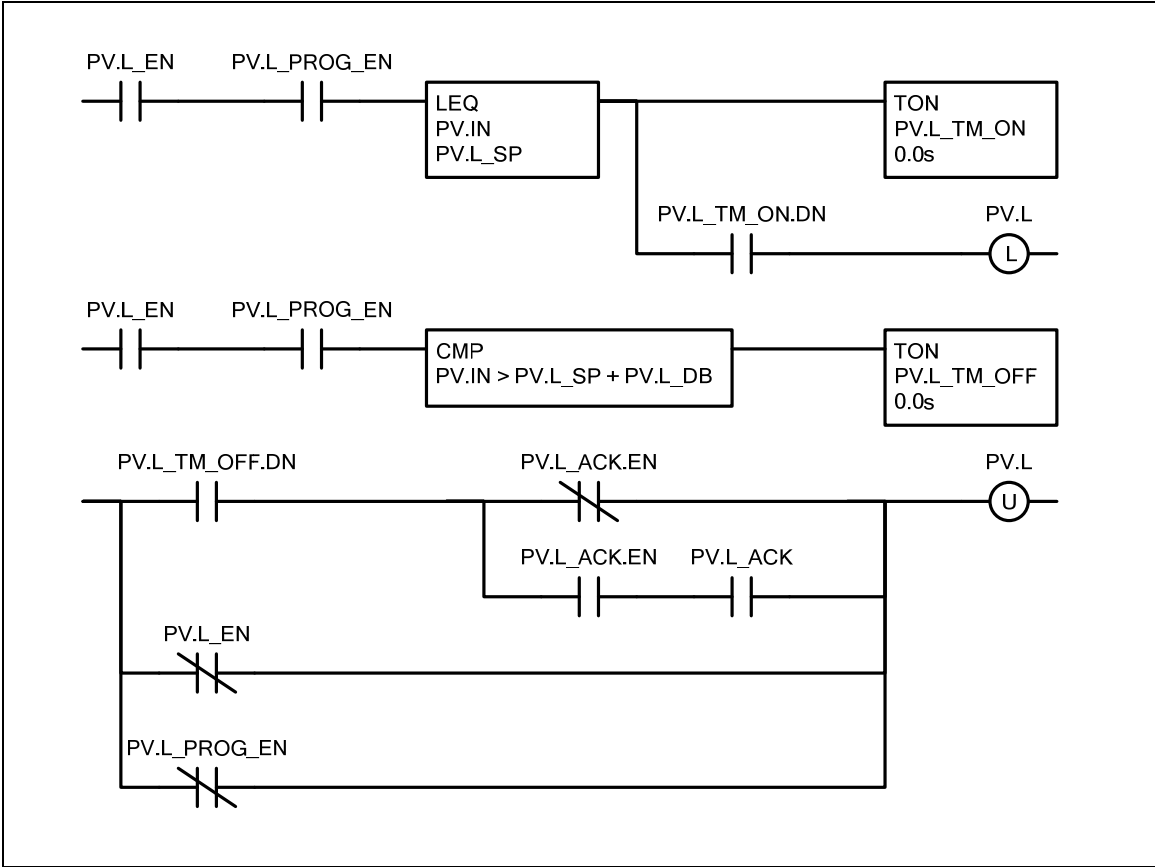


Figure 3-23 Subroutine ANALOG_ALARM Low Alarm Logic

In Figure 3-24 we get the one shot of each alarm. We will use this one shot to reset the horn silence.

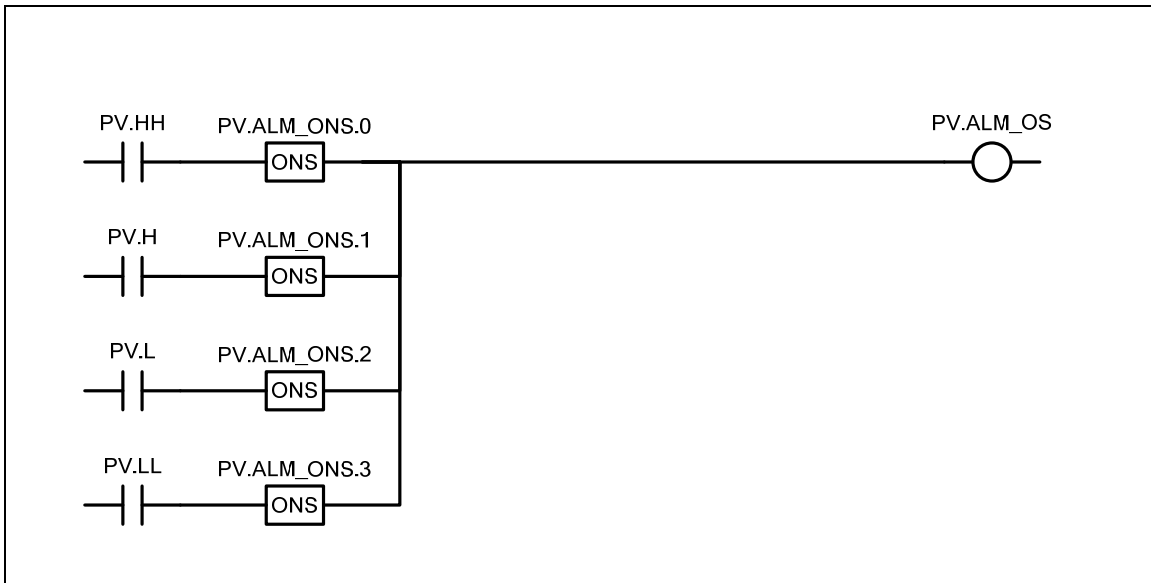


Figure 3-24 Subroutine ANALOG_ALARM summation one shot

The ANALOG_ALARM subroutine then returns the same variable that is acted upon. In Figure 3-25 the last rung of the subroutine contains the RET instruction. The only purpose of this instruction is to copy the contents of the PV variable into the return parameter.

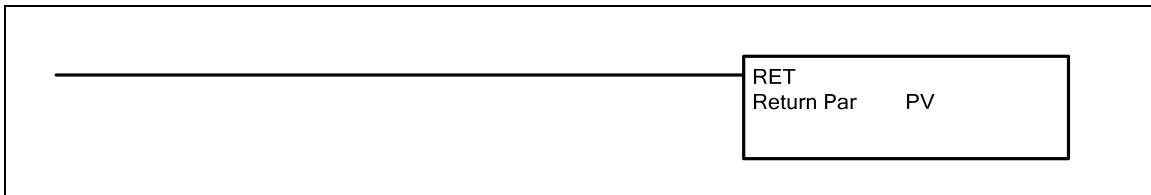


Figure 3-25 Subroutine ANALOG_ALARM Process Variable Return

In Figure 3-26 we begin the first rung for the analog indicator. This logic would be located outside of the subroutine. Each alarm is enabled. If there were any conditions that enable the alarm they would be located here.

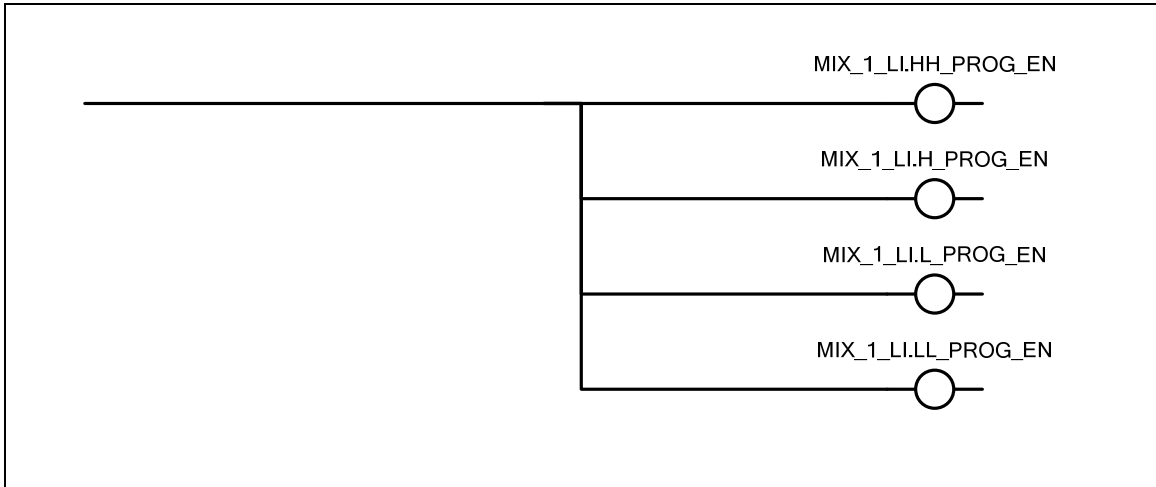


Figure 3-26 Analog Indicator Alarm Enable

The variable MIX_1_LT is defined as a REAL. It contains the scaled value of the level transmitter in engineering units. In this case it would be percent. The transmitter variable can be an alias to an analog input card if the input is scaled in the card configuration. Otherwise it would be an internal variable that has been scaled from the raw input of an analog card. In Figure 3-27 we move the value from the level transmitter MIX_1_LT into the input of the level indicator MIX_1_LI.IN. The contents of the user defined variable MIX_1_LI is then passed to the subroutine variable PV. When the subroutine returns, the PV variable is then copied back to MIX_1_LI via the jump to subroutine instruction JSR.

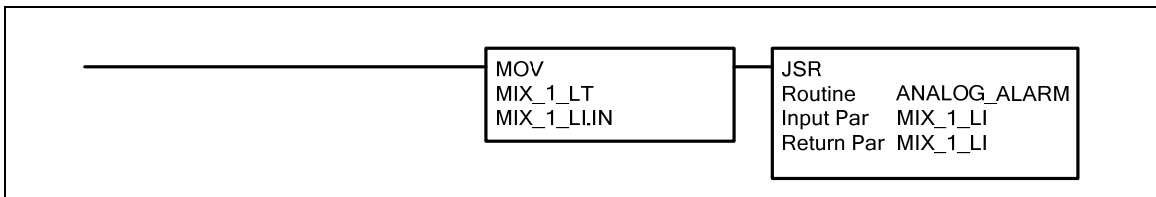


Figure 3-27 Analog Indicator Jump to Subroutine

In Figure 3-28 the alarm one shot that is returned from the subroutine is used to reset the horn silence bit. Typically the horn silence bit is latched on by a horn silence push button. This bit would then be used as a normally closed contact in the horn logic to turn the horn off. Each time an alarm occurs in your system, a one shot from that alarm will reset the horn silence. This will cause the horn to sound until the horn silence push button is pressed.

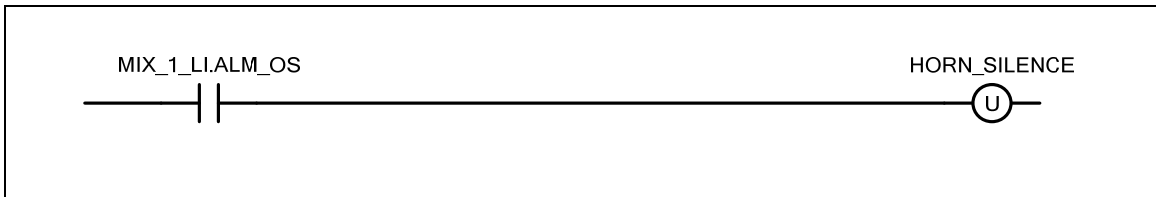


Figure 3-28 Analog Alarm Horn Silence Reset

PID Controller

The faceplate for a PID (Proportional, Integral, Derivative) controller is shown in Figure 3-29. The remote setpoint is located to the left of the local setpoint. The operator can change the mode of the controller to local or remote. In the remote mode, the remote setpoint is copied into the local setpoint. In the local mode the operator can change the local setpoint. The remote setpoint would be written to by some sequential logic outside of the controller logic. The process variable is shown to the right of the setpoint along with a corresponding bar graph. The process variable alarms are indicated with triangles to the right of the process variable bar graph. They function in the same way as was described in the analog indicator alarms in the previous section. The deviation alarm setpoints are indicated by the bars to the right of the process variable bar graph. The height of these bars are determined by the deviation setpoints. These deviation setpoints will also move vertically with the controller local setpoint. The deviation setpoint is divided into four components. The High High, the High, the Low and the Low Low. Each of these components can be enabled. The deviation alarm indicators are visible if they have been enabled. The percent change buttons will affect the setpoint if the controller is in local mode and automatic mode. If the controller is in manual mode then the percent change buttons will affect the output.

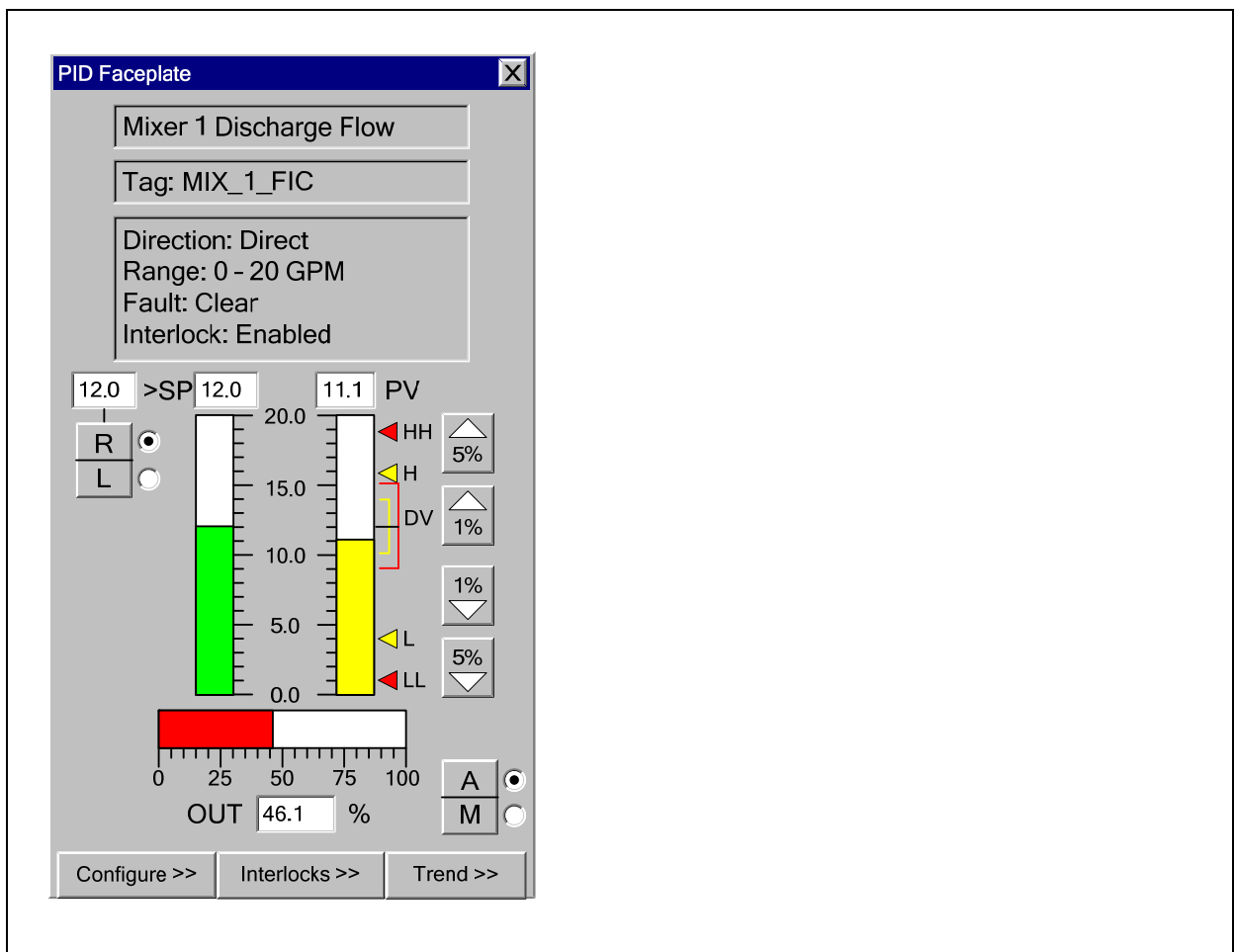


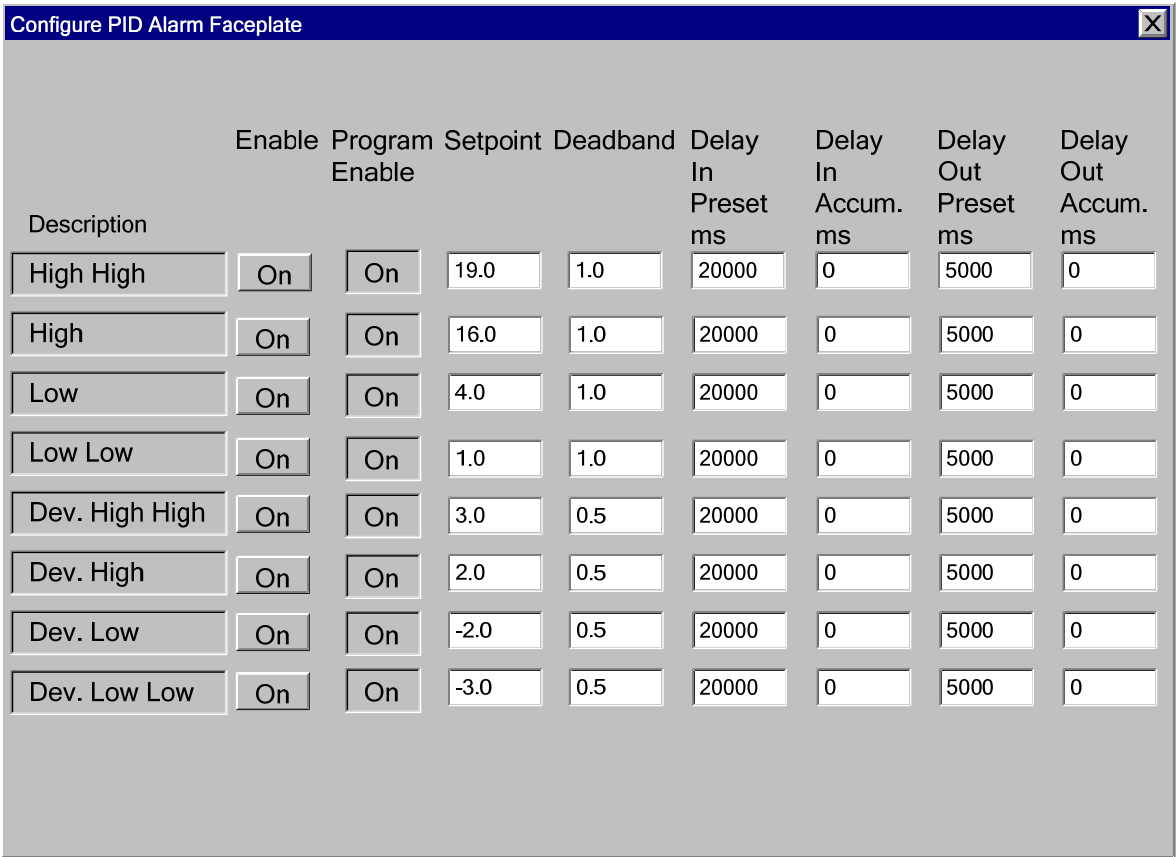
Figure 3-29 PID Controller Faceplate

Ladder logic Programming Techniques By Duane Snider

The range of the controller looks at the minimum and maximum engineering unit variables within the PID.

The target setpoint before the ramp is displayed in the input box above the bar graph. This is setpoint SP1. The actual setpoint is ramped until the target setpoint is reached. This actual setpoint is SP2. The bar graph will indicate the value of SP2.

The faceplate for the alarm configuration of the PID controller is shown in Figure 3-29.



The screenshot shows a window titled "Configure PID Alarm Faceplate" with a close button (X) in the top right corner. The window contains a table with the following columns: Description, Enable, Program Enable, Setpoint, Deadband, Delay In Preset ms, Delay In Accum. ms, Delay Out Preset ms, and Delay Out Accum. ms. The table lists eight alarm types: High High, High, Low, Low Low, Dev. High High, Dev. High, Dev. Low, and Dev. Low Low. Each row has a corresponding "On" button for the "Enable" column. The "Program Enable" column also has "On" buttons for each row. The "Setpoint" column contains numerical values, and the "Deadband" column contains numerical values. The "Delay In Preset ms" column contains the value 20000 for all rows. The "Delay In Accum. ms" column contains the value 0 for all rows. The "Delay Out Preset ms" column contains the value 5000 for all rows. The "Delay Out Accum. ms" column contains the value 0 for all rows.

Description	Enable	Program Enable	Setpoint	Deadband	Delay In Preset ms	Delay In Accum. ms	Delay Out Preset ms	Delay Out Accum. ms
High High	<input type="button" value="On"/>	<input type="button" value="On"/>	19.0	1.0	20000	0	5000	0
High	<input type="button" value="On"/>	<input type="button" value="On"/>	16.0	1.0	20000	0	5000	0
Low	<input type="button" value="On"/>	<input type="button" value="On"/>	4.0	1.0	20000	0	5000	0
Low Low	<input type="button" value="On"/>	<input type="button" value="On"/>	1.0	1.0	20000	0	5000	0
Dev. High High	<input type="button" value="On"/>	<input type="button" value="On"/>	3.0	0.5	20000	0	5000	0
Dev. High	<input type="button" value="On"/>	<input type="button" value="On"/>	2.0	0.5	20000	0	5000	0
Dev. Low	<input type="button" value="On"/>	<input type="button" value="On"/>	-2.0	0.5	20000	0	5000	0
Dev. Low Low	<input type="button" value="On"/>	<input type="button" value="On"/>	-3.0	0.5	20000	0	5000	0

Figure 3-30 PID Controller Process Variable Alarm Faceplate

Figure 3-31 shows the trended variables for the PID controller. The Proportional, Integral, and derivative PID configuration variables are also set on this faceplate in order to tune the loop.

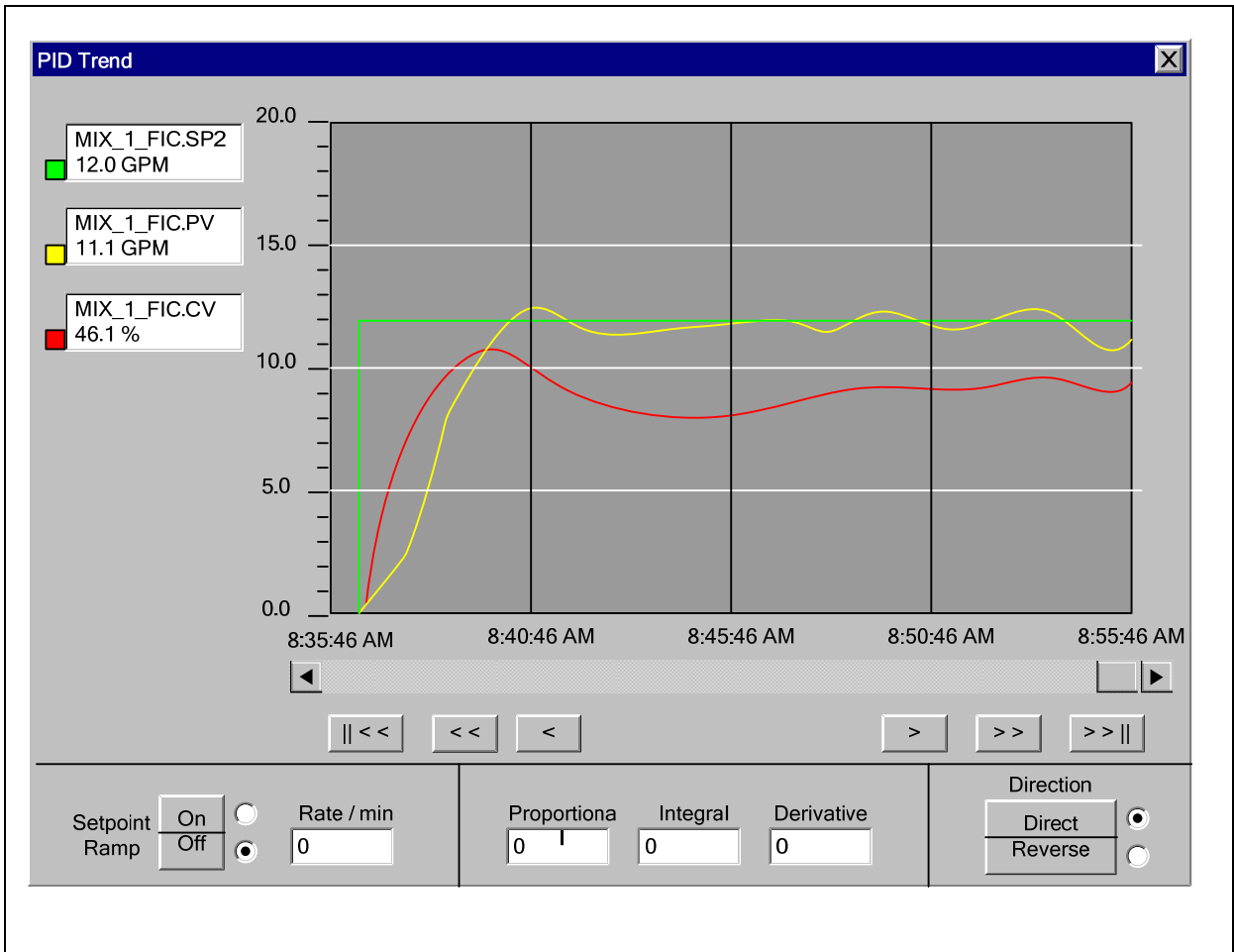


Figure 3-31 PID Controller Trending Faceplate

The user defined data type for the PID controller is defined in Table 3-10. Note that the process variable alarm, PVA, and the deviation alarm DVA, are user defined variables of the type ANA_ALM that was defined in the analog indicator controller that we discussed in the previous section.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	HMI description	DESC_STR
TAG	HMI tag name	TAG_STR
UNITS	HMI units	UNITS_STR
SSA	HMI Auto select	Boolean
SSR	HMI remote	Boolean
SSE	HMI PID enable	Boolean
SS_RAMP	HMI setpoint ramp select	Boolean
SS_RATIO	HMI setpoint ramp select	Boolean

AUTO	Auto bit set by remote sequence	Boolean
PID	PID variable	PID
PVA	Process variable alarm	ANA_ALM
DVA	Deviation alarm	ANA_ALM
DV	Deviation	Real
SP_REM	Remote setpoint	Real
SP1	Setpoint target before ramp	Real
SP2	Setpoint actual during ramp	Real
SP_RATE	Ramp setpoint rate in units/min	Real
PV	Process variable	Real
CV	Control Variable	Real

Table 3-10 User Defined Data Type PID

In Figure 3-32 we begin the logic for the PID controller. This rung enables the process variable alarms when the feed pump is started. The timer prevents low flow alarms from occurring when the pump is not running or is not up to speed. The low flow alarm would in turn be used to interlock the pump off if there is no flow after the pump has been started. This will prevent the pump from being damaged by running for an extended time with no fluid circulating through it. MIX_1_M2_HCX is the auxiliary input of the pump holding contactor.

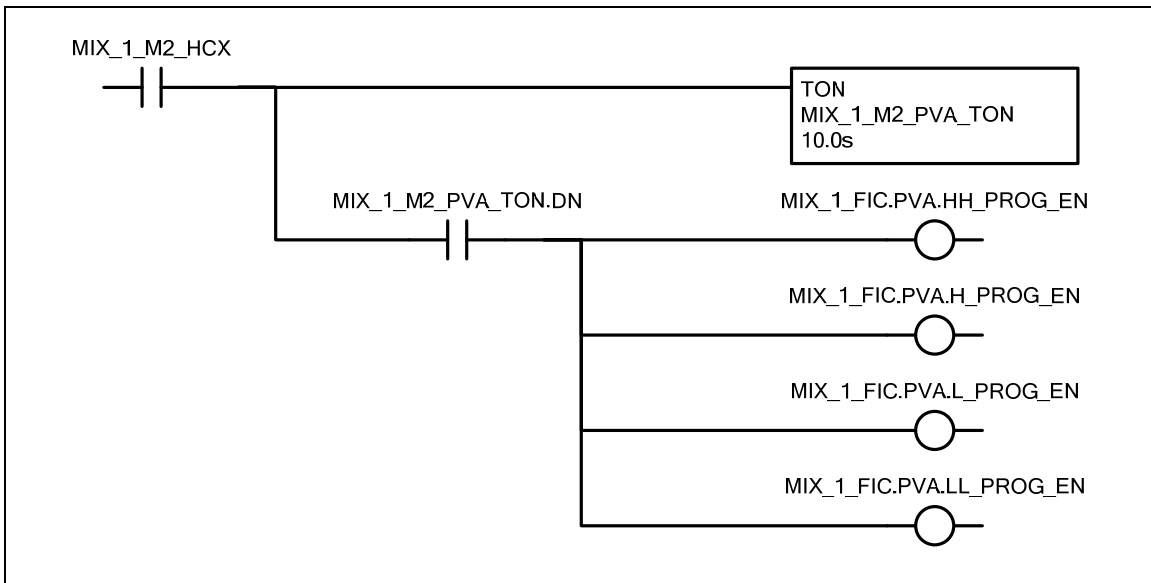


Figure 3-32 PID Controller Process Variable Alarm Enable

In Figure 3-33 the deviation alarms are also enabled when the pump is started. The deviation alarm requires a longer start up time than the process variable alarms to allow the PID controller to stabilize.

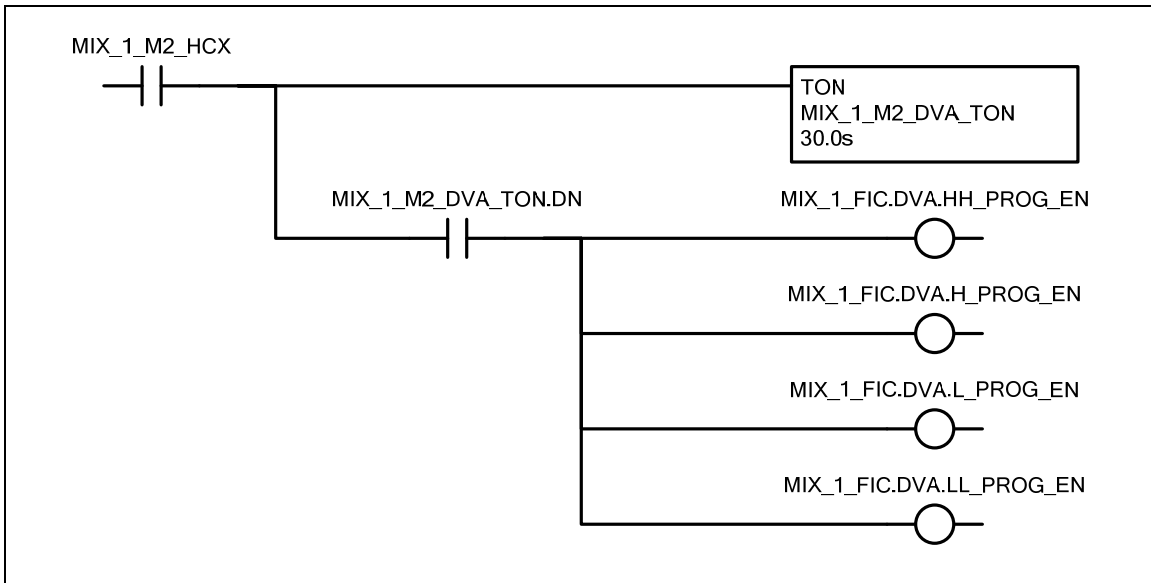


Figure 3-33 PID Controller Deviation Alarm Enable

In Figure 3-34 the scaled value of the flow transmitter is moved into the process variable of the PID controller. This is done primarily for convenience. During program development, if you copy the code from one controller to the next, you only have to change this rung to change the PV.

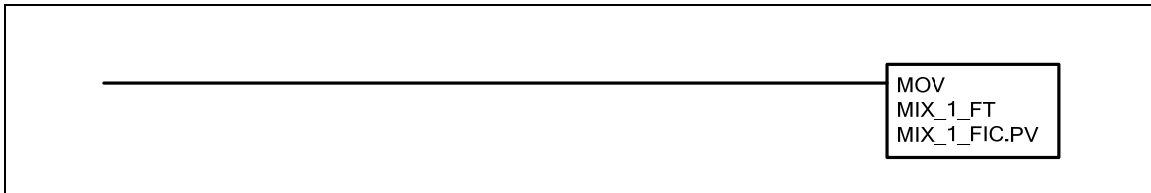


Figure 3-34 PID Controller Setting the UDT Process Variable

In Figure 3-35, after the process variable has been updated, it is moved into the input variable of the process variable alarm. The subroutine ANALOG_ALARM is then executed. We described this subroutine in the analog indicator controller in the previous section. The deviation is calculated and then the routine is called again using the deviation as the input variable.

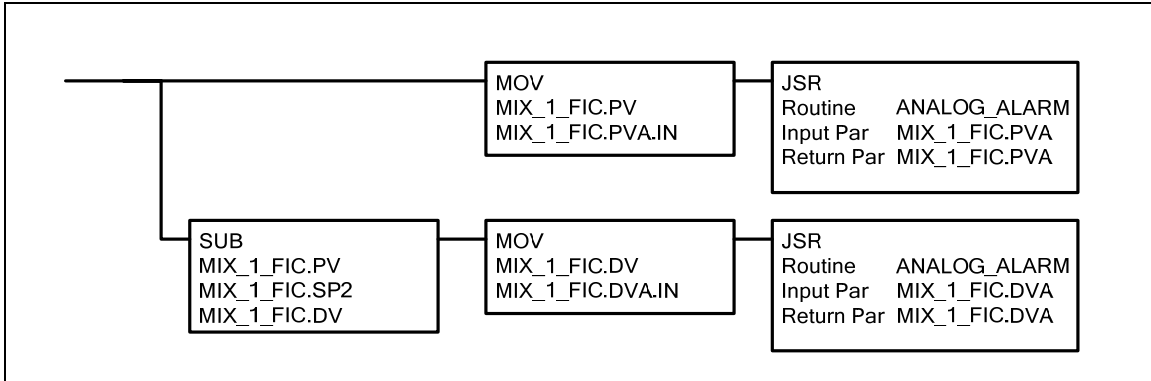


Figure 3-35 PID Controller Calling the ANALOG_ALARM Routine

In Figure 3-36 the horn silence is reset if an alarm occurs.

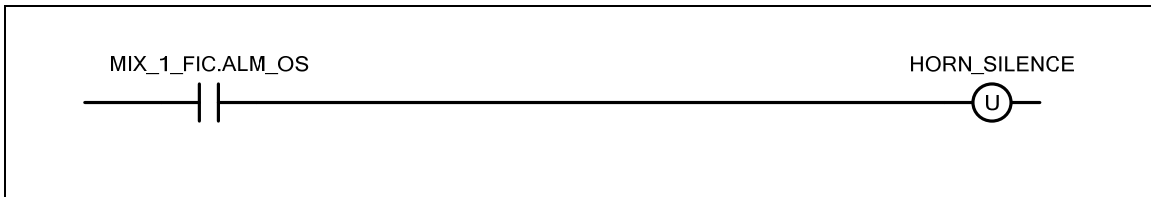


Figure 3-36 PID Controller Horn Silence Reset

In Figure 3-37 the remote setpoint is moved into setpoint 1 if the controller is in the remote mode. The remote setpoint could be written to by some other remote logic or some other controller. SP1 is set by the operator from the HMI faceplate, if the controller is not in the remote mode.

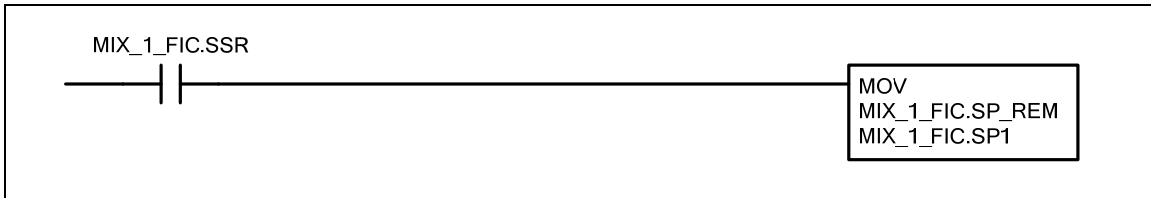


Figure 3-37 PID Controller Using the Remote Setpoint

The scan time is used to calculate the ramped setpoint. In Figure 3-38 the get system value instruction, GSV, is used to get the current value of the clock. The elapsed time is then subtracted from the previous clock value and the result is the total elapsed time for the current program scan. The correct scan time is calculated when the wall clock wraps around from the maximum double integer value to zero. It makes sense to locate this logic in the main routine of the program so that other controllers can use this SCAN_TIME variable.

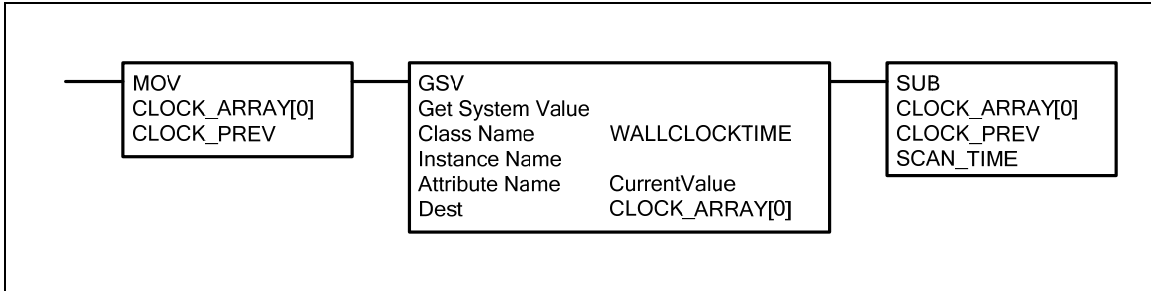


Figure 3-38 Getting the System Variable WALLCLOCKTIME to calculate SCAN_TIME

The variables are defined in Table 3-11.

Tag	Description	Type
CLOCK_ARRAY[0]	Accepts the current value of the WALLCLOCKTIME	DINT[2]
CLOCK_PREV	The previous scans CLOCK value	DINT
SCAN_TIME	The elapsed program scan time in microseconds	DINT

Table 3-11 Clock Variable Definition

Figure 3-39 shows the logic to ramp the setpoint. If ramping is selected and a change occurs in SP1 either by the operator or the remote program, SP2 is adjusted each program scan by the amount defined by the setpoint rate, SP_RATE. The rate is in units per minute. In this case it is gallons per minute. The expression in the compute statement converts the rate to units per microseconds. That value is then multiplied by the scan time which gives the number of units that the setpoint is adjusted for each program scan. Refer to Appendix B for a detailed explanation of unit conversion calculations.

If ramping is not selected then SP1 is moved into SP2.

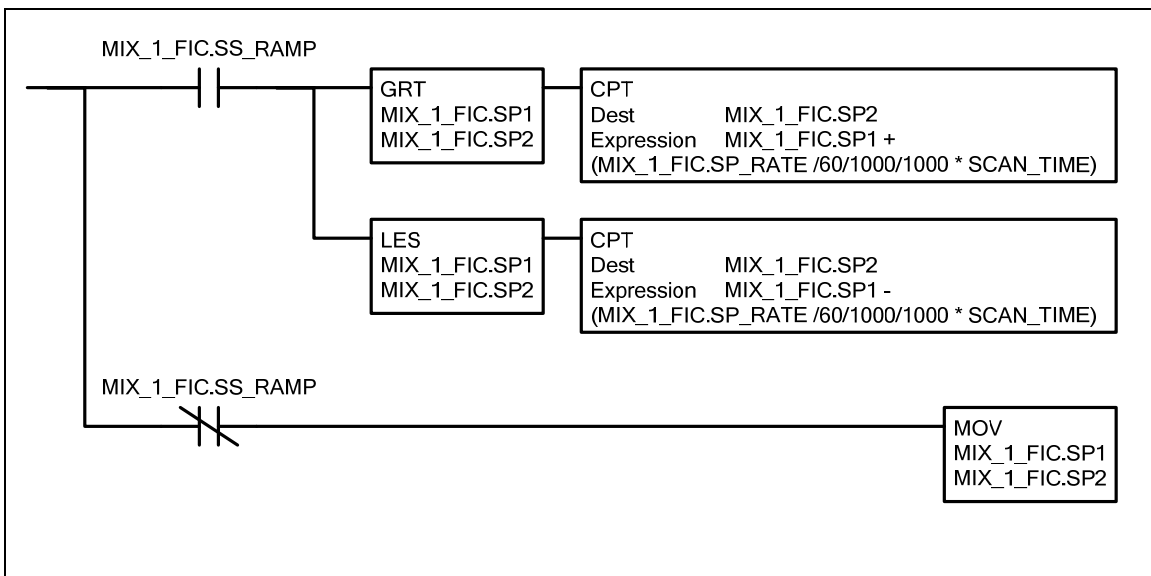


Figure 3-39 PID Controller Process Calculating the Ramped Setpoint

In Figure 3-40 the PID instruction is put in manual mode with the SWM element of the predefined data type PID. The PID data type has many elements that determine how the PID instruction behaves. The ramped setpoint, SP2, is moved into the PID element SP. The PID instruction is enabled with the enable selector switch. I did not include this selector switch on the faceplate. It could be eliminated altogether but I like to have the ability to disable the PID instruction even if it only through the programming software.

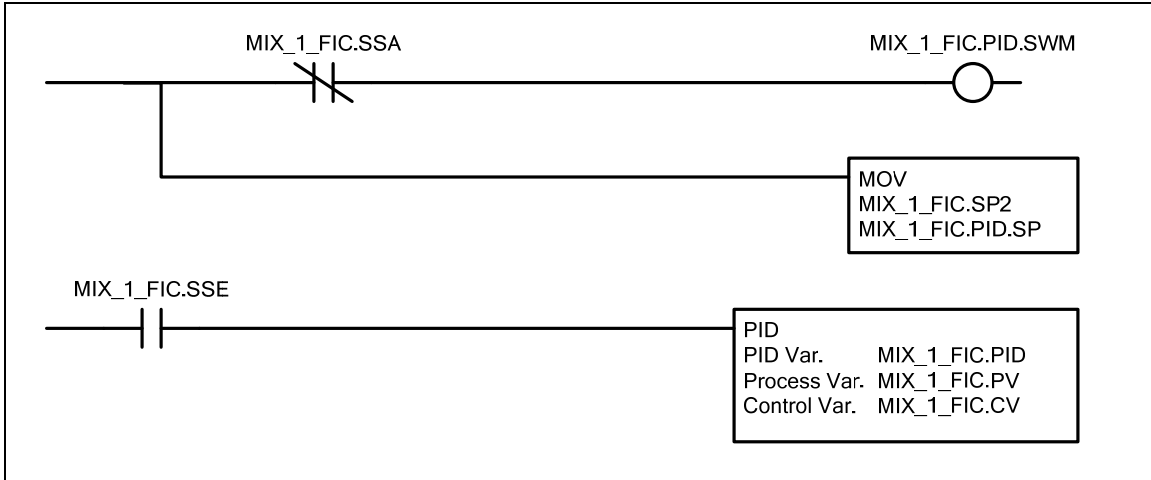


Figure 3-40 PID Controller Instruction and Manual Mode

The PID instruction adjusts the control variable until the process variable is equal to the setpoint.

In Figure 3-41 the control variable is moved into the remote setpoint of the variable frequency drive controller. The faceplate for the VFD controller is shown in Figure 3-8. If this controller is in remote mode then the motor will change speed according to the PID control variable MIX_1_FIC.CV.

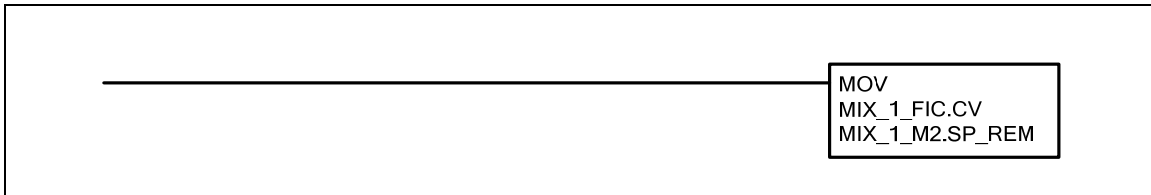


Figure 3-41 PID Controller Setting the Control Variable to the VFD Remote Setpoint

Ratio Controller

The ratio controller is used to ratio one feed ingredient to another. In this example the setpoint of one MIX_1_FIC3 controller is ratioed to the setpoint of MIX_1_FIC2 controller. The faceplate for the ratio controller is shown in Figure 3-42.

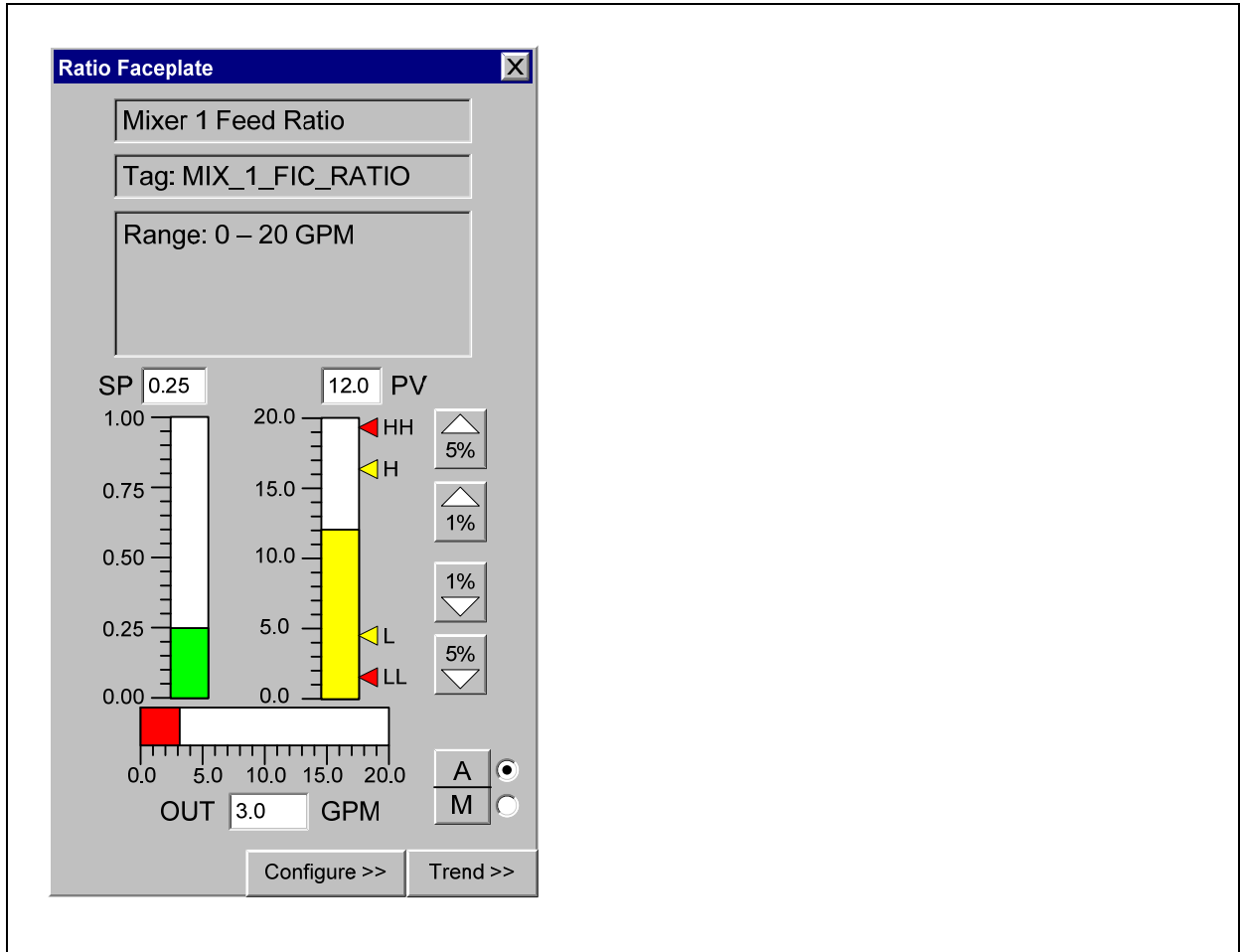


Figure 3-42 Ratio Controller Faceplate

The configure faceplate would be the same as the analog indicator configuration faceplate. This faceplate allows you to configure the process variable alarms. The trend faceplate would show the process variable and the output.

Ladder logic Programming Techniques By Duane Snider

The user defined data type for the ratio controller is shown in Table 3-12. You would name the data type `RATIO_CTRL`. Then create a variable called `MIX_1_FIC_RATIO` using this data type.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
DESC	HMI description	DESC_STR
TAG	HMI tag name	TAG_STR
UNITS	HMI units	UNITS_STR
EGUH	High engineering units	Real
EGUL	Low engineering units	Real
SSA	HMI Auto select	Boolean
SSE	HMI enable	Boolean
PVA	Process variable alarm	ANA_ALM
SP	Setpoint target before ramp	Real
PV	Process variable	Real
CV	Control Variable	Real

Table 3-12 User Defined Data Type `RATIO_CTRL`

The logic for the ratio controller is straight forward. We multiply the process variable by the ratio setpoint and put the result into the control variable. I did not include the enable selector switch on the faceplate but you could if you want.

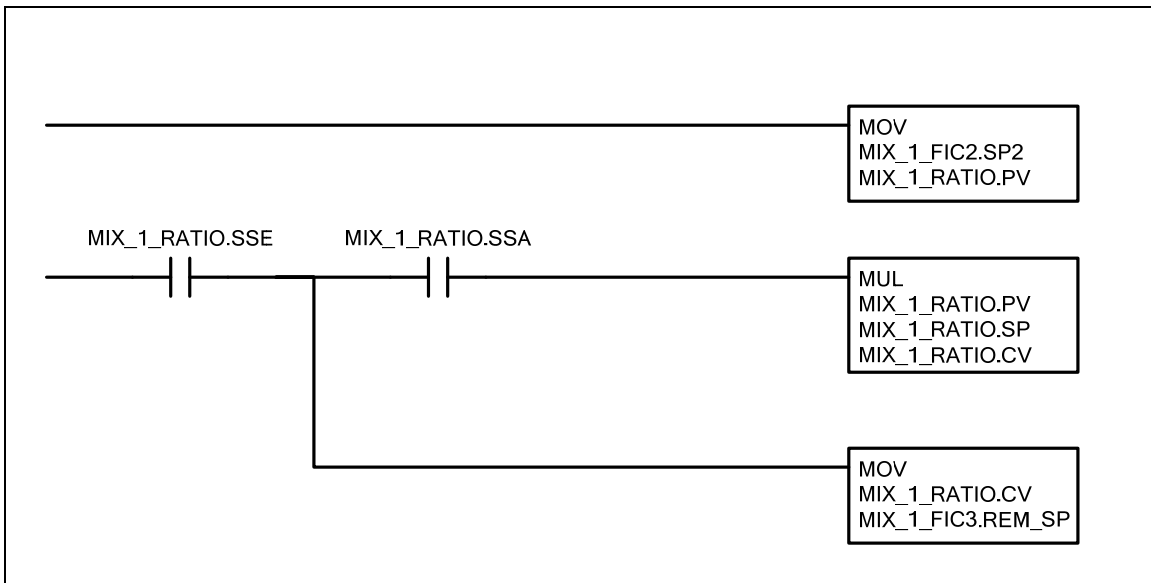


Figure 3-43 Ratio Controller Calculating the Ratio and Setting the FIC Remote Setpoint

Chapter 3 Controllers

In Figure 3-44 the ratio alarms are setup using the ANALOG_ALARM routine. Remember that the ratio process variable is the setpoint of the first FIC controller. The only time an alarm will occur is when an operator enters a setpoint that is too high. If we were using the actual process variable of the FIC to ratio from, then a high flow rate would cause a ratio alarm.

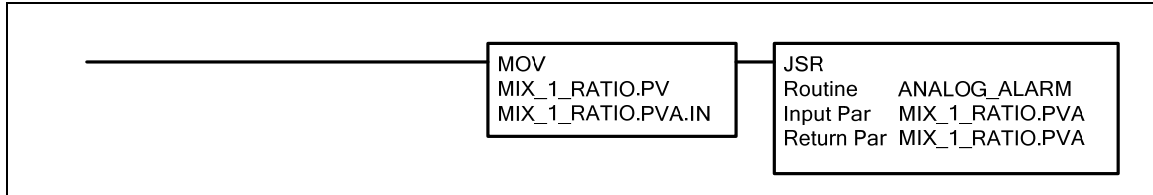


Figure 3-44 Ratio Controller Call to the ANALOG_ALARM Subroutine

Chapter 4 State Logic

State logic can be used to control a sequential process. In state logic an integer is used to determine which step or state the process is in. one advantage to using an integer is that the number can be changed manually to any step. There are however some disadvantages to using a single integer to control the step.

Advantages

- The step can be easily changed manually.
Since the step is an integer it can be changed with an HMI.
- Steps can be repeated without resetting the sequence.
The sequence can move from one state to any other state just by checking the conditions and then moving the appropriate value into the integer step number.
- The current step of the sequence can be easily seen by examining the integer value.

Disadvantages

- You can not do parallel branching with a single integer.

An integer can only have one value at any one time. If you need to do parallel branching, then a new integer variable would be needed for each branch. This would make setting the step manually from an HMI more difficult since the operator would have to deal with multiple variables. You could limit the operator to changing the step to a number above or below the branch. That would mean you have to write some code either in the HMI or PLC to check the value that the operator has entered before the step can be changed in the PLC. Create a new variable for the operator to enter the data. Check that variable against a valid list of steps. If the step number is valid then move it to the PLC step. If it is not then let the operator know that an invalid step number has been entered.

- You don't have an output to tie a description of each step to.

This may seem trivial but it is not. Without this description you tend to go back and forth between the step logic and the output logic verifying that you have programmed the correct step numbers into the outputs.

I would tend to use state logic for a process control verses machine control. I find that I often want to repeat steps in a process. For example, if I am feeding into a scale I may need to do a bump cycle several times to achieve tolerance.

State Logic – First Form

Figure 4-1 shows how the steps are implemented using state logic. Note how the step increments by 10 for each step. This allows you to insert steps after the fact without having to re-write your whole program. The second and third form of state logic that are described later in this chapter are just variations of the first form described here. In the second form a bit is used for each output affected by the sequence. That bit is set or reset during the execution of the sequence. The third form uses an output for each step to allow us somewhere to tie a description of the step to.

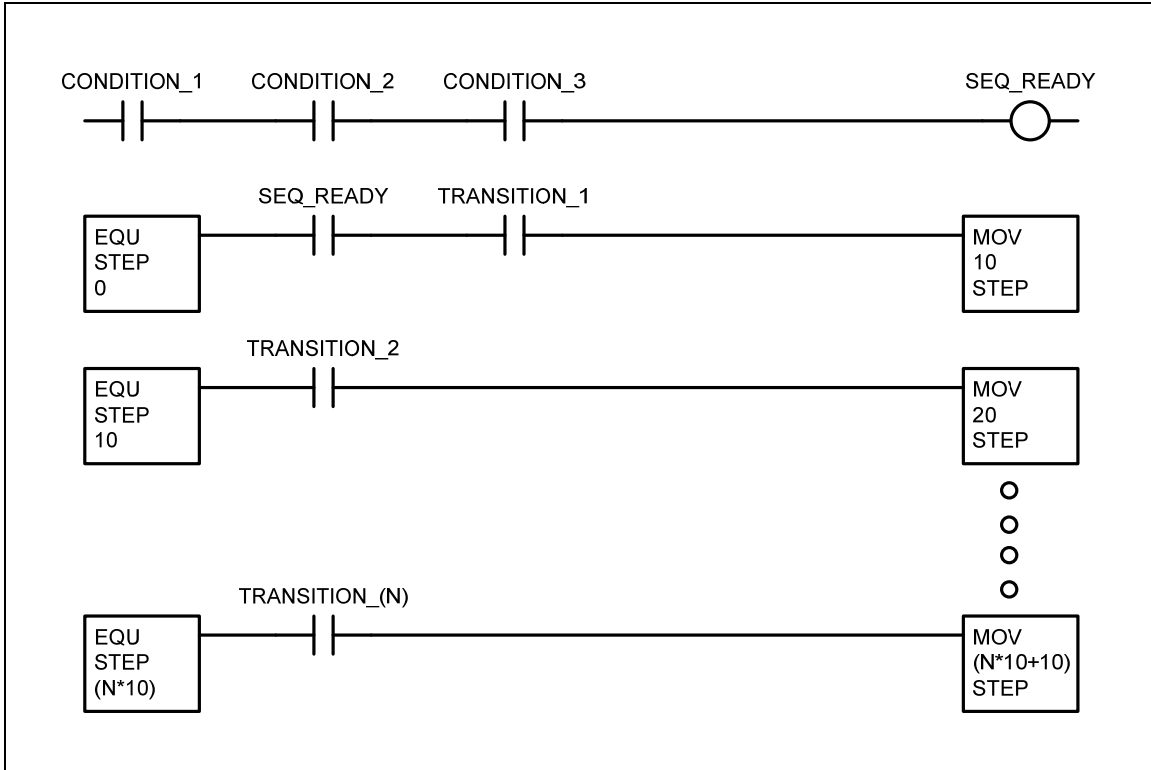


Figure 4-1 State Logic First Form Example Step

Figure 4-2 shows how the end of cycle is programmed using state logic.

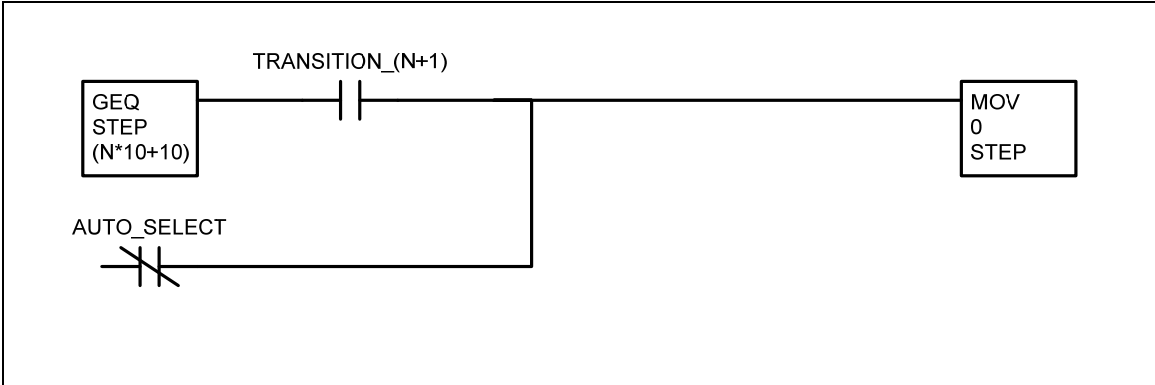


Figure 4-2 State Logic First Form End Of Cycle

Figure 4-3 shows how the outputs are programmed using the first form state logic. I have broken the output logic up into 2 rungs for clarity. The OUTPUT_1_AUTO bit can be displayed on an HMI to let the operator know what state the sequence is trying to move the output to when the system is in auto. This is especially important if the manual mode does not reset the sequence. In this case the operator can move an output to the off position in manual. When the sequence is returned to the auto mode the output can change state.

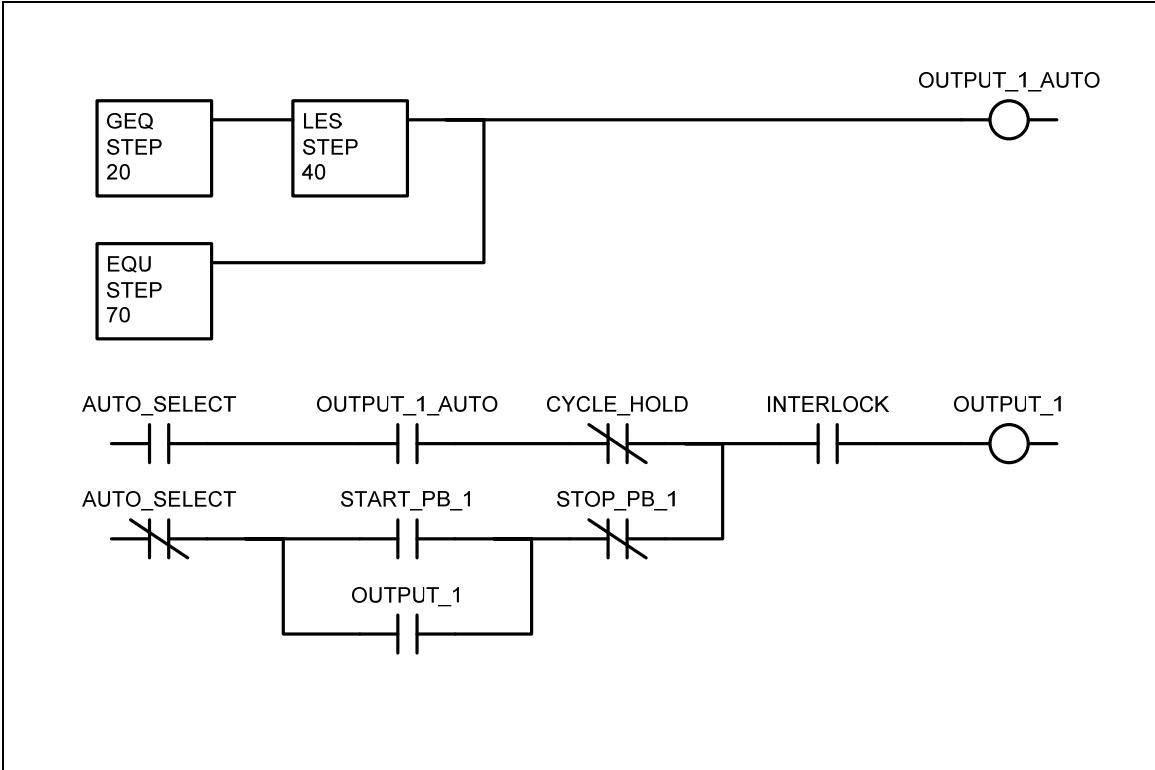


Figure 4-3 State Logic First Form Output Logic

State Logic – Second Form

The second state logic form is a variation of the first. An auto bit is assigned to each output that is affected by the sequence. This bit is set or reset for each step that affects the output. I have also seen this bit called a cascade bit. Since we are latching this auto bit, it will not reset at the end of the sequence. We will have to write code that resets the auto bits when the step is equal to zero. One disadvantage to this second form is that there is no indication of what step(s) are affecting the output when we view the output logic. There is a bit more house keeping to deal with since you have to make sure that the auto bits are reset at the end of the sequence. Another disadvantage is that unless you include the state of all of the auto bits in each step you will be unable to change the step manually and expect all of the auto bits to change to their correct state for that step. One reason I like this form though, is that you can see what is supposed to happen at each step when you browse through the sequence. Figure 4-4 shows how the second form state logic is implemented.

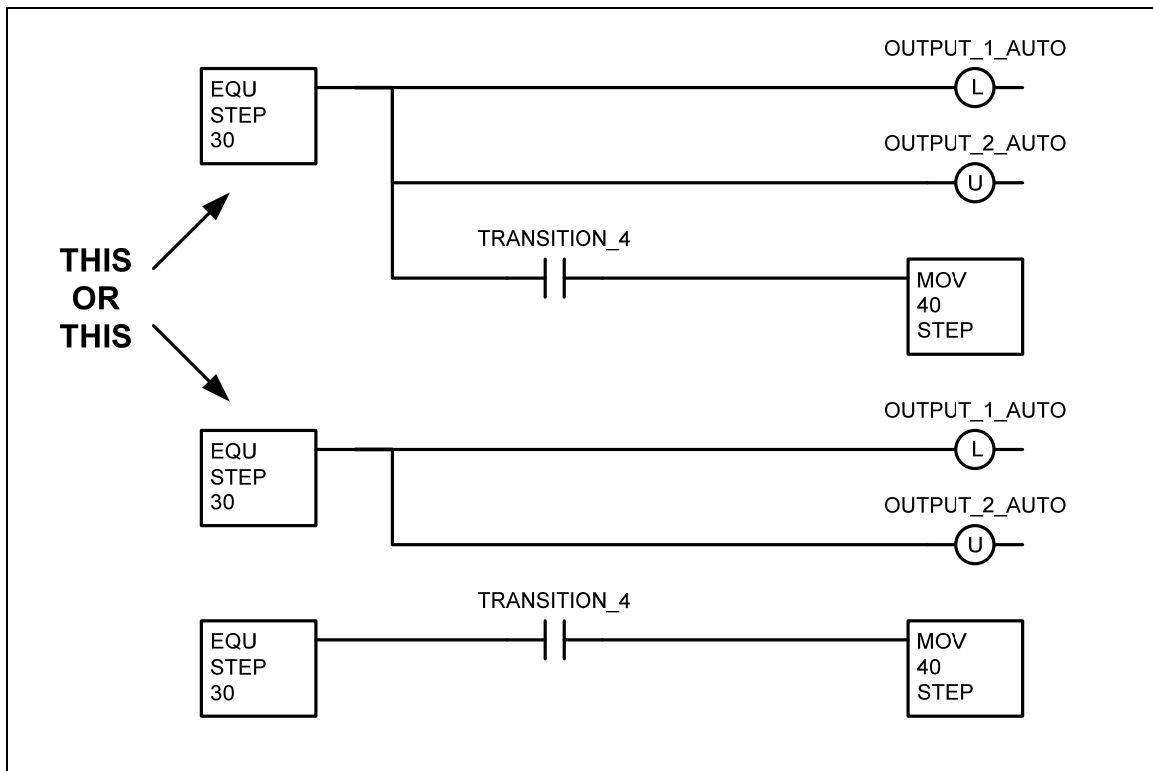


Figure 4-4 State Logic Second Form Example Step

Figure 4-5 shows how the outputs are programmed using the second form state logic. Note that we can not see which step is affecting the output when examining the output logic.

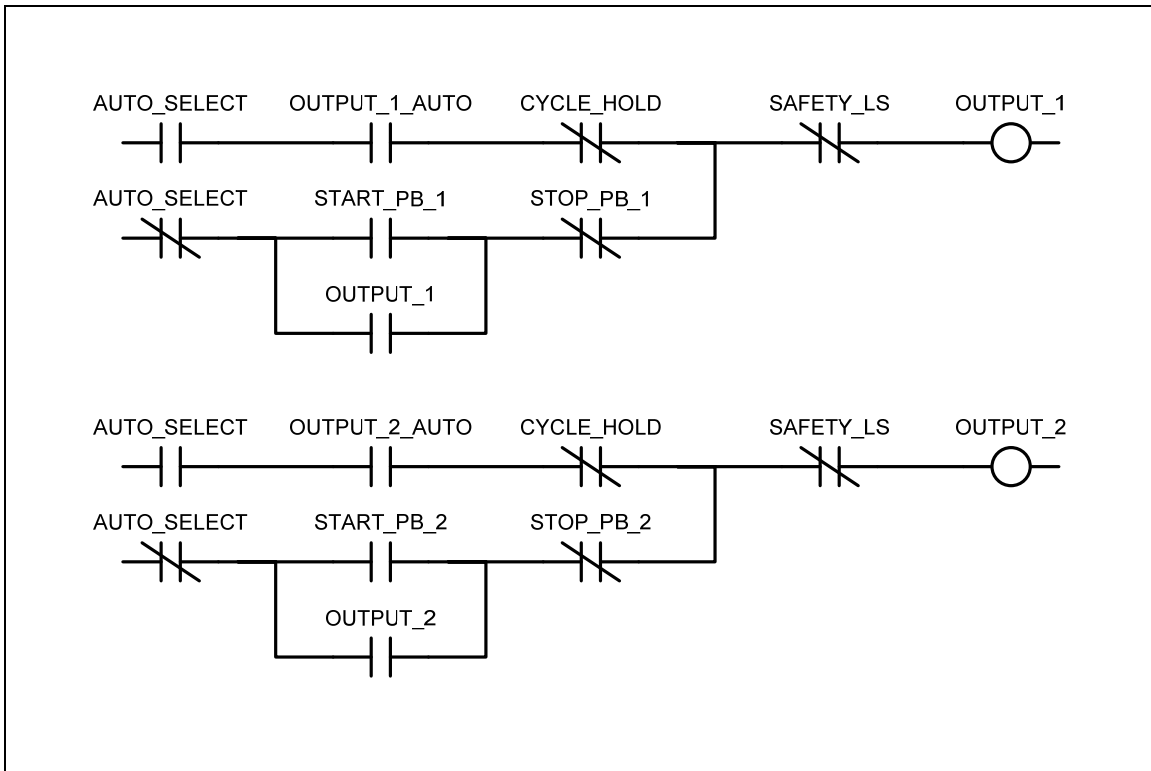


Figure 4-5 State Logic Second Form Output Logic

Gain in Weight Feeder

The gain in weight feeder example consists of three dry ingredient silos feeding into a mixer. Load cell sensors are connected to a weight transmitter. This transmitter could be an analog 4-20Ma signal or it could be a smart transmitter that would make the weight available to the PLC via some communication network. Each silo has an air actuated slide gate that discharges the material into the mixer.

To weigh material into the mixer, the feed valve for the silo is opened. A cut-off weight is calculated based upon the setpoint and the in-flight. When the cut-off is reached, the valve will close and the settling timer is started. When the timer is done, the weight is checked to see if it is within tolerance. If the weight is within tolerance the sequence will end. If the weight is under tolerance then the valve will bump open for a preset time in order to achieve tolerance. The bump cycle will repeat for up to 5 times. If the tolerance is not achieved then an under tolerance alarm is generated after the 5th bump. The operator is prompted to reset the bump counter and continue the bump or, the alarm can be acknowledged and the process will continue. If the weight goes above the tolerance value, then an over tolerance alarm is generated and the operator must acknowledge the alarm to continue the process.

In Figure 4-6 the gain in weight feeder is illustrated.

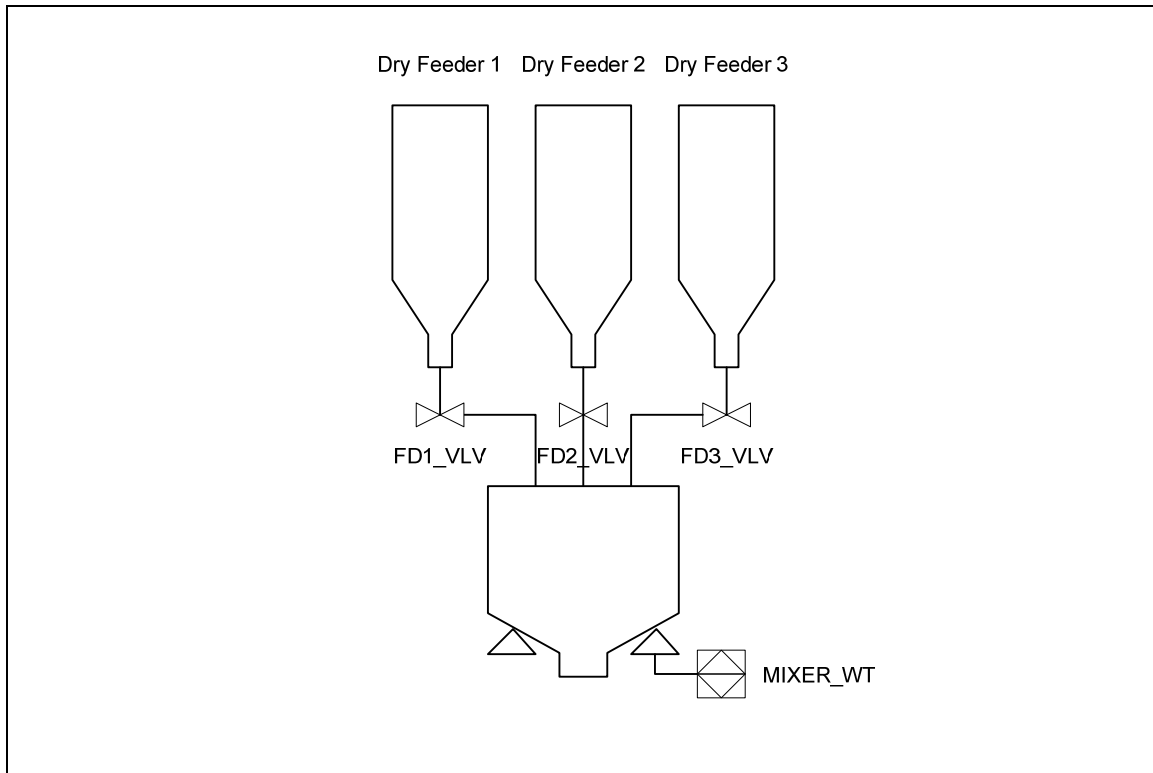


Figure 4-6 Gain in Weight Feeder Process Diagram

The sequence diagram can be used to visualize the steps in a sequence. It is similar to a flowchart and can be considered to be the same as a sequential function chart (SFC). Although SFC programming is a program language in some PLC and DCS systems, it is only used as a visual aid in this book to show the progression of the process sequence being described. The SFC uses steps and transitions to perform operations in the process. A step represents a state of the process in which actions are performed. These actions will be in the form of outputs being manipulated to control the process. The transition is a true or false condition of the process that must occur before the next step in the process is executed.

In Figure 4-7 the SFC is used to illustrate the flow of the feed sequence.

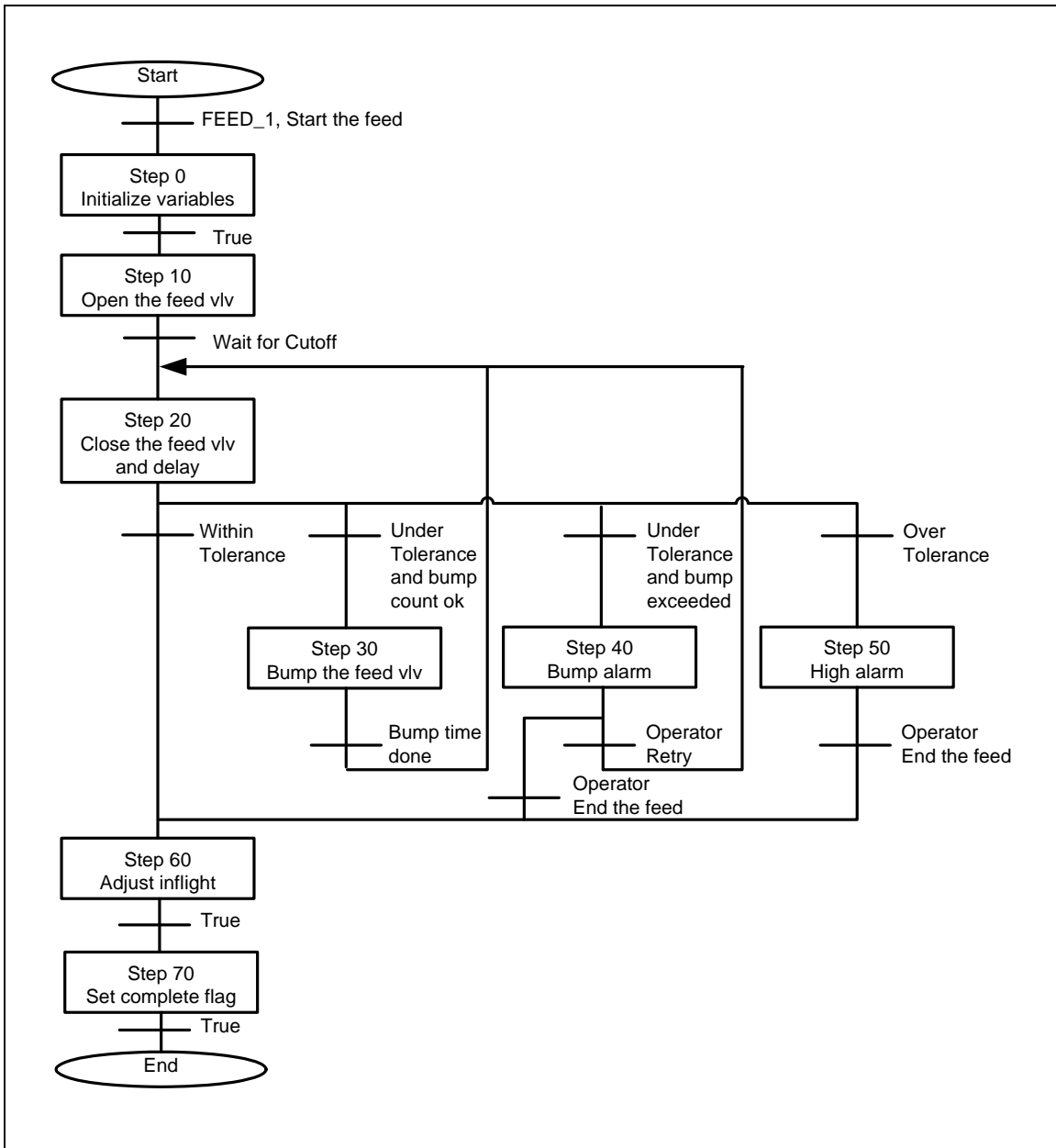


Figure 4-7 Gain in Weight Feeder SFC

In Table 4-1 we describe the variables that we will use in the feed logic.

<i>Tag</i>	<i>Description</i>	<i>Type</i>
MIXER_WT	The value of the mixer weight	Real
FD1_STEP	The feed sequence step	Dint
FD1_START_WT	The weight of the mixer when the feed is started	Real
FD1_SETPOINT	The required amount to feed into the mixer	Real
FD1_TARGET	The target weight of the mixer at the end of the feed	Real
FD1_INFLIGHT	The estimated or calculated amount that will drop into the mixer after the valve is closed	Real
FD1_CUTOFF	The weight at which the valve is closed in order to reach the target weight.	Real
FD1_TOLERANCE	The amount that the feed can vary and still be acceptable	Real
FD1_LOW_POINT	The lowest acceptable feed amount	Real
FD1_HIGH_POINT	The highest acceptable feed amount.	Real
FD1_BUMP_CNT	The number of times the feed valve has been bumped in order to reach the target	Dint
FD1_FLT_INC	The amount the in-flight will be adjusted for the next feed if the original cutoff did not result in a feed that is within tolerance.	Real

Table 4-1 Gain in Weight Feeder Tags

In Figure 4-8 we initiate the feed sequence. FEED_1 is the sequence mode. In Chapter 5 Batch Control we will discuss how the feed mode can be set when the sequence is initiated by a batch controller. The mode will also allow the sequence to be stopped and still maintain the step that the sequence is on. This will allow the sequence to be restarted.

In this first step, the mixer weight is saved into the variable FD1_START_WT. the bump count is reset FD1_BUMP_CNT, and the bump latch is reset. The bump latch is used at the end of the sequence as a flag to adjust the in-flight.

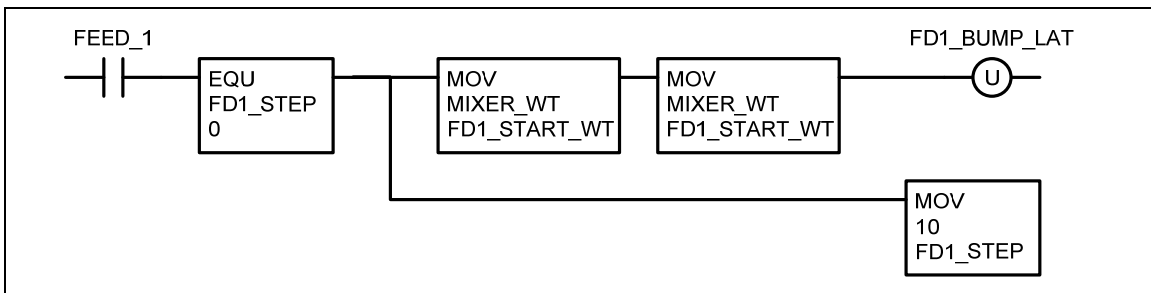


Figure 4-8 Gain in Weight Feeder Initialize

In Figure 4-9 we calculate the target weight by adding the start weight with the setpoint. The inflight is then subtracted from the target to give us the cutoff weight. This calculation will remain active during the entire sequence. This will allow the operator to change the setpoint if necessary during the feed.

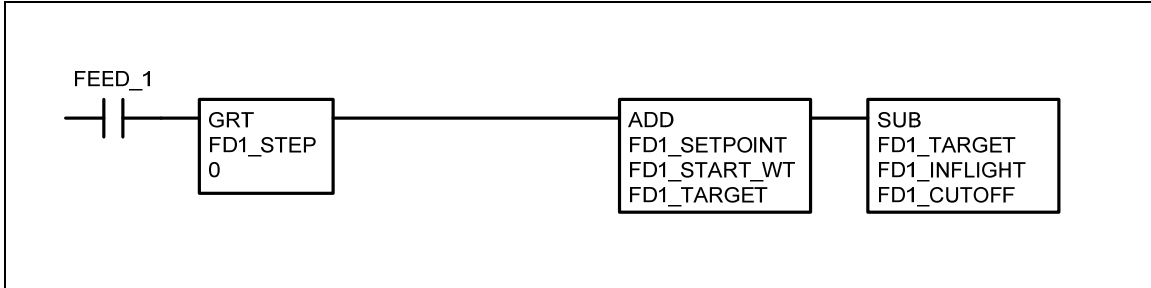


Figure 4-9 Gain in Weight Feeder Calculate the Cut-Off

In Figure 4-10 the feed valve is opened to begin the feed. The mixer weight is monitored until it reaches the cutoff. At the feed cutoff, the step is incremented. In the next step, the valve will be closed.

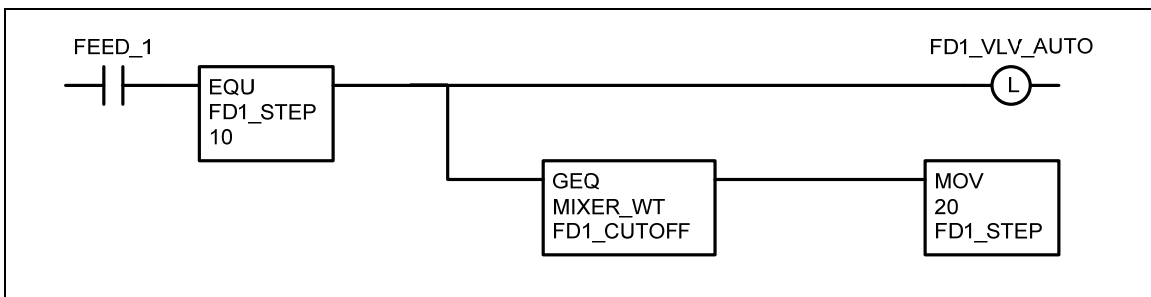


Figure 4-10 Gain in Weight Feeder Opening the Feed Valve

In Figure 4-11 the low point and the high point are calculated. These values are then compared to the mixer weight to see if the weight is within tolerance. The bump count is also checked. Again these calculations and comparisons will remain active during the entire feed sequence.

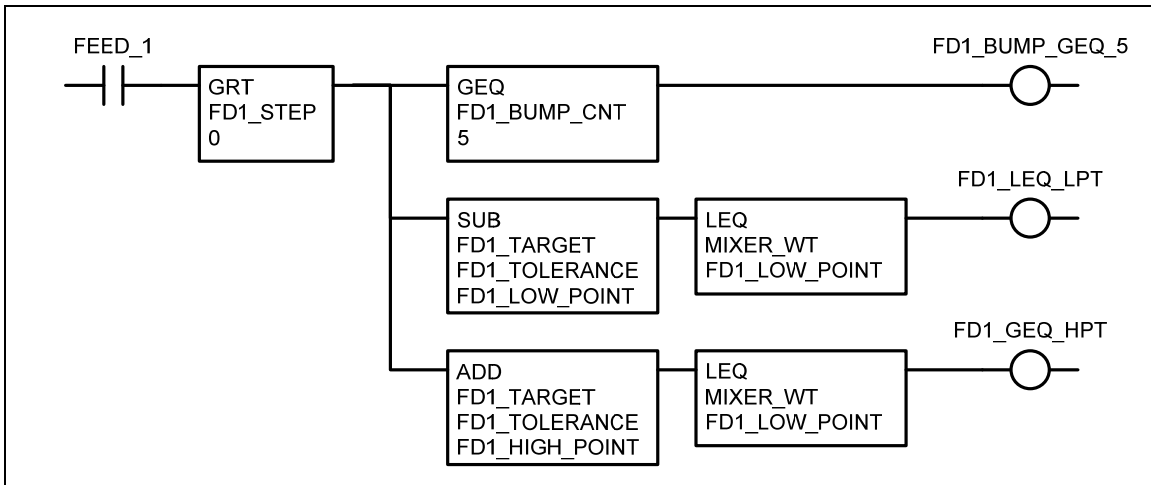


Figure 4-11 Determining the Bump Count, the Low Point and the High Point

At this point the mixer weight has reached the cutoff. In Figure 4-12 we close the valve and start a settling timer. This settling timer will allow the material that is in the pipe after the valve closes to reach the mixer. This is the in-flight material. The settling timer will also allow the scale to stabilize. After the scale has stabilized, the weight is checked to see if it is within tolerance. If the weight is under tolerance then we begin the bump cycle provided that the bump limit has not been reached. If the weight is over tolerance then we will display an alarm and ask the operator to acknowledge the high weight alarm before ending the sequence.

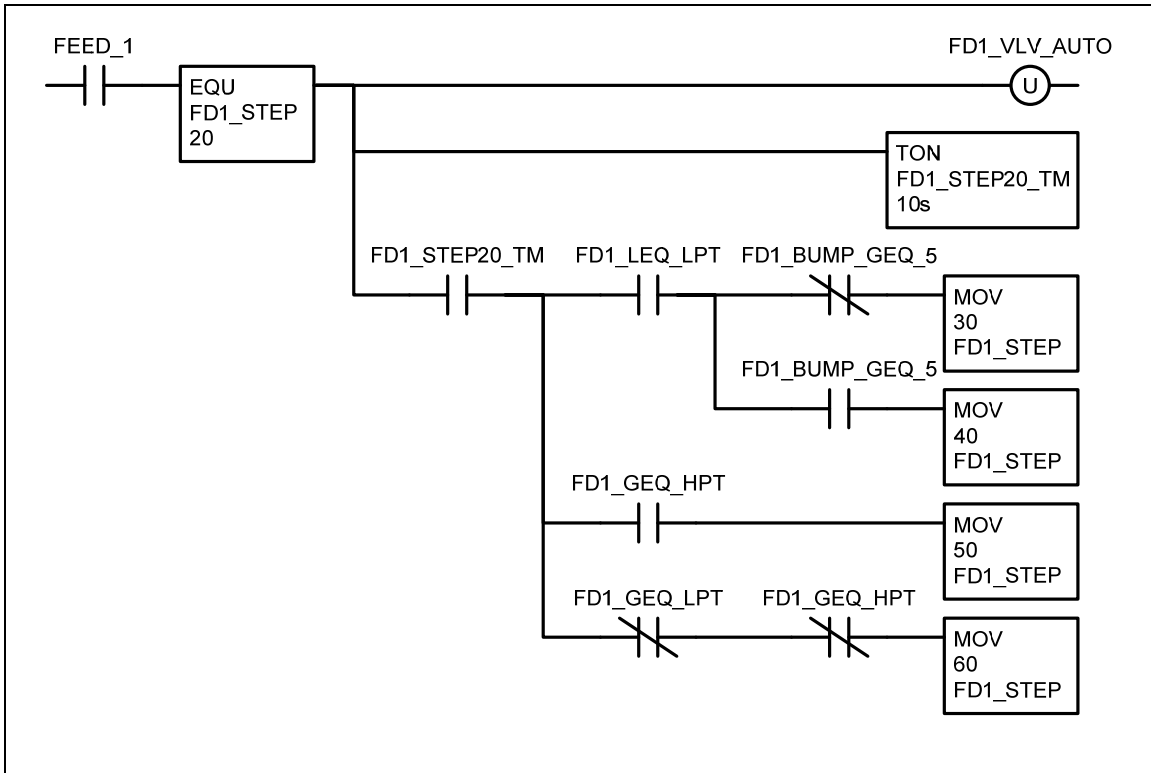


Figure 4-12 Closing the Feed Valve and Checking the Bump, Low Point and High Point

The sequence reaches step 30 if the weight is under tolerance. In Figure 4-13 the feed valve is bumped for ½ second and then the sequence returns to step 20 where the feed valve is closed. We also increment the bump count. FD1_BUMP_LAT lets us know that the feed was under tolerance so when the sequence is ended the in-flight can be adjusted automatically.

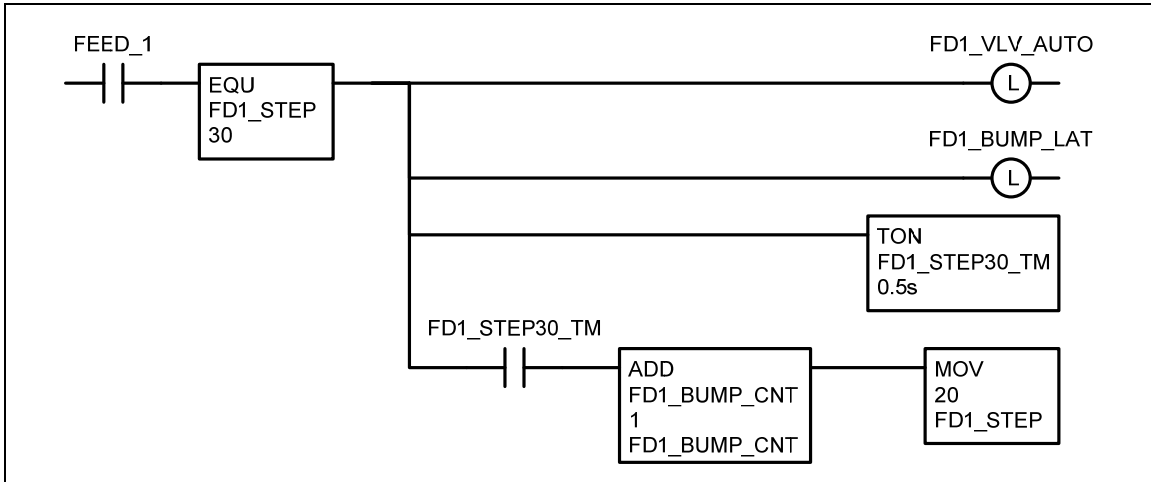


Figure 4-13 Gain in Weight Feeder Bumping the Feed Valve to Achieve Tolerance

We reach step 40 when we have bumped 5 times and the weight is still under tolerance. At this point it is up to the operator to continue the bump cycle by resetting the bump count or allowing the sequence to continue to the end.

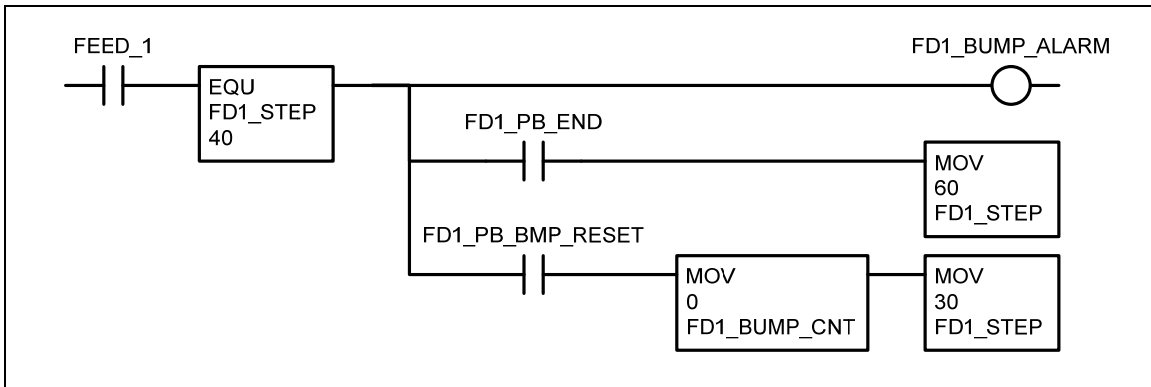


Figure 4-14 Gain in Weight Feeder Bump Alarm

We reach step 40 if the weight is over tolerance. An alarm is displayed and the operator must press the push button to allow the cycle to end. We decrease the in-flight for the next feed provided that the cycle did not bump and automatic in-flight adjustment is selected. If the cycle has bumped and we get a high alarm, it means that after the cutoff, the weight was originally under tolerance. This means that the bump must have caused the high alarm. Under this condition the in-flight will be adjusted to account for the under tolerance condition. Also, the bump time may be too long. This will cause too much material transfer into the mixer during the bump.

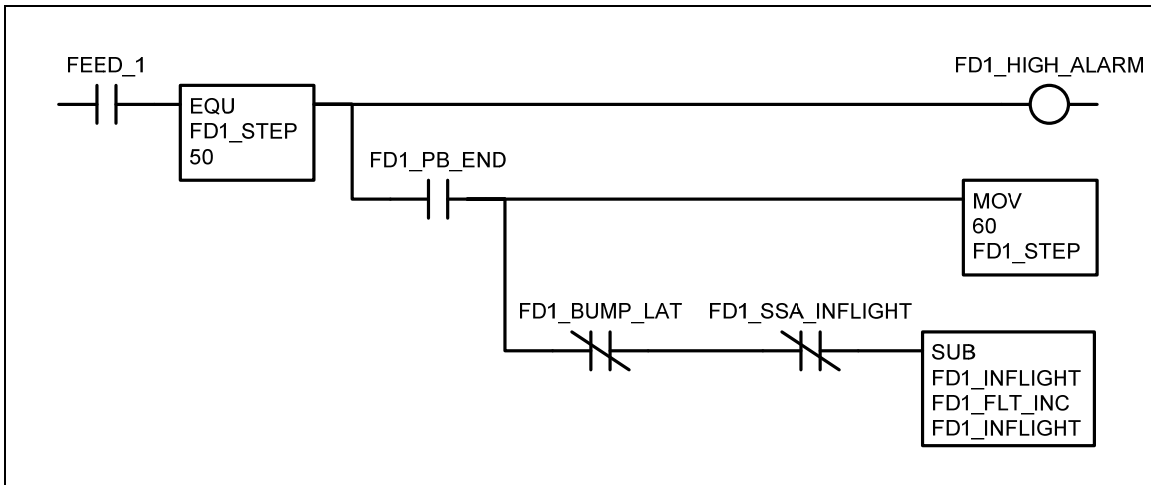


Figure 4-15 Gain in Weight Feeder High Weight Alarm

In Figure 4-16 we adjust the in-flight if the weight was under tolerance and automatic in-flight adjustment is selected.

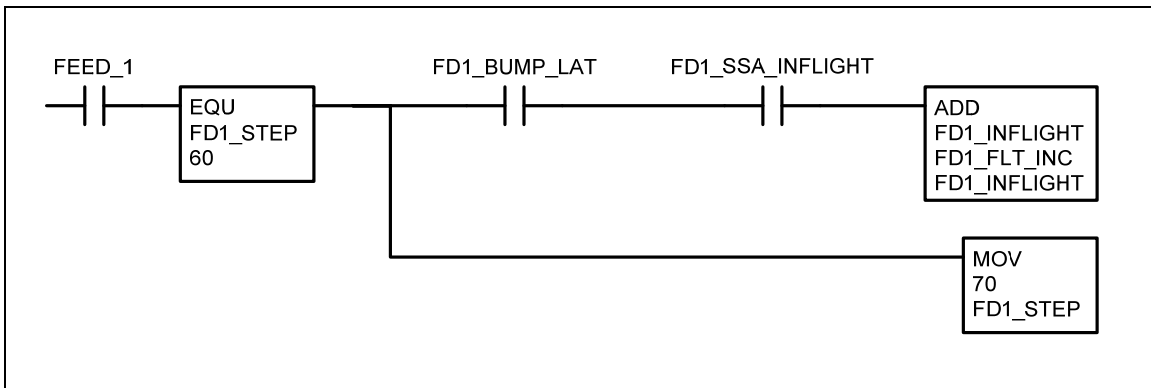


Figure 4-16 Gain in Weight Feeder Adjusting the In-flight

Chapter 4 State Logic

In Figure 4-17 we reset the feed mode and turn on a complete bit. I have chosen not to reset the feed step to zero. In this case it would be up to the batch controller to reset the step and turn on FEED_1 in order to initiate the next feed from silo 1.

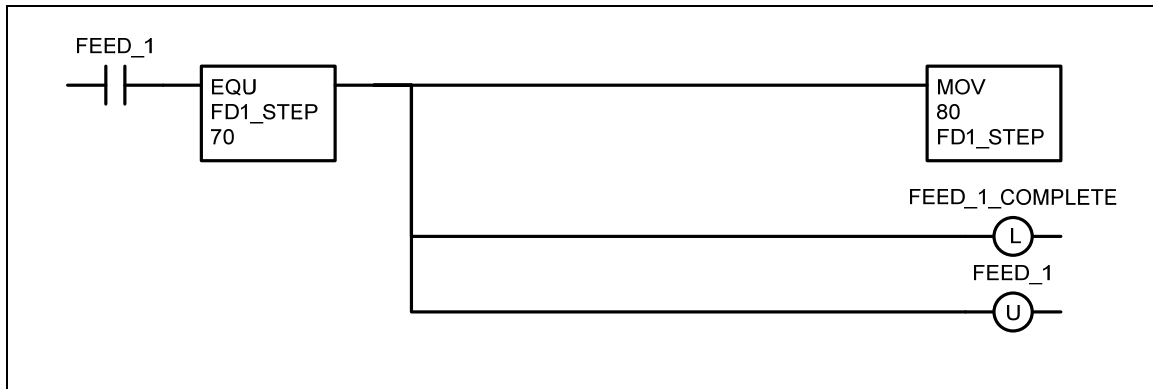


Figure 4-17 Gain in Weight Feeder Complete

State Logic – Third Form

The third form state logic sets all of the auto bits that are controlled by the sequence at each step. This is done by copying the preset state for each bit and setpoint into the destination of each controller.

Table 4-2 shows the elements in a user defined data type with all of the auto bits and setpoints that are affected by the sequence. This data type is called SEQ_AUTO. We then create a variable for each step with this data type, STEP20_AUTO, STEP30_AUTO, etc. We also create a destination with the same type that the valve controllers will use, CTRL_AUTO. I would not create an array of type SEQ_AUTO and then use these array elements in the steps. If I need to insert a step, I would have to move all of the elements in the array that follow the step I want to insert. By using separate variables, I can just create a new variable of type SEQ_AUTO when I need to insert a step.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
VLV1_AUTO_SET	Valve 1 auto set	Boolean
VLV1_AUTO_RESET	Valve 1 auto reset	Boolean
VLV2_AUTO_SET	Valve 2 auto set	Boolean
VLV2_AUTO_RESET	Valve 2 auto reset	Boolean
VLV3_AUTO_SET	Valve 3 auto set	Boolean
VLV3_AUTO_RESET	Valve 3 auto reset	Boolean
VLV4_AUTO_SET	Valve 4 auto set	Boolean
VLV4_AUTO_RESET	Valve 4 auto reset	Boolean
VLV5_AUTO_SET	Valve 5 auto set	Boolean
VLV5_AUTO_RESET	Valve 5 auto reset	Boolean
LVL1_AUTO_SET	Level controller 1 auto on	Boolean
LVL1_SP	Level controller 1 auto setpoint	Real
LVL1_GEQ_SP	Level controller Greater or Equal to Setpoint	Boolean
LVL1_LEQ_SP	Level controller Less or Equal to Setpoint	Boolean
LVL1_DELAY_SP	Level controller at setpoint delay	Dint
TEMP1_AUTO_SET	Temperature controller 1 auto on	Boolean
TEMP1_SP	Temperature controller 1 auto setpoint	Real
TEMP1_GEQ_SP	Temp. 1 controller Greater or Equal to Setpoint	Boolean
TEMP1_LEQ_SP	Temp. controller Less or Equal to Setpoint	Boolean
TEMP1_DELAY_SP	Temp. controller at setpoint delay	Dint
DELAY_SP	Delay timer	Dint

Table 4-2 User Defined Data Type SEQ_AUTO

In Figure 4-18 the step setpoints are copied to the controller auto setpoints. A jump to the subroutine CONFIRM is made. The subroutine will set the transition PENDING_CONFIRM and allow the sequence to continue to the next step.

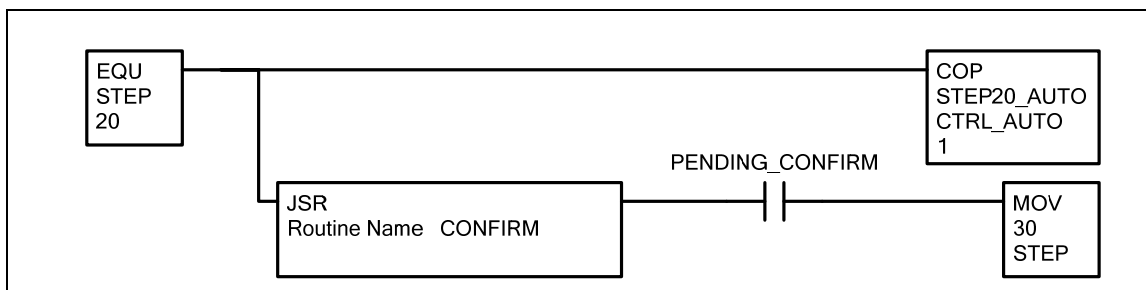


Figure 4-18 State Logic Third Form Example Step

Like the first form state logic, the third form allows the sequence step to be changed by the operator to any state, and all of the outputs will be set correctly for that state.

In Figure 4-2 we monitor the variable STEP20_AUTO, and set the auto bits to open the first 2 valves and close the remaining valves. Valves VLV1 and VLV2 are opened. Valves VLV3 and VLV4 are closed. Valve VLV5 is not changed. The level and temperature controllers are also unchanged.

<i>Name</i>	<i>Value</i>	<i>Type</i>
STEP20_AUTO.VLV1_AUTO_RESET	0	Boolean
STEP20_AUTO.VLV1_AUTO_SET	1	Boolean
STEP20_AUTO.VLV2_AUTO_RESET	0	Boolean
STEP20_AUTO.VLV2_AUTO_SET	1	Boolean
STEP20_AUTO.VLV3_AUTO_RESET	1	Boolean
STEP20_AUTO.VLV3_AUTO_SET	0	Boolean
STEP20_AUTO.VLV4_AUTO_RESET	1	Boolean
STEP20_AUTO.VLV4_AUTO_SET	0	Boolean
STEP20_AUTO.VLV5_AUTO_RESET	0	Boolean
STEP20_AUTO.VLV5_AUTO_SET	0	Boolean
STEP20_AUTO.LVL1_AUTO_SET	0	Boolean
STEP20_AUTO.LVL1_SP	0	Real
STEP20_AUTO.LVL1_GEQ_SP	0	Boolean
STEP20_AUTO.LVL1_LEQ_SP	0	Boolean
STEP20_AUTO.LVL1_DELAY_SP	0	Dint
STEP20_AUTO.TEMP1_AUTO_SET	0	Boolean
STEP20_AUTO.TEMP1_SP	0	Real
STEP20_AUTO.TEMP1_GEQ_SP	0	Boolean
STEP20_AUTO.TEMP1_LEQ_SP	0	Boolean
STEP20_AUTO.TEMP1_DELAY_SP	0	Dint
STEP20_AUTO.DELAY_SP	0	Dint

Table 4-3 Step 20 Auto Setpoints

In Figure 4-19 the outputs are programmed using the user defined variable CTRL_AUTO. Each valve would be programmed in a similar way.

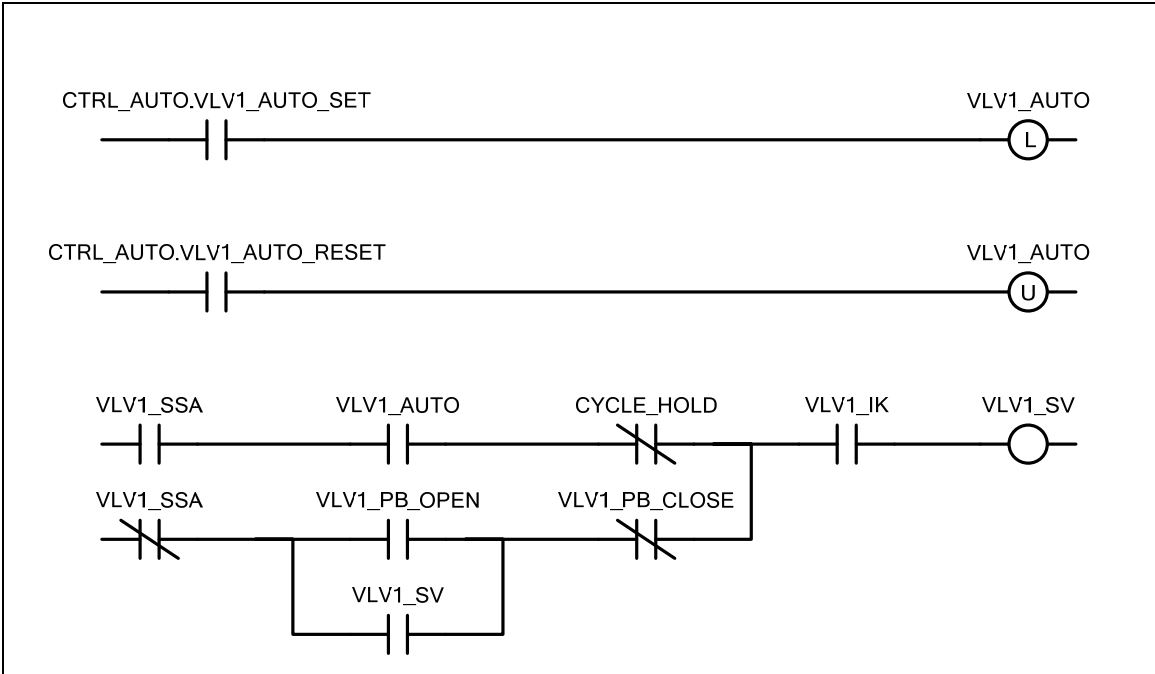


Figure 4-19 State Logic Third Form, Valve Output Logic

The first rung in the routine CONFIRM is shown in Figure 4-20. The step is checked to see if it has changed. This will indicate that this is the first scan of the routine since the step change. The bit RESET_TIMERS will then be used to re-initialize the timers used in the routine. The current step is moved into the PREVIOUS_STEP variable

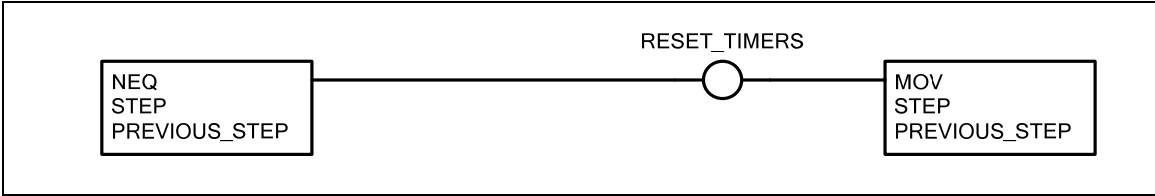


Figure 4-20 Confirm Routine, Reset the Timers

In Figure 4-21 the valve position is confirmed. The confirm bits for each transition type will be summed at the end of the routine. This will allow the sequence to continue to the next step. Each valve would be programmed in the same way. This code is also in the CONFIRM routine.

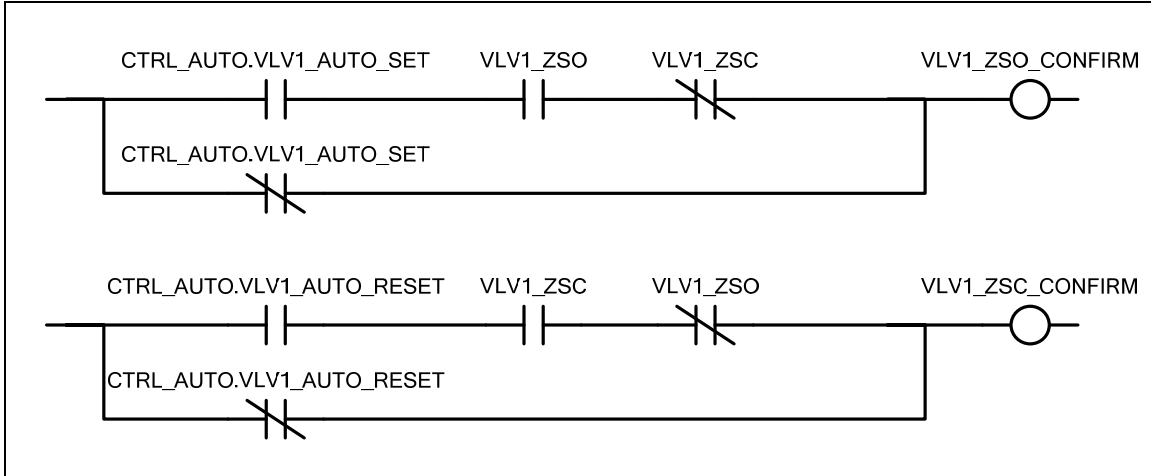


Figure 4-21 Confirm Routine Valve Position Confirm.

In Figure 4-22 the level is compared against the setpoint. When the setpoint is reached the confirm bit is turned on. The delay timer allows the sequence to remain at or above the setpoint for the duration of the delay preset. If the level falls below the setpoint then the timer is reset. Each comparison transition is programmed similarly.

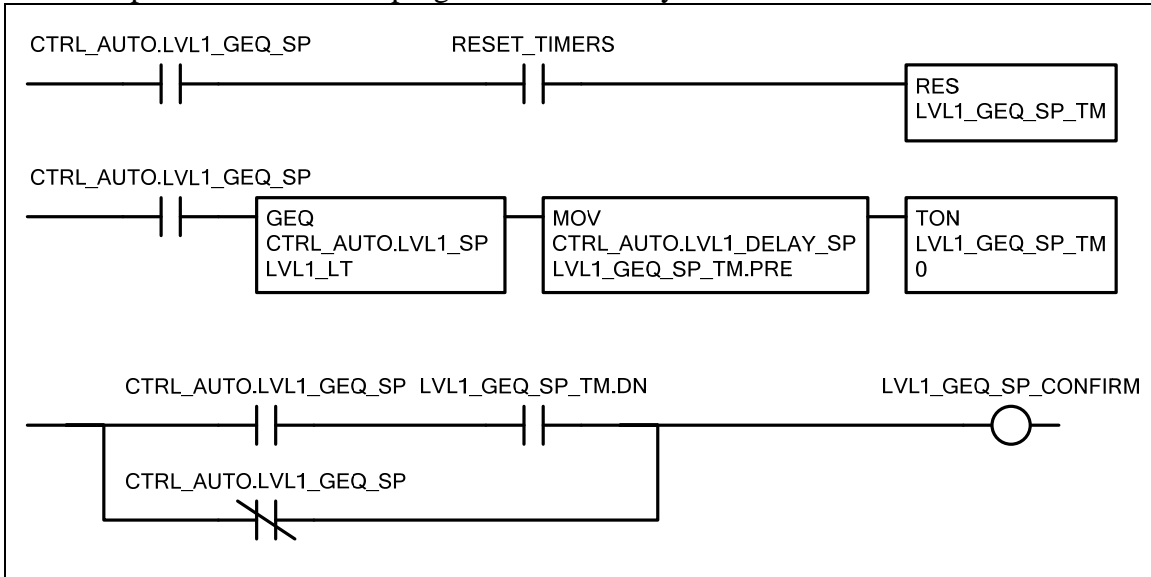


Figure 4-22 Confirm Routine Setpoint Comparison Confirm

In Figure 4-23 the delay transition is shown. This transition does not have an enable like the others. If a delay is not desired then the preset is set to zero for that step.

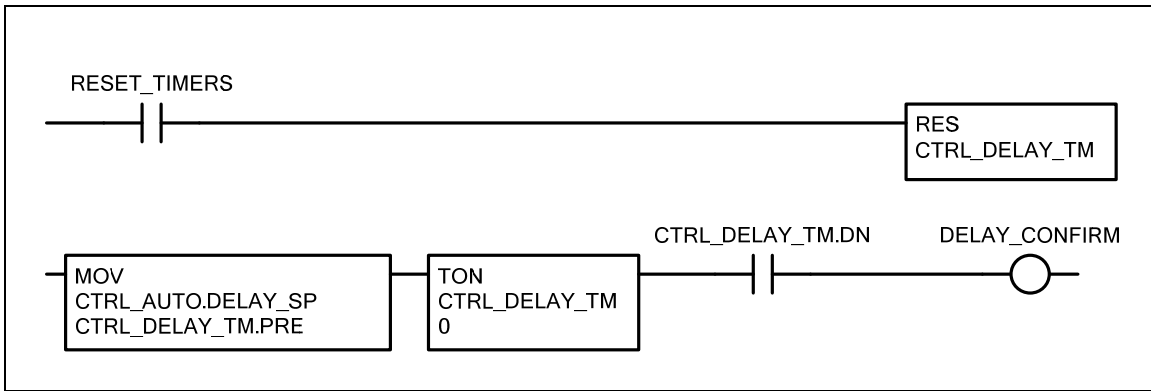


Figure 4-23 Confirm Routine, DELAY_CONFIRM

In Figure 4-24 all of the confirm bits for each transition are summed into the PENDING_CONFIRM bit. This variable is then used to transition the sequence to the next step. This is the last rung in the CONFIRM routine.

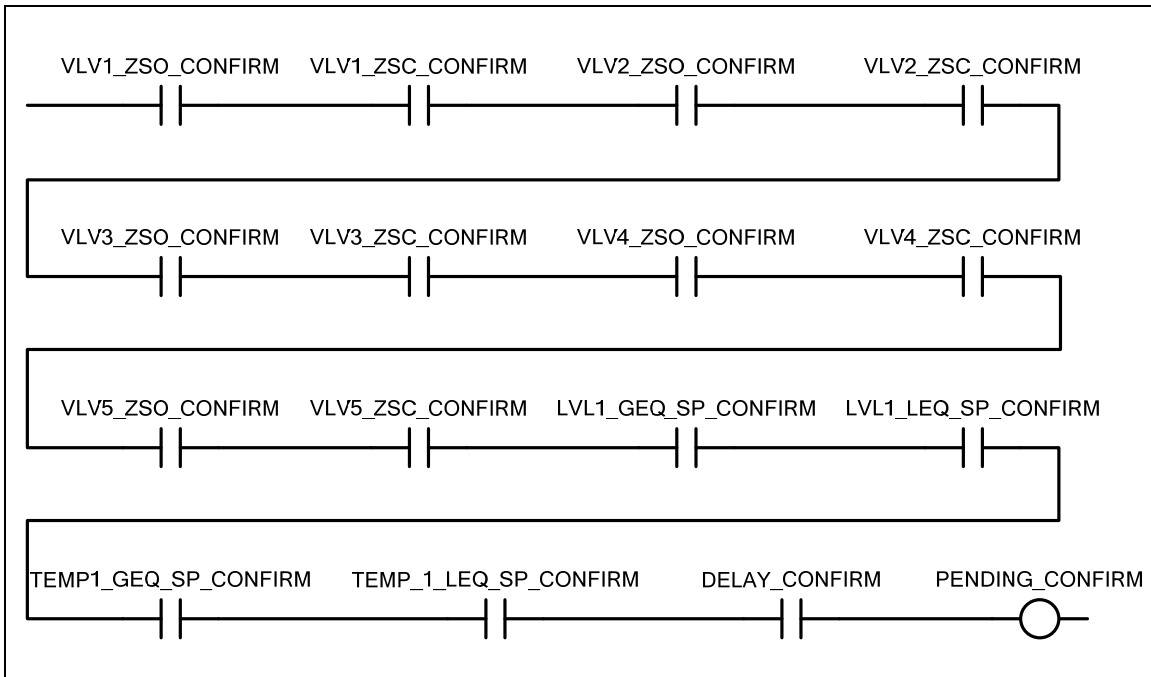


Figure 4-24 The end of the Confirm Routine Confirm Summation

Chapter 5 Batch Control

Batch control can be divided into two categories. There is the ISA-88.01 Batch Control Standard¹. And, there is all the rest. There are several companies who are selling products that comply with the ISA standard. These products are usually very functional and very expensive. The architecture usually consists of a computer where the recipes reside and a controller where the phases reside. The computer controls the execution of the recipe and instructs the controller when to execute each phase in the recipe.

We will consider a method of batch control that implements part of the ISA standard. This type a batch architecture is especially valid for an OEM user who is cost conscious and can reuse the batch engine repeatedly on new systems. We will use a database to store our recipes and then download those recipes into the controller where they will be executed. A phase is a specific task that can be included in a recipe. A recipe will consist of phases and parameters which are executed in a sequence that can be changed. Each phase will be represented by a unique number. These numbers will be stored in PLC memory in the order that the phases are to be executed.

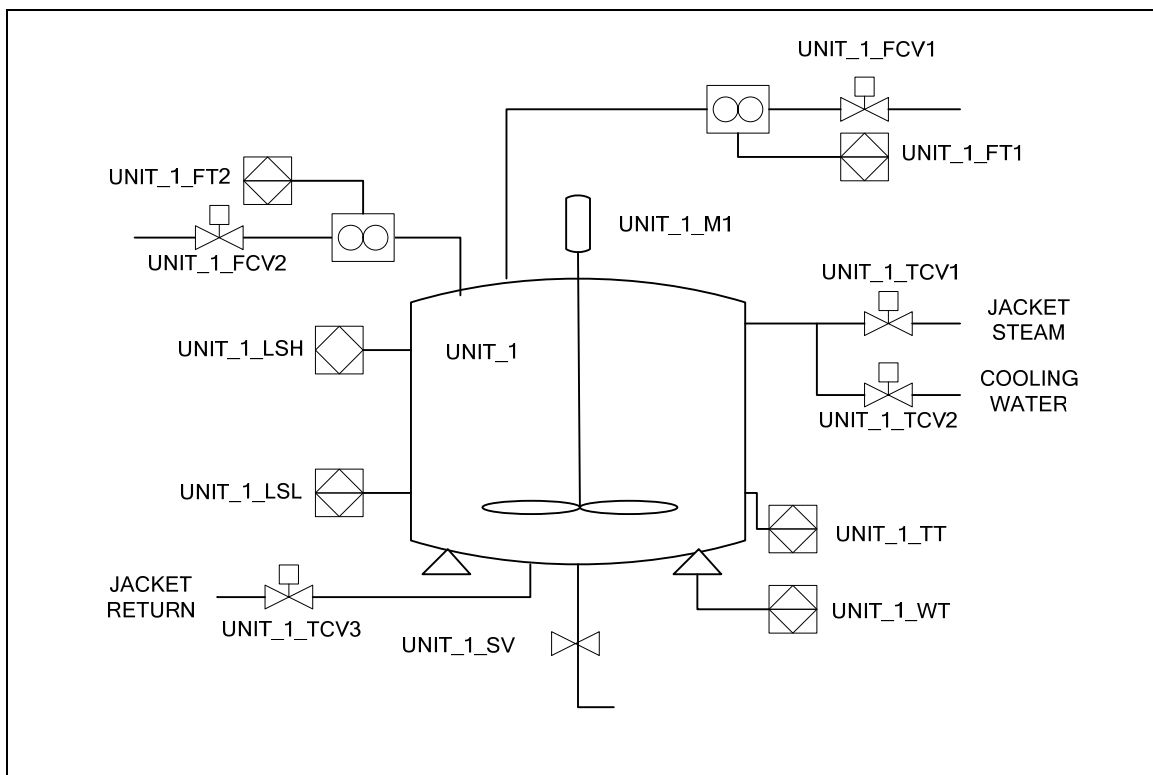


Figure 5-1 The batching process and instrumentation diagram P&ID

¹ Refer to the ISA (Instrument Society of America) website for a copy of the 88.01 Batch standard. <http://www.isa.org>. Or, refer to Rockwells RSBatch online manual for their implementation of the standard. <http://www.ab.com/manuals/swrsi/BATCHTD001AENP.pdf>

Batch Logic

The batch logic is divided into the following groups.

- Recipe Management – This block of logic will control the editing and manipulation of each recipe.
- Batch Control – This block of logic will control how and when each phase is executed in the recipe.
- Phase Logic – Each phase is programmed separately. The batch control logic will initiate the phase. When the phase is complete its status is changed to complete. This complete status can be monitored to allow the sequence to continue.
- Controller Logic- Each physical device such as a motor or valve will have logic that will allow the device to be controlled by the phase or by manual control.

Figure 5-2 shows how the batching program can be organized.

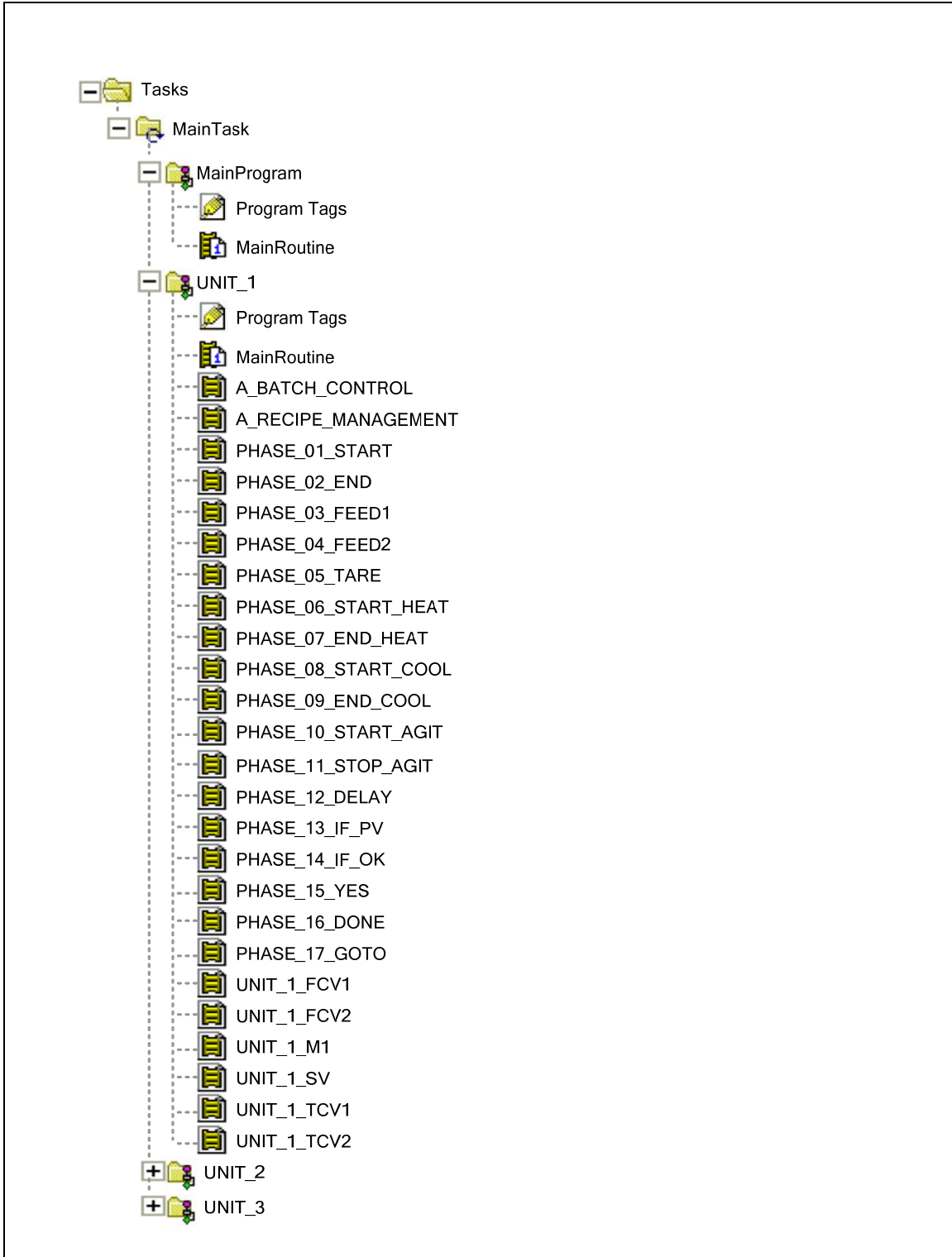


Figure 5-2 Batching program organization

Recipe Organization

We can organize the batch data into a few user defined data type arrays. Recipe data is stored in one array. Phase data is stored in another. Another array will contain the report data. Table 5-1 shows the user defined data type for the recipe. An array of this data type will constitute a recipe or sequence of phases that are executed by the batch controller.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
STEP_ID	Step Id	Dint
PHASE_ID	Phase ID	Dint
P1	Parameter 1	Real
P2	Parameter 2	Real
P3	Parameter 3	Real

Table 5-1 Recipe User Defined Data Type REC

When a phase is initiated, the recipe parameters will be copied to the phase parameters. This will allow the phase parameters to be changed manually without affecting the recipe. Also if the phase is initiated externally outside the recipe sequence, it will have its own copy of the recipe parameters. These phase parameters will be indexed by the phase number. The file length is determined by the maximum number of phases.

Table 5-2 lists each variable in the phase user defined data type.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
RS	Running Start	Boolean
R	Running	Boolean
RC	Running complete	Boolean
RP	Report Pointer	Dint
SI	Step Index	Dint
P1	Parameter 1	Real
P2	Parameter 2	Real
P3	Parameter 3	Real

Table 5-2 User Defined Data Type PHASE

This approach is efficient in the way it accesses all of the phases in the recipe. However, it is wasteful in that all phases will not use all of the parameters that have been reserved in memory for them. Another approach would save a parameter pointer for each phase. All of the parameters would be stored in the same file and stacked one on top of the other. This approach would be more efficient in memory but it would require more programming overhead to deal with the variable length parameter storage. We could also stack the phases and parameters in the same file in either a fixed or variable length block of memory. The phase type would determine how much memory it would use in the variable length method. The phases and parameters can also be organized contiguously in memory with each phase and parameters etc. requiring a fixed amount of memory. I would tend to steer away from any method requiring a variable length block of memory. This will simplify the logic required to deal with the data. Also, the type of processor that you are using may determine the best method of organizing the data.

A record of the batch history would include report parameters for each instance of each phase in the recipe. This batch history can be saved in its entirety in the PLC until the recipe has completed or the history data can be uploaded to an HMI individually for each phase as it occurs during the execution of the recipe.

Let's assume we are storing all of the report parameters in the PLC until the end of the batch. If we design our recipe where a phase can be ran more than once during the execution of the recipe then we need to be able to save more report parameters than we have steps in the recipe. If we are looping through some phases then the number of initiated phases can quickly exceed the amount of memory reserved for the report parameters. This can be handled by truncating the batch report when the memory is exceeded. Or, we can store the report parameters with a step index thus saving only the report parameters for the last instance of the phase that has completed.

Table 5-3 shows the user defined data type for the report parameter array. As each phase is initiated the batch controller will increment a report pointer. This pointer value is then associated with the current phase that is initiated. This pointer will then give the phase a location for recording its batch parameters so that each phase will be recorded in the order that it is initiated. For example let's assume that a feed material phase with a phase ID of 4 is at step 15 of the recipe. Let's also assume that there was some sort of loop in the recipe sequence which allowed the report pointer to reach 21. Then as the batch controller initiates step 15 the data for that phase is saved in position 21 of the recipe report array.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
STEP_ID	Report Step	Dint
PHASE_ID	Report Phase	Dint
START_TM	Report Phase Start Time	Time
END_TM	Report Phase End Time	Time
MIX_WT	Mixer Weight	Real
MIX_TT	Mixer Temperature	Real
P1	Report Parameter 1	Real
P2	Report Parameter 2	Real
P3	Report Parameter 3	Real

Table 5-3 Report Parameter User Defined Data Type REP

Figure 5-3 shows the data flow for the recipe parameters. The stored recipes will be stored in an array called STG[]. The running recipe is REC[]. The phase parameters are stored in array PHASE[]. The controller parameters are stored in local controller variables. To start a batch the recipe is selected from recipe storage. The recipe is then copied into the running recipe. At this point the operator could modify the parameters in the running recipe without effecting recipe storage. The batch is then started. As each phase is initiated the recipe parameters are copied to phase parameters. The operator could modify the phase parameters after the phase is started or the phase could be executed externally outside of the batch controller. In this case the phase parameters could be modified without changing the running recipe parameters. When the phase is initiated the phase logic copies the phase parameters to the controller parameters. For example, the start heat phase would copy the temperature setpoint from the start heat phase parameters to

the local heating controller. The mode of the controller could then be changed to local control, and the setpoint of the heat controller could be changed without changing the phase parameters.

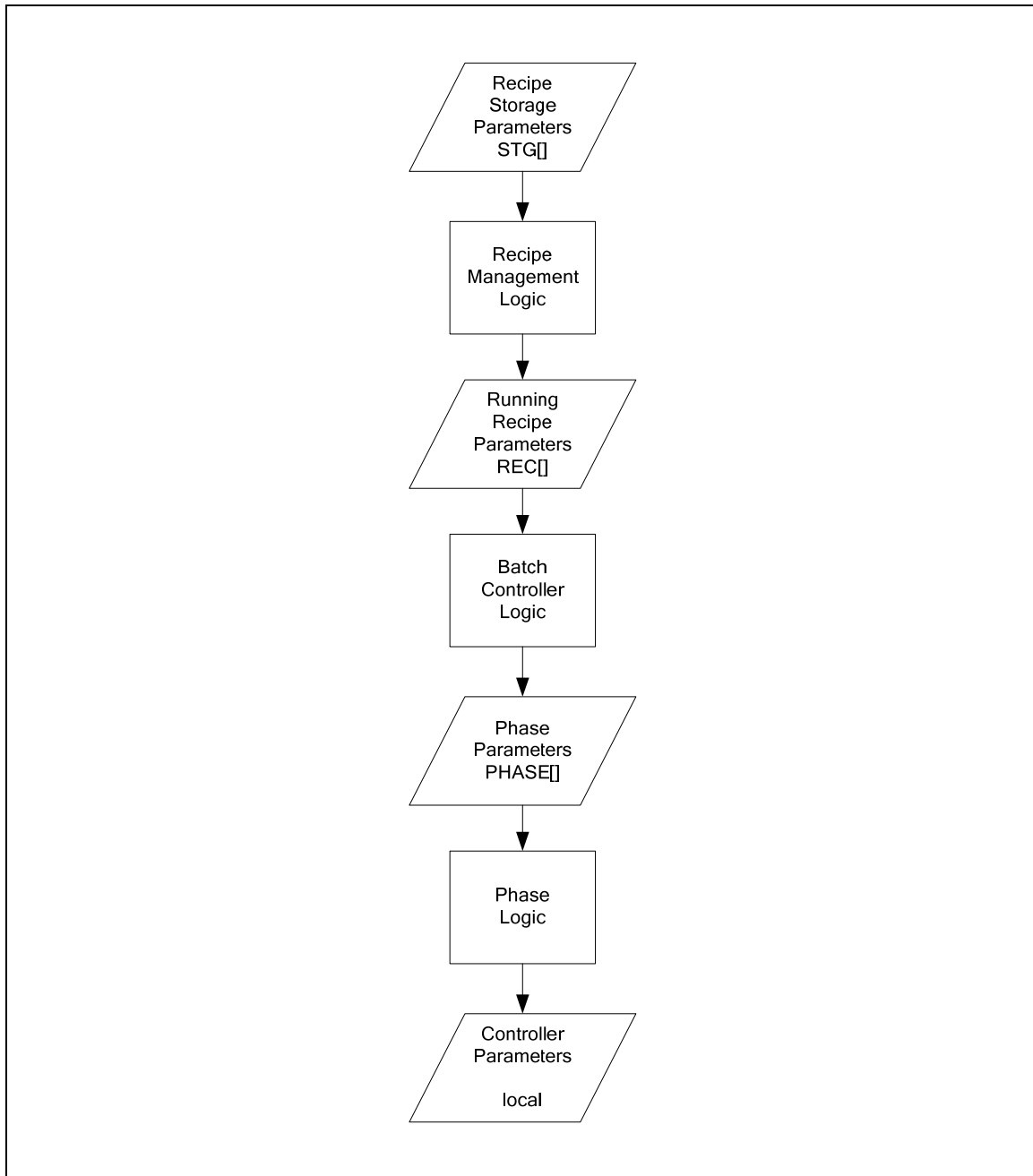


Figure 5-3 Recipe Parameter Data Flow

Recipe Management

Before you can determine how to program the recipe management in the PLC we need to ask and answer a few questions.

- How many recipes are there?
The answer to this question may determine how we store the recipes. If there are only a few recipes we may be able to store them all in the PLC. If there are many we will need to store them on a computer and download them when they are needed.
- Who is going to be responsible for managing the recipes?
The answer to this question may determine what type of application will be required to develop and manage the recipes.
- Where are the recipes going to be stored?
- What kind of hardware is available to manage and edit the recipes?
We can either manage the recipes from a computer and then download them to the PLC or we can edit recipes in the PLC directly from a HMI or graphic terminal.
- How much time and money do you want to spend to write recipe management logic?
Time is money. You have to weigh your investment before you begin programming. We will go into detail for the ladder logic portion of batch control. If you determine that you need a database then you have to apply additional expertise in database and communication programming.

The batch controller will execute one copy of the recipe. This is called the running recipe. The recipe is edited using the edit recipe. The running recipe can also be viewed through the HMI. It must however be loaded into the edit recipe for editing. Figure 5-4 shows how the recipes can be organized.

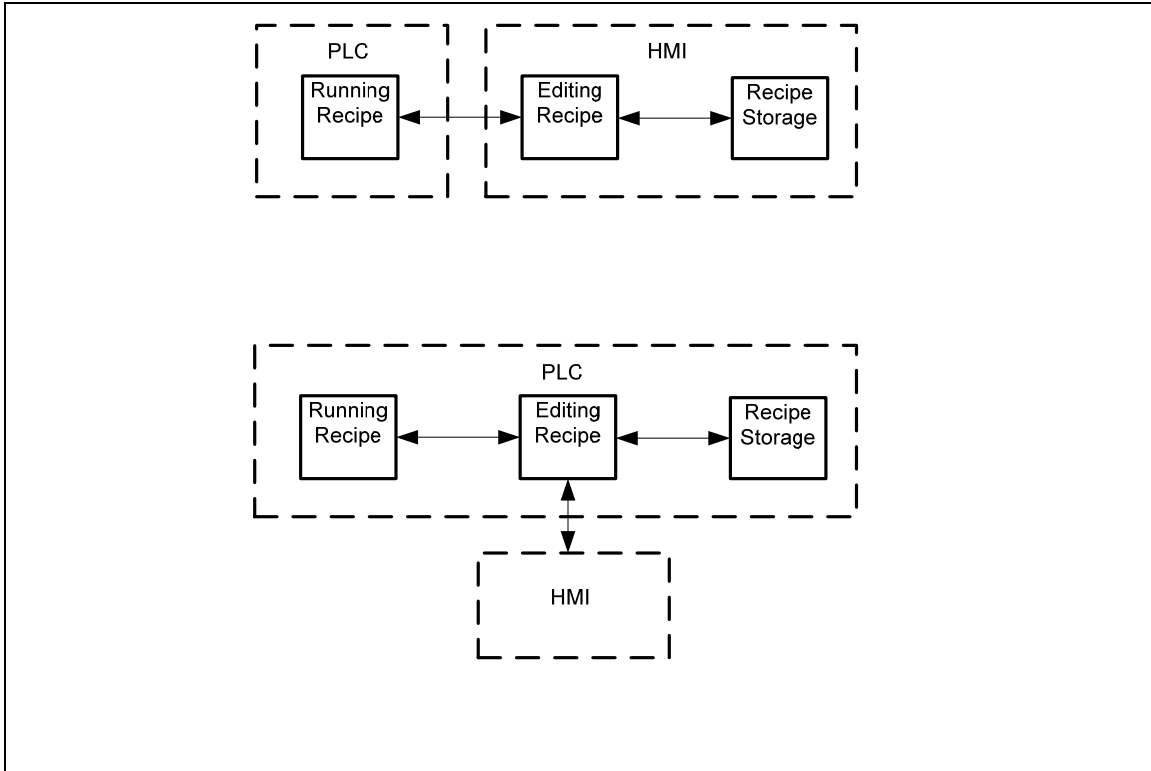


Figure 5-4 Recipe Storage Options

Batch Control

There are several ways that you can design the way your phases are executed. There are several reasons why you may choose one method over the other. These could include PLC memory limitations, Recipe simplicity, or data file length limits.

The first method would require that each phase complete before the sequence continues. This would eliminate the need for any transitions since they would be built into the phase. The batch controller would then only have to monitor the complete status of the current phase and then continue to the next phase. This would make your recipes simple but by doing this, the batch controller would only allow one phase at a time to be running.

With another approach we would include a continue flag with all or some phases. This flag would be considered an additional parameter in each phase. The continue flag would be available to the operator or recipe manager. If the flag were set the batch controller would initiate the phase and then continue to the next. If the flag were not set the batch controller would wait for the phase to complete before continuing to the next phase. An additional phase would be needed to check the complete status of a phase. This phase would then be considered a transition. One of the parameters to this phase would be the phase number which would act as a pointer to the phase whose status we are checking. This approach will allow more than one phase at a time to be running without necessarily increasing the size and complexity of the recipe.

Another approach would make the batch controller initiate the phase and then continue to the next. The recipe would then consist of phases and transitions. This approach more closely

resembles the ISA-88.01 batch standard but it also makes the recipe longer and more complex. We could store phases in one file and transitions in another. If we did this then there would need to be a one to one relationship between phases and transitions so that we could use a single pointer to point to both. For those phases that we would not want to stop and wait, we would make a True or null transition. We could also store the phases and transitions in the same file. By doing this we only need to include in the recipe those transitions where we are waiting for some event. Since the phases and transitions reside in the same space we could consider them the same. Transitions would then be hybrid phases that would cause the batch controller to stop and wait for the phase to complete. This method would still use the continue flag to let the batch controller know when to stop or continue without waiting for the phase to be done. The only difference would be the continue flag would not be available to the recipe manager.

Up until now we have spoke of the batch controller as either stopping and waiting for the phase to complete, or initiating a phase and then continuing. At any given point in time we could monitor the batch and we would have one phase or transition where the batch controller was waiting before it would continue. This type of batch control will allow more than one phase to be active at a time but only in a limited way. In order to achieve true parallel branching in your recipe we will consider an additional approach. The batch controller will scan the recipe in a cyclical fashion. Each phase and transition will have an active status and a complete status. As the batch controller scans the recipe it will consider the current phase and the next phase. If the status is active the batch controller will move to the next phase without taking any action. If the phase is complete then the batch controller will reset the active status. The next phase is marked as active. If the next phase had been run previously then the complete status is cleared. A begin branch phase will allow multiple phases to be activated. This phase activates the next phase and then causes the batch controller to jump to the next branch phase. An end branch phase causes the batch controller to continue on the next phase. Transitions would be treated like phases. When the transition condition is true then the transition phase would be complete and the next phase would be activated by the batch controller. As you can see this method would allow true parallel branching in a recipe. However, this approach is complex and most processes I have found that most small to medium sized applications don't require true parallel branching in the recipes.

In Figure 5-5 the batch reset push button initializes the batch control variables. The report parameters are cleared and the phase status and parameters are cleared. The report parameter array is cleared by copying element zero of the array to element 1 with a length of 199. The copy instruction works by copying the first element of the source to the first element of the destination. Then the second element is copied and so on. Since the source and destination arrays are the same, this has the effect of copying the first element, in this case REP[0] to the remaining elements in the array. Since the report parameters will be stored in the REP array beginning with element 1, element zero can be used to clear the array.

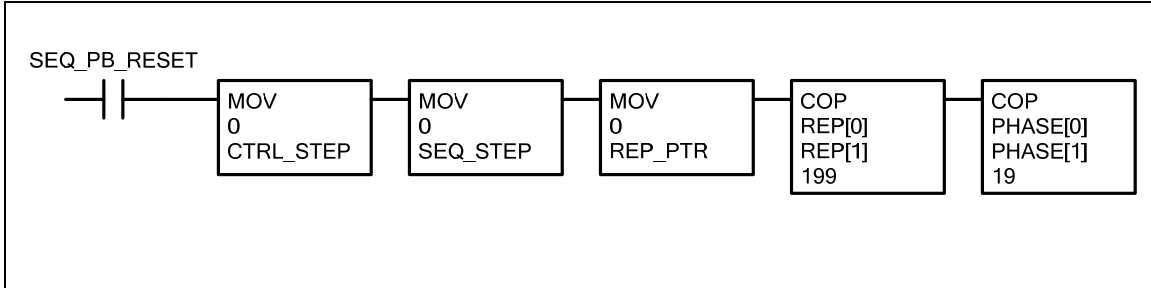


Figure 5-5 Batch Control Reset

After the sequence has been reset, the start push button is used to set the sequence to the first phase in the recipe. The batch controller can either be in the auto or single step mode.

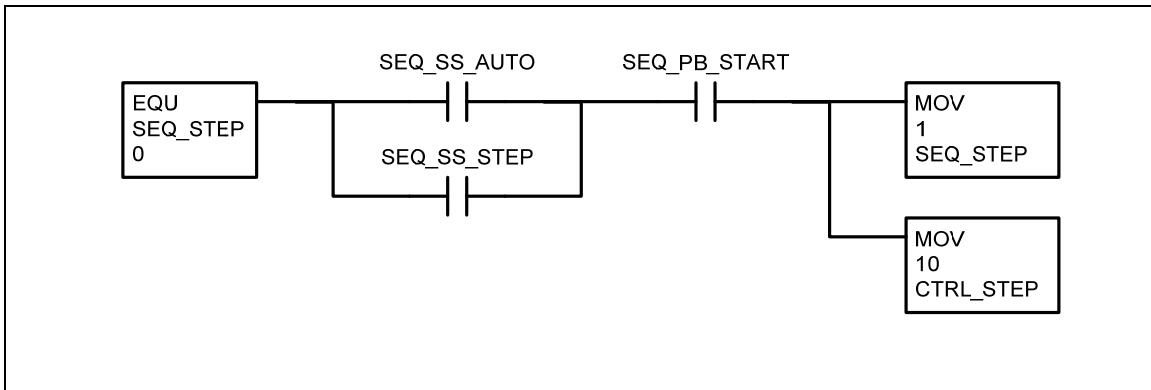


Figure 5-6 Batch Control Initiate the Batch Controller

In Figure 5-7 the phase id pointed to by the sequence step is moved from the recipe user defined array into the variable SEQ_PHASE. The transition bit is used to identify which phases that the batch controller must stop at and wait for the complete bit to be set by the phase. The transition bit is similar to the continue flag discussed earlier but it is not stored in the recipe or the phase array. And, it can not be changed as part of the recipe.

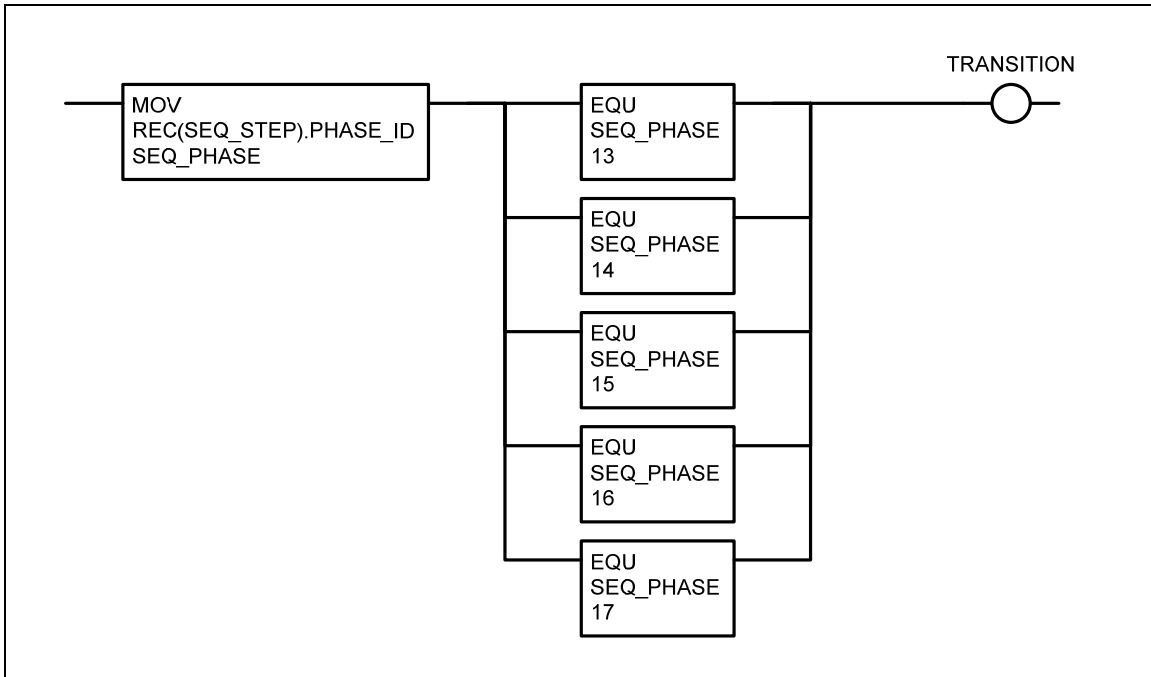


Figure 5-7 Batch Control Identify Transition Phases

In Figure 5-8 the phase is started by setting the running start bit (.RS) for the phase. The running complete bit is reset (.RC). The UPDATE_PHASE bit is set to update the phase parameters in the next rung. The sequence phase must not be equal to zero. It will be zero when the end of the recipe is reached.

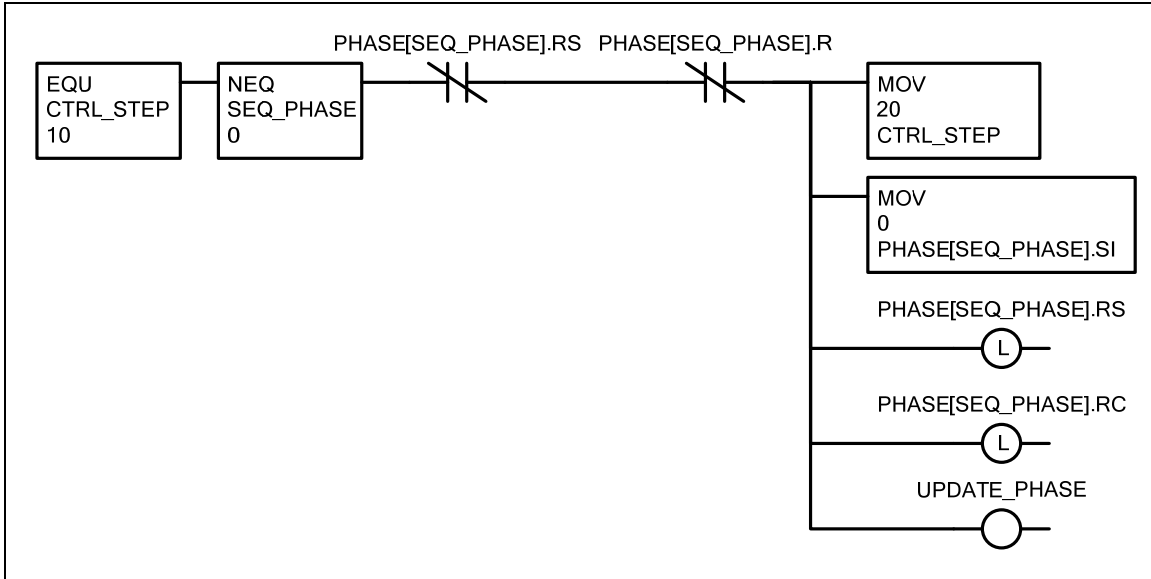


Figure 5-8 Batch Control Reset the Phase and Prepare to Update Parameters

In Figure 5-9 the parameters are copied from the recipe parameters into the phase parameters. The report pointer is incremented and is moved into the phase parameter RP. The phase will use this pointer to copy its report parameters into the report array. We also put the phase ID in the report array along with the step ID.

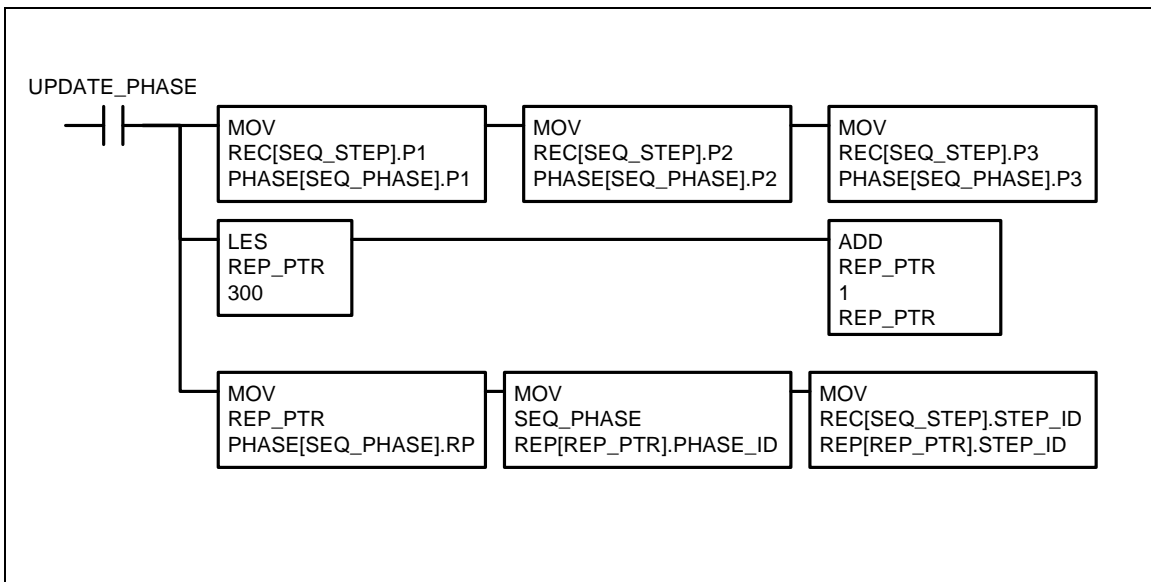


Figure 5-9 Batch Control Update the Phase Parameters and Set the Report Pointer

If the phase is a transitional then the complete bit must be set by the phase logic.

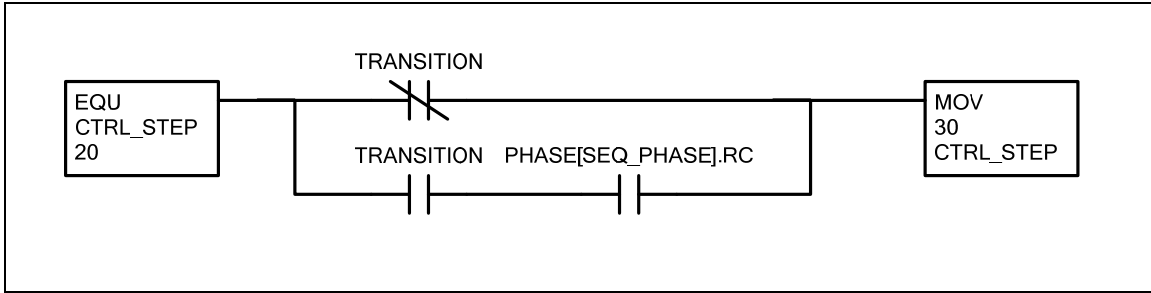


Figure 5-10 Batch Control Wait for Transition Phases to Complete

In Figure 5-11 the sequence step is incremented to point to the next phase in the recipe. The batch control step CTRL_STEP is reset to initiate the next phase on the next program scan.

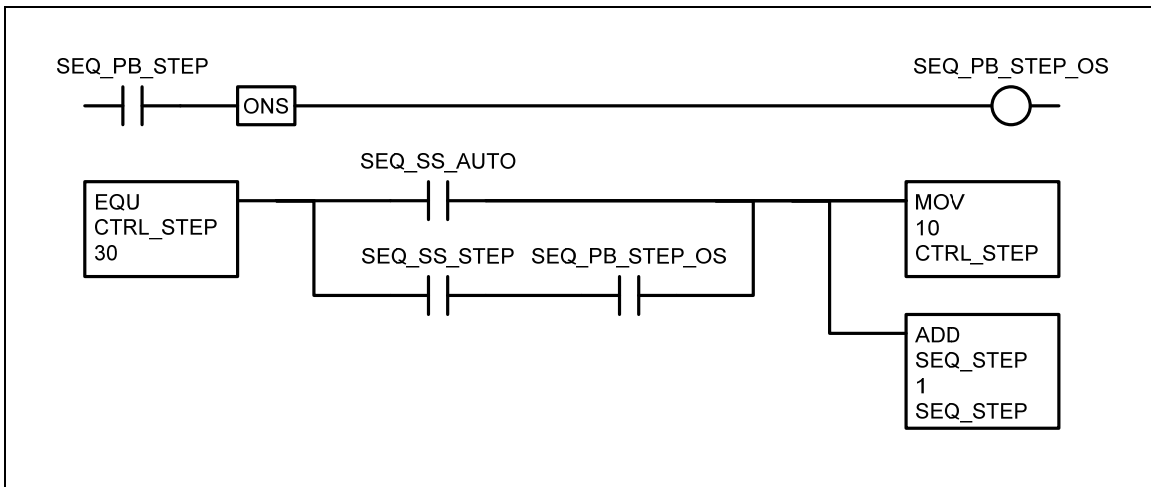


Figure 5-11 Batch Control Increment the Batch Controller Sequence Step

Phase Logic

Defining the phases

The following is a list of phases for a mixing process that we will use as an example. An alias is defined for each element in the PHASE array.

<i>Alias</i>	<i>Array</i>	<i>Phase Name</i>	<i>Parameter 1</i>	<i>Parameter 2</i>	<i>Parameter 3</i>
START	PHASE[1]	Start			
END	PHASE[2]	End			
FEED1	PHASE[3]	Feed Material 1	Amount		
FEED2	PHASE[4]	Feed Material 2	Amount		
TARE	PHASE[5]	Tare Scale			
START_HEAT	PHASE[6]	Start Heat	Temperature		
END_HEAT	PHASE[7]	End Heat			
START_COOL	PHASE[8]	Start Cooling	Temperature		
END_COOL	PHASE[9]	End Cooling			
START_AGIT	PHASE[10]	Start Agitator	RPM		
STOP_AGIT	PHASE[11]	Stop Agitator			
DELAY	PHASE[12]	Delay	Preset		
IF_PV	PHASE[13]	If Process Variable	PV pointer	Setpoint	Operator
IF_OK	PHASE[14]	If OK Operator Prompt	Message Pointer	Value	
IF_YES	PHASE[15]	If Yes Operator Prompt	Message Pointer	Value	Else GoTo
IF_DONE	PHASE[16]	Wait For Phase Complete	Step Number		
GOTO	PHASE[17]	Go To Step	Step Number		

Table 5-4 Phase List

Chapter 5 Batch Control

Each phase will begin with the same logic as shown in Figure 5-12. The phase start time is recorded in the report parameter. The word “Phase” would be replaced with the phase alias name. For example, Phase.R would become START.R for the start phase. The report pointer, Phase.REP_PTR, is set in the batch control logic. The report parameters are saved in the array REP[] of user defined variables described earlier in Table 5-3. The batch engine will turn on the phase running start bit Phase.RS. The batch engine will also reset the start phase step index (Phase.SI). The step index is set to 10 and the phase running bit, Phase.R, is set.

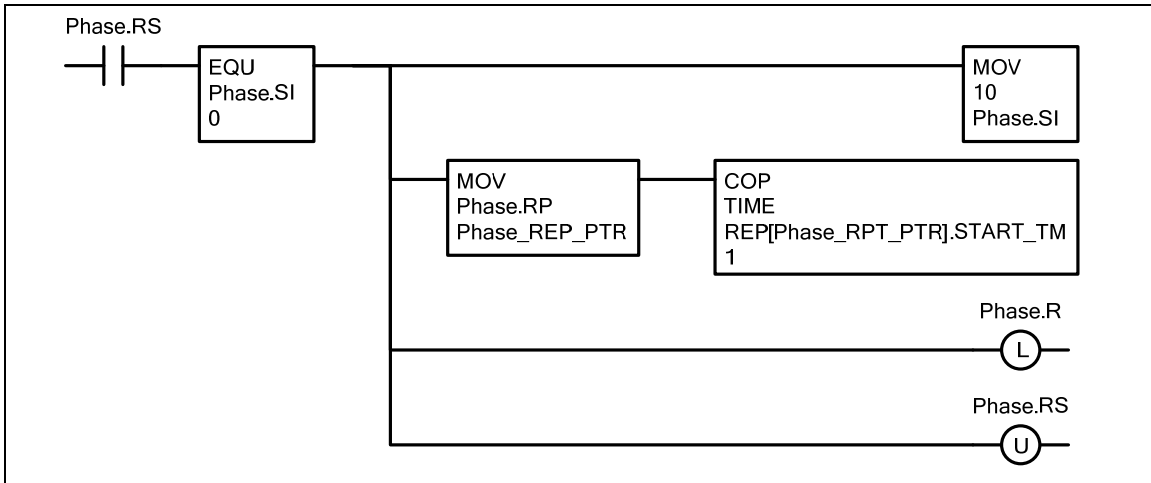


Figure 5-12 Common Phase Logic Set the Start Time and Run

In Figure 5-13 additional report parameters are recorded. This logic will remain active as long as the phase is running. Again this rung would be repeated in the logic for each phase. The mixer weight and mixer temperature are recorded. The phase parameters are recorded. Any changes that the operator makes to the phase parameters while the phase is active will show up in the report parameter. The phase end time is also recorded.

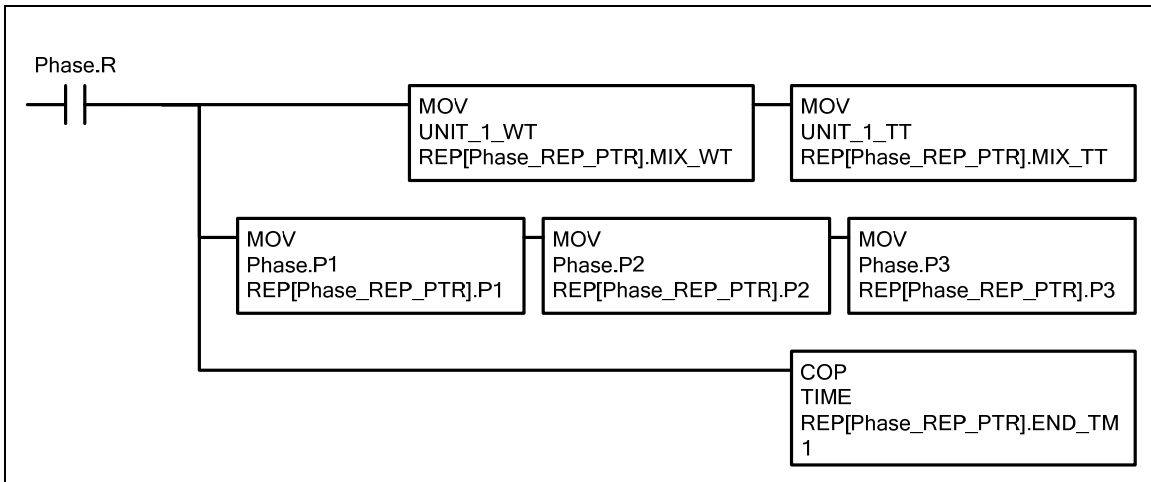


Figure 5-13 Common Phase Logic Record the Phase Parameters and Set the End Time

Figure 5-14 contains the logic that is repeated at the end of each phase. The step index can be adjusted to match the phase sequence.

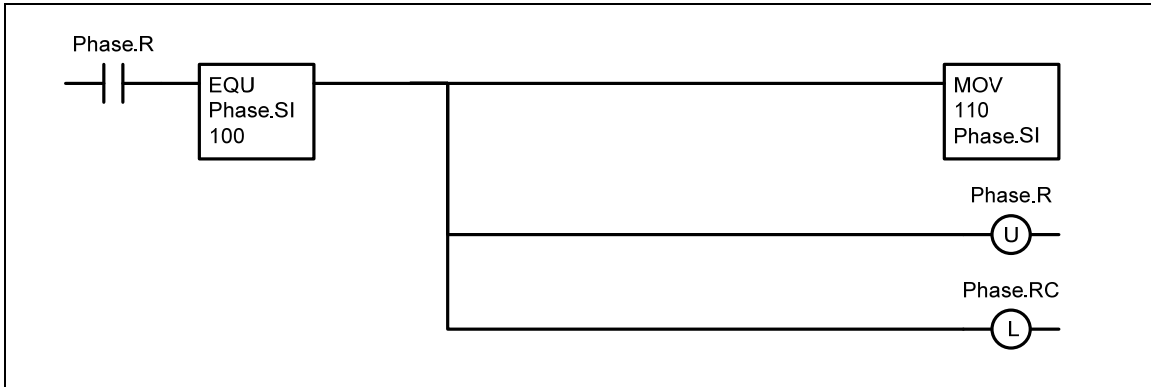


Figure 5-14 Common Phase Logic Phase Running Complete

Start

The Start phase will close all valves and check any conditions that are necessary before the batch begins.

In Figure 5-15 we reset the auto bit for each valve ie UNIT_1_SV_AUTO. This bit will be used to open or close the valve when the mode of the valve is auto. The mode of the valve is changed to auto with the auto select bit ie UNIT_1_SV_SSA. The start phase could also be used to clear totals, set the start batch time, etc. The Start Phase could also just check to make sure that all of the devices are in the auto mode and alarm if they are not. This would prevent the phase from overriding an operator action.

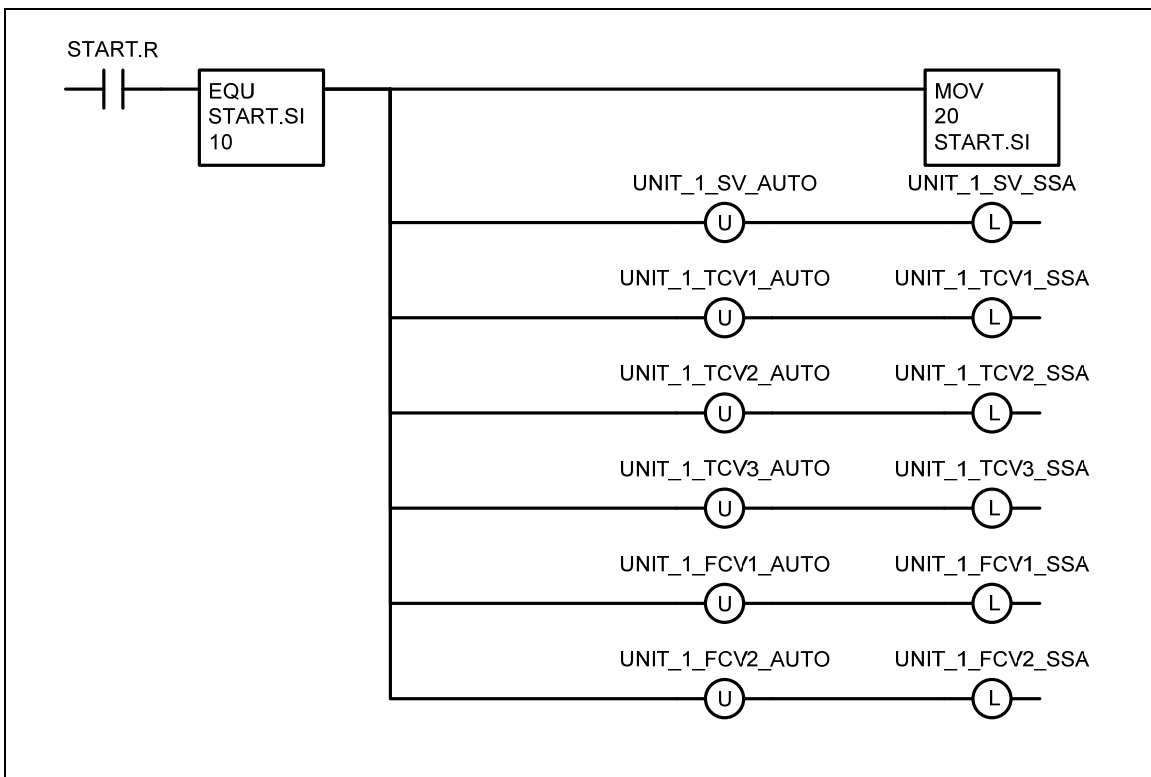


Figure 5-15 Start Phase Logic

End

The End phase will close all valves and check any conditions that are necessary to end the batch. The end phase would be used at the end of the recipe to do any housekeeping that is necessary to put the mixer in a resting state. It could also be used to increment a batch count.

Figure 5-16 shows the auto bit being reset for all of the valves.

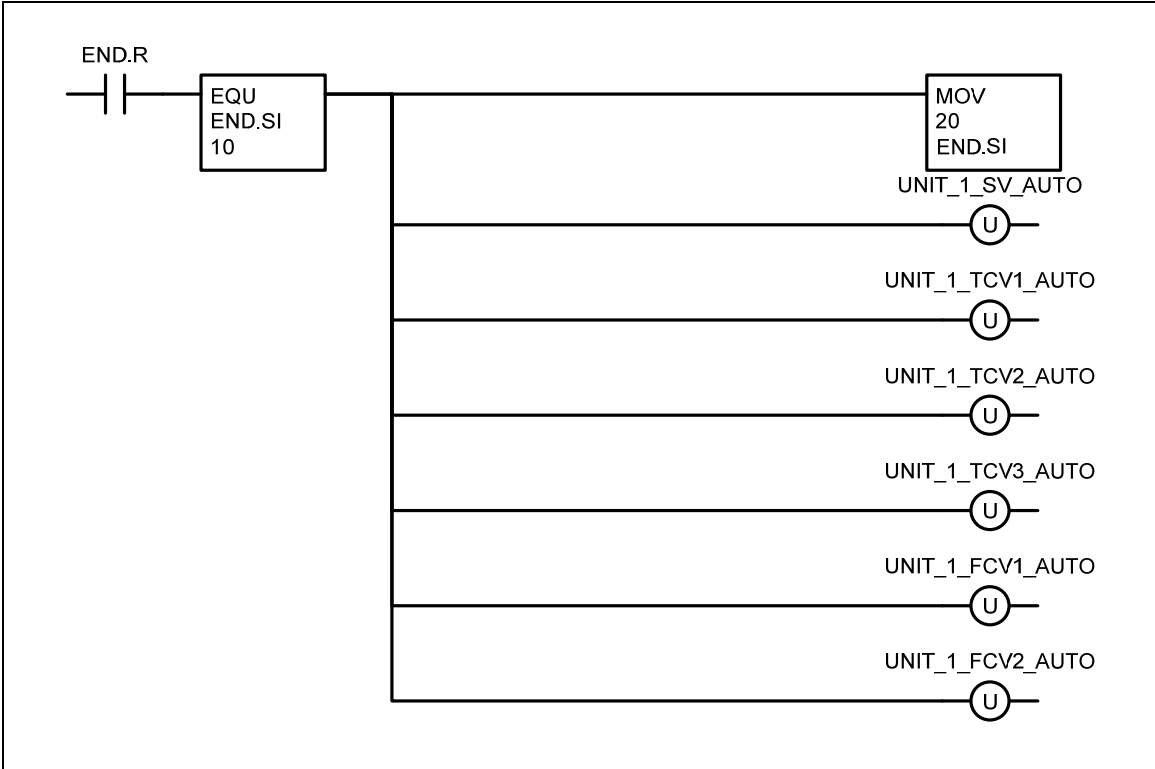


Figure 5-16 End Phase Logic

Feed Material 1

The Feed Material 1 phase will open the feed valve for material 1 and feed in the amount specified in parameter 1. A cut-off will be calculated based upon the in-flight. When the cut-off is reached, the valve will close and the tolerance timer is started. When the tolerance time is reached the weight is checked to see if it is in tolerance. If the weight is within tolerance the phase status is set to complete. If the weight is under tolerance then an alarm will occur and the operator must manually feed additional material into the tank or press a continue button that will allow the phase to complete. If the weight is over tolerance then an alarm occurs and the operator must hit a continue button to allow the phase to complete.

We check to make sure that the feed amount is not zero and then we reset the flow total FT1_TOTAL. The feed valve is opened and the step is incremented. If a recipe exists that does not require this feed, then the feed amount will be set to zero. The feed phase could be eliminated from the recipe also, but this would mean that we would have to keep track of an additional recipe. The feed amount is stored in the recipe, in parameter 1. The batch controller moves this parameter into a local phase parameter. In this case FEED1.P1. The phase could act directly on the recipe parameter without moving it into a local phase variable. However, there are several reasons why the recipe parameter should be moved to a local phase variable. By doing this we make it easier to examine the amount from a computer terminal. Also, if the phase is executed externally outside the control of the batch controller then the amount can be entered locally without changing the recipe. If the phase fails to complete because of insufficient material the operator has the option of changing the feed amount in order to allow the phase to complete. For these reason it is always recommended that any recipe parameters be moved to local phase parameters.

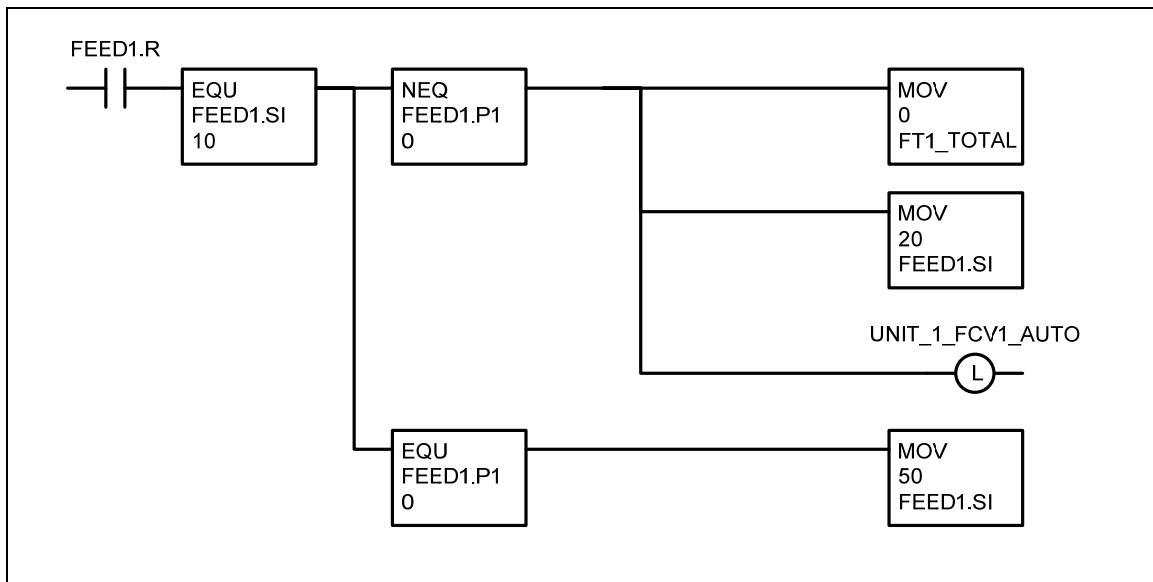


Figure 5-17 Feed Material 1 Phase Open the Feed Valve

In this example the flow total, FT1_TOTAL, is incremented when valve FCV1 is opened.

In Figure 5-18 the feed cutoff is calculated based upon the inflight. The inflight is the amount of material that will flow through the totalizer before the valve completely closes. If the mixer weight is used to shut off the feed valve then the inflight would be the amount of material that will drain into the mixer after the valve is closed. This value is usually measured during commissioning. If the same material is always fed through the same line at the same rate, then the inflight may not have to be adjusted. However, if the inflight can change based upon the process then some method to change this value should be considered. One method of adjusting the inflight would be to automatically increment or decrement this value if a tolerance alarm occurs. If the inflight was based mostly on the material being fed then it could be put into the recipe as a parameter.

The sequence will wait for the total amount fed to reach the cutoff amount. The valve is then closed and the feed phase step index is incremented. This is the normal operation. Now let's consider what happens if there is insufficient material to complete the feed. The operator can change the amount, FEED1.P1, in order to make the phase continue to the next step. We could also have programmed a bypass switch around the greater than or equal to instruction (GEQ). Or we could allow the operator to change the step index directly from a computer terminal.

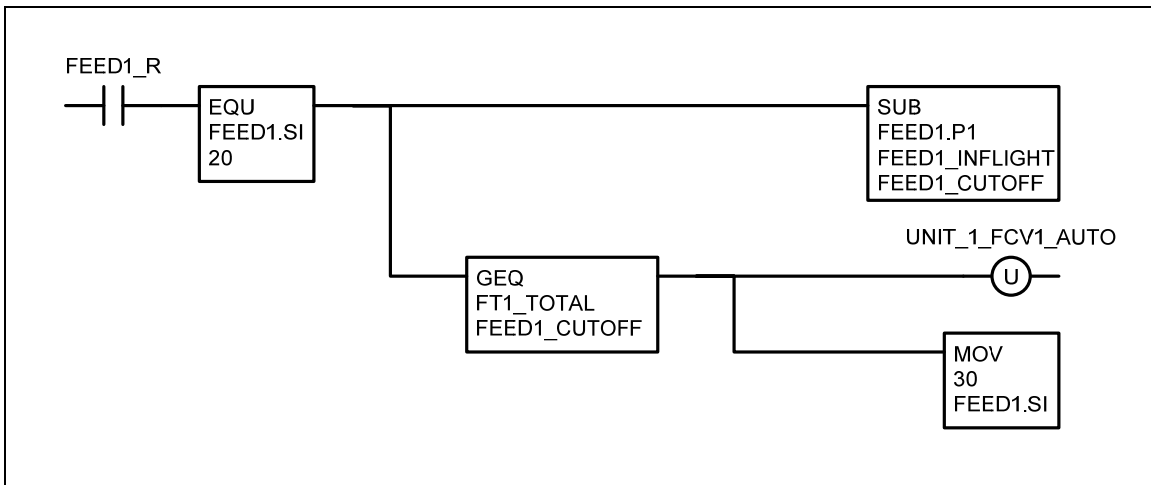


Figure 5-18 Feed Material 1 Phase Open the Feed Valve

In Figure 5-19 the amount fed is checked to see if it is within tolerance. If the feed total is out of tolerance then an alarm is sounded and the operator must press a push button to cause the sequence to continue. The tolerance is in the same units as the feed amount such as pounds or gallons. The tolerance could also be expressed in percentage. In this case the low point and high point would be calculated from this percentage. The tolerance could be a recipe parameter or a local phase variable. The feed tolerance timer has two functions it allows the valve to close and the totalizer to settle. It also allows the pipe to drain into the mixer after the valve is closed before continuing to the next step.

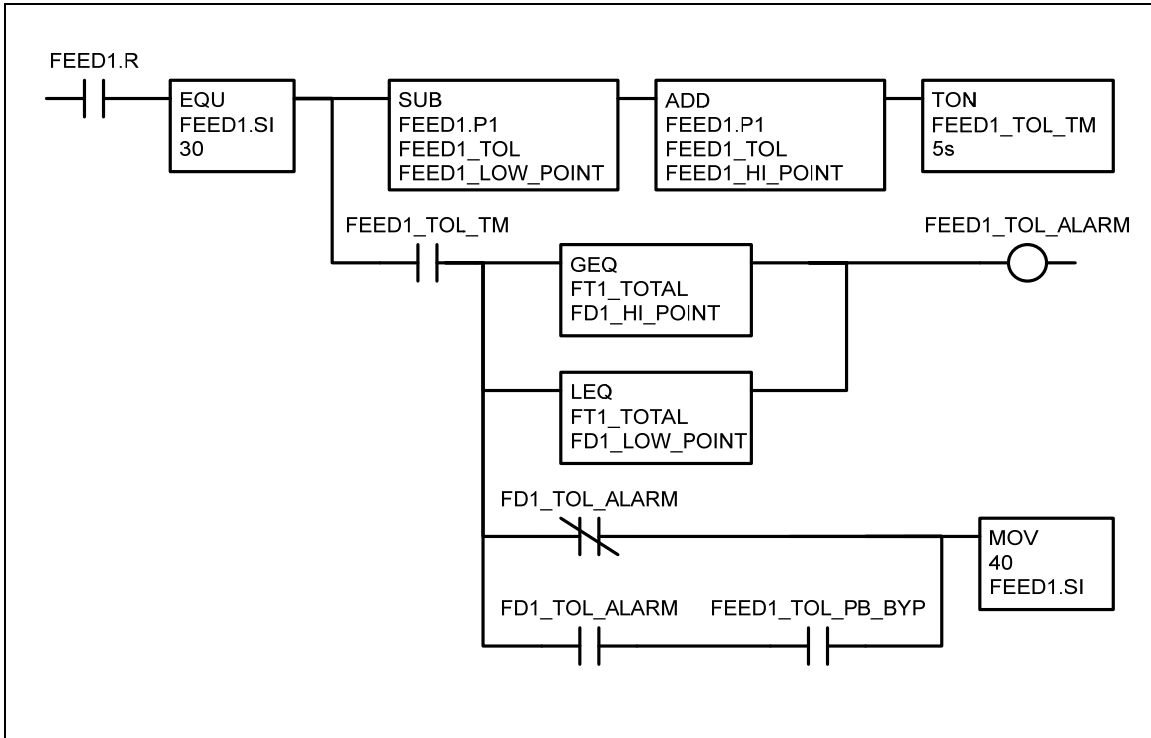


Figure 5-19 Feed Material 1 Phase Check for Tolerance

Feed Material 2

The Feed Material 2 phase will work the same as the Feed Material 1 phase. Your process will dictate the number and types of feeds.

Tare Scale

The Tare Scale phase will issue a tare command to the scale. The phase will complete when a valid tare is achieved.

In Figure 5-20 we set the tare output on for 2 seconds and then increment the step.

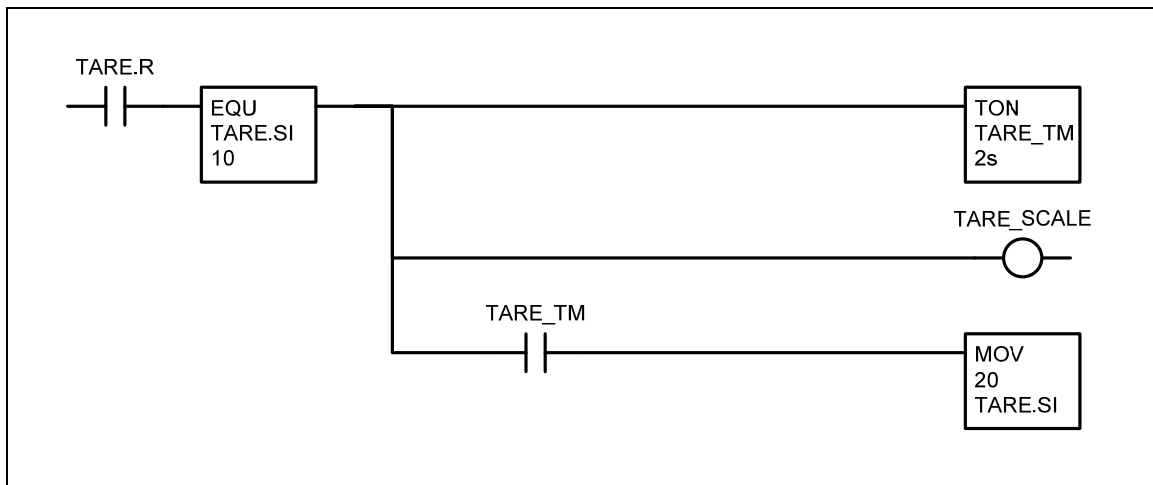


Figure 5-20 Tare Scale Phase Logic

Start Heat

The Start Heat phase will enable the jacket heat controller with the setpoint in parameter 1. The temperature control valve is an on off valve. The controller will include a deadband. The End Heat phase is used to disable the controller. The phase complete status will be set immediately. The If Process variable phase is used to check the temperature in the process if necessary.

The start heat phase is started and ended the same as the previously described phases. In Figure 5-21 the setpoint of the start heat phase is passed to the heat controller and the heat controller is turned on.

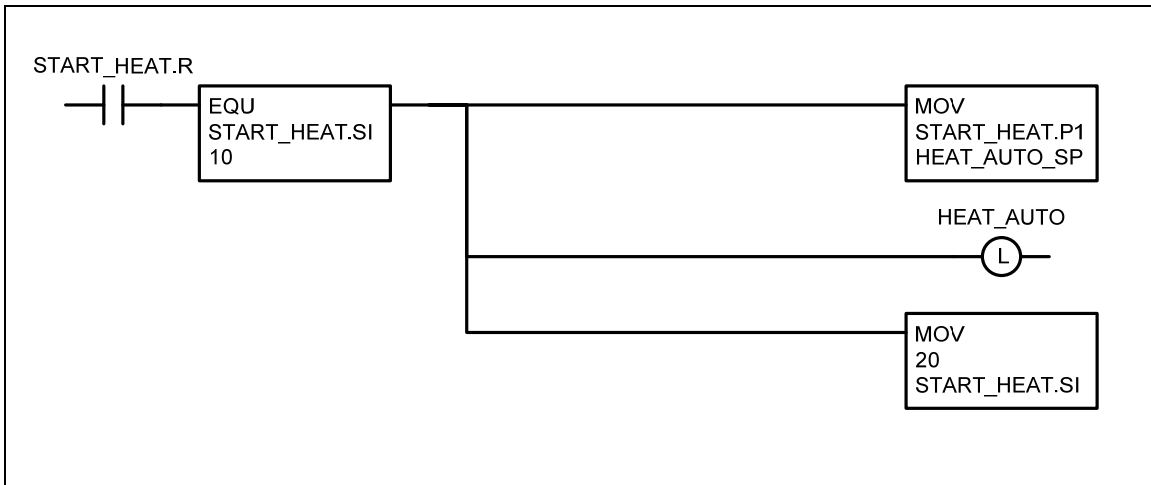


Figure 5-21 Start Heat Phase Logic

End Heat

The End Heat phase will disable the jacket heat controller.

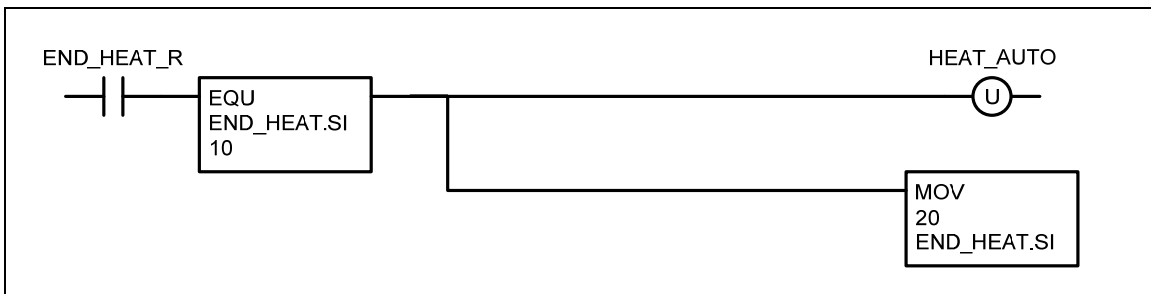


Figure 5-22 End Heat Phase Logic

Start Cooling

The Start Cooling phase will enable the jacket cooling controller with the setpoint in parameter 1. This phase will work the same but opposite of the Start Heat phase.

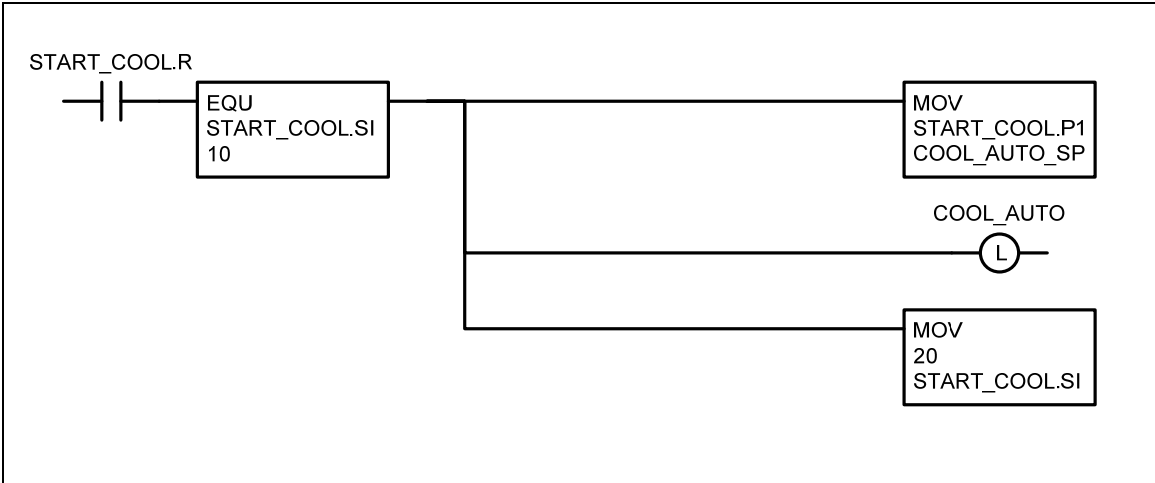


Figure 5-23 Start Cooling Phase Logic

End Cooling

The End Cooling phase will disable the jacket cooling controller.

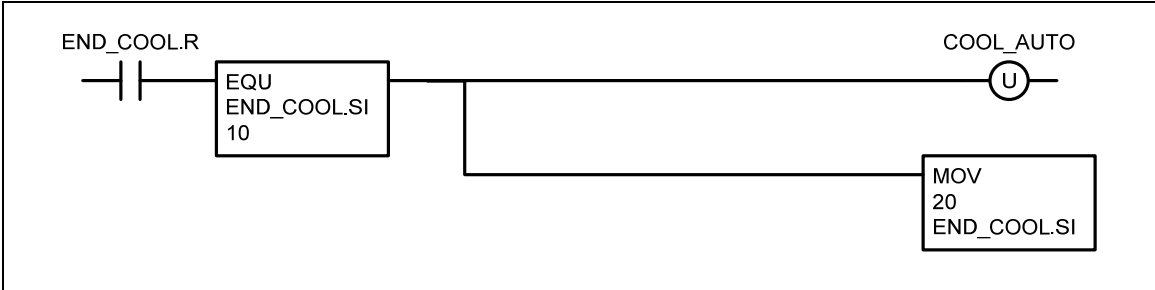


Figure 5-24 End Cooling Phase Logic

Start Agitator

The Start Agitator phase will start the agitator with a setpoint in RPM specified by parameter 1. The phase complete status will be set when the Agitator reaches the setpoint. A tolerance value is subtracted from the agitator setpoint to account for error in the feedback RPM of the agitator.

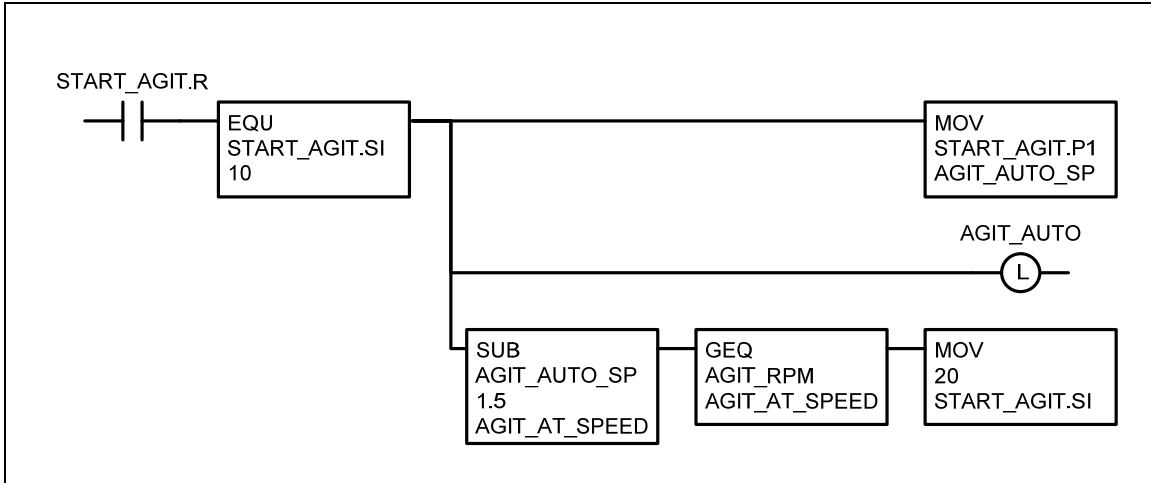


Figure 5-25 Start Agitator Phase Logic

Stop Agitator

The Stop Agitator phase will, you guessed it, stop the agitator.

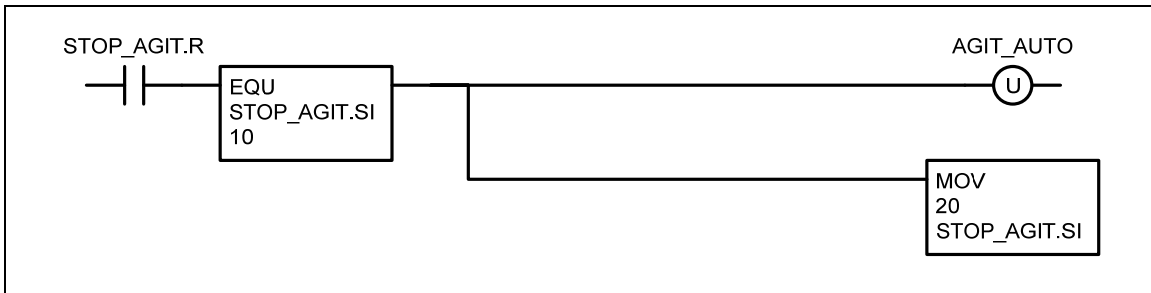


Figure 5-26 Stop Agitator Phase Logic

If Process Variable

The If Process Variable phase is a transitional phase. The batch sequence will stop at this phase until the phase is complete. The first parameter PV Pointer, is a number that determines which process variable will be tested. Table 5-5 shows the PV Pointer table.

Pointer	Process Variable
1	Temperature
2	Scale Weight
3	Agitator RPM
4	Process Timer

Table 5-5 If Phase PV Pointer Table

Parameter 2 is the value the process variable will be tested against. Parameter 3 is a number which represents which operator will be used to test the process variable. Table 5-6 lists the value of the operator.

Operator	
1	Equal to
2	Less Than
3	Less Than or Equal to
4	Greater Than
5	Greater Than or Equal to

Table 5-6 If Pv Comparison Operator Table

Parameter 1, IF_PV.P1, determines which process variable is tested.

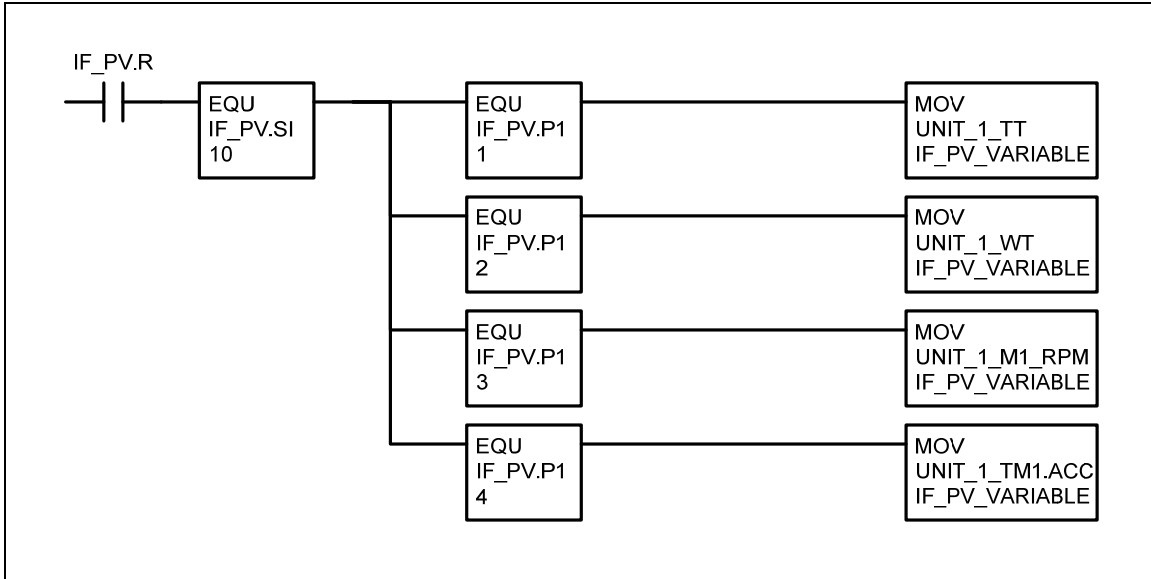


Figure 5-27 If PV Phase Set Process Variable for Comparison

Parameter 2, IF_PV.P2, determines what test is performed. Parameter 3, IF_PV.P3, is the value that the process variable is compared to.

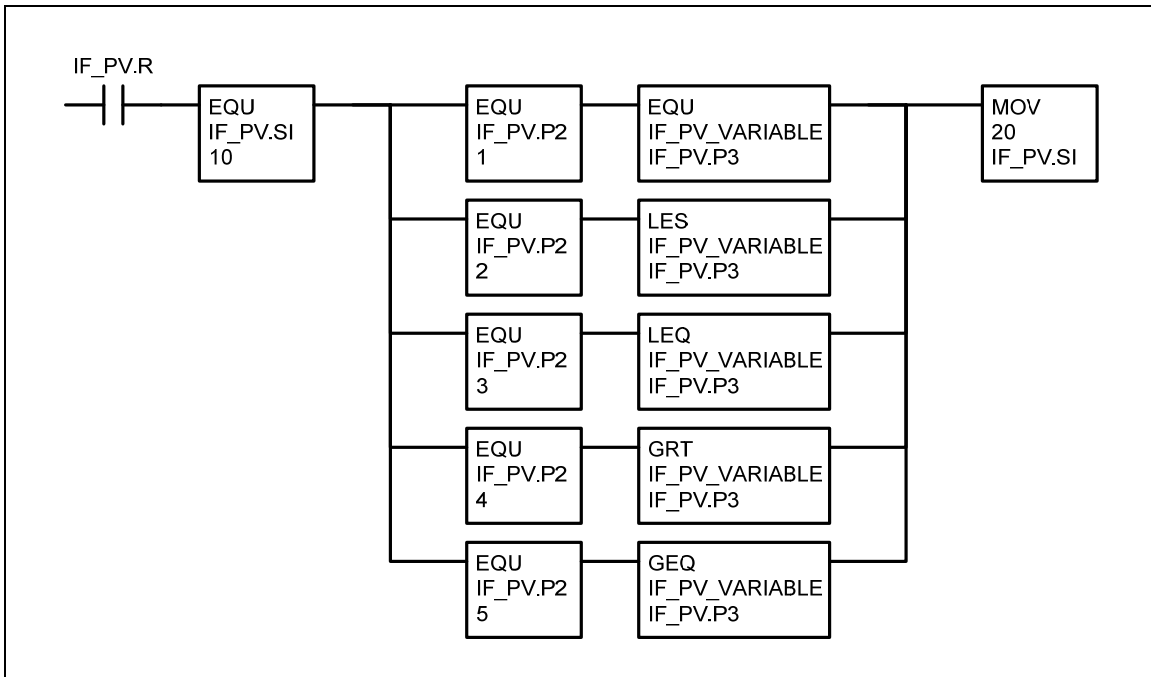


Figure 5-28 If PV Phase Wait for True Compare

Delay

The Start timer phase will reset and then a process timer. This timer can be used to agitate for a certain amount of time, or to hold heat for a preset time. The If Process Variable phase can be used to test the value of the process timer in the recipe. Or, the phase will complete when the process timer is done.

In Figure 5-29 we reset the timer.

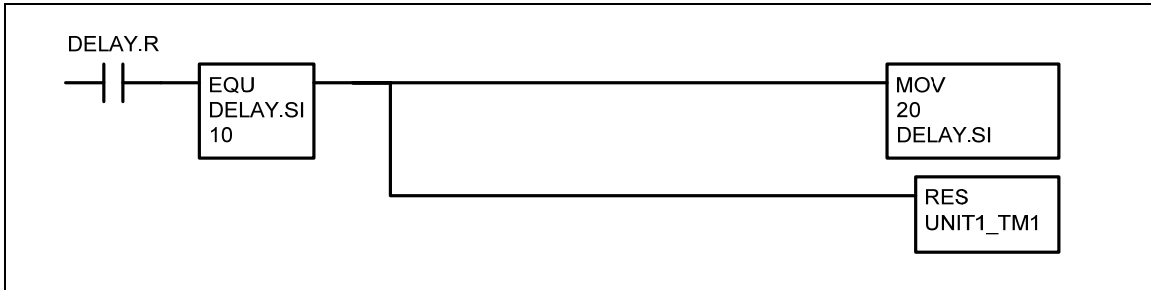


Figure 5-29 Delay Phase Reset the Delay Timer

In Figure 5-30 the retentive timer, RTO, instruction is started. If the sequence is put into hold the timer will stop. When the timer is complete the step index is incremented and the phase will complete. Parameter 1 ,DELAY.P1, is the phase parameter that contains the preset of the timer. This parameter is set from the recipe parameter. The value can be changed by the operator if necessary while the phase is active.

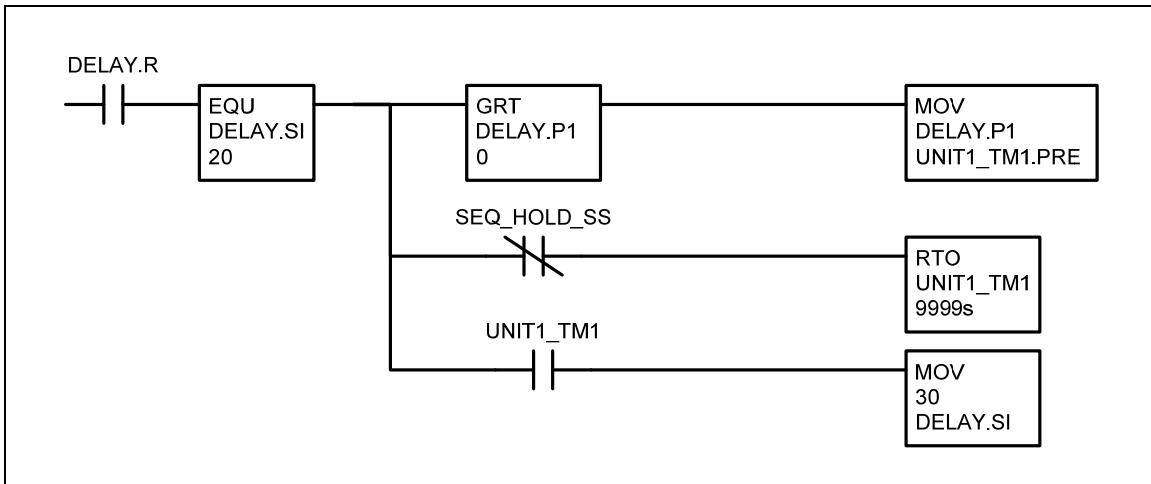


Figure 5-30 Delay Phase Wait for Delay Timer

If Ok Operator Prompt

When the If Ok Operator Prompt phase is active a message is displayed on the batch operator terminal. Parameter 1 of the phase contains the message pointer. The message array size is 20 so we make sure that parameter 1 does not exceed this value before we use it as a pointer. Parameter 2 contains a value that can be displayed along with the message. The operator is required to press an ok button on the screen before the phase will complete. This phase can be used to prompt the operator to do a hand add to the mixer for example. The If OK phase is a transitional phase. The batch sequence will stop at this phase until the phase is complete.

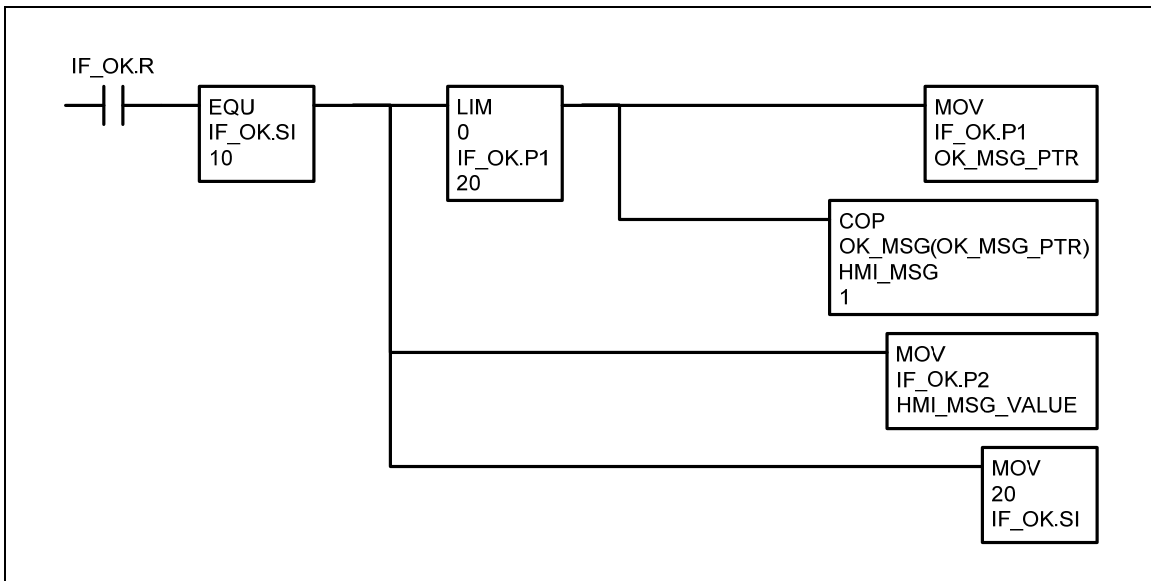


Figure 5-31 If OK Phase Display the OK Message to the Operator

In Table 5-7 we show the strings in the OK_MSG[] array. We used a message pointer instead of the actual message in the recipe parameter to reduce the amount of processor memory used for storing messages.

OK_MSG[1]	Add syrup lbs
OK_MSG[2]	Add butter lbs
OK_MSG[3]	Add sugar lbs
OK_MSG[4]	Add cocoa lbs

Table 5-7 OK Phase Operator Messages

In Figure 5-32 the one shot of the ok push button causes the step index to be incremented. The one shot is necessary if there are consecutive ok messages that the operator must answer in the recipe. The one shot forces the operator to release and then press the push button again for each message. If the phase is running, i.e. IF_OK.R is on. Then the ok message is displayed along with the OK push button. When the phase is complete the OK push button is made invisible indicating that the question has been answered. The message can then remain visible letting the operator see the last message that was displayed.

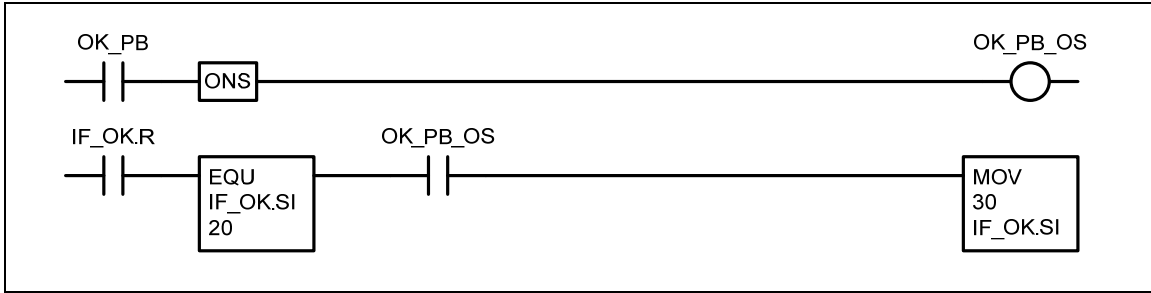


Figure 5-32 If OK Phase Wait for the Ok Button From the Operator

If Yes Operator Prompt

The if Yes Operator Prompt phase displays a message to the operator specified by parameter 1 of the recipe. Parameter 2 contains a floating point value that is displayed along with the message. A Yes push button and a No push button are displayed to the operator. If the operator presses the Yes push button, the sequence continues to the next step. If the operator presses no, then the sequence step is changed to the If Else step which is set by parameter 3 of the recipe. The If Yes phase is a transitional phase. The batch sequence will stop at this phase until the phase is complete or the sequence step has been changed by the phase.

YES_MSG[1]	Take a sample (yes, no)
YES_MSG[2]	Sample ok (yes, no)

Table 5-8 If Yes Phase Messages

In Figure 5-33 the message is displayed along with a value in parameter 2.

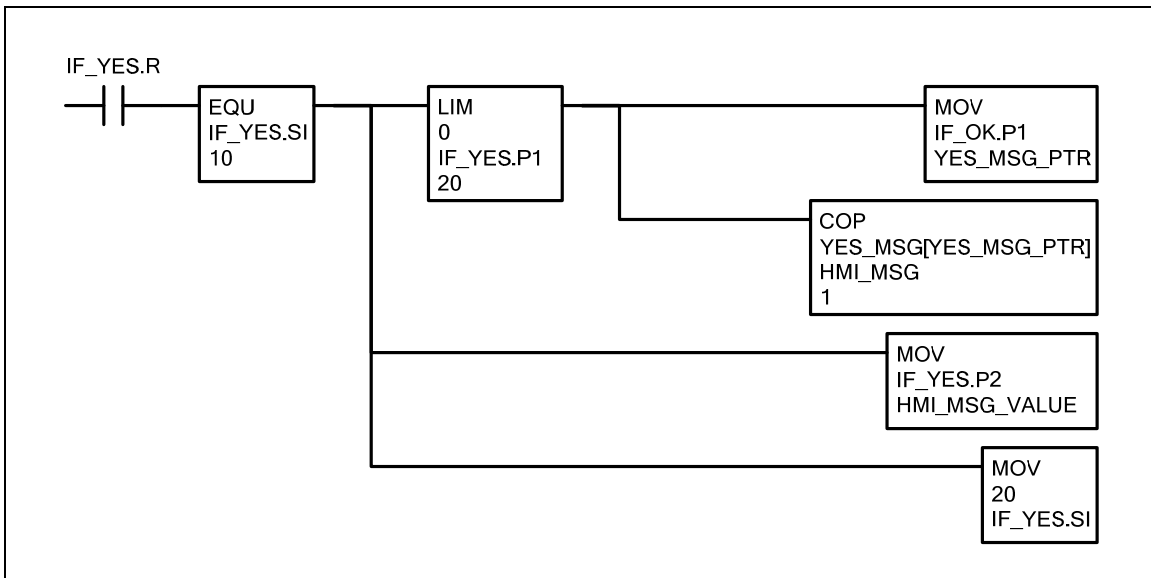


Figure 5-33 If Yes Phase Display the Message to the Operator

In Figure 5-34 we check to see which push button the operator has chosen.

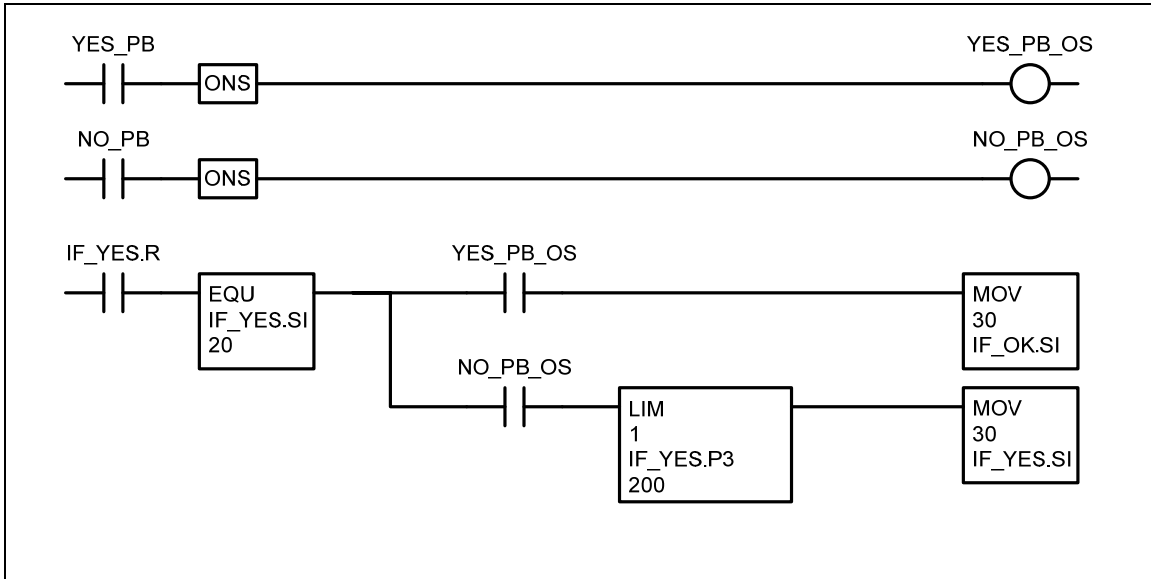


Figure 5-34 If Yes Phase Wait for the Operator Response

In Figure 5-35 the operator has chosen no. We then increment a pointer to find the matching step ID in the recipe. The pointer is incremented by 1 each program scan until the step ID is found.

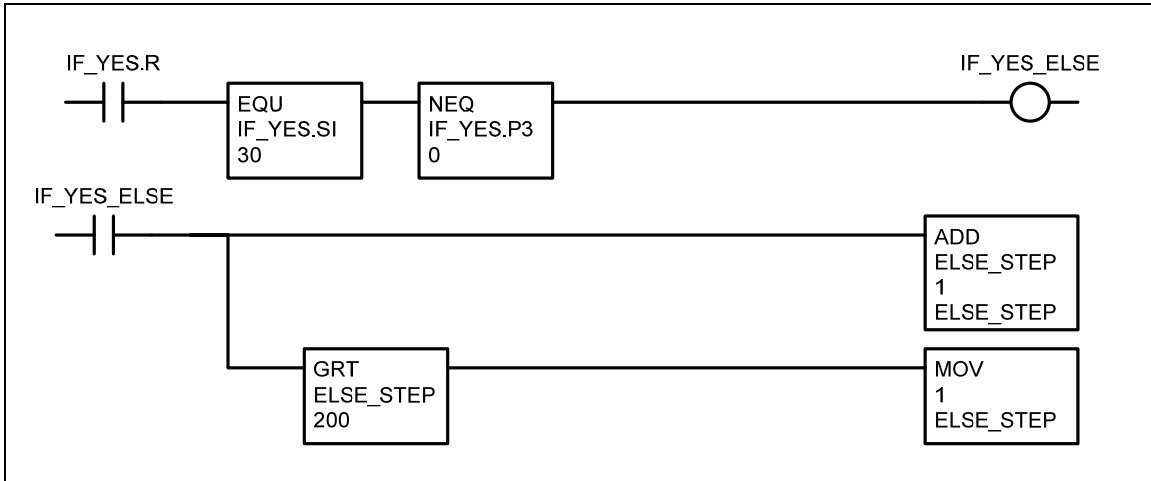


Figure 5-35 If Yes Phase When No is Selected Set the Else Step for the Batch

In Figure 5-36 we compare the step ID pointed to by ELSE_STEP. When we find the correct step, the sequence step number is changed, the phase running bit is reset and the phase complete bit is set.

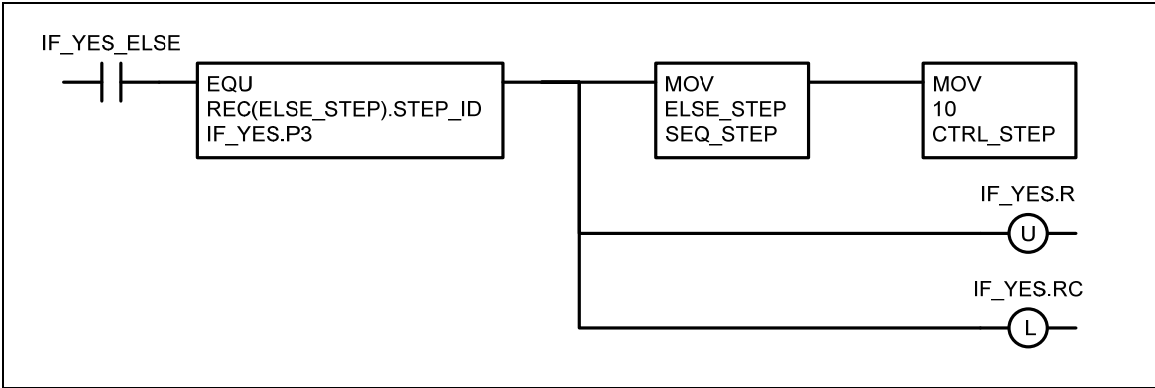


Figure 5-36 If Yes Phase Set the Batch Controller to the Else Step

Wait for Phase Complete

The Wait for Phase Complete phase monitors the complete bit for a phase within recipe. Parameter 1 is the step ID of the phase that is monitored. This phase is a transitional phase. The batch sequence will stop at this phase until the phase is complete

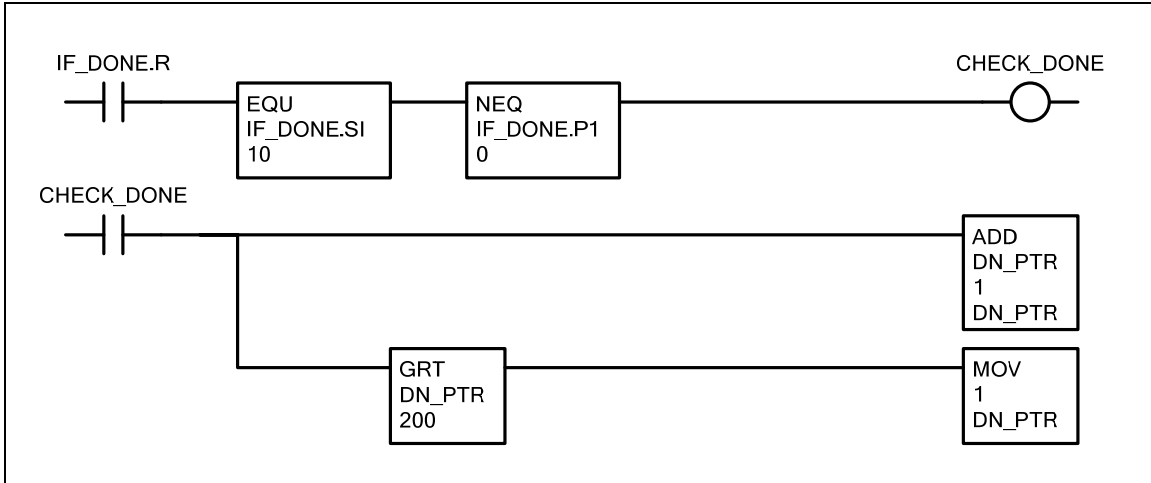


Figure 5-37 If Done Phase Increment the Phase Done Pointer

In Figure 5-38 the recipe STEP_ID is checked to see if it matches parameter 1. The phase number for that step in the recipe is moved to PH_PTR. The complete status is then checked for that phase and the step index is incremented allowing the phase to complete and the recipe to index to the next phase.

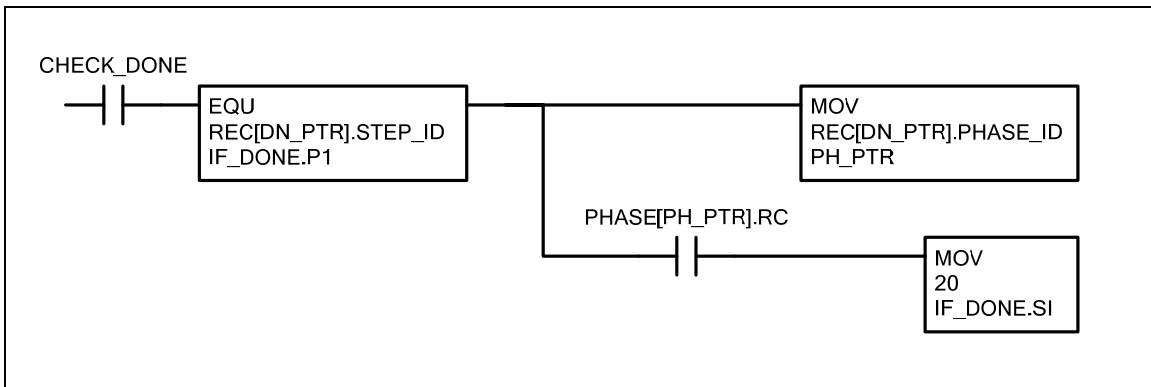


Figure 5-38 If Done Phase Check if the Phase is Complete

Go to Step

The Go to Step phase causes the batch sequence to jump to the recipe step with a step ID that matches parameter 1.

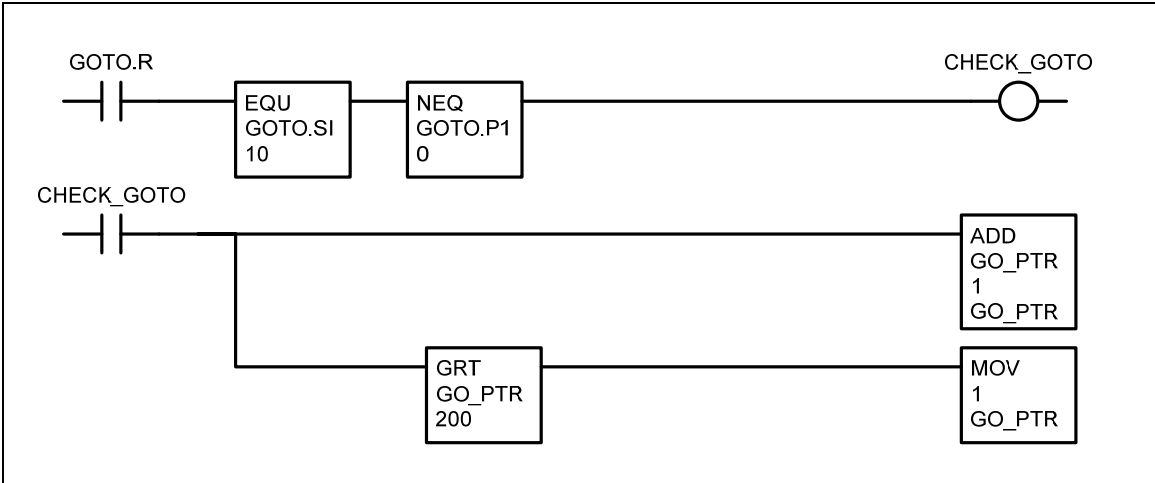


Figure 5-39 Go to Step Phase Increment the Go To Pointer

After the step ID is found in the recipe the sequence step is changed.

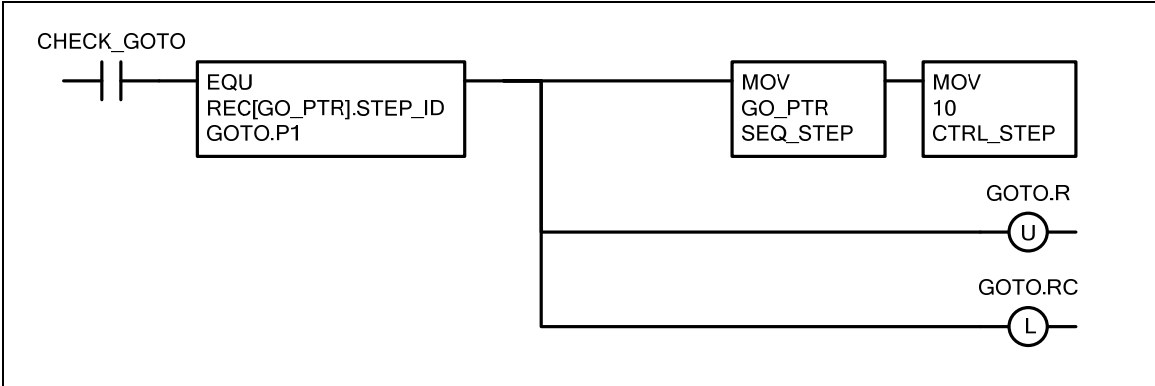


Figure 5-40 Go to Step Phase When the Phase is Found Set the Sequence Step

Creating a Recipe

In Table 5-9 we put a simple recipe together using the phases that we have defined. The step id is used as a label where other phases can refer to the step. The STEP_ID is incremented by 10 for each step of the phase. This allows us to insert a step without renumbering all of the parameters that refer to steps in the recipe. The IF_DONE phase is used to wait for the completion of some phases. Other phases are assumed to be complete. We could however test for completion of each phase if there is some doubt. Also note that FEED1 and FEED2 will both be running while the operator is dumping two hand-add ingredients with the IF_OK phase.

<i>Step</i>	<i>Step_ID</i>	<i>Phase</i>	<i>P1</i>	<i>P2</i>	<i>P3</i>	<i>Comment</i>
1	10	START				Close all valves
2	20	TARE				Tare the scale
3	30	FEED1	500			Start 500lb of feed 1
4	40	FEED2	300			Start 300lb of feed 2
5	50	IF_OK	1	120.5		Msg 1, Add syrup lbs, 120.5
6	60	IF_OK	2	4.0		Msg 2, Add butter lbs, 4.0
7	70	START_AGIT	50			Agitate at 50 rpm
8	80	IF_DONE	30			Wait for feed 1, ID 30
9	90	IF_DONE	40			Wait for feed 1, ID 40
10	100	START_HEAT	160			Heat to 160
11	110	IF_PV	1	5	160	Wait for GEQ 160 F°
12	120	STOP_HEAT				
13	130	DELAY	300			Hold heat for 5 minutes
14	140	IF_DONE	130			Wait for delay to be done
15	150	STOP_AGIT				
16	160	START_COOL				
17	170	IF_PV	1	2	95	Wait for LEQ 95 F°
18	180	STOP_COOL				
19	190	END				

Table 5-9 Recipe Table

Chapter 6 Sequential Machine Logic

Sequential Machine logic defines steps or states through which your system progresses. Relay logic can be used to execute those steps.

The Sequence Diagram

Figure 6-1 shows a sequence diagram.

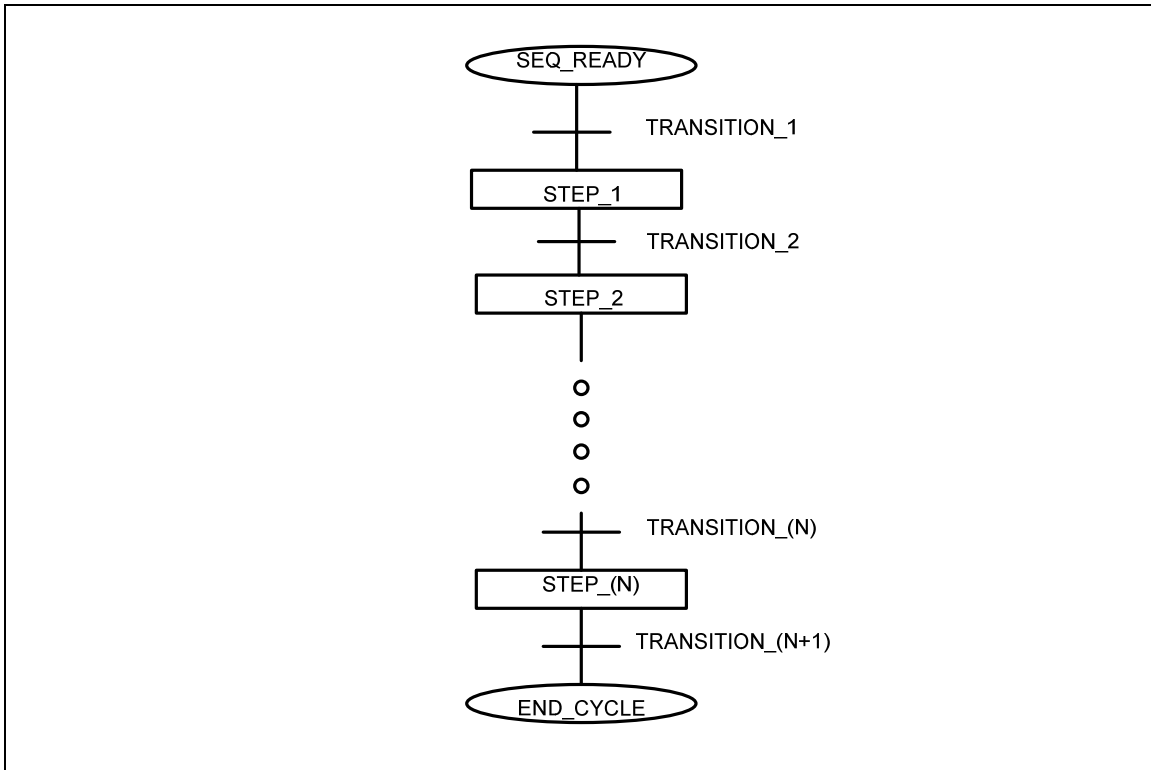


Figure 6-1 The Sequence Diagram

The sequence begins with step 1 and continues until the last step is executed. The sequence can not restart at step 1 until the end of the sequence has been reached and the sequence has been reset.

Let's assume we have a machine that we are programming. If this machine were to operate on one part at a time then there may only be one sequence that would define the operation of that machine. If the machine had several stages to it, and each stage operated on a part, then each stage would have at least one sequence that would define its operation. As the part moves from one stage to the next the last step of the first sequence could be part of the conditions that would initiate the subsequent sequence. The first sequence could then be reset allowing the next part into the first stage.

Programming the steps

Figure 6-2 shows how the SFC is translated into ladder logic.

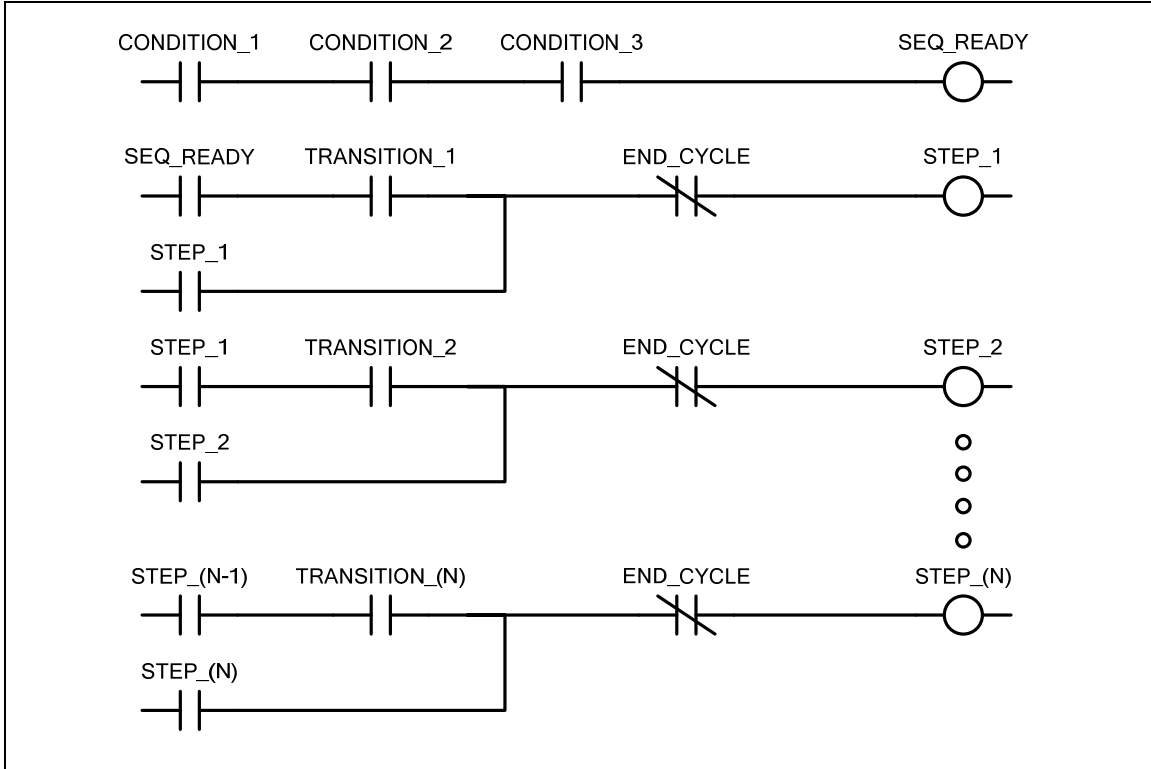


Figure 6-2 Sequential Machine Control Step Logic

Rung 1 represents all of the initial conditions that must be satisfied in order for the sequence to start. The initial conditions should reflect the machine or process state that would allow it to execute the automatic sequence. The SEQ_READY bit is the sum of all of these conditions and is set to the on state when all of the initial conditions are met. Although there are only 3 initial conditions shown in rung 1, more conditions can be added and multiple rungs can be summed into the SEQ_READY bit. Once all of the initial conditions are met, there should be one transition that would normally start the sequence. This could be a start push-button, a limit switch, or a photo-eye for example. The first transition could have been put in with the initial conditions and the function would be logically the same. However, by separating the first transition from the initial conditions a distinction can be made between them. Each step cascades into the next. In Rung 2, if step 1 is on, we are waiting for transition 2 to occur before the sequence proceeds to step 3. The sequence continues until the last step is reached, step (N), where N is the number of steps in the sequence.

A transition can include more than one limit switch or interlock for example. You may be tempted to put multiple transitions in each step that would allow the sequence to continue. Try to put in only those conditions that are necessary to verify that the machine or process is in the correct state in order to move to the next step. The transition should indicate clearly what the

normal condition is that will cause the sequence to progress. Remember that we verified that the machine was in safe and known position when we started the sequence in step 1. And, any anomalies that occur during the execution of the sequence will be checked using fault logic.

The transitions in the examples given will be replaced by the actual limit switch or other device that causes the sequence to progress. No internal bit is defined for the transition.

Each step in the sequence is represented by an internal bit i.e. STEP_1. The step bit will be used later in the program to determine the state of outputs that are to be affected by the sequence.

I like to include the step number in a tag that is used as a step bit. An example would be LFT1_STEP1. Translated this is lift 1 step 1. In this way I can quickly identify what type of bit it is without having to read the associated description. The description would say “Lower clamp and Raise the Lift”. The description then identifies 2 outputs that are affected by the step. The down side to putting the step number in the tag however is that if you need to insert a step into the sequence you will have to renumber all of your steps or add an additional suffix to the step. An example would be LFT1_STEP1A. I like to reserve the “A”, “B” etc. suffixes for branching steps. This means that we will have a conflict in the naming convention. If your sequence has a lot of steps you may want to number your steps by 10. In this way, you can insert steps into your sequence without having to renumber all of your steps.

The ladder logic implementation of the sequence differs from the SFC in one way that is worth noting. In an SFC the step becomes active when the previous transition becomes true. The step remains active until the subsequent transition becomes true. At that time the step becomes inactive and is changed to a complete status. In the ladder logic form, the step remains active until the entire sequence is complete. At which time the END_CYCLE bit is energized and the sequence resets by resetting each step. This is important as you will see later because when we program the outputs we can set an output to come on at one step and go off at another. You might say that we could have done the same thing by latching an output-auto bit that would turn on the output when a step comes on and unlatch it on a subsequent step. This is true, but the disadvantage to this approach is that if you are examining the output rung, you can not determine which step has turned the output on or off.

We could also have latched on the step and then unlatched that step in a subsequent step. This method introduces several problems. The first problem is that a step can be latched and then unlatched before rest of the program sees the step on. This will occur if two consecutive transitions are true. We could overcome this by introducing an additional rung or branch that would delay the unlatch by one scan. Another disadvantage to latching and then unlatching the step is that when examining the sequence it is difficult to zero in on the step that is active. The reason for this is if you examine a rung prior to the current step, the step will be off. If you examine a rung past the current step, the step will also be off. If however, the steps remain on and you examine a rung prior to the current step then you know to scroll down to get to the current step. If you examine a rung past the current step then you know to scroll up to get to the current step.

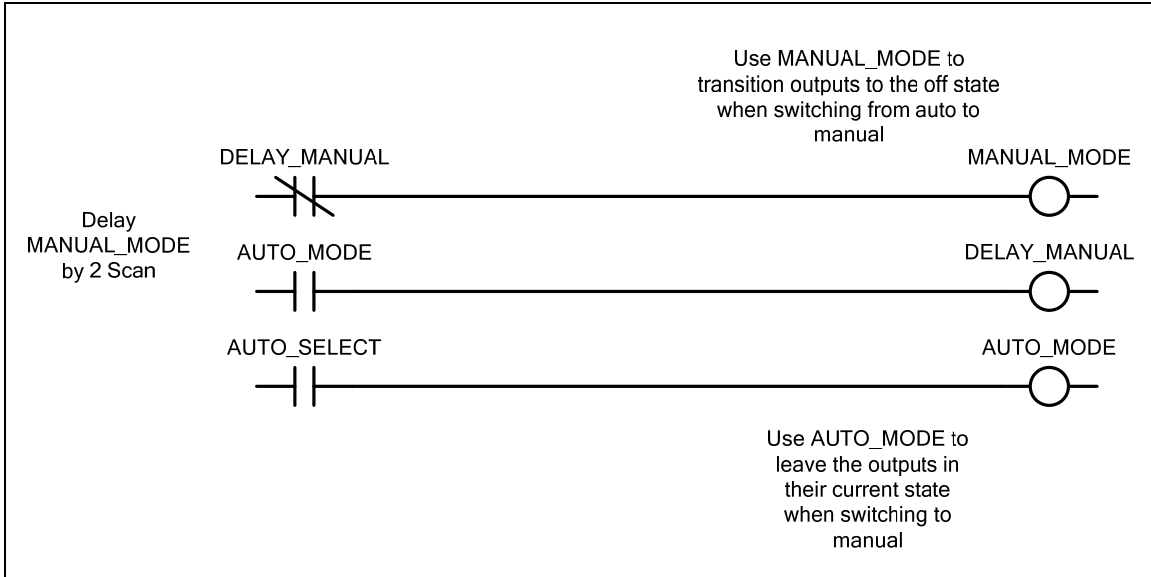


Figure 6-3 Sequential Machine Control Manual Mode

Figure 6-3 shows how the auto mode can be implemented. `AUTO_SELECT` is an input, either from an HMI or a hard-wired selector switch wired to an input on the PLC. If we are using an HMI then we would use `AUTO_SELECT` to turn on the internal bit `AUTO_MODE`. This will synchronize the HMI input with the rest of the program. If we are using a hard-wired switch to an input then we do not have to synchronize the input. Since in most processors the inputs are scanned at the beginning of the program scan, the entire program will see the same state of that input. When an HMI is used to toggle an internal bit in the PLC, this is usually done asynchronous to the program scan. That means that the bit can change states anywhere in the program. By using `AUTO_SELECT` to turn on `AUTO_MODE` all of the program following this rung will see the same state in the same scan. I have shown the auto logic here because it will be used next to end the cycle. However, we could also move these rungs of the auto logic in Figure 6-3 to the beginning of the sequence logic for clarity if we wanted. You must maintain the order of these rungs for the `MANUAL_MODE` to work correctly. We will use the `AUTO_MODE` later to leave outputs in their last state when transitioning from the auto mode to the manual mode. We will use `MANUAL_MODE` to transition outputs to the off state when going from the auto mode to the manual mode. `MANUAL_MODE` will change states 1 program scans after `AUTO_MODE` changes state. This will allow the `END_CYCLE` bit to kill the sequence which in turn will turn all of the outputs off prior to the `AUTO_MODE` bit going to the off state.

Programming the end of cycle

Figure 6-4 shows how the END_CYCLE bit is programmed.

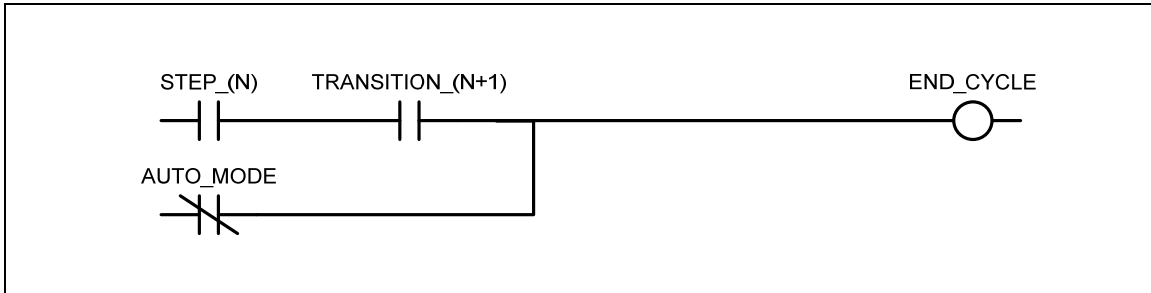


Figure 6-4 Sequential Machine Control End of Cycle

The END_CYCLE bit comes on when the sequence completes or when the machine is put in the manual mode (or not AUTO_MODE). Each step bit remains on until the cycle completes. The END_CYCLE bit then resets all of the steps. It may be that you need to keep the machine in manual without resetting the sequence. If so then exclude the not AUTO_MODE bit. You will still need some way of resetting the sequence if the cycle does not complete normally. A separate cycle reset button could be used.

Programming a timed step

Figure 6-5 shows how a timed step can be implemented.

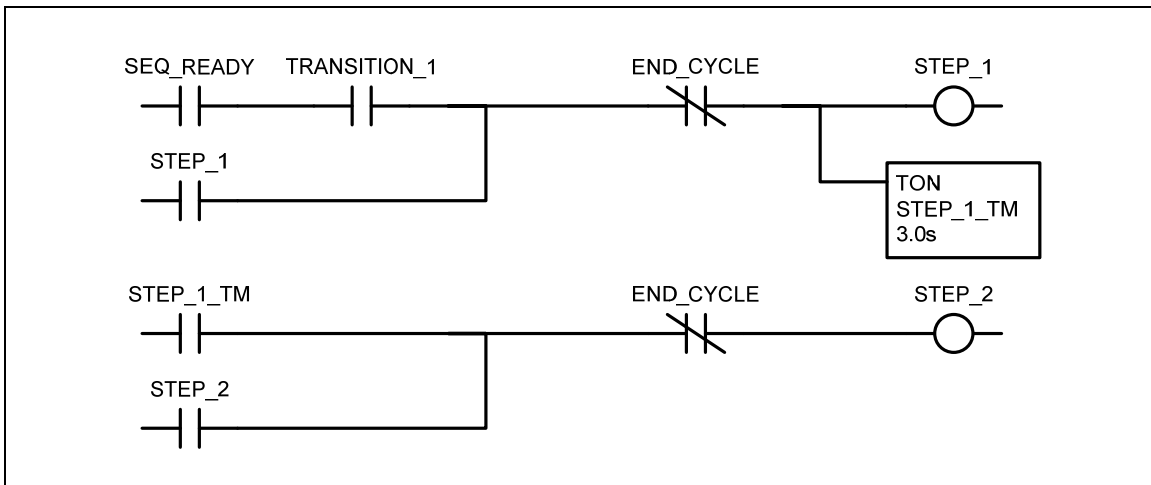


Figure 6-5 Sequential Machine Control Timed Step

Maintaining the step with power loss

Figure 6-6 shows how to implement the step logic in order to maintain the auto sequence when a power loss occurs on the PLC. You would also have to ensure that a power loss to the auto mode will not turn on the end of cycle.

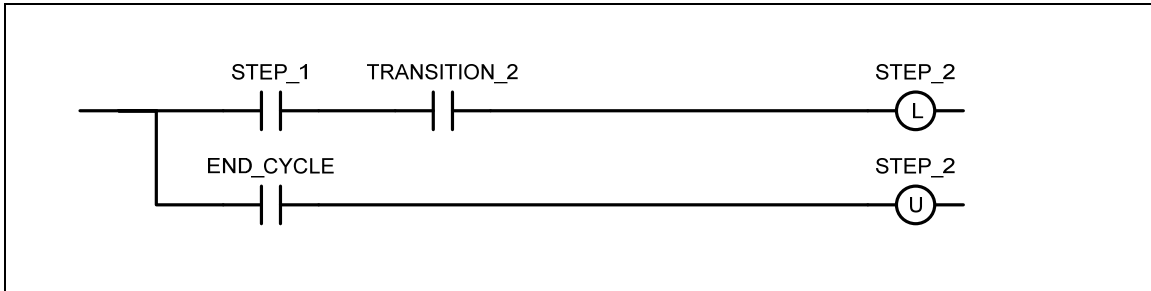


Figure 6-6 Sequential Machine Control Latched Step

Programming the Outputs

Bumpless transfer to manual

Figure 6-7 shows how the outputs are programmed. The step bits will remain on as long as the system remains in auto. We can see in this example the output will come on at step 1 and go off at step 3. Also, the output will come on at step 5 and remain on until the end of cycle. You can see that the output can then be programmed to come on and go off without having to latch up any bits at one step and unlatch them at another. This is one of the main reasons why I like this method over others. You can in one rung see how the sequence affects the output. To achieve a bumpless transfer from auto to manual we use the AUTO_MODE bit described earlier. One of the most important advantages of leaving the step bits on as the sequence progresses is that we can directly program the outputs. By doing this we can immediately see what step is setting the output on or keeping it off. If we had programmed the manual mode such that it does not reset the sequence then we would need to precede the step bits by a normally open AUTO_MODE contact.

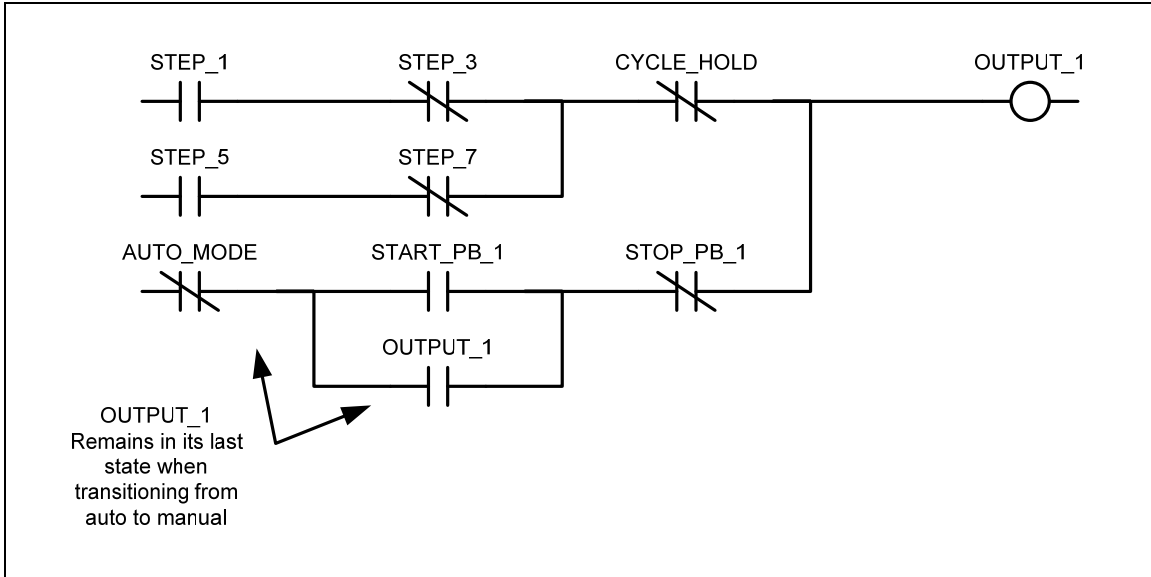


Figure 6-7 Sequential Machine Control Bumpless Transfer Output Logic

Transitioning outputs to the off state

In Figure 6-8 the MANUAL_MODE bit is used to transition the output to the off state when changing modes from auto to manual.

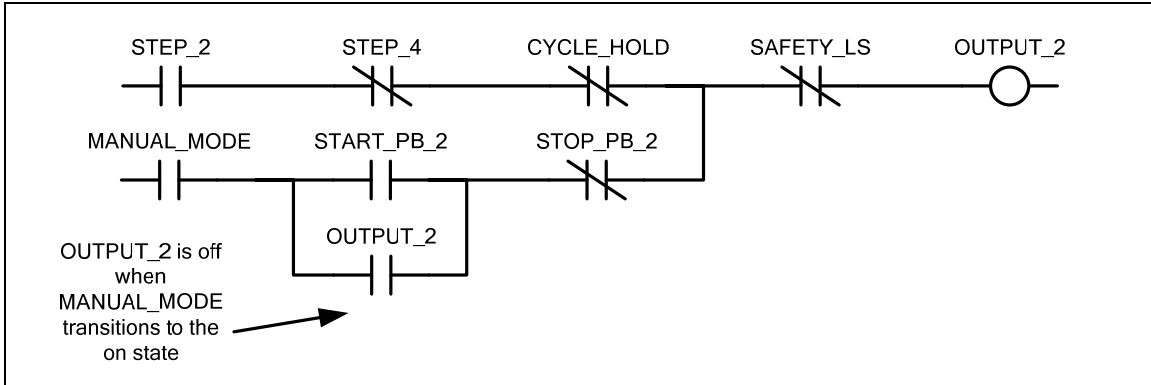


Figure 6-8 Sequential Machine Control Output Logic

Programming faults and cycle hold

Figure 6-9 shows a typical motor fault. The fault will occur if the motor fails to start or if the contact freezes closed when the starter is de-energized. The fault is latched because the fault is used to put the cycle in hold which in turn will shut off the output to the motor starter, MOTOR_1_OUT. I have shown a reset push button to unlatch the fault. The manual mode (not AUTO_MODE) could also be used to reset the fault. When the reset is placed at the bottom of the rung it will override the fault condition even if the fault condition remains true. If you do not want the reset to override the fault condition, then the reset can be moved above the fault logic or a normally closed contact of the MOTOR_1_FLT_TM in series with the reset would prevent the unlatch.

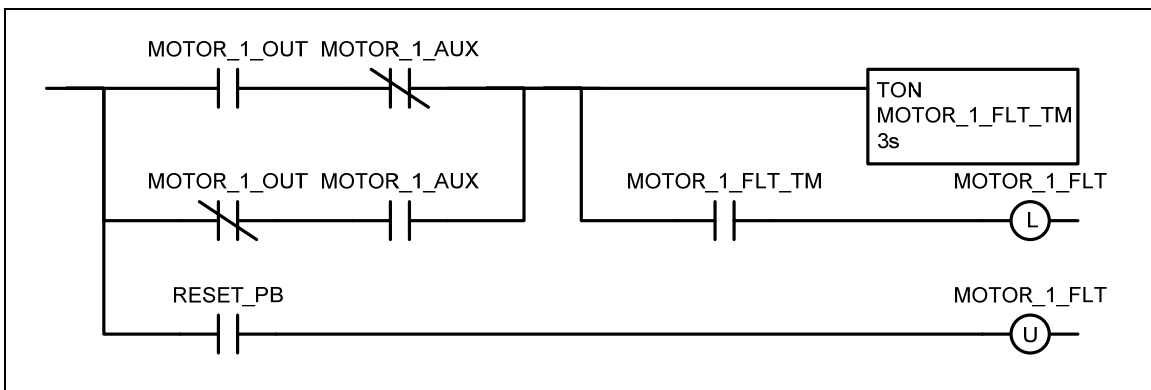


Figure 6-9 Sequential Machine Control Motor Fault

Figure 6-10 shows a typical fault for a single solenoid operated valve with an open and closed limit switch. The valve is normally closed.

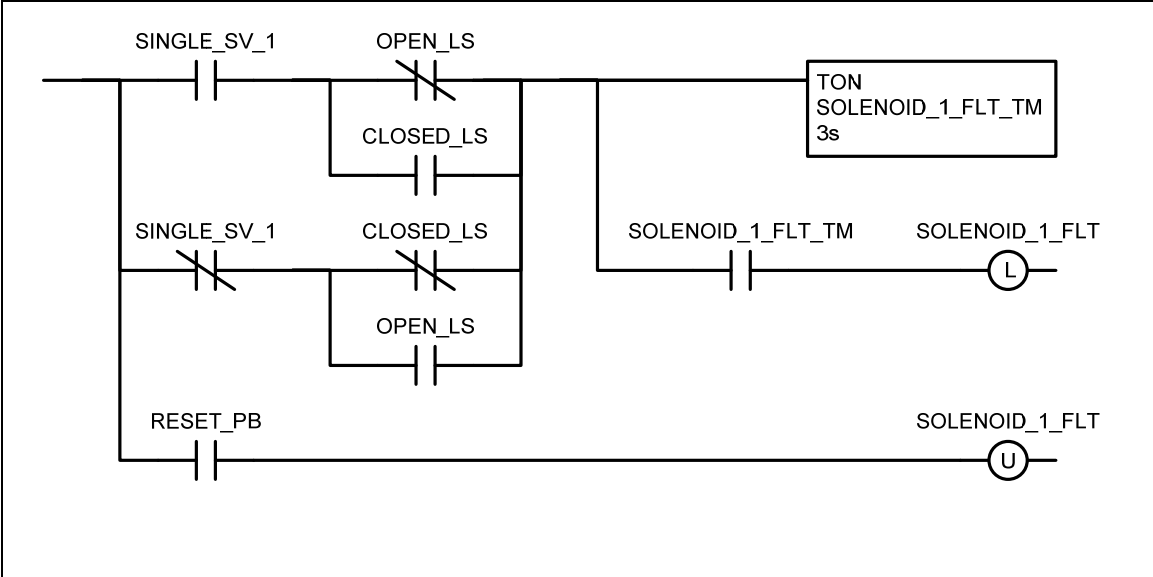


Figure 6-10 Sequential Machine Control Single Solenoid Valve Fault

Figure 6-11 shows how a double solenoid valve fault is programmed. The normal condition of the valve is to drive the valve fully open or closed when one or the other solenoid is energized.

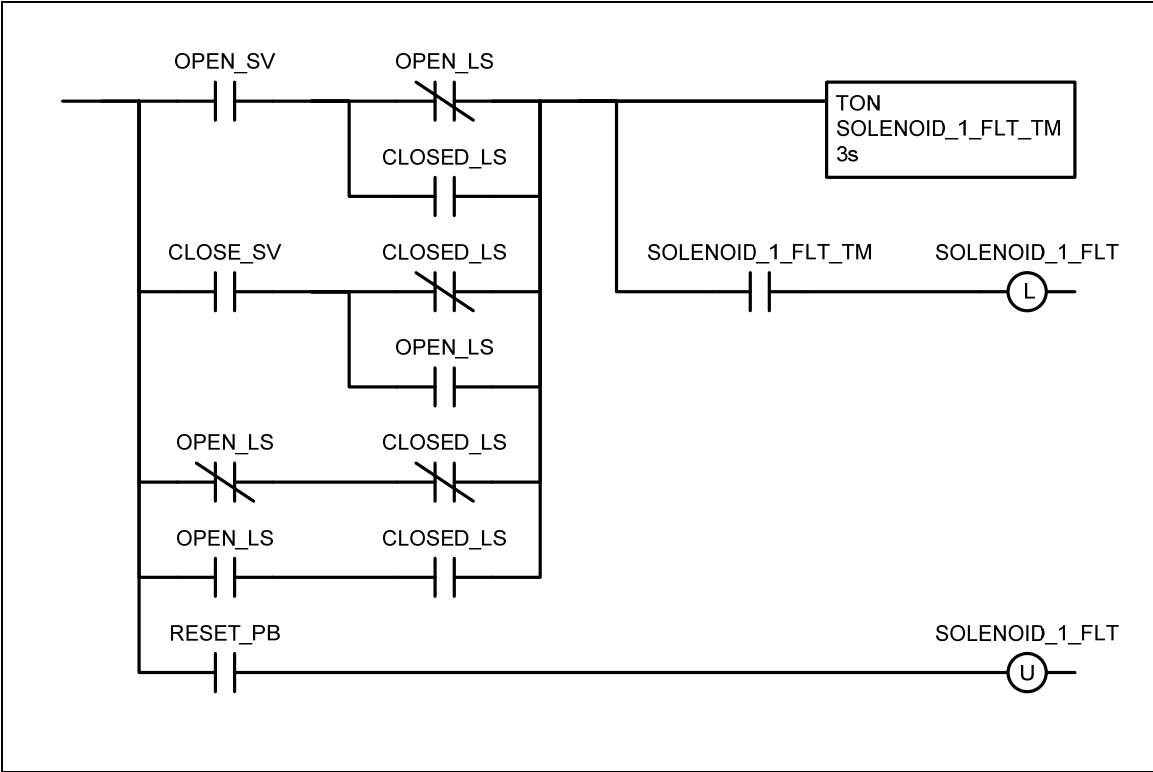


Figure 6-11 Sequential Machine Control Double Solenoid Valve Fault

Figure 6-12 shows a step excess time fault. You would use this type of fault on motions that should occur within a minimum amount of time. If the time is exceeded it could mean that an over-travel has occurred and the machine should be put in hold to avoid a safety problem. The excess time fault could also mean that an operator had not pushed a button within the specified time. In this case the fault should only be annunciated and should not put the cycle in hold. If we don't put the cycle in hold then we would probably use a normal output instead of a latched output so that the fault would reset itself when the next step is achieved.

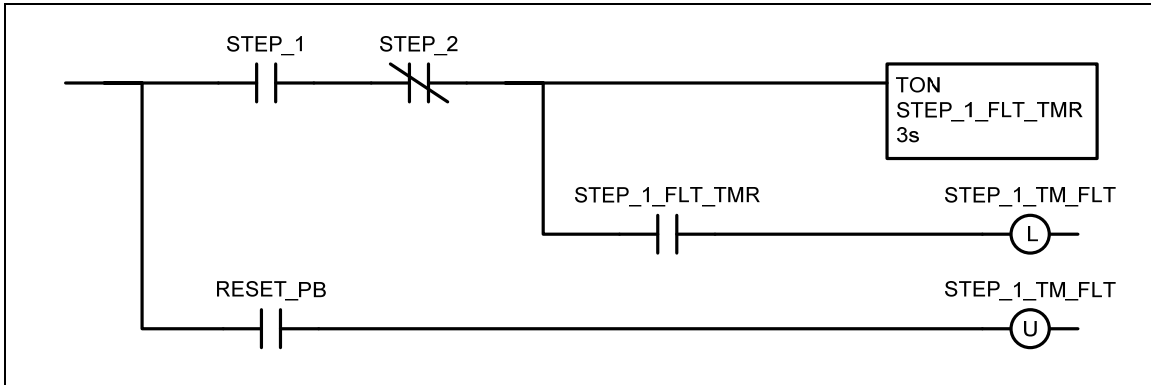


Figure 6-12 Sequential Machine Control Step Excess Time Fault

Figure 6-13 shows a typical sequence fault. This fault indicates that a transition has been missed. It could be used for example on a lift which is supposed to go to slow speed on an intermediate limit switch. If the limit switch is missed then the stop limit switch will cause the fault to occur and put the sequence in hold.

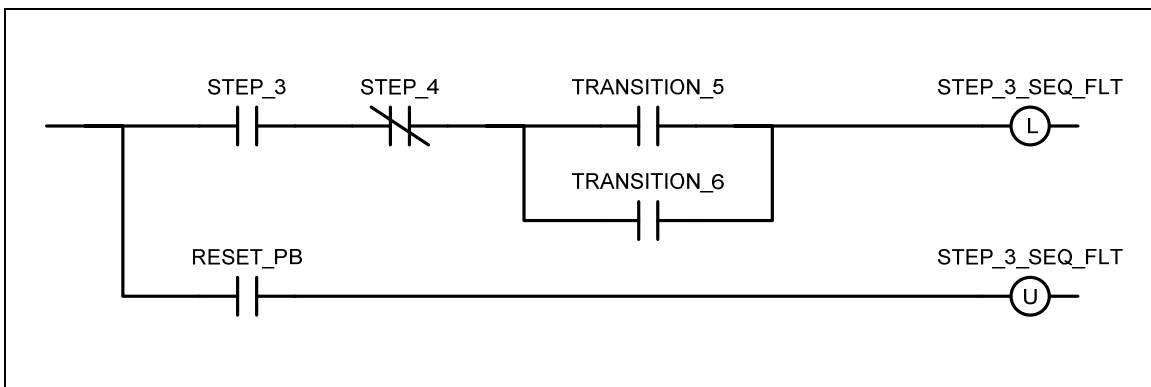


Figure 6-13 Sequential Machine Control Sequence Fault

Figure 6-14 shows how all of the faults can be summed into the CYCLE_HOLD bit. When the cycle is in the hold state all machine motion is stopped.

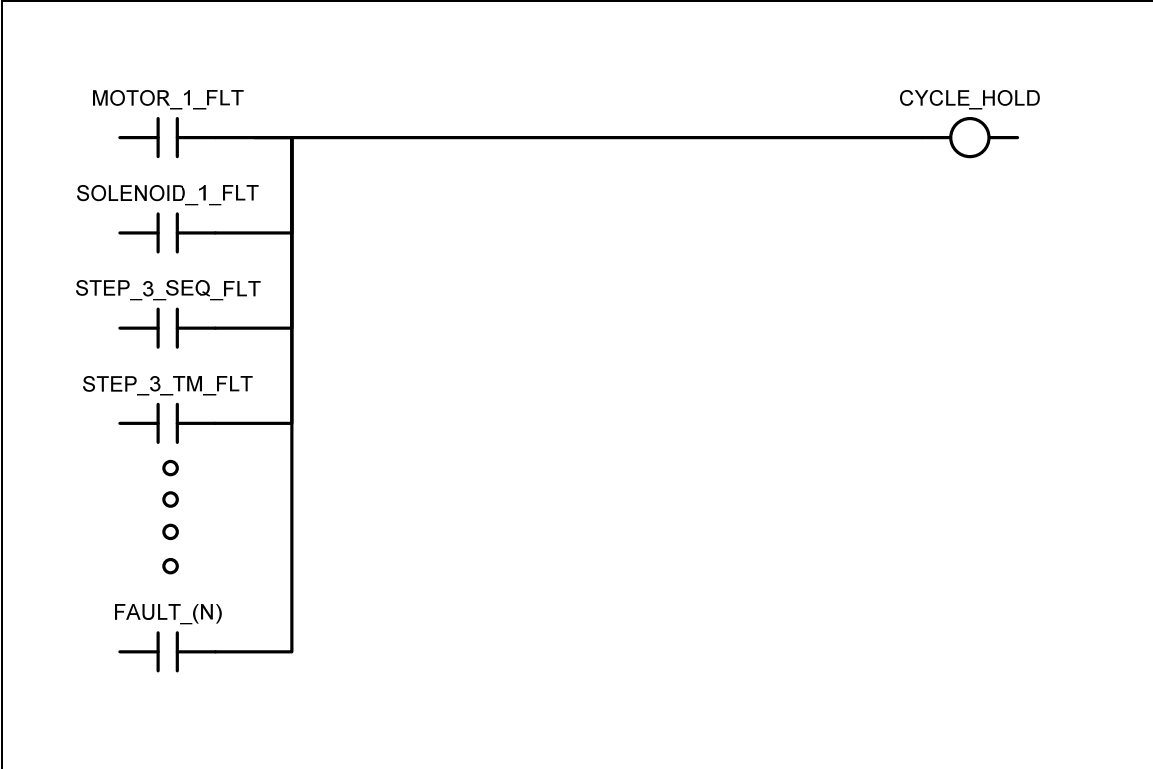


Figure 6-14 Sequential Machine Control Cycle Hold

The CYCLE_HOLD bit could also be used in the step logic to prevent the sequence from moving to the next step. You need to be careful if you do this however, so that you don't miss a transition. If it is possible for your machine to miss the transition then create an additional step whose only transition is (not CYCLE_HOLD). If you are using the (not AUTO_MODE) to reset your faults then the sequence is reset along with the cycle hold state. So, there is no reason to keep the sequence from progressing with the hold mode. Figure 6-15 shows the CYCLE_HOLD bit in the step sequence.

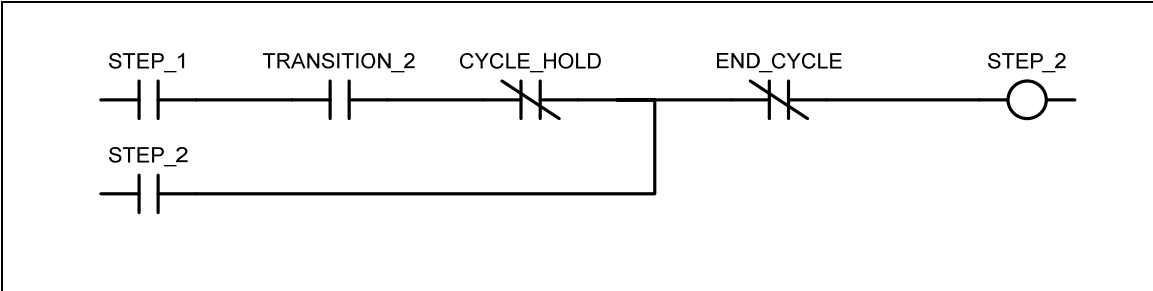


Figure 6-15 Sequential Machine Control Holding a Step with Cycle Hold

Programming sequence paths

Initiating the sequence

There may be some intermediate step(s) in your cycle where you need to start the sequence without going back to step 1. Figure 6-16 shows an SFC where the sequence can be started from step 1 or from step 4. Let's consider a lift that brings a part into the lift, raises it to upper level, discharges the part, and then returns to the lower level. Step 1 would bring the part into the lift. If the sequence was reset with a part on the lift heading up, then the operator would take the cycle out of auto and move the lift to the raised position. The operator would then put the sequence back into auto. The sequence would then pick up at an intermediate step that would send the part out of the lift and then lower the lift.

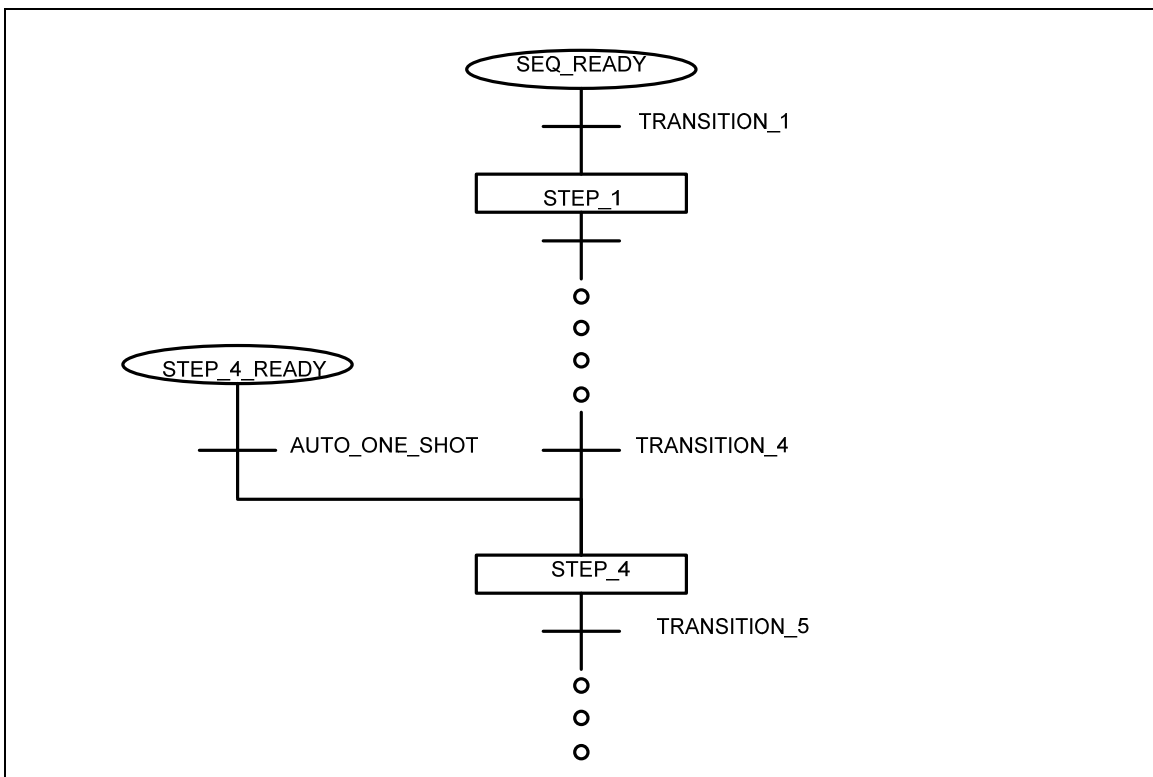


Figure 6-16 Sequential Machine Control mid Cycle Start SFC

Figure 6-17 shows how the SFC is implemented in ladder logic. Note how step 1 and step 4 have interposing interlocks. This keeps step 1 from starting after the sequence has been started from step 4 and visa versa. If there are multiple start positions in the sequence then each step must interlock out the rest of the start positions.

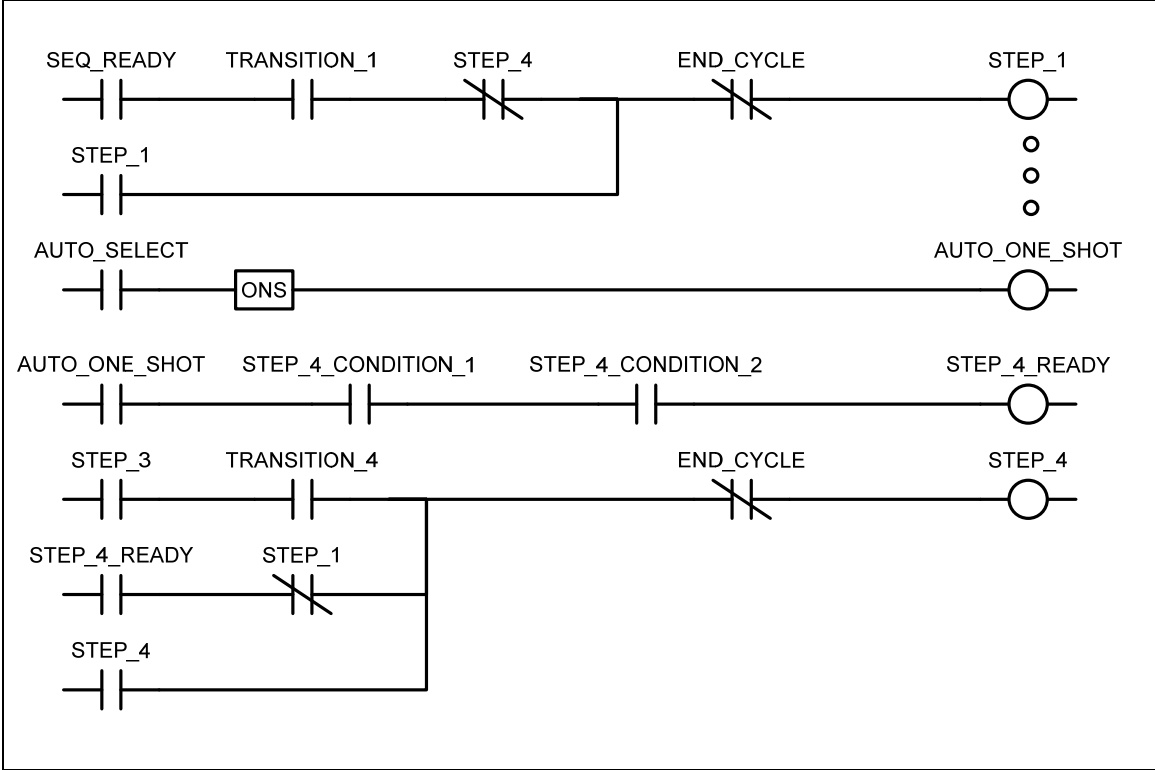


Figure 6-17 Sequential Machine Control mid Cycle Start Logic

Selection branch diverge

Figure 6-18 shows how an SFC that separates into 2 divergent paths. The sequence will take either path “A” or path “B”. Only one path is executed.

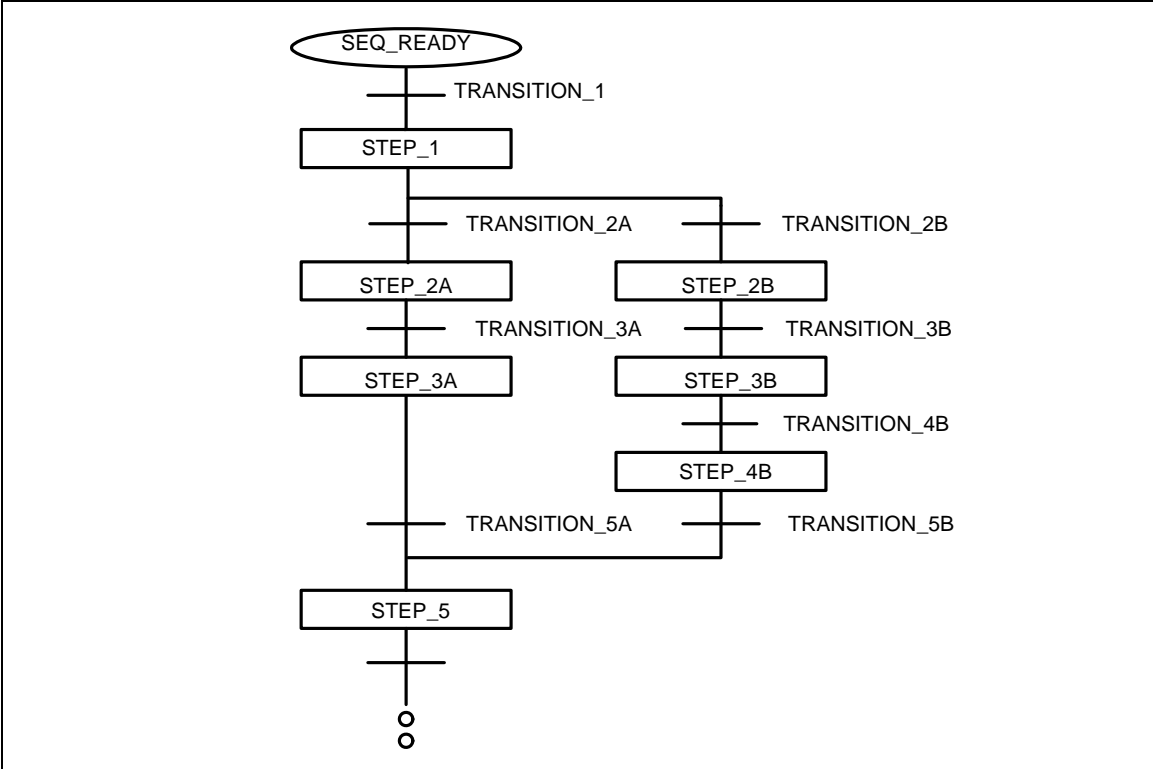


Figure 6-18 Sequential Machine Control Selection Branch Diverge SFC

Figure 6-19 shows the ladder logic implementation of the divergent path which represents the SFC in Figure 6-18.

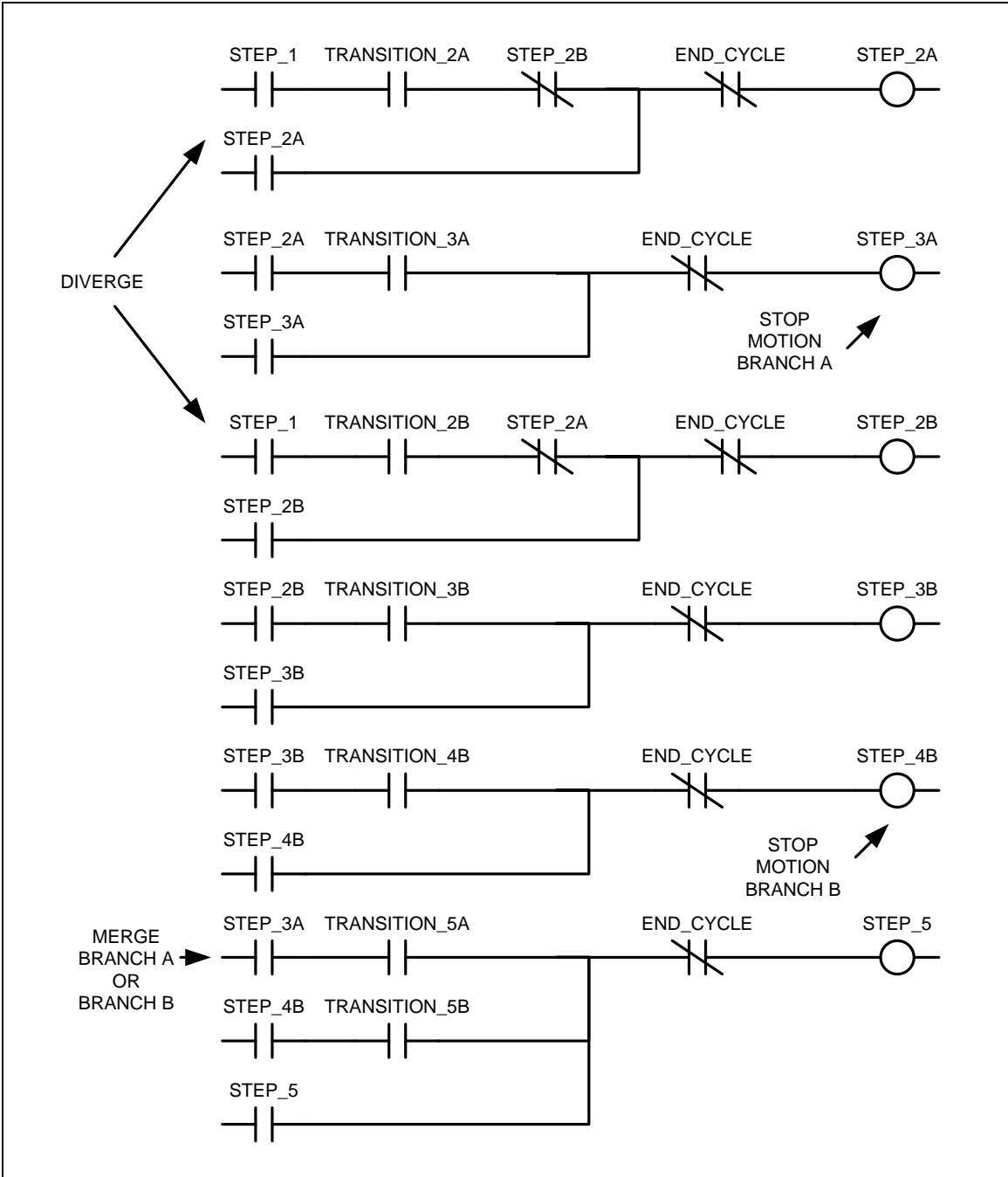


Figure 6-19 Sequential Machine Control Selection Branch Diverge Logic

Parallel branch diverge

Figure 6-20 shows the SFC for a parallel divergent branch. Step 2A and Step 2B are both initiated when transition 2 is true.

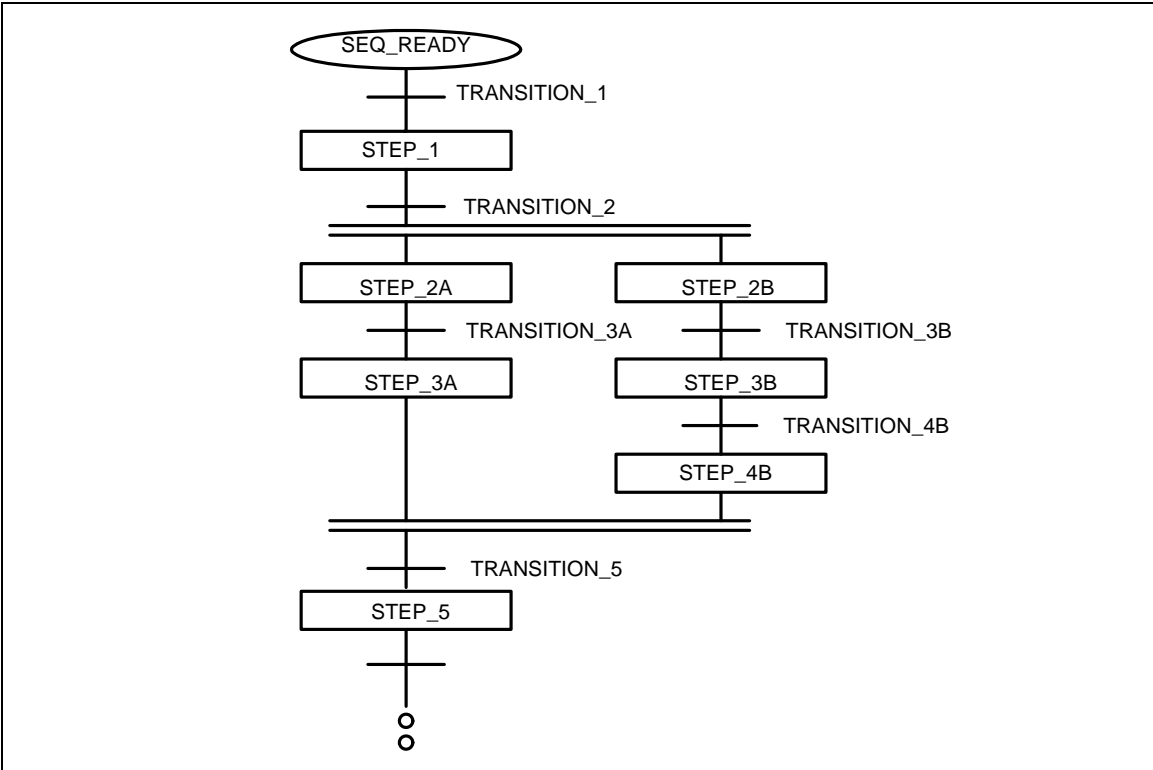


Figure 6-20 Sequential Machine Control Parallel Branch Diverge SFC

Figure 6-21 shows the ladder logic implementation of the parallel path which represents the SFC in Figure 6-20. The cycle will continue at step 5 when motion has stopped in branch “A” indicated by step 3A and that motion has stopped in branch “B” indicated by step 4B. Transition 5 can be eliminated if we are only to wait for motion to stop on both branches and then proceed.

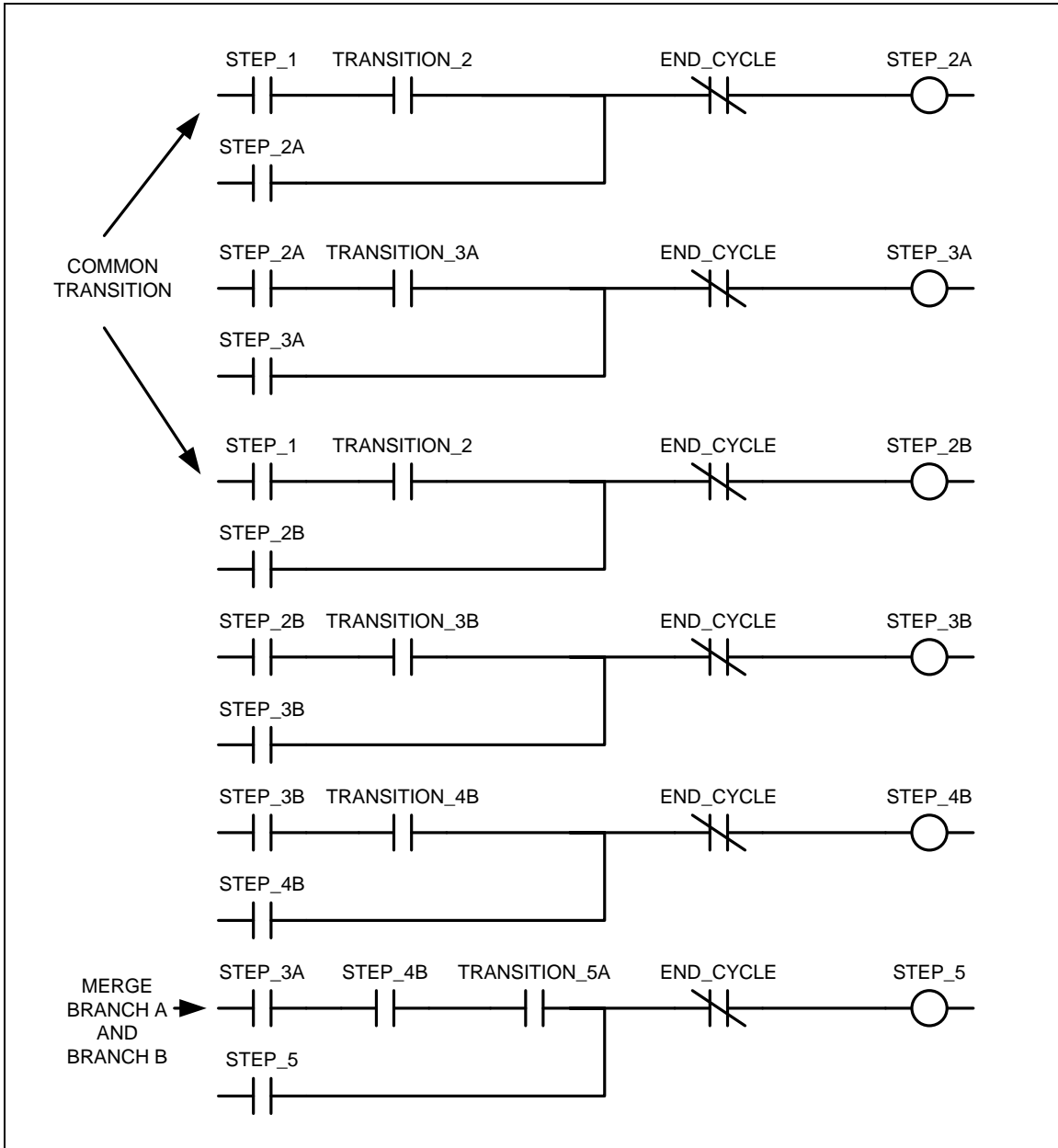


Figure 6-21 Sequential Machine Control Parallel Branch Diverge Logic

Chapter 7 Determining Priority

First Logic for 2 stations

Let us assume that we have two tanks. We want to discharge each tank when it is full. If both tanks are full, they should alternate based upon which tank became full first. This type of problem is common. The same problem exists if you want to alternate the use of pumps for example. An additional use is when two or more stations merge into one on a package conveyor.

In Figure 7-1, when a station is ready, it becomes first if the other station is not already first. Only one station can be first. The station that is first will remain first until it is no longer ready. At that time the other station will become first if it is ready. I call this priority logic “First Logic”.

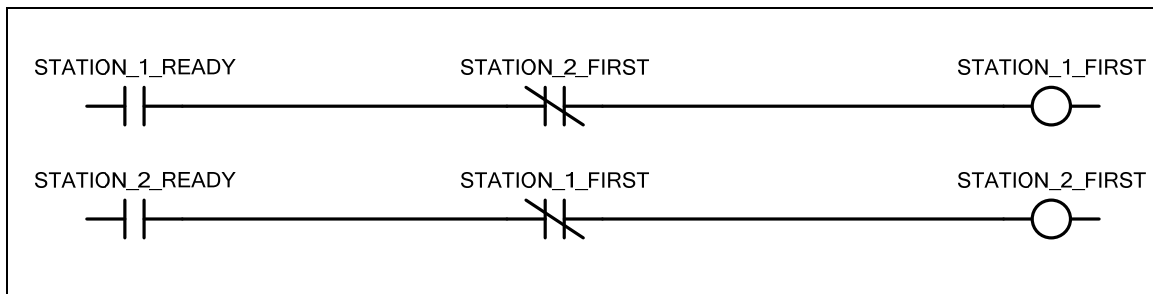


Figure 7-1 First Logic for 2 Stations

Figure 7-2 shows three vessels. UNIT_1 and UNIT_2 can discharge into UNIT_3. Unit 1 and 2 are filled until the high level switch is made.

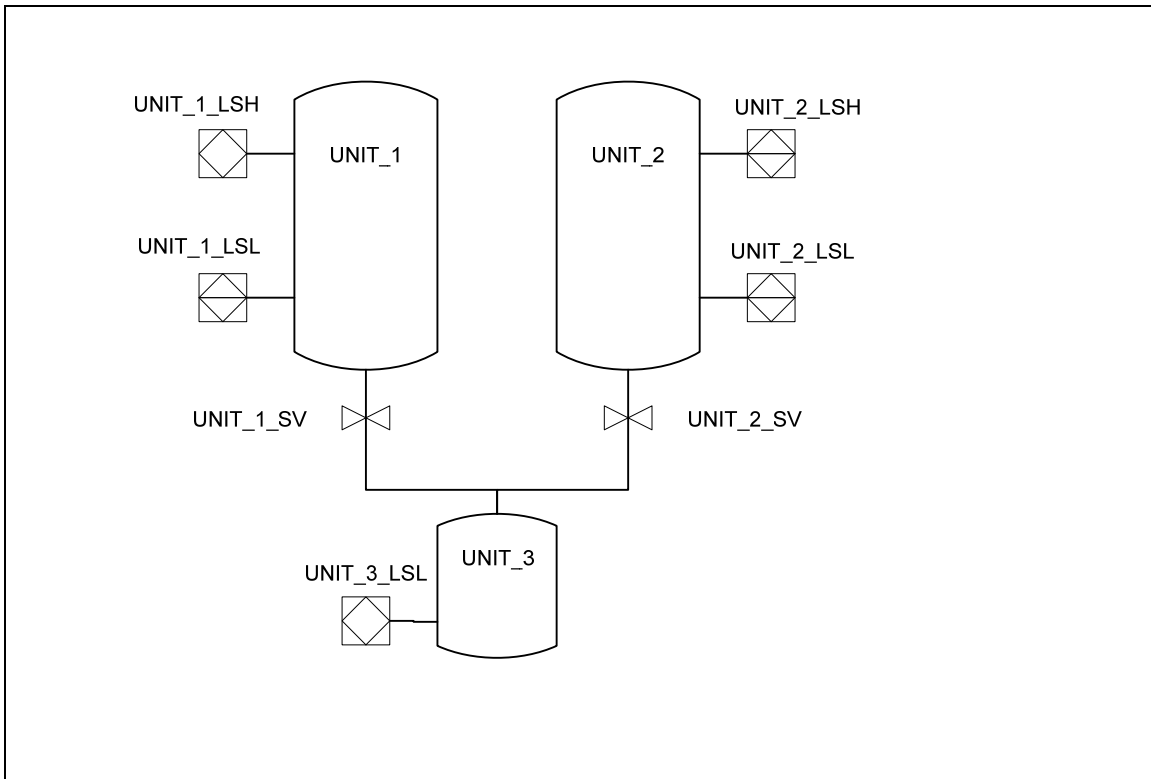


Figure 7-2 First Logic Example Process

Figure 7-3 shows the logic necessary to discharge unit 1 and 2 after they are filled. The first two rungs keep track of the priority in which the units are discharged. In this case the first unit that reaches the high level switch will discharge first.

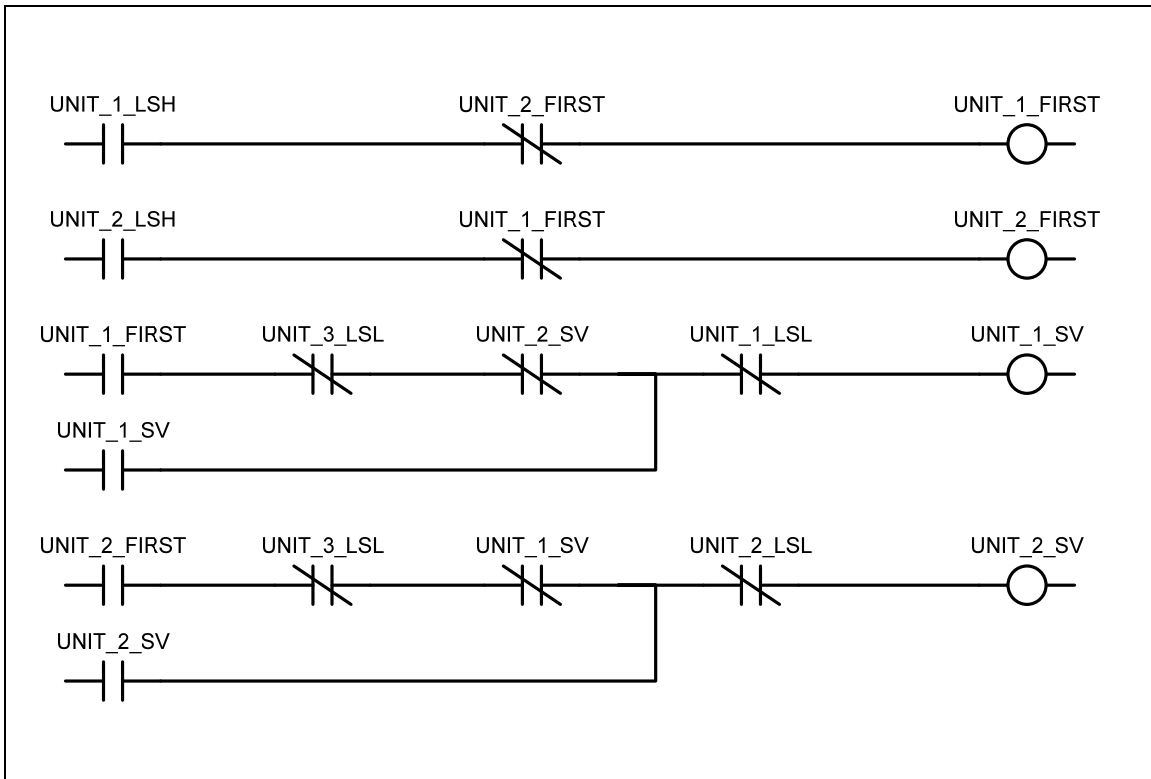


Figure 7-3 First Logic Example

First Logic for 3 or more stations

Figure 7-4 shows 3 work cells on a conveyor system. Parts arrive to the work cell on a carrier and are stopped at a physical stop. When the work is complete at the cell, the stop is lowered and one part is released from the cell.

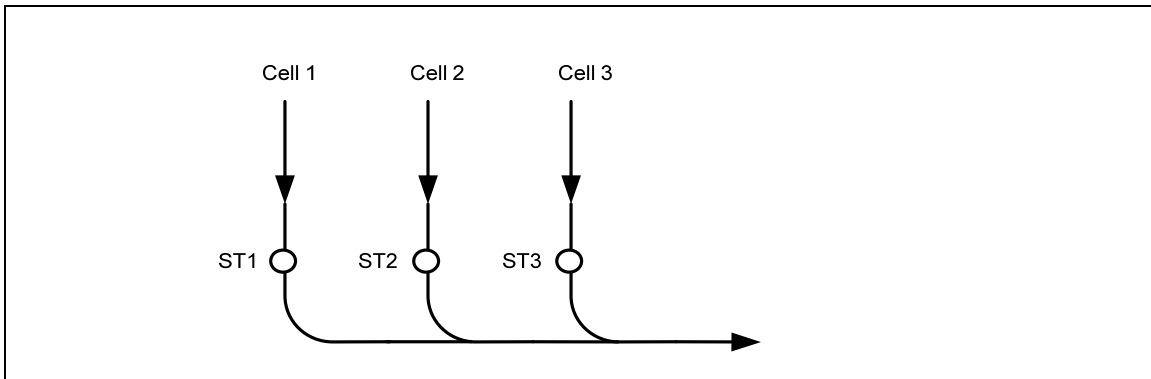


Figure 7-4 Three Station First Logic Example

With three stations we have to keep track of the first and second priority. By default the last station ready, is third. Figure 7-5 shows the logic to keep track of the station that is second. This logic only allows one station to be second. Also, when a station becomes first, then the second bit is turned off. This allows the next station to become second.

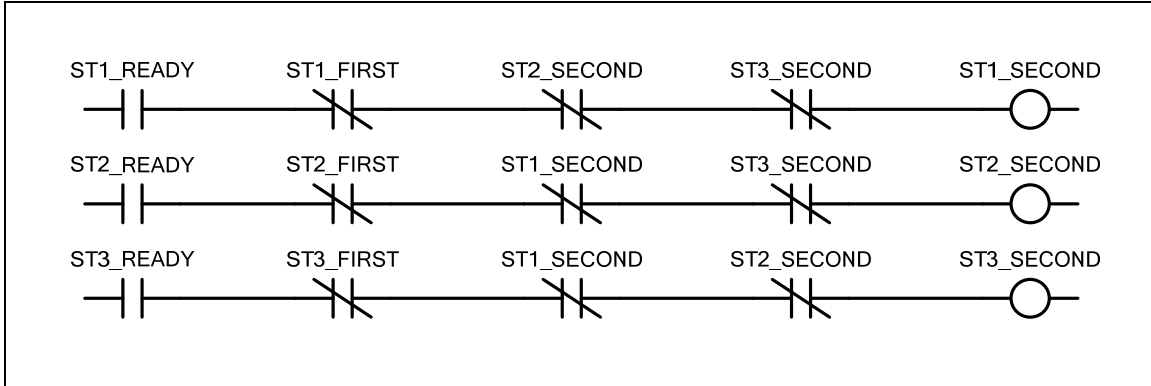


Figure 7-5 Three Station First Logic Second Priority

Figure 7-6 sets the first bit for each station. Only one station can be first. It must have been second before it can become first. When a station becomes first the second bit is reset. So, we have to latch up the first bit. The station remains first until it is no longer ready. This would occur, for example, when a part leaves the station.

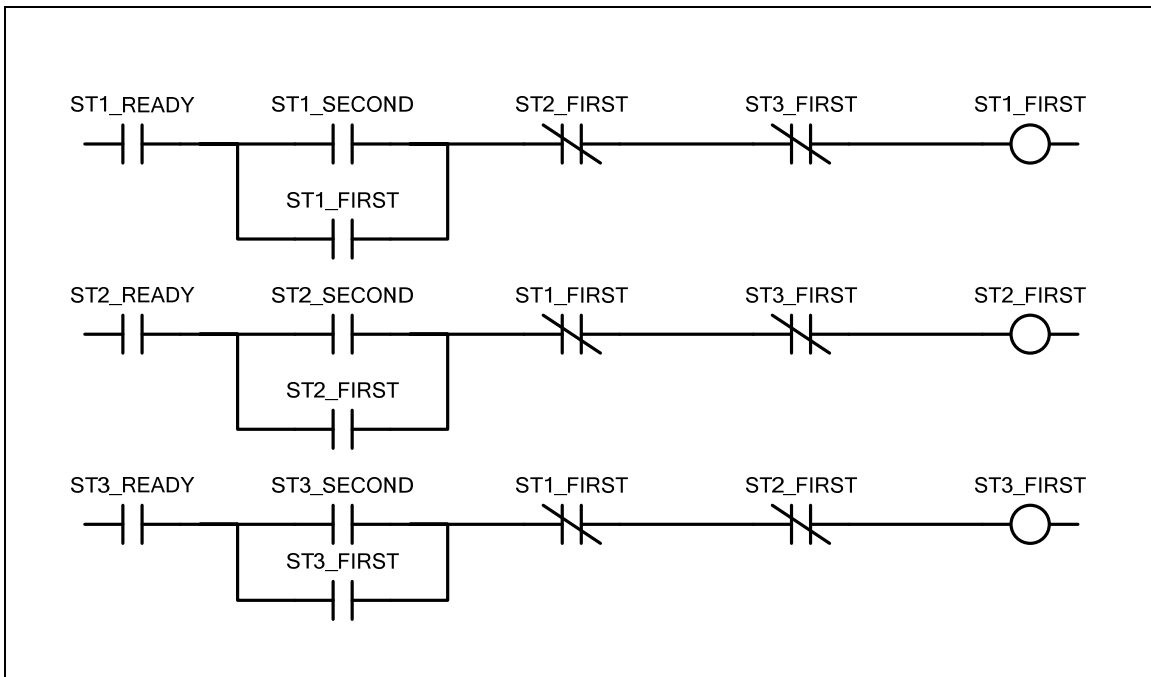


Figure 7-6 Three Station First Logic First Priority

One advantage of this method is that a stop can be taken out of either the first or the second position and the remaining stations will move into the vacated position.

When using this approach, we need to have one bit for each priority that we want to keep track of. If there are 3 stations, then we must keep track of the first and second priorities. If we have four stations then, we must keep track of first, second, and third. We don't need a rung to keep track of the last place because the last station in the queue will default to last place when it is ready. The number of rungs is then determined by the number of stations that can enter the queue. Figure 7-7 shows the equation for the number of rungs required to program the priorities.

<p>S = The number of stations R = The number of rungs</p> $R = (S - 1) \times S$

Figure 7-7 First Logic Number of Rungs Calculation

As you can see, if we have ten stations, then the number of rungs required is ninety. For a small number of stations, perhaps five or less, then this method is valid. However, for a large number of stations, we need to consider a different method.

Priority Stack

Let's assume that we have ten stations. Each station is assigned a number. Station 1 is assigned 1 Station 2 is assigned a 2 etc. We can create a priority stack which is an array of double integers. The station numbers are loaded into the array when the station is ready. Each position in the array determines what priority the station will have. Array element 1 will have the highest priority and array element 10 will have the lowest. We will write some logic that will allow us to load a station number into the top of the array at position 10 and then the station number will be shifted down in the array until it becomes the highest priority at position 1. As other stations become ready they will stack up behind the first. We will also allow any station to be pulled from the stack no matter what priority position it occupies. This may occur for example if a station is disabled after it has been loaded into the stack. Otherwise, we could use a simple FIFO to determine the priority. An additional benefit of this method is, we can display the priority array on an HMI allowing the operator to see the order that the stations occupy in the priority stack.

In Table 7-1, we define the user defined variable PRIORITY.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
READY	Station is ready	Boolean
FIRST	Station is first priority	Boolean
LOADED	Station is loaded into the stack	Boolean

Table 7-1 User Defined Data Type PRIORITY

In Table 7-2 the variables are defined for the priority stack.

<i>Tag</i>	<i>Description</i>	<i>Type</i>
STATION	Station array	PRIORITY[11]
PRIORITY_STACK	Holds the station number	DINT[11]
A	For Next loop index from 1 to 10	DINT
B	Index from 10 to 1	DINT
FSC_CTRL	File search control variable	CONTROL

Table 7-2 Priority Stack Variable Definition

In Figure 7-8 we execute the STACK FOR/NEXT routine. The routine executes one time for each station. So, in this example we are determining the priority for 10 stations.

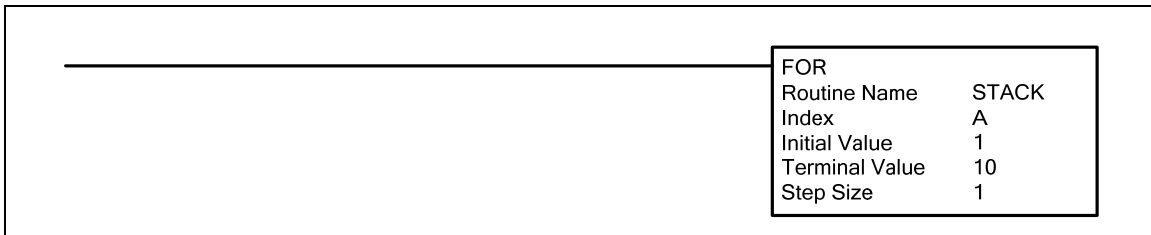


Figure 7-8 Priority Stack Call to the Stack Routine

Figure 7-9 shows the first rung of the STACK routine. When a station becomes ready, the station number is moved into the highest stack position. The LOADED bit is checked to make sure that the number is only loaded one time into the stack. If two stations become ready in the same program scan we have to prevent them from both writing into position 10 before the previous station has moved down in the stack. So, we verify that position 10 is equal to zero before the move is made.

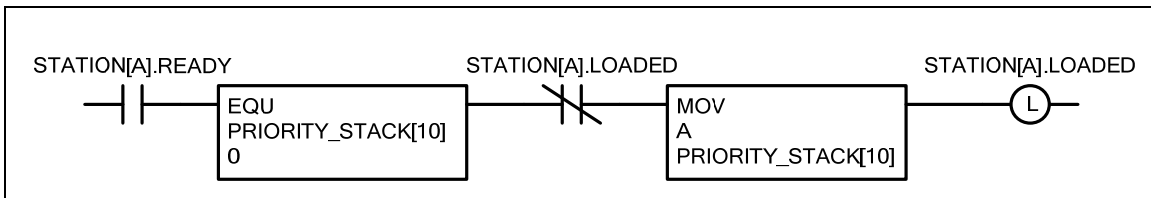


Figure 7-9 Priority Stack Routine STACK Move the Station Number into the Stack

In Figure 7-10 the station number is shifted down to the next available position in the stack. As stations become ready they will stack up behind the previous stations. When a position in the stack is vacated, then the stations behind that position are shifted down. We use the variable B to sequence from position 10 to position 1 of the array. This will allow the station number to be moved from the position 10 to 1 in one program scan.

Chapter 7 Determining Priority

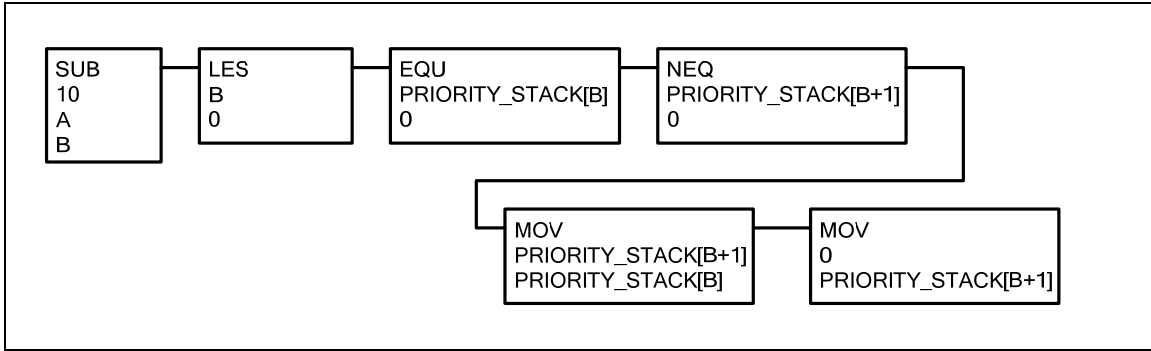


Figure 7-10 Priority Stack Routine STACK Shift the Station Number to Priority One

In Figure 7-11 we check the stack to see if any stations that are loaded are no longer ready. When a station is found then the station number is cleared. This will allow any stations behind this vacated position to shift down in the stack.

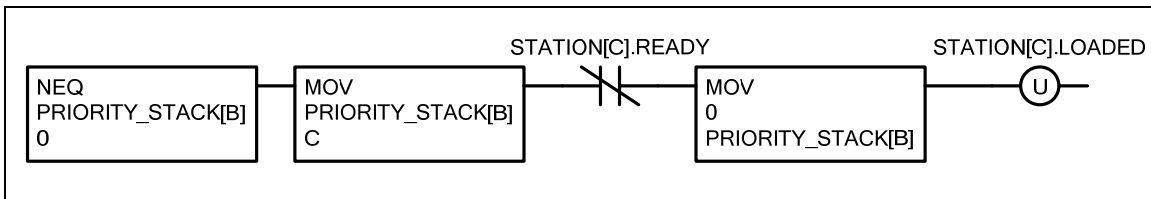


Figure 7-11 Priority Stack Routine STACK Clear the Station Number from the Stack

In Figure 7-12 the first position of the stack is checked. If a station is ready and the bottom of the stack is equal to that station number then the FIRST bit is set for that station. All other FIRST bits are reset so that only one station FIRST bit is set.

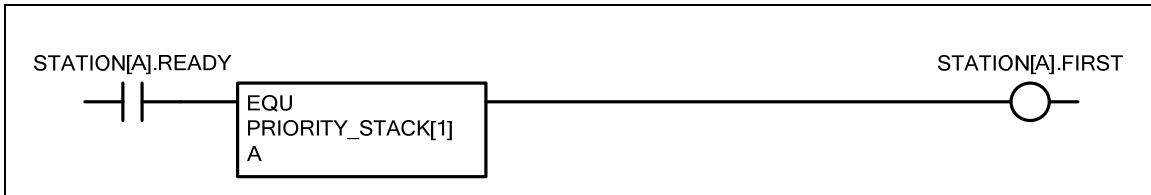


Figure 7-12 Priority Stack Routine STACK Determine the First Priority

When the FIRST bit is set, the part at the station can be released. Once, the part has cleared the station the READY bit would be reset. Our routine will then clear the station number from the array allowing any stations still in the array to move to a higher priority.

Chapter 8 Sortation

Tracking with an encoder

Figure 8-1 shows an example box sort conveyor. Boxes are introduced to the sorter at the scanner. The barcode is read which determines the destination. The barcode is sent to the PLC through a serial interface when the front edge of the box leaves the scanner area. This is accomplished through a photo-eye which is connected directly to the scanner. An incremental encoder is used to determine the distance that each box has moved down the conveyor. A high speed counter card interprets the encoder pulses and provides an integer value that changes proportionally with the distance traveled. The integer value will change from 0 to 32767 and will then wrap back around to 0. A photo-eye at the entrance of each sort lane is used to verify that a box is in position to divert and to accurately time the diverter. Each diverter consists of pop-up rollers powered by a single air operated solenoid.

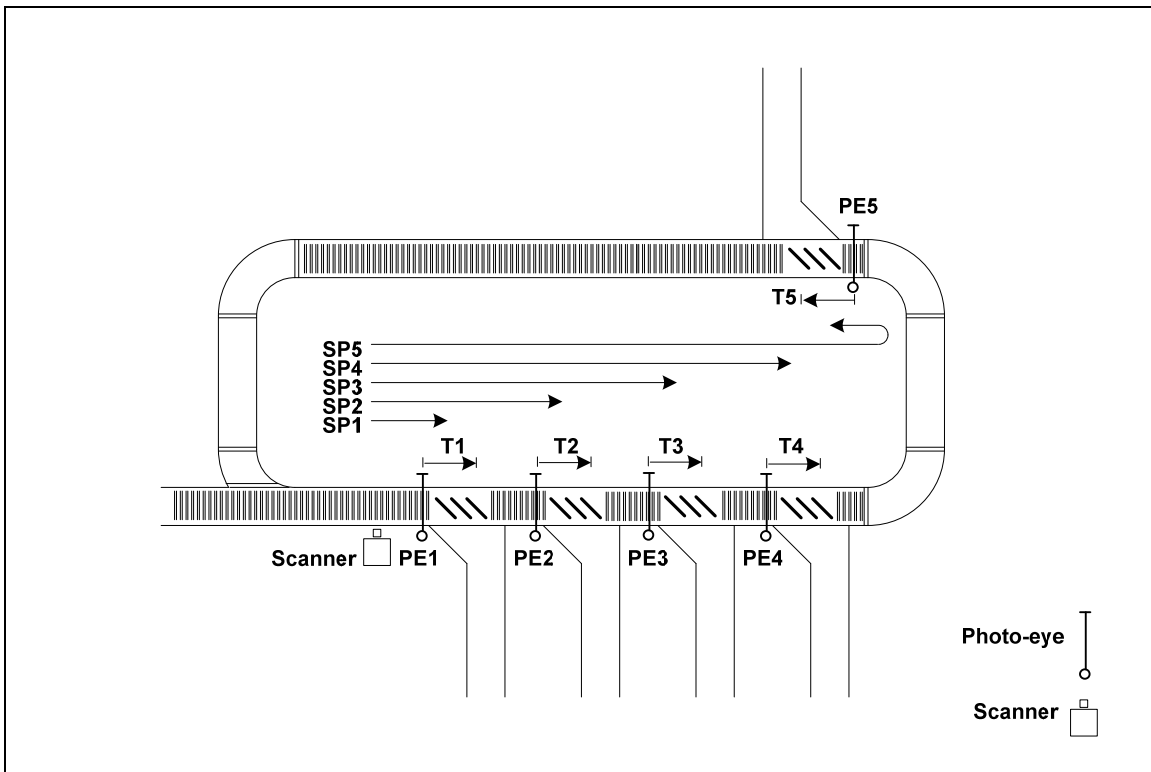


Figure 8-1 Sortation Conveyor Layout

In Figure 8-2 a 2 character barcode is read and put into the destination string BARCODE. The one shot ARL_CTRL_DN_OS is turned on for each read of the scanner.

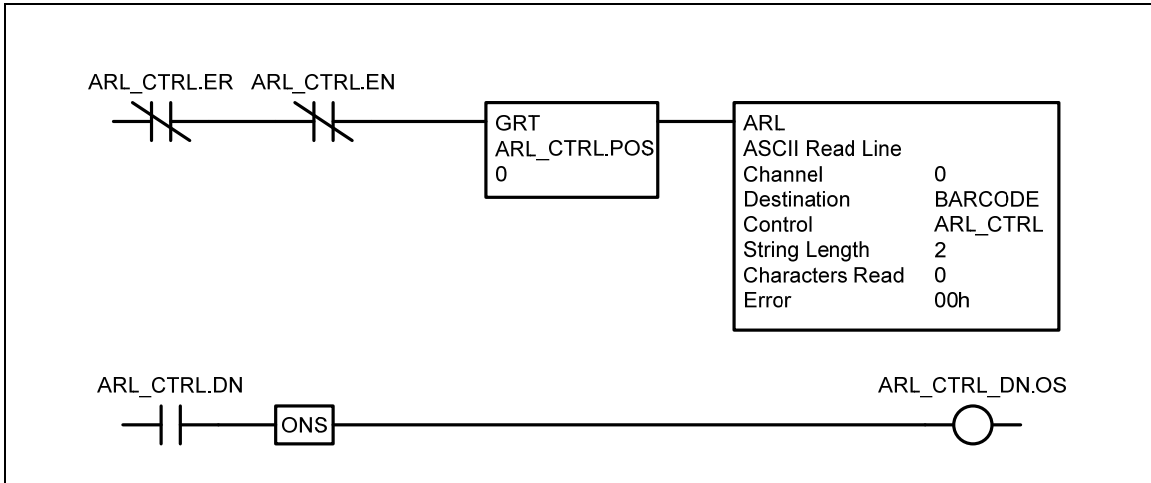


Figure 8-2 Reading the Barcode thru the Serial Port

After the barcode is read we perform a simple validation check on the barcode and turn on VALID_READ. The barcode string is converted into an integer variable SORT_LANE. Figure 8-3 shows the logic.

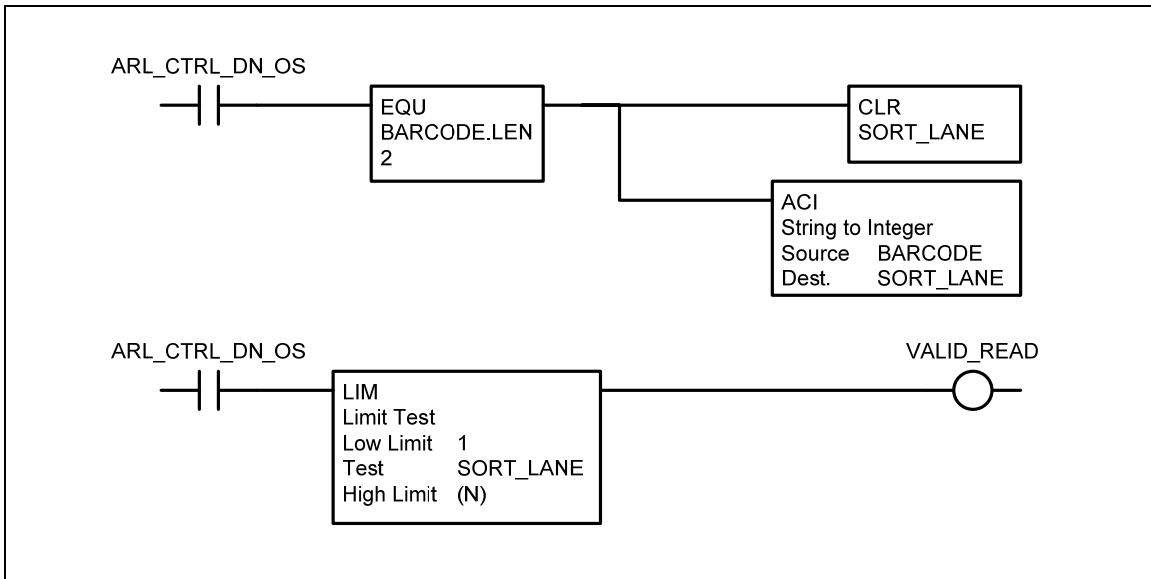


Figure 8-3 Checking for a Valid Barcode Read

In Figure 8-4 we take the current value of the encoder and store it into the lane FIFO. The length of each FIFO is determined by the maximum number of boxes that can be in transit to the sort lane. A FIFO is created for each lane.

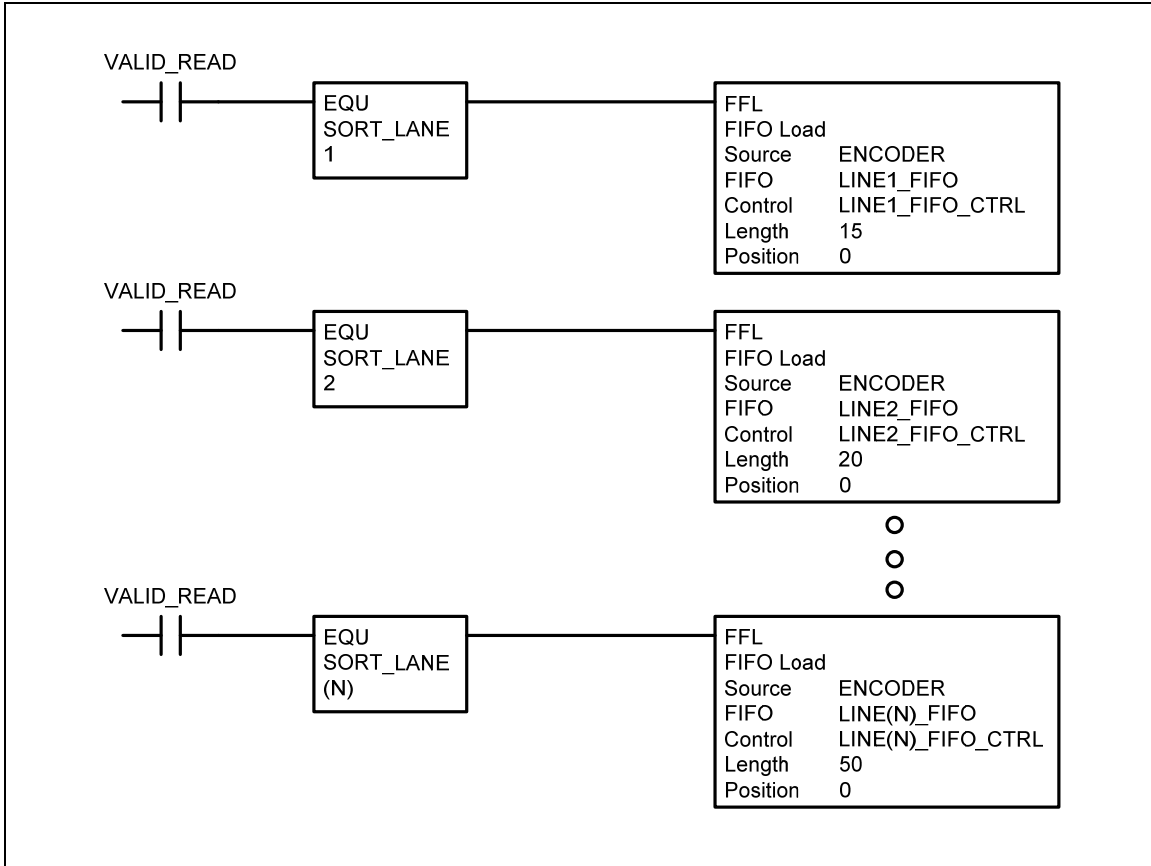


Figure 8-4 Loading the Sort Lane FIFOs

In Figure 8-5 the FIFO is unloaded into START_1. This is the starting encoder value for the next in line box to be diverted. We set the wait for setpoint bit, WAIT_FOR_SP to allow us to wait for the box to reach the diverter. After the box reaches the diverter, then this bit is reset and the FFU instruction will unload the starting position for the next box to be diverted to this lane.

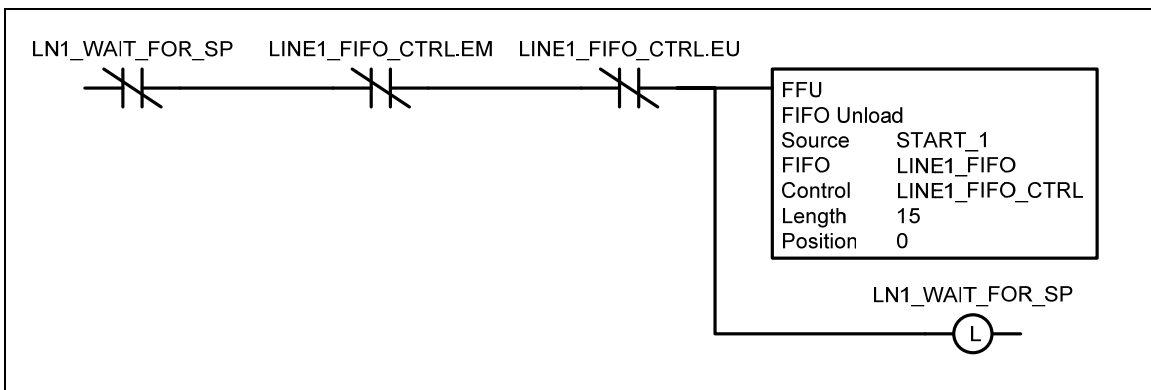


Figure 8-5 Unloading the Sort Lane FIFOs

In Figure 8-5 we consider the use of an incremental encoder whose current value is in the variable ENCODER. The variable START_1 is the saved value of the encoder when the box was inducted into the sorter at the barcode reader. The calculated value OFFSET_1 is the number of pulses the box has traveled from the time it left the scanner. The setpoint, SETPOINT_1, is the number of pulses from the induction to the sort lane. The offset value is then compared to the setpoint to determine when to divert the box. Because the encoder counts to a maximum value and then resets to zero, we have to consider two cases to calculate the offset. The first case is when the setpoint is reached before the encoder wraps around. The second is when the encoder resets to zero before the setpoint is reached.

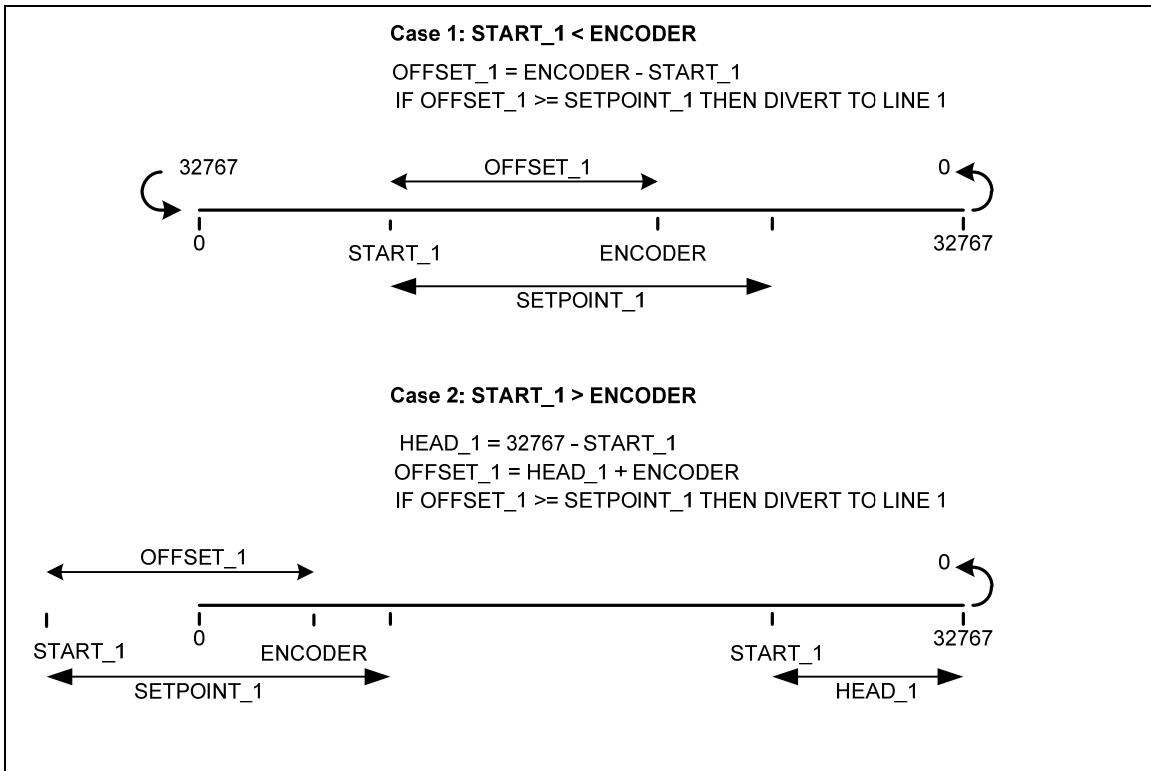


Figure 8-6 The Wrap Around Encoder Diagram

In Figure 8-7 we do the math and wait for the box to reach the transfer. When the encoder has reached the setpoint, the bit LN1_ENABLE_XFR is turned on. The wait for setpoint bit is then reset allowing the FFU instruction to retrieve the start value for the next box. This means that LN1_ENABLE_XFR is on for one program scan.

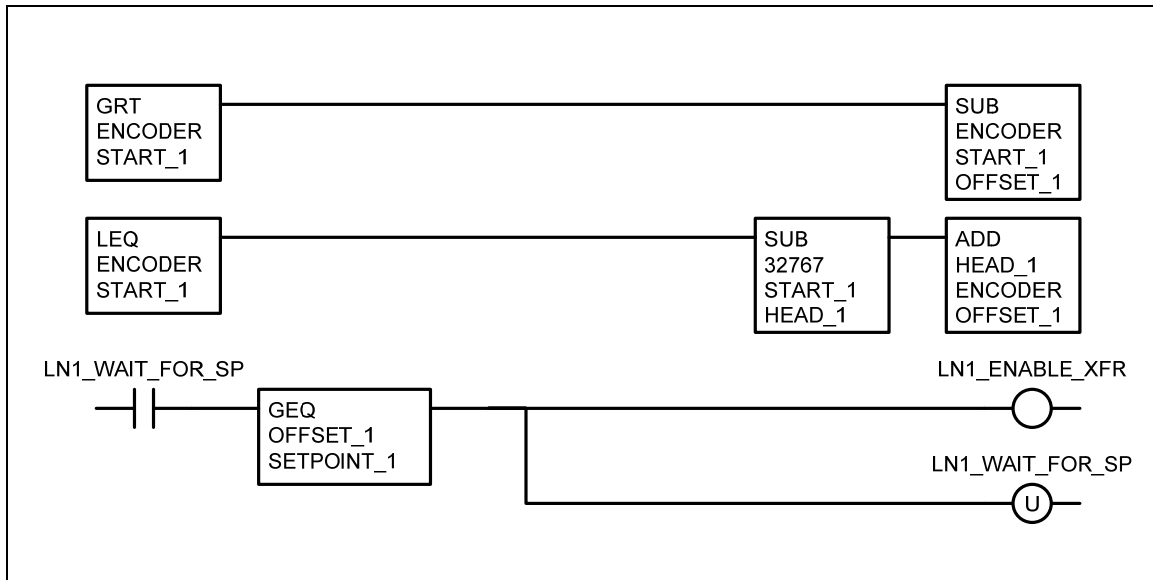


Figure 8-7 Enabling the Transfer with the Encoder Window

After the encoder has reached the setpoint, we check to see if there is a box at the diverter. This is done by starting a timer when the divert photo eye prior the diverter is made. This timer then provides us with a window of time in which the diverter is enabled. If the encoder reaches the setpoint during this time, then it is assumed that the box in front of the diverter is the one that is supposed to be diverted. The divert photo eye also allows us to compensate for any slippage that may have occurred while the box has been traveling down the conveyor. If the encoder reaches setpoint and the diverter has not been enabled, then no transfer occurs for that setpoint. We would then assume that the box may have been taken off the line prior to the diverter, or the box failed to reach the diverter when it was supposed to for some other reason. If the box had been slowed because it was an odd size for example, then it would miss the diverter. In Figure 8-8 a divert timer is started when the photo eye at the transfer is made.

When the sorter is first commissioned, all of the setpoints for each sort lane have to be set. I like to turn on a horn or light when the setpoint has been reached. I will do one lane at a time and enable the horn or light logic for that lane only. I put one box through the scanner and wait for the box to arrive at the diverter. I would then adjust the setpoint when I hear the horn. On a roller conveyor some slippage will occur between the box and the conveyor. This may cause different box sizes and weights to travel down the conveyor at different rates. Running several box sizes through the sorter will allow you to adjust for this. After you determine which boxes run faster, then adjust the setpoint so that the horn sounds when the box is at the far end of the divert timer window. This will allow the slower boxes to be enabled on the front side of the timer window.

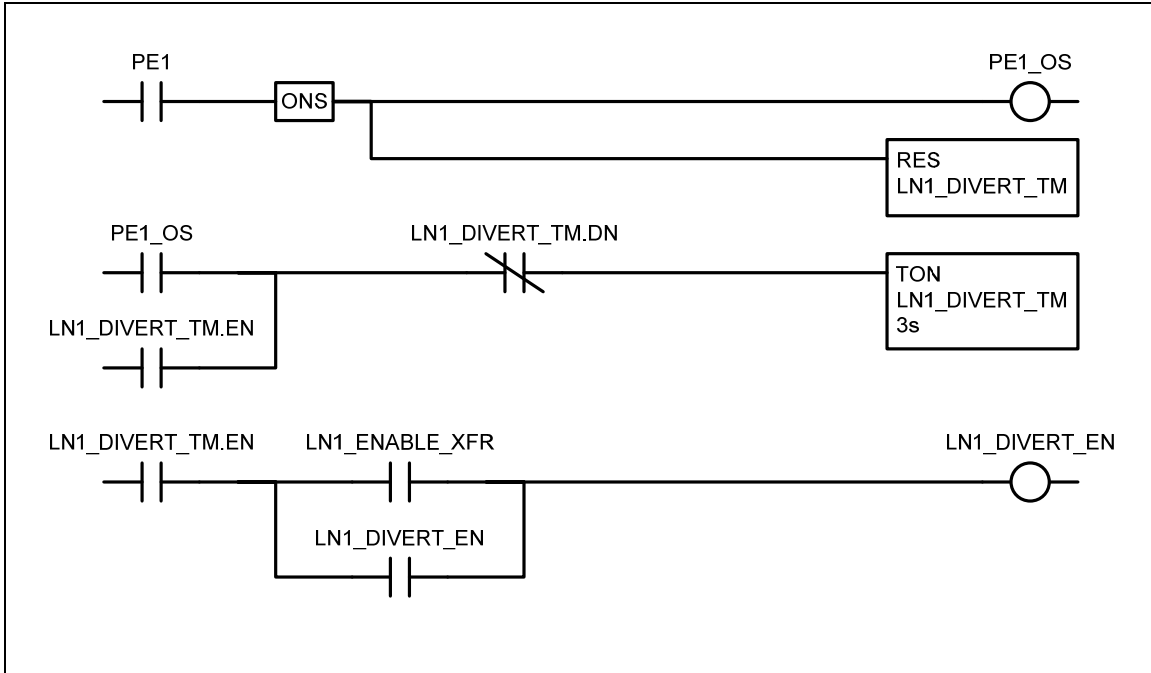


Figure 8-8 Checking if the Sort Lane Photo Eye is in the Encoder Window

When the divert timer is done and the diverter has been enabled the divert solenoid is energized and the box is sent into the sort lane. Because the timer is started when the photo eye is first made, all of the boxes will be diverted with their front edge aligned to far rail of the sort lane. Short and long boxes will all be aligned at this edge. If you need to align the boxes on the near edge of the sort lane, then the negative transition of the photo eye would be used to start the timer. Starting the timer on the negative edge of the photo eye would however provide a much smaller window that we could catch the setpoint with. To compensate for this we would enable the setpoint when the photo eye is made or the divert timer is timing. Figure 8-9 shows the logic necessary to divert the box into the sort lane.

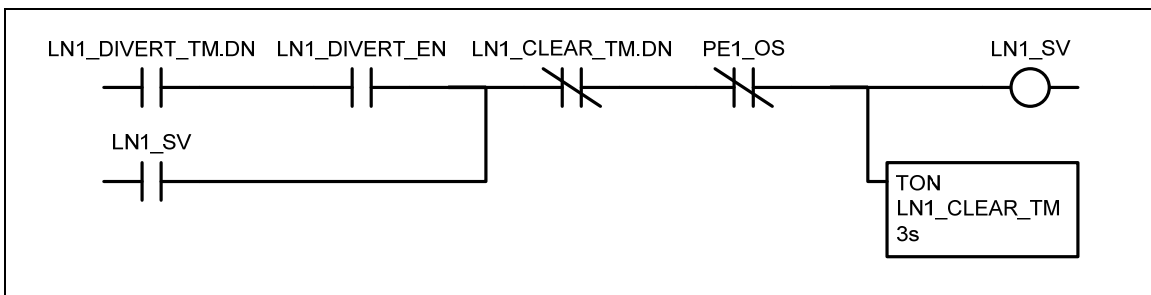


Figure 8-9 The Sort Lane Solenoid Logic

Tracking with timers

In the previous example we used an encoder to track the boxes to the sort lane. We can also use timers to track the box to the sort lane. A timer is started for each box that goes by the scanner. The preset for that timer is set based on the destination of the box. The timer address is moved into the lane FIFO. When we unload the FIFO we wait for the timer to get done and then the box is diverted.

In Figure 8-10 a timer pointer is incremented when a valid read occurs on the barcode scanner. We turn on the enable bit for the timer pointed to by TM_PTR. There is a preset amount of time that it takes for a box to reach each lane. These presets are stored in the array TM_PRE[]. The preset for the destination SORT_LANE is moved into the preset of the timer pointed to by TM_PTR.

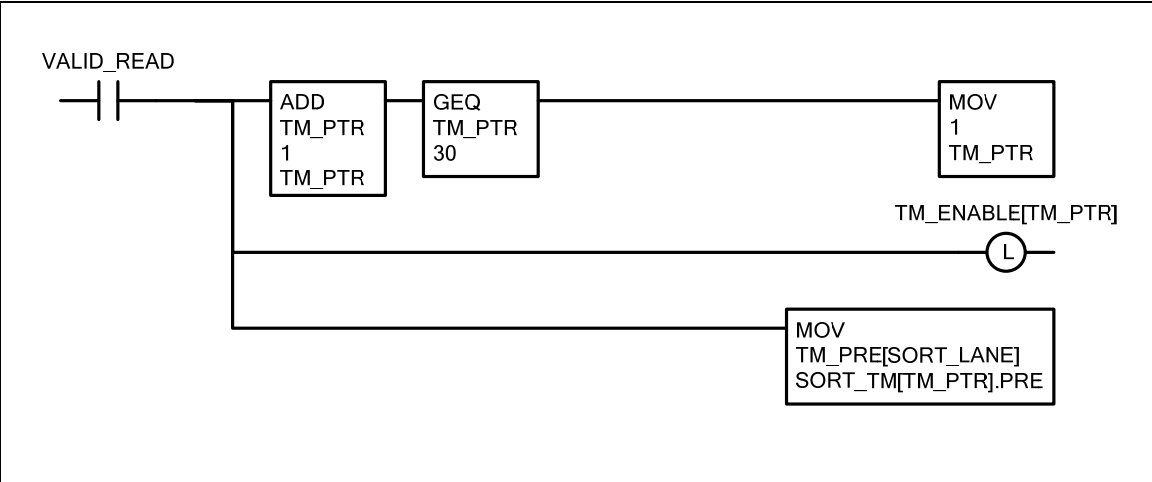


Figure 8-10 Enabling the Tracking Timer

In Figure 8-11 each timer is started if it is enabled. The done bit of the timer is seen by the program for 1 scan and then the timer is reset on the next scan. The timer logic is repeated for each timer. The number of timers in the timer array is determined by the maximum number of boxes that can be in transit on the conveyor.

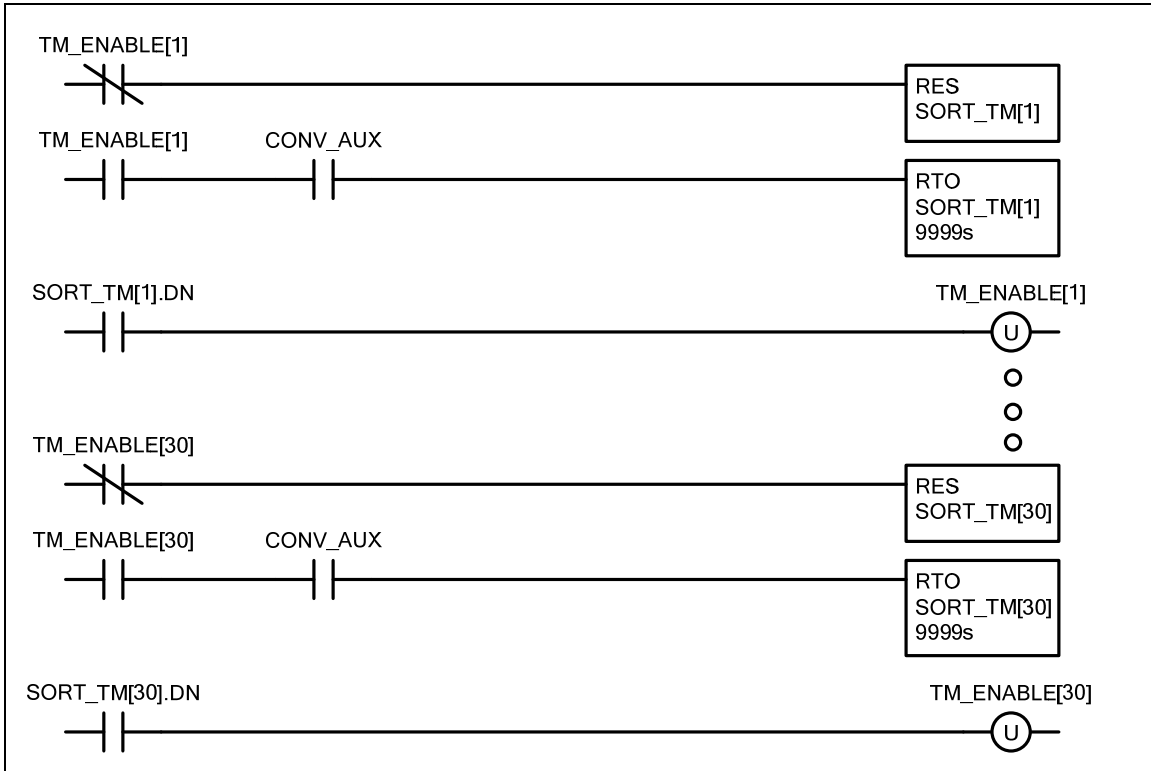


Figure 8-11 The Tracking Timer Logic

In Figure 8-11 the same logic is repeated for each timer. This logic can be reduced if we use the FOR instruction to perform a for-next loop on the timer logic. This instruction is not available in all PLC's. The FOR instruction shown in Figure 8-12 executes the routine TM_LOOP thirty times. The Index variable "A" is incremented for each iteration.

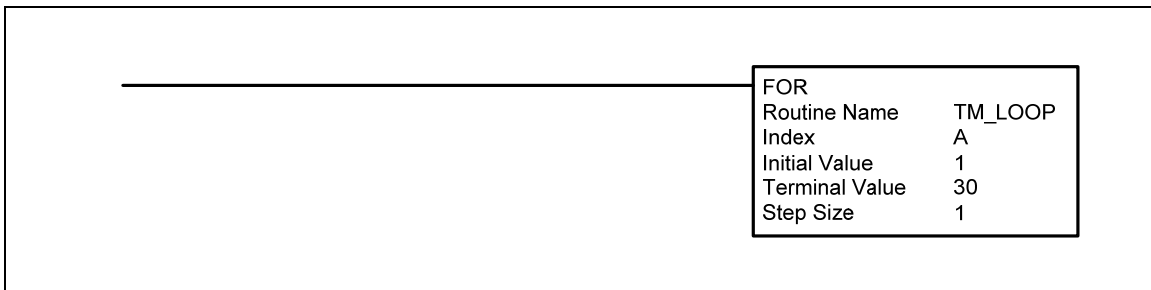


Figure 8-12 Executing the Tracking Timer Loop

In Figure 8-13 the routine uses the index variable “A” to execute the logic for each timer.

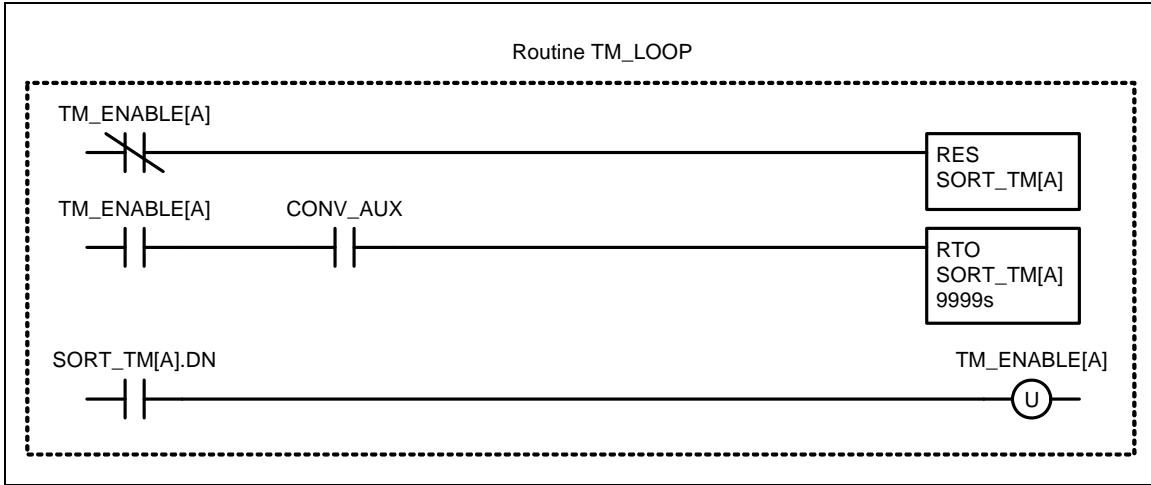


Figure 8-13 Tracking Timer Routine TM_LOOP

In Figure 8-14 the timer address is loaded into the lane FIFO.

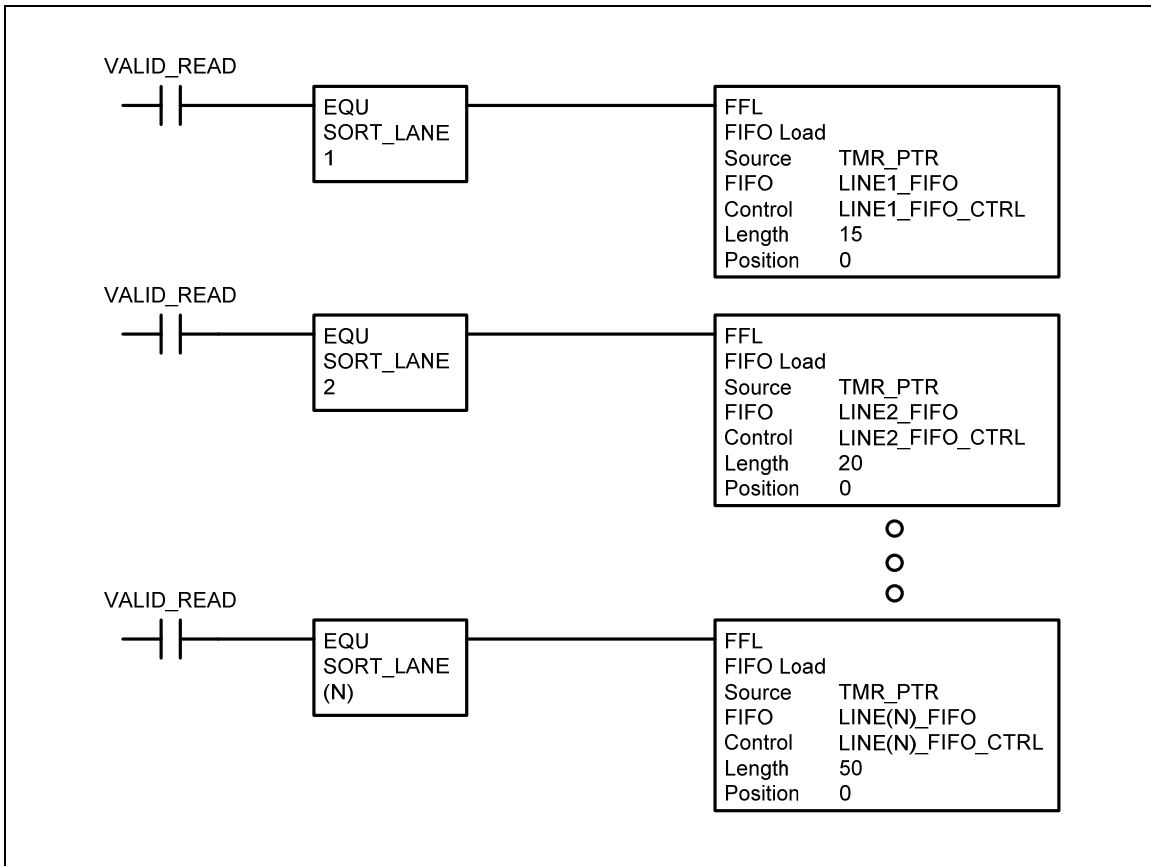


Figure 8-14 Loading the Timer Pointer into the Sort Lane FIFO

In Figure 8-15 the FIFO is unloaded into the lane timer pointer LINE1_TM_PTR.

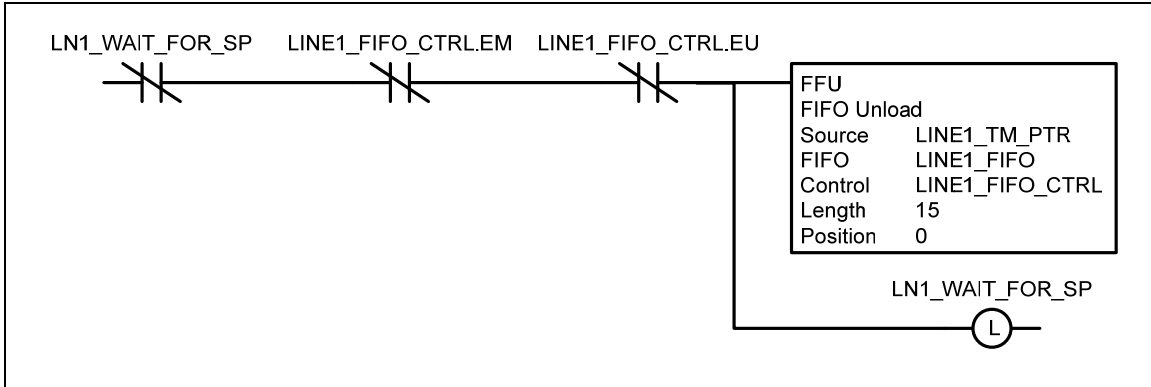


Figure 8-15 Unloading Timer Pointer from the Sort Lane FIFO

In Figure 8-16 the lane transfer is enabled when the timer is done. Refer back to Figure 8-9 in the encoder logic to finish out the divert logic for the timers. It is the same.

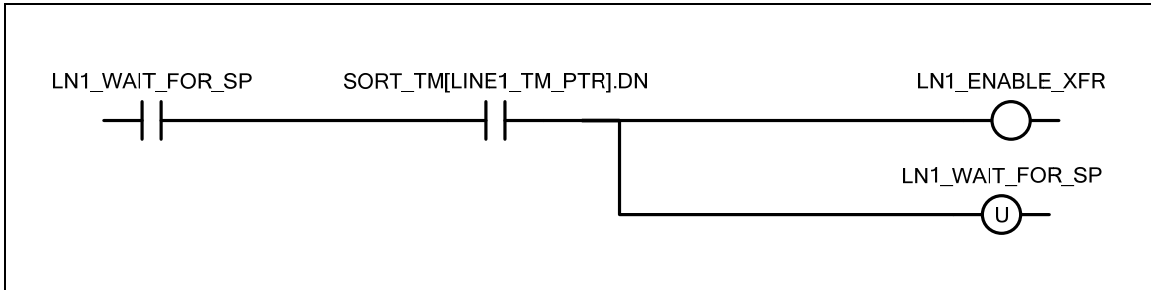


Figure 8-16 Enabling the Sort Lane with the Tracking Timer

Chapter 9 Zone Control

Zone control is a method of controlling the number of items that can enter a zone on a conveyor or machine. The number of parts that can be in the zone at any one time will determine if a bit is used to track the part or a counter is used to track multiple parts.

Power and Free, Stop to Stop, no accumulation

Let's consider a Power and free conveyor² with 2 stops. A power and free conveyor consists of 2 parts as the name implies. The powered part is a chain riding around an I beam or in between 2 pieces of channel. Carriers ride freely on a lower set of channels. The carriers are disengaged from the chain when they enter a stop or when they accumulate behind another carrier. Carriers will also disengage from the chain when they are being transferred from one chain onto another. In these cases the carrier will be pushed on to the second chain with an air operated pusher, or the back of the carrier will catch the next chain dog that will push the carrier onto the second chain.

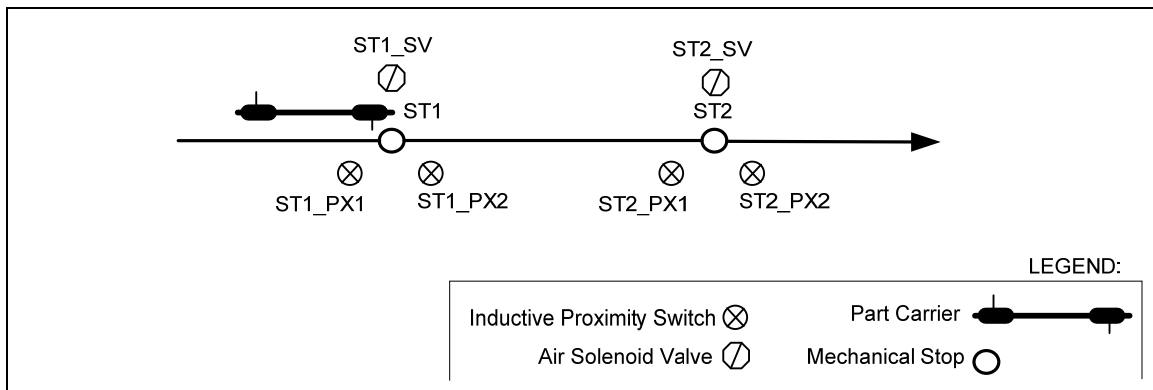


Figure 9-1 Power and Free Stop to Stop no Accumulation

² There are several Power and free conveyor manufacturers. Visit their websites to get a detailed understanding of their fundamentals. Southern System, <http://www.ssiconveyors.com>

Chapter 9 Zone Control

Figure 9-2 shows how a power and free program can be organized. The AA_UPDATE_IO routine is used to synchronize the inputs with the program scan. We discussed the reasoning for this in Chapter 1. I prefixed the routine with AA so that it would sort alphabetically to the top of the routines. The MainRoutine calls each of the other routines with a jump to subroutine instruction.

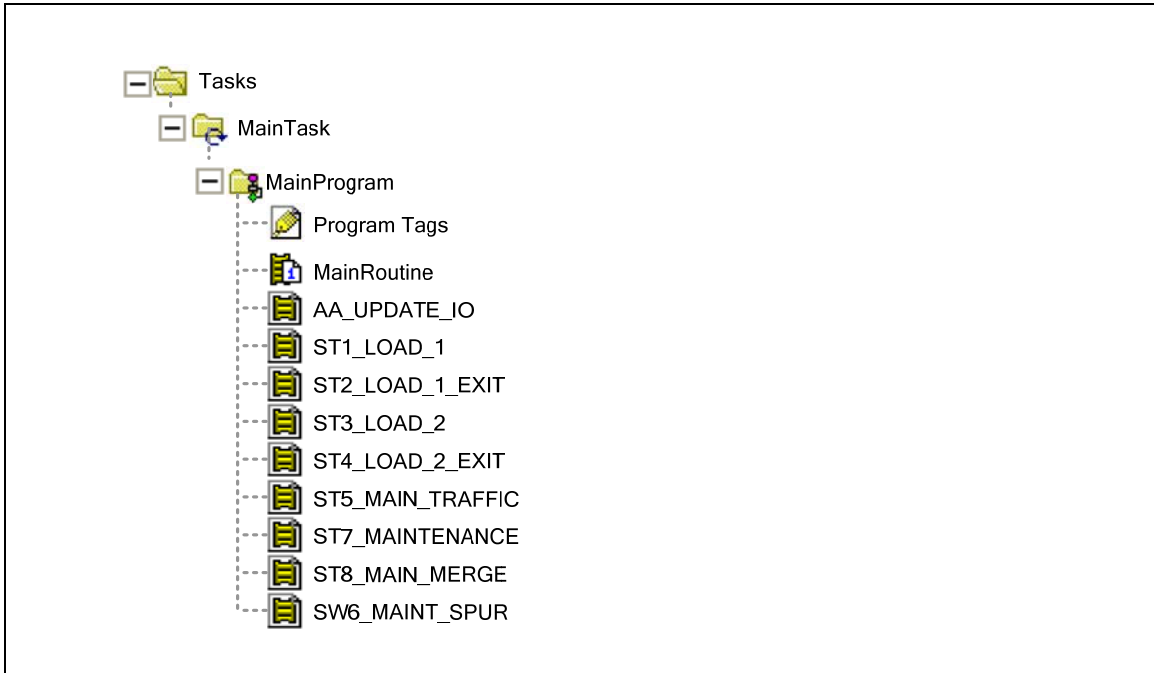


Figure 9-2 Power and Free Program Organization

Figure 9-3 shows how stop 1 is programmed. A carrier is present when proximity switch ST1_PX1 is made. ST1_PX1 is not maintained when the carrier enters the stop so, internal bit ST1_PRES is latched on indicating that a carrier is in the stop. The carrier remains present until ST1_PX2 is made. The stop is opened (ST1_SV is energized) when the carrier is present and stop 2 is not full (normally closed ST2_FULL). The carrier will then catch the next chain dog and move out of stop 1. Stop 1 is closed when the carrier passes ST1_PX2. Stop 2 becomes full when ST1_PX2 is made. Stop 2 is clear when stop 2 is open and ST2_PX2 is made. We make sure that stop 2 is open and then closed before we clear zone 2. Stop 2 is then programmed the same as stop 1 if the downstream zones are the same.

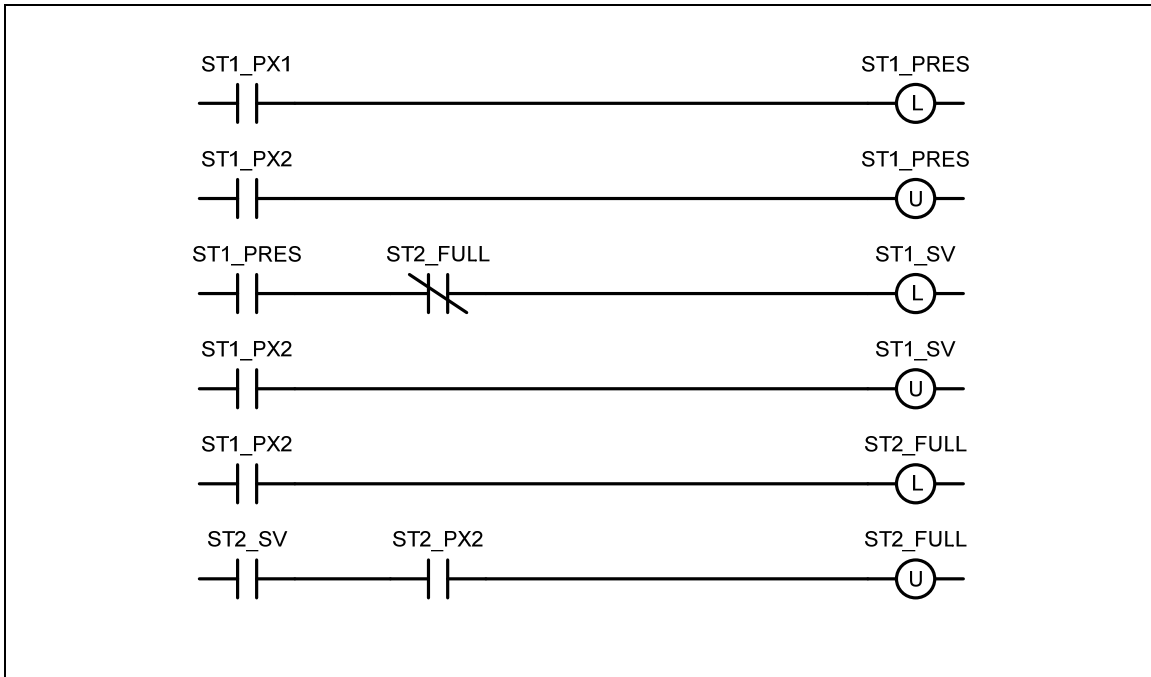


Figure 9-3 Power and Free Stop to Stop no Accumulation Logic

Note how ST2_FULL is being unlatched only during one program scan since ST2_PX2 will clear the zone and then unlatch ST2_SV in the same way that ST1_SV is unlatched by ST1_PX2. In order for this to be true however, the logic for stop 2 must be after the unlatch for ST2_FULL in the program scan.

Power and Free, Choke Zone

Figure 9-4 shows a typical choke zone on a power and free conveyor. Only one carrier is allowed in the choke zone. This has two effects on the way the system operates. The first is that it does not matter how many carriers have accumulated in front of the choke zone. The second is that the spacing between carriers can be no closer than the length of the choke zone when they are not accumulated behind stop 2. There are several reasons why a choke zone would be used. I will list a few here.

- To limit the number of carriers on a vertical rise or fall
- To keep carriers from accumulating around a curve.
- To provide spacing between carriers.

Usually only 1 carrier would be allowed in the choke zone. However a count zone might be used within the choke zone. For example, on a vertical rise, the number of carriers being lifted may be limited to 2 carriers due to the carrying capacity of the conveyor itself, even though more carriers would physically fit.

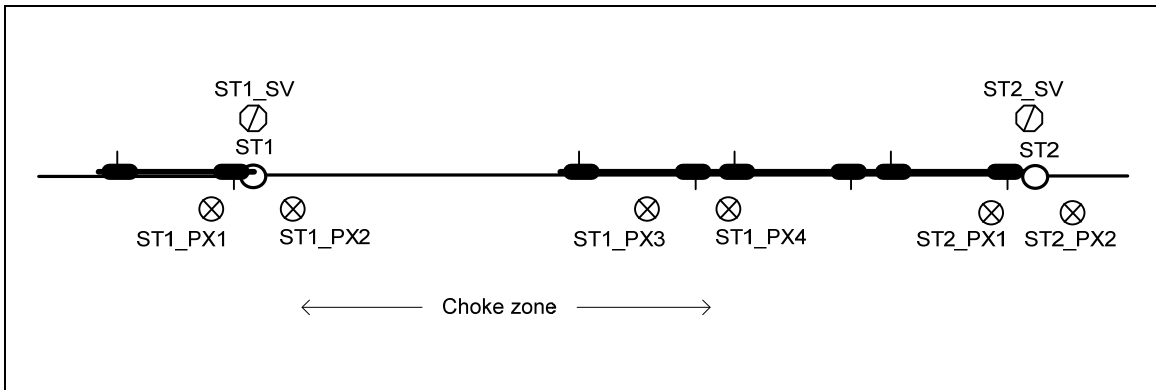


Figure 9-4 Power and Free Choke Zone

Figure 9-5 shows the logic for a choke zone that allows only one carrier in the zone.

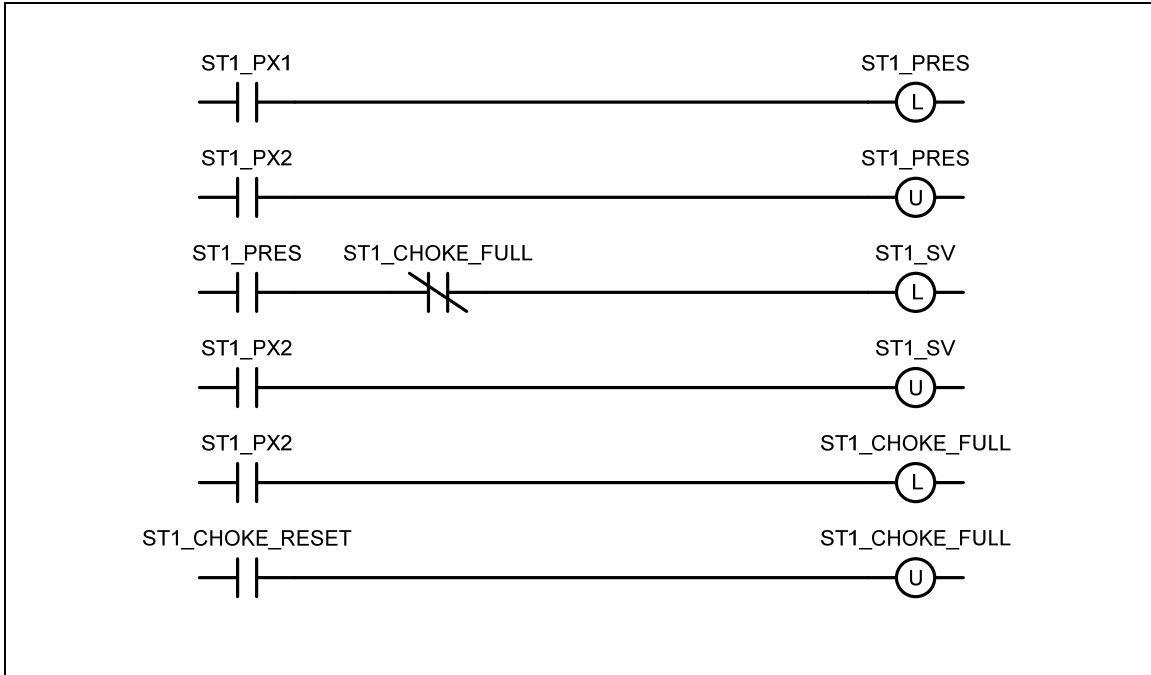


Figure 9-5 Power and Free Choke Zone Logic

In order to clear the choke zone ST1_PX3 must be made and then ST1_PX4. The idea is to prevent the choke zone from being inadvertently cleared by a single sensor. Let's assume that as the carrier makes ST1_PX4, it is then accumulated behind another carrier. In this situation if ST1_PX4 were the only device that cleared the choke zone then a new carrier entering the choke zone would be cleared by the carrier ahead of it. Carriers will also have a tendency to rock backward when they are accumulated. This would prevent us from solving the problem by using a one shot instruction and clearing the choke zone solely on the one shot of a single sensor. A mechanical anti-backup device between ST1_PX3 and ST1_PX4 prevents the carrier from rolling back to ST1_PX3 after having made ST1_PX4. In Figure 9-6 ST1_CHOKE_RESET is made for one program scan. It should be noted that we don't care how many times a sensor is made in order to make a zone full. If the sensor is being made then the zone must be full.

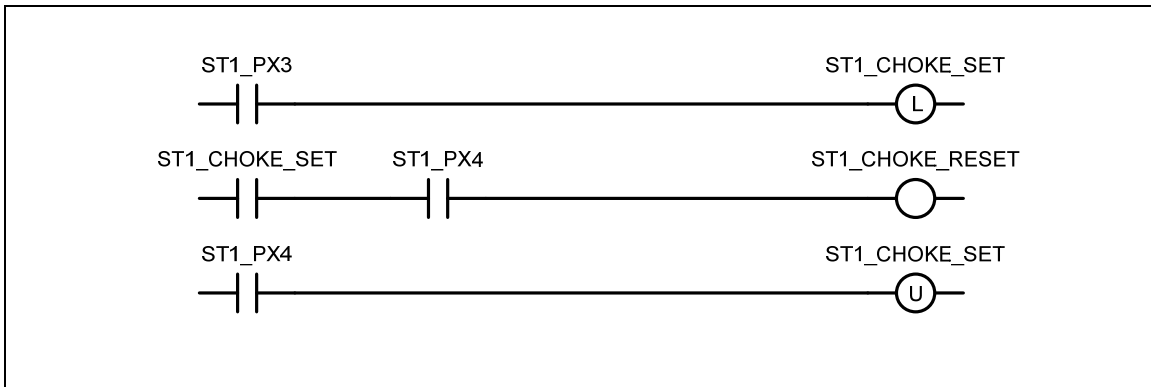


Figure 9-6 Power and Free Choke Zone Reset Logic

Power and Free, Count Zone

In Figure 9-7 Stop 2 is designed to hold 3 carriers. A counter is used in the program to keep track of the number of carriers in the stop.

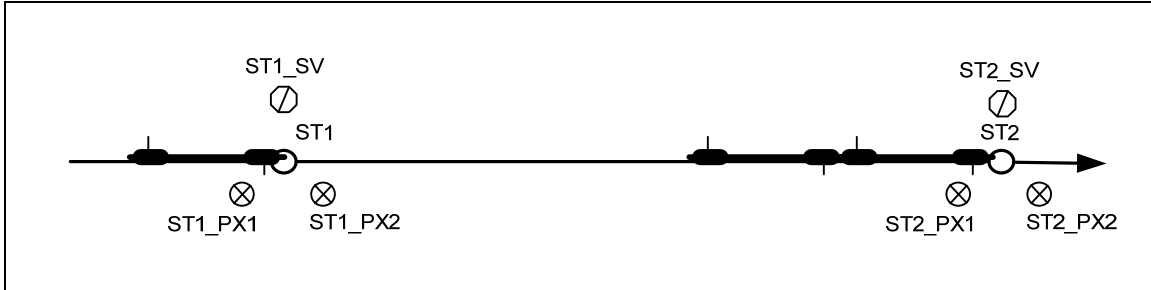


Figure 9-7 Power and Free Count Zone

In Figure 9-8 the count zone is incremented when stop 1 is opened and decremented when the carrier leaves stop 2. Again, the rung order is important. We could not have put ST1_PX2 in the count up rung unless we moved this rung above the unlatch for ST1_SV. Also, we use ST1_SV to increment the count zone in order to make use of the sequence of hitting ST1_PX1 and then ST1_PX2. Unlike the choke zone, if we receive multiple signals from a single device at the entry of the zone, the counter will increment for each transition of the input causing an error in the count. We could also have used ST2_SV to decrement the zone. However, if the stop failed to open when the solenoid was energized, then an extra carrier would be let into the zone. This may cause a conveyor jam or some other mechanical problem if the zone was not designed for this extra carrier.

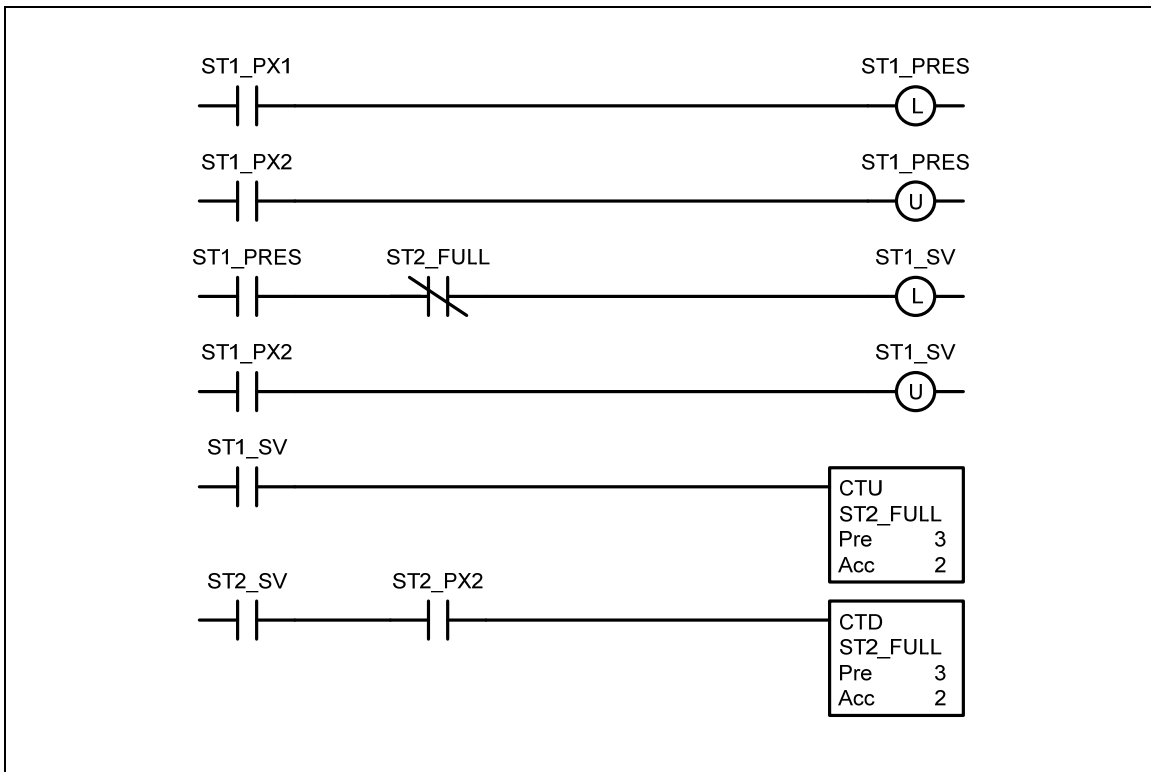


Figure 9-8 Power and Free Count Zone Logic

Power and Free, Merge into a count zone

Figure 9-9 shows a merge from 2 tracks into 1. We will assume that both tracks are powered by the same chain. The chain is shown with the dashed line. At the merge, one chain must peel out. As you can see it is important that you don't release more than one carrier into the merge. If you do, you may be asked to hone your skills with a very heavy crow bar. MRG1_PX2 is positioned so that it is tripped by the trailing trolley of the carrier. The rear trolley actuator is on the opposite side of the track from the front trolley actuator. Because we are using only one device to clear the merge, it must be certain that the accumulation of carriers can not cause MRG1_PX2 to be maintained. We will assume that the carriers are ready to release as soon as they enter the stop. We will alternate the release of the stops 1 and 2 in a first come first serve fashion. There are 2 zones defined. The first zone is the merge. The merge is defined between each stop and MRG1_PX2. The second zone is the count zone behind stop 3. The merge and the count zone overlap. The designation of PX2 is assigned to the merge prox because this is commonly used as the zone clear, not because it is the second device in the merge.

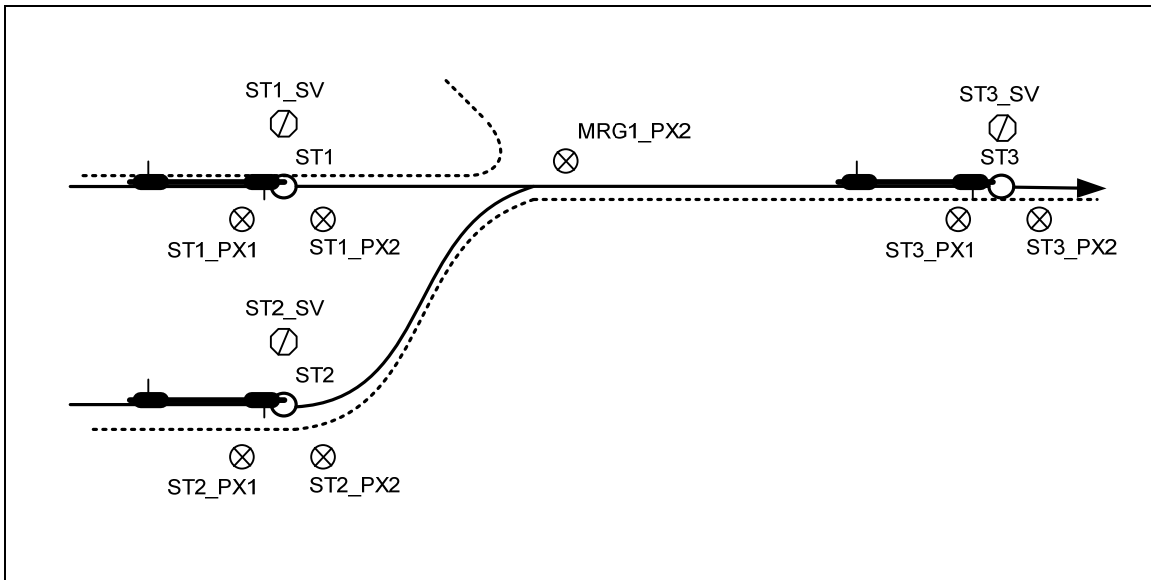


Figure 9-9 Power and Free Merge into a Count Zone

Chapter 9 Zone Control

Figure 9-10 shows the logic for stop 1. I have shown the “First” logic for stop 1 and 2 here. However, it does not matter where you put it. Refer to Chapter 7 Determining Priority, for a discussion on first logic.

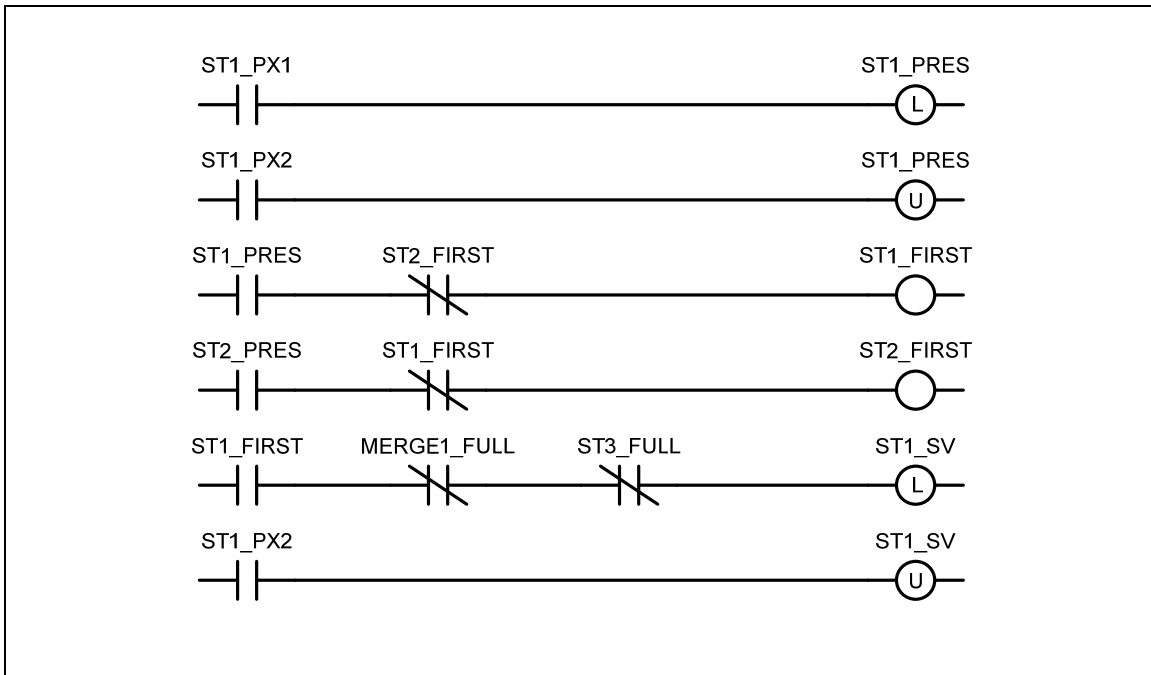


Figure 9-10 Power and Free Merge Stop 1 Logic

Figure 9-11 shows the logic for stop 2. We don't have to program the “First” logic since we included it in the logic for stop 1.

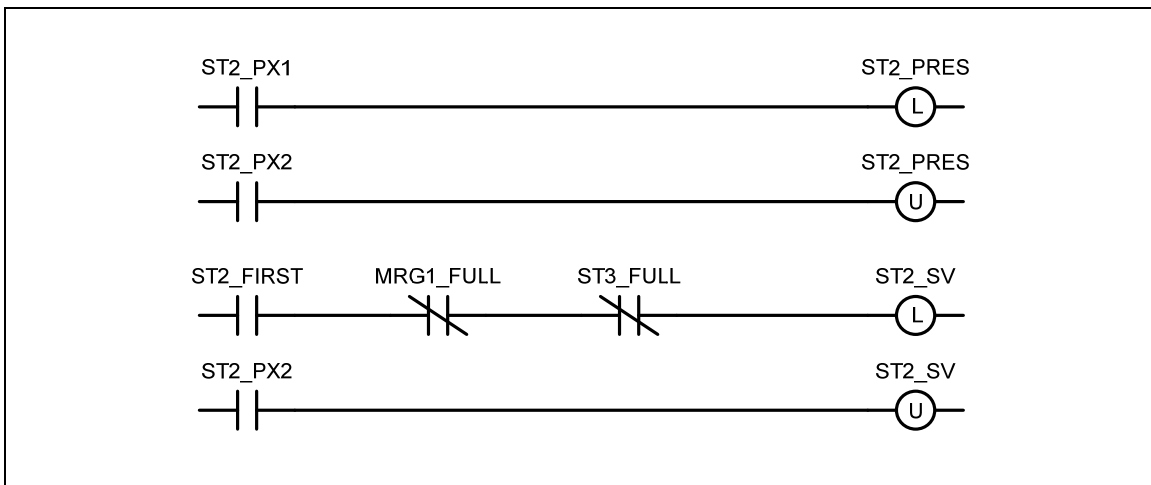


Figure 9-11 Power and Free Merge Stop 2 Logic

The logic for the merge full is shown in Figure 9-12. The merge is treated like a choke zone. The merge zone is cleared with MRG1_RESET which is similar to the choke zone reset. You can also see that either stop will be closed for some amount of time before the merge clears. This will allow the rung with the CTU instruction to transition from false to true which is required for the CTU instruction to work properly.

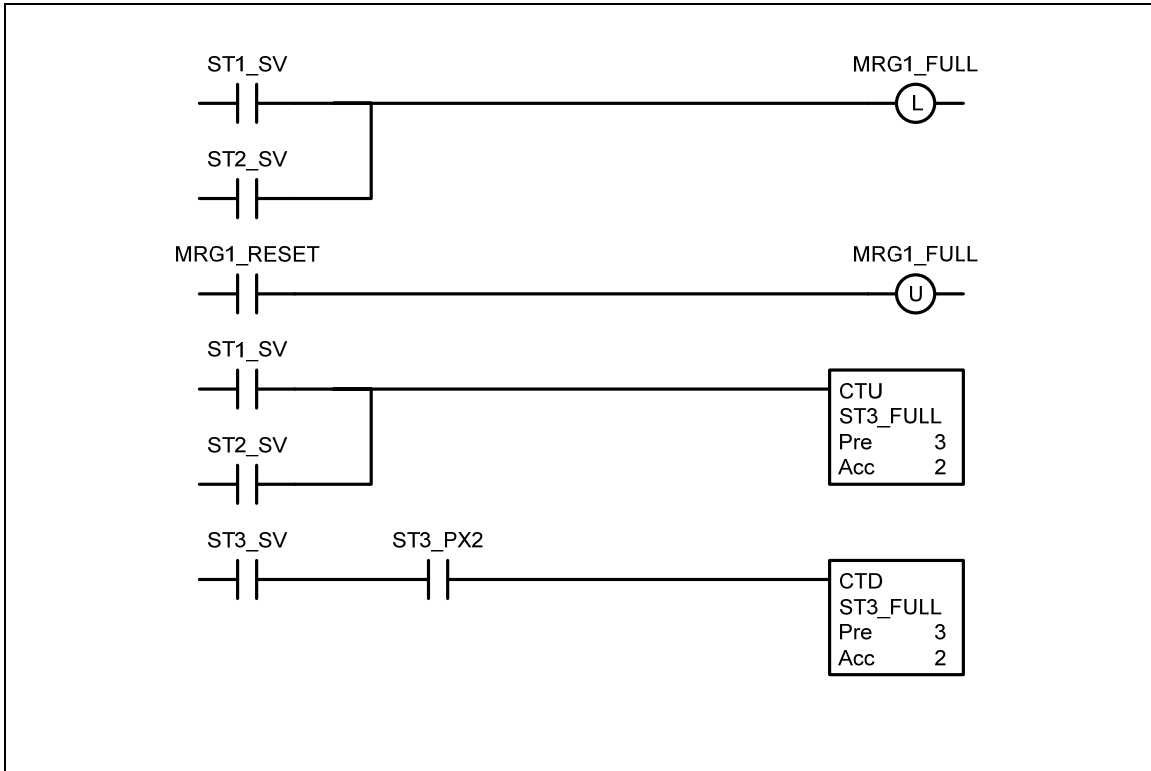


Figure 9-12 Power and Free Merge Full and Down Stream Count Zone

In Figure 9-13 the merge reset is programmed in a similar way as a choke reset.

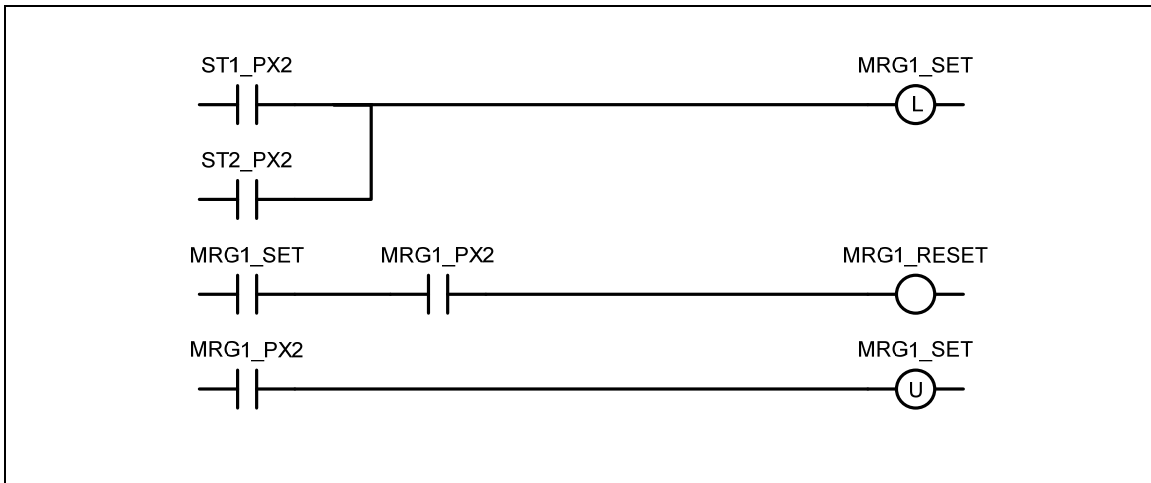


Figure 9-13 Power and Free Merge Full Reset

It would be a good idea to check that MRG1_PX2 is not made when the either stop 1 or stop 2 clears. Figure 9-14 shows the logic for a merge fault. This kind of fault would stop the chain.

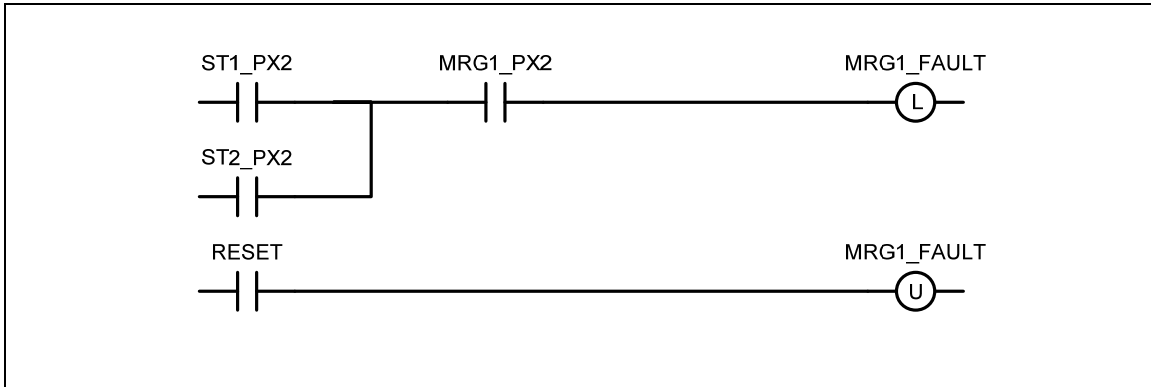


Figure 9-14 Power and Free Merge Fault

Power and Free, Track Switch

Figure 9-15 shows a power and free track switch into 2 choke zones. The switch is powered by a double solenoid with 2 outputs SW1_SVR, and SW1_SVL. The left and right position of the track switch can be verified with limit switches SW1_LSL, and SW1_LSR. SW1_PX1 is used to ready the switch clear proxes in the same way we readied the choke zone clear with a PX3. The operator will determine which direction the carrier will go by pushing the send carrier to the left pushbutton ST1_PBL or to the right with the ST1_PBR push button. The carrier at stop 1 will remain there until one of these buttons is pressed. There are 3 down stream zones, the switch full zone, the choke left full, and the choke right full.

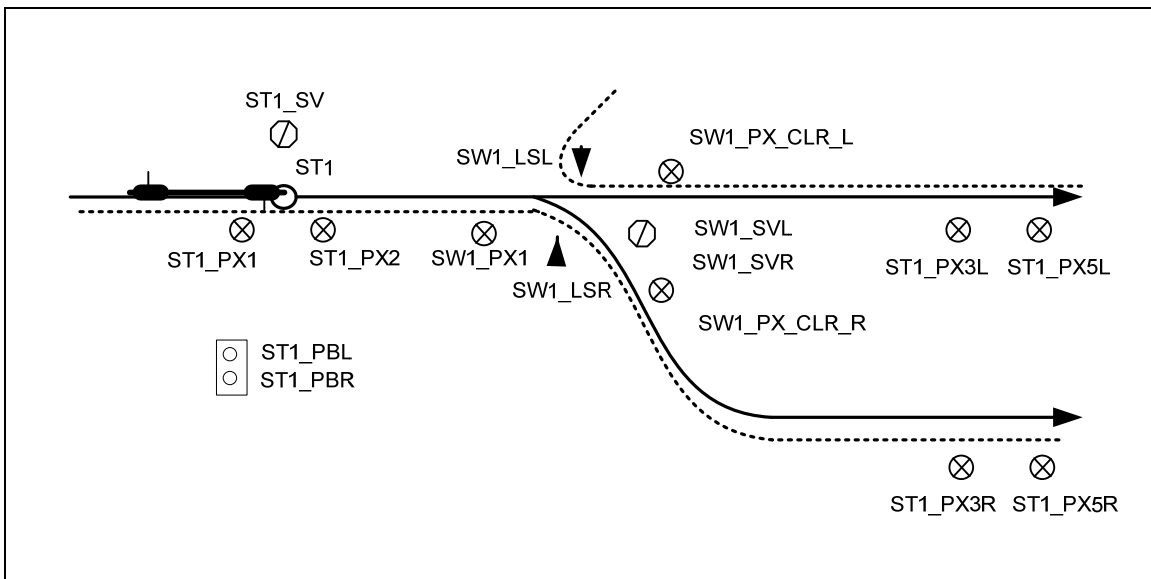


Figure 9-15 Power and Free Track Switch

Figure 9-16 shows the ladder logic for the carrier present and the push button latches. The push button latches determine the destination of the carrier. The operator can press one button and then change his mind and press the other button. The final destination is set once the track switch is set and the stop is opened.

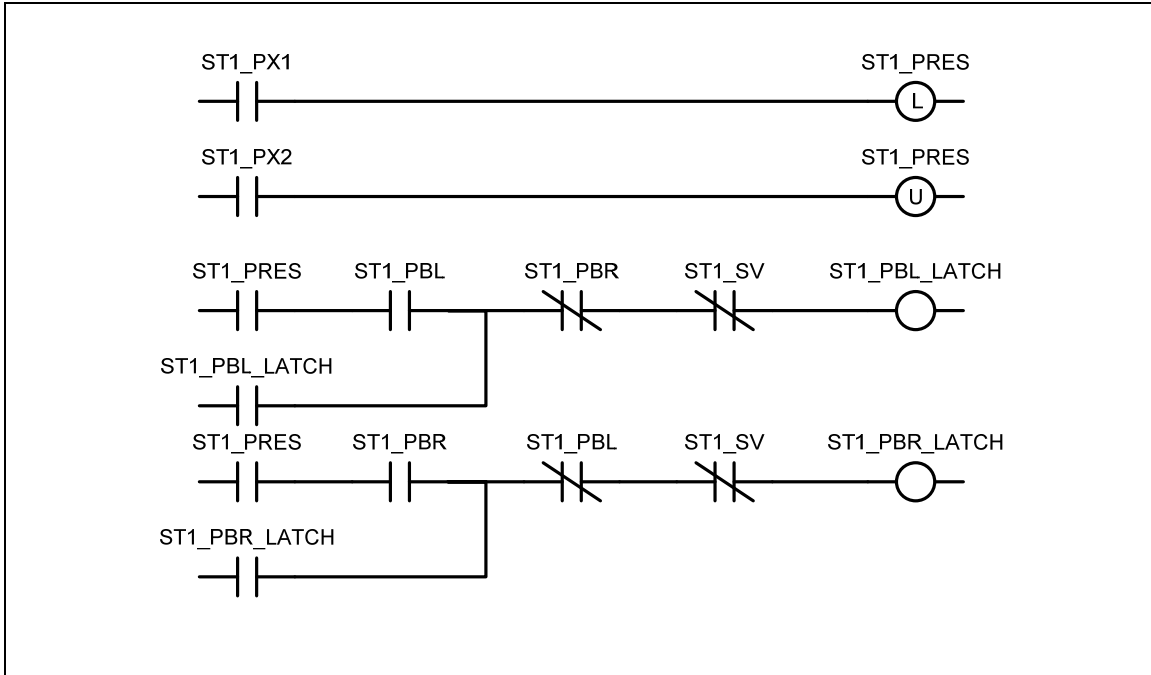


Figure 9-16 Power and Free Track Switch Push Button Left and Right Latches

Figure 9-17 shows the ladder logic for stop 1 solenoid. The track switch position is verified before the stop is opened. If sending one carrier right and then the next carrier to the left, the second carrier can leave the stop as soon as the first carrier clears the track switch and the track is switched to the second position. If two consecutive carriers are going to the same destination then the first carrier must clear the choke zone before the second carrier is released.

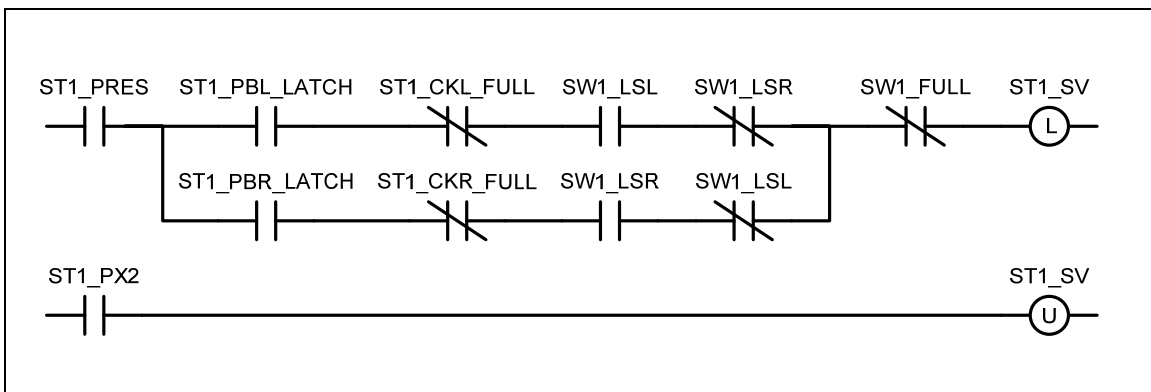


Figure 9-17 Power and Free Track Switch Stop Logic

Chapter 9 Zone Control

Figure 9-18 shows the logic for the switch full zone. The switch clear could be programmed several ways. Separate right and left clear bits could have been used. I could have included the switch position limits SW1_LSL and SW1_LSR in the clear logic. I can not come up with a good reason why I would or would not use them. I think the code that I have written here is sufficient to get the job done.

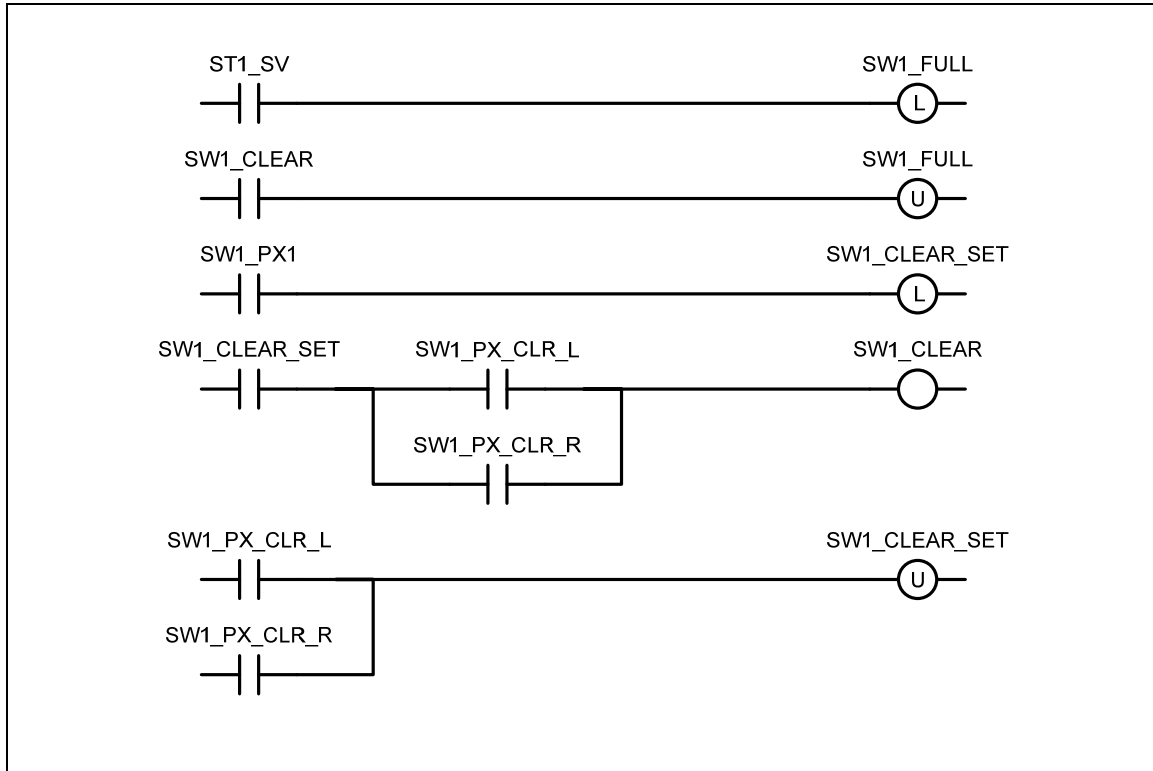


Figure 9-18 Power and Free Track Switch Full and Clear

Figure 9-19 shows the logic for the track switch solenoids. Some programmers may have latched the outputs until the carrier clears the track. I don't think this is necessary.

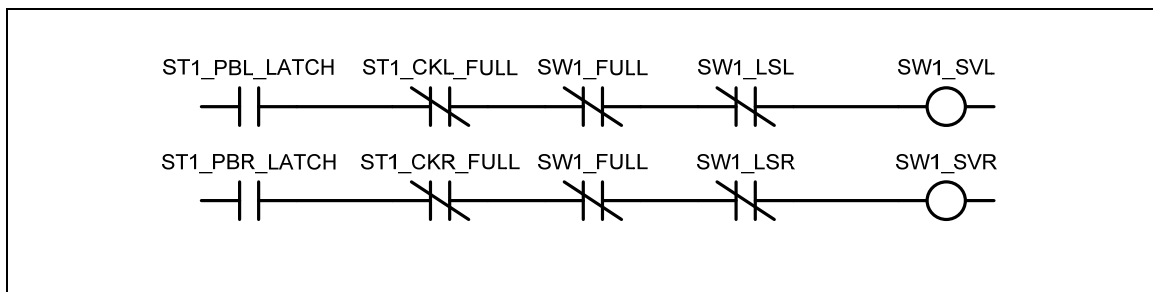


Figure 9-19 Power and Free Track Switch Left and Right Solenoid Valves

Figure 9-20 shows the logic for the left choke zone. The right choke zone is programmed the same way.

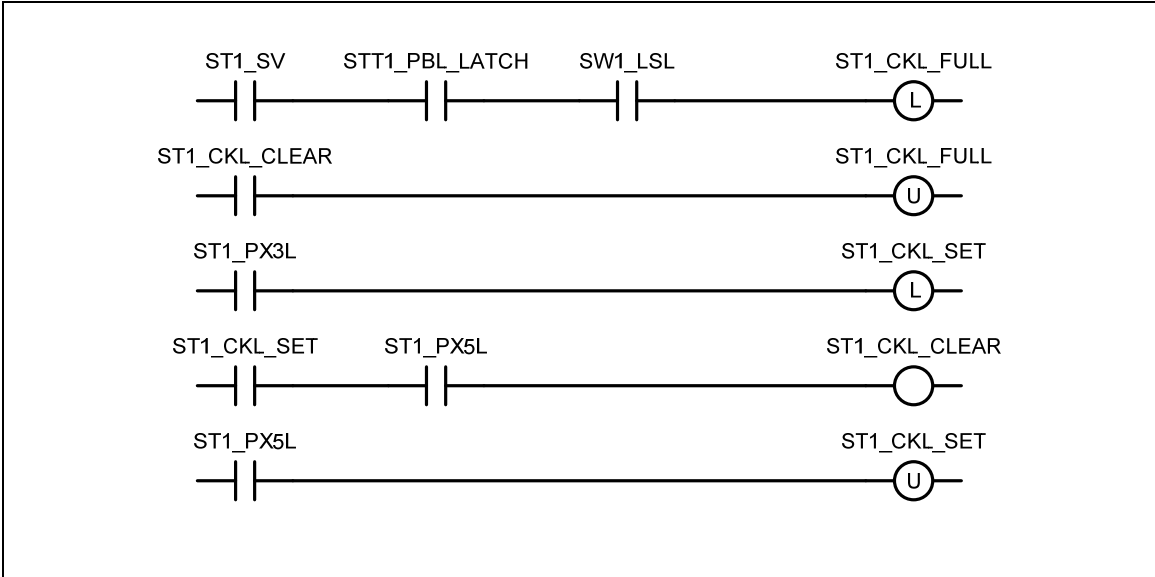


Figure 9-20 Power and Free Track Switch Down Stream Choke Left and Right Full

Chapter 10 Tips and Tricks

Toggle Push Button

In Figure 10-1 the START latch changes state with each push of the start/stop push button. This is a simple circuit that will allow you to use one push button to start and stop a motor.

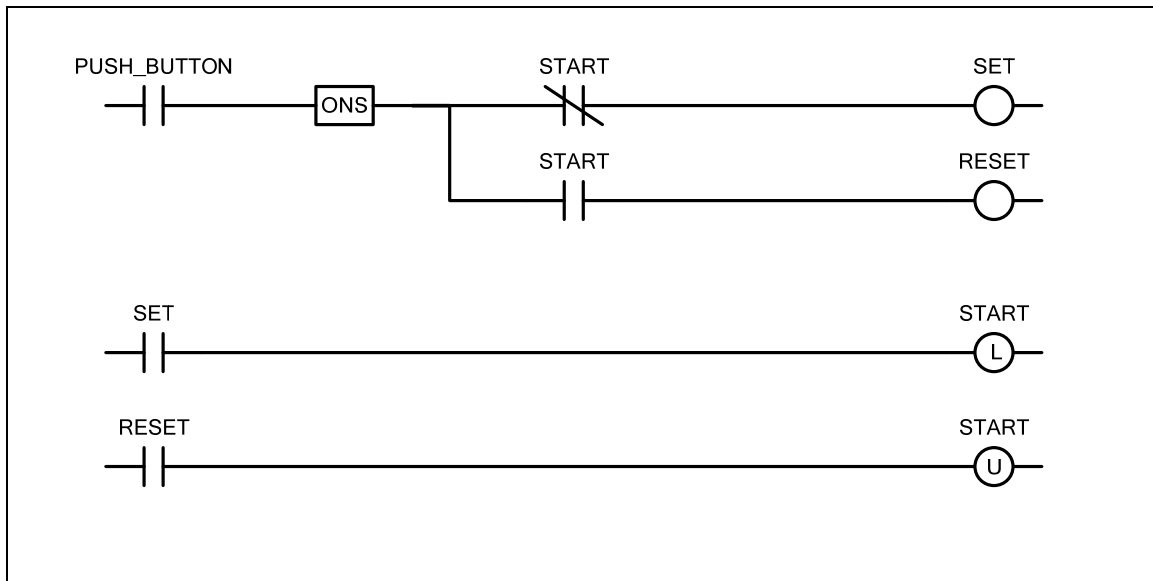


Figure 10-1 Toggle Push Button

Cascading Start Stop

Figure 10-2 shows how a continuous stream of conveyors can be started and stopped in a cascade fashion. Conveyor 1 is the first conveyor in the stream. It feeds conveyor 2. Conveyor 2 feeds conveyor 3 and so on until conveyor (N) is reached. Conveyor 1 is the most upstream conveyor and would be considered the feed conveyor. Conveyor (N) is the most downstream conveyor. When CONVEYOR_START is turned on each TOF is enabled until conveyor (N) starts. When conveyor (N) is started, the CONV(N)_TON is enabled and begins timing. When the timer is done, the next upstream conveyor is started. This continues until conveyor 1 is started. Each conveyor starts one after the other in a cascading fashion starting with the most downstream conveyor to the most upstream. This will prevent a surge in the power required to start the system.

Now, if CONVEYOR_START is turned off, conveyor 1 is immediately turned off and stops the feed. CONV1_TOF begins timing and when the timer completes Conveyor 2 is turned off. The conveyors will continue to turn off in the reverse order that they were started until the most downstream conveyor is stopped. This will allow the conveyor to be cleaned out. The TOF presets should be set to the time it takes to clean the conveyors out. If an AUX fault occurs on a conveyor in the middle of the stream, the faulted motor and all of the upstream conveyors are immediately turned off. Since the AUX fault is in the TOF circuit, the downstream conveyors

will begin to shut down. If we had not wanted the downstream conveyors to cascade off after the fault, we would put the fault in the TON circuit.

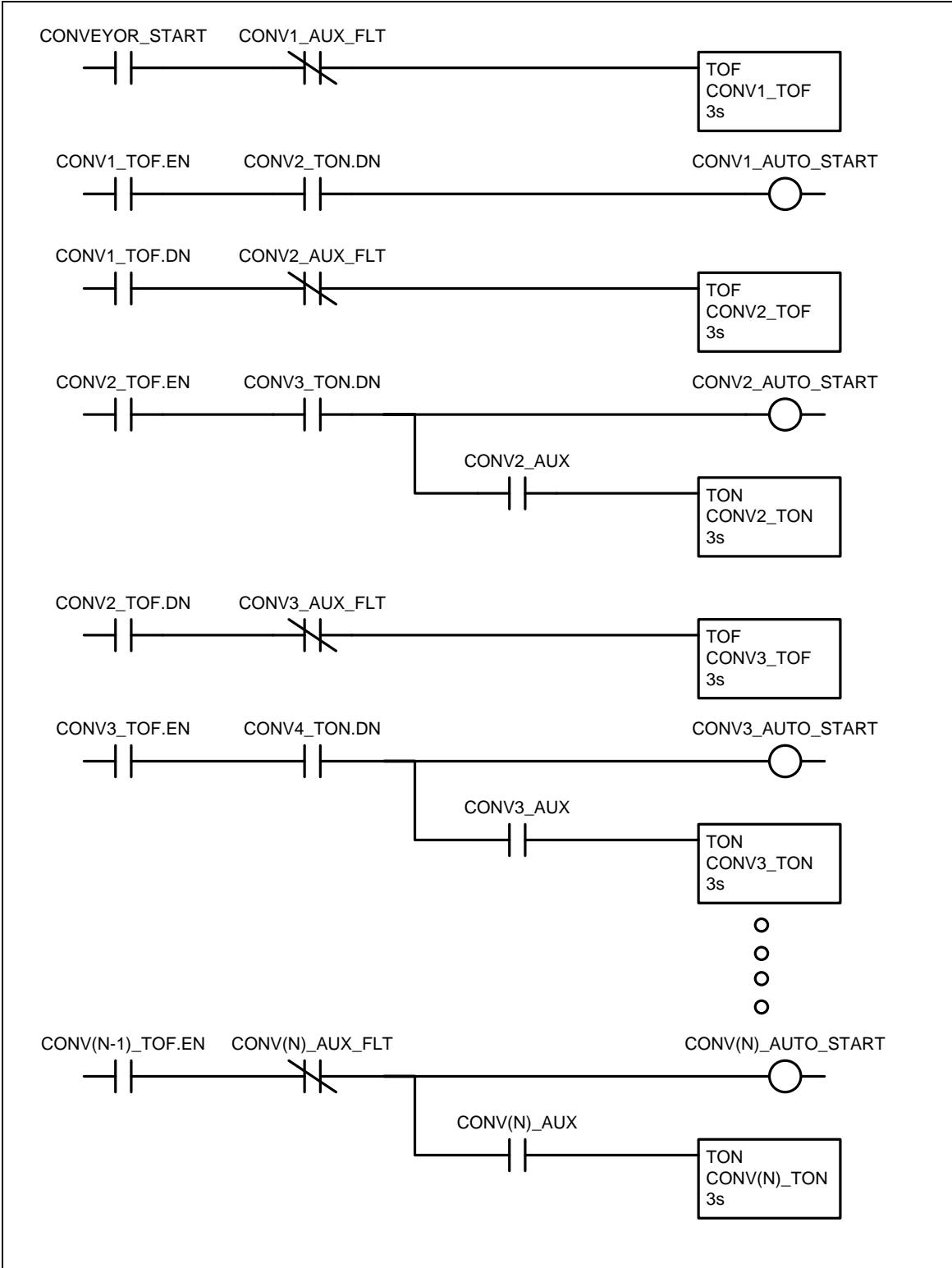


Figure 10-2 Cascading Start and Stop Logic

In Figure 10-3 the loss of INTERLOCK_1 will shut off conveyor 3 immediately. The upstream conveyors will then shut off. The downstream conveyors will cascade off with the TOF timers. When INTERLOCK_2 is off only conveyor 3 and the upstream conveyors are shut off.

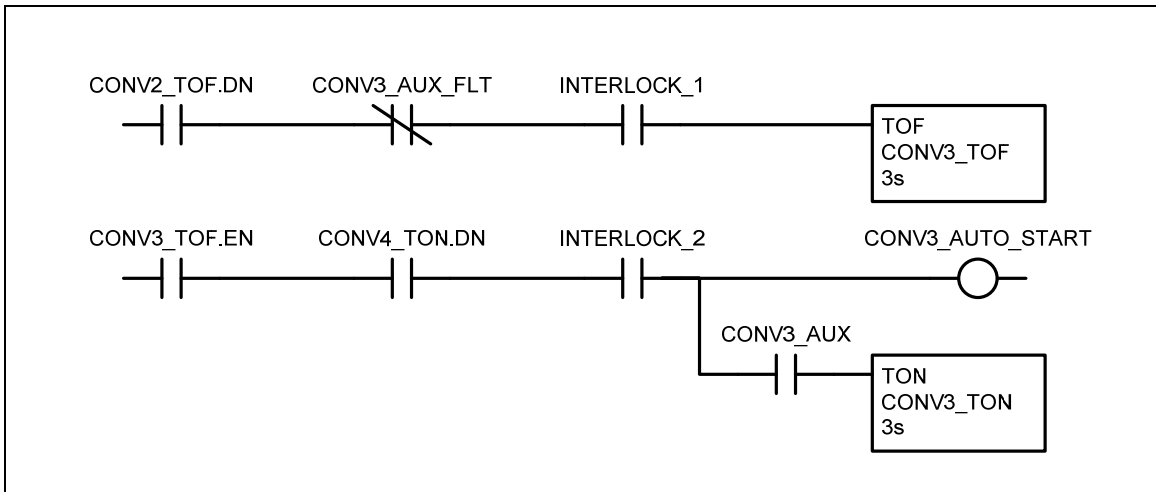


Figure 10-3 Cascading Start Stop Interlocks

Figure 10-4 shows how the downstream shut down time can be modified depending on the status of an interlock. This will still allow the conveyors to cascade off but the next downstream conveyor will shut off faster.

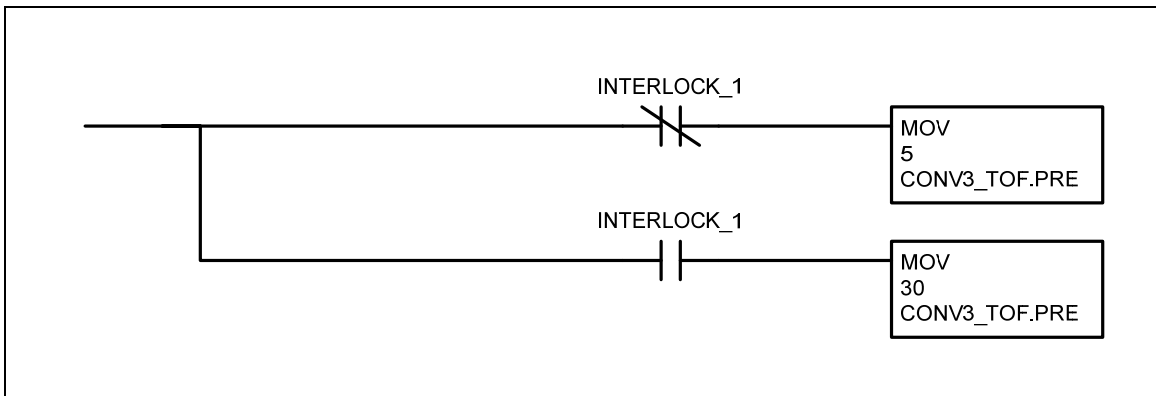


Figure 10-4 Cascading Start Stop Fast Shutdown

A simple message display

In this example we will program a message display that uses 16 discrete outputs connected to the PLC. An additional strobe output will cause the message to be displayed. The message display stores messages such as alarms. Each message is numbered. The stored message is displayed by setting the message display inputs to the message number, and then turning on the strobe input. If multiple alarms are active, then each alarm is displayed for 5 seconds.

In Figure 10-5 we increment a message pointer, MSG_PTR, if we are not currently displaying a message. MSG_NUM will be equal to 0 if no message is displayed. The pointer will increment by one each program scan. When the pointer exceeds the last valid message number then the pointer is reset back to 0.

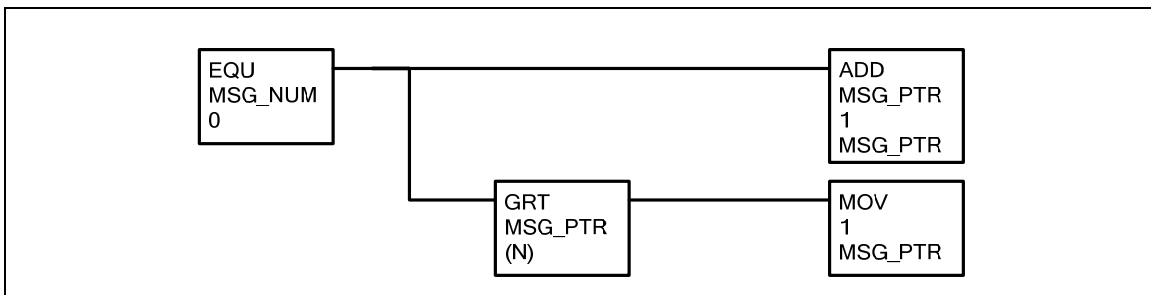


Figure 10-5 Message Display Increment the Message Pointer

Each alarm is assigned a number. When the message pointer is equal to that number and the alarm is on, then the pointer is moved to the message number. In Figure 10-6 this is repeated for each alarm.

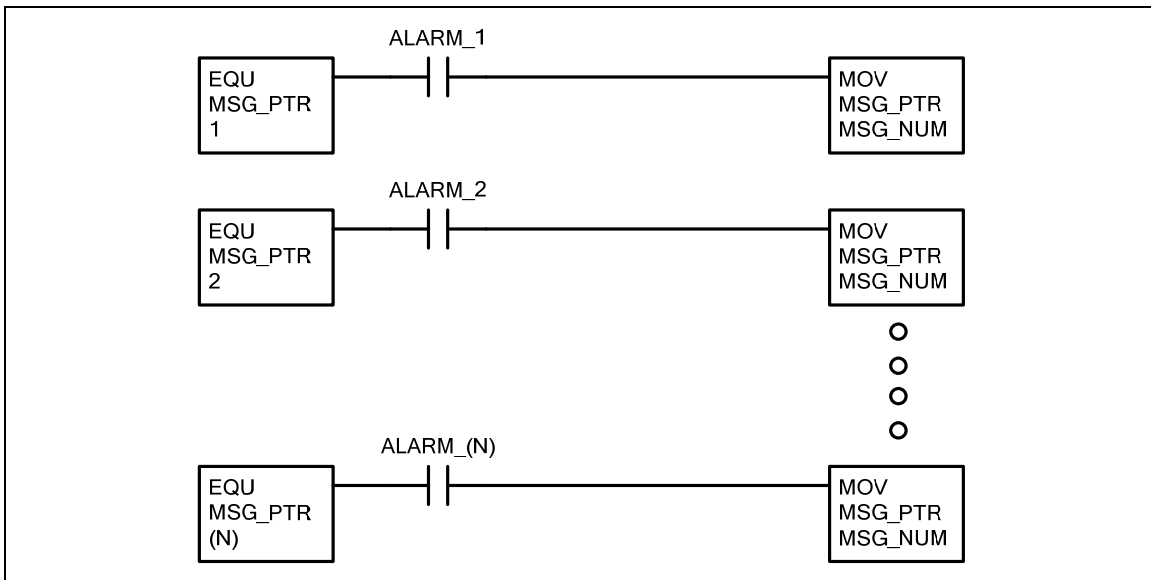


Figure 10-6 Message Display Check the Alarm and Set the Message Number

In Figure 10-7 when MSG_NUM is not equal to zero, then we display the message. We set the message display outputs to MSG_NUM. Then when timer MSG_TM1 is done we turn on the strobe output. This will cause the message to be displayed on the message display. The strobe remains on until timer MSG_TM2 is done. Then, we turn off the strobe and set the outputs to zero. We all allow the message outputs to remain at off until timer MSG_TM3 is done. Then we clear the MSG_NUM which will allow the next message to be displayed.

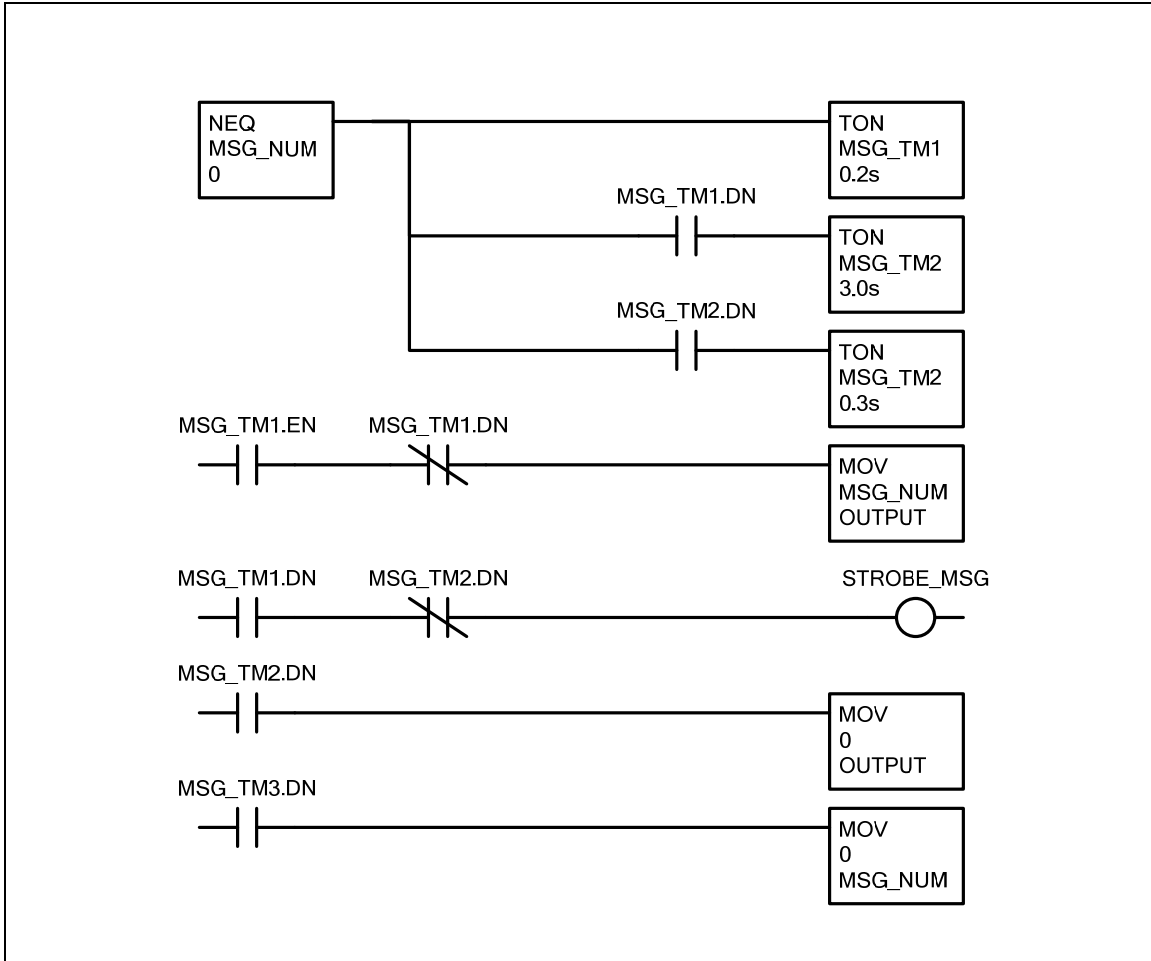


Figure 10-7 Display the Message

Buffering Transactions

There are two common ways in which data is collected from a PLC and then saved to a database, time based and Event based. In the time based method, data points are sampled on specified time intervals. Most HMI applications include some form of data logging functions that are time based. The second method is event based. In this method we want data transferred from the PLC when a specific event occurs. This event could be, when a batch is complete for example. Data is collected uninterrupted until some maintenance function needs to occur on the computer that is collecting the data. When this occurs data is not collected during the time that the computer is offline. To prevent this scenario we need to buffer the data in the PLC. The example that we will consider will buffer event based data in the PLC.

Let's assume that we have two blending hoppers. Coffee is feed into a blend hopper until the desired amount is reached. After the scale settles, the actual coffee amount, along with other supporting information, needs to be sent to a database. Each blend hopper functions independently from the other and the same type of data will be collected from each.

In Table 10-1 we describe the user defined data type for the blend data.

<i>UDT Member</i>	<i>Description</i>	<i>Type</i>
BLEND_NAME	Blend name	BLEND_STR
ROASTER	Roaster number	Dint
ROAST_NUMBER	Roast number	Dint
SILO	Source silo	Dint
WEIGHT	Weight from the silo	Real
TIME	Time and date stamp	DATE_TIME
LAST_FEED	Flag indicating that this is the last feed for this blend	Boolean

Table 10-1 User Defined Data Type BLENDING_XACT

In Table 10-2 we define the variables used to buffer and upload the data to the database.

<i>Tag</i>	<i>Description</i>	<i>Type</i>
BLEND_1_COMPLETE	Silo feed is complete for blend scale 1	Boolean
BLEND_1_COMPLETE_ONS	ONS	Boolean
BLEND_1_DATA	Transaction data for blend scale 1	BLENDING_XACT
BLEND_2_COMPLETE	Silo feed is complete for blend scale 2	Boolean
BLEND_2_COMPLETE_ONS	ONS	Boolean
BLEND_2_DATA	Transaction data for blend scale 2	BLENDING_XACT
BLEND_UPLOAD_READY	Transaction data is ready for database	Boolean
BLENDING_BUFFER	Transaction data for blend scale 2	BLENDING_XACT[100]
BLENDING_FIFO	Pointer FIFO	DINT[100]
BLENDING_FIFO_CTRL	FIFO control variable	CONTROL
BLEND_FIFO_LOAD	Input parameter to subroutine	BLENDING_XACT
BLEND_FIFO_UNLOAD	FIFO unload and transaction for upload	BLENDING_XACT
BLENDING_FIFO	Pointer FIFO	DINT[100]
BLENDING_XACT_PTR	Pointer into BLENDING_BUFFER	DINT
DUMMY	Dummy output for subroutine instruction	Boolean

Table 10-2 Buffering Transaction Tag Definition

In Figure 10-8 we determine the size of the transaction buffer. The size will then be used in the program. This will allow us to change the dimension of the `BLENDING_BUFFER[]` array without having to change the ladder logic. We can set the buffer size to utilize most of the unused memory in the PLC. To free up memory in the future, we only have to change the size of the `BLENDING_BUFFER[]` array and the `BLENDING_FIFO` array.

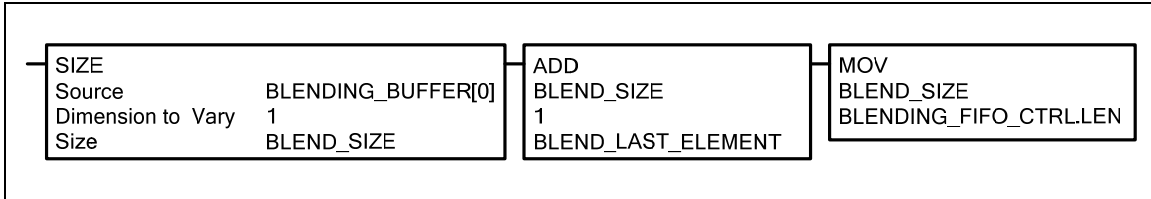


Figure 10-8 Buffering Transactions Set the FIFO Size

In Figure 10-9, when a blend is complete the current data for that blend is held in `BLEND_1_DATA`. The data is used as the input parameter to the `BLEND_UPLOAD` subroutine.

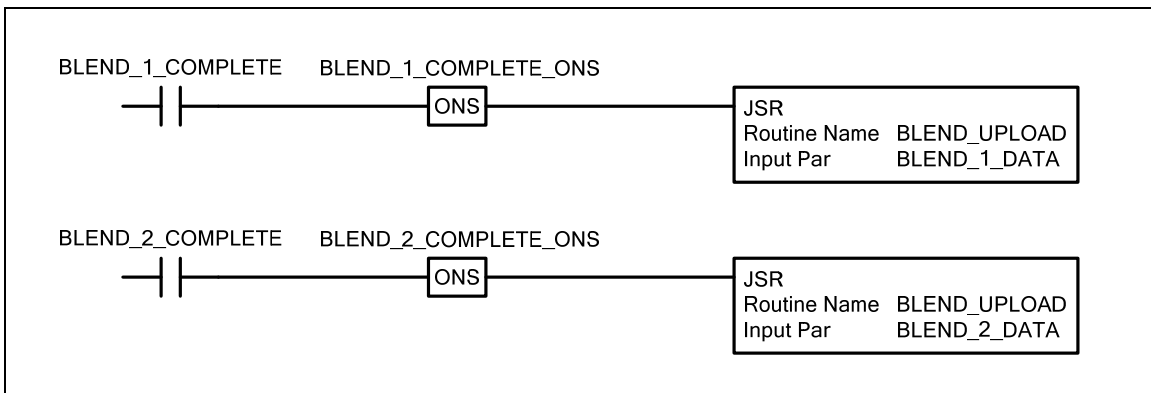


Figure 10-9 Buffering Transactions Call the `BLEND_UPLOAD` Routine

In Figure 10-10 we start the first rung of the `BLEND_UPLOAD` subroutine. This routine will load the transaction into the `BLENDING_BUFFER[]` array and then load array pointer into the `BLENDING_FIFO[]` array.

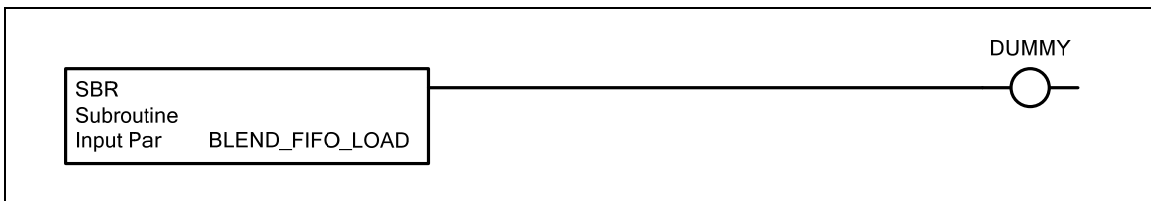


Figure 10-10 `BLEND_UPLOAD` Routine Receive the Input Paramters

Figure 10-11 increments the transaction pointer

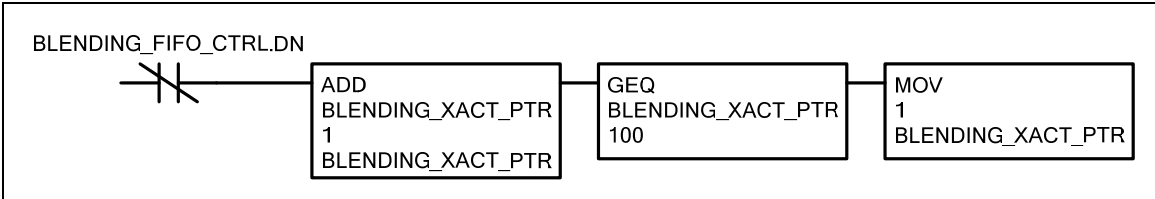


Figure 10-11 BLEND_UPLOAD Routine Increment the Transaction Pointer

In Figure 10-12 the blending data which is now in the variable BLENDING_FIFO_LOAD, is copied into the BLENDING_BUFFER[]. The pointer to this transaction is then loaded into the FIFO. This is the last rung in the routine.

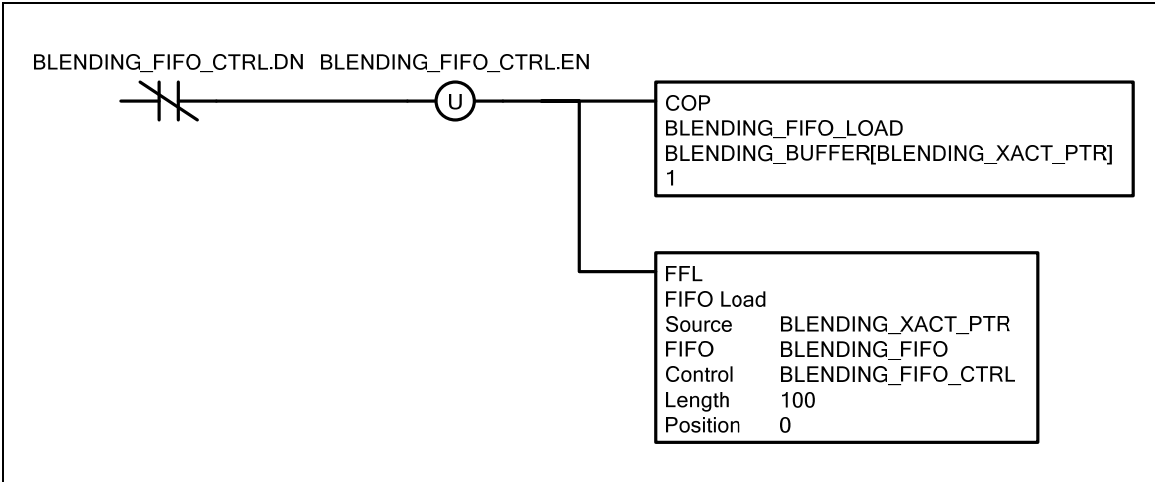


Figure 10-12 End BLEND_UPLOAD Load the Transaction Table and FIFO the Pointer

The pointer is unloaded from the FIFO in the main routine. In Figure 10-13 we unload the FIFO and set the BLEND_UPLOAD_READY bit. This indicates to the HMI or database program that data is ready to be uploaded from the PLC into the database. The HMI will reset this bit when the data has been stored in the database. Then the next pointer can be unloaded from the FIFO.

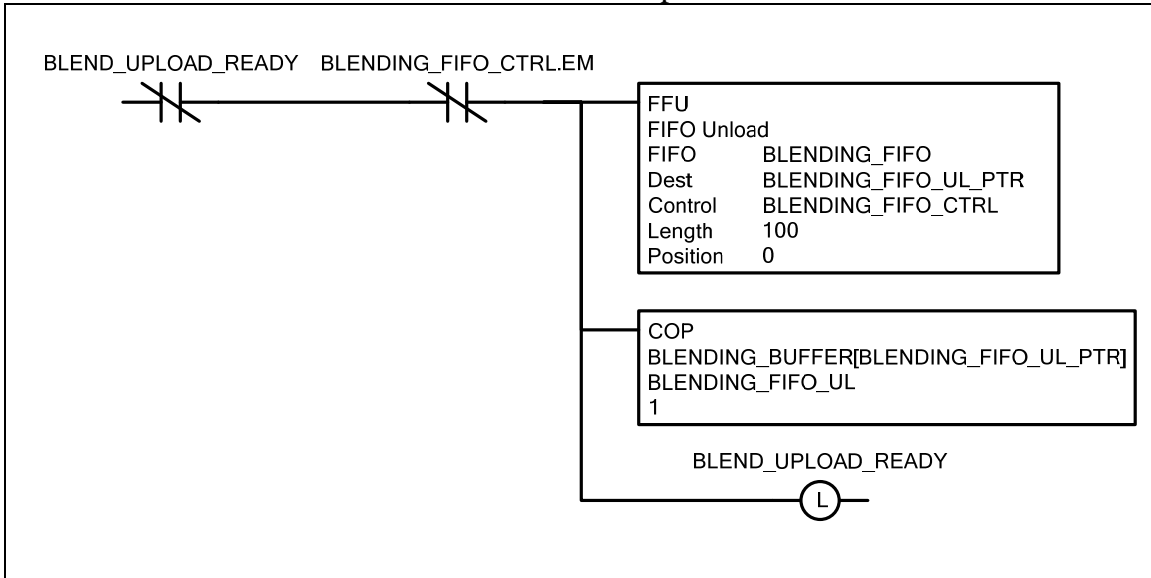


Figure 10-13 Buffering Transactions Unload the Stack Pass the Transaction

Appendix A Binary Numbers

One problem with writing a technical book like this, is trying to determine what the expertise of the reader is. Should I write a book on programming techniques without providing the background information that is required to understand the more advanced concepts? A colleague of mine passed on a great quote that he had heard. “There are ten kinds of people that understand binary, those that do and those that don’t. For those that do understand, there are only two kinds.” Now its time for you to join those of us that do understand.

A light bulb can have two states on or off. These states can be represented by a single digit with a value of 1 or 0. If we group these binary digits together we can represent a number besides one or zero by assigning each combination of the two digits to a number. In Table A-1 we can see that there are 4 combinations of numbers that can be represented by two binary digits.

Binary	Decimal
0 0	0
0 1	1
1 0	2
1 1	3

Table A-1

These combinations can be represented by their number equivalent.

In Table A-2 we can expand the combinations to 8 when we include an additional bit.

Binary	Decimal
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

Table A-2

As we increase the number of bits, the number of combinations also increase. A group of 16 bits is called a word. Each bit in a 16 bit binary word represents a power of two. Two to the third power (ie. 2^3) is $2 \times 2 \times 2 = 8$. In Table A-3 the value of each bit in the binary 16 bit word is shown. Bit 15 is the sign bit. It determines if the value is positive or negative. If the value is negative then the bits 0 to 14 are expressed in the two’s compliment form. The two’s compliment is determined by reversing the value of each bit and then adding 1 to the entire word.

	Sign	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Dec	+,-	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table A-3

Ladder logic Programming Techniques By Duane Snider

The decimal equivalent for the binary number 0110 0001 0100 0001 is the sum

Dec	+, -	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	
Bit		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary		0	1	1	0	0	0	0	1	0	1	0	0	0	0	0	1

Table A-4

$$(16384 + 8192 + 256 + 64 + 1) = 24897$$

Appendix B Useful Calculations

This is not a complete manual on all engineering calculations. However, I have ran across a few that I perform more often than others. So, I will list them here.

Unit Conversions

Stoichiometry is one of the most useful things I learned in my freshman chemistry class. It is the math used in chemistry. It includes methods for unit conversions. If two values are equal to each other such as 1 gallon is equal to 4 quarts then the result of dividing one value by the other is one. We can then multiply this ratio with anything without affecting the true value of what we started with. Let me give you an example.

Lets say we want to convert 5 gallons of milk to pints.

We start with the basic conversions.

1 Gallon = 4 Quarts 1 Quart = 2 pints

We then write these in their ratio form.

Put the units that we will be canceling on the bottom.

$$\frac{4 \text{ Quart}}{1 \text{ Gallon}} \quad \text{And} \quad \frac{2 \text{ Pint}}{1 \text{ Quart}}$$

Take what you want to convert, in this case 5 gallons.

Then multiply that by the ratio(s).

$$\frac{5 \text{ Gallon}}{1} \times \frac{4 \text{ Quart}}{1 \text{ Gallon}} \times \frac{2 \text{ Pint}}{1 \text{ Quart}}$$

Now cancel the units on the top with the matching ones on the bottom.

The units that are left are your resulting units.

$$\frac{5 \text{ Gallon}}{1} \times \frac{4 \text{ Quart}}{1 \text{ Gallon}} \times \frac{2 \text{ Pint}}{1 \text{ Quart}}$$

We then multiply all of the top numbers together and the bottom numbers together.

$$\frac{5 \text{ Gallon}}{1} \times \frac{4 \text{ Quart}}{1 \text{ Gallon}} \times \frac{2 \text{ Pint}}{1 \text{ Quart}} = \frac{5 \times 4 \times 2 \text{ Pint}}{1 \times 1 \times 1} = 40 \text{ Pint}$$

Figure B-1

Figure B-2 shows how we can calculate the distance a part moves during 1 program scan.

We want to find out how far a part moves down a conveyor in 1 program scan.

We know that the program scan time is 20ms and the conveyor speed is 120fpm
 1 scan = 20 ms 120 ft = 1 min

$$\frac{20 \text{ ms}}{1 \text{ scan}} \times \frac{1 \text{ sec}}{1000 \text{ ms}} \times \frac{1 \text{ min}}{60 \text{ sec}} \times \frac{120 \text{ ft}}{1 \text{ min}} \times \frac{12 \text{ in}}{1 \text{ ft}} = \frac{0.48 \text{ in}}{\text{scan}}$$

So, the part moves almost ½ inch in 1 program scan.

Figure B-2

You have to multiply the inches/scan by the number of program scans that it would take to decide what to do after reading a sensor. You can then determine the reaction distance from the time that you sense a part until you can react to it. Also there are mechanical reaction times for any solenoid or motor that would have to be added.

In this example we calculated the reaction time. But what we are really interested in is repeatability. If we assume that the mechanical reaction times are consistent then we can delete them from the equation. We can compensate for them by adjusting the location of the sensor. We can not however, cancel out the scan time even though it may be very consistent. The reason is, the sensor can be activated any time during the program scan and the inputs are introduced to the logic at the beginning of the program scan. We can cancel out additional program scans from the repeatability as long as it takes the same number of scans to react to the input, and the scan time is reasonably consistent. So, we can then say that the repeatability is equal to one program scan. For our example, this means we can consistently react to our sensor within ½ inch repeatability.

Scaling

Often you will have to scale a number from raw units to engineering units. Figure B-3 shows the equation for scaling a raw input value to a scaled value. You should use floating point variables for your calculation in a PLC. If you use integers the result may be rounded when the calculations are performed.

In = Unscaled Raw input value
 Output = Scaled value
 RawMin = Minimum unscaled input value
 RawMax = Maximum unscaled input value
 EuMin = Minimum scaled Engineering Units
 EuMax = Maximum scaled Engineering Units

$$\text{Output} = ((\text{In} - \text{RawMin}) \times \frac{(\text{EuMax} - \text{EuMin})}{\text{RawMax} - \text{RawMin}}) + \text{EuMin}$$

Figure B-3

Appendix C Instruction Set

Bit Instructions









	Function	Description
	Examine On or Normally open	The XIC instruction tests the data bit to see if it is set
	Examine Off or Normally closed	The XIO instruction tests the data bit to see if it is cleared.
	Output	When the OTE instruction is enabled, the controller sets the data bit. When the OTE instruction is disabled, the controller clears the data bit.
	Latch	When enabled, the OTL instruction sets the data bit. The data bit remains set until it is cleared, typically by an OTU instruction. When disabled, the OTL instruction does not change the status of the data bit.
	Unlatch	When enabled, the OTU instruction clears the data bit. When disabled, the OTU instruction does not change the status of the data bit.
	One shot	The ONS instruction enables the remainder of the rung for one program scan. When the ONS instruction is disabled the remainder of the rung is disabled.
	One shot rising	The OSR instruction works the same as the ONS instruction except that an output bit is included.
	One shot falling	Similar to the OSR instruction except it works on the falling edge.

Table C-1

Timers and Counters


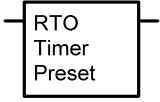
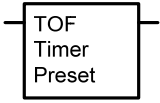
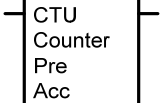
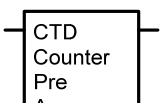

	Function	Description
	Timer on	The TON instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is true). The DN bit is set when the preset is equal to the accumulated. The timer is reset when it is disabled.
	Retentive timer on	The RTO instruction is a retentive timer that accumulates time when the instruction is enabled. The DN bit is set when the preset is equal to the accumulated. An RES instruction is typically used to reset the timer.
	Timer off	The TOF instruction is a non-retentive timer that accumulates time when the instruction is enabled (rung-condition-in is false). The DN bit is set when the preset is equal to the accumulated. The timer is reset when it is enabled.
	Count up	When enabled and the .CU bit is cleared, the CTU instruction increments the counter by one. When enabled and the .CU bit is set, or when disabled, the CTU instruction retains its .ACC value. The DN bit is set when the preset is equal or greater than the accumulated.
	Count down	The CTD instruction is typically used with a CTU instruction that references the same counter structure. When enabled and the .CD bit is cleared, the CTD instruction decrements the counter by one. When enabled and the .CD bit is set, or when disabled, the CTD instruction retains its .ACC value.
	Reset	The RES instruction resets a TIMER, COUNTER, or CONTROL structure.

Table C-2

Compare instructions

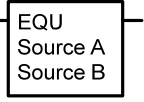
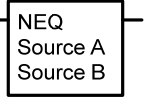
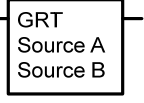
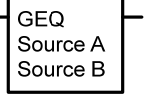
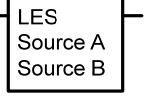
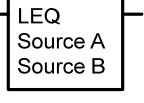
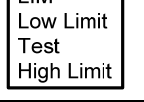
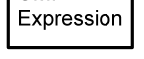
	Function	Description
	Equal	The EQU instruction tests whether Source A is equal to Source B. Valid data types are SINT, INT, DINT, REAL, String type.
	Not equal	The NEQ instruction tests whether Source A is not equal to Source B. Valid data types are SINT, INT, DINT, REAL, String type.
	Greater than	The GRT instruction tests whether Source A is greater than Source B. Valid data types are SINT, INT, DINT, REAL, String type.
	Greater than or Equal to	The GEQ instruction tests whether Source A is greater than or equal to Source B. Valid data types are SINT, INT, DINT, REAL, String type.
	Less than	The LES instruction tests whether Source A is not less than Source B. Valid data types are SINT, INT, DINT, REAL, String type.
	Less than or Equal to	The LEQ instruction tests whether Source A is less than or equal to Source B. Valid data types are SINT, INT, DINT, REAL, String type.
	Limit Test	The LIM instruction tests whether the Test value is within the range of Low Limit to the High Limit. Valid data types are SINT, INT, DINT, REAL.
	Compare	The CMP instruction performs a comparison on the arithmetic operations you specify in the expression. Valid data types are SINT, INT, DINT, REAL.

Table C-3

Compute/Math Instructions

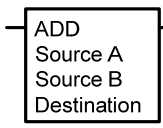
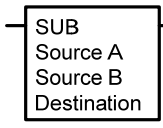
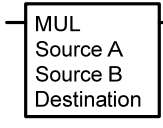
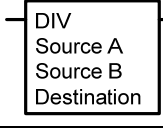
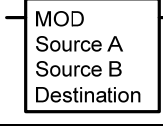
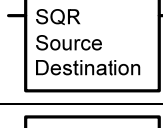
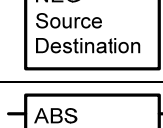
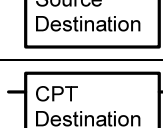
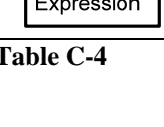
	Function	Description
	Add	The ADD instruction adds Source A to Source B and places the result in the Destination. Valid data types are SINT, INT, DINT, REAL.
	Subtract	The SUB instruction subtracts Source B from Source A and places the result in the Destination. Valid data types are SINT, INT, DINT, REAL.
	Multiply	The MUL instruction multiplies Source A with Source B and places the result in the Destination. Valid data types are SINT, INT, DINT, REAL.
	Divide	The DIV instruction divides Source A by Source B and places the result in the Destination.
	Modulo	The MOD instruction divides Source A by Source B and places the remainder in the Destination.
	Square root	The SQR instruction computes the square root of the Source and places the result in the Destination.
	Negate	The NEG instruction changes the sign of the Source and places the result in the Destination.
	Absolute value	The ABS instruction takes the absolute value of the Source and places the result in the Destination.
	Compute	The CPT instruction performs the arithmetic operations you define in the expression.

Table C-4

Move/Logical Instructions

	Function	Description
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> MOV Source Destination </div>	Move	The MOV instruction copies the Source to the Destination. The Source remains unchanged. Valid data types are SINT, INT, DINT, REAL.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> MVM Source Mask Destination </div>	Masked move	The MVM instruction copies the Source to a Destination and allows portions of the data to be masked. Valid data types are SINT, INT, DINT.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> AND Source A Source B Destination </div>	Bitwise and	The AND instruction performs a bitwise AND operation using the bits in Source A and Source B and places the result in the Destination. Valid data types are SINT, INT, DINT.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> OR Source A Source B Destination </div>	Bitwise inclusive or	The OR instruction performs a bitwise OR operation using the bits in Source A and Source B and places the result in the Destination. Valid data types are SINT, INT, DINT.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> XOR Source A Source B Destination </div>	Bitwise exclusive or	The XOR instruction performs a bitwise XOR operation using the bits in Source A and Source B and places the result in the Destination. Valid data types are SINT, INT, DINT.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> SWPB Source Order Mode Destination </div>	Swap byte	The SWPB instruction rearranges the bytes of a value.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> NOT Source Destination </div>	Bitwise not	The NOT instruction performs a bitwise NOT operation using the bits in the Source and places the result in the Destination. Valid data types are SINT, INT, DINT.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> CLR Destination </div>	Clear	The CLR instruction clears all the bits of the Destination. Valid data types are SINT, INT, DINT, REAL.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> BTD Source Source Bit Destination Dest Bit Length </div>	Bit field distribute	The BTD instruction copies the specified bits from the Source, shifts the bits to the appropriate position, and writes the bits into the Destination. Valid data types are SINT, INT, DINT.

Table C-5

File Misc. Instructions

	Function	Description
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> FAL Control Length Position Mode Dest Expression </div>	File arithmetic and logic	The FAL instruction performs copy, arithmetic, logic, and function operations on data stored in an array.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> FSC Control Length Position Mode Expression </div>	File search and compare	The FSC instruction compares values in an array, element by element.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> COP Source Destination Length </div>	File copy	The COP and CPS instructions copy the value(s) in the Source to the values in the Destination. The Source remains unchanged.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> FLL Source Destination Length </div>	File fill	The FLL instruction fills elements of an array with the Source value. The Source remains unchanged. Valid data types are SINT, INT, DINT, REAL, structure.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> AVE Array Dim to Vary Destination Control Length Position </div>	Average	The AVE instruction calculates the average of a set of values. Valid data types are SINT, INT, DINT, REAL.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> SRT Array Dim to Vary Control Length Position </div>	Sort	The SRT instruction sorts a set of values in one dimension (Dim to vary) of the array into ascending order. Valid data types are SINT, INT, DINT, REAL.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> STD Array Dim to Vary Destination Control Length Position </div>	Standard Deviation	The STD instruction calculates the standard deviation of a set of values in one dimension of the Array and stores the result in the Destination. Valid data types are SINT, INT, DINT, REAL.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> SIZE Source Dim to Vary Size </div>	Size in elements	The SIZE instruction finds the size of a dimension of an array.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> CPS Source Destination Length </div>	File synchronous copy	The COP and CPS instructions copy the value(s) in the Source to the values in the Destination. The Source remains unchanged.

Table C-6

File/Shift Instructions

	Function	Description
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> BSL Array Control Source Bit Length </div>	Bit shift left	The BSL instruction shifts the specified bits within the Array one position left. Valid data type is DINT.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> BSR Array Control Source Bit Length </div>	Bit shift right	The BSR instruction shifts the specified bits within the Array one position right. Valid data type is DINT.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> FFL Source FIFO Control Length Position </div>	FIFO load	The FFL instruction copies the Source value to the FIFO array. Use the FFL instruction with the FFU instruction to store and retrieve data in a first-in/first-out order. When used in pairs, the FFL and FFU instructions establish an asynchronous shift register. Valid data types for the source and the FIFO array are SINT, INT, DINT, REAL, String, structure.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> FFU FIFO Destination Control Length Position </div>	FIFO unload	The FFU instruction unloads the value from position 0 (first position) of the FIFO and stores that value in the Destination. The remaining data in the FIFO shifts down one position. Use the FFU instruction with the FFL instruction to store and retrieve data in a first-in/first-out order.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> LFL Source LIFO Control Length Position </div>	LIFO load	The LFL instruction copies the Source value to the LIFO. Use the LFL instruction with the LFU instruction to store and retrieve data in a last-in/first-out order. When used in pairs, the LFL and LFU instructions establish an asynchronous shift register. Valid data types for the source and the LIFO array are SINT, INT, DINT, REAL, String, structure.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> LFU LIFO Destination Control Length Position </div>	LIFO unload	The LFU instruction unloads the value at .POS of the LIFO and stores 0 in that location. Use the LFU instruction with the LFL instruction to store and retrieve data in a last-in/first-out order.

Table C-7

Sequencer Instructions

	Function	Description
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> SQI Array Mask Source Control Length Position </div>	Sequencer input	The SQI instruction detects when a step is complete in a sequence pair of SQO/SQI instructions.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> SQO Array Mask Destination Control Length Position </div>	Sequencer output	The SQO instruction sets output conditions for the next step of a sequence pair of SQO/SQI instructions.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> LFL Array Source Control Length Position </div>	Sequencer load	The SQL instruction loads reference conditions into a sequencer array.

Table C-8

Program Control Instructions

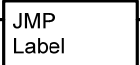

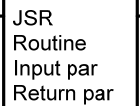
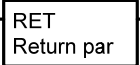
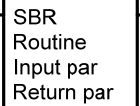
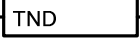
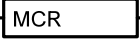



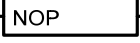
	Function	Description
	Jump to label	When enabled, the JMP instruction skips to the referenced LBL instruction and the controller continues executing from there. When disabled, the JMP instruction does not affect ladder execution.
	Label	The JMP and LBL instructions skip portions of ladder logic.
	Jump to subroutine	The JSR instruction jumps execution to a different routine. The SBR instruction passes data to and executes a routine. The RET instruction returns the results.
	Return from subroutine	See the JSR instruction.
	Subroutine label	See the JSR instruction.
	Temporary end	The TND instruction acts as a boundary.
	Master control reset	The MCR instruction, used in pairs, creates a program zone that can disable all rungs within the MCR instructions.
	User Interrupt disable	The UID instruction and the UIE instruction work together to prevent a small number of critical rungs from being interrupted by other tasks.
	User Interrupt enable	See the UID instruction.
	Always false	The AFI instruction sets its rung-condition-out to false.
	No operation	The NOP instruction functions as a placeholder.

Table C-9

For/Break Instructions


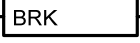
	Function	Description
	For	The FOR instruction executes a routine repeatedly.
	Break	The BRK instruction interrupts the execution of a routine that was called by a FOR instruction.

Table C-10

Special Instructions

	Function	Description
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> FBC Source Reference Cmp. Ctrl. Length Position Result Ctrl. Length Position </div>	File bit compare	When enabled, the FBC instruction compares the bits in the Source array with the bits in the Reference array and records the bit number of each mismatch in the Result array.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> DDT Source Reference Result Cmp. Ctrl. Length Position Result Ctrl. Length Position </div>	Diagnostic Detect	When enabled, the DDT instruction compares the bits in the Source array with the bits in the Reference array, records the bit number of each mismatch in the Result array, and changes the value of the Reference bit to match the value of the corresponding Source bit.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> DTR Source Mask Reference </div>	Data transition	The DTR instruction passes the Source value through a Mask and compares the result with the Reference value. The DTR instruction also writes the masked Source value into the Reference value for the next comparison. The Source remains unchanged.
<div style="border: 1px solid black; padding: 5px; width: fit-content;"> PID PID Var. Process Var. Tieback Control Var. Master Loop Inhold Bit Inhold Val Setpoint Process Var Output </div>	PID	The PID instruction controls a process variable such as flow, pressure, temperature, or level.

Table C-11

Trig Functions


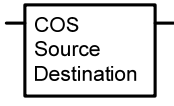

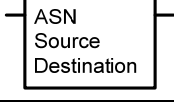
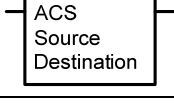
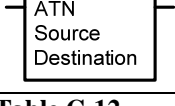
	Function	Description
	Sine	The SIN instruction takes the sine of the Source value (in radians) and stores the result in the Destination.
	Cosine	The COS instruction takes the cosine of the Source value (in radians) and stores the result in the Destination.
	Tangent	The TAN instruction takes the tangent of the Source value (in radians) and stores the result in the Destination.
	Arc sine	The ASN instruction takes the arc sine of the Source value and stores the result in the Destination (in radians).
	Arc cosine	The ACS instruction takes the arc cosine of the Source value and stores the result in the Destination (in radians).
	Arc tangent	The ATN instruction takes the arc tangent of the Source value and stores the result in the Destination (in radians).

Table C-12

Advance Math Instructions



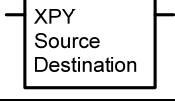
	Function	Description
	Natural Log	The LN instruction takes the natural log of the Source and stores the result in the Destination.
	Log base 10	The LOG instruction takes the log base 10 of the Source and stores the result in the Destination.
	X to the power of Y	The XPY instruction takes Source A (X) to the power of Source B (Y) and stores the result in the Destination.

Table C-13

Math Conversions


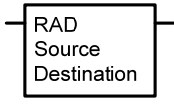

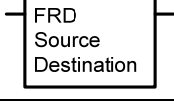
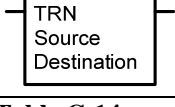
	Function	Description
	Radians to degrees	The DEG instruction converts the Source (in radians) to degrees and stores the result in the Destination.
	Degrees to radians	The RAD instruction converts the Source (in degrees) to radians and stores the result in the Destination.
	To BCD	The TOD instruction converts a decimal value ($0 \leq \text{Source} \leq 99,999,999$) to a BCD value and stores the result in the Destination.
	From BCD	The FRD instruction converts a BCD value (Source) to a decimal value and stores the result in the Destination.
	Truncate	The TRN instruction removes (truncates) the fractional part of the Source and stores the result in the Destination.

Table C-14

ASCII Serial Port

	Function	Description						
<table border="1"> <tr> <td>WRT</td> <td>Channel</td> <td>Source</td> <td>Port Control</td> <td>Length</td> <td>Chars Sent</td> </tr> </table>	WRT	Channel	Source	Port Control	Length	Chars Sent	ASCII Write	The AWT instruction sends characters of the Source array to a serial device.
WRT	Channel	Source	Port Control	Length	Chars Sent			
<table border="1"> <tr> <td>AWA</td> <td>Channel</td> <td>Source</td> <td>Port Control</td> <td>Length</td> <td>Chars Sent</td> </tr> </table>	AWA	Channel	Source	Port Control	Length	Chars Sent	ASCII write append	The AWA instruction sends characters of the Source array to a serial device and appends either one or two predefined characters.
AWA	Channel	Source	Port Control	Length	Chars Sent			
<table border="1"> <tr> <td>ARD</td> <td>Channel</td> <td>Destination</td> <td>Port Control</td> <td>Length</td> <td>Chars Read</td> </tr> </table>	ARD	Channel	Destination	Port Control	Length	Chars Read	ASCII read	The ARD instruction removes characters from the buffer and stores them in the Destination.
ARD	Channel	Destination	Port Control	Length	Chars Read			
<table border="1"> <tr> <td>ARL</td> <td>Channel</td> <td>Source</td> <td>Port Control</td> <td>Length</td> <td>Chars Read</td> </tr> </table>	ARL	Channel	Source	Port Control	Length	Chars Read	ASCII read line	The ARL instruction removes characters from the buffer and stores them in the Destination.
ARL	Channel	Source	Port Control	Length	Chars Read			
<table border="1"> <tr> <td>ABL</td> <td>Channel</td> <td>Port Control</td> <td>Char Count</td> </tr> </table>	ABL	Channel	Port Control	Char Count	ASCII test for buffer line	The ABL instruction counts the characters in the buffer up to and including the first termination character.		
ABL	Channel	Port Control	Char Count					
<table border="1"> <tr> <td>ACB</td> <td>Channel</td> <td>Port Control</td> <td>Char Count</td> </tr> </table>	ACB	Channel	Port Control	Char Count	ASCII characters in buffer	The ACB instruction counts the characters in the buffer.		
ACB	Channel	Port Control	Char Count					
<table border="1"> <tr> <td>ACL</td> <td>Channel</td> <td>Clear Read</td> <td>Clear Write</td> </tr> </table>	ACL	Channel	Clear Read	Clear Write	ASCII clear buffer	The ACL instruction immediately clears the buffer and ASCII queue.		
ACL	Channel	Clear Read	Clear Write					

Table C-15

ASCII String Instructions

	Function	Description
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> FIND Source Search Start Result </div>	Find a string	The FIND instruction locates the starting position of a specified string within another string.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> INSERT Source A Source B Start Destination </div>	Insert string into string	The INSERT instruction adds ASCII characters to a specified location within a string.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> CONCAT Source A Source B Destination </div>	String concatenate	The CONCAT instruction adds ASCII characters to the end of a string.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> MID Source Qty Start Destination </div>	Extract part of a string	The MID instruction copies a specified number of ASCII characters from a string and stores them in another string.
<div style="border: 1px solid black; padding: 2px; width: fit-content;"> DELETE Source Qty Start Destination </div>	Delete characters from a string	The DELETE instruction removes ASCII characters from a string.

Table C-16

ASCII Conversion

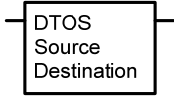
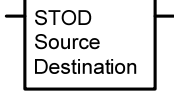

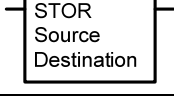
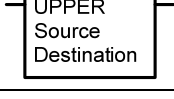
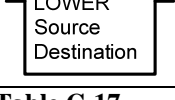
	Function	Description
	DINT to string	The DTOS instruction produces the ASCII representation of a value.
	String to DINT	The STOD instruction converts the ASCII representation of an integer to an integer or REAL value.
	Real to string	The RTOS instruction produces the ASCII representation of a REAL value.
	String to real	The STOR instruction converts the ASCII representation of a floating-point value to a REAL value.
	Upper case	The UPPER instruction converts the alphabetical characters in a string to upper case characters.
	Lower case	The LOWER instruction converts the alphabetical characters in a string to lower case characters.

Table C-17

Appendix D Table of Figures

Figure 1-1 Normally Open Logic	1-9
Figure 1-2 Normally Closed Logic	1-10
Figure 1-3 And Logic.....	1-10
Figure 1-4 Or Logic	1-10
Figure 1-5 Latch Logic	1-11
Figure 1-6 Scan Order.....	1-11
Figure 1-7 I/O Synchronization to Program Scan.....	1-12
Figure 1-8 Direct Array Access	1-14
Figure 1-9 Using the Program Scan to Loop thru an Array.....	1-14
Figure 1-10 Using the FOR Instruction to Loop thru an Array	1-14
Figure 1-11 Routine WT_LOOP	1-15
Figure 1-12 FAL instruction Loop.....	1-15
Figure 1-13 Control Logix Program Structure.....	1-16
Figure 3-1 HMI Mixer process	3-25
Figure 3-2 Controller Program Organization.....	3-27
Figure 3-3 Discrete Motor Controller Faceplate.....	3-28
Figure 3-4 Discrete Motor Controller Interlocks	3-30
Figure 3-5 Discrete Motor Controller Interlock summation.....	3-31
Figure 3-6 Discrete Motor Controller Contactor Circuit	3-31
Figure 3-7 Discrete Motor Controller Alarm Logic	3-32
Figure 3-8 Variable Frequency Drive Controller Faceplate	3-33
Figure 3-9 Variable Frequency Drive Controller Using the Remote Setpoint	3-34
Figure 3-10 Variable Frequency Drive Controller Scaling the Output.....	3-35
Figure 3-11 Single Solenoid Valve Controller Faceplate	3-36
Figure 3-12 Single Solenoid Controller Interlocks	3-38
Figure 3-13 Single Solenoid Valve Controller Interlock Summation	3-38
Figure 3-14 Single Solenoid Valve Controller Output Logic.....	3-38
Figure 3-15 Single Solenoid Valve Controller Alarm Logic.....	3-39
Figure 3-16 Double Solenoid Valve Controller Open Output Logic.....	3-40
Figure 3-17 Double Solenoid Valve Controller Close Output Logic	3-40
Figure 3-18 Double Solenoid Valve Controller Open Alarm Logic.....	3-41
Figure 3-19 Analog Indicator Faceplate	3-42
Figure 3-20 Analog Alarm Faceplate	3-43
Figure 3-21 Analog Alarm Subroutine ANALOG_ALARM.....	3-45
Figure 3-22 Subroutine ANALOG_ALARM High Alarm Logic	3-45
Figure 3-23 Subroutine ANALOG_ALARM Low Alarm Logic	3-46
Figure 3-24 Subroutine ANALOG_ALARM summation one shot.....	3-47
Figure 3-25 Subroutine ANALOG_ALARM Process Variable Return	3-47
Figure 3-26 Analog Indicator Alarm Enable	3-48
Figure 3-27 Analog Indicator Jump to Subroutine	3-48
Figure 3-28 Analog Alarm Horn Silence Reset.....	3-48
Figure 3-29 PID Controller Faceplate.....	3-49
Figure 3-30 PID Controller Process Variable Alarm Faceplate	3-50
Figure 3-31 PID Controller Trending Faceplate.....	3-51

Table of Figures

Figure 3-32 PID Controller Process Variable Alarm Enable.....	3-52
Figure 3-33 PID Controller Deviation Alarm Enable.....	3-53
Figure 3-34 PID Controller Setting the UDT Process Variable	3-53
Figure 3-35 PID Controller Calling the ANALOG_ALARM Routine	3-54
Figure 3-36 PID Controller Horn Silence Reset.....	3-54
Figure 3-37 PID Controller Using the Remote Setpoint.....	3-54
Figure 3-38 Getting the System Variable WALLCLOCKTIME to calculate SCAN_TIME....	3-55
Figure 3-39 PID Controller Process Calculating the Ramped Setpoint.....	3-55
Figure 3-40 PID Controller Instruction and Manual Mode	3-56
Figure 3-41 PID Controller Setting the Control Variable to the VFD Remote Setpoint.....	3-56
Figure 3-42 Ratio Controller Faceplate	3-57
Figure 3-43 Ratio Controller Calculating the Ratio and Setting the FIC Remote Setpoint.....	3-58
Figure 3-44 Ratio Controller Call to the ANALOG_ALARM Subroutine	3-59
Figure 4-1 State Logic First Form Example Step	4-61
Figure 4-2 State Logic First Form End Of Cycle	4-62
Figure 4-3 State Logic First Form Output Logic	4-62
Figure 4-4 State Logic Second Form Example Step.....	4-63
Figure 4-5 State Logic Second Form Output Logic.....	4-64
Figure 4-6 Gain in Weight Feeder Process Diagram	4-65
Figure 4-7 Gain in Weight Feeder SFC	4-66
Figure 4-8 Gain in Weight Feeder Initialize	4-67
Figure 4-9 Gain in Weight Feeder Calculate the Cut-Off.....	4-68
Figure 4-10 Gain in Weight Feeder Opening the Feed Valve	4-68
Figure 4-11 Determining the Bump Count, the Low Point and the High Point	4-69
Figure 4-12 Closing the Feed Valve and Checking the Bump, Low Point and High Point	4-70
Figure 4-13 Gain in Weight Feeder Bumping the Feed Valve to Achieve Tolerance.....	4-71
Figure 4-14 Gain in Weight Feeder Bump Alarm	4-71
Figure 4-15 Gain in Weight Feeder High Weight Alarm	4-72
Figure 4-16 Gain in Weight Feeder Adjusting the In-flight	4-72
Figure 4-17 Gain in Weight Feeder Complete.....	4-73
Figure 4-18 State Logic Third Form Example Step.....	4-75
Figure 4-19 State Logic Third Form, Valve Output Logic	4-76
Figure 4-20 Confirm Routine, Reset the Timers	4-76
Figure 4-21 Confirm Routine Valve Position Confirm.	4-77
Figure 4-22 Confirm Routine Setpoint Comparison Confirm	4-77
Figure 4-23 Confirm Routine, DELAY_CONFIRM.....	4-78
Figure 4-24 The end of the Confirm Routine Confirm Summation	4-78
Figure 5-1 The batching process and instrumentation diagram P&ID	5-79
Figure 5-2 Batching program organization.....	5-81
Figure 5-3 Recipe Parameter Data Flow.....	5-84
Figure 5-4 Recipe Storage Options.....	5-86
Figure 5-5 Batch Control Reset	5-88
Figure 5-6 Batch Control Initiate the Batch Controller	5-88
Figure 5-7 Batch Control Identify Transition Phases	5-89
Figure 5-8 Batch Control Reset the Phase and Prepare to Update Parameters.....	5-90
Figure 5-9 Batch Control Update the Phase Parameters and Set the Report Pointer	5-90

Ladder logic Programming Techniques By Duane Snider

Figure 5-10 Batch Control Wait for Transition Phases to Complete..... 5-91
Figure 5-11 Batch Control Increment the Batch Controller Sequence Step..... 5-91
Figure 5-12 Common Phase Logic Set the Start Time and Run..... 5-93
Figure 5-13 Common Phase Logic Record the Phase Parameters and Set the End Time 5-93
Figure 5-14 Common Phase Logic Phase Running Complete..... 5-94
Figure 5-15 Start Phase Logic..... 5-95
Figure 5-16 End Phase Logic..... 5-96
Figure 5-17 Feed Material 1 Phase Open the Feed Valve 5-97
Figure 5-18 Feed Material 1 Phase Open the Feed Valve 5-98
Figure 5-19 Feed Material 1 Phase Check for Tolerance 5-99
Figure 5-20 Tare Scale Phase Logic 5-100
Figure 5-21 Start Heat Phase Logic 5-101
Figure 5-22 End Heat Phase Logic 5-101
Figure 5-23 Start Cooling Phase Logic..... 5-102
Figure 5-24 End Cooling Phase Logic..... 5-102
Figure 5-25 Start Agitator Phase Logic 5-103
Figure 5-26 Stop Agitator Phase Logic 5-103
Figure 5-27 If PV Phase Set Process Variable for Comparison 5-105
Figure 5-28 If PV Phase Wait for True Compare 5-105
Figure 5-29 Delay Phase Reset the Delay Timer..... 5-106
Figure 5-30 Delay Phase Wait for Delay Timer 5-106
Figure 5-31 If OK Phase Display the OK Message to the Operator..... 5-107
Figure 5-32 If OK Phase Wait for the Ok Button From the Operator 5-108
Figure 5-33 If Yes Phase Display the Message to the Operator 5-108
Figure 5-34 If Yes Phase Wait for the Operator Response..... 5-109
Figure 5-35 If Yes Phase When No is Selected Set the Else Step for the Batch 5-109
Figure 5-36 If Yes Phase Set the Batch Controller to the Else Step..... 5-110
Figure 5-37 If Done Phase Increment the Phase Done Pointer 5-111
Figure 5-38 If Done Phase Check if the Phase is Complete 5-111
Figure 5-39 Go to Step Phase Increment the Go To Pointer 5-112
Figure 5-40 Go to Step Phase When the Phase is Found Set the Sequence Step 5-112
Figure 6-1 The Sequence Diagram 6-114
Figure 6-2 Sequential Machine Control Step Logic 6-115
Figure 6-3 Sequential Machine Control Manual Mode 6-117
Figure 6-4 Sequential Machine Control End of Cycle..... 6-118
Figure 6-5 Sequential Machine Control Timed Step 6-118
Figure 6-6 Sequential Machine Control Latched Step..... 6-119
Figure 6-7 Sequential Machine Control Bumpless Transfer Output Logic..... 6-120
Figure 6-8 Sequential Machine Control Output Logic 6-121
Figure 6-9 Sequential Machine Control Motor Fault 6-121
Figure 6-10 Sequential Machine Control Single Solenoid Valve Fault 6-122
Figure 6-11 Sequential Machine Control Double Solenoid Valve Fault..... 6-122
Figure 6-12 Sequential Machine Control Step Excess Time Fault..... 6-123
Figure 6-13 Sequential Machine Control Sequence Fault 6-123
Figure 6-14 Sequential Machine Control Cycle Hold 6-124
Figure 6-15 Sequential Machine Control Holding a Step with Cycle Hold 6-124

Table of Figures

Figure 6-16 Sequential Machine Control mid Cycle Start SFC	6-125
Figure 6-17 Sequential Machine Control mid Cycle Start Logic	6-126
Figure 6-18 Sequential Machine Control Selection Branch Diverge SFC	6-127
Figure 6-19 Sequential Machine Control Selection Branch Diverge Logic	6-128
Figure 6-20 Sequential Machine Control Parallel Branch Diverge SFC	6-129
Figure 6-21 Sequential Machine Control Parallel Branch Diverge Logic	6-130
Figure 7-1 First Logic for 2 Stations	7-131
Figure 7-2 First Logic Example Process	7-132
Figure 7-3 First Logic Example	7-133
Figure 7-4 Three Station First Logic Example	7-133
Figure 7-5 Three Station First Logic Second Priority	7-134
Figure 7-6 Three Station First Logic First Priority	7-134
Figure 7-7 First Logic Number of Rungs Calculation	7-135
Figure 7-8 Priority Stack Call to the Stack Routine	7-136
Figure 7-9 Priority Stack Routine STACK Move the Station Number into the Stack	7-136
Figure 7-10 Priority Stack Routine STACK Shift the Station Number to Priority One	7-137
Figure 7-11 Priority Stack Routine STACK Clear the Station Number from the Stack	7-137
Figure 7-12 Priority Stack Routine STACK Determine the First Priority	7-137
Figure 8-1 Sortation Conveyor Layout	8-138
Figure 8-2 Reading the Barcode thru the Serial Port	8-139
Figure 8-3 Checking for a Valid Barcode Read	8-139
Figure 8-4 Loading the Sort Lane FIFOs	8-140
Figure 8-5 Unloading the Sort Lane FIFOs	8-140
Figure 8-6 The Wrap Around Encoder Diagram	8-141
Figure 8-7 Enabling the Transfer with the Encoder Window	8-142
Figure 8-8 Checking if the Sort Lane Photo Eye is in the Encoder Window	8-143
Figure 8-9 The Sort Lane Solenoid Logic	8-143
Figure 8-10 Enabling the Tracking Timer	8-144
Figure 8-11 The Tracking Timer Logic	8-145
Figure 8-12 Executing the Tracking Timer Loop	8-145
Figure 8-13 Tracking Timer Routine TM_LOOP	8-146
Figure 8-14 Loading the Timer Pointer into the Sort Lane FIFO	8-146
Figure 8-15 Unloading Timer Pointer from the Sort Lane FIFO	8-147
Figure 8-16 Enabling the Sort Lane with the Tracking Timer	8-147
Figure 9-1 Power and Free Stop to Stop no Accumulation	9-148
Figure 9-2 Power and Free Program Organization	9-149
Figure 9-3 Power and Free Stop to Stop no Accumulation Logic	9-150
Figure 9-4 Power and Free Choke Zone	9-151
Figure 9-5 Power and Free Choke Zone Logic	9-152
Figure 9-6 Power and Free Choke Zone Reset Logic	9-152
Figure 9-7 Power and Free Count Zone	9-153
Figure 9-8 Power and Free Count Zone Logic	9-153
Figure 9-9 Power and Free Merge into a Count Zone	9-154
Figure 9-10 Power and Free Merge Stop 1 Logic	9-155
Figure 9-11 Power and Free Merge Stop 2 Logic	9-155
Figure 9-12 Power and Free Merge Full and Down Stream Count Zone	9-156

Ladder logic Programming Techniques By Duane Snider

Figure 9-13 Power and Free Merge Full Reset.....	9-156
Figure 9-14 Power and Free Merge Fault.....	9-157
Figure 9-15 Power and Free Track Switch.....	9-157
Figure 9-16 Power and Free Track Switch Push Button Left and Right Latches.....	9-158
Figure 9-17 Power and Free Track Switch Stop Logic.....	9-158
Figure 9-18 Power and Free Track Switch Full and Clear.....	9-159
Figure 9-19 Power and Free Track Switch Left and Right Solenoid Valves.....	9-159
Figure 9-20 Power and Free Track Switch Down Stream Choke Left and Right Full.....	9-160
Figure 10-1 Toggle Push Button.....	10-161
Figure 10-2 Cascading Start and Stop Logic.....	10-162
Figure 10-3 Cascading Start Stop Interlocks.....	10-163
Figure 10-4 Cascading Start Stop Fast Shutdown.....	10-163
Figure 10-5 Message Display Increment the Message Pointer.....	10-164
Figure 10-6 Message Display Check the Alarm and Set the Message Number.....	10-164
Figure 10-7 Display the Message.....	10-165
Figure 10-8 Buffering Transactions Set the FIFO Size.....	10-167
Figure 10-9 Buffering Transactions Call the BLEND_UPLOAD Routine.....	10-167
Figure 10-10 BLEND_UPLOAD Routine Receive the Input Paramters.....	10-167
Figure 10-11 BLEND_UPLOAD Routine Increment the Transaction Pointer.....	10-168
Figure 10-12 End BLEND_UPLOAD Load the Transaction Table and FIFO the Pointer...	10-168
Figure 10-13 Buffering Transactions Unload the Stack Pass the Transaction.....	10-169