Chapter 8 Math Functions

Introduction

The introduction of mathematical operations in the PLC provided major benefits to control logic. Numeric data could be combined with logic to provide more powerful control strategies. For instance, decisions could be made concerning mathematical operations concerning counts of products, weights of a product, the temperature of an oven or any numeric variable in a process.

Siemens Math Instructions

The instructions are briefly divided into three categories: Compare Blocks, Math Blocks and Move Blocks. First will be the Compare blocks:

Comparator operations	
CMP ==	Equal
CMP <>	Not equal
CMP >=	Greater or equal
CMP <=	Less or equal
CMP >	Greater than
CMP <	Less than
IN_RANGE	Value within range
OUT_RANGE	Value outside range
–|OK|–	Check validity
–|NOT_OK|–	Check invalidity

Fig. 8-1 Siemens Comparator Operations

Again, instructions given have the TIA definition quoted below. The full definition for these instructions can be found in the reference manual.

Compare Instruction

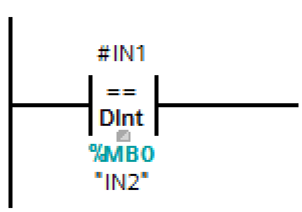


Fig. 8-2 Siemens Compare If Equal Operations

"You use the compare instructions to compare two values of the same data type. When the LAD contact comparison is TRUE, then the contact is activated. When the FBD box comparison is TRUE, then the box output is TRUE.

Relation type	The comparison is true if:
==	IN1 is equal to IN2
<>	IN1 is not equal to IN2
>=	IN1 is greater than or equal to IN2

<=	IN1 is less than or equal to IN2
>	IN1 is greater than IN2
<	IN1 is less than IN2”

In Range and Out of Range Instructions

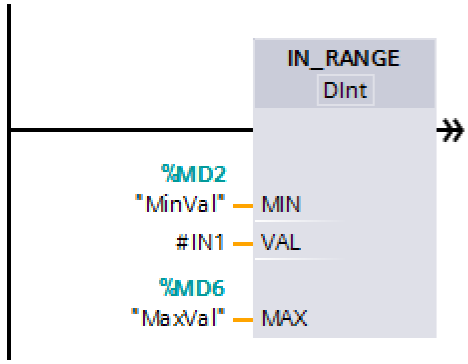


Fig. 8-3 Siemens Test If In Range Instruction

“You use the IN_RANGE and OUT_RANGE instructions to test whether an input value is in or out of a specified value range. If the comparison is TRUE, then the box output is TRUE. The input parameters MIN, VAL, and MAX must be the same data type.”

Relation type	The comparision is TRUE if:
IN_RANGE	MIN <= VAL <= MAX
OUT_RANGE	VAL < MIN or VAL > MAX

OK and Not OK Instructions

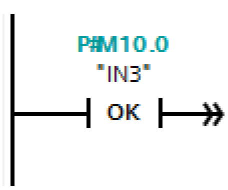


Fig. 8-4 Siemens Test If Data OK Instruction

“You use the OK and NOT_OK instructions to test whether an input data reference is a valid real number according to IEEE specification 754. When the LAD contact is TRUE, the contact is activated and passes power flow. When the FBD box is TRUE, then the box output is TRUE.”

Instruction	The Real number test is TRUE if:
OK	The input value is a valid Real number
NOT_OK	The input value is not a valid Real number

Math Instructions

Name	Description
Math functions	
CALCULATE	Calculate
ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
MOD	Return remainder of division
NEG	Create twos complement
INC	Increment
DEC	Decrement
ABS	Form absolute value
MIN	Get minimum
MAX	Get maximum
LIMIT	Set limit value
SQR	Form square
SQRT	Form square root
LN	Form natural logarithm
EXP	Form exponential value
SIN	Form sine value
COS	Form cosine value
TAN	Form tangent value
ASIN	Form arcsine value
ACOS	Form arccosine value
ATAN	Form arctangent value
FRAC	Return fraction
EXPT	Exponentiate

Fig. 8-5 Siemens Table of Math Functions

Add, Sub, Mul, Div, Calculate

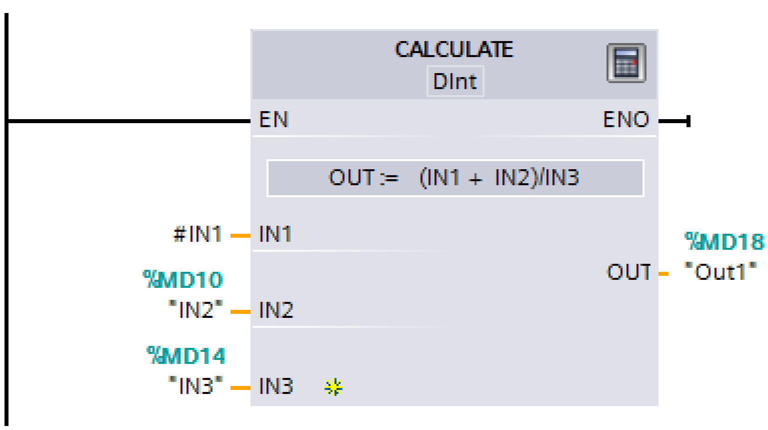


Fig. 8-6 Siemens Calculate Instruction

"You use a math box instruction to program the basic mathematical operations:

ADD:	Addition	(IN1 + IN2 = OUT)
SUB:	Subtraction	(IN1 - IN2 = OUT)
MUL:	Multiplication	(IN1 * IN2 = OUT)
DIV:	Division	(IN1 / IN2 = OUT)

An Integer division operation truncates the fractional part of the quotient to produce an integer output.

When enabled (EN = 1), the math instruction performs the specified operation on the input values (IN1 and IN2) and stores the result in the memory address specified by the output parameter (OUT). After the successful completion of the operation, the instruction sets ENO = 1."

ENO status	Description
1	No error
0	The Math operation result value would be outside the valid number range of the data type selected. The least significant part of the result that fits in the destination size is returned.
0	Division by 0 (IN2 = 0): The result is undefined and zero is returned.
0	Real/LReal: If one of the input values is NaN (not a number) then NaN is returned.
0	ADD Real/LReal: If both IN values are INF with different signs, this is an illegal operation and NaN is returned.
0	SUB Real/LReal: If both IN values are INF with the same sign, this is an illegal operation and NaN is returned.
0	MUL Real/LReal: If one IN value is zero and the other is INF, this is an illegal operation and NaN is returned.
0	DIV Real/LReal: If both IN values are zero or INF, this is an illegal operation and NaN is returned.

Mod

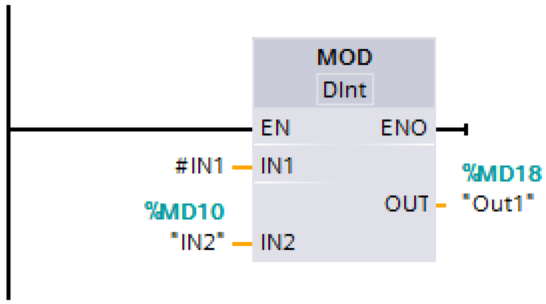


Fig. 8-7 Siemens Modulo Instruction

"You use a MOD (modulo) instruction for the IN1 modulo IN2 math operation. The operation IN1 MOD IN2 = IN1 - (IN1 / IN2) = parameter OUT."

NEG

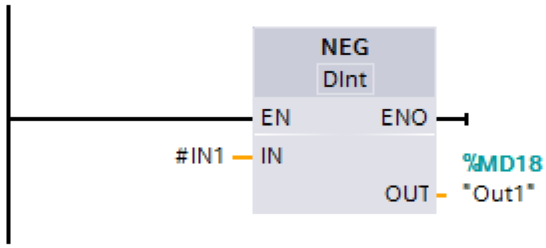


Fig. 8-8 Siemens
Negation Instruction

"You use the NEG (negation) instruction to invert the arithmetic sign of the value at parameter IN and store the result in parameter OUT."

Increment and Decrement

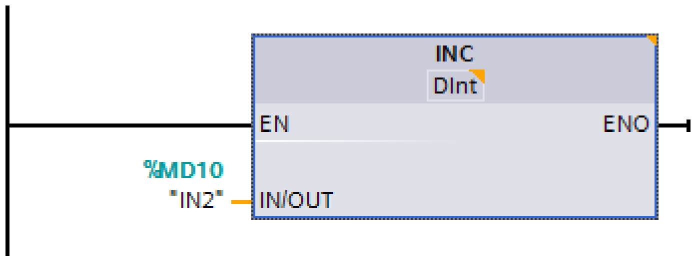


Fig. 8-9 Siemens
Increment Instruction

"You use the INC and DEC instructions to:

Increment a signed or unsigned integer number value
INC (increment): Parameter IN/OUT value +1 = parameter IN/OUT value

Decrement a signed or unsigned integer number value
DEC (decrement): Parameter IN/OUT value - 1 = parameter IN/OUT value"

Absolute Value

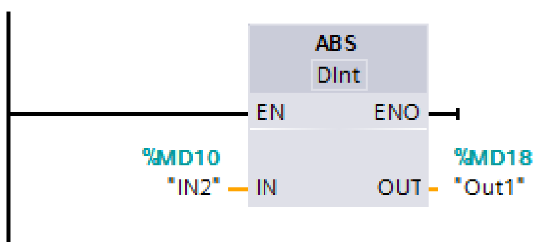


Fig. 8-10 Siemens
Absolute Value Instruction

"You use the ABS instruction to get the absolute value of a signed integer or real number at parameter IN and store the result in parameter OUT."

MIN and MAX

"You use the MIN (minimum) and MAX (maximum) instructions as follows:

> MIN compares the value of two parameters IN1 and IN2 and assigns the minimum (lesser) value to parameter OUT.
>
> MAX compares the value of two parameters IN1 and IN2 and assigns the maximum (greater) value to parameter OUT."

Limit

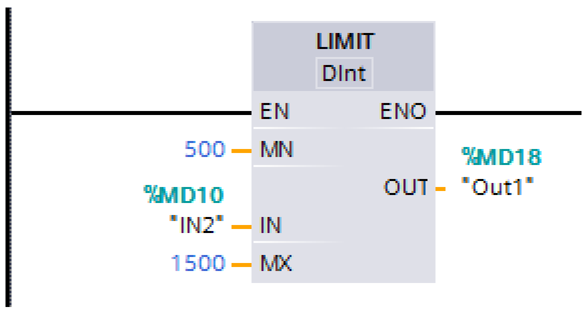


Fig. 8-11 Siemens Limit Instruction

"You use the Limit instruction to test if the value of parameter IN is inside the value range specified by parameters MIN and MAX. The OUT value is clamped at the MIN or MAX value, if the IN value is outside this range.

> If the value of parameter IN is inside specified range, then the value of IN is stored in parameter OUT.
>
> If the value of parameter IN is outside of the specified range, then the OUT value is the value of parameter MIN (if the IN value is less than the MIN value) or the value of parameter MAX (if the IN value is greater than the MAX value)."

Floating-Point Math

You use the floating point instructions to program mathematical operations using a Real or LReal data type:

SQR: Square (IN 2 = OUT)
SQRT: Square root (√IN = OUT)
LN: Natural logarithm (LN(IN) = OUT)
EXP: Natural exponential (e IN =OUT), where base e = 2.71828182845904523536
SIN: Sine (sin(IN radians) = OUT)
COS: Cosine (cos(IN radians) = OUT)
TAN: Tangent (tan(IN radians) = OUT)
ASIN: Inverse sine (arcsine(IN) = OUT radians), where the sin(OUT radians) = IN
ACOS: Inverse cosine (arccos(IN) = OUT radians), where the cos(OUT radians) = IN
ATAN: Inverse tangent (arctan(IN) = OUT radians), where the tan(OUT radians) = IN
FRAC: Fraction (fractional part of floating point number IN = OUT)
EXPT: General exponential (IN1 IN2 = OUT)

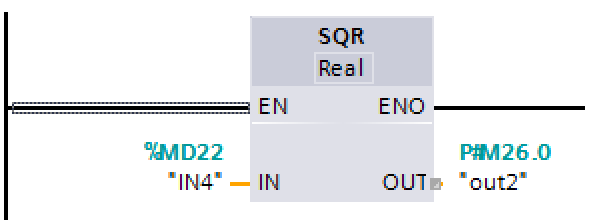


Fig. 8-12 Siemens Square Instruction

MOVE Operations

Move operations	
MOVE	Move value
FieldRead	Read field
FieldWrite	Write field
MOVE_BLK	Move block
UMOVE_BLK	Move block uninterruptible
FILL_BLK	Fill block
UFILL_BLK	Fill block uninterruptible
SWAP	Swap

Fig. 8-13 Siemens Move Operations

"Use the move instructions to copy data elements to a new memory address and convert from one data type to another. The source data is not changed by the move process.

MOVE: Copies a data element stored at a specified address to a new address
MOVE_BLK: Interruptible move that copies a block of data elements to a new address
UMOVE_BLK: Uninterruptible move that copies a block of data elements to a new Address"

MOVE_BLK, UMOVE_BLK

"The MOVE instruction copies a single data element from the source address specified by the IN parameter to the destination address specified by the OUT parameter. The MOVE_BLK and UMOVE_BLK instructions have an additional COUNT parameter. The COUNT specifies how many data elements are copied. The number of bytes per element copied depends on the data type assigned to the IN and OUT parameter tag names in the PLC tag table."

Fill

"You use the FILL_BLK and UFILL_BLK instructions as follows:

> FILL_BLK: The interruptible fill instruction fills an address range with copies of a specified data element.
> UFILL_BLK: The uninterruptible fill instruction fills an address range with copies of a specified data element."

Swap

"You use the SWAP instruction to reverse the byte order for two-byte and four-byte data elements. No change is made to the bit order within each byte. ENO is always TRUE following execution of the SWAP instruction."

Convert

"You use the CONVERT instruction to convert a data element from one data type to another data type. Click below the box name and then select IN and OUT data types from the dropdown list. After you select the (convert from) data type, a list of possible conversions is shown in the (convert to) dropdown list. Conversions from and to BCD16 are restricted to the Int data type. Conversions from and to BCD32 are restricted to the DInt data type."

Round and Truncate

"ROUND converts a real number to an integer. The real number fraction is rounded to the nearest integer value (IEEE - round to nearest). If the Real number is exactly one-half the span between two integers (i.e. 10.5), then the Real number is rounded to the even integer. For example, ROUND (10.5) = 10 or ROUND (11.5) = 12."

Ceiling and Floor

"CEIL converts a real number to the smallest integer greater than or equal to that real number (IEEE - round to +infinity).
FLOOR converts a real number to the greatest integer smaller than or equal to that real number (IEEE - round to -infinity)."

Scale and Normalize

“SCALE_X scales the normalized real parameter VALUE where (0.0 <= VALUE <= 1.0) in the data type and value range specified by the MIN and MAX parameters:

OUT = VALUE (MAX - MIN) + MIN”

Allen-Bradley SLC Math Instructions

A Review of Math Function Blocks from SLC-500 Reference Manual:

“Instruction	Name	Purpose
ADD	Add	Adds source A to source B and stores the result in the destination
SUB	Subtraction	Subtracts source B from source A and stores the result in the destination
MUL	Multiply	Multiplies source A by source B and stores the result in the destination
DIV	Divide	Divides source A by source B and stores the result in the destination and the math register
DDV	Double Divide	Divides the contents of the math register by the source and stores the result in the destination and the math register
CLR	Clear	Sets all bits of a word to zero
SQR	Square Root	Calculates the square root of the source and places the integer result in the destination
SCP	Scale with Parameters	Produces a scaled output value that has a linear relationship between the input and scaled values
SCL	Scale Data	Multiplies the source by a specified rate, adds to an offset value, and stores the result in the destination
ABS	Absolute	Calculates the absolute value of the source and places the result in the destination
CPT	Compute	Evaluates an expression and stores the result in the destination
SWP	Swap	Swaps the low and high bytes of a specified number of words in a bit, integer, ASCII, or string file
ASN	Arc Sine	Takes the arc sine of a number and stores the result (in radians) in the destination
ACS	Arc Cosine	Takes the arc cosine of a number and stores the result (in radians) in the destination
ATN	Arc Tangent	Takes the arc tangent of a number and stores the result (in radians) in the destination
COS	Cosine	Takes the cosine of a number and stores the result in the destination
LN	Natural Log	Takes the natural log of the value in the source and stores it in the destination
LOG	Log to the Base 10	Takes the log base 10 of the value in the source and stores the result in the destination
SIN	Sine	Takes the sine of a number and stores the result in the destination
TAN	Tangent	Takes the tangent of a number and stores the result in the destination
XPY	X to the Power of Y	Raise a value to a power and stores the result in the destination”

Some instructions are available for all processors. Some of the more advanced instructions are only available with the more powerful processors.

An example of a math block used in logic follows:

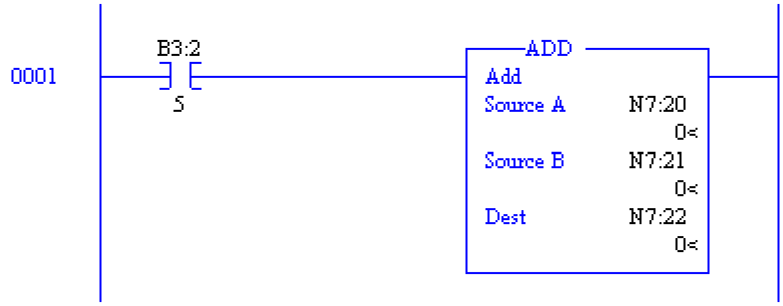


Fig. 8-14 A-B SLC ADD Instruction

The ADD Block above adds the contents of N7:20 to the contents of N7:21 and saves the result in location N7:22. This occurs only when B3:2/5 is true.

Using one-shot logic with the ADD Block:

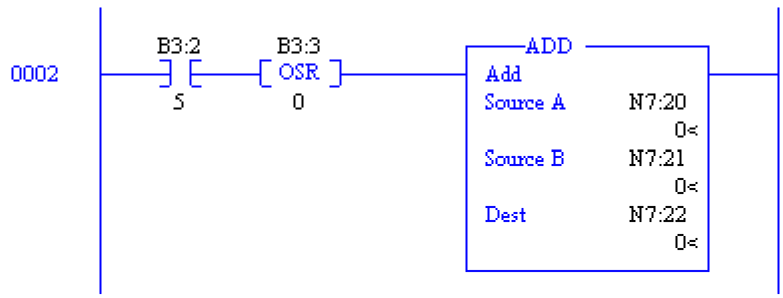


Fig. 8-15 A-B SLC ADD Instruction with One Shot

The ADD Block above adds the contents of N7:20 to the contents of N7:21 and saves the results in N7:22 only on the leading edge of B3:2/5. The use of the one-shot allows only a single occurrence of the calculation and is a very efficient way to execute math operations.

Use of the Continuous Execution Math Block:

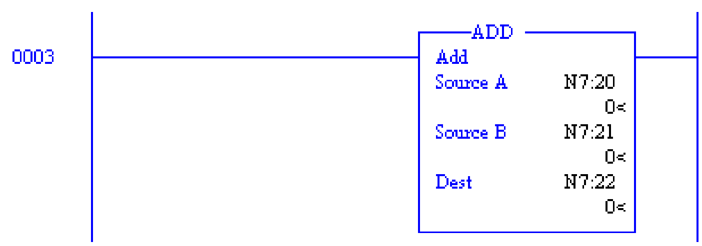


Fig. 8-16 A-B SLC Continuous ADD

The ADD Block above adds the same data as earlier. In this example, however, execution of the ADD Block occurs continuously (once each scan).

Use of Constants in Math Blocks:

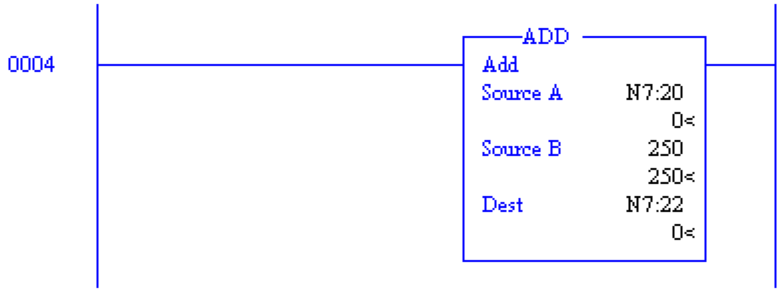


Fig. 8-17 A-B SLC Using Constants in ADD Block

In this example, the ADD Block adds a variable (N7:20) to a constant (250) and places the results in a variable location (N7:21).

The decision must be made when programming how to treat constants. If the number in Source B above is never changed, then entering 250 into Source B is the preferred approach. If the number is to be changed at a later date, however, use the addressing approach below and store the number 250 in N7:21.

There is a dilemma with entering a constant in a storage location. This occurs if a second person changes the value in the N7 location at a later date. The documentation of the listing does not keep the constant's value and the value of the constant 250 is lost unless it is kept in a rung comment (a very good reason to have rung comments). A compromise is reached with the addition of a rung to load the constant's value to the variable location at power-up or with the restart of RUN mode. This approach employs use of the first scan bit, S:1/15.

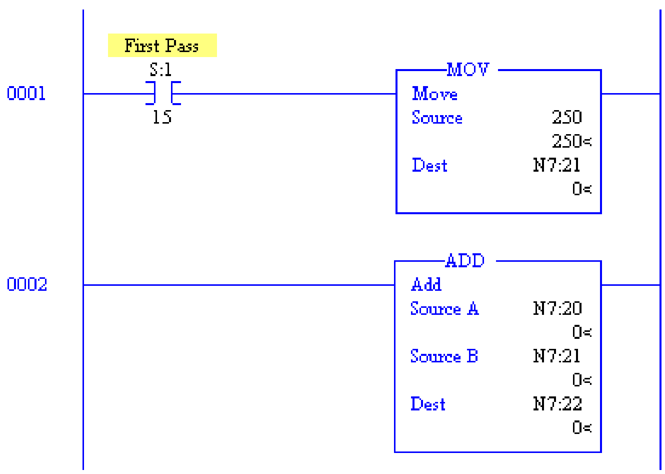


Fig. 8-18 A-B SLC Initializing a Constant in ADD Block

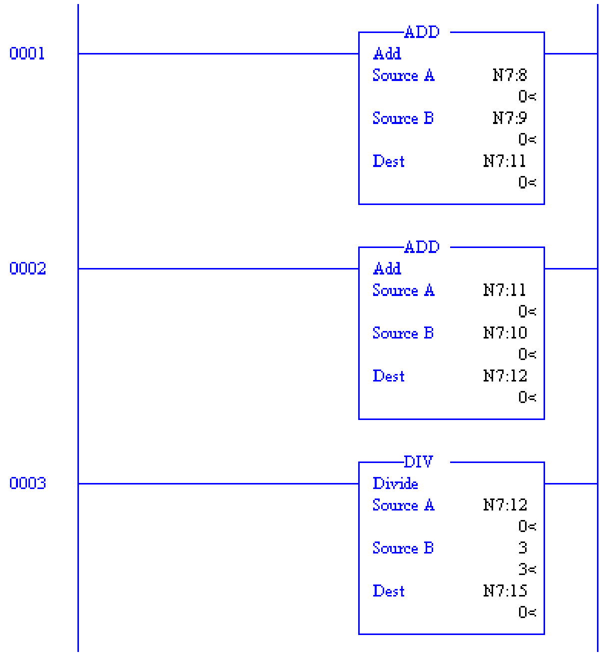
The above two rungs show an example of an ADD Block using a constant that may be changed but is restored each occurrence that power is restored or the processor is returned to the RUN mode.

Multiple Step Calculations

A calculation may require more than one block as shown below in Fig. 8-8. If the file N7 is used, the address refers to an integer. If the file F8 (not applicable to MicroLogix processors), the number refers to a floating point number. Mixing of integer and floating-point numbers in the operation may be used if floating point is allowed by the processor. Some of the instructions listed in Table 8-1 are not allowed on the MicroLogix processors.

Example of Use of Calculation Blocks:

Three numbers are stored in N7:8, N7:9 and N7:8. Find the average of these three numbers using PLC logic. Store the results in N7:15.



In this example, the average of the numbers in N:8, N7:9 and N7:10 are found. The average is continuously being calculated and stored in N7:15.

Temporary storage locations are used to hold partial results of the operation. These locations are N7:11 and N7:12.

Fig. 8-19 Multiple Blocks Perform Single Operation

Compare Instructions

Another group of instructions use math expressions as contacts and are listed below. They form the group of Comparison Instructions:

“Mnemonic	Name	Purpose
EQU	Equal	Test whether two values are equal
NEQ	Not Equal	Test whether one value is not equal to a second value
LES	Less Than	Test whether one value is less than a second value
LEQ	Less Than or Equal	Test whether one value is less than or equal to a second value
GRT	Greater Than	Test whether one value is greater than another
GEQ	Greater Than or Equal	Test whether one value is greater than or equal to a second value
MEQ	Masked Comparison for Equal	Test portions of two values to see whether they are equal. Compares 16-bit data of a source address to 16-bit data at a reference address through a mask
LIM	Limit Test	Test whether one value is within the limit range of two other values”

An example of the Compare Block is seen below:

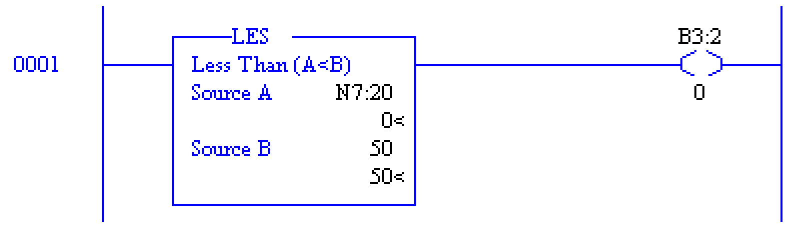


Fig. 8-20 A-B SLC Compare Block

This rung turns on B3:2/0 as long as the number in N7:20 is less than 50. N7:20 ranges from -32768 to +32767

Math instructions are found at the output of a rung while comparison instructions are found as contacts that allow the output to operate under certain conditions. In the example, when the number in N7:0 is less than the number 100, the subtraction block will be performed. The subtraction block subtracts the number in N7:3 from the number in N7:2 and stores the result in N7:4.

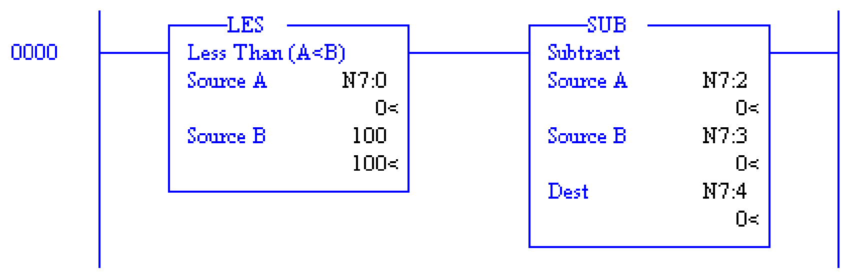


Fig. 8-21 A-B SLC Compare Block followed by Math SUB

Mixing of Relay Contacts and Compare Instruction:

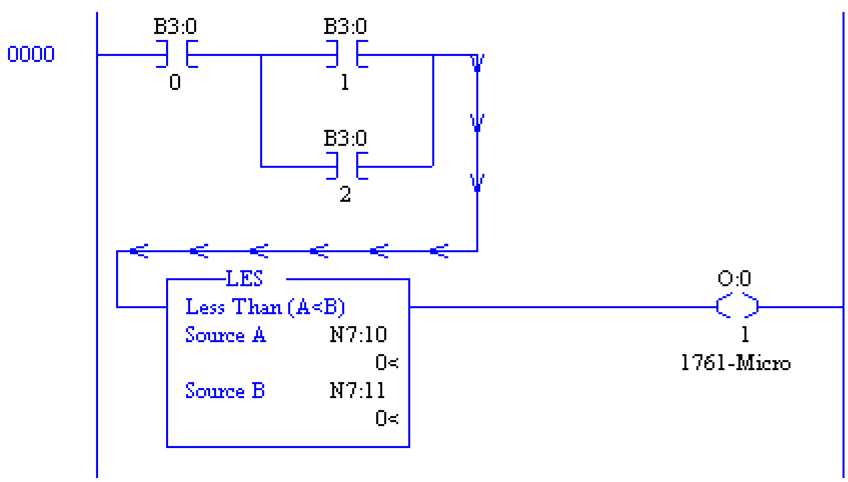


Fig. 8-22 A-B SLC Mixed Contact and Compare Logic

The circuit above shows the combination of Comparison and contact logic turning on an output (O:0/1).

Multiple Compare Instructions:

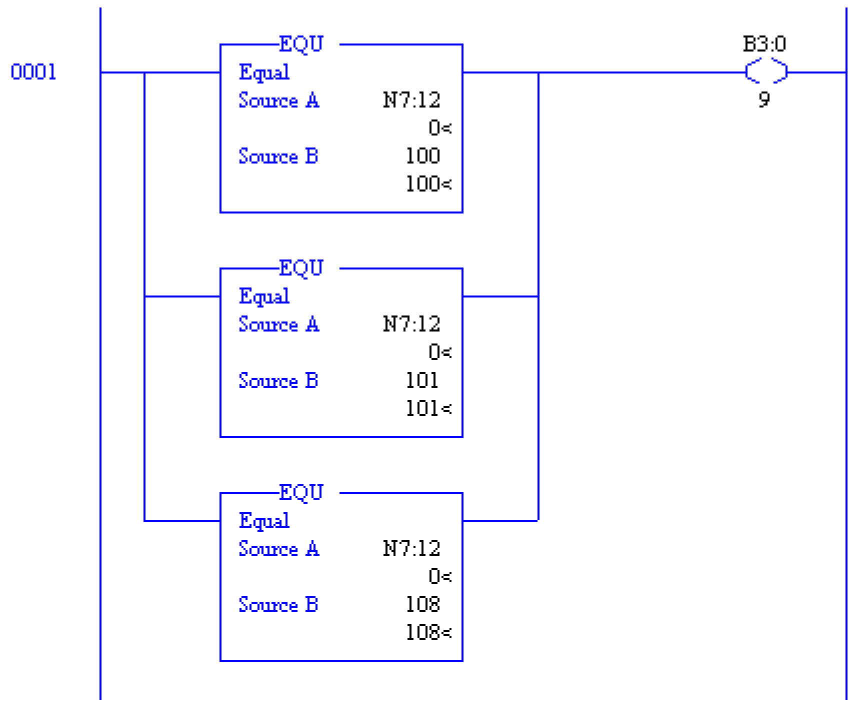


Fig. 8-23 A-B SLC Multiple Compare Logic

The circuit above gives a combination of Comparison statements that turn on output B3:0/9. If N7:12 is equal to any of the values 100, 101, or 108, the output turns on. This is an example of 'OR' logic. If N7:12 = 100 or N7:12 = 101 or N7:12 = 108, then turn on B3:0/9.

LIMIT Compare Block

The Limit Test (LIM) uses three values to determine if a value is within a certain limit. The three values are:

- Low Limit
- Test Value
- High Limit

If the test parameter is a constant, then both the low and high limit values must be word addresses. If the test parameter is a word address, then the low and high limit values may be either a constant or word address.

The LIM instruction passes power when the test value is between the two limits of the lower and upper limit. It is false when either above the high limit or below the lower limit. When the lower limit is greater than the upper limit, LIM true when the test value is greater than the lower number (stored in the upper limit) or less than the higher number (stored in the lower limit).

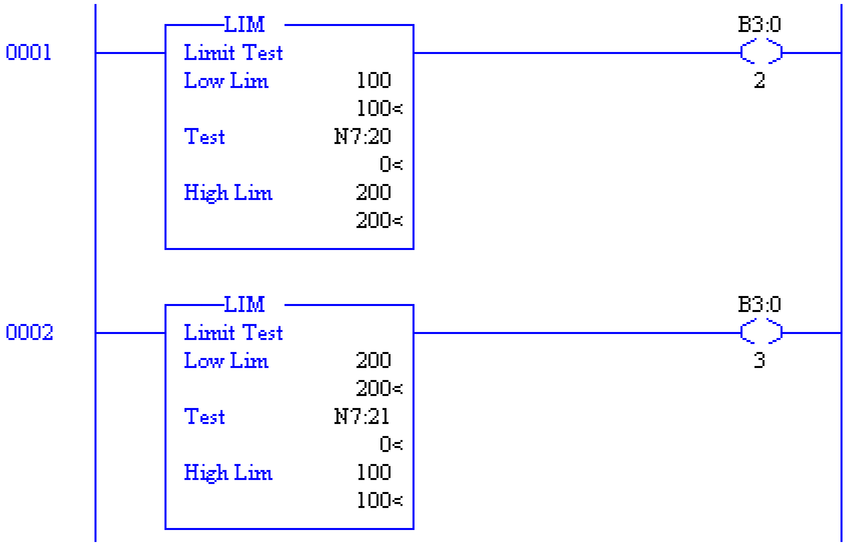


Fig. 8-24 A-B SLC Limit Test Instruction

For example, in rung 0001 above, B3:0/2 will turn on if the value in N7:20 lies between 100 and 200. In rung 0002, B3:0/3 will turn on if the value in N7:21 is greater than 200 or is less than 100.

Other Compare Examples

To insert high and low limits on a number, use the following circuit:

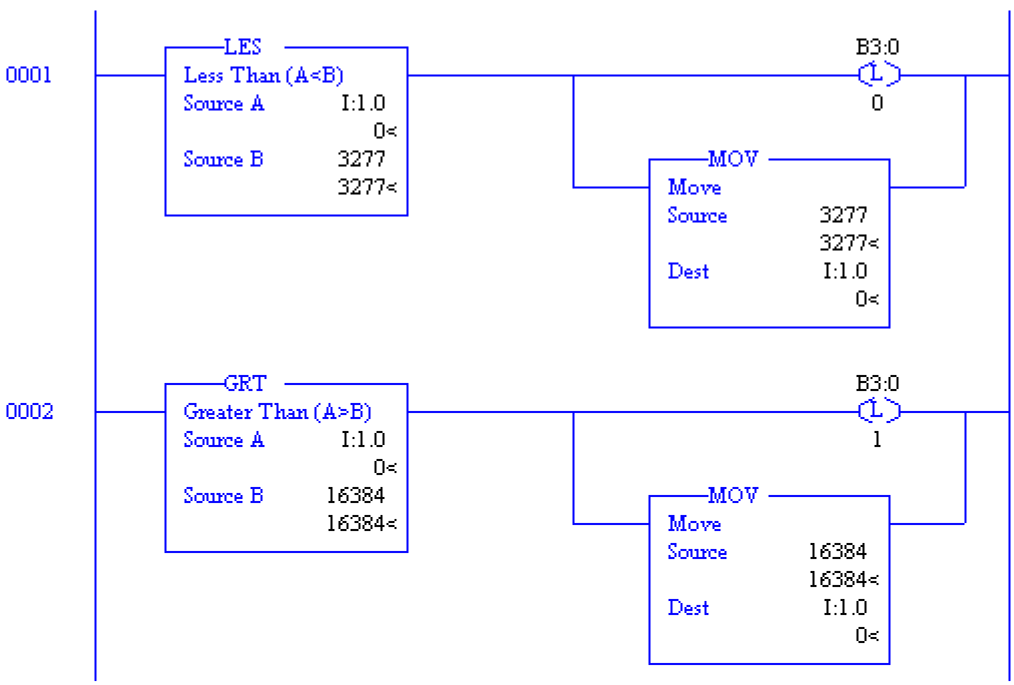


Fig. 8-25 A-B SLC High and Low Limit Test Instructions

When the number input into I:1.0 ranges less than 3277 or higher than 16384, the number will be limited to the high value of 16384 or to the low value of 3277.

Comparison values to turn on a valve:

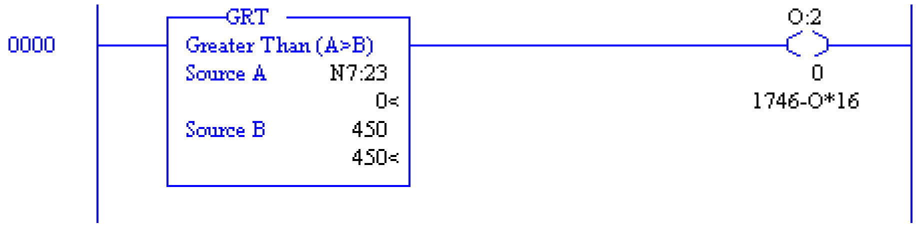
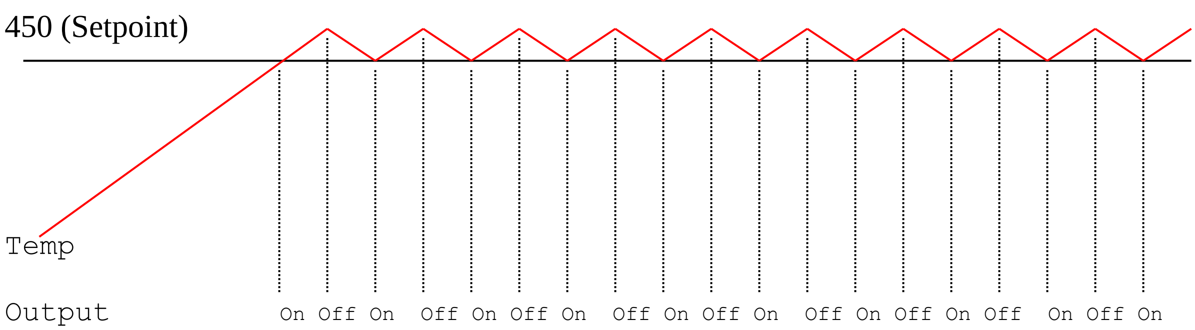


Fig. 8-26 A-B SLC Comparision to Turn on Valve

While the circuit above may be useful in turning on or off an output, usually a dead band must be inserted in the circuit so that output O:2/0 will not chatter on and off. Results of the circuit of are shown in the graph below. Note the chatter as the output turns on and off at a high frequency. To slow the frequency to a much slower rate, the number in Source B must be made to vary and change so that the output must climb higher than 450 when increasing but turn off at a lesser value than 450 when falling.

For instance, if the output provided cooling to a process and the number 450 represents a temperature setpoint. N7:23 represents the value of the temperature of the process. If the temperature rises above 450 degrees, then turn on some cooling through output O:2/0. The circuit may work well as is, but if the output tends to chatter on and off, a dead-band circuit must be provided.



Output turns On-Off per Rung 0000.

Fixing the High-Frequency On-Off with a Dead Band:

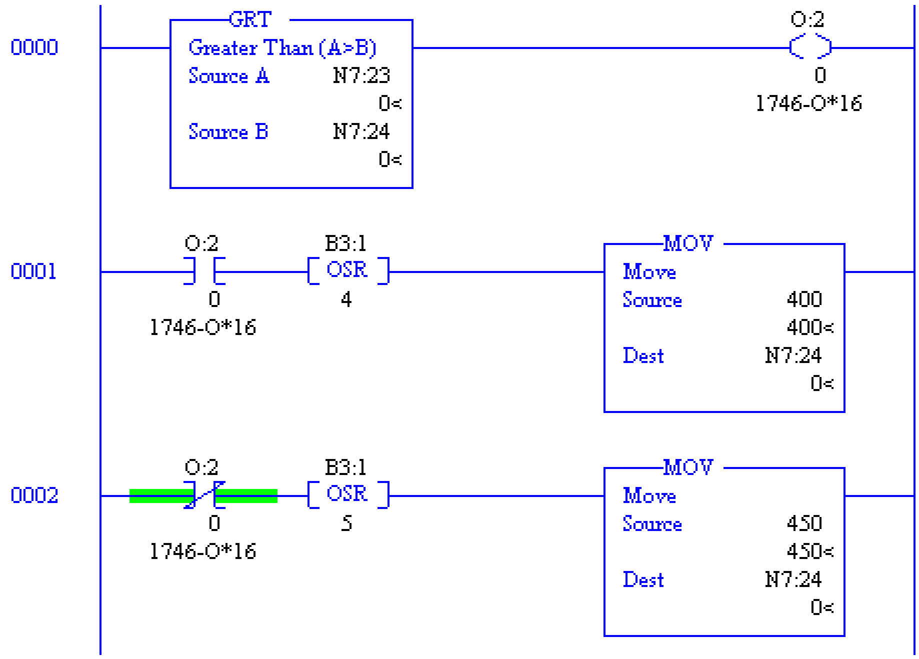
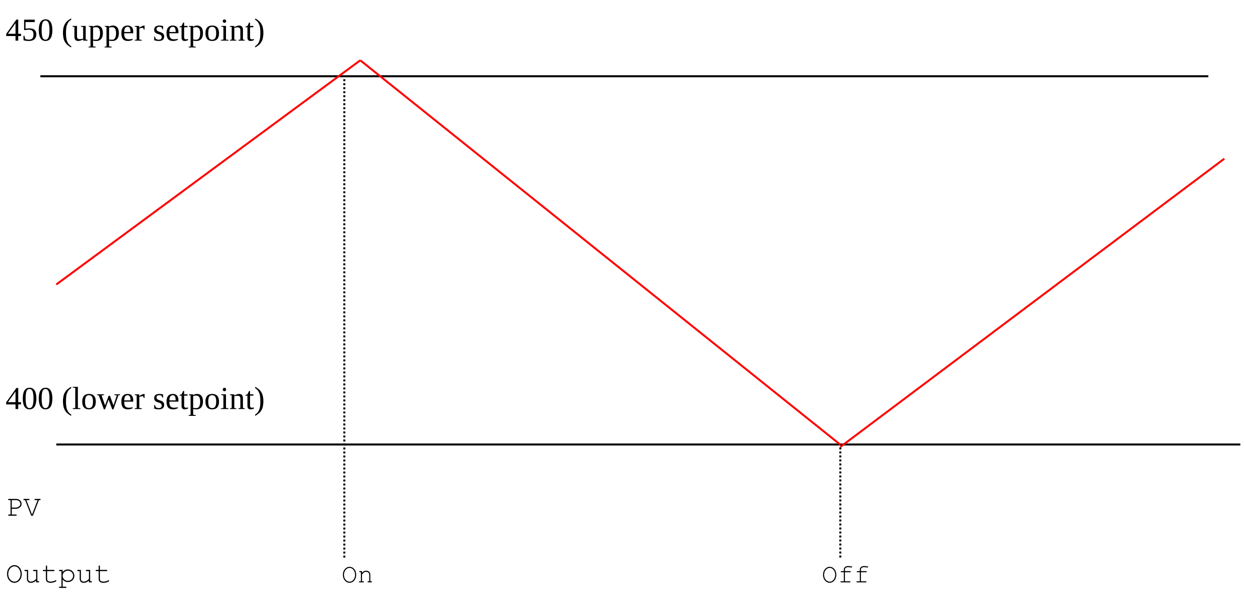


Fig. 8-27 A-B SLC Logic for Dead Band Application

The circuit above provides a dead-band between 400 and 450 degrees. When the temperature rises above 450, the coolant solenoid turns on. When the temperature decreases to 400 degrees, the coolant turns off. Providing a dead-band keeps the coolant solenoid from constantly turning on and off. Care must be taken to initialize the circuit above to 450. One approach to this is with a rung triggered by S:1/15 (First Scan). One could also place 450 into N7:24 manually. The second approach is not as trustworthy, however, since programs tend to be written over and constants lost with programs that change after the program has been working for days or even years.



Use of Memory using Latch or Seal with Compare

Combination of memory circuits with compare operations was subtly introduced in the example of the valve above. The two figures below provide functionally equivalent control of the dead-band problem with the temperature controller. However, the circuits employ standard memory logic to provide the same functional dead-band control.

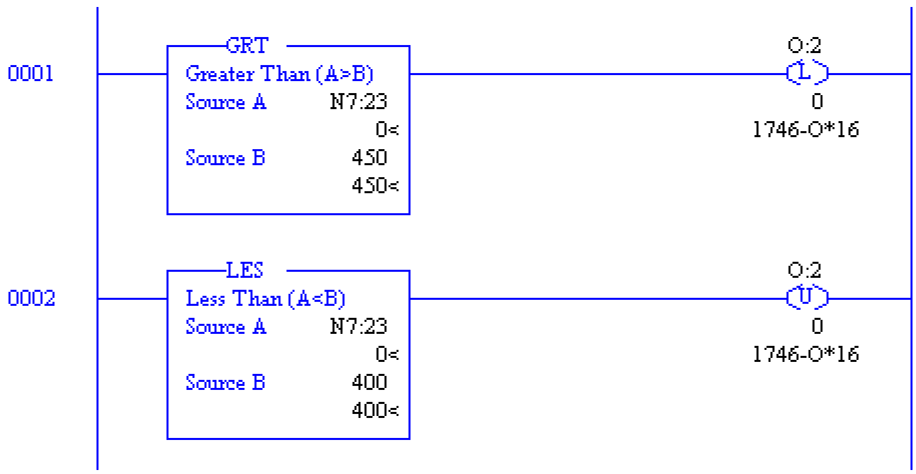


Fig. 8-28 A-B SLC Logic for Dead Band Application – a Second Approach

This program uses latch-unlatch logic to turn on and off the output with a dead-band of 50 between the turn-on at 450 and turn-off at 400.

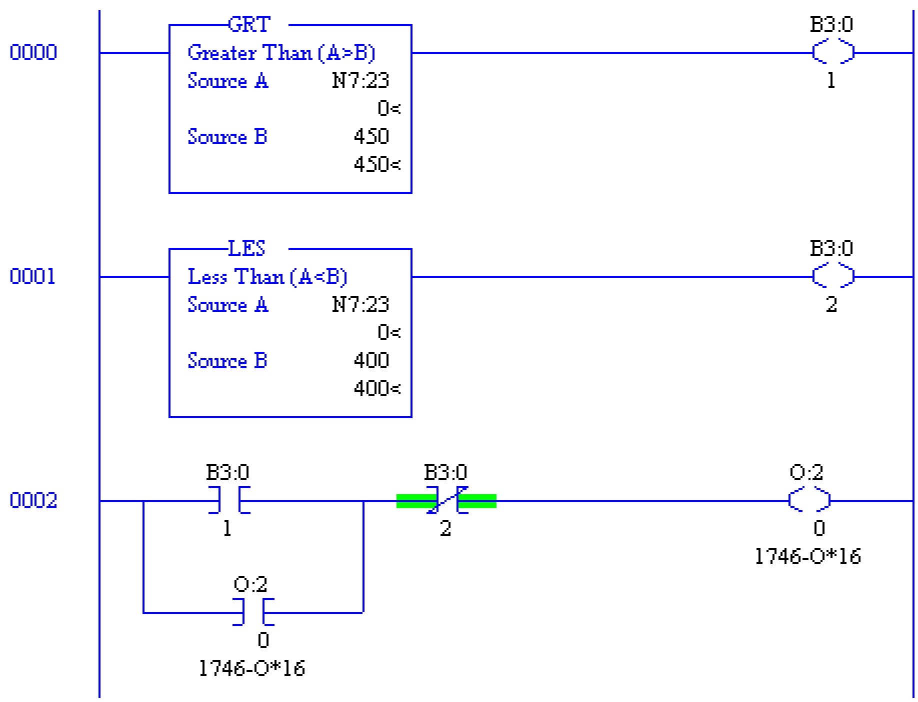


Fig. 8-29 A-B SLC Logic for Dead Band Application – a Third Approach

This program uses seal-circuit logic to turn on and off the output with a dead-band of 50 between the turn-on at 450 and turn-off at 400. Seal-circuit logic may be the preferred logic to use since it needs little additional logic to work correctly under all conditions.

Start-up conditions must be considered for each of the three approaches. With the first circuit, the initial status of N7:23 must be considered. Location N7:23 should be initialized to 450. With the latch logic and the seal logic, the circuit will automatically initialize to a known value in all cases which is preferred. In general, using a seal circuit eliminates unnecessary logic to initialize the logic.

What is used as a memory component in the first of the three circuits above?

Status Table

When an arithmetic operation is performed, results from the operation are stored in the Status Table in a manner similar to the operation of a microprocessor. If two numbers are added, the carry and overflow are set or reset. Other operations use similar status bits. They are reviewed in the Table 8-3 below: (from Allen-Bradley's SLC-500 Instruction Reference Manual)

"With this Bit:		The Controller:
S:0/0	Carry (C)	Sets if carry is generated; otherwise cleared
S:0/1	Overflow (V)	Indicates that the actual result of a math instruction does not fit in the designated destination
S:0/2	Zero (Z)	Indicates a 0 value after a math, move, or logical instruction
S:0/3	Sign (S)	Indicates a negative (less than 0) value after a math, move, or logic instruction
S:5/0	Minor Error Overflow	Set upon detection of a mathematical overflow or division by zero. If set upon execution of an END statement or a Temporary End (TND) instruction, or an I/O Refresh (REF), the recoverable major error code 0020 is declared."

Also, from the same manual, on Changes to the Math Register, S:13 and S:14, the status words for long integer math are described as follows:

"Status word S:13 contains the least significant word of the 32-bit values of the MUL and DDV instructions. It contains the remainder for DIV and DDV instructions. It also contains the first four BCD digits for the Convert from BCD (FRD) and Convert to BCD (TOD) instructions.

Status word S:14 contains the most significant word of the 32-bit values of the MUL and DDV instructions. It contains the unrounded quotient for DIV and DDV instructions. It also contains the most significant digit (digit 5) for the TOD and FRD instructions."

Another table from the same reference book describes the math overflow or 32-bit integer math function. It is as follows:

"Address	Classification	Description
S:2/14	Dynamic Config	Math Overflow Selection Bit Set this bit when you intend to use 32-bit addition and subtraction. When S:2/14 is set, and the result of an *ADD*, *SUB*, *MUL*, or *DIV* instruction cannot be represented in the destination address (underflow or overflow), - the overflow bit S:0/1 is set. - the overflow trap bit S:5/0 is set, and - the destination address contains the unsigned truncated least significant 16 bits of the result. The default condition of S:2/14 is reset (0). When S:2/14 is reset, and the result of an *ADD*, *SUB*, *MUL*, or *DIV* instruction cannot be represented in the destination address (underflow or overflow), - the overflow bit S:0/1 is set, - the overflow trap bit S:5/0 is set, and - the destination address contains 32767 if the result is positive or –32768 if the result is negative. Note: The status of bit S:2/14 has no effect on the *DDV* instruction. Also, it has no effect on the math register content when using *MUL* and *DIV* instructions. To program this feature, use the Data Monitor function to set or clear this bit. To provide protection from inadvertent data monitor alteration of your selection, program an unconditional *OTL* instruction at address S:2/14 to ensure the new math overflow operation. Program an unconditional *OTU* instruction at address S:2/14 to ensure the original math overflow operation."

This table shows a description of the operation to set S:2/14 when performing a 16 bit non-signed arithmetic operation instead of a 15 bit signed operation.

Status Table Math Section from RSLogix 500

Display of the Math portion of the Status Table can be seen by opening the Status Table file S2 and then the Math tab.

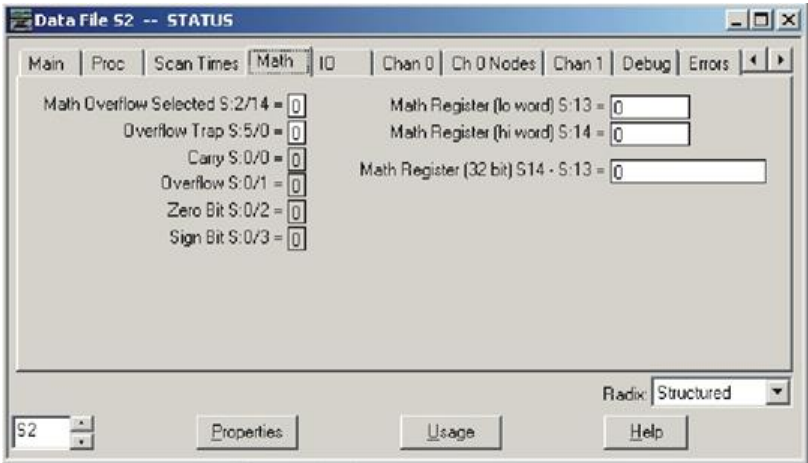


Fig. 8-30 A-B SLC Status Table, Math Portion

Notice the Math Register locations S:13 and S:14. The high order portion of the 32-bit number is saved in the 'hi word' and the low order portion is saved in the 'lo word'.

A very good example of 32-bit addition is included in the Allen-Bradley SLC-500 Reference Manual. The introduction to the section on 32-bit math is important and should be reviewed prior to tackling this subject.

In general, math is done with the programmer choosing to use either integers or floating-point numbers. If possible, always use floating-point math. Integer math constantly must be concerned with overflowing the range of the number 32767 or –32768.

Using Advanced Math Instructions

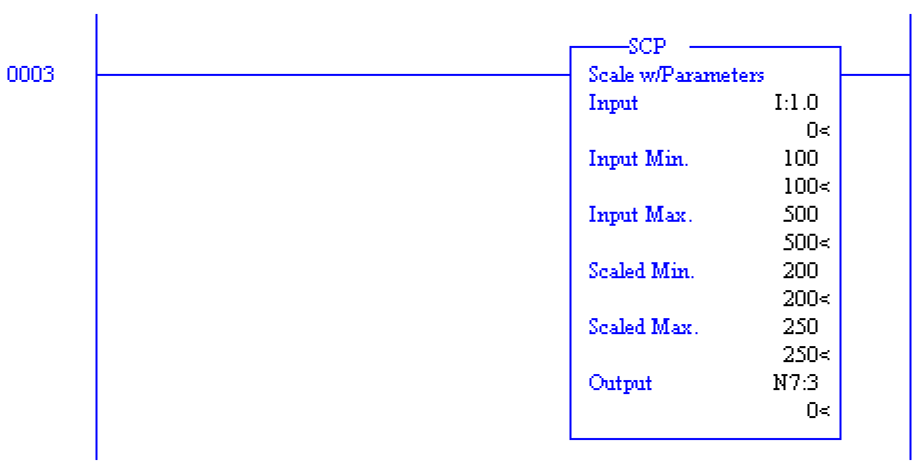


Fig. 8-31 A-B SLC Scale with Parameters Instruction

A very useful command found under the Advanced Math tab of RSLogix 500, is the *SCP* or Scale with Parameters block. The block above scales the input found at I:1.0 from an expected input value ranging from 100 to 500 to an output value ranging from 200 to 250. This function allows for scaling of input and output values, multiplication of any number by a second number with an offset or any other mathematical operation of the form y = mx + b without performing the operation of the programmer mathematically calculating m or b.

Math Overflow Problem

Rungs should be examined before entered to insure that the rung will not allow a numeric overflow. The following is an example of a circuit that may experience an overflow:

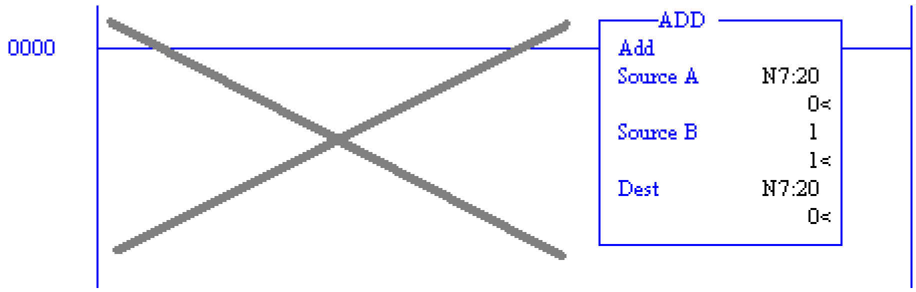


Fig. 8-32 A-B SLC Math Overflow Problem

In the figure above, after the processor is turned to the run mode, the ADD Block of rung 0000 is executed each scan rung 0000 is executed. Each scan the *ADD* Block is executed and 1 is added to N7:20. Since the result is stored in N7:20, this location is incremented by 1. Since a scan may take approximately 5 msec, in about 160 seconds, the number in N7:20 will reach 32767. This instruction will fault the machine and the processor will turn off. The fault light will turn on.

Care must be taken to not program rungs such as rung 0000 above unless the contents of N7:20 are reset or limited in another rung. Use one-shots before math operations as a general rule to avoid situations such as the mistake above.

On the other hand, the use of a rung such as the one above may be useful in a program to calculate time duration of a scan. If properly used with a timer, the ADD Block may be programmed to calculate the average scan time of the program.

Status Table Contents on Faults

To clear a fault that stops the program from running and to diagnose the problem, click on S2 – STATUS in the Data Files section of the Project Tree. Then click Errors and read the description and analysis of the error in the window. An alternate method is simply reset the error from the Command Line and hope the fault doesn't reoccur. It usually does. Make that, it always does.

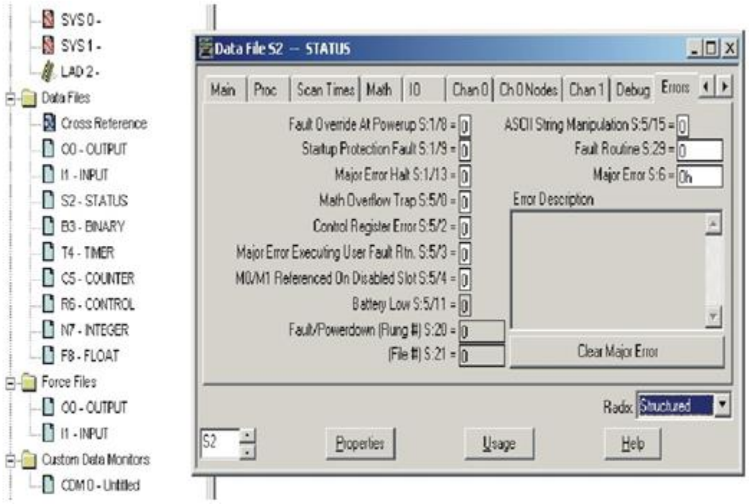
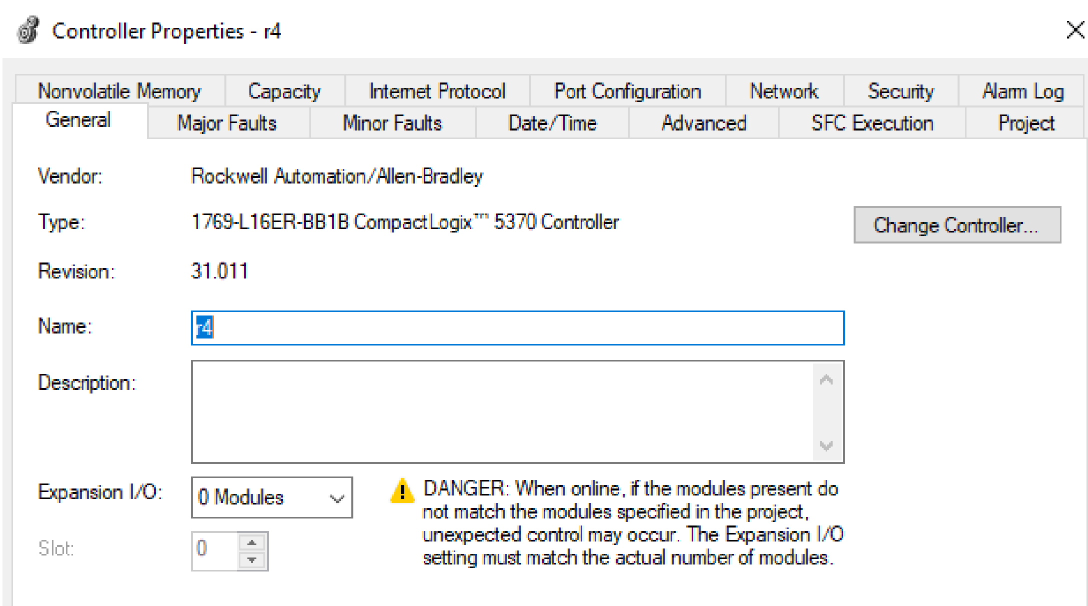


Fig. 8-33 Errors are described in the Error Description window.

The CompactLogix Processor L16ER-BB1B gives the following properties page which is similar to the above SLC status page:



The RSLogix help topic "Access Runtime Controller Configuration and Status" describes the following:

S:FS	First Scan flag
S:N	Negative flag
S:Z	Zero flag
S:V	Overflow flag
S:C	Carry flag
S:MINOR	Minor Fault flag

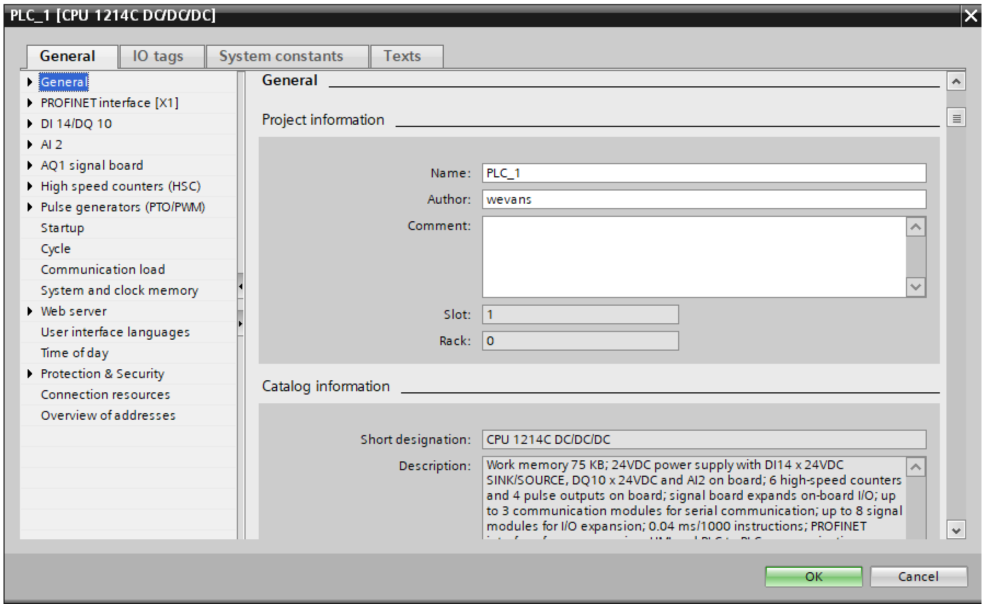
Any other processor information must be obtained by using the GSV (get system variable) instruction.

When entering a GSV/SSV instruction, specify the object and its attribute to access. In some cases, there will be more than one instance of the same type of object. Be sure to specify the object name. For example, each task has its own TASK object that requires specifying the task name to gain access.

These are the GSV/SSV objects. The objects available for access are dependent on the controller.

AddOnInstructionDefinition
Axis
Controller
ControllerDevice
CoordinateSystem
CST
DF1
FaultLog
HardwareStatus
Message
Module
MotionGroup
Program
Redundancy
Routine
Safety
SerialPort
Task
TimeSynchronize
WallClockTime

Siemens gives similar results with the following pages when choosing the properties tab. We are not able to see every similar device for Siemens listed above. However, the help function for TIA provides a broad range of options for finding similar functions. We will discuss in Chapter 18 OB100, the Siemens start-up object block. This is the block containing instructions for start-up of the machine or re-initializing the machine after a shutdown.



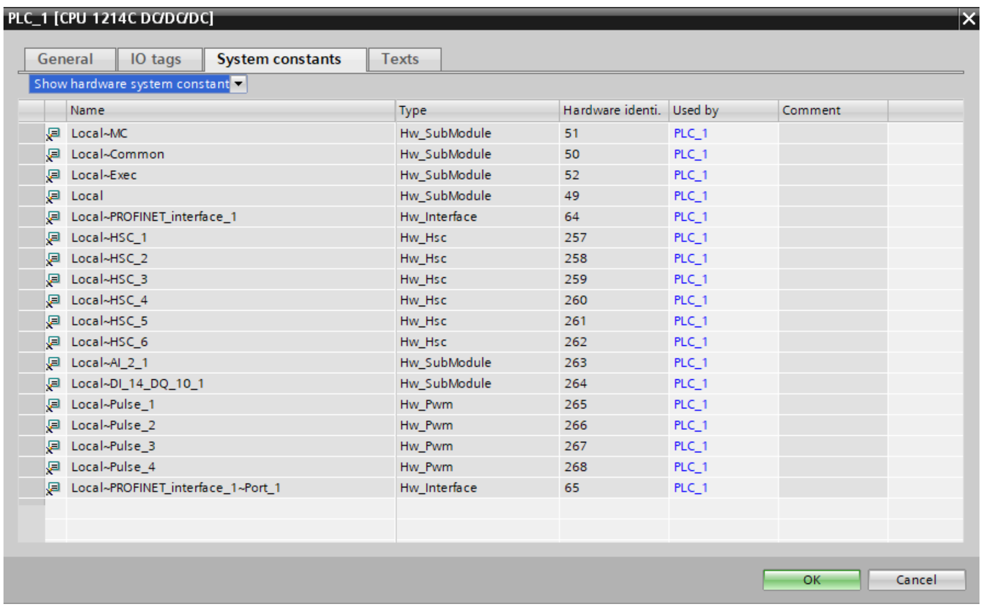


Fig. 8-34 Siemens Properties Pages

Use of Add Blocks

Use of Add blocks using the same Source and Destination addresses may be used. When programmed, a one-shot should be added to guarantee that the Add block does not continue for many scans. As an example, consider using a button to add 5% or 1% to a value in N7:29 in the range 1000 to 5000. The range of the ADD is 40 if 1% is chosen or 200 if 5% is chosen.

The circuit may be programmed as follows:

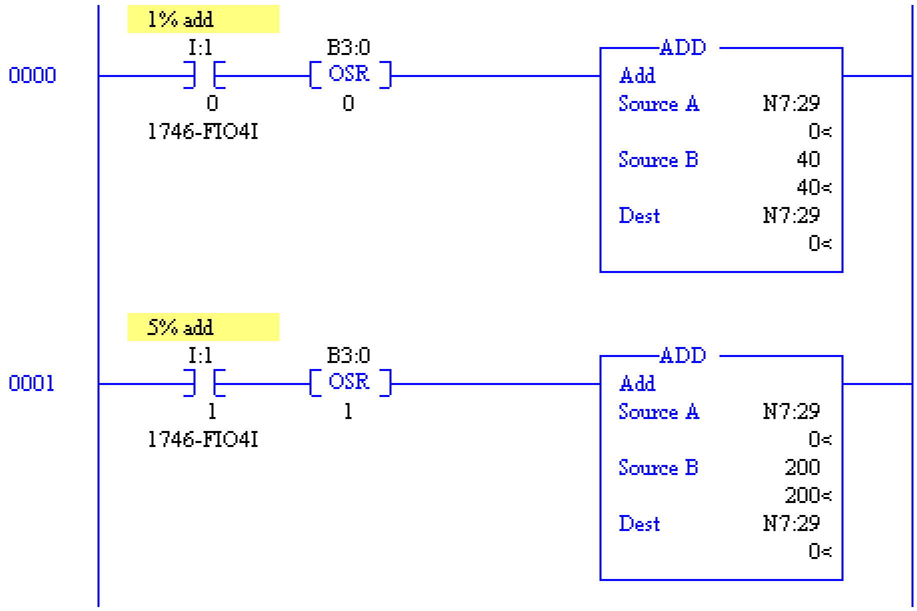


Fig. 8-35 Example of 1% or 5% ADD for Variable

The two ADD Blocks above increment a number in N7:29 by either 1% or 5%.

The following circuit decrements the number in N7:29 by 1% or 5%.

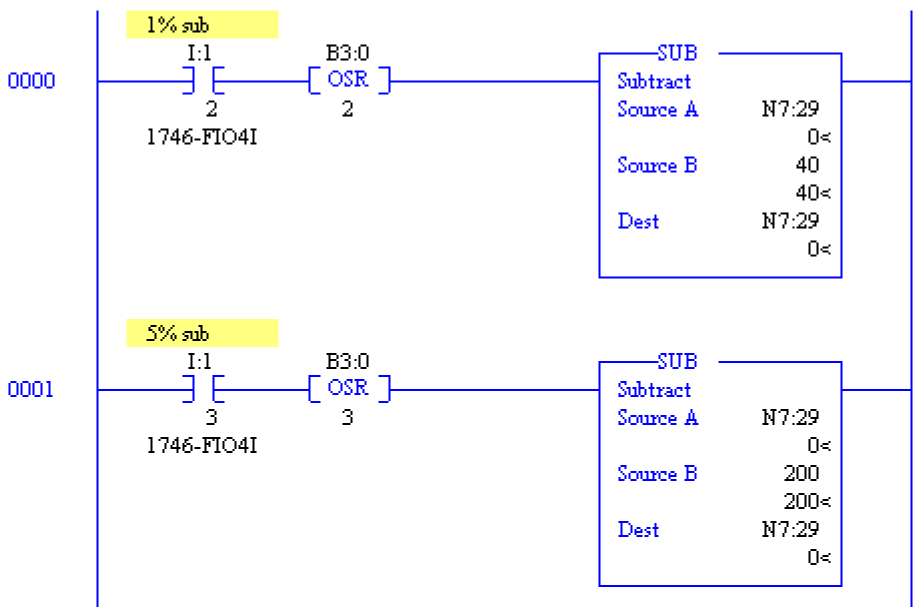


Fig. 8-36 Example of 1% or 5% Decrement for Variable

To protect the limits of 1000 or 5000 in N7:29, add the following:

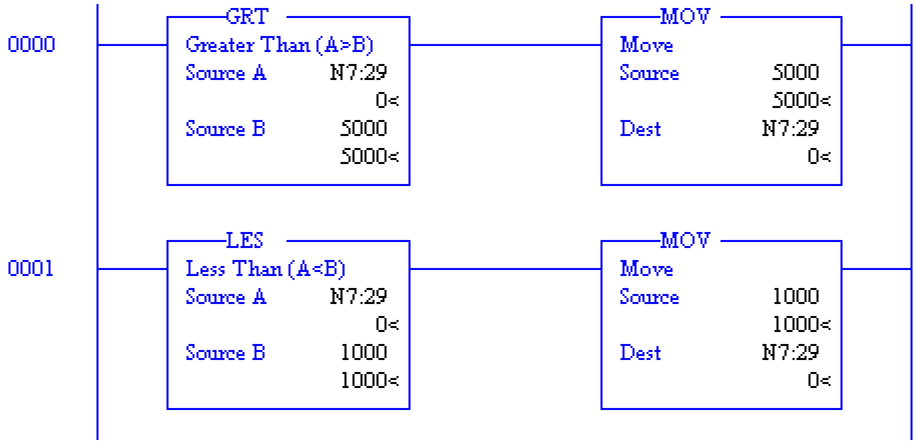


Fig. 8-37 Example of Limit Protection for Variable

CPT Calculation Block

The CPT or Compute instruction is a very flexible instruction that is accessible only on SLC 5/03 through 5/05 processors. It provides expression statements similar to a computer language.

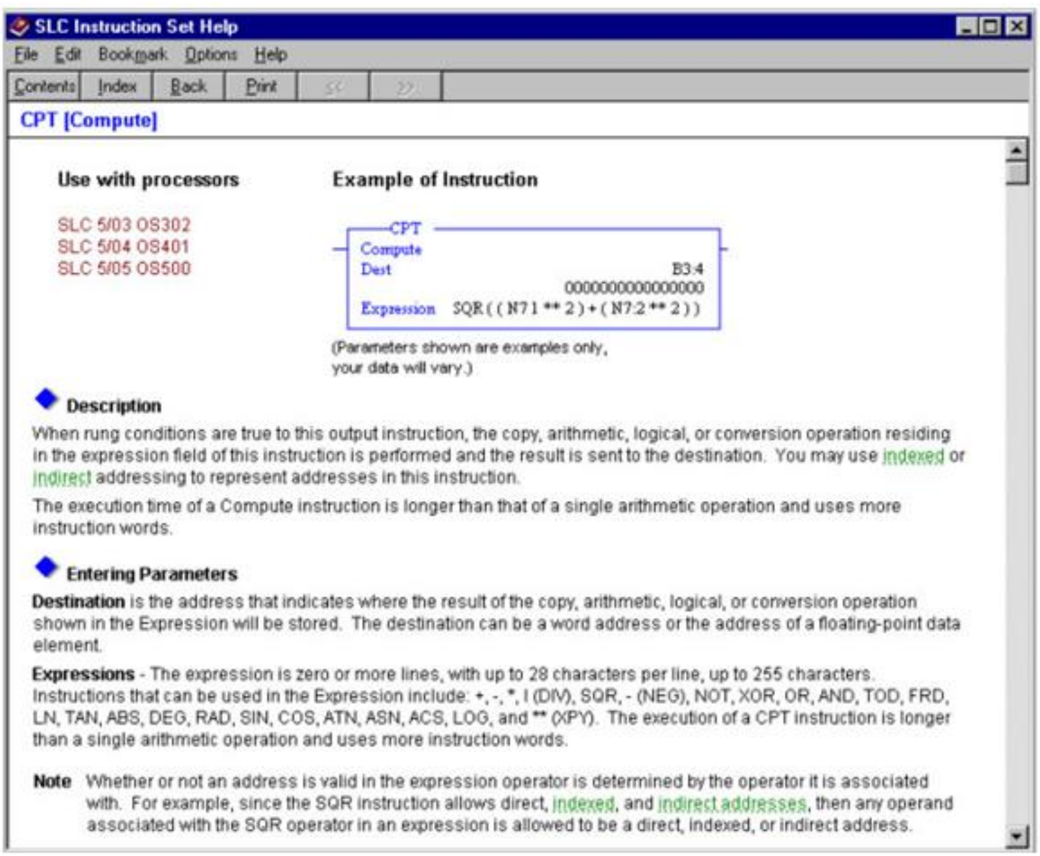


Fig. 8-38 The Compute Block Instructions

The CPT or Compute instruction is a very flexible instruction that is accessible only on SLC 5/03 through 5/05 processors. It provides expression statements similar to a computer language. Multiple variables may be manipulated similarly to a statement in the language BASIC.

Mixing Counters and Math

With the math instructions available, it is now possible to combine circuits using relay contacts and coils, timers, counters and math operations. Coordination between the various rungs will continue to become more complicated as each new instruction is introduced.

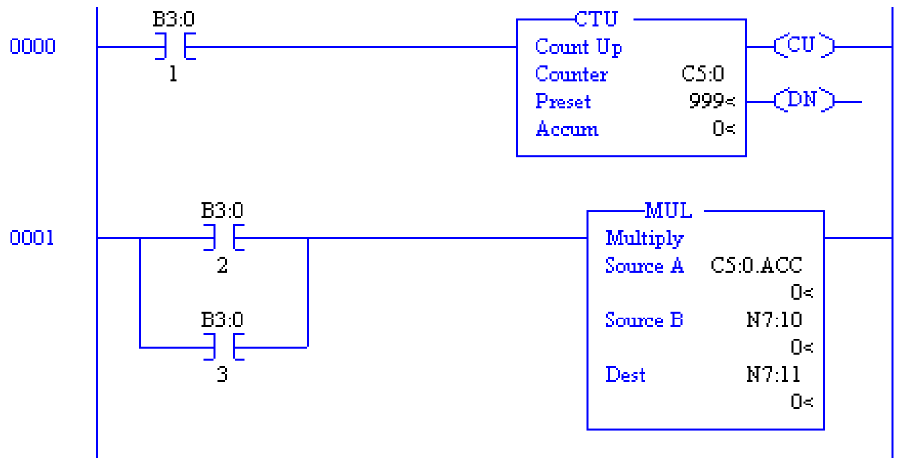


Fig. 8-39 Example of Mixing Counting and Math

Mixing of Data Types

A very good feature of the PLC/5 and SLC architecture in general is the ability to mix numeric types in the same instruction. For instance, the instruction below adds the contents of F8:2 to N7:10 and places the results in B3:18. As can be seen, 0 + 0 = 0.

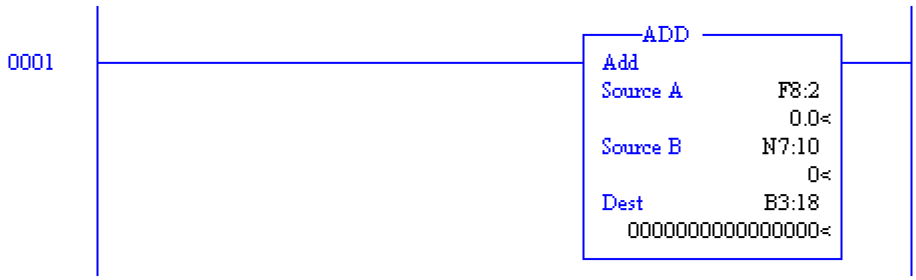
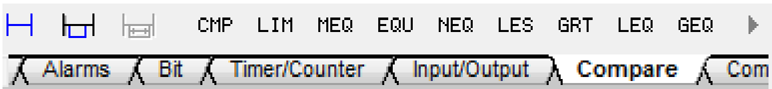


Fig. 8-40 Example of Mixing of Data Types

To mix numeric types, the difference between integer and floating point numbers must be taken into consideration. If an F location is added to an N location and the result is placed in an F location, the fraction will be kept. However, if the result is placed in an N location, the fraction will be lost. If this is desired, then mixed formatting is to be used.

CompactLogix Math Instructions



Compare

If the CMP instruction finds the expression true, the rung-condition-out is set to true.

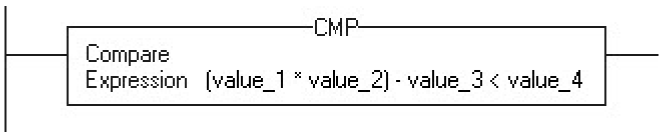


Fig. 8-41 A-B CompactLogix Compare Instruction

If you enter an expression without a comparison operator, such as value_1 + value_2, or value_1, the instruction evaluates the expression as follows:

If the expression is non-zero, tung-condition-out is set to true. If the expression is zero, the rung-condition-out is set to false. See the following:

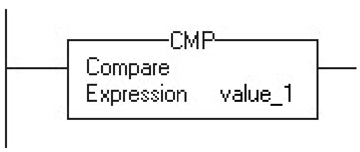


Fig. 8-42 A-B CompactLogix Compare Instruction Compared to Zero

Limit Test (CIRC)

Example of Low Limit ≤ High Limit:
0 ≤ value ≥ 100, set light_1. If value < 0 or value > 100, turn off light_1.

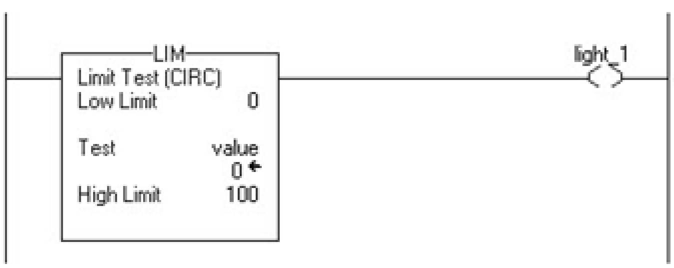


Fig. 8-43 A-B CompactLogix Limit Instruction Testing for Value In Range

Example of Low Limit ≥ High Limit:
When value ≥ 0 or value < -100, set light_1. If value < 0 or value > -100, clear light_1.

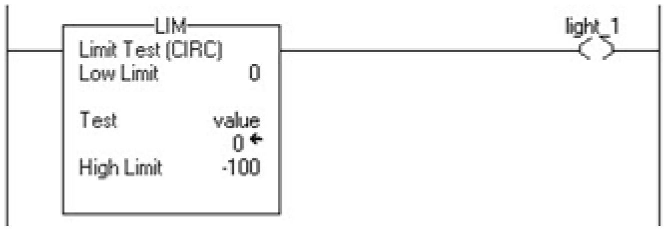


Fig. 8-44 A-B CompactLogix Limit Instruction Testing for Value Outside the Range

Mask Equal

The following example shows a true rung-condition-out for an MEQ command:

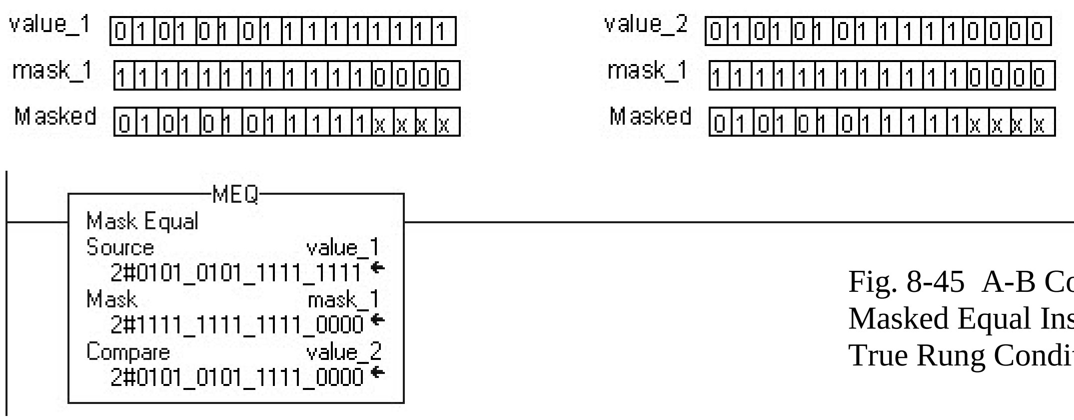


Fig. 8-45 A-B CompactLogix Masked Equal Instruction True Rung Condition

The following example shows rung-condition-out for an MEQ command that failed or was false:

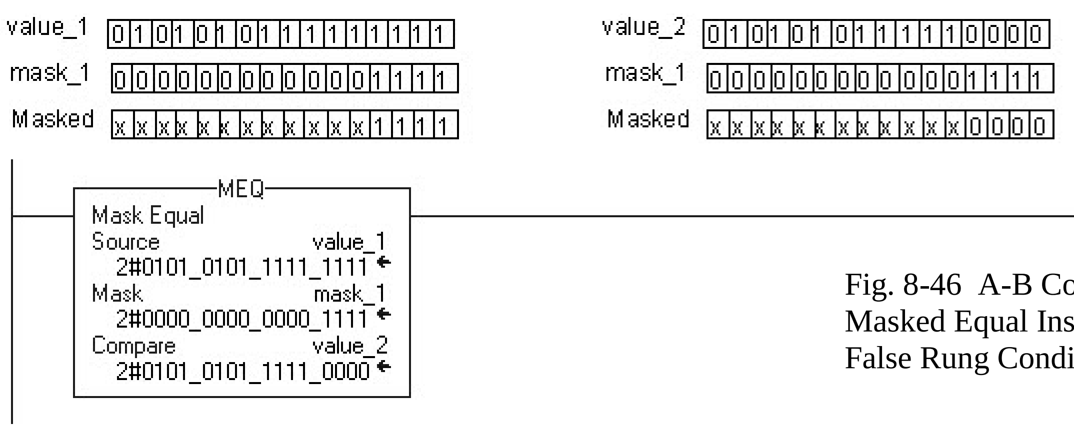


Fig. 8-46 A-B CompactLogix Masked Equal Instruction False Rung Condition

Equal

If value_1 is equal to value_2, set the rung-condition-out to true for the EQU command.

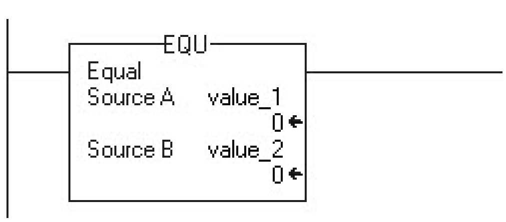


Fig. 8-47 A-B CompactLogix Equal Instruction

Not Equal

If value_1 is not equal to value_2, set the rung-condition-out to true for the EQU command.

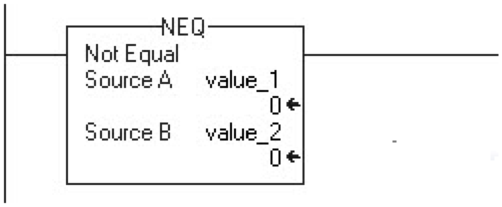


Fig. 8-48 A-B CompactLogix
Not Equal Instruction

Less Than (A<B)

If value_1 is less than value_2, set the rung-condition-out to true for the LES command.

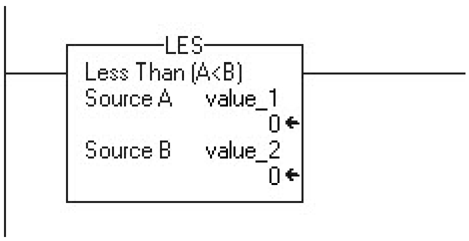


Fig. 8-49 A-B CompactLogix
Not Equal Instruction

Greater Than (A>B)

If value_1 is greater than value_2, set the rung-condition-out to true for the GRT command.

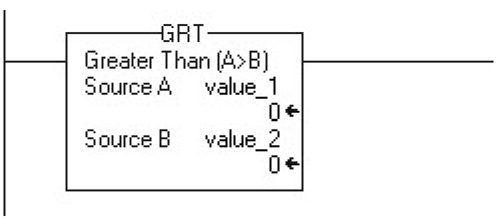


Fig. 8-50 A-B CompactLogix
Greater Than Instruction

Less Than or Eql (A<=B)

If value_1 is less than or equal to value_2, then set the rung-condition-out to true for the LEQ command.

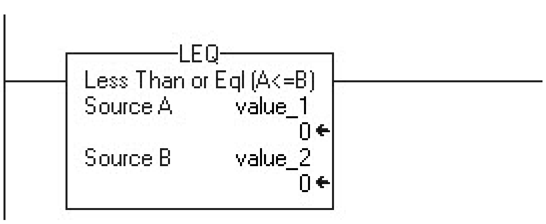


Fig. 8-51 A-B CompactLogix
Less Than or Equal Instruction

Grtr Than or Eql (A>=B)

If value_1 is greater than or equal to value_2, then set the rung-condition-out to true for the GEQ command.

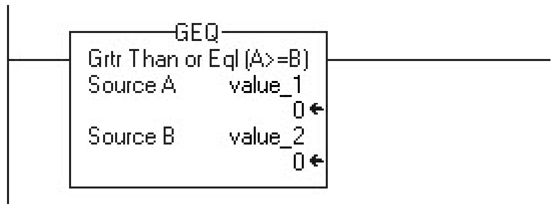


Fig. 8-52 A-B CompactLogix Greater Than or Equal Instruction

Compact Logix Compute/Math Instructions

Compute

For the example, the CPT instruction, evaluates value_1 multiplied by 5 and divides the result by the result of value_2 divided by 7, placing the final result in result_1.

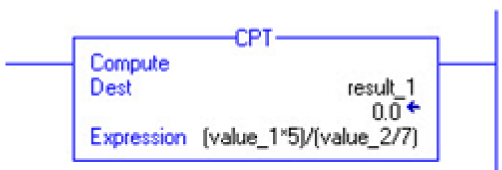


Fig. 8-53 A-B CompactLogix Compute Instruction

Add

The ADD instruction adds Source A to Source B and places the result in the Destination.

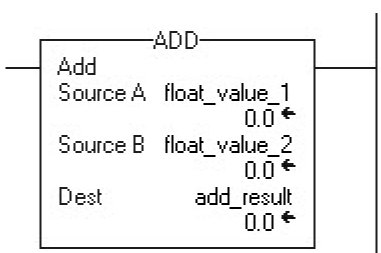


Fig. 8-54 A-B CompactLogix ADD Instruction

Subtract

Subtract float_value_2 from float_value_1 and place the result in subtract_result.

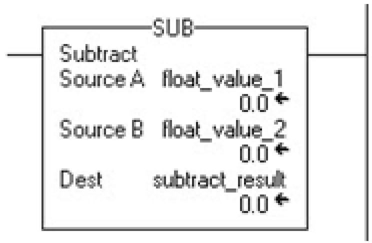


Fig. 8-55 A-B CompactLogix SUB Instruction

Multiply

Multiply float_value_1 by float_value_2 and place the result in multiply_result.

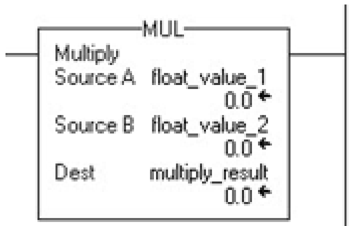


Fig. 8-56 A-B CompactLogix Multiply Instruction

Divide

Divide float_value_1 by float_value_2 and place the result in divide_result.

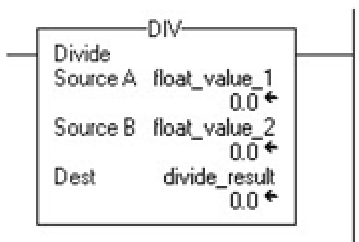


Fig. 8-57 A-B CompactLogix Divide Instruction

Modulo

Divide dividend by divisor and place the remainder in remainder. In the example, 3 goes into 10 three times with remainder 1. The value remainder is saved in the destination.

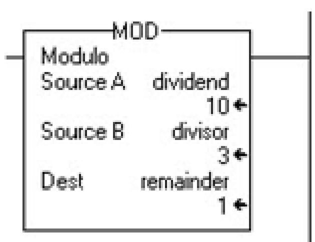


Fig. 8-58 A-B CompactLogix Modulo Instruction

Square Root

Calculate the square root of value_1 and place the result in sqr_result.

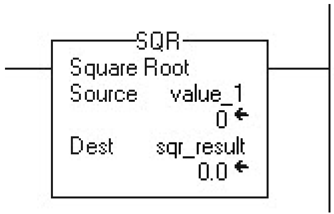


Fig. 8-59 A-B CompactLogix
Square Root Instruction

Negate

Change the sign of value_1 and place the result in negate_result.

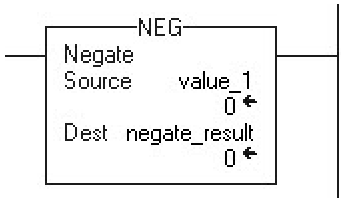


Fig. 8-60 A-B CompactLogix
Negate Instruction

Absolute Value

Place the absolute value of value_1 into value_1_absolute. In the example, the absolute value of -4 is +4.

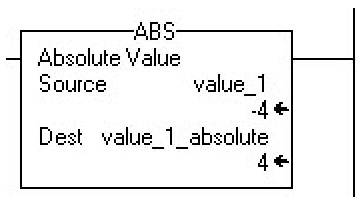
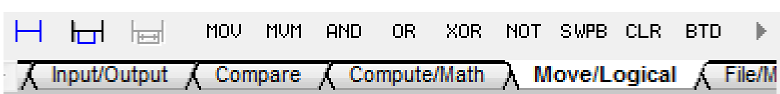


Fig. 8-61 A-B CompactLogix
Absolute Value Instruction

Move Instructions



If you want to:	Use this instruction:
copy a value	MOV
copy a specific part of an integer	MVM
move bits within an integer or between integers	BTD
clear a value	CLR

Move (MOV)

The MOV instruction copies the Source to the Destination. The Source remains unchanged.

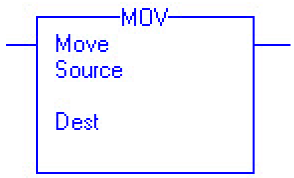
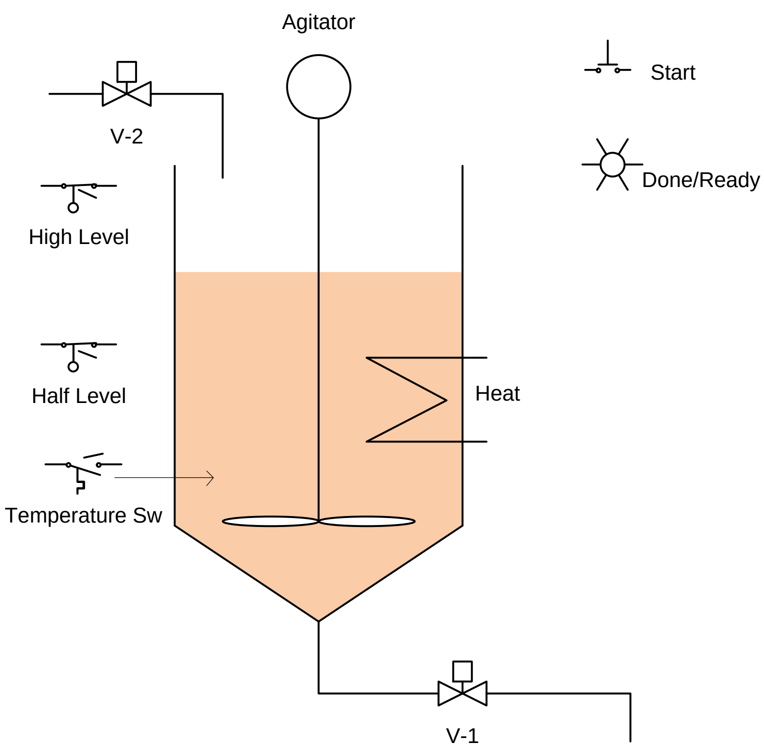


Fig. 8-62 A-B CompactLogix Move Instruction

An Additional Look at the Juice Condenser

The Juice Condenser project was introduced in chapter 5 and discussed again in chapter 6. The juice condenser problem includes memory that the use of numbers encourage a new look.

The operation included a fill, a condensate portion and a drain. These operations were not to be overlaid but rather were to be consecutive. This leads to a memory circuit that includes more than one set of events.



(Fig. 5-1 The Juice Maker)

Fig. 8-63 Processor Needing Numbers for States

Each memory circuit must be exclusive of the other two events and must occur in a proper sequence. For example, the fill operation must occur first, then the condensate operation and finally the drain operation. This may be expressed using a number representing the state of the operation. For example:

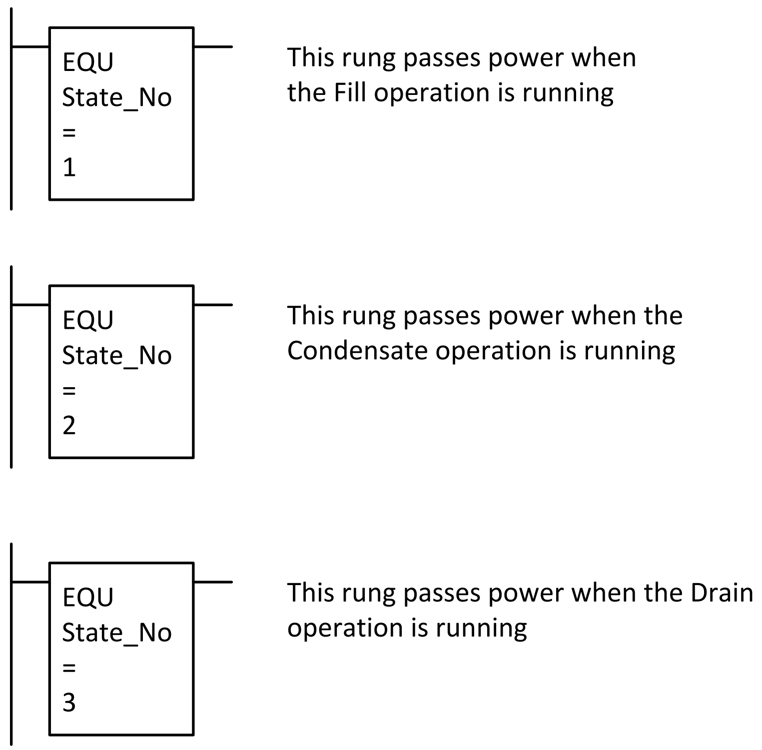


Fig. 8-64 Step Numbers for the Process

The three operations must be done in order. This requires that before the first operation starts, the requirement that there is not a fill, condensate or drain action presently active must be determined. This can be expressed in the start portion of the fill operation as:

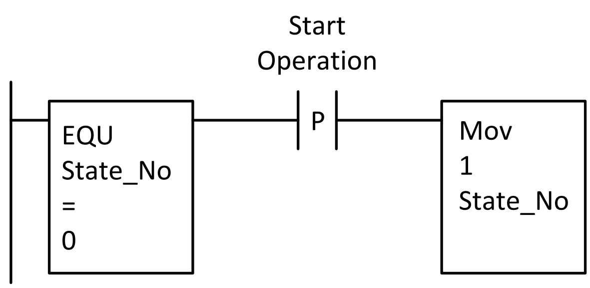


Fig. 8-65a Method of Moving Between States

Succeeding operations must likewise be programmed using a start portion with the prior operation present.

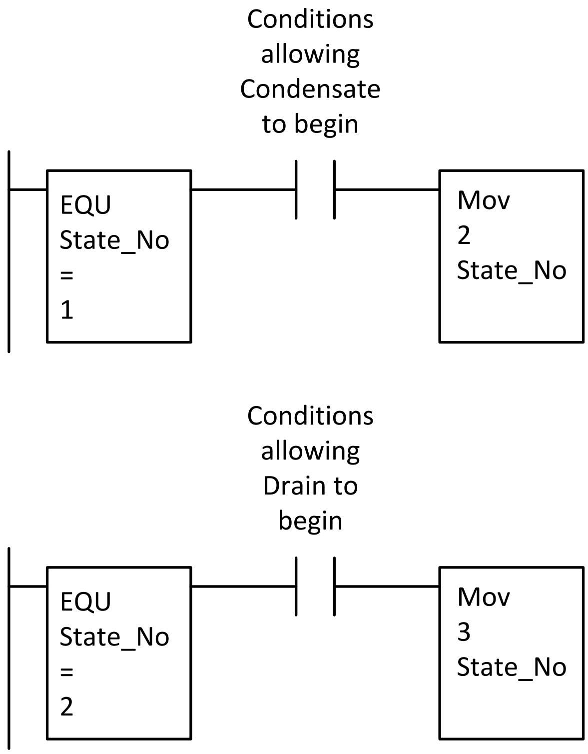


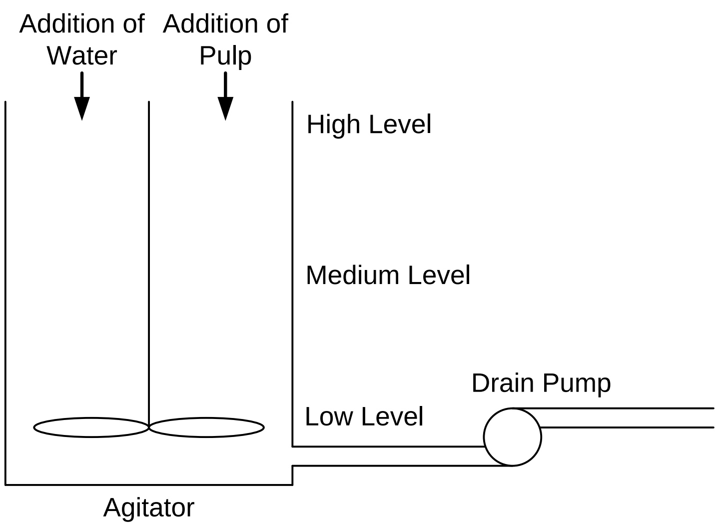
Fig. 8-65b Method of Moving Between States cont.

The conclusion of this problem is left as an exercise.

Paper Making Process

A process with logic, timers, compare and math is a mixing tank for making paper. The process works as follows:

1. A tank is filled to medium level with water.
2. The tank continues to fill with water and paper is added through a loss-in-weight feeder.
3. A stirrer starts as soon as the paper begins to be added and continues until a batch is made and dumped and the level falls to the low level.
4. The water finishes when the high level is reached and the paper finishes being added.
5. The water and paper continue to be mixed until the paper is thoroughly mixed into a slurry by use of a timer.
6. The tank is emptied by pumping the mixture out of the tank until the low level switch is reached. Then the process of making a new batch begins again.

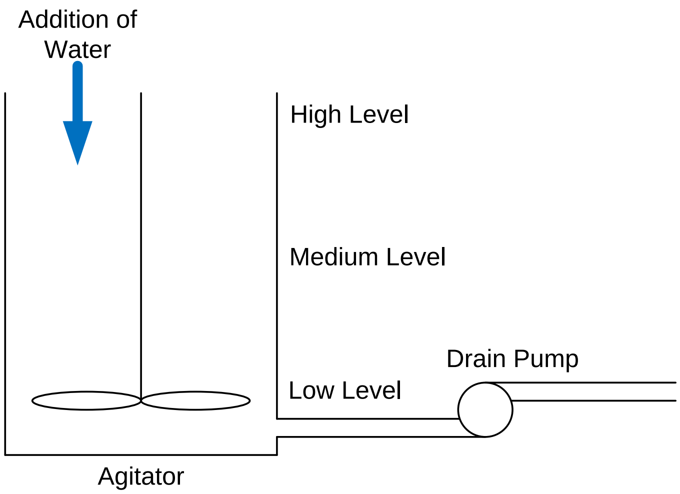


Water is added through a valve.

Paper is added through a scale with a numeric input and an output to run. The input is a number with the amount of weight in a hopper. When the output is on, the hopper on the scale will discharge paper. The scale weight will decrease. When the scale weight has decreased a set amount, the batch is done and the output is turned off. This is called a loss-in-weight feeder. The hopper is filled from above when the hopper is not being used. The hopper is filled by an operator manually.

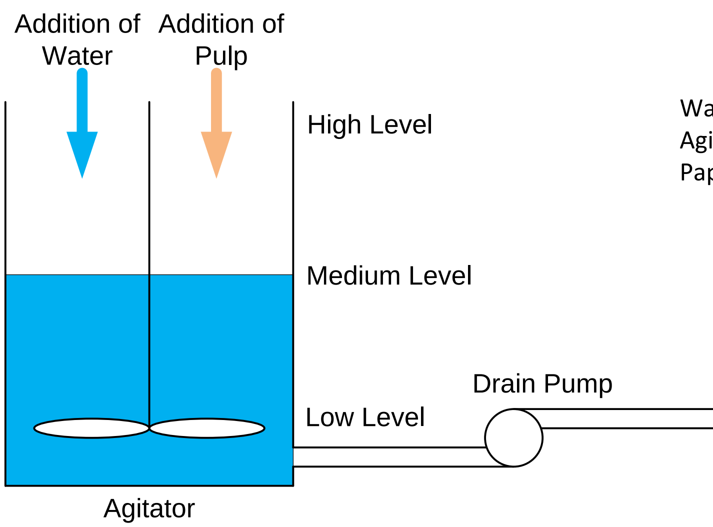
Fig. 8-66a Paper Making Process Described

The most difficult part of the program is the control of the loss-in-weight feeder. The number is read as an integer and may be at any value in a range. The program is to start at this value, turn on an output and watch the number representing the weight decrement a set amount and turn off the output.

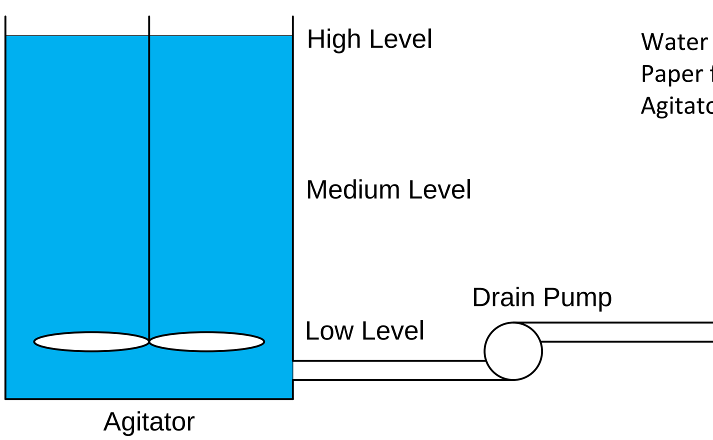


Starting a new batch:
Tank is empty
Water starts filling

Fig. 8-66b Paper Making Process Described



Water fills to Medium Level
Agitator starts
Paper starts filling



Water fills to High Level
Paper fills to weight set- point
Agitator continues to run for preset time

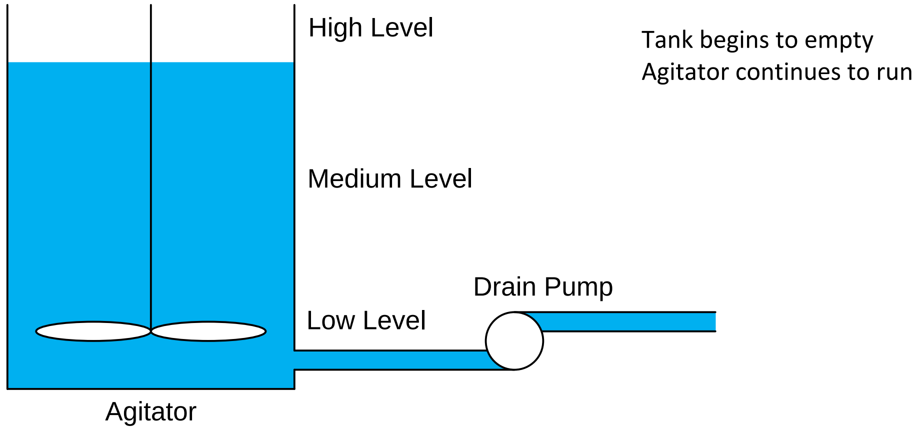


Fig. 8-66c Paper Making Process Described

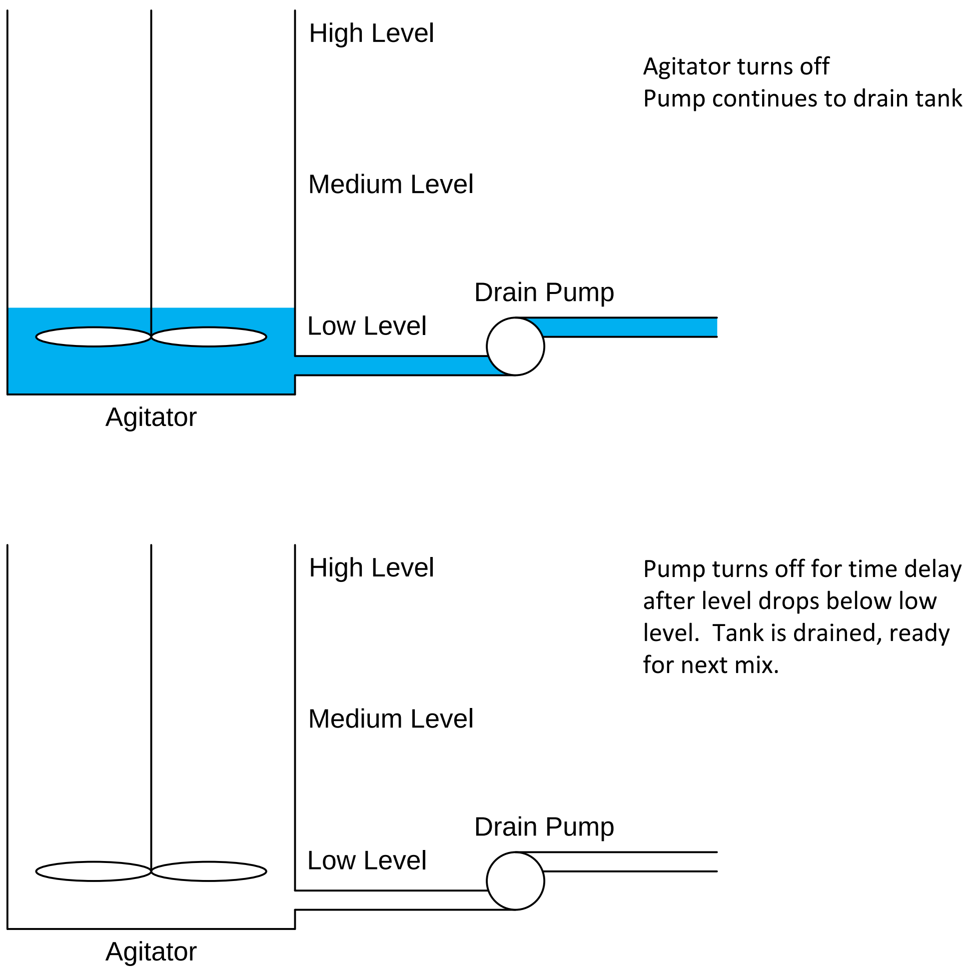


Fig. 8-66d Paper Making Process Described

To start the program for the paper-making process, begin by providing a start-up rung. The rung must provide a memory circuit (probably a seal circuit) to start the process. The program uses an action by an operator to start the mixing operation. It is always good to check all the level switches for proper status but the low level switch is the only switch that must be checked for proper level prior to starting. The low level switch must be off. The other two switches may be checked as well but are secondary to the low level switch for control of the batch. If either one of these two switches report a water level, the switch should be replaced or cleaned. It is not working properly or the low level switch is not working properly. Alert a maintenance person if this is a problem of any magnitude.

Where to begin the rest of the program is the responsibility of the programmer. Concentrate on one event at a time. Write down the requirements.

First, water must be added
Next, paper is added through a loss-in-weight scale
Next, other ingredients are added
Next, stirring occurs

When the paper pulp is to be added, start with a box of pulp and a vibratory feeder. Add the following:

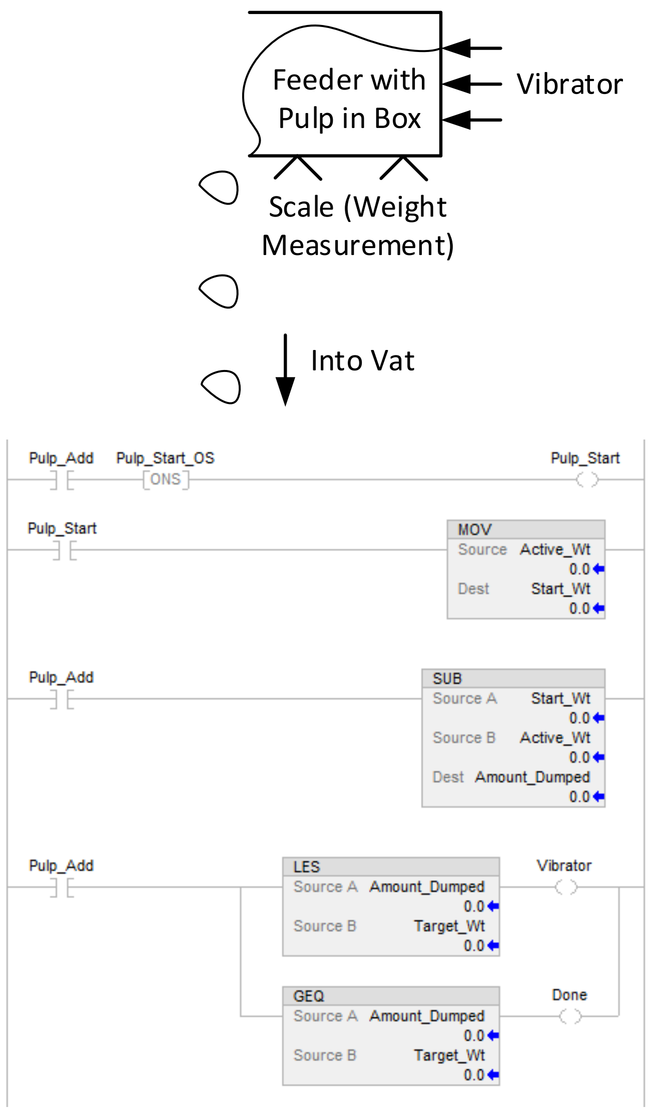


Fig. 8-67 Paper Making Process – Loss in Weight Feed

Stepping Program for Machine

The following machine is similar to the conveyor of previous chapters except we now are filling a box with material coming from a process line. The name of this device is a festooner. The machine moves back and forth placing the material in the box. The box is weighed and when a weight is achieved, the box is full. There is usually an automatic knife that cuts the material and starts the material in a second box. The logic is similar to the previous conveyors in that there is a motor that drives the product to the right and then to the left. End of travel photo-eyes reverse the movement. The end of process is achieved when a weight is exceeded. A start button begins the action. Provide a .5 sec dwell when the photo-eye sees the product to stop the movement before reversing the motor.

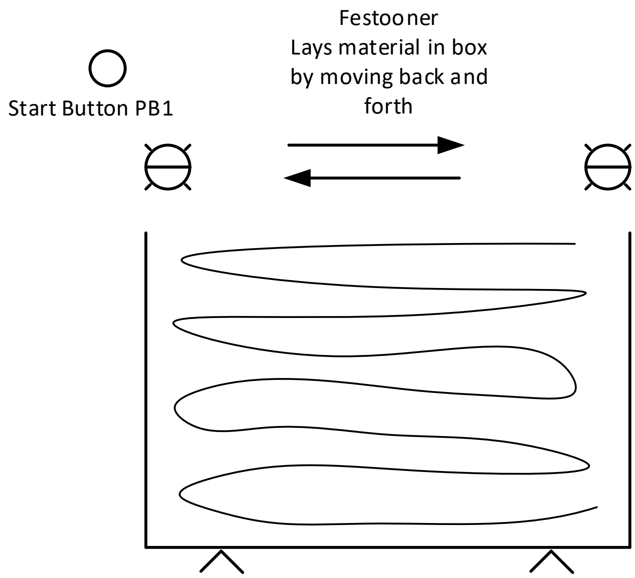
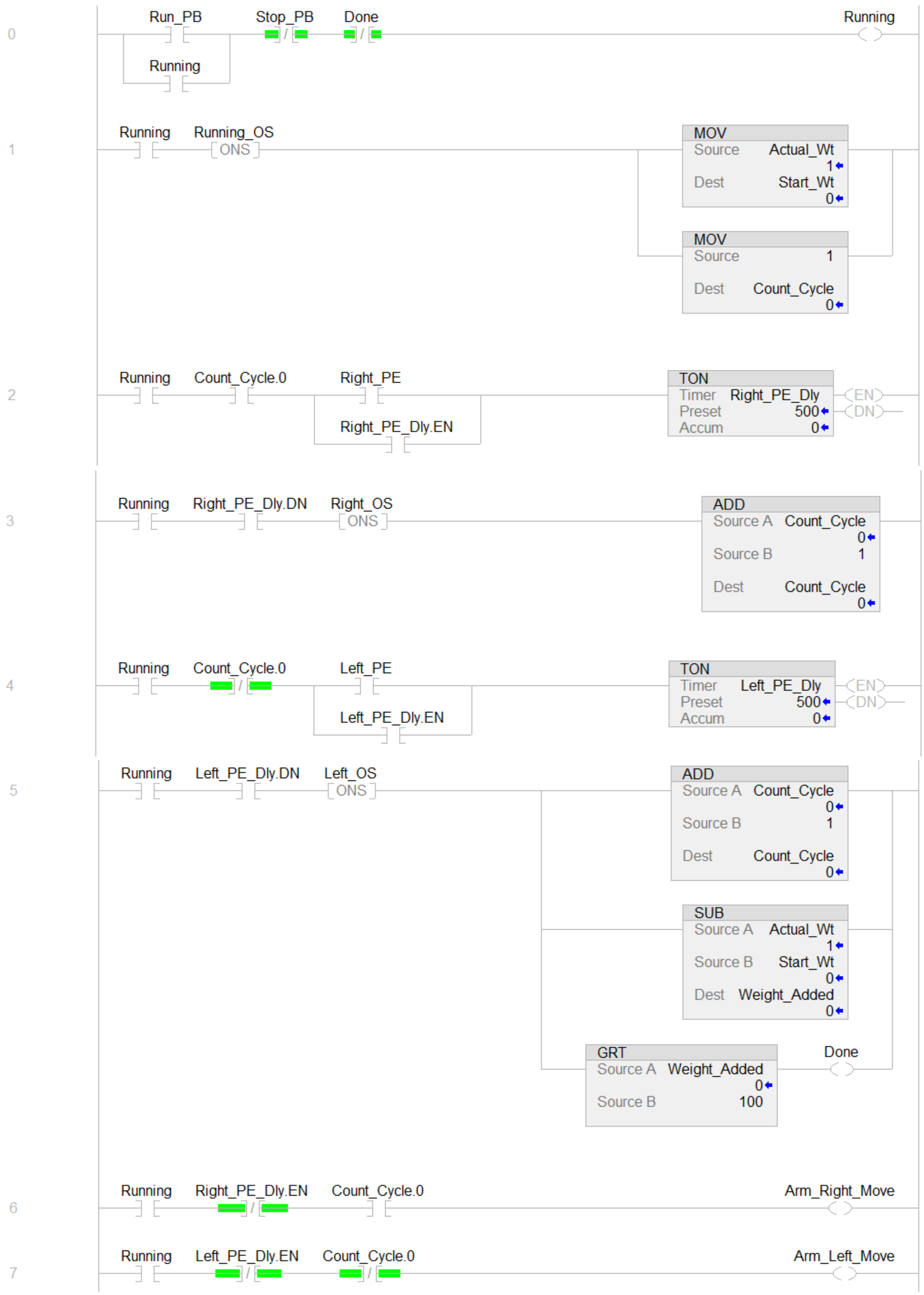


Fig. 8-68a Festooner Process

An actual festooner is pictured below with no box included. The material is placed on the table without enclosure. In the program on next page, the right-most bit of Count_Cycle is used to drive the machine right or left.

Fig. 8-68b

Festooner Process


0
Run_PB
Stop_PB
Done
Running
Running
1
Running
Running_OS
ONS
MOV
Source Actual_Wt
1
Dest Start_Wt
0
MOV
Source 1
Dest Count_Cycle
0
2
Running
Count_Cycle.0
Right_PE
Right_PE_Dly.EN
TON
Timer Right_PE_Dly
Preset 500
Accum 0
EN
DN
3
Running
Right_PE_Dly.DN
Right_OS
ONS
ADD
Source A Count_Cycle
0
Source B 1
Dest Count_Cycle
0
4
Running
Count_Cycle.0
Left_PE
Left_PE_Dly.EN
TON
Timer Left_PE_Dly
Preset 500
Accum 0
EN
DN
5
Running
Left_PE_Dly.DN
Left_OS
ONS
ADD
Source A Count_Cycle
0
Source B 1
Dest Count_Cycle
0
SUB
Source A Actual_Wt
1
Source B Start_Wt
0
Dest Weight_Added
0
GRT
Source A Weight_Added
0
Source B 100
Done
6
Running
Right_PE_Dly.EN
Count_Cycle.0
Arm_Right_Move
7
Running
Left_PE_Dly.EN
Count_Cycle.0
Arm_Left_Move

Number Systems

Decimal numbers

All number systems are positional in that each digit is weighted differently. For example, to write the number 287.54 represents 287.54 units of something. The 2 at the extreme left is the most important digit and is usually referred to as the most significant digit. The 4 on the right of the number is the least significant and is usually not as highly desired as the number on the left. For instance if 287.54 represented $287.54, we would be very interested if the number were $387.54 or even $187.54, especially if this was a bill we were to pay. On the other hand, we could hardly care if the 4 at the right of the number were a 5, 3, 8, 9 or whatever. The weights of the number represent the following:

$$2 \cdot 100 + 8 \cdot 10 + 7 \cdot 1 + 5 \cdot 0.1 + 4 \cdot 0.01$$

or:

$$2 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0 + 5 \cdot 8\text{-}^1 + 4 \cdot 8\text{-}^2$$

Binary numbers

Binary numbers are used in microprocessors, programmable logic controllers, and in all digital circuits. The binary number system only contains 2 digits: 0 and 1. Each digit is called a 'bit' and can contain either the value 0 or 1. The binary number system is like the decimal number system a positional number system and is written in the same general manner as decimal except that 10^x is replaced with 2^x.

$$d_n \cdot 2^{n-1} + \ldots + d_4 \cdot 2^3 + d_3 \cdot 2^2 + d_2 \cdot 2^1 + d_1 \cdot 2^0$$

Here d_n is the n^{th} digit (counting from right to left). If the number is an 8-bit number (called a byte), n is 8 (8 digits) and the binary number is 10101001, then it is calculated like this:

$$\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

$$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$1 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$$

= 169 (decimal)

or

$$10101001_2 = 169_{10}$$

Another example:

10011101

1 0 0 1 1 1 0 1

$1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$

$= 157_{10}$

or

$10011101_2 = 157_{10}$

The radix of binary numbers is 2.

Hexadecimal Numbers

Hexadecimal numbers have 16 different digits (radix 16). The 6 first letters of the alphabet are used for the last 6 digits in the hexadecimal system. The digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F representing the decimal numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 and 15 respectively. Upper case for A-F is optional.

The weights of numbers in hexadecimal is as follows:

$d_n \cdot 16^{n-1} + \ldots + d_4 \cdot 16^3 + d_3 \cdot 16^2 + d_2 \cdot 16^1 + d_1 \cdot 16^0$

d_n is the n^{th} digit (counting from right to left). This is the same as in any positional number system. If the number is a 4-digit number, n is 4 (4 digits) and the hexadecimal number is 5A2E or:

5 A 2 E

$5 \cdot 16^3 + 10 \cdot 16^2 + 2 \cdot 16^1 + 14 \cdot 16^0$

$5 \cdot 4096 + 10 \cdot 256 + 2 \cdot 16 + 14 \cdot 1$

= 23086 (decimal)

or

$5A2E_{16} = 23086_{10}$

Octal Numbers

The octal number system is used a little in plc texts and an explanation is as follows:

The octal number system contains 8 digits numbered from 0 to 7. To convert from octal to decimal, apply the following general formula:

$d_n \cdot 8^{n-1} + \ldots + d_4 \cdot 8^3 + d_3 \cdot 8^2 + d_2 \cdot 8^1 + d_1 \cdot 8^0$

d_n is the n^{th} digit (counting from right to left).

A conversion of the octal number 2417 to decimal follows:

2 4 1 7

$2 \cdot 8^3 + 4 \cdot 8^2 + 1 \cdot 8^1 + 7 \cdot 8^0$

$2 \cdot 512 + 4 \cdot 64 + 1 \cdot 8 + 7 \cdot 1$

= 1295 (decimal)

or

$2417_8 = 1295_{10}$

Converting between the binary and hexadecimal numbers:

To convert from binary to hexadecimal, simply line the bits in 4 bit groups as follows:

1001011101011010_2

Binary	1001	0111	0101	1010
Decimal	9	7	5	10
Hexadecimal	9	7	5	A

or $1001011101011010_2 = 975A_{16}$

Hexadecimal numbers are usually easier to remember because they are shorter than binary numbers.

Another example:

$E551A0F8_{16}$

Hexadecimal	E	5	5	1	A	0	F	8
Decimal	14	5	5	1	10	0	15	8
Binary	1110	0101	0101	0001	1010	0000	1111	1000

$= 11100101010100011010000011111000_2$

The table shown below summarizes the digits.

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Binary Addition

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 carry 1
1 + 0 + carry = 0 carry 1
1 + 1 + carry = 1 carry 1

These are the rules for binary addition.

To see binary addition at work:

Carry				1	1	1	1				
Number 1	0	1	0	0	1	1	0	1	1	0	0
+ Number 2	0	1	0	1	1	0	1	1	0	1	0
Results	1	0	1	0	1	0	0	0	1	1	0

Binary addition may take place in ladder logic. Instructions are provided to carry out this function (ADD), but it is worthwhile to examine the process of binary addition using ladder logic. In Figs. 8-35 and 8-36, logic to add two numbers using only combinational logic is shown.

Since Bit 0 does not have a *carry_in*, half-adder logic may be employed but only for this bit. It can be seen that half-adder logic is simpler than full-add logic by comparing Fig. 8-35 (Half-Adder) to Fig. 8-36 (Full Adder).

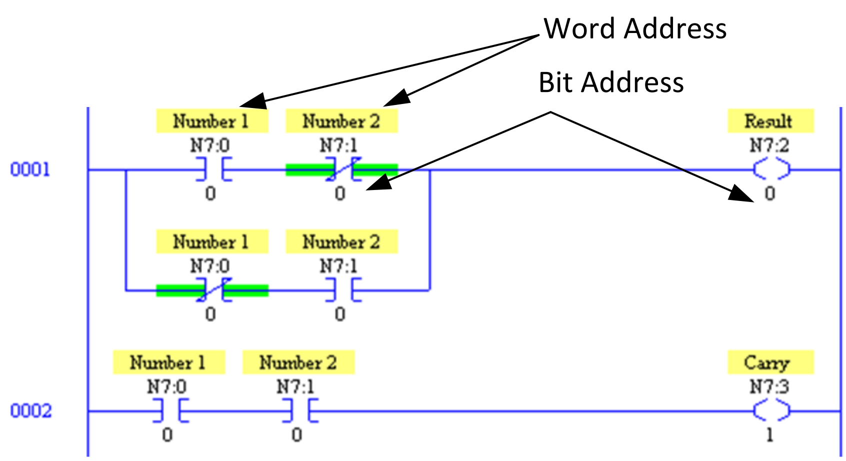


Fig. 8-69
Half Adder
Logic

The number in location N7:0 is added to the number in N7:1. The result is stored in location N7:2. The carry is located in N7:3. The same locations are used for remaining bits of the word shown in Fig. 8-36. Full adder logic for each remaining bit from 1 to 15 is required. The logic must be duplicated for each bit. Carry_In is from the prior bit. The Carry_In for bit 1 is found in

Carry_Out of bit 0.

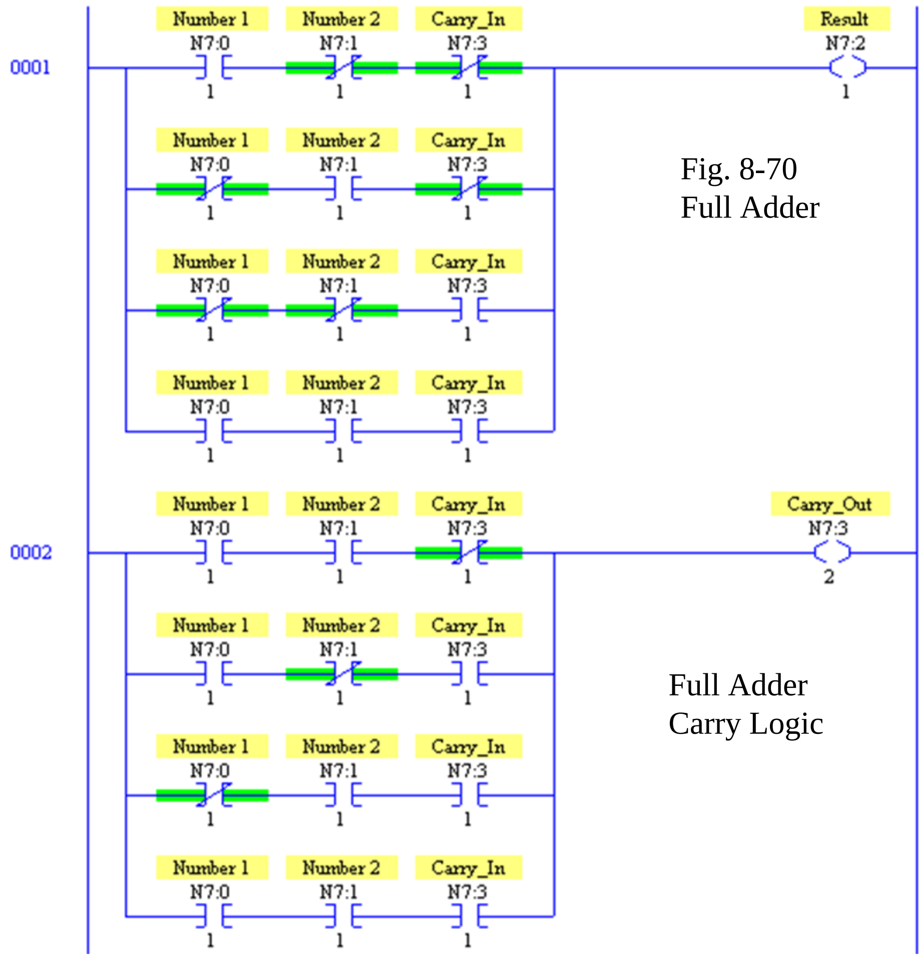


Fig. 8-70
Full Adder

Full Adder
Carry Logic

It is worth noting that to actually build this logic requires a great deal of time unless the Copy-Paste function is employed. Once the logic is built, changing the bit numbers in the logic is all that is required for succeeding bits from 2 to 15.

Binary Subtraction

To perform binary subtraction, the easiest method is to find the 2's complement of the second number and then add the two numbers together. To find the 2's complement, invert all the bits (1's complement and add 1).

To find the 2's complement:

number	0	1	0	0	1	1	0	1	0	1	1
1's complement	1	0	1	1	0	0	1	0	1	0	0
+1											1
2's complement	1	0	1	1	0	0	1	0	1	0	1

Then add the 2's complement to the first number.

A second method of finding the 2's complement requires the use of a memory bit. The rule requires that bits from the original number be copied to the 2's complement number starting at the right-most bit. The rule applies until a "1" is encountered. The first "1" is copied but a memory bit is set after which the bits are "flipped". Try this rule. It works and may be employed using ladder logic and a Latch bit to quickly find the 2's complement of a number.

Again, logic must be added to complete the function using rungs similar to rungs 4 and 5 of this figure but using bits 2 through15.

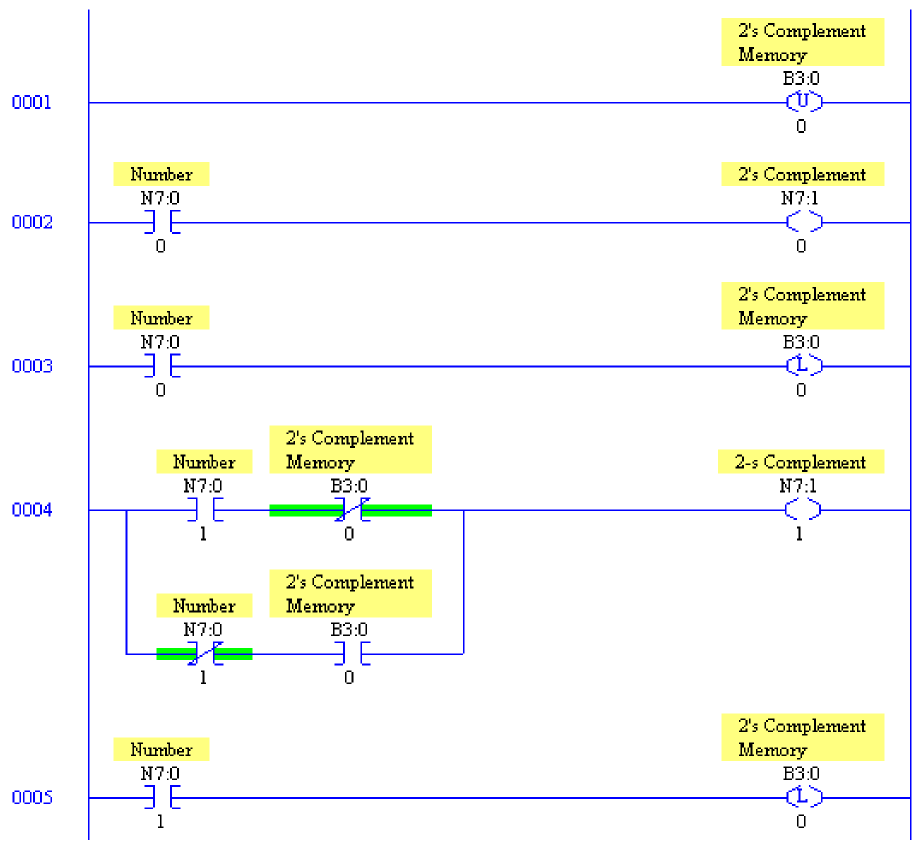


Fig. 8-71
Rung 1,2, 3
Bit 0 Logic

Rung 4, 5
Bit 1 Logic which must be repeated for Bits 2-15

Binary Comparisons

To find if two numbers are equal, use the Equal Block. Use of combinational ladder logic may be employed as well. Using combination logic may be easy to employ but should not be used in PLC programming instead of the Equal Block. See below an example of the use of combinational ladder logic to determine if the number in N7:0 is equal to the number in C5:3.ACC. Notice that only the high bite or bits 8-15 are being checked for equality. If all bits are to be checked, the rung must double in size to check for the 8 additional bits (bits 0-7).

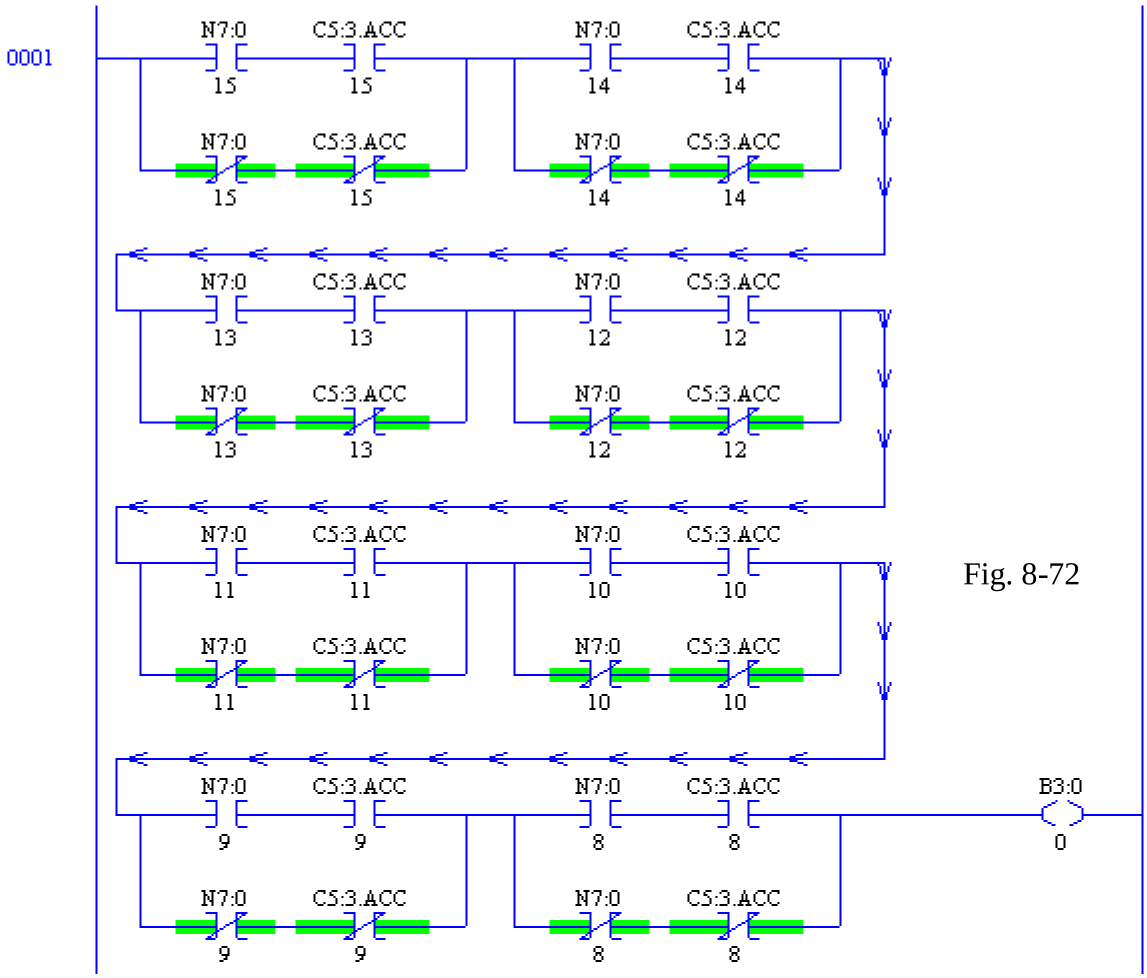


Fig. 8-72

Radix

The term "radix" is used to describe the number base used to display numbers. For B3, notice the natural radix is binary although several other bases may be used.

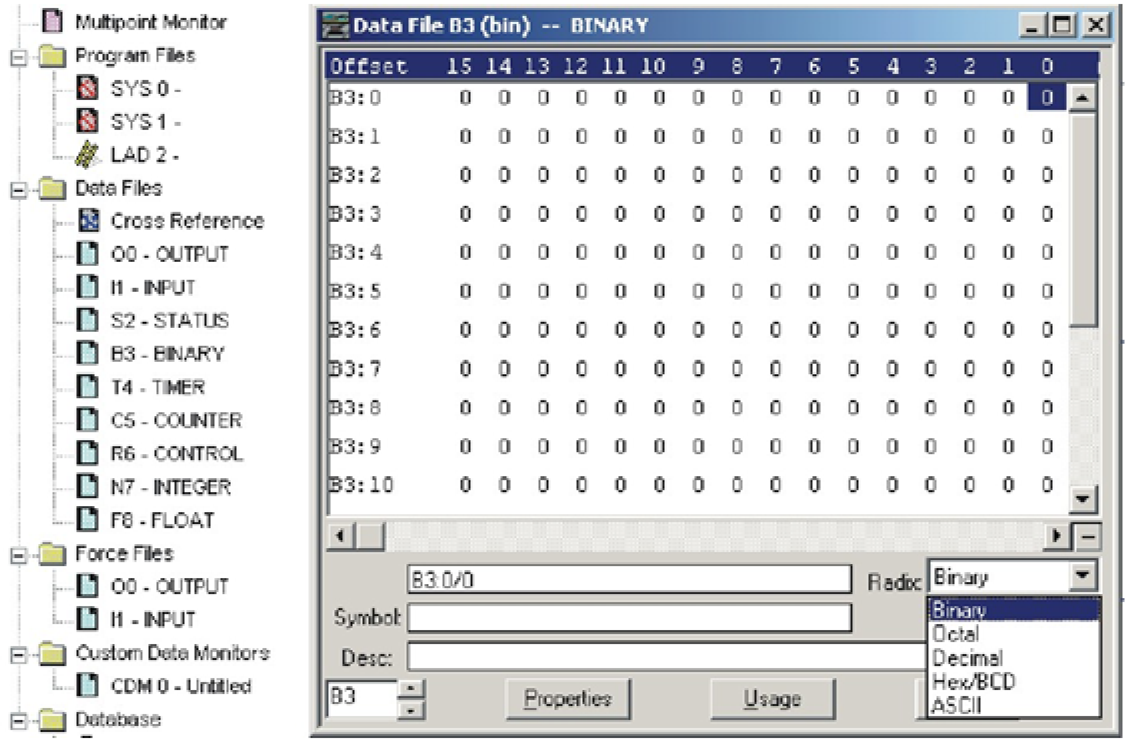


Fig. 8-73   Use Radix to Display Binary Layout

Likewise, for integer numbers, decimal is the natural base used although any of the same group as binary can be chosen.

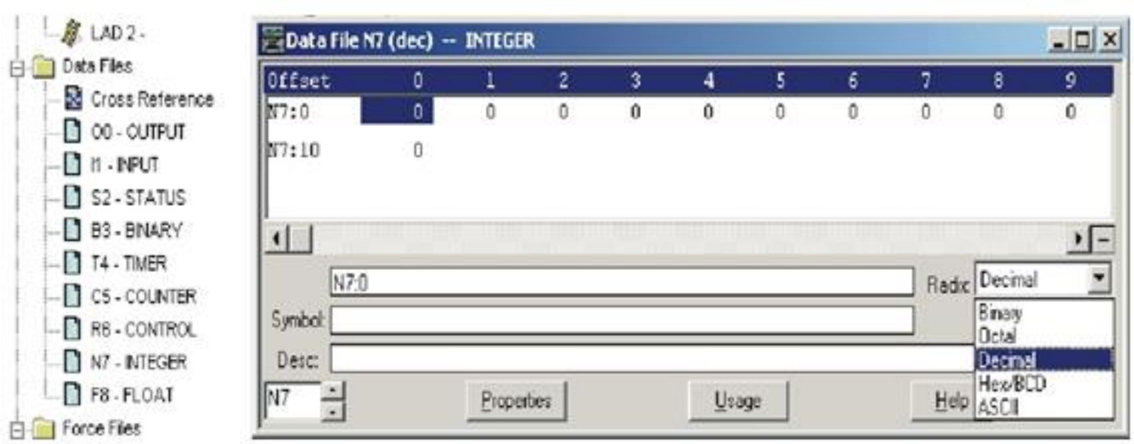


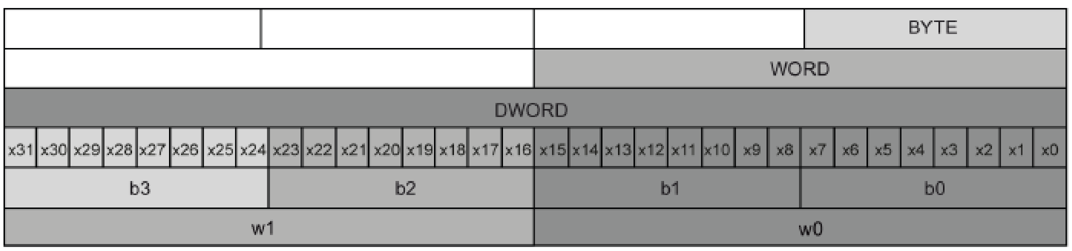
Fig. 8-74   Use Radix to Display Integer Layout

Accessing a "slice" of a tagged data type

PLC tags and data block tags can be accessed at the bit, byte, or word level depending on their size. The syntax for accessing such a data slice (Siemens) is as follows:

- "<PLC tag name>".xn (bit access)
- "<PLC tag name>".bn (byte access)
- "<PLC tag name>".wn (word access)
- "<Data block name>".<tag name>.xn (bit access)
- "<Data block name>".<tag name>.bn (byte access)
- "<Data block name>".<tag name>.wn (word access)

A double word-sized tag can be accessed by bits 0 - 31, bytes 0 - 3, or word 0 - 1. A word sized tag can be accessed by bits 0 - 15, bytes 0 - 2, or word 0. A byte-sized tag can be accessed by bits 0 - 8, or byte 0. Bit, byte, and word slices can be used anywhere that bits, bytes, or words are expected operands.

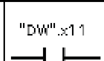
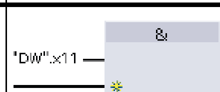
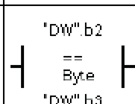
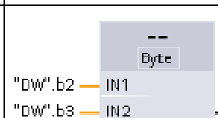
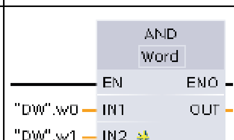
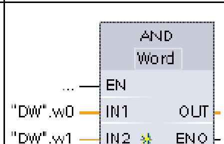


Note

Valid data types that can be accessed by slice are Byte, Char, Conn_Any, Date, DInt, DWord, Event_Any, Event_Att, Hw_Any, Hw_Device, HW_Interface, Hw_Io, Hw_Pwm, Hw_SubModule, Int, OB_Any, OB_Att, OB_Cyclic, OB_Delay, OB_WHINT, OB_PCYCLE, OB_STARTUP, OB_TIMEERROR, OB_Tod, Port, Rtm, SInt, Time, Time_Of_Day, UDInt, UInt, USInt, and Word. PLC Tags of type Real can be accessed by slice, but data block tags of type Real cannot.

Examples

In the PLC tag table, "DW" is a declared tag of type DWORD. The examples show bit, byte, and word slice access:

	LAD	FBD	SCL
Bit access			`IF "DW".x11 THEN ... END_IF;`
Byte access			`IF "DW".b2 = "DW".b3 THEN ... END_IF;`
Word access			`out:= "DW".w0 AND "DW".w1;`

Accessing a tag with an AT overlay (Also Siemens)

The AT tag overlay allows you to access an already-declared tag of a standard access block with an overlaid declaration of a different data type. You can, for example, address the individual bits of a tag of a Byte, Word, or DWord data type with an Array of Bool. To overlay a parameter, declare an additional parameter directly after the parameter that is to be overlaid and select the data type "AT". The editor creates the overlay, and you can then choose the data type, struct, or array that you wish to use for the overlay.

Example

This example shows the input parameters of a standard-access FB. The byte tag B1 is overlaid with an array of Booleans:

B1		Byte
AT	AT "B1"	Array [0..7] of Bool
AT[0]		Bool
AT[1]		Bool
AT[2]		Bool
AT[3]		Bool
AT[4]		Bool
AT[5]		Bool
AT[6]		Bool
AT[7]		Bool

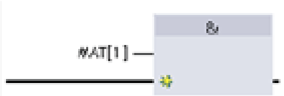
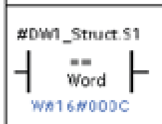
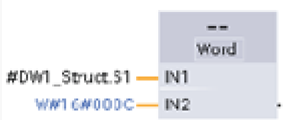
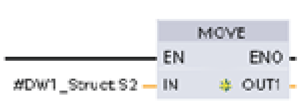
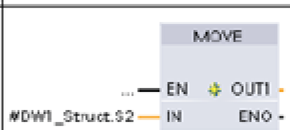
Table 4- 6   Overlay of a byte with a Boolean array

7	6	5	4	3	2	1	0
AT[0]	AT[1]	AT[2]	AT[3]	AT[4]	AT[5]	AT[6]	AT[7]

Another example is a DWord tag overlaid with a Struct:

DW1		DWord
DW1_Struct	AT "DW1"	Struct
S1		Word
S2		Byte
S3		Byte

The overlay types can be addressed directly in the program logic:

LAD	FBD	SCL
		IF #AT[1] THEN ... END_IF;
		IF (#DW1_Struct.S1 = W#16#000C) THEN ... END_IF;
		out1 := #DW1_Struct.S2;

Allen-Bradley Data Slice

Below is an example of an expansion of a 32-bit word test_tag1 with the tag displayed as a DINT and the bits addressed sequentially starting with bit 0 through bit 31.

Scope: MainProgram Show: All Tags

Name	Data Type	Description	External Access	Constant	Style
test_tag1	DINT		Read/Write	☐	Decimal
test_tag1.0	BOOL		Read/Write		Decimal
test_tag1.1	BOOL		Read/Write		Decimal
test_tag1.2	BOOL		Read/Write		Decimal
test_tag1.3	BOOL		Read/Write		Decimal
test_tag1.4	BOOL		Read/Write		Decimal
test_tag1.5	BOOL		Read/Write		Decimal
test_tag1.6	BOOL		Read/Write		Decimal
test_tag1.7	BOOL		Read/Write		Decimal
test_tag1.8	BOOL		Read/Write		Decimal
test_tag1.9	BOOL		Read/Write		Decimal
test_tag1.10	BOOL		Read/Write		Decimal
test_tag1.11	BOOL		Read/Write		Decimal
test_tag1.12	BOOL		Read/Write		Decimal
test_tag1.13	BOOL		Read/Write		Decimal
test_tag1.14	BOOL		Read/Write		Decimal
test_tag1.15	BOOL		Read/Write		Decimal
test_tag1.16	BOOL		Read/Write		Decimal
test_tag1.17	BOOL		Read/Write		Decimal
test_tag1.18	BOOL		Read/Write		Decimal
test_tag1.19	BOOL		Read/Write		Decimal
test_tag1.20	BOOL		Read/Write		Decimal
test_tag1.21	BOOL		Read/Write		Decimal
test_tag1.22	BOOL		Read/Write		Decimal
test_tag1.23	BOOL		Read/Write		Decimal
test_tag1.24	BOOL		Read/Write		Decimal
test_tag1.25	BOOL		Read/Write		Decimal
test_tag1.26	BOOL		Read/Write		Decimal
test_tag1.27	BOOL		Read/Write		Decimal
test_tag1.28	BOOL		Read/Write		Decimal
test_tag1.29	BOOL		Read/Write		Decimal
test_tag1.30	BOOL		Read/Write		Decimal
test_tag1.31	BOOL		Read/Write		Decimal

Summary of Addressing Individual Bits

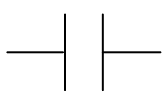
For a 16 bit integer, we have:

Bit	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Siemens M	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0.0	1.7	1.6	1.5	1.4	1.3	1.2	1.1	1.0
Siemens Addr	.x15	.x14	.x13	.x12	.x11	.x10	.x9	.x8	.x7	.x6	.x5	.x4	.x3	.x2	.x1	.x0
Logix	.15	.14	.13	.12	.11	.10	.9	.8	.7	.6	.5	.4	.3	.2	.1	.0
SLC	/15	/14	/13	/12	/11	/10	/9	/8	/7	/6	/5	/4	/3	/2	/1	/0

Data Slice Last Look

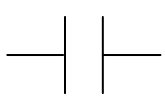
With the above table, we can assign contacts or coils referencing specific bits in the word test_tag1. These examples show addressing for the most significant or sign bit.

test_tag1.x15



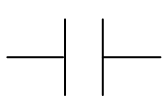
Siemens Addr

test_tag1.15



A-B Logix

N7:0/15



A-B SLC

Summary

In summary, the chapter discusses manipulation of numeric data in the PLC. Math Function Blocks were described first. Math Function Blocks replace the relay coil at the right of a rung. Execution of the function block happens with either regular or one-shot contacts. The Math Function Block may also be programmed to execute continuously.

Compare Instructions allow power to pass in a manner similar to relay contacts. A Compare Instruction, *LIM*, was discussed and found to control a contact over a range of numbers. Comparison Instructions with memory were used to provide a dead-band for switching on or off an output. Various memory circuits were demonstrated in programs to provide this control algorithm.

The Status Table in coordination with math operations was discussed. Several Status Table locations described the use of math operations and locations of math holding registers. Thirty-two bit integer math was briefly described. The status table is also used to control the 32-bit math operation. Floating point math as well as floating-point_integer math instructions were also discussed.

The *SCP* or Scale with Parameters Instruction was summarized. A method of protecting against numeric overflow was described. *ADD* and *SUB* blocks to add or subtract 1% or 5% from a number were also programmed.

Use of *Instruction Help* with the example of the *CPT* block was also given. Logic was developed showing the mixing of logic for counters and math.

A paper-making process was described with many implications for the use of math in PLC logic.

Various number systems were described including decimal, hexadecimal and octal. The Radix box in RSLogix 500 was described.

Labs and exercises follow.

Exercises

1. Write a rung to turn on a coil when the value in int_1 > 20 and int_2 < 25.

2. Write a rung to turn on a coil when the value in int_1 < 20 or float_1 > 8.

3. Write a rung to turn on a coil when the value in byte_3 < 120.

4. Multiply 10 * int_1 and display results in an integer variable.

5. Divide the contents of int_4/(constant = 4).

6. Write logic to create six time intervals as in the traffic light problem but using only one timer: (Only provide enough code to illustrate the concept.)

7. Two numbers are stored in float_1 and float_2. If these numbers represent the sides of a triangle, find the length of the hypotenuse.

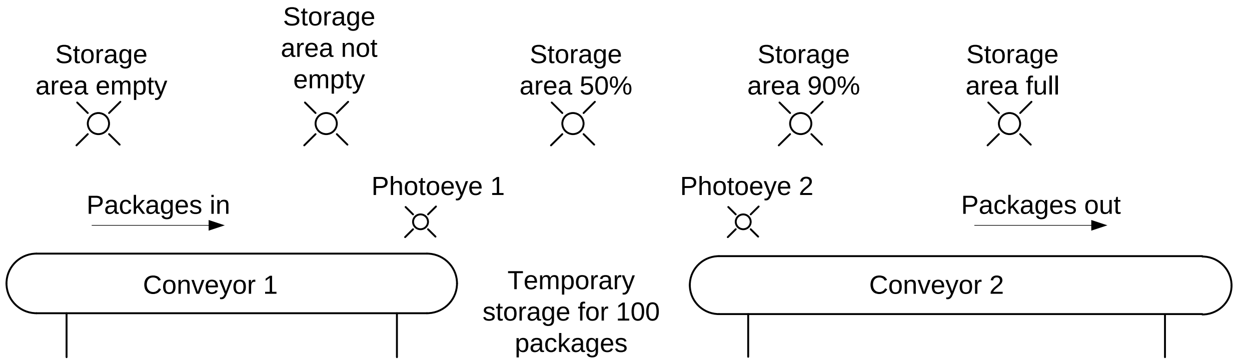
8. Write a single rung to turn on an output when the value in int_1 is less than 200, greater than 1000 or equal to 555:

9. The accumulated number of widgets on the production line is found in counter widget_count_accum. Each widget is worth 35 cents. Write ladder logic to show in total_cost the total worth of widgets on the line. What are the numerical limitations of your calculation?

10. Write a rung of logic that turns on when the number in int_1= 22. Use only relay contacts and one output coil.

11. Three numbers are stored in int_1, int_2 and int_3. Find the average of these three numbers and store the results.

12. The following conveyor system has five outputs, lights for percent complete of packages going down conveyor 1 to conveyor 2. Write a program to turn on these lights based on the fact that packages must pass photoeye 1 to enter the storage area and pass photoeye 2 to exit.



13. Use Fig. 8-25 and describe how to program high and low limit tests with Siemens and CompactLogix Ladder Logic.

14. Use Fig. 8-27 and describe how to program a dead band application with Siemens and CompactLogix Ladder Logic.

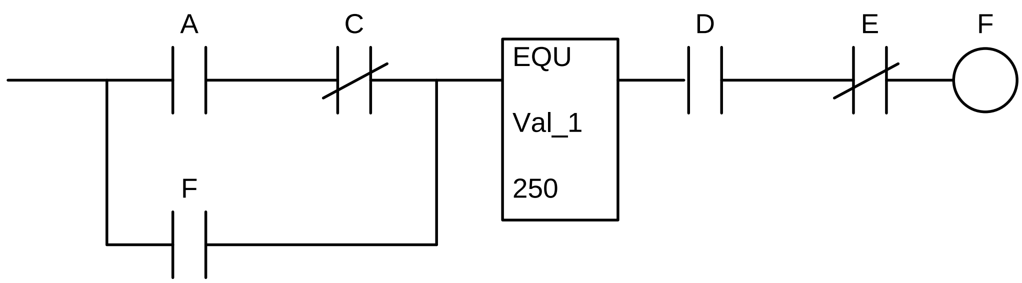
15. Find the equivalent instruction to the SCP instruction for the Siemens and CompactLogix processors.

16. Describe a program statement to add 1% to the full scale value of a variable using the Siemens and CompactLogix processors.

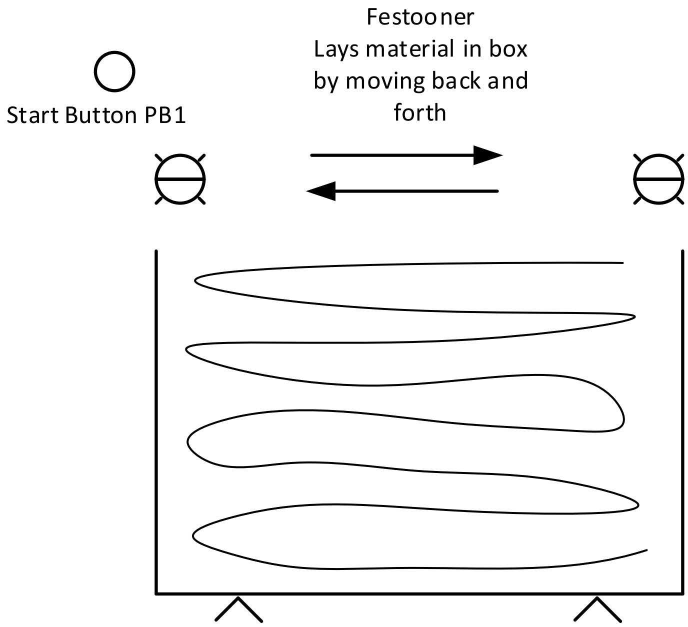
17. Write the loss-in-weight portion of the paper-making process in which weight is lost from a scale that feeds pulp to the batch. The scale is represented by a number that must decrease a set amount from an arbitrary number downward a set amount. During the feed cycle, an output is on running a loss-in-weight feeder, usually a vibratory feed device.

18. Finish the Juice Condensate program using numbers for states.

19. Convert the following seal circuit to a S/R circuit.



20. Write logic to place the material in the festooner box:



21. The festooner pictured above now lays material in a box by moving an arm back and forth. The action reverses after a small time delay after hitting the end-of-travel photo-eye. The box is considered complete when the weight exceeds 100 pounds or 50 back-and-forth actions – whichever occurs first. Design an I/O table and write a program in Ladder Logic to perform the action.

22. The festooner pictured above is changed to lay material into the box with an arm that moves left and right with the following conditions:

    The sensors at each end stop the movement and, after a short time delay, reverse the motion. The scale under the box records the weight at the beginning of the fill. The arm stops close to when 100# is met. The arm is to come to rest at the left sensor. If the weight isn't yet at 100# but closer to 100# than if the arm were to move through another pass, the arm is to stop and the operation is over. The goal is to place as close to 100# material as possible with the fill ending at the left sensor.

23. Write a program that turns on a heater when the temperature falls less than 300 ^{o}F and turns off the heater when the temperature exceeds 325 ^{o}F for the liquid in a vessel. The temperature is input in a variable labelled 'temp'. In a table, define each variable used by type.

24. Design a ramp up block for the lab below using the servo motor (Ramp to setpoint).

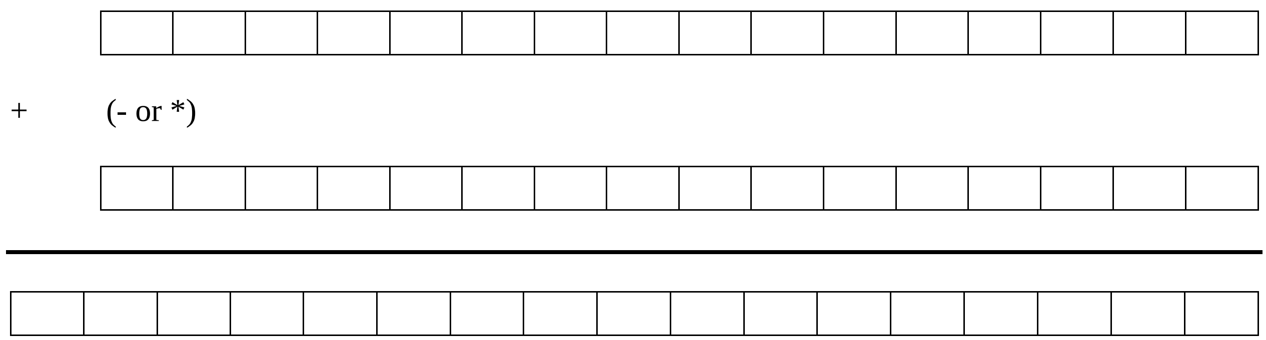
Lab 8.1 Integer Math

Lab 8.1a Using only contacts and coils, add two integer numbers found in two integer numbers. Counts may be 8 bit, 16 bit or 32 bit in length.

Lab 8.1b Using only contacts and coils, subtract one integer number from another integer.

Lab 8.1c Using only contacts and coils, multiply two integer numbers.

Use the following as a guide to labs 8.1:



Lab 8.1d Using only contacts and coils, create an equivalent of an up counter. Use inputs for counting up and count reset. Turn on an output when the count equals a preset. For memory, use latch or coil outputs. Make the counter 8 bits long or from 0 to 255. Use a constant in an integer location for the compare (between 0 and 255).

Lab 8.1e Using only contacts and coils, create an equivalent of a down counter. Use inputs for counting down and count reset. Turn on an output when the count equals a preset. For memory, use latch or coil outputs. Make the counter 8 bits long or from 0 to 255. Use a constant in an integer location for the compare (between 0 and 255).

Again, quoting from the Siemens Easy Book: "

6.5.1 Watch tables and force tables

You use "watch tables" for monitoring and modifying the values of a user program being executed by the online CPU. You can create and save different watch tables in your project to support a variety of test environments. This allows you to reproduce tests during commissioning or for service and maintenance purposes.

With a watch table, you can monitor and interact with the CPU as it executes the user program. You can display or change values not only for the tags of the code blocks and data blocks, but also for the memory areas of the CPU, including the inputs and outputs (I and Q), peripheral inputs (I:P), bit memory (M), and data blocks (DB).

With the watch table, you can enable the physical outputs (Q:P) of a CPU in STOP mode. For example, you can assign specific values to the outputs when testing the wiring for the CPU.

STEP 7 also provides a force table for "forcing" a tag to a specific value. For more information about forcing, see the section on forcing values in the CPU (Page 340) in the "Online and Diagnostics" chapter.

Note

The force values are stored in the CPU and not in the watch table.

You cannot force an input (or "I" address). However, you can force a peripheral input. To force a peripheral input, append a ":P" to the address (for example: "On:P").

6.5.2 Cross reference to show usage

The Inspector window displays cross-reference information about how a selected object is used throughout the complete project, such as the user program, the CPU and any HMI devices. The "Cross-reference" tab displays the instances where a selected object is being used and the other objects using it. The Inspector window also includes blocks which are only available online in the cross-references. To display the cross-references, select the "Show cross-references" command. (In the Project view, find the cross references in the "Tools" menu.)

Note

You do not have to close the editor to see the cross-reference information.

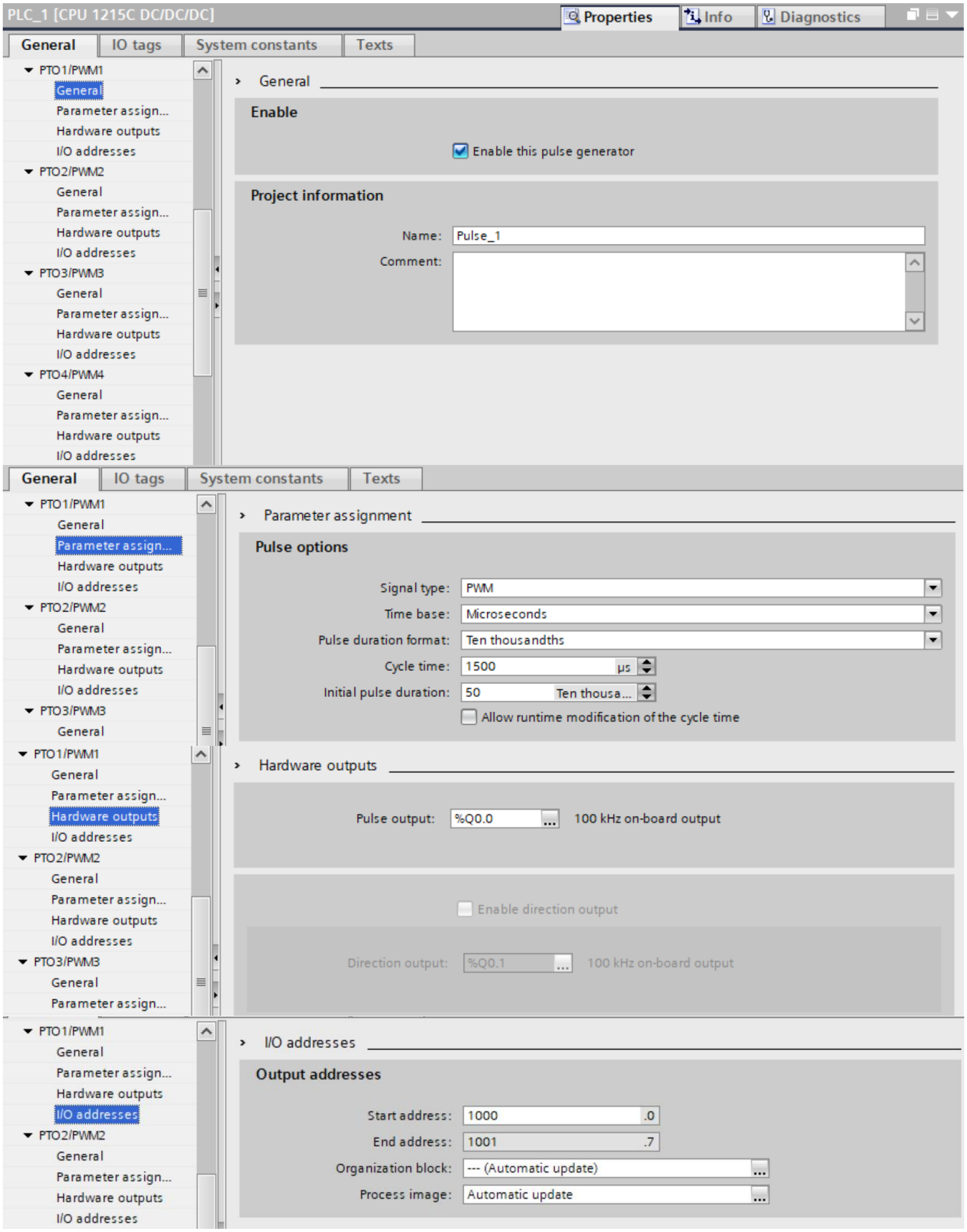
You can sort the entries in the cross-reference. The cross-reference list provides an overview of the use of memory addresses and tags within the user program.

- When creating and changing a program, you retain an overview of the operands, tags and block calls you have used.
- From the cross-references, you can jump directly to the point of use of operands and tags.
- During a program test or when troubleshooting, you are notified about which memory location is being processed by which command in which block, which tag is being used in which screen, and which block is called by which other block.

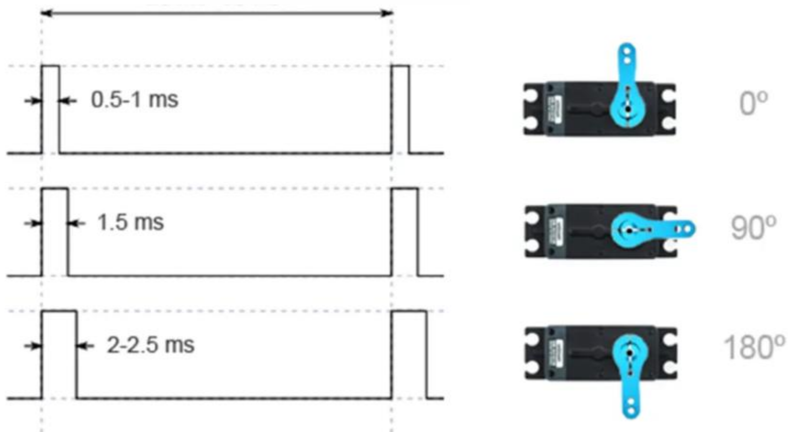
"

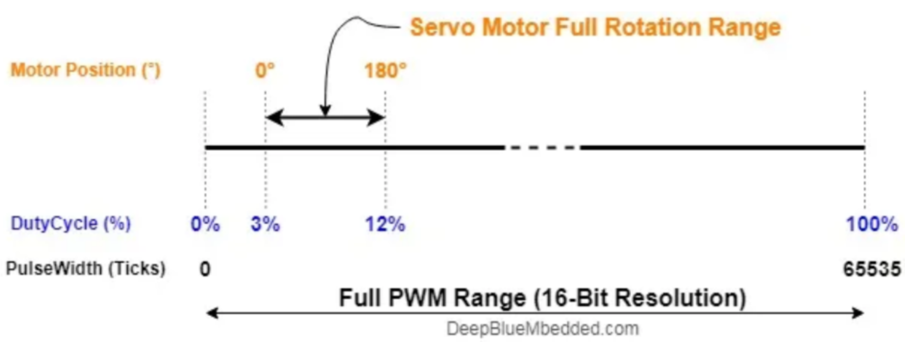
Lab 8.2 PWM and RAMP

In this lab we are introduced to the PWM output of the Siemens 1200 PLC. Described below is the configuration of the PWM output channels of the first 4 outputs of the Siemens S7-1215 DCDCDC processor. The configuration shown is just part of the process to program a pwm output.



The picture below is a hobby servo controller. The movement and control is based on a number in the output word associated with the output. With a configuration as above, the pulse width is determined by the servo specification. The specification below shows 1500 micro seconds or 1.5 msec duration. Both servo applications below use the HS-422 servo motor.





ANNOUNCED SPECIFICATION OF HS–422 STANDARD DELUXE SERVO

1.TECHNICAL VALUES

CONTROL SYSTEM	: +PULSE WIDTH CONTROL 1500usec NEUTRAL	
OPERATING VOLTAGE RANGE	: 4.8V TO 6.0V	
OPERATING TEMPERATURE RANGE	: -20 T0 +60°C	
TEST VOLTAGE	: AT 4.8V	AT 6.0V
OPERATING SPEED	: 0.21sec/60° AT NO LOAD	0.16sec/60° AT NO LOAD
STALL TORQUE	: 3.3kg.cm(45.82oz.in)	4.1kg.cm(56.93oz.in)
OPERATING ANGLE	: 45°/ONE SIDE PULSE TRAVELING 400usec	
DIRECTION	: CLOCK WISE/PULSE TRAVELING 1500 TO 1900usec	
CURRENT DRAIN	: 8mA/IDLE AND 150mA/NO LOAD RUNNING	
DEAD BAND WIDTH	: 8usec	
CONNECTOR WIRE LENGTH	: 300mm(11.81in)	
DIMENSIONS	: 40.6x19.8x36.6mm(1.59x0.77x1.44in)	
WEIGHT	: 45.5g(1.6oz)	

The instruction for driving the PWM output is the CTRL_PWM instruction. This instruction is explained in the Easy Manual and copied below: "

Programming made easy

6.3 Powerful instructions make programming easy

6.3.7 Pulse-width modulation (PWM)

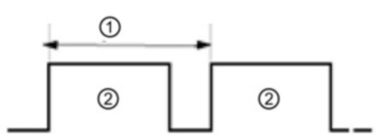
The CTRL_PWM instruction is available in the Pulse group of the Extended instructions.

Table 6- 20 CTRL_PWM instruction

LAD / FBD	SCL	Desciption

	`"ctrl_pwm_db"(`
	`PWM:=W#16#0,`
	`ENABLE:=False,`
	`BUSY=>_bool_out_,`
	`STATUS=>_word_out_);`
		The CTRL_PWM instruction provides a fixed cycle time output with a variable duty cycle. The PWM output runs continuously after being started at the specified frequency (cycle time). The pulse width is varied as required to affect the desired control.

When you insert the CTRL_PWM instruction in your code block, you create the DB for the instruction from the "Call options" dialog. The CTRL_PWM instruction stores the parameter information in the DB and controls the data block parameters.



Duty cycle can be expressed, for example, as a percentage of the cycle time or as a relative quantity (such as 0 to 1000 or 0 to 10000). The pulse width can vary from 0 (no pulse, always off) to full scale (no pulse, always on).

① Cycle time

② Pulse width time

The PWM output can be varied from 0 to full scale, providing a digital output that in many ways is the same as an analog output. For example, the PWM output can be used to control the speed of a motor from stop to full speed, or it can be used to control position of a valve from closed to fully opened.

"

Lab 8.2A

This lab requires the number in QW1000 to be modulated between two numbers to engage the vacuum and disengage the vacuum. The numbers and command to turn the vacuum on and off must be determined by trial and error.

“

Home > Lynxmotion > Robotic Arms > Arm Accessories > **Lynxmotion Vacuum Gripper Kit**

Share

Click to enlarge

Be the first to review this product

Lynxmotion Vacuum Gripper Kit

by Lynxmotion

In stock

Product Highlights

- Compatible with AL5 series of robotic arms
- Pickup any small light object that has a smooth exterior finish
- Requires super-glue and common hand tools to construct
- Uses one servo channel

Product Code : **RB-Lyn-339**

USD $**41.95**

Saw Better Price? Start Price Match

Qty

1

10 to 100 USD $40.69 each

100 and more USD $39.47 each

Feedback

ADD TO CART

Add to Wishlist

Add to Compare

Ask our experts

Ask a question on the RobotShop co

Support

Description

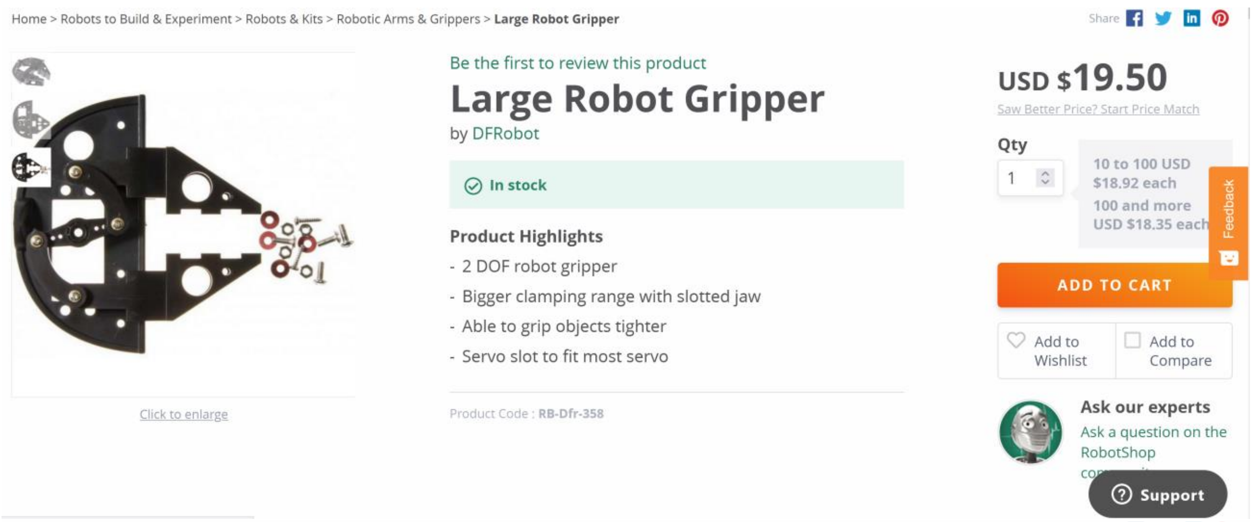
- Compatible with AL5 series of robotic arms
- Pickup any small light object that has a smooth exterior finish
- Requires super-glue and common hand tools to construct
- Uses one servo channel

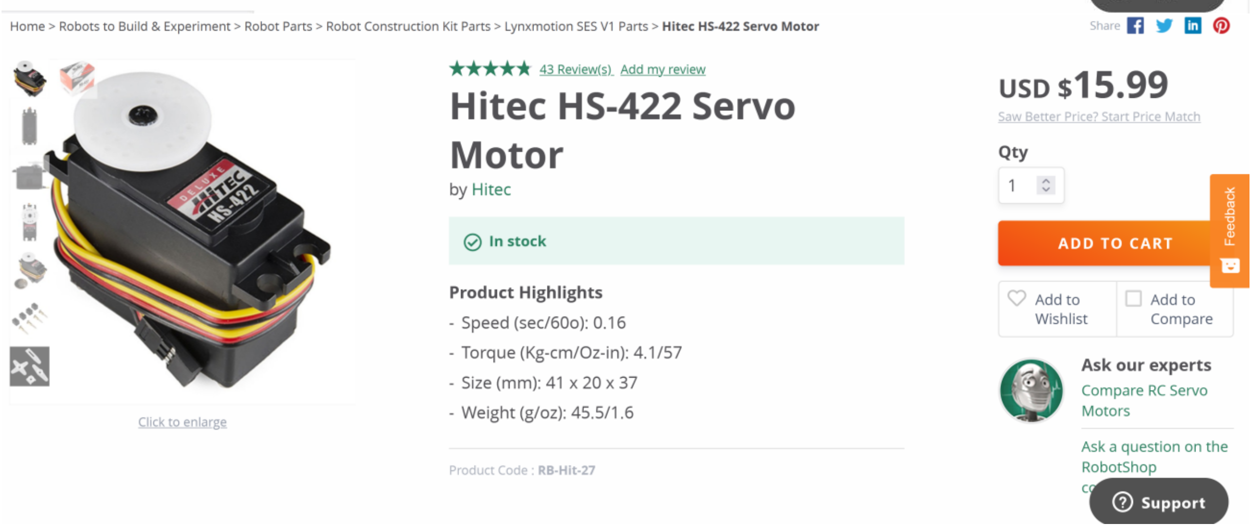
The **Lynxmotion Vacuum Gripper Kit** is a fun accessory to the AL5 series of robotic arms. This unique gripper uses an inexpensive syringe as the vacuum source. In the testing manufacturer was able to hold 3.5 ounces for over 30 minutes. The gripper will pickup any small light object that has a smooth exterior finish. The Vacuum Gripper uses one servo channel which removes/adds air to the vacuum tube.

”

Lab 8.2B

This lab requires the number in QW1000 to be modulated between two numbers to open and close the gripper. The numbers and command to open and close must be determined by trial and error. There must be a ramping of the numbers between the open and close values in order to move the gripper gradually instead of in a jerking manner. The speed of the move should be a variable controlled by the program.





The time to close and time to open should be programmable and controlled. The speed at which these grippers close should be a variable in the program. The signal is wired as shown below.

From the Hitec Manual:

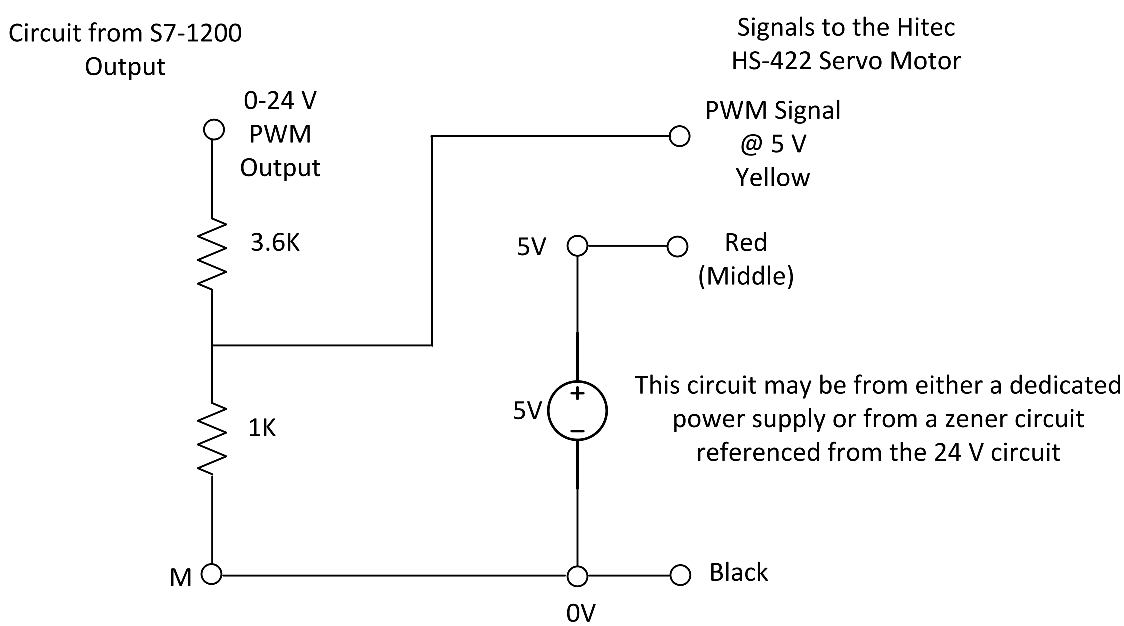
Pulse Data
All Hitec servos require 3-5V peak to peak square wave pulse. Pulse duration is from 0.9mS to 2.1mS with 1.5mS as center. The pulse refreshes at 50Hz (20mS).

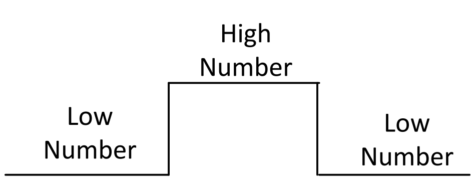
Voltage Range
All Hitec Servos can be operated within a 4.8V-6V. range.
Only the HS-50 operates exclusively with 4 Nicad cells (4.8 volt).

Wire Color Meanings
On all Hitec servos the Black wire is 'ground', the Red wire (center) is 'power' and the third wire is 'signal'.

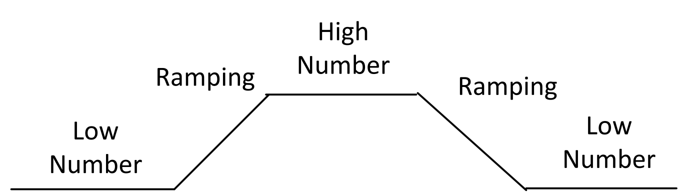
Direction of Rotation
All Hitec servos turn Clockwise direction (CW)

The circuit below shows the electrical connection and design to be used in the program for controlling the two grippers:



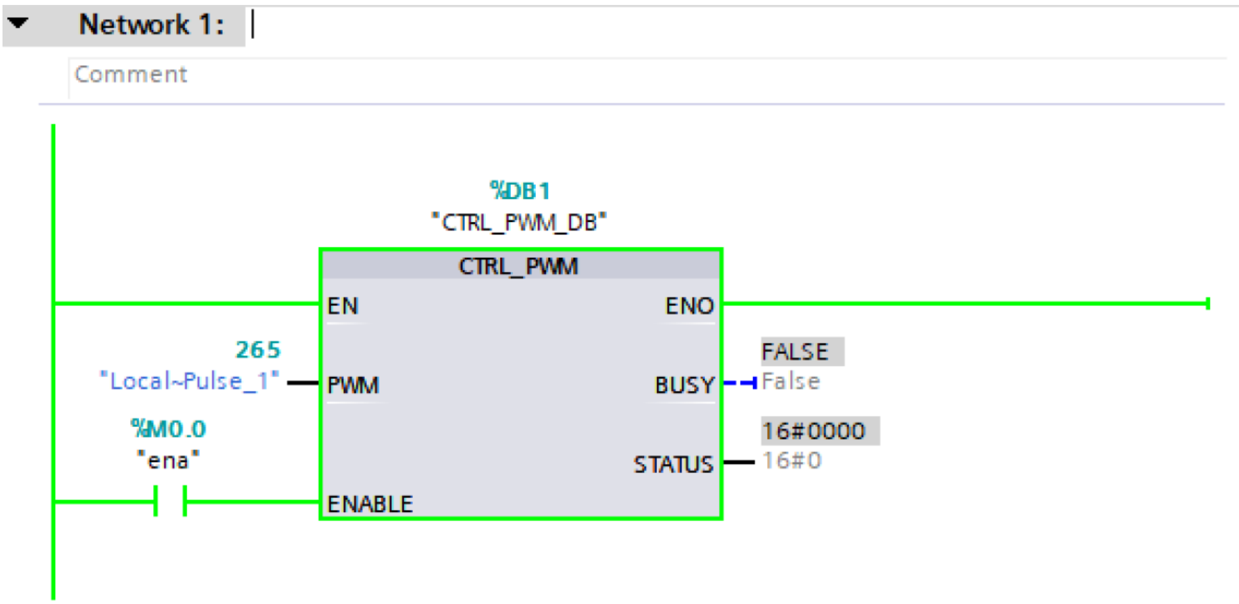


Circuit for Vacuum Gripper

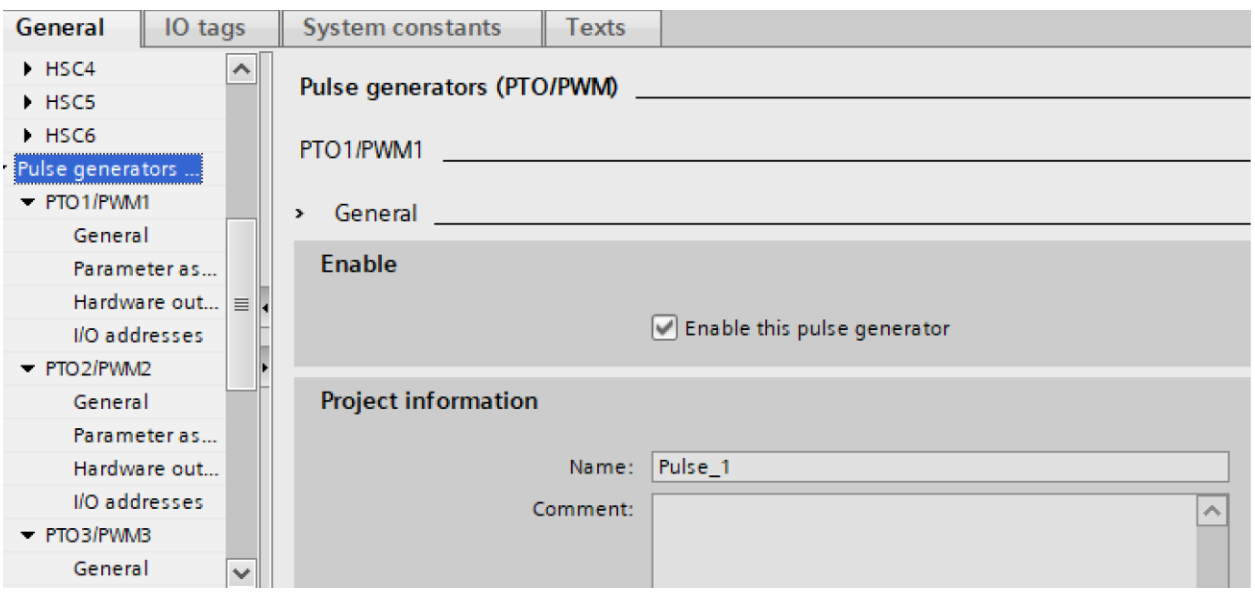


Circuit for Hand Gripper

The following figures outline the method of setting up the PWM output for the PLC and servo. The next figure shows the PWM instruction which must be added to the program. It may be inserted in any OB that is active. This instruction is inserted in OB1.

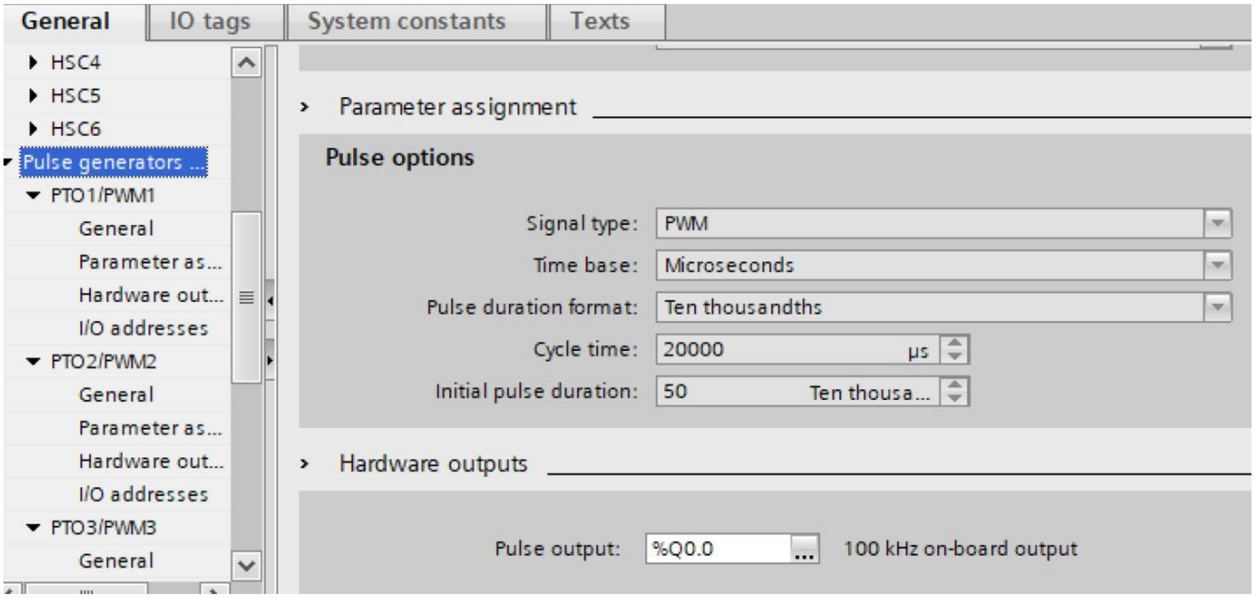


The configuration of the PWM for output on output 0.0 for the above servo is as follows:

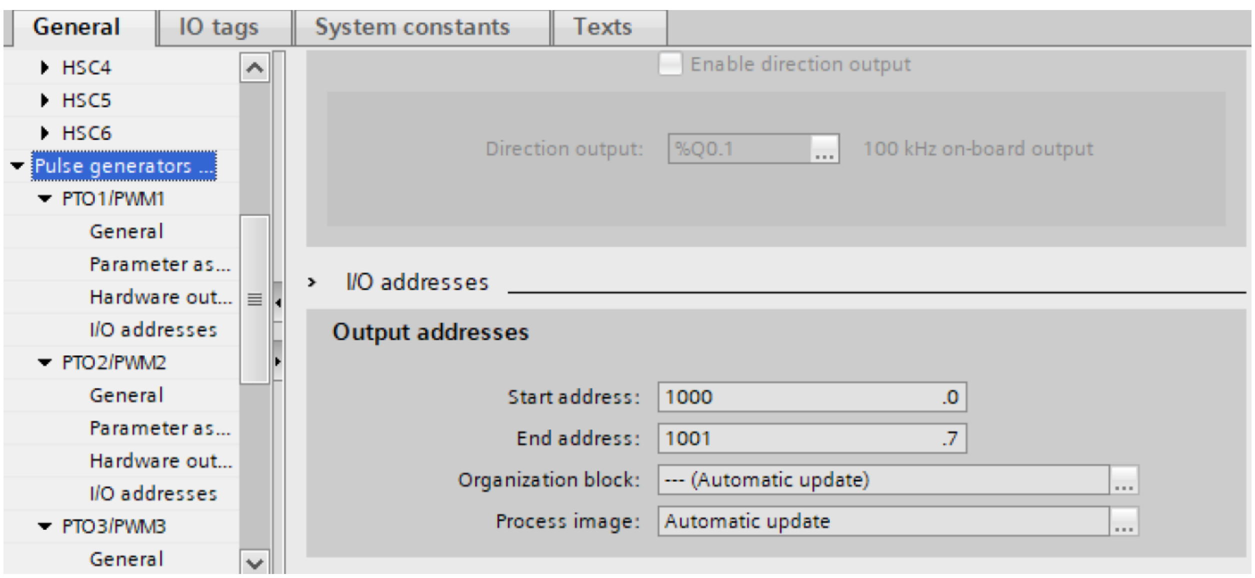


This must be done before the program is loaded and run. It should also be saved before running.

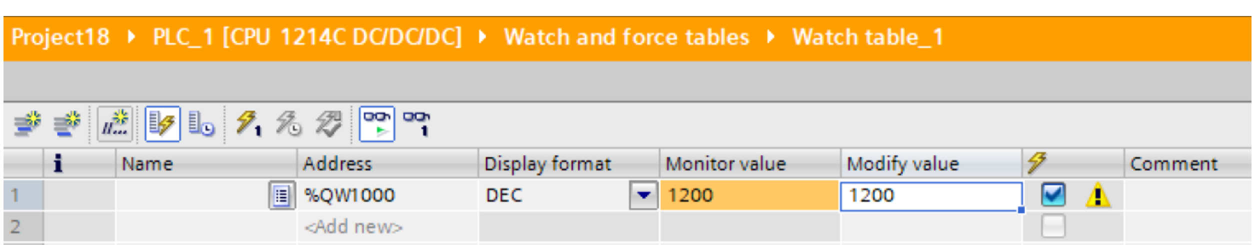
The following sets up the pulse duration for the servo above that requires a pulse duration of 20 msec.



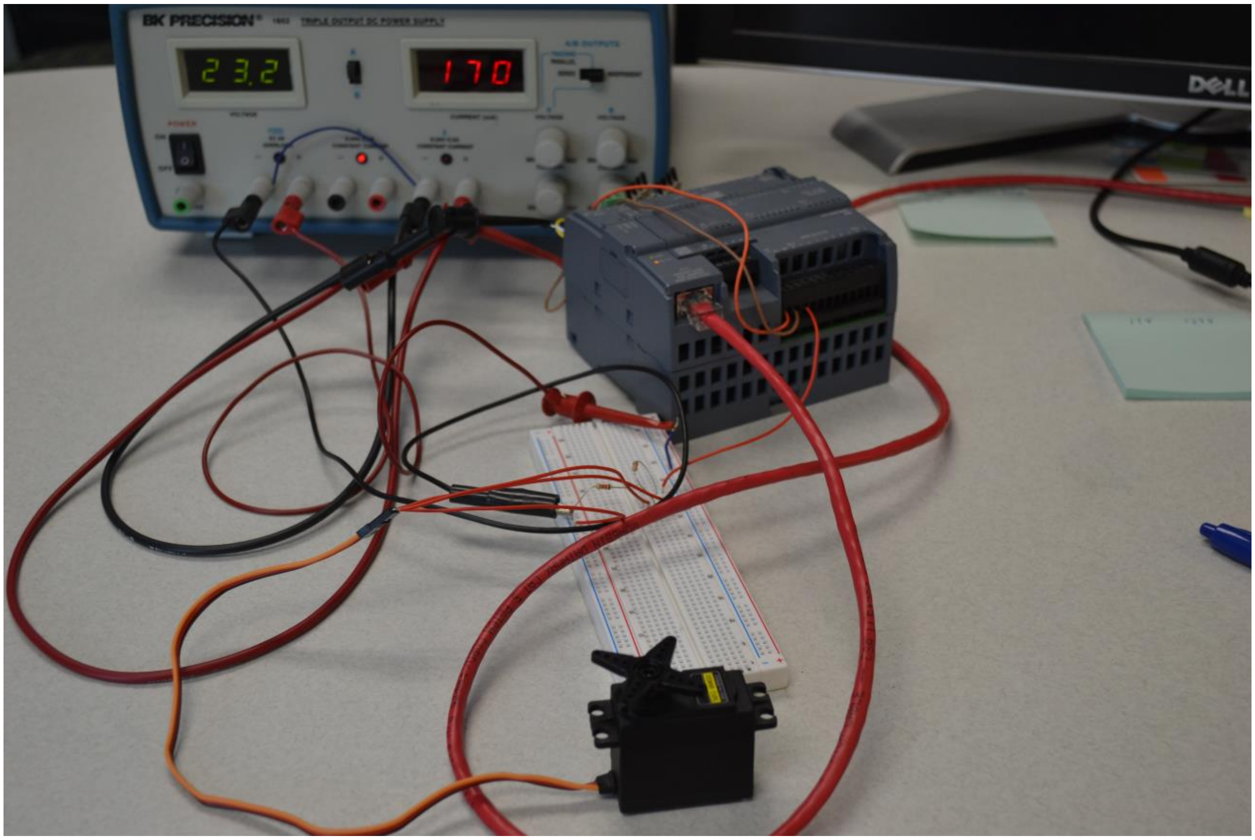
The following sets up the output word for entry of the pulse length in QW1000 (includes 1001):



The above initialization allows the user (program) to input values in the QW1000 location to test the servo using the Watch Table:



The experiment concludes with the values for the limits of QW1000 found. Completely clockwise is 175 and completely counter-clockwise is 1275. The servo ranges from 0 to 180^o in the process. The power supply shown is a good way to provide the +5V and +24 V to the process. This power supply has a variable supply A and B that can be set to close to +24 V.



The experiment does not end here. This lab continues with a program in Lab Text Ch. 8. The inclusion of four servo controllers in Ch. 13 in the discussion of arrays and Lab Text Ch. 31 where the program for moving of the robot built with the four servo controllers is discussed.