PLC Programming with

STUDIO 5000

Course: Level 2

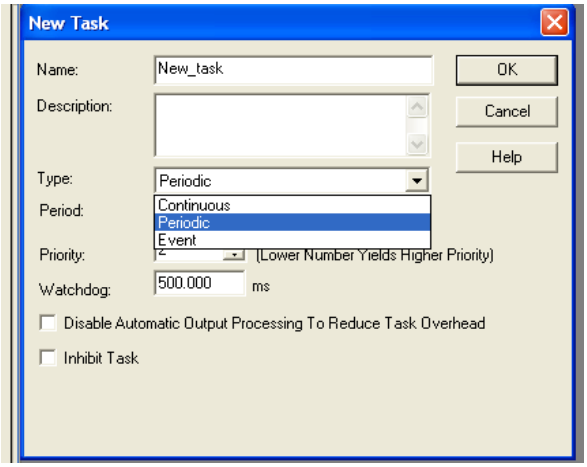
Program Allen-Bradley ControlLogix and CompactLogix PLCs with Rockwell Automation's Studio 5000 and FactoryTalk View Studio

TABLE OF CONTENTS

Introduction To Studio 5000

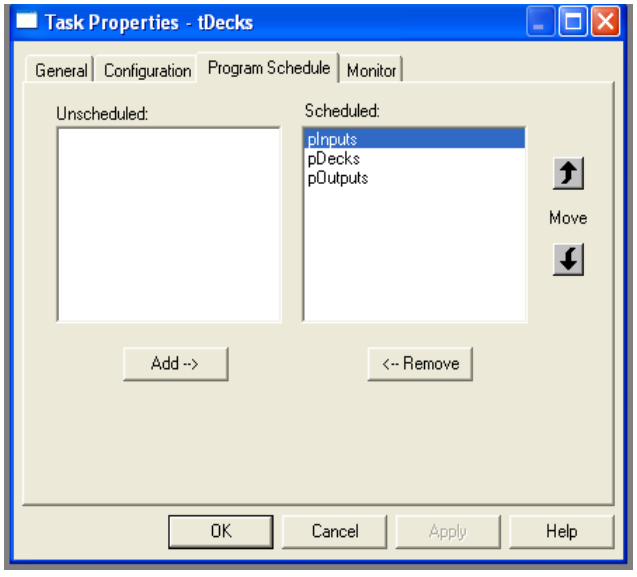
Tasks: Configure Controller Execution

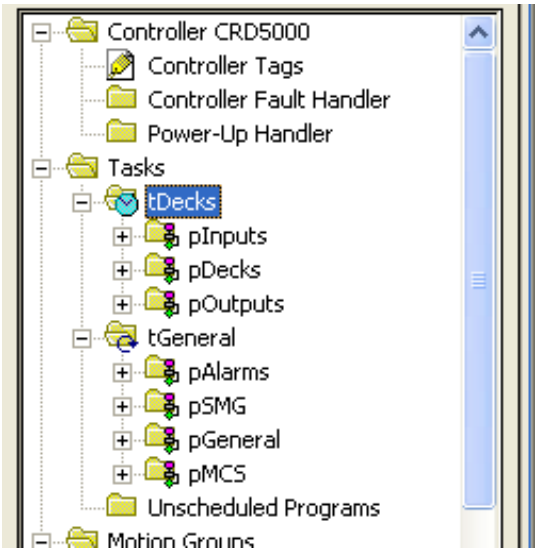
A task provides scheduling and priority information for a set of one or more programs. You can configure tasks as either continuous, periodic, or event



Programs: Group Data and Logic

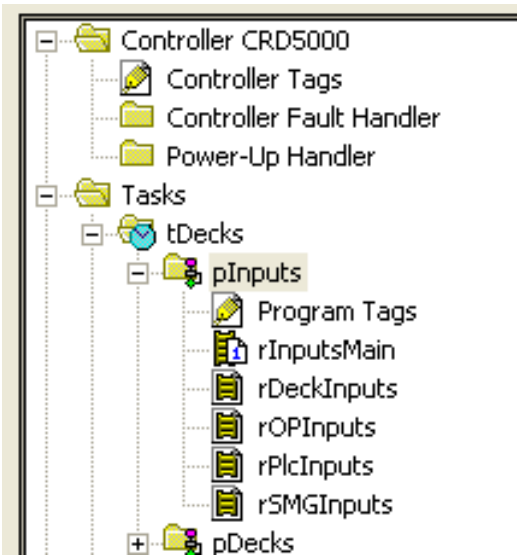
A task contains programs, each with its own routines and program-scoped tags. Once a task is triggered (activated), all the programs assigned to the task execute in the order in which they are listed in the Controller Organizer.

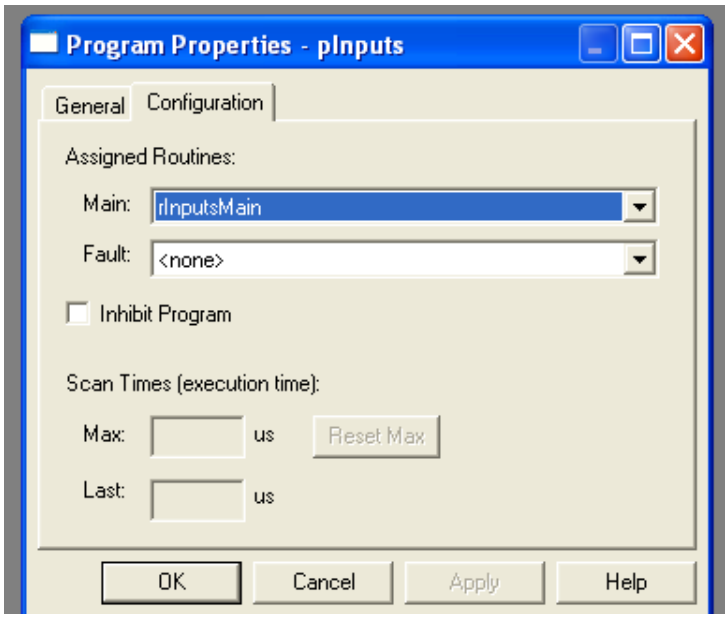




Routines: Encapsulate Executable Code Written In A Single Programming Language

Each program has a main routine that is the first routine to execute within a program. Use logic, such as the Jump to Subroutine (JSR) instruction, to call other routines. You can also specify an optional program fault routine.





Manage User Tasks

If you want logic to execute	Then use a	Description
All of the time	Continuous task	The continuous task runs in the background. Any CPU time not allocated to other operations or tasks is used to execute the continuous task. • The continuous task runs all the time. When the continuous task completes a full scan, it restarts immediately. • A project does not require a continuous task. If used, there can be only one continuous task
• At a constant period (such as every 100 ms) • Multiple times within the scan of your other logic	Periodic task	A periodic task performs a function at a specific time interval. Whenever the time for the periodic task expires, the periodic task: • interrupts any lower priority tasks. • executes one time. • returns control to where the previous task left off.
Immediately when an event occurs	Event task	An event task performs a function only when a specific event (trigger) occurs. Whenever the trigger for the event task occurs, the event task: • interrupts any lower priority tasks. • executes one time. • returns control to where the previous task left off.

Specify Task Priorities

Each task in the controller has a priority level. A higher priority task (such as 1) interrupts any lower priority task (such as 15). The continuous task has the lowest priority and is always interrupted by a periodic or event task.

If a periodic or event task is executing when another is triggered and both tasks are at the same priority level, the tasks time slice execution time in 1 ms increments until one of the tasks completes execution.

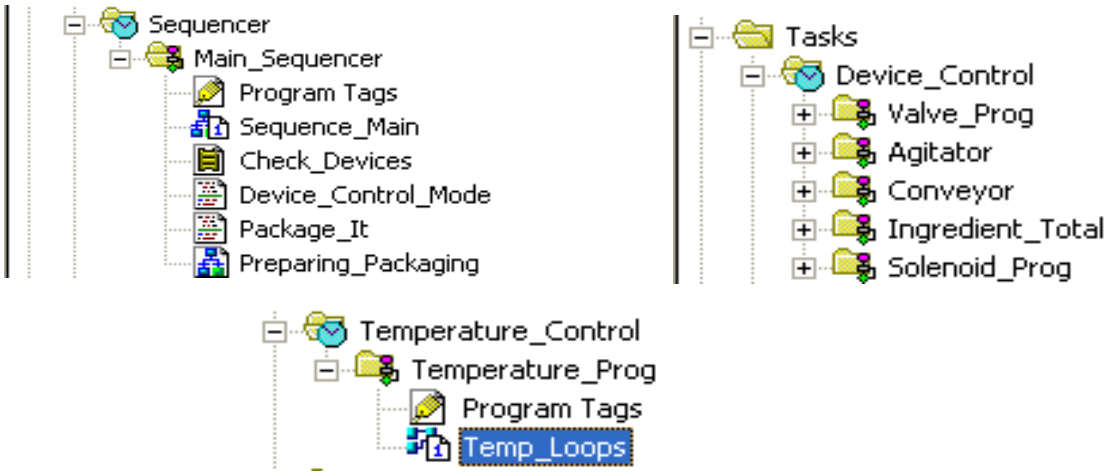
When to Use Tasks, Programs, and Routines

Comparison	**Task**	**Program and Equipment Phase**	**Routine**	**Add-On Instruction**
Quantity available	Varies by controller (4, 6, 8, or 32)	32 program and equipment phases (combined) per task (100 for ControlLogix and SoftLogix controllers)	Unlimited number of routines per program	Unlimited number of Add-On Instructions in a project
Function	Determines how and when code will be executed	Organizes groups of routines that need to share a common data area	Contains executable code (relay ladder, function block diagram, sequential function chart, or structured text)	Contains executable code (relay ladder, function block diagram, or structured text)
Use	• Most code should reside in a continuous task • Use a periodic task for slower processes or when time-based operation is critical • Use an event task for operations that require synchronization to a specific event	• Put major equipment pieces or plant cells into isolated programs • Use programs to isolate different programmers or create reusable code • Configurable execution order within a task • Isolate individual batch phases or discrete machine operations	• Isolate machine or cell functions in a routine • Use the appropriate language for the process • Modularize code into subroutines that can be called multiple times	• Standardize modules of code • Very specific or focused operations • Extensions to the base instruction set • Encapsulate an instruction from one language for use in another language • Instance based monitoring of logic and data
Considerations	• A high number of tasks can	Data spanning multiple	• Subroutines with multiple	• If you have a lot of parameters or

	be difficult to debug • May need to disable output processing on some tasks to improve performance • Tasks can be inhibited	programs must go into controller-scoped area • Listed in the Controller Organizer in the order of execution	calls can be difficult to debug • Data can be referenced from program-scoped and controller-scoped areas • Calling a large number of routines impacts scan time • Listed in the Controller Organizer as Main, Fault, and then alphabetically	specialized options, consider multiple Add-On Instructions • Calling a large number of Add-On Instructions impacts scan time • Must use cross-reference or find to locate calls to an Add-On Instruction • Can edit offline only • Supports only some datatypes. • Changes to data values must be made for each instance

Task

- Used to isolate device controls: Valves, agitators solenoids
- Used to set up Sequencer in processes
- Used to isolate Temperature, Flow, level control loops
- Simulation task may be set up for testing purpose

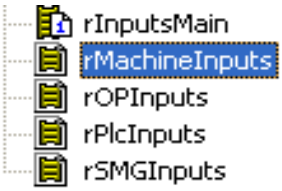


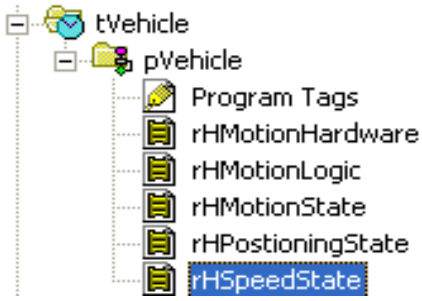
Programs

- Isolate Input, output , logic
- Reuse code for equipment using same logic and tags
- Isolate Alarms, communication, configuration preset and constants

Routines

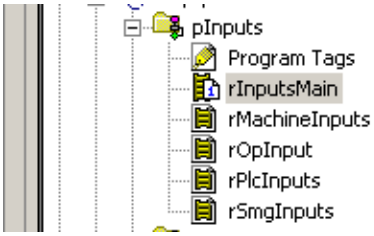
- Used to group or divide sections of Machinery example: equipment, operator panels, Electrical panels
- Used to isolate Process vertical and horizontal movement, Machine state, transitions and hardware

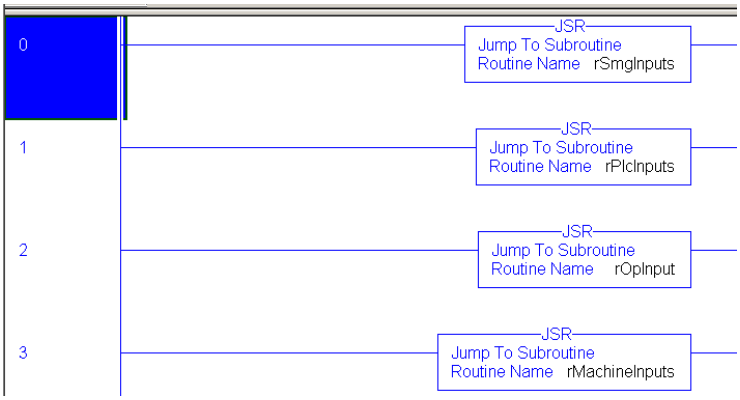




Jump Subroutine (JSR)

- Use a subroutine to store recurring sections of program logic that can be accessed from multiple program files.
- A subroutine saves memory because you program it only once.
- Program Organization





- JSR instructions direct the processor to go to a separate subroutine file within the ladder program, scan that subroutine file once, and return to the point of departure.
- If required, defines the parameters passed to and received from the subroutine.

- The processor does not update I/O until it reaches the end of the main program (after executing all subroutines). Outputs in subroutines are left in their last state.

Address Data

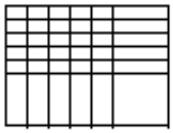
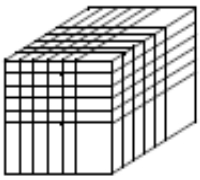
- Logix5000 controllers support **atomic data types**, such as **BOOL, SINT, INT, DINT, LINT**, and **REAL**.
- Also support compound data types, such as **arrays**, **predefined structures** and **user-defined structures.**
- CPU reads and manipulates 32-bit data values.
- Minimum memory allocation for a tag is 4 bytes. When tag that stores data that is less than 4 bytes, the controller allocates 4 bytes, but the data only fills the part it needs.

Data Types

Data Type	**Byte 0**					
	31	**16**	**15**	**8**	**7 1**	**0**
BOOL	**Allocated But Not Used**					**0 or 1**
SINT	**Allocated But Not Used**				**-128…127**	
INT	**Allocated But Not Used**		**-32,768…32767**			
DINT	**-2,147,483,648…2,147,483,647**					
REAL	**-3.40282347E38...-1.17549435E-38 (negative values)** **0** **1.17549435E-38...3.40282347E38 (positive values)**					

Arrays

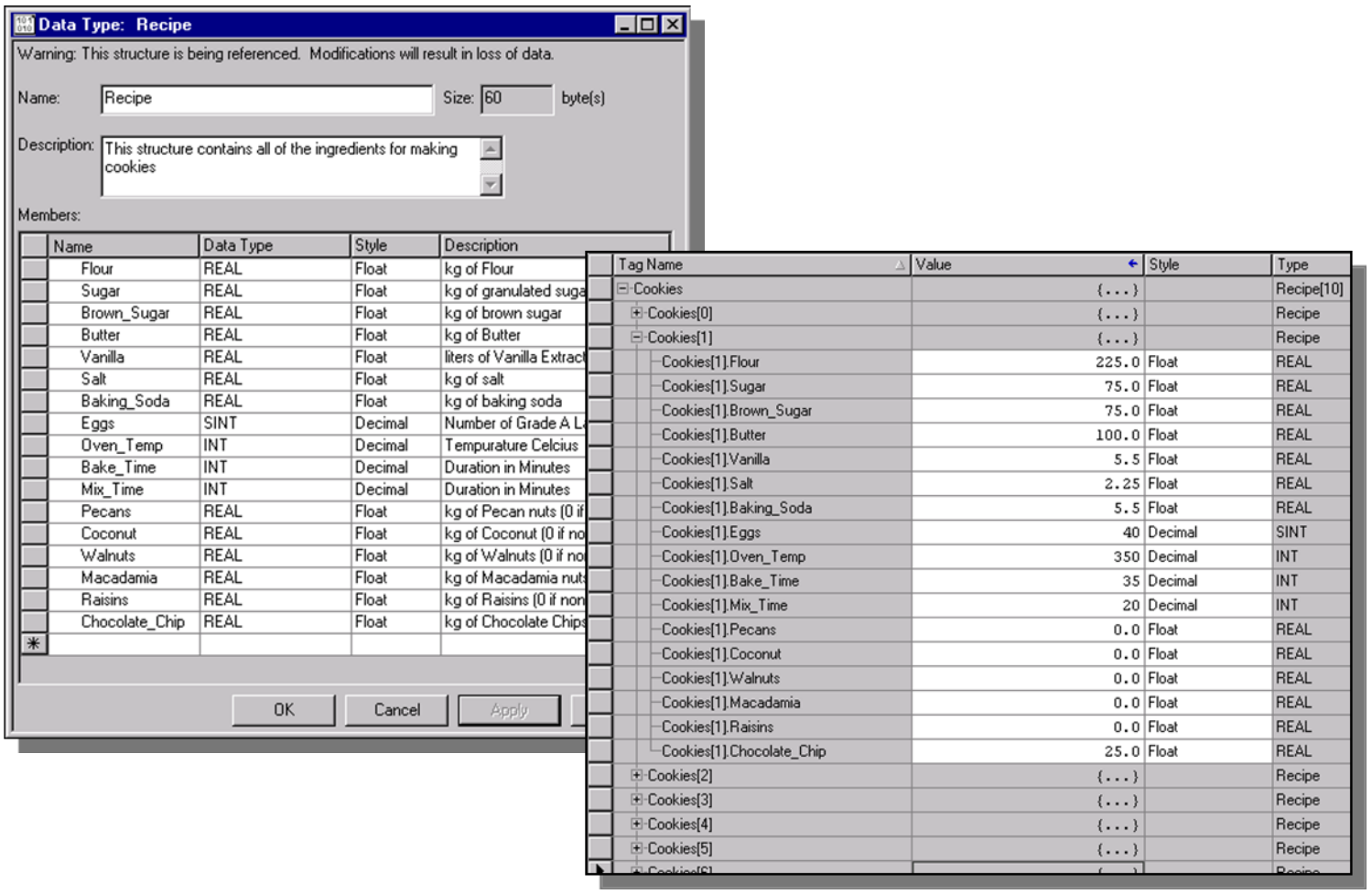
- An array allocates a contiguous block of memory to store a specific data type as a table of values.
- Tags support arrays in one, two, or three dimensions.
- User-defined structures can contain a single-dimension array as a member of the structure

This array	Stores data like	For Example				
One dimension		Tag name	Type	Dimension 0	Dimension 1	Dimension 2
		one_d_array	DINT[7]	7	--	--
		Total number of elements = 7 Valid subscript range DINT[*x*] where *x*=0...6				
Two dimension		Tag name	Type	Dimension 0	Dimension 1	Dimension 2
		two_d_array	DINT[4,5]	4	5	--
		Total number of elements = 4 * 5 = 20 Valid subscript range DINT[*x*,*y*] where *x*=0...3; *y*=0...4				
Three dimension		Tag name	Type	Dimension 0	Dimension 1	Dimension 2
		three_d_array	DINT[2,3,4]	2	3	4
		Total number of elements = 2 * 3 * 4 = 24 Valid subscript range DINT[*x*,*y*,*z*] where *x*=0...1; *y*=0...2, *z*=0...3				

The data type you select for an array determines how the contiguous block of memory gets used.

User Defined Structures

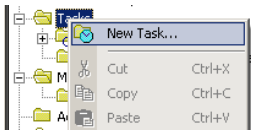
- IEC61131-3 compliant user defined data types
- Provides flexibility in control system design
- Allows for logical grouping of control data
- Allows mixing of data types in a structure
- Results in self-documenting code

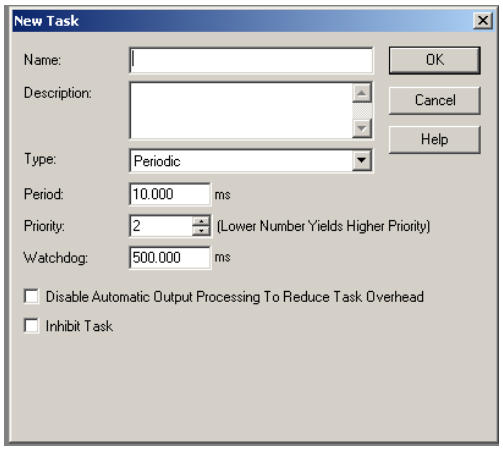


CREATE TASK

Right click Task folder → click New Task

Set *Name (prefix with "t")*
Type
Period
Priority

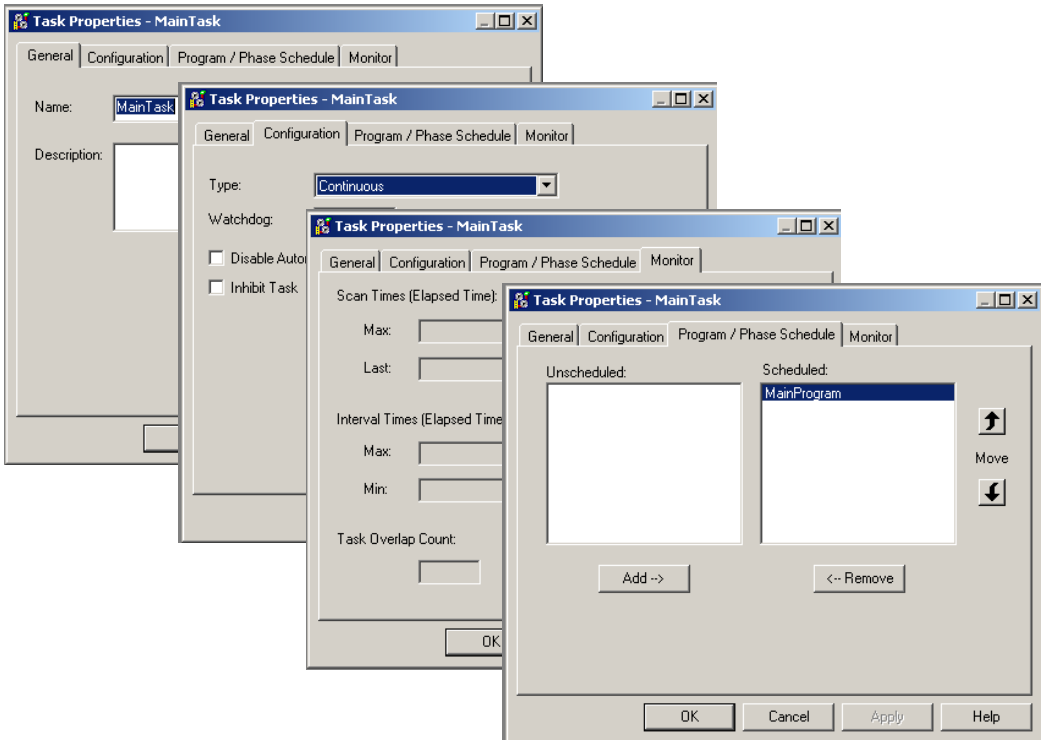




TASK PROPERTIES

Right click named task folder → click properties

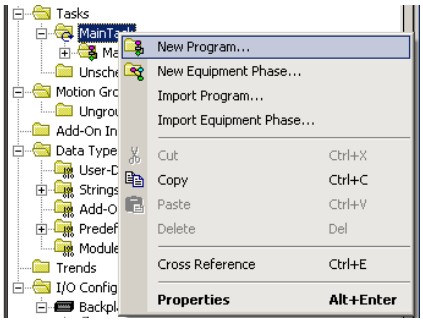
Tabs *Configuration*
Program schedule
Monitor

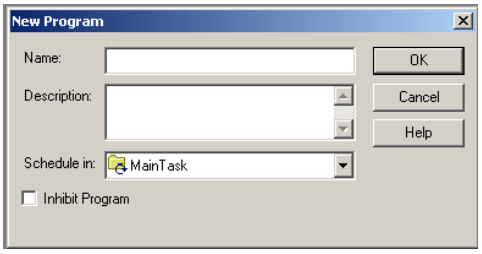


CREATE PROGRAMS

Right click name Task folder → click New Program

Set *Name (prefix with "p")*
Schedule
Inhibit Program



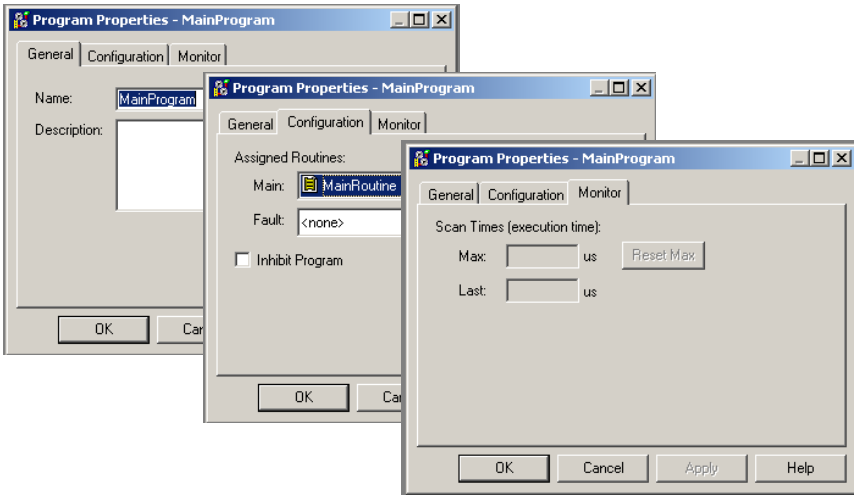


PROGRAM PROPERTIES

Right click named Program folder → click properties

Tabs Configuration

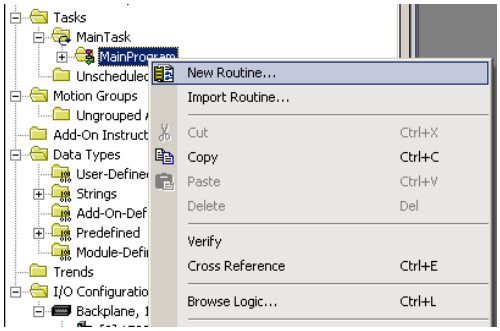
→Set *Main routine*
Fault routine (Instruction fault ex. Division by 0 or overflow)

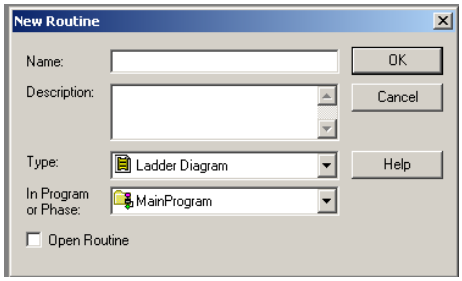


CREATE ROUTINE

Right click name Program folder → click New Routine

Set *Name (prefix with "r")*
Type
In Program

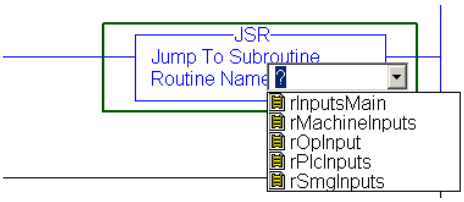




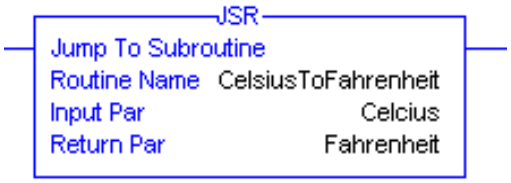
CREATE SUBROUTINE

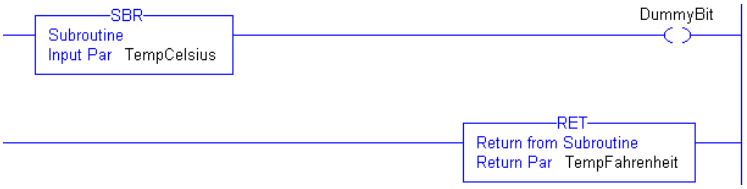
Drag and drop instruction into rung from the program control menu in instruction tool bar**.**

From the Routine Name drop down menu select a previously created routine to jump to



To pass input or return parameters enter the tags in the In Par And Return Par Section. The subroutine SBR and RET Instructions must be used to accept and return parameters

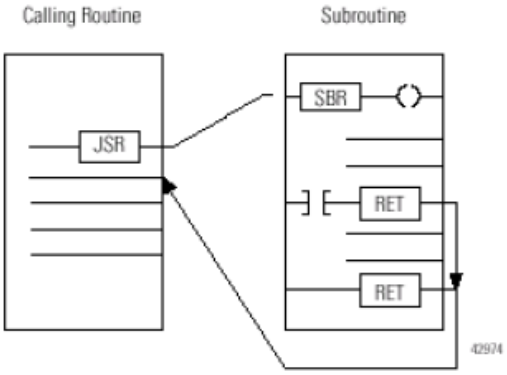




- Enter At least minimum expected Input and Return Expected or controller will Fault
- The JSR, SBR, and RET instructions all have special menu options available from the Edit > Edit Ladder
- Choose Add Input Parameter ,Add Output Parameter and Remove Instruction Parameter

JSR

1. If you want to copy data to a tag in the subroutine, enter an input parameter.
2. If you want to copy a result of the subroutine to a tag in this routine, enter a return parameter.
3. Enter as many input and return parameters as you need.



SBR

1. If the JSR instruction has an input parameter, enter an SBR instruction.
2. Place the SBR instruction as the first instruction in the routine.
3. For each input parameter in the JSR instruction, enter the tag into which you want to copy the data.

RET

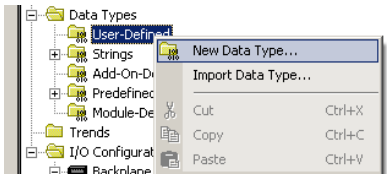
1. If the JSR instruction has a return parameter, enter an RET instruction.
2. Place the RET instruction as the last instruction in the routine.
3. For each return parameter in the JSR instruction, enter a return parameter to send to the JSR instruction.
4. In a ladder routine, place additional RET instructions to exit the subroutine based on different input conditions, if required. (Function block routines only permit one RET instruction.)

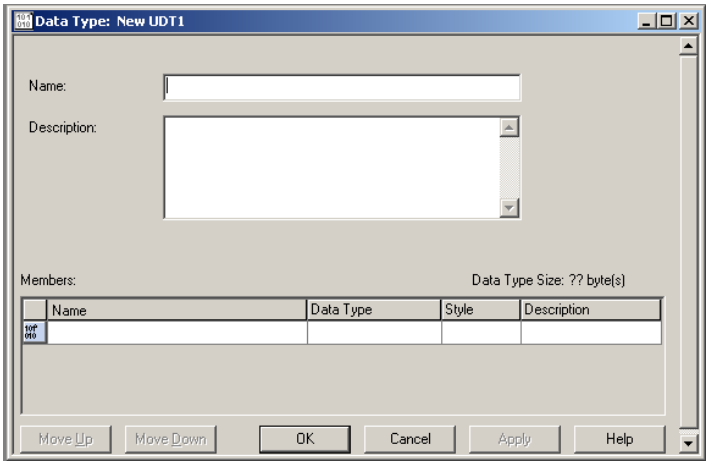
CREATE USER DEFINED TYPE

Click Data Type folder→ Right click name UDF folder → click New Data Type

Set *Name*

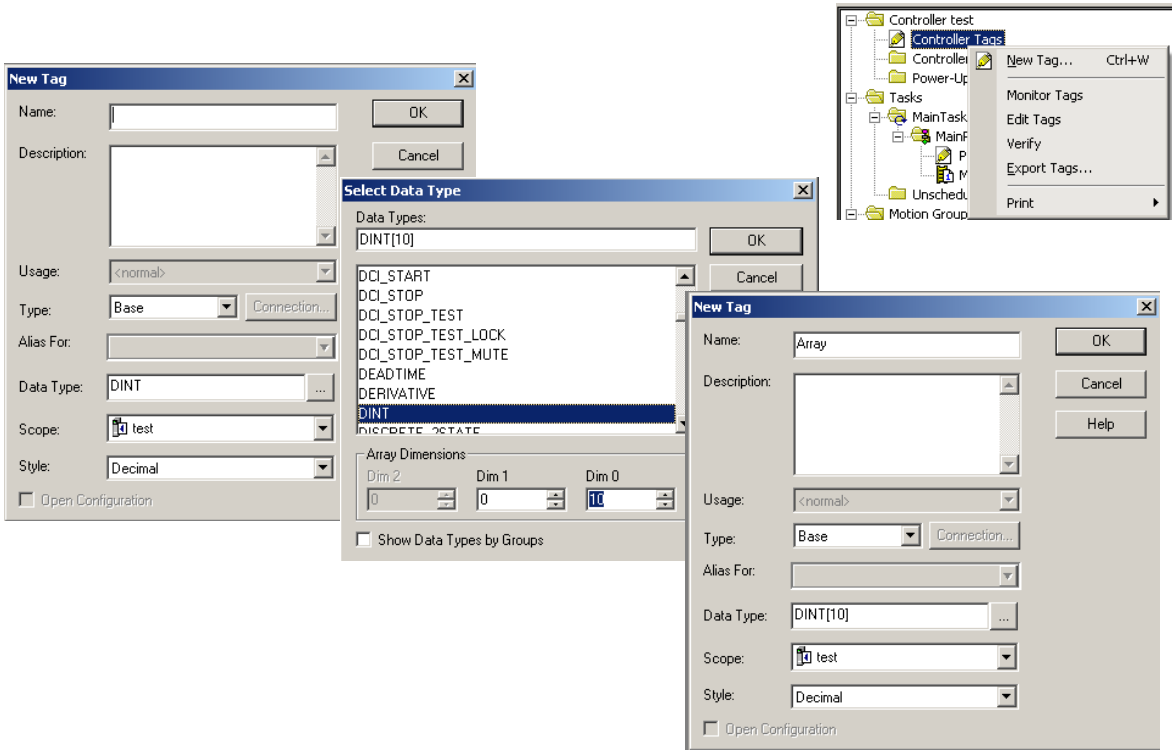
Members





CREATE AN ARRAY TAG STRUCTURE

Right Click Contoller tag and select New tag-> press the elispes next to **Data Type** and Seclect a Data Type.In the Dim 0 change the array dimensions and press OK



ASSIGNMENTS

Assignment: 1

Create the following:

3-task→ continuous, periodic, event task

Continuous Task – tCommunication

(1) -Program → pSystemCom

pSystemCom routines →rSystemComMain

rtelegramSend,

rTelegramRecieve

Periodic Task – t*Equipment*

(3)-Program → pInputs, pMachine, pOutputs

pInput program's routines →rInputsMain

rConveyor Inputs

rOpInput

rPlcInputs

rSmgInputs

pMachine program's routines→r Conveyor Main

rAuto

rMaintenance

rHardware

pOutputs program's routines→rOutputsMain

rConveyor Output

rOpOutput

rPlcOutput

rSmgOutput

Event Task – tShutdown

1 -Program → pPlantShutDown

pPlantShutDown program's routines

→ rShutdown

1 User defined data type→Conveyor

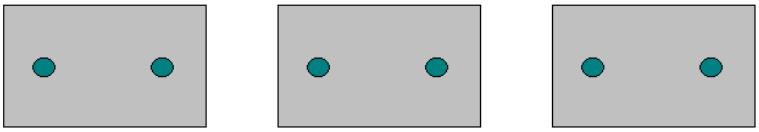
Members: states (dint), dInput (dint), dOutput (dint), dLogicWords (array of dint[10]), tLogic (array of timers[10]) , dLogicBits (dint)

Notes:

Equipment: starter, load and limit switches, photocells, Aux contacts
Operator panel: joysticks pushbuttons, switches, lights, solenoids
SMG (Small maintenance group): maintenance, auto, off, cycle stop, cycle start
PLC: auto, semi, e-stop (Key switches), System fault light and reset button

Assignment 2

Write the Logic for 1 conveyor, called deck#2 that has 2 photo eyes. There are 2 decks adjacent to the deck, the "previous deck" deck #1 and the "next deck" deck #3. If the Previous deck has a unit that is ready to send and the deck#2 is empty, the deck motor should start loading the Unit. Once the unit has reached the second photo eye of deck #2, it will be loaded and the motor will stop. If the next deck is empty and ready to receive a unit then the deck #2's Motor will start unloading the unit. When Both Photo eyes are clear, deck #2 will be considered empty and the motor will stop.

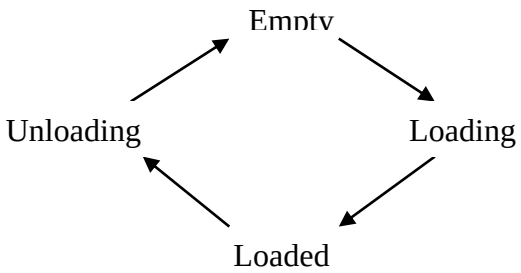


deck #1 deck #2 deck #3

Using the User Type" Conveyor" define Map following IO to pInput and pOutput programs and the appropriate routines.

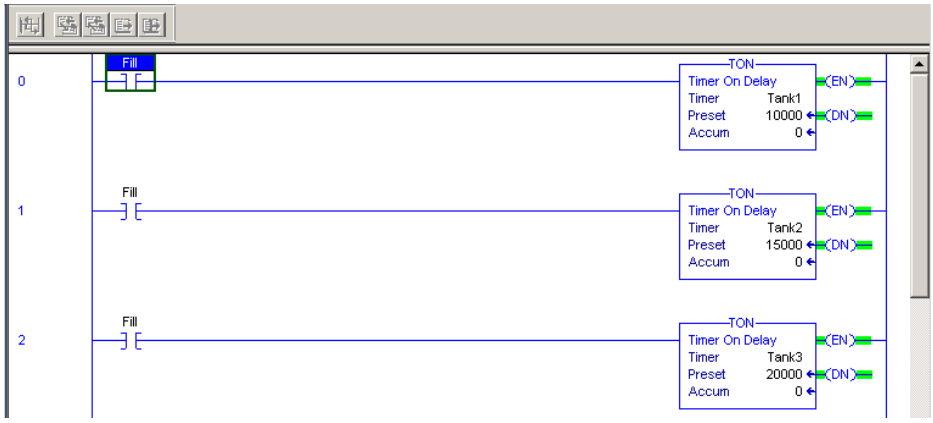
Small Maintenance Group Inputs:	Small Maintenance GroupOutputs:
Cycle Start Maintenance Mode Auto Mode Machine Inputs North Photo Eye South Photo Eye Operator Panel Inputs Motor Forward Pushbutton Motor Reverse Pushbutton	Cycle Start Indication Light Maintenance Mode Indication Light Auto Mode Indication Light Machine Outputs: Motor Forward Motor Reverse Operator Panel Outputs: Motor Forward Light Motor Reverse Light

In Program file pMachine, complete the routine rAuto, using the following State machine.



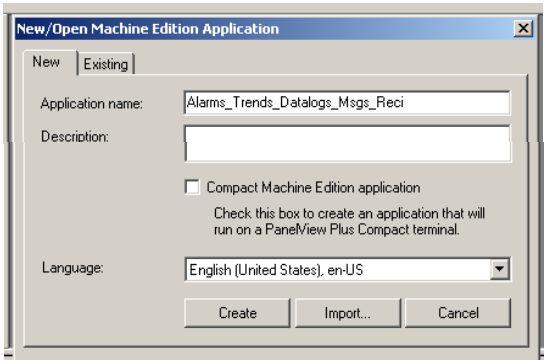
Recipe Application
Using the State Conditions, create Logic that will control the physical outputs in routine rHardware.
Data Logging, Trends, Alarms, Messaging and Recipes

In Studio 5000 create a routine to Simulate 3 tanks filling to different levels.
Download the program to the controller.

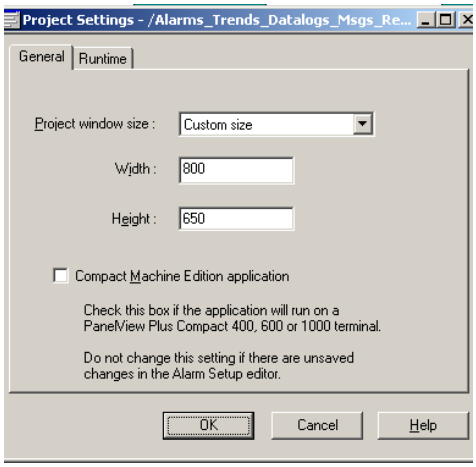


Open FTA Studio ME and create an application.

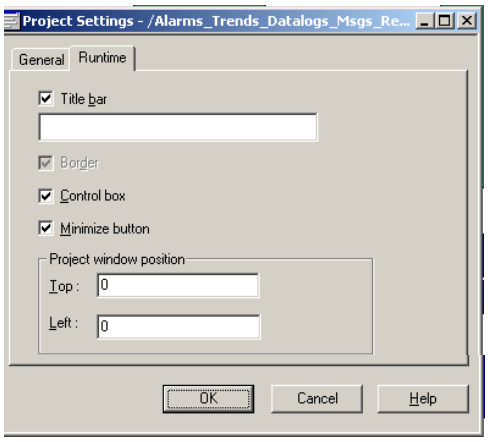
Right Click Project Setting And **Select** Open.



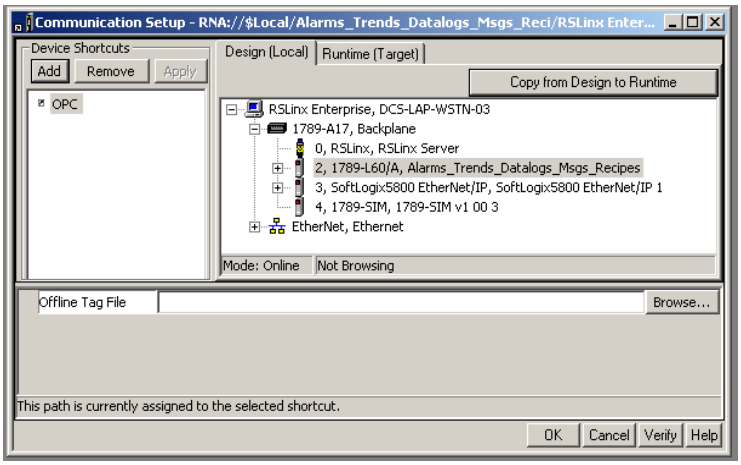
In the General tab, **Select** Customize size and change the width to 800 and the height 650.



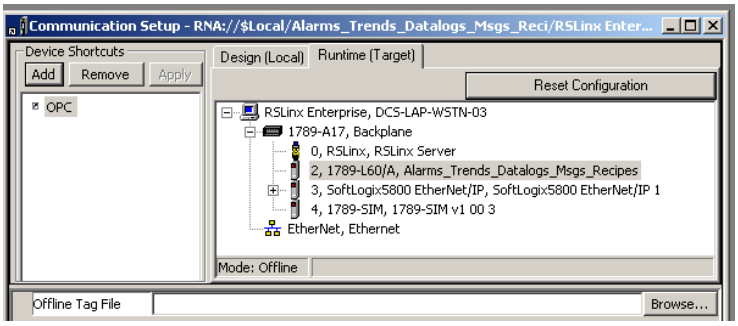
On the Runtime Tab, check the Title bar box.



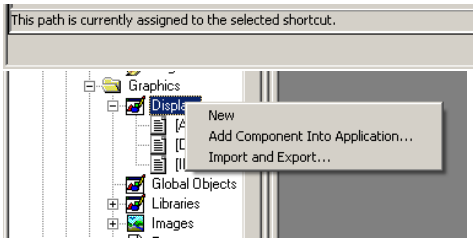
Through Rslinx Enterprise establish communication by creating a device short (topic)pointing to the controller, with your dowloaded application. Press “Copy from Design to Runtime.”



The Runtime target will be the same as design (local) configurations.
Then Press OK button.

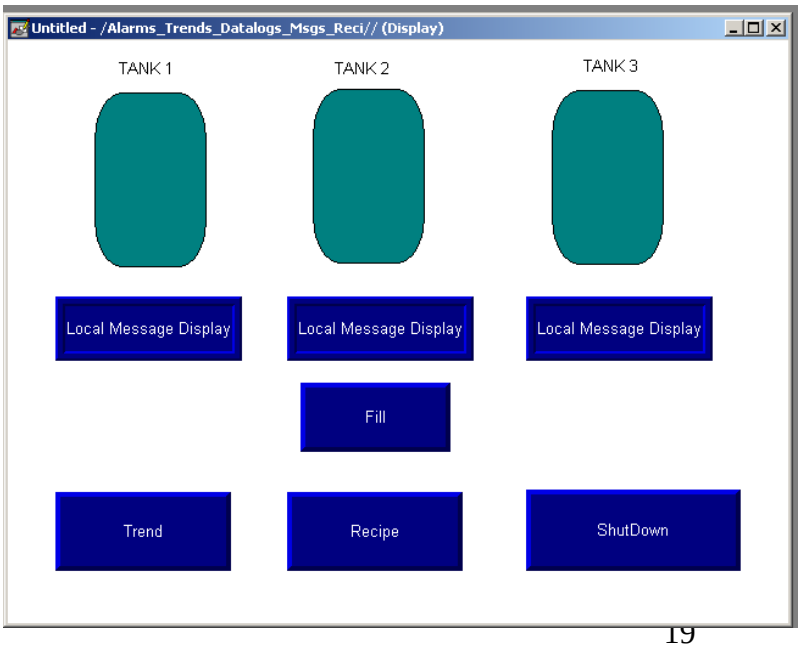


Go to the Graphics Folder and **Right Click** the display icon and **Select** New.

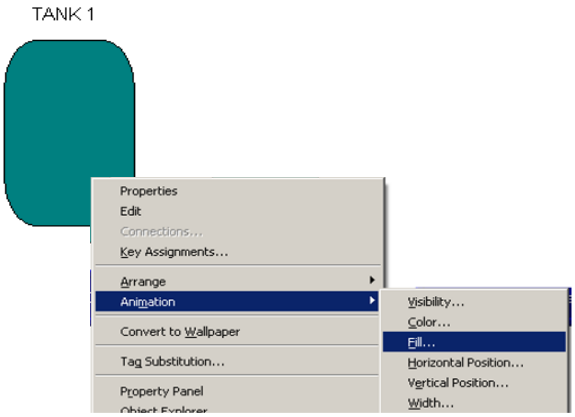


Create the following graphic display:

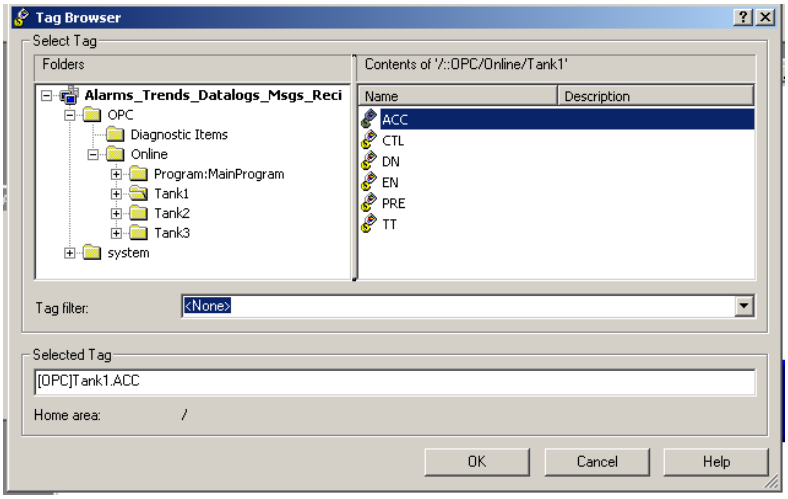
- 3 tank using rounded rectangles
- 3 local Message displays
- 1 maintained push button to control the fill condition
- 2 “Goto Display” buttons, one for a trend the other for a recipe
- 1 shut down button



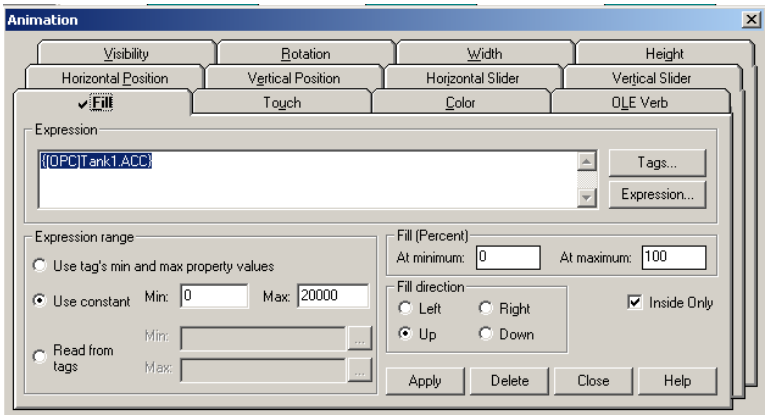
Right Click the tank, highlight animation and **Select** fill.



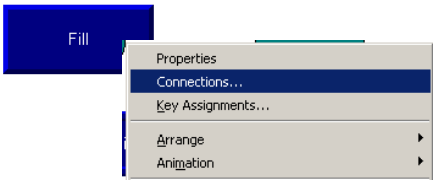
In the animation screen, Press tags. In the tag browser find your device short cut **Select** tank folder, then **Select** .ACC value and Press O.K



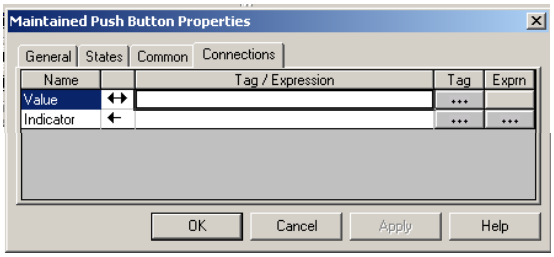
The tag will be displayed in the animation screen. Under the expression text box set the expression range to constant with a min of 0 and a max of Tank's Preset Value. In the fill direction section check inside only.



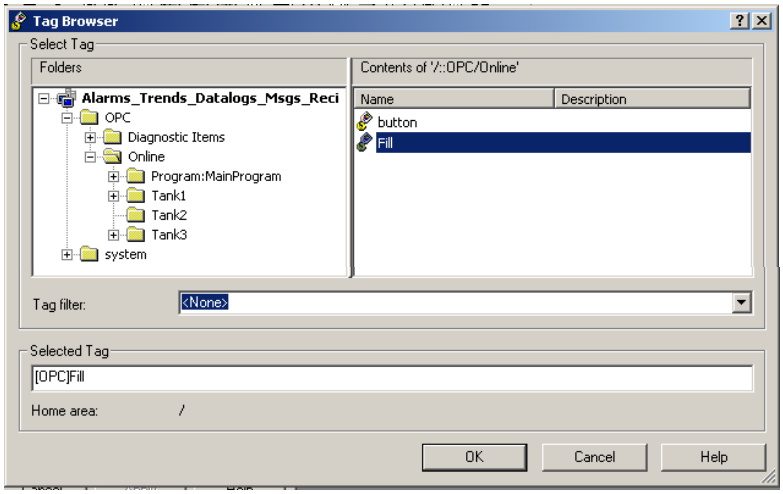
Right Click the fill button and **Select** Connection



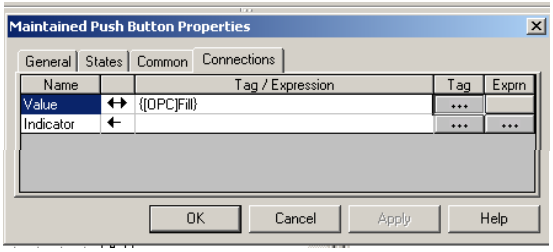
On the Connection tab **Press** ellipse box under the tag column.



In the tag browser, **Select** the online folder or the program folder and then **Select** the fill tag. Press OK.



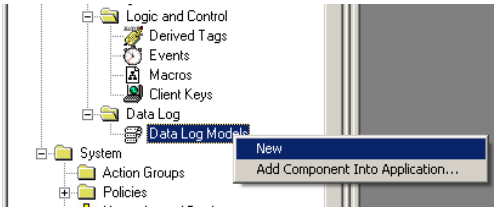
Under Tag/Expression column, the fill tag will be displayed. You can test display to verify that the tank can be controlled by the fill push button.



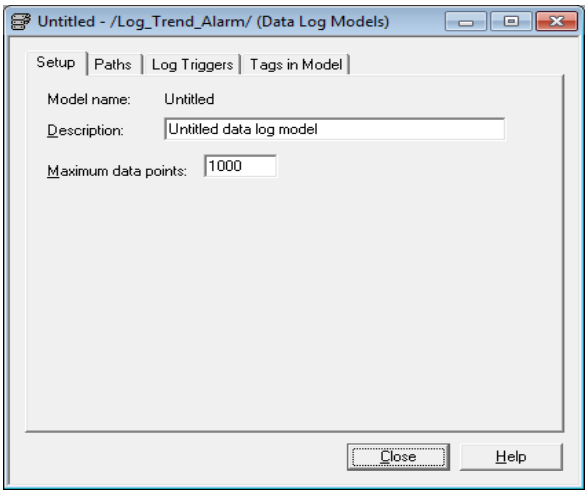
Data Logging

Data log is a FactoryTalk® View component that collects and stores tag values. Logged data will be stored in an internal file set and can be displayed in trends or archived for future.
A data log model defines which tags to log data for, when to log the data, and where to log the data. The data log model can contain up to 100 analog or digital tags and you can log a maximum of 1,000,000 points for version 7.00.00 or later.

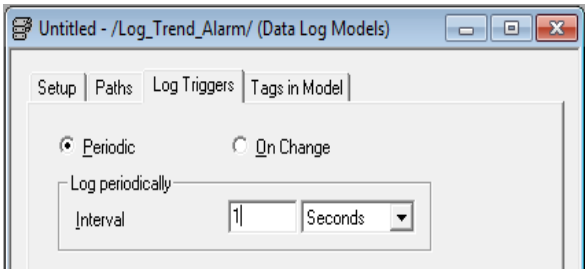
In FactoryTalk View Explorer, expand the data Log folder And right click the Data Log Model and Select New



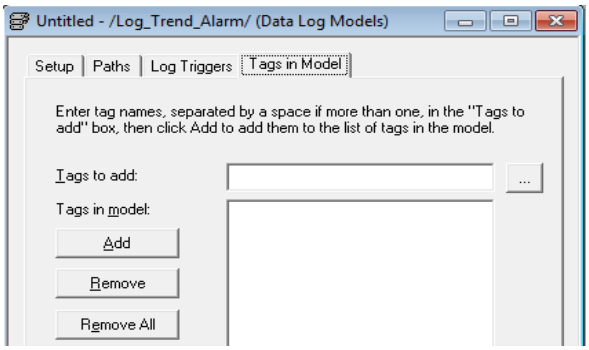
Enter a Description and accept the default Max Data Point which is 1000.



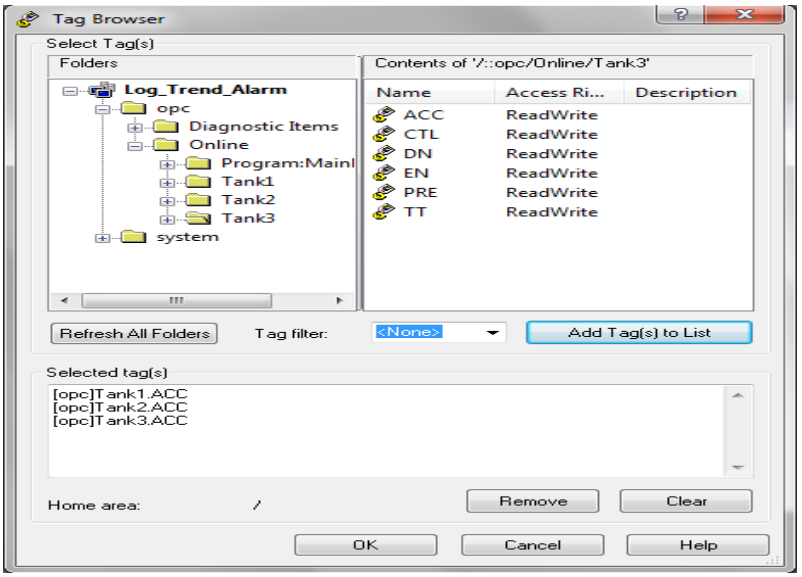
On the Log Triggers tab Select Period and change the Log periodically interval to 1 second



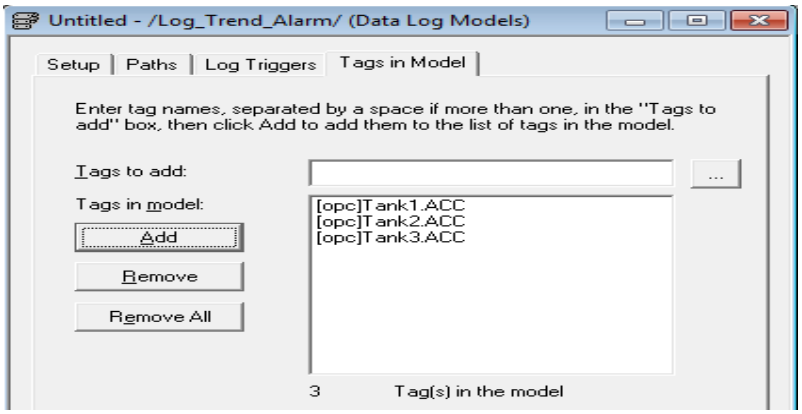
Next to the Tag(s) to add box press the ellipse to Access the tag browser.



Through the tag browser select the appropriate tags and press ok.

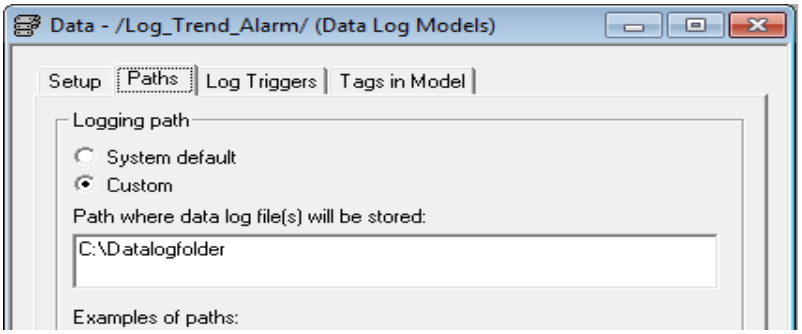


Press Add and attach tags to the model.



Go to the Paths tab and choose custom To add a path to the data logging files.

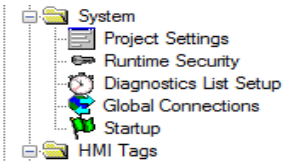
Then press Close to create the Data Log Component.

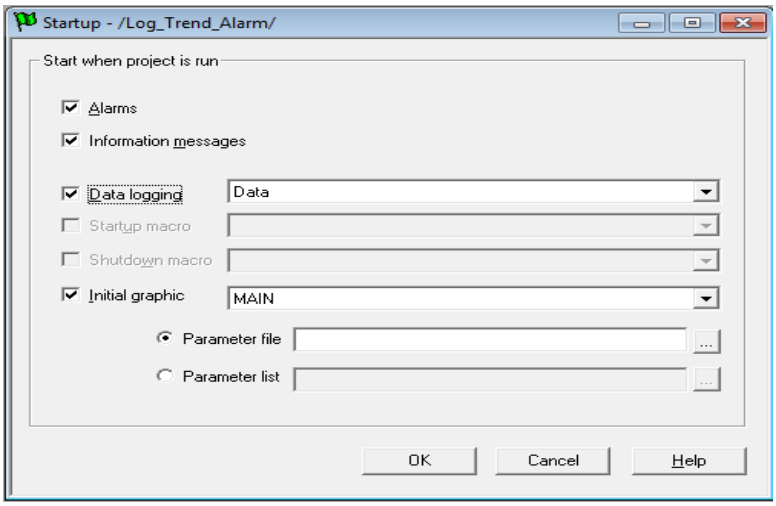


The application will prompt you to save Component, Give it an appropriate name and press ok

The new component will be visible in the project explorer under the Data log folder

After The Data Log Model is created the logging must be enabled. To enable it in FactoryTalk Explorer double click the Startup and check the Data Logging.
In the Drop down box select the Data logging model that you have created Then Press OK.





At this point the data logging is set and Ready to start. To start it the application must Run. Go to Menu bar under Application Menu click Test Application. At this stage FactoryTalk View will create Run Time application and will run it. There will be a alarm message about “Local Message Display” just close the alarm window. One folder with the name of Data Log Model will be created in the Data Log Path. In this folder two files which hold the log data are created. Try to find them. Check the file size. Then in the factory talk change the Maximum data points and check the size of these two files. If you have any problem doing this ask your teacher for proper instructions.
Press Shutdown in the application window to close the application.

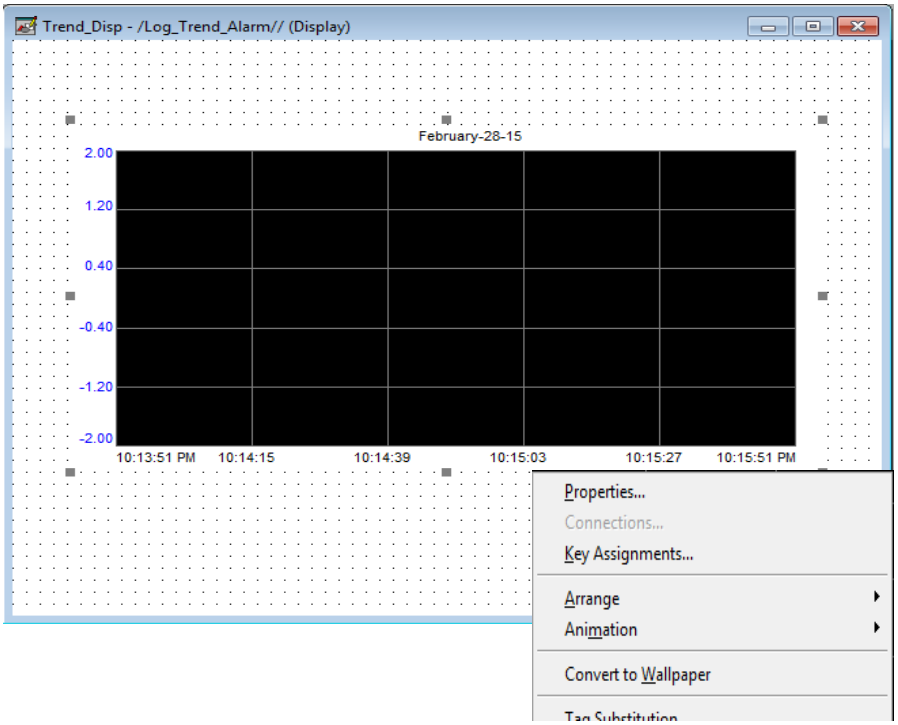
.

Trends

A trend is a visual representation, or chart, of current or historical tag values. A trend provides an operator with a way to track plant activity as it is happening.

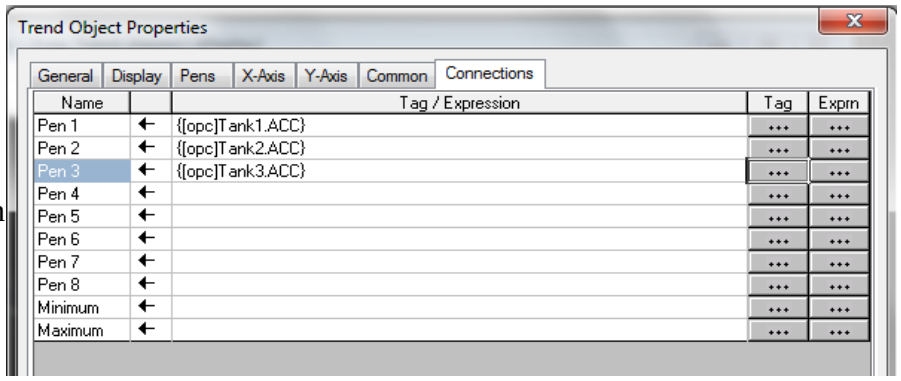
In the object toolbar drag and drop a trend in to the Display

Right click the Trend and select Properties

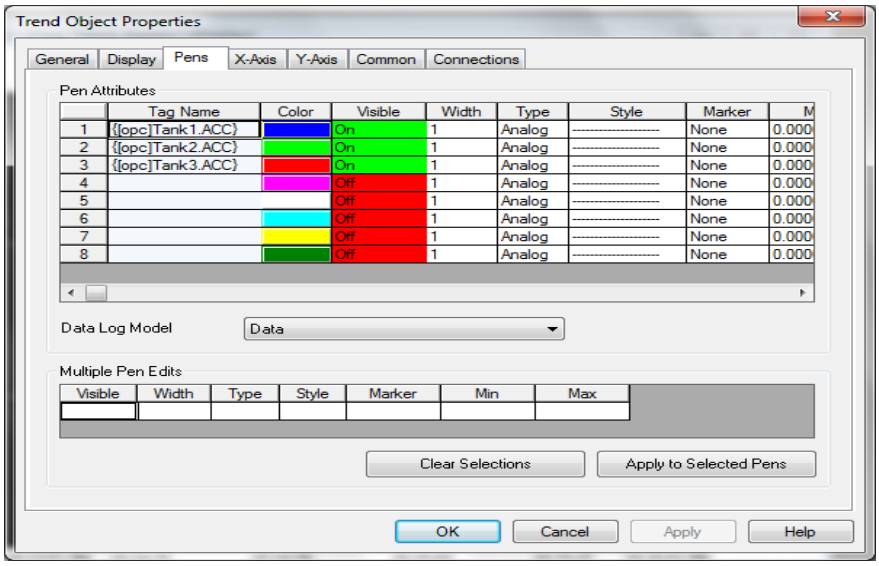


In the Connections tab select Tags You want to have their graph.

If you do not choose a Data log Model For these Tags, the graphs will be drawn but the data will be instantaneous and the logged data will not be shown.

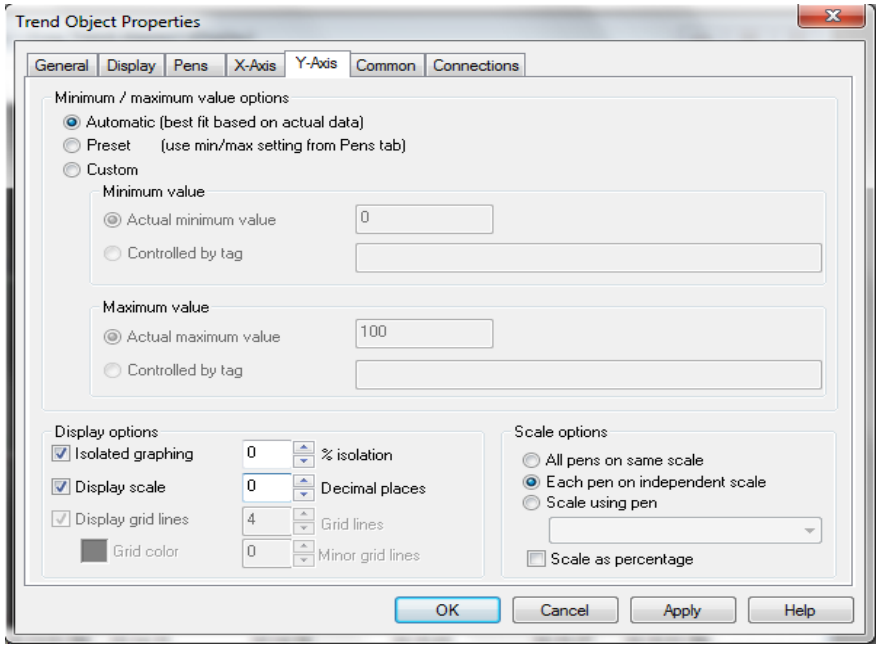


In the Pens tab, select Data Log Model you created in the previous step From the drop down box.

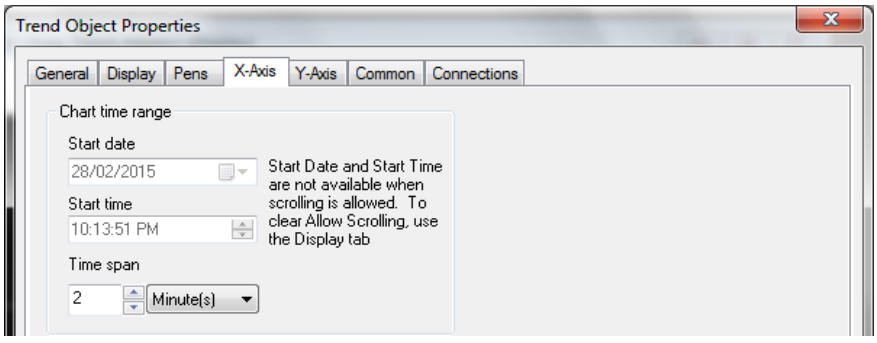


In the Y – Axis tab under the "Minimum/maximum value options", **Select** Automatic.

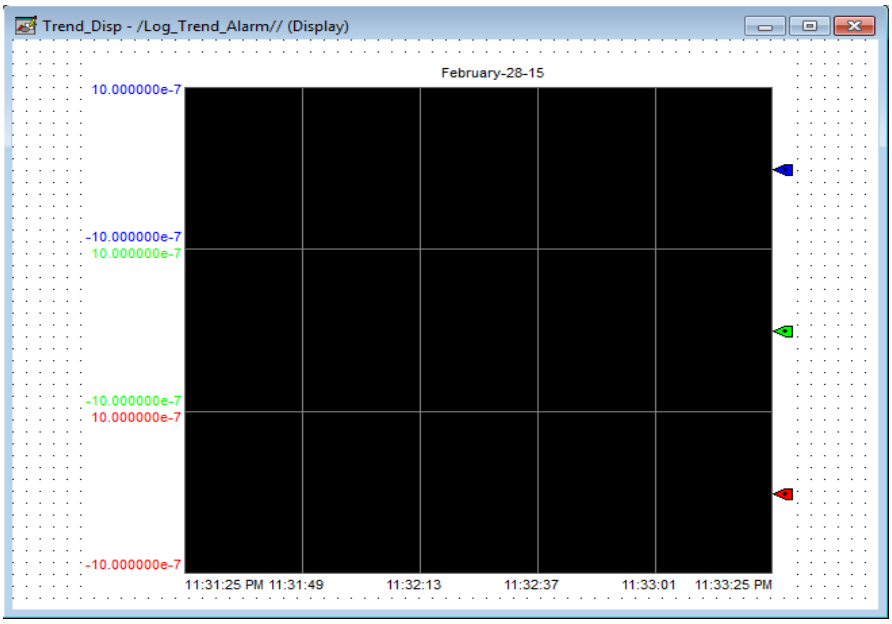
In the Display options **Select** Isolate graphing.



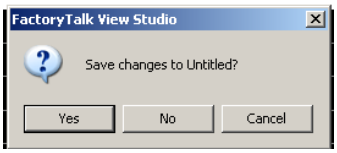
In the X-Axis tab, under the "Chart timeRange". Set the time spans to 2 Minutes.

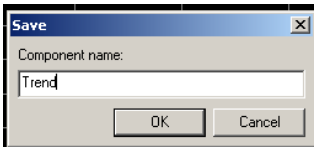


Test the display.

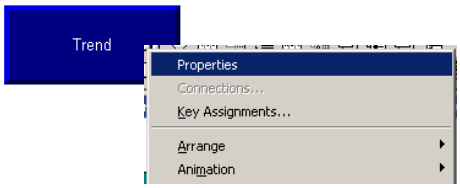


Press the close icon in the top right hand Corner. Save the change and Name the Component "Trend_Disp"

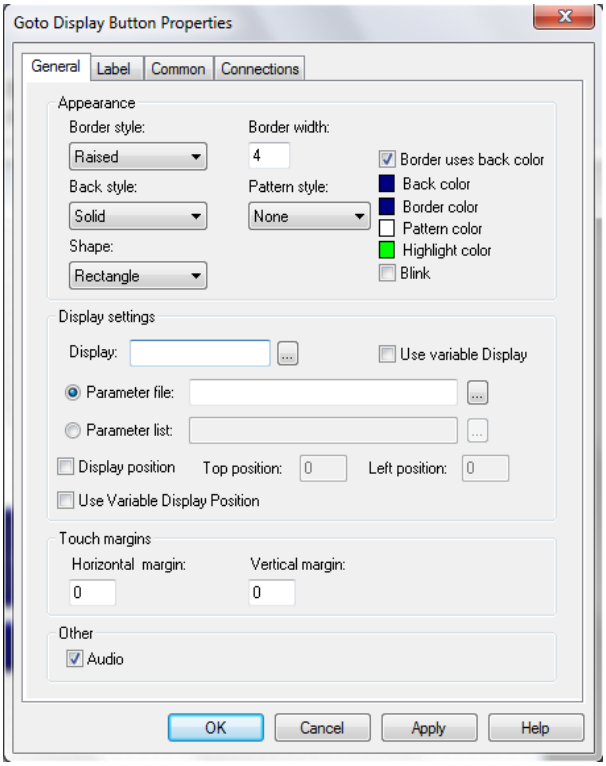




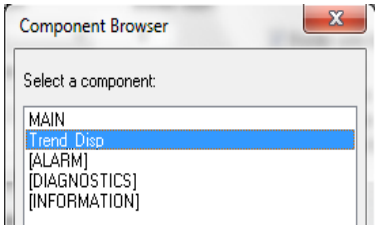
Go to main display **Right** Click the "Trend" Goto Display Push Button and **Select** Properties .



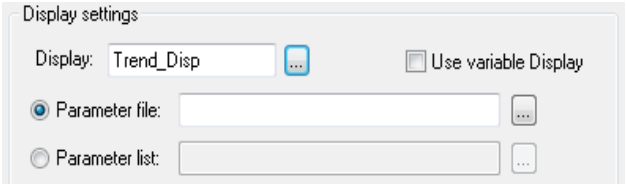
In the general tab, under the Display setting Section, Press the display ellipse. [...]



Select the "Trend_Disp" Display from the Component browser.



The Display Setting should look like the following:

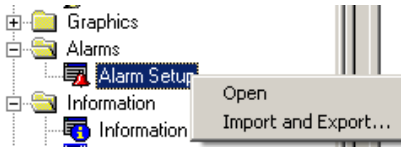


Press OK.

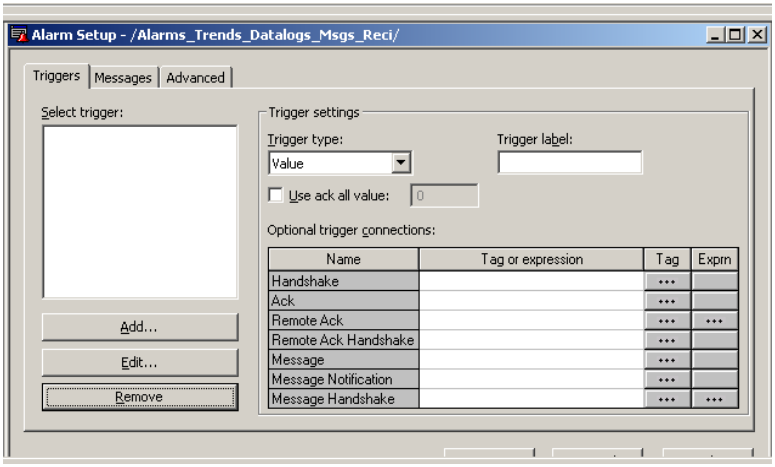
Alarm Setup

An alarm occurs when something goes wrong or is about to go wrong. Alarms can signal that a device or process has ceased operating within acceptable, predefined limits, and can indicate breakdown, wear, or process malfunctions. Alarms are also used to indicate the approach of a dangerous condition. Alarms are an important part of most plant control applications because an operator must know the instant something goes wrong. It is often equally important to have a record of the alarm and whether the alarm was acknowledged.

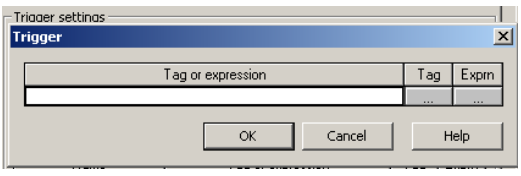
From the Project explore Expand the **Alarms Folder** and **Right Click** and **Press Open** or **double click** on **Alarms Setup** Icon.



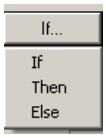
On the Trigger Tab, **Press** the **Add** button.



On **Trigger** screen press the grey **Ellipse** ... underneath the tag column

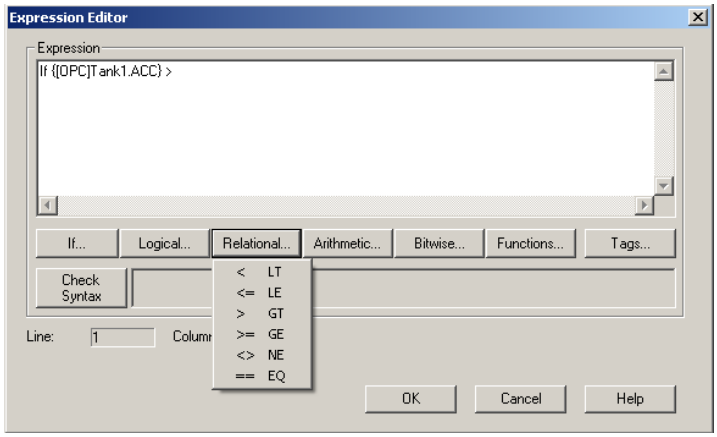


From the "**If**" menu select if, from the tag browser.

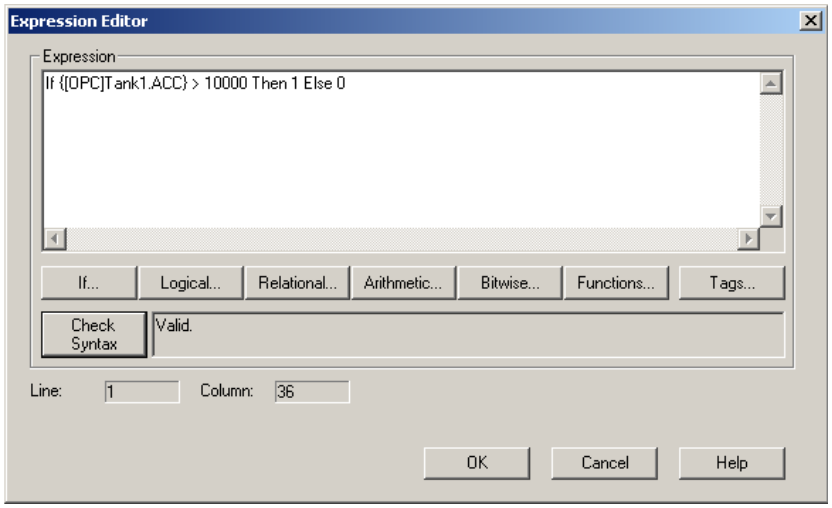


then **Press** the **Tags** button and **select** the **Tank1.ACC** value

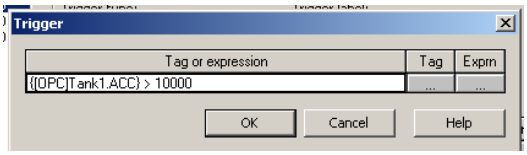
Then press the **Relational** button and Select the **"> GT"** symbol.



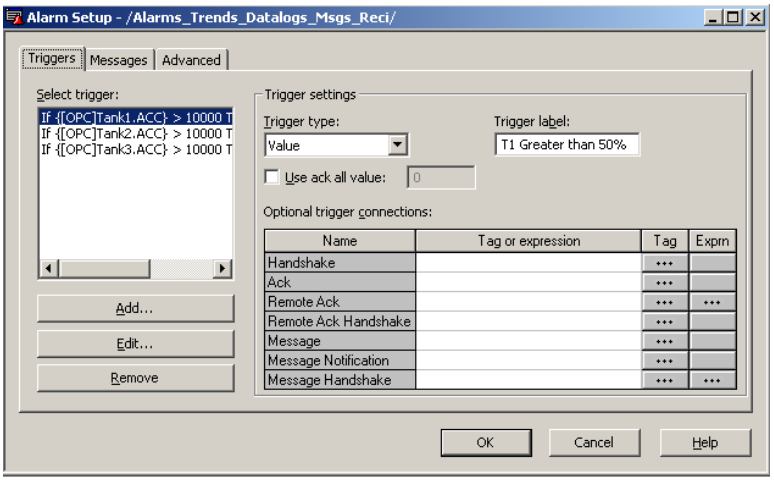
Finish the expression by using "If Menu", so that it looks Like the following and **Press Ok**.



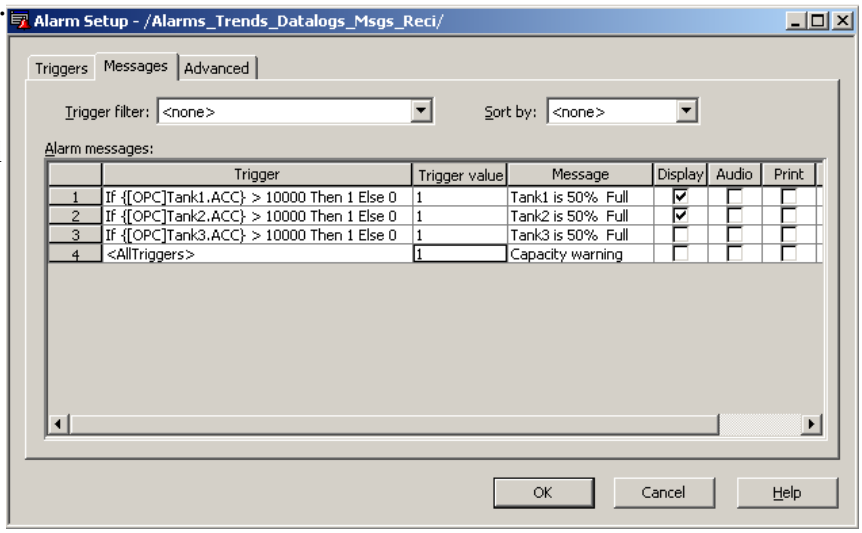
The expression that was built will show in the Trigger screen. Press ok



From the trigger type drop down Box. Select value and in the Trigger label box give the trigger A title. Follow the same procedure for Tank2 and 3 .ACC values

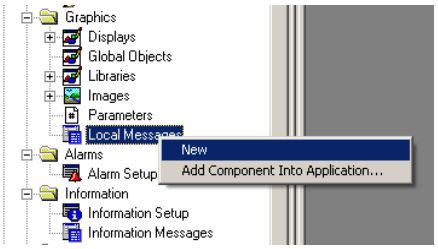


ON the message tab, In the Alarm Message Section, under the trigger column select the Individual triggers for Tank 1,2,and 3. Then enter 1 under the trigger value column and enter" Tank is 50% full " under the message column. Under trigger column, select all triggers and make the message "Capacity warning

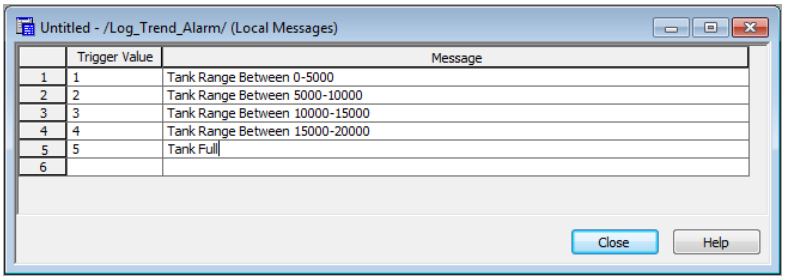


Messaging

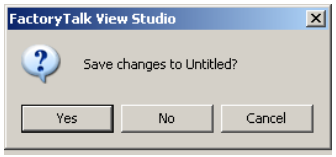
In the graphic folder, **Right Click** the local message icon and **Select** New.



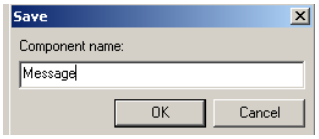
Set the trigger Value and give the appropriate message description . Press close.



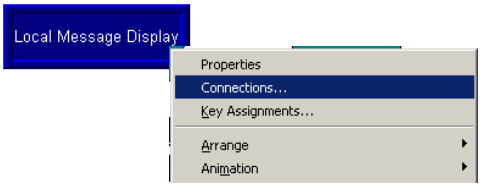
It will prompt you to save the changes. **Press** Yes.



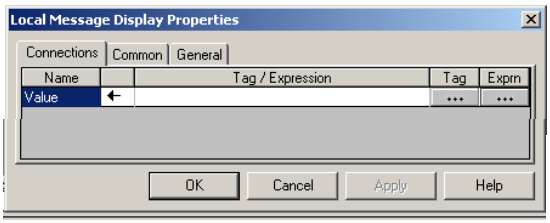
Give the new message component an appropriate name



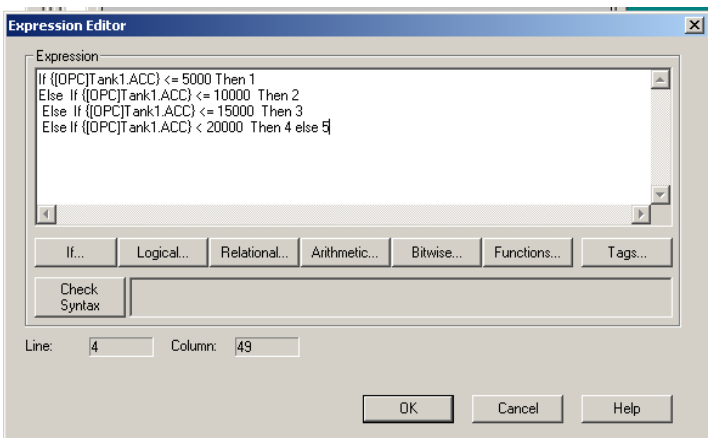
In the Main screen, **Right Click** the "local message Display" and **Select** Connection.



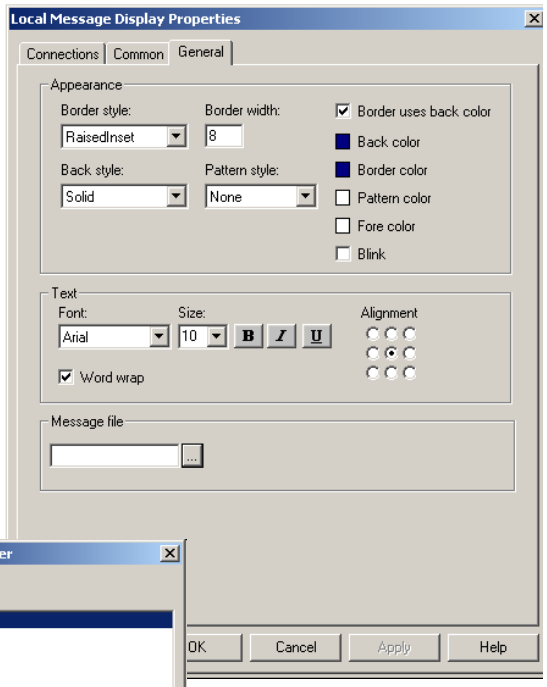
In the Connection tab Press the Exprn elispe.



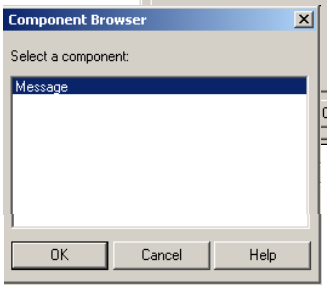
Enter the following Statements in the Expression Editor:
If {[OPC]Tank1.ACC} <= 5000 Then 1
Else If {[OPC]Tank1.ACC} <= 10000 Then 2
Else If {[OPC]Tank1.ACC} <= 15000 Then 3
Else If {[OPC]Tank1.ACC} < 20000 Then 4 else 5



Go to the general tab, and in the message file section
Press the elispse...

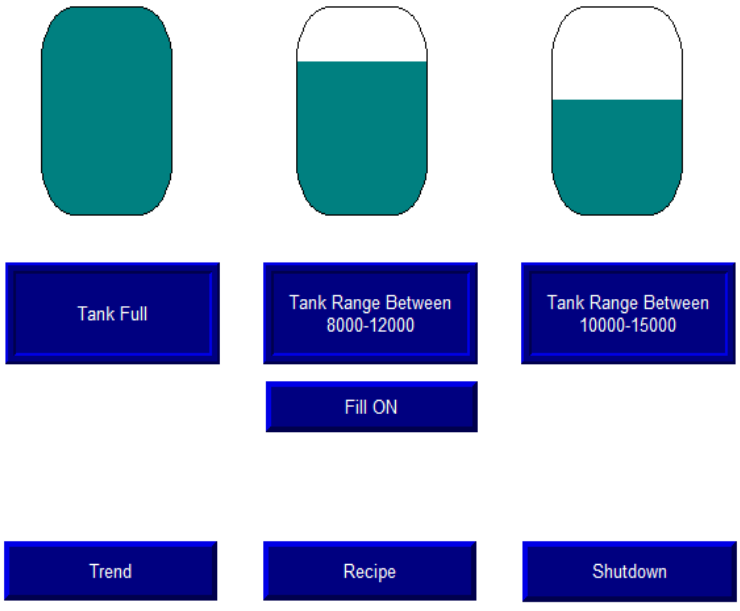


From the component browser,
Select the local Message component.



Create two other messages and
Fill the message and trigger values
According to other Tanks
Do the previous procedure for other
Tanks and change the values in the
expression according to the Timer's
preset values

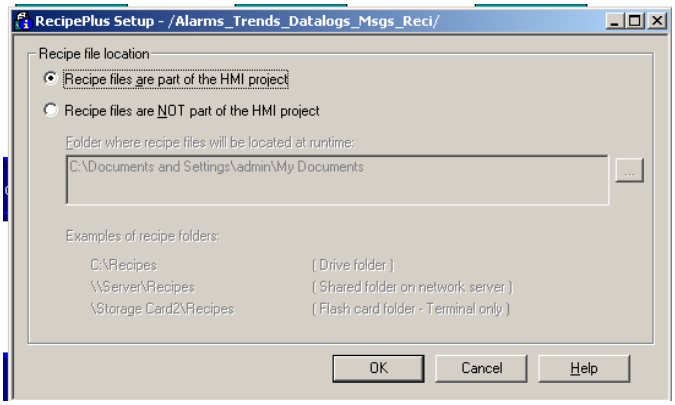
Test the Main Screen Display.



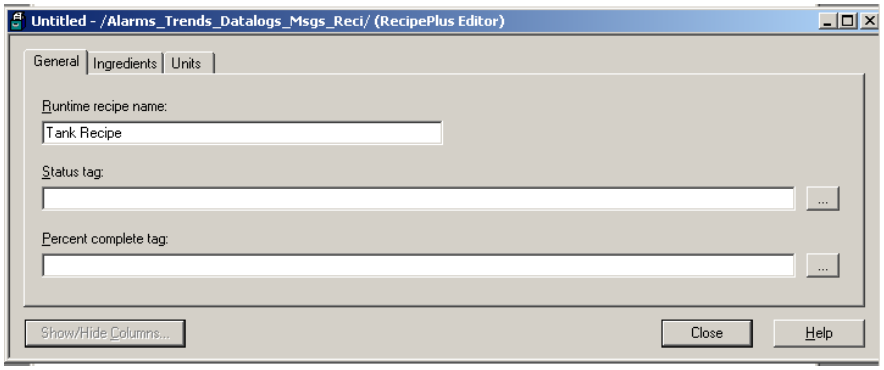
Recipes

Expand (+) the RecipePlus folder .
Double click on the PecipePlus Setup Icon.

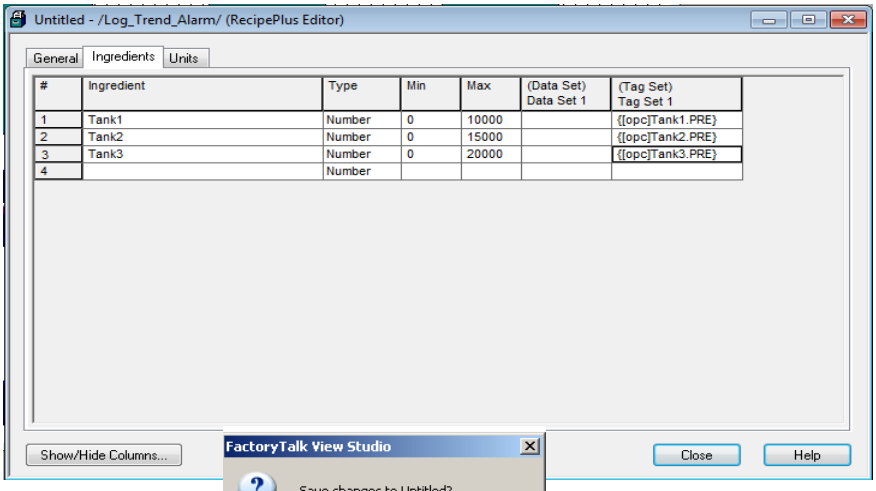
Make sure the “recipe files are part of the HMI projecct “ is **Selected**.



Right Click the RecipePlus Editor and **Select** new. Under the general tab, give the recipe an approriate name.

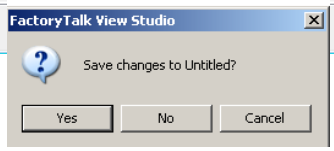


In the ingredients tab, list the ingredient and their min /max value. Under the Tag Set **Right Click** and **Select** Tag Browser and choose the tag needed to be substitiutted in the recipe

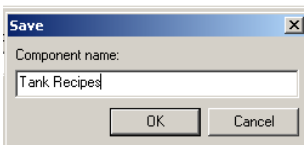


Click Close.

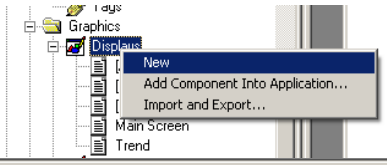
Click yes to save changes.



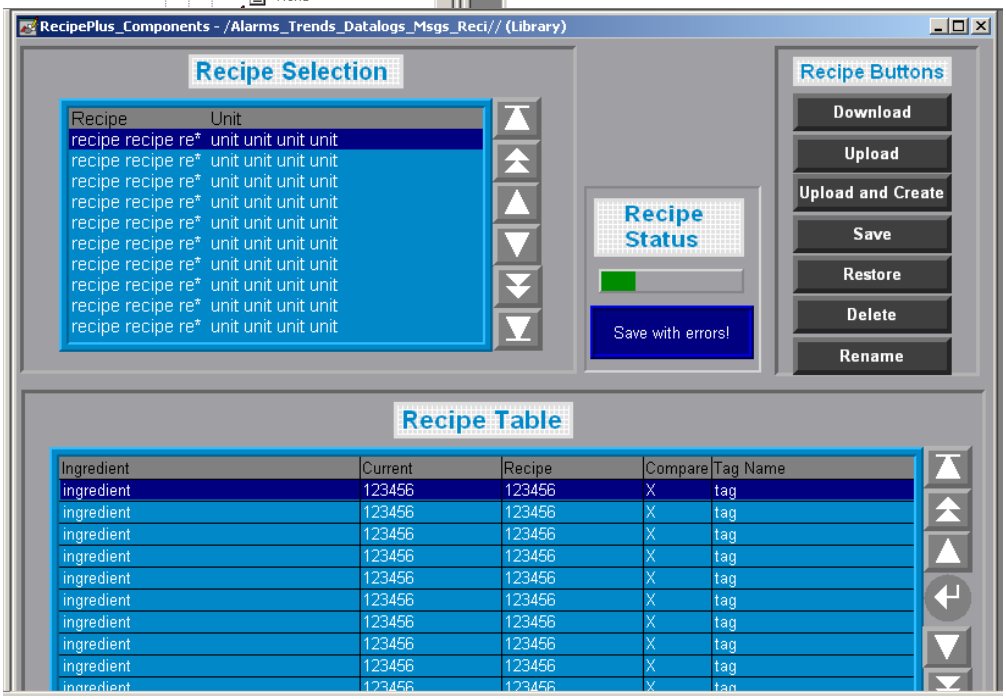
Give the recipes component an approriate name and Press ok



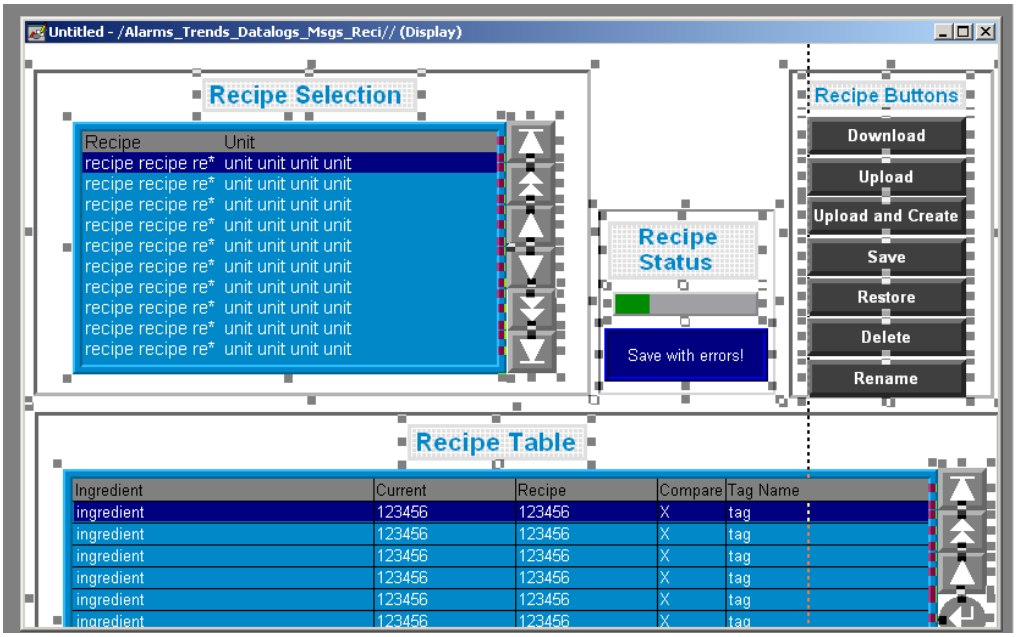
Create a new display by **Right Clicking** the display Icon and **Selecting** New.



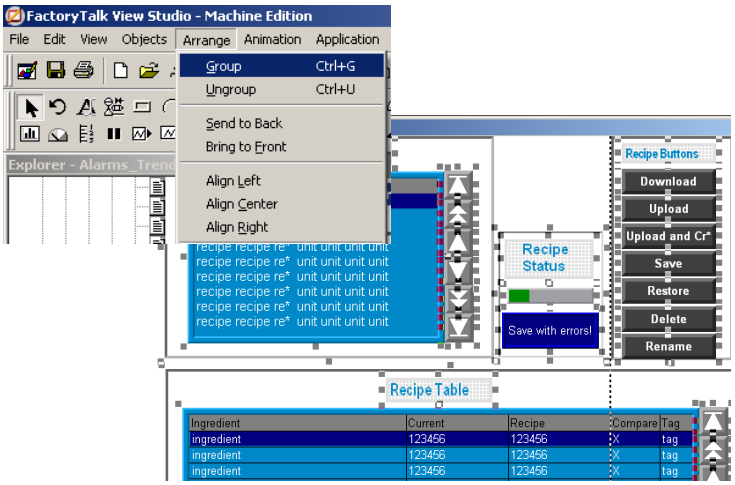
Expand (+) the Library Icon and **Double Click** on the RecipePlus component.



Click on the RecipePlus component screen and press Crtl-A (to select All) and drag and drop the items to the new display.

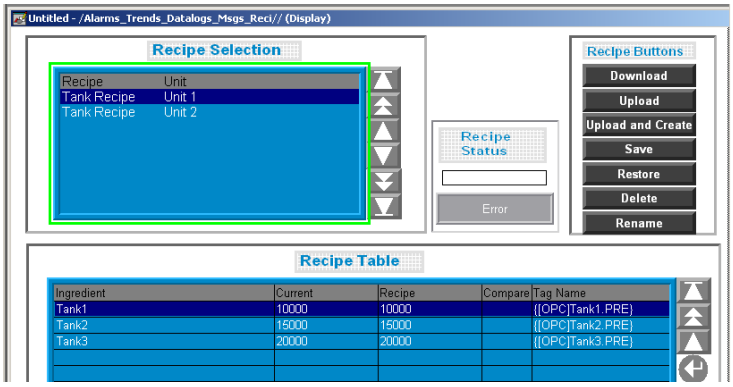


In the main toolbar click on arrange and **Select** group.



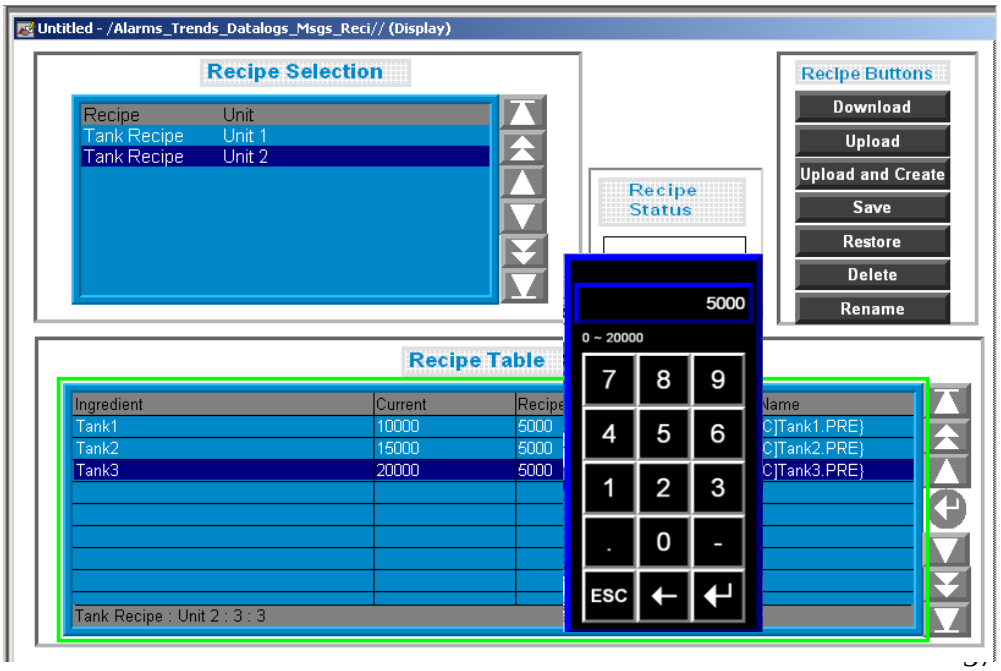
Test Recipe Display:

1. **Select** "Unit 1" and Upload and then Save.



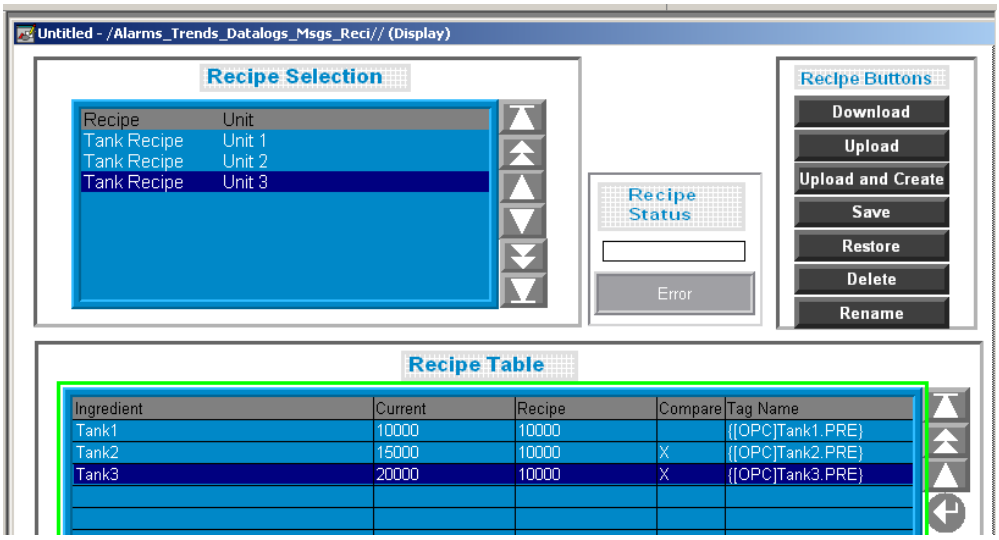
2. Upload and Create "Unit 2". In the recipe table press the enter symbol to set a recipe value The values are follows:

Tank 1 -> 5000
Tank 2 -> 5000
Tank 3 -> 5000



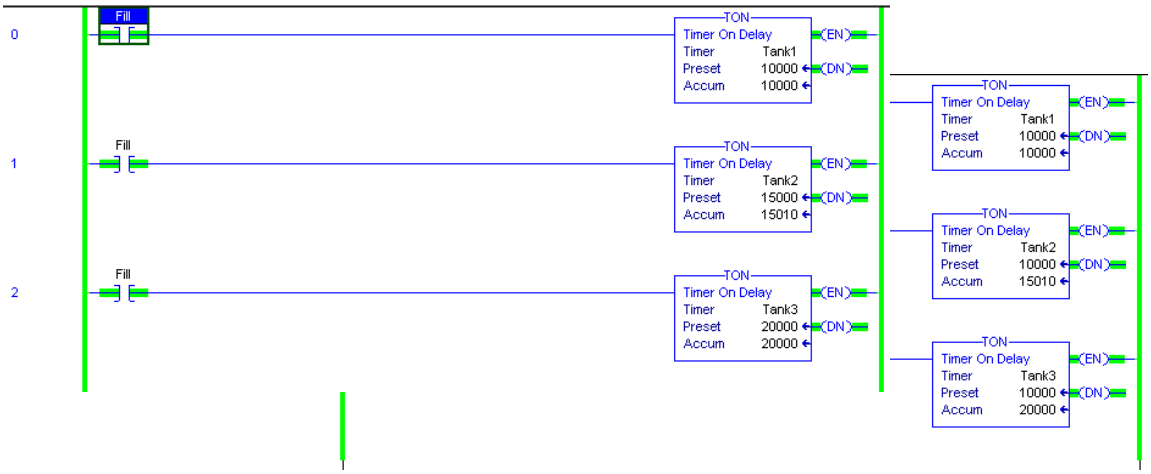
3. Follow the same steps for unit 3 as for unit 2. The values should be:

Tank 1 -> 10000
Tank 2 -> 10000
Tank 3 -> 10000

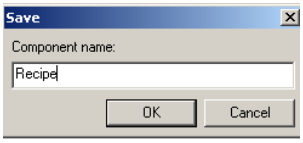


4. **Select** the different recipes and download

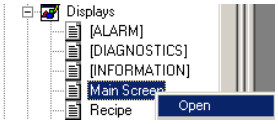
Verify that the values changed in the controller



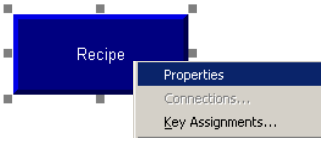
Close the recipe Screen and save the component with the appropriate Name



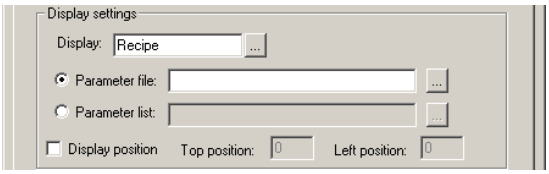
Goto the Main Screen



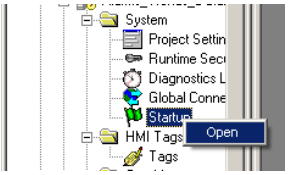
Right click the Goto display Button for the recipe screen and **Select** Properties.



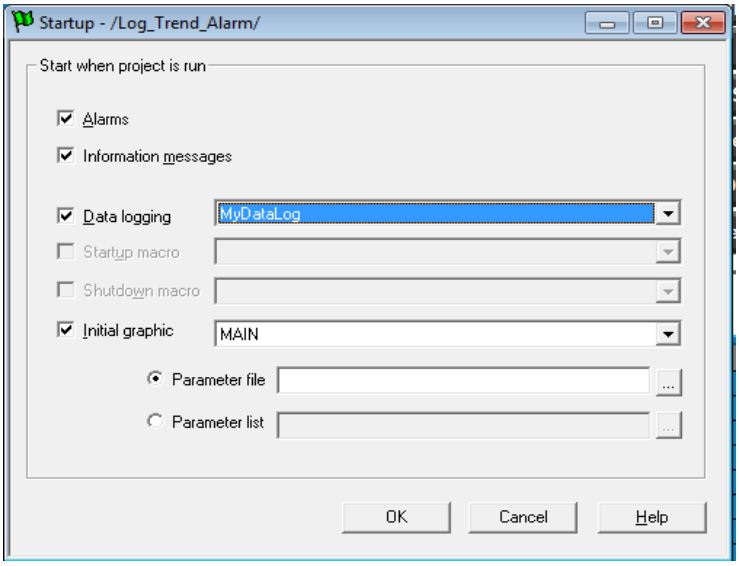
In the general Tab, under the Display setting Section, Press the display ellipse. **Select** the Recipe Display from the Component browser



Expand (+) the system folder and **Right Click** the Start up folder and **Select** Open



Select the Alarms, Information messages, Data logging and Initial graphic Check Box. In drop down list for Data logging **Select** DataLog
and in the Initial graphic select Main Screen



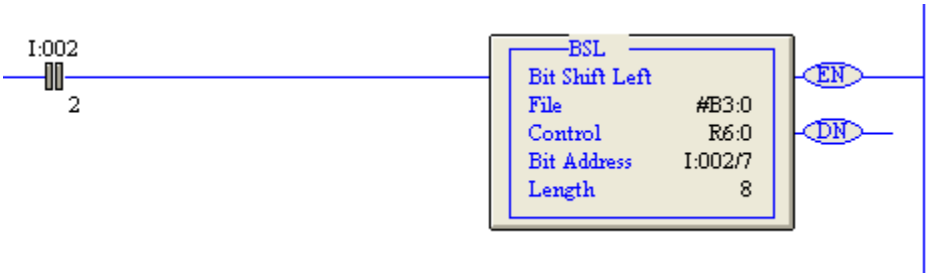
BSL – BSR - BTD

The following instructions are going to be covered in this section:

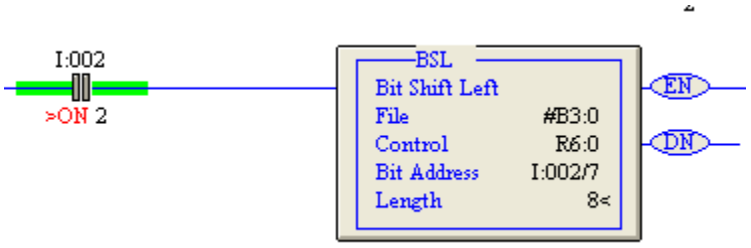
BSL (Bit Shift Left)
BSR (Bit Shift Right)
BTD (Bit Field Distributor)

The BSL and BSR instructions are part of the “File Shift Instruction” family. The other instructions from that family are the FIFO and LIFO instructions; we will handle those instructions later in this course.

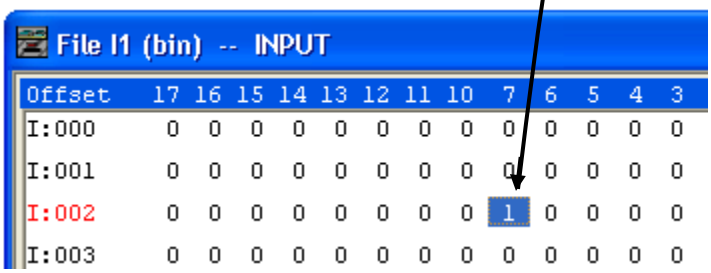
The Bit Shift Left instruction can be used to shift or index 1 bit position to the left on every false to true transition of the rung. When a BSL instruction is used, it will shift all bits in the file to the left on every false to true transition of the rung. The length in the instruction determines the amount of bits that is used in the array. The length does not have to be the same length as a word or multiple words. It can be a part of a word. If you make the length for example 24, it will use 16 bits of the first word and another 8 from the next word. If you use part of a word you can no longer use the rest of that word for something else, because the bits are shifted in that word anyway. The maximum length in a PLC 5 is 16,000.



In the above example, the BSL is used to move a bit in B3:0 to the left every time I:002/02 goes from false to true. The “Bit Address” I:002/07 is called the load bit. It will take what is in that address (1 or 0) and shift that to the left in #B3:0. The last bit exits the array and is send into the Unload bit **(R6:0/UL).**



Lets take a look and see what happens when this instruction is used. In the instruction to the right I:002/02 goes true.The ***load bit I:002/07*** is **1**. The ***first bit*** will now appear in B3:0. If load bit I:002/07 is reset to **0** and we trigger I:002/02 again we will have the result that you can see in step 2. If the load bit does not change state and in step 3 we trigger I:002/02 we shift another **0** in the array. In step 4 the load bit was changed to **1** again and you see the result in B3:0. A **1** was shifted into the array.



B3:0 (Step 1) **0000 0000 0000 0001**

B3:0 (Step 2) **0000 0000 0000 0010**

B3:0 (Step 3) **0000 0000 0000 0100**

B3:0 (Step 4) **0000 0000 0000 1001**

If we keep loading the array from the example above we could have something like the picture below. The array is loaded with data and in step 8 we have a **1** in the last position of the array. In step 9 that bit is shifted out of the array and into the unload bit of the control element **R6:0** (R6:0/UL),as you can see in the picture below. As you can see in step 9, the bit that is shifted into the Unload bit of the control word is still in B3:0. For this reason you cannot use the remaining part of the word that is used for the array.

B3:0 (Step 6) **0000 0000 0010 0111**

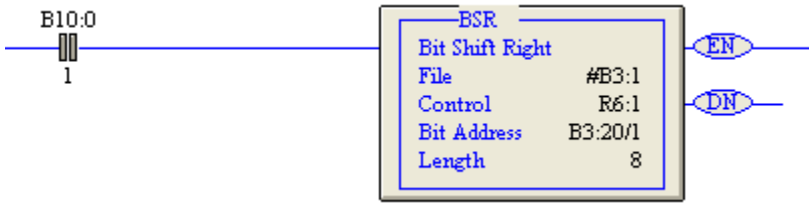
B3:0 (Step 7) **0000 0000 0100 1110**

B3:0 (Step 8) **0000 0000 10011100**

B3:0 (Step 9) **0000 0001 00111001**

Note: Instead of using the R6:0/UL bit, it would be possible to use B3:0/8. However, for best practices it is recommended to use the unload bit of the control word. This is in case you modify the length of the array (R6:0.LEN) at runtime with an instruction (MOVE).

The Bit Shift Right instruction (BSR) works the same as the BSL. The difference is the direction the bits shift and where the start of the array is. If you look for example at the picture to the right, you see a BSR that also uses B3:1. The Control address is R6:1, and the load bit is B3:20/1. The length is 8. This means that when this instruction is triggered, the load bit will be inserted in the array starting at B3:1/7. The last bit in the array is B3:1/0. The bit stored at B3:1/0 will exit the array into the Unload bit (R6:1).



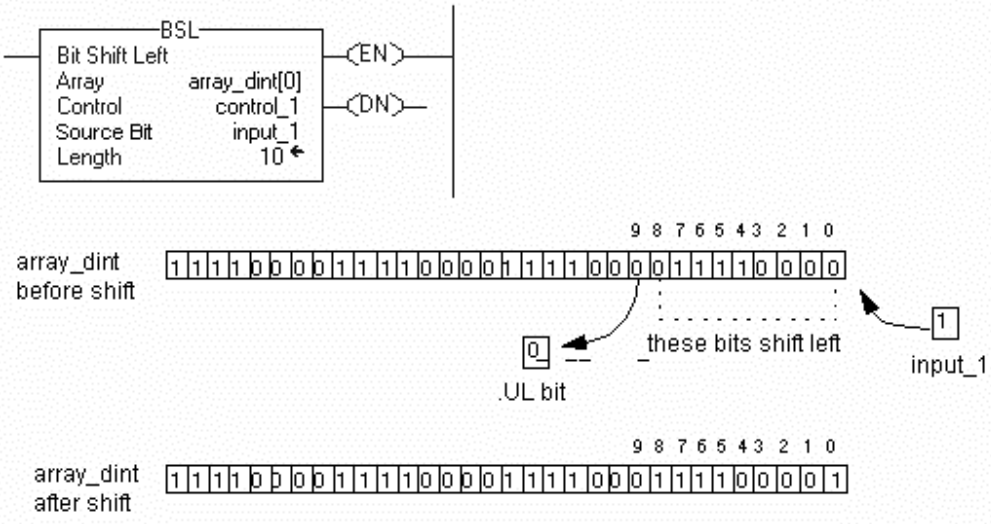
Example of BSL in Studio 5000

Example

Ladder Diagram

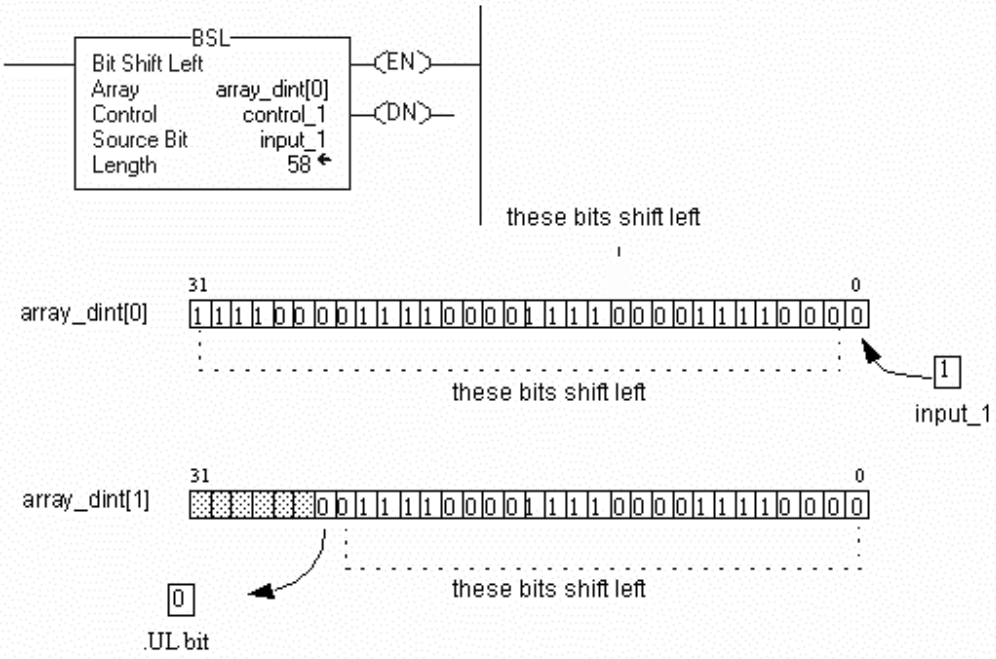
Example 1

When enabled, the BSL instruction starts at bit 0 in *array_dint[0]*. The instruction unloads *array_dint[0].9* into the .UL bit, shifts the remaining bits, and loads *input_1* into *array_dint[0].0*. The remaining bits (10-31) are invalid.



Example 2

When enabled, the BSL instruction starts at bit 0 in *array_dint[0]*. The instruction unloads *array_dint[1].25* into the .UL bit, shifts the remaining bits, and loads *input_1* into *array_dint[0].0*. The remaining bits (31-26 in *array_dint[1]*) are invalid.



Assignment 3

Lab: For this lab we are going to use the start and stop button and all the pilot lights. With the start and stop button, we start and stop the machine at any given time. The pilots lights will be used to simulate motors. When the start button is pressed the first motor will start up. After 10 seconds the second motor will start. 10 seconds after the second motor the third motor will start up. This procedure will continue until all motors are on or the stop button is pressed. This can easily be done with timers, the problem is when we need to turn the motors off again. It needs to be done in stages so if all motors are on and the stop button is pressed, motor 8 is turned on and 10 seconds later motor 7 and so on until all motors are off. This could be an installation where pressure has to be built up gradually. And shut down again in stages. The best way would be to use a BSL to load a **1** into an array and a BSR to shift a **0** into the array.

Requirements:

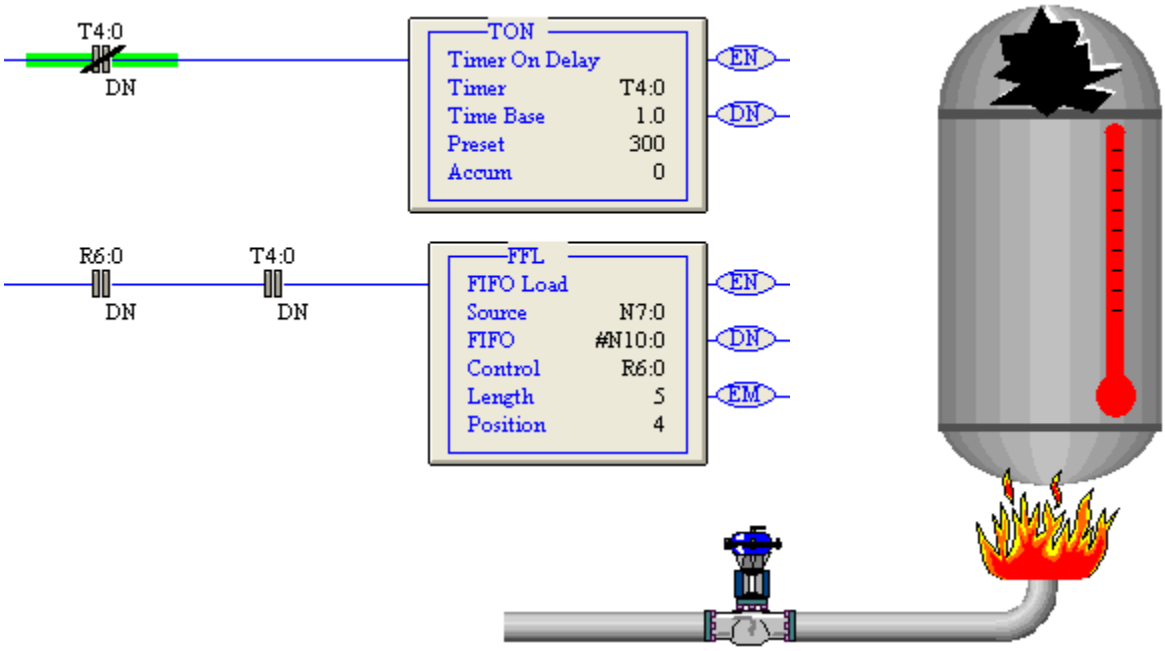
1. Use Studio 5000
2. Create Start, Stop, and 8 motors in FactoryTalk View Studio Machine Edition

Optional: Use 1734 Remote IO for input and output. Use only 4 motors.

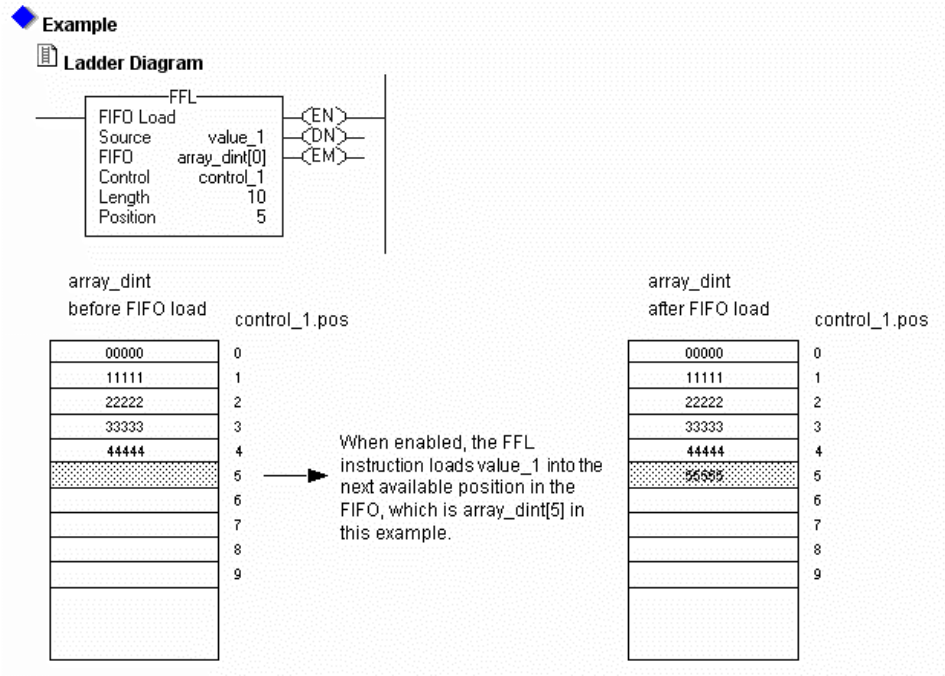
FIFO and LIFO

Use the FFL instruction with the FFU instruction to store and retrieve data in a first-in/first-out order. When used in pairs, the FFL and FFU instructions establish an asynchronous shift register. Typically, the Source and the FIFO are the same data type. When enabled, the FFL instruction loads the Source value into the position in the FIFO identified by the .POS value. The instruction loads one value each time the instruction is enabled, until the FIFO is full. You must test and confirm that the instruction doesn't change data that you don't want it to change. The FFL instruction operates on contiguous memory. In some cases, the instruction loads data past the array into other members of the tag. This happens if the length is too big and the tag is a user-defined data type.

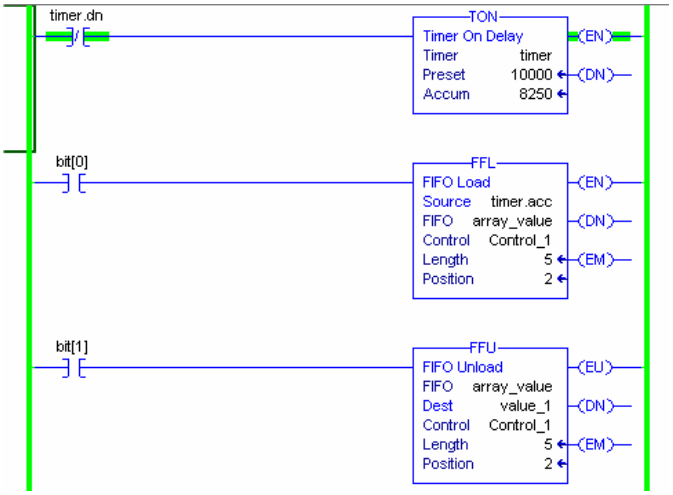
The FIFO and LIFO instructions could be used for many different reasons. In recipes they could be used to load values from different recipe files into registers to be used in the program, like set points for temperatures and product quantities.
The FFL instructions could be used to log temperatures from a process into a stack and later reviewed to make sure the process was within limits. In the picture below a temperature reading is done every 5 minutes, stored in the stack for 25 minutes. Or the FFU could be used to send temperature values to a machine in predetermined time intervals to heat something in stages.



FFL



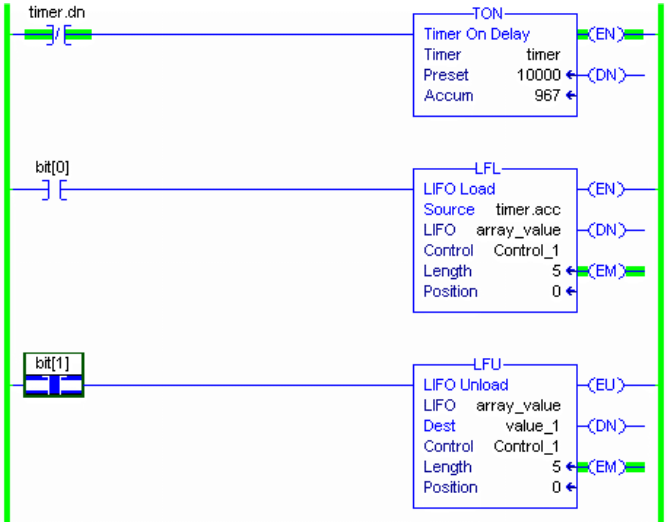
The instructions work as follows. When **bit[0]** goes from false to true, the value stored in **timer.acc** will move to **array_valve** in the stack. On the next false to true transition of **bit[0]** it will take again the value from **timer.acc** and move it to the next word in the file(stack). It will do this until the stack is full. The DN bit in **control_1** is true at this point. If the input instruction is triggered again when the stack is already full, nothing will happen. You have to unload at least one value before you can use the FFL again. If you trigger **bit[1]** five times the stack is completely empty again and the **control_1**/EM bit will be set. Make sure that you use the same control word when using these instructions as a pair.



When the stack is full, control_1/DN bit is set and triggering bit[0] will have no effect. Values have to be unloaded before you can load again.

You could unload only 1 value and then load again. In the below example the stack gets completely unloaded. When the stack is empty the control_1/EM bit is set. Use the FFU instructions to unload the stack.

LFL



The next pair of instructions is for the LIFO, the **LFL** and **LFU**. The LFL instructions works similar to the FFL instruction. It will take the value from the source and loads it in the stack. The difference is when we start unloading the stack. The last value loaded will be unloaded first to the destination. In the example below only the actions of the **LFU** are displayed because the **LFL** works the same as the **FFL**.

Description

Use coordinated pairs of FIFO (FFL/FFU) instructions to move data into and out of a data block (known as a "stack") on a First In First Out basis. These instructions may be programmed on separate rungs for separate control.

Operation

With each false-to-true rung transition, the FFL will move the value in the source into the next available word in the stack. With each false-to-true rung transition, the FFU will move the value in the starting word of the stack into the destination (and shift down all remaining values).

Entering Parameters

Source
Enter the word address storing the value to be loaded into the stack.

FIFO
Enter the address of 1st element of an array with proper length for both FFL/FFU instructions. This array will make the FIFO . This will be the starting or base location of the stack.

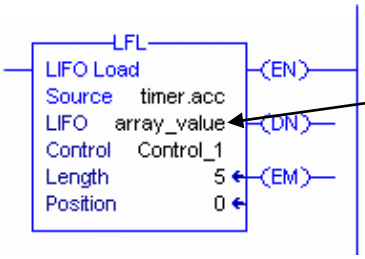
Control
Enter a control (R) file address. Enter the same control element for both instructions in the FFL/FFU matched pair. Do not duplicate a control element used in any other instruction.

Length
Enter a decimal number defining the size of the stack.

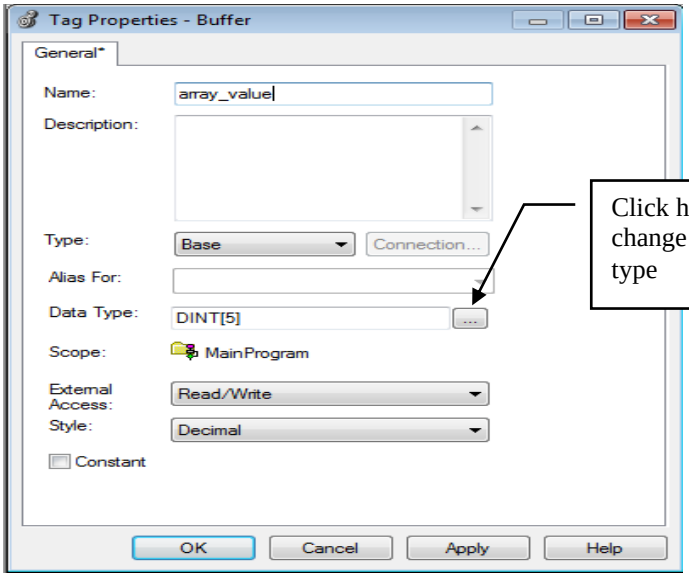
Position
Generally, enter a zero. (This points to the next available word location in the stack for the FFL to load a value).

Caution: Except when pairing stack instructions, do not use the same control address in any other instruction. Unexpected operation could result with possible equipment damage and/or personal injury.

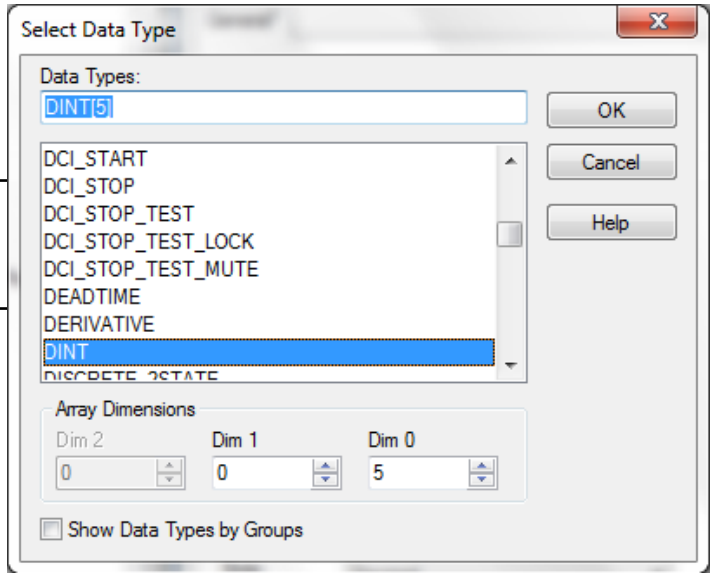
CHANGE DIMENSION



Right click on the **array_value** tag and go to Edit "array value tag" Properties

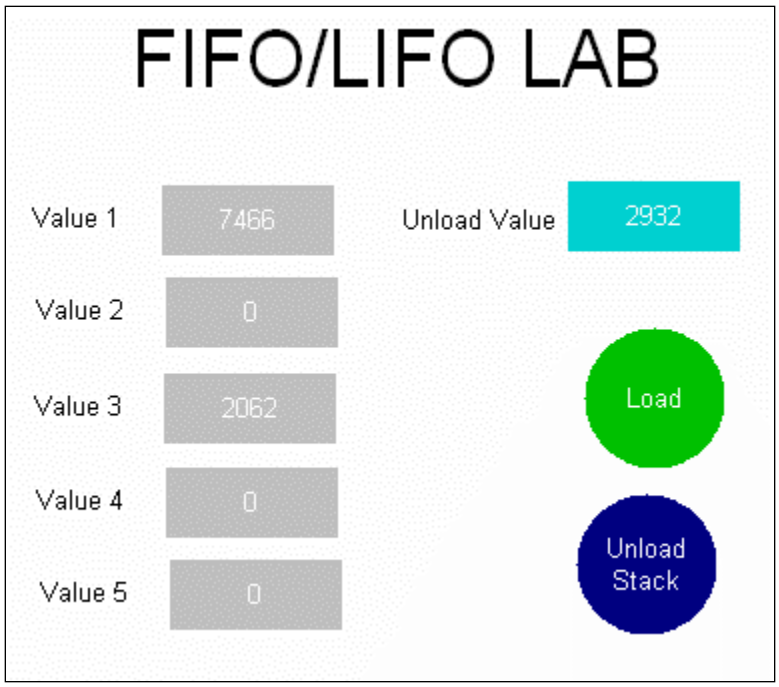


Click here to change data type



Assignment 4

Lab:
Make an application that loads a value from the 1734 analog input card every 10 seconds and load it into a stack. Create a numeric display in FactoryTalk View Studio to show values in the stack. When the stack is full, use a button in FactoryTalk View Studio to unload the values manually to the 1734 analog output card. The HMI should look similar to the picture below.

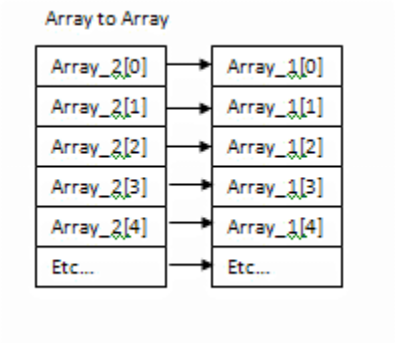


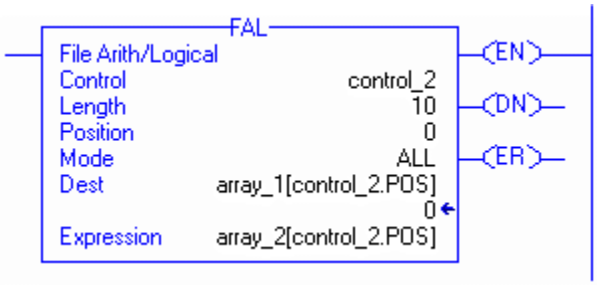
File Arithmetic and Logic (FAL)

The FAL instruction can be configured and used in many different ways. You can copy, perform logic, arithmetic and function operations on data that is stored in the data table. It works similar to a CPT instruction. The difference is that the FAL can work with multiple words.
For a more detailed description and explanation of the FAL instruction, refer to the help file in Studio 5000 and/or got to www.ab.com and download the "Instruction Set Reference" guide. You can find that on this site by selecting "manuals" and then control processors.

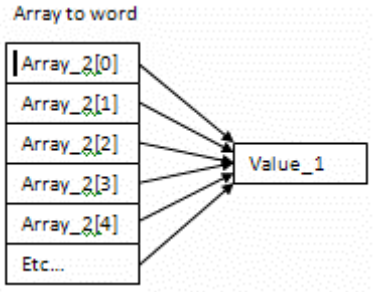
The FAL instruction can perform operations on a single word or a file. The copy operation for example can copy between files, from word to file or from file to word.

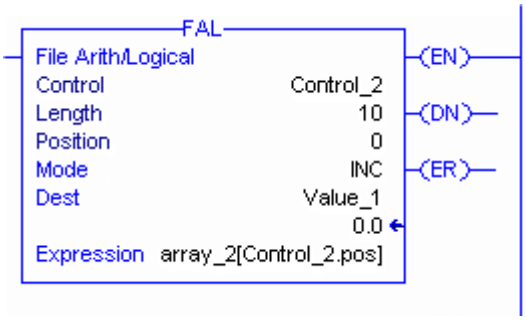
In the example you see here, an **Array to Array copy** will be performed. When the rung executes, 10 words from array_2 will be copied to array_1. All words are copied at once because the Mode is set to "ALL". You have the option to fill in "ALL" , "Incremental" or any decimal value.
For example, if you enter a value of 2 in the Mode entry box you will copy 2 words at a time on a false to true transition until all words are copied and the DN bit is set.





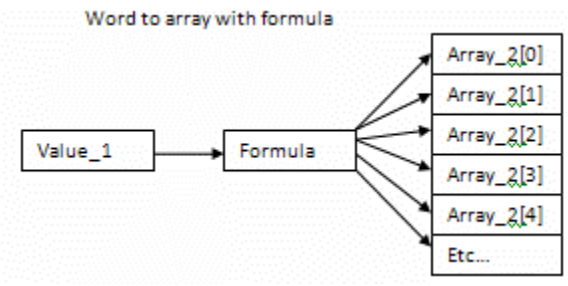
Next example will copy 1 word at a time from the file array_2 to a single word Value_1. The incremental mode is selected so on every false to true transition a word is copied until all 10 words are copied and the DN bit is set. If you trigger the instruction again when the DN bit is set it will just start at the beginning again. By now you have noticed that selecting between file or word in either expression or destination is done with the [ ] sign. Using the ALL mode is of no use here because you will only see the last word of the stack in the destination on a false to true transition of the rung.

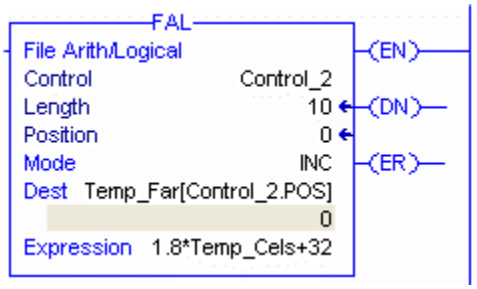




The example on the previous pages was about copying files only. Math can be performed also. In the beginning of the course we made a subroutine file to convert 2 or more Fahrenheit values to Celsius. The FAL instruction could be used to get the same result without a subroutine. Inorder to do that you select a destination array. In the

Expression you can enter your formula so each time the rung become true the value will be store in the array element.





LAB 1

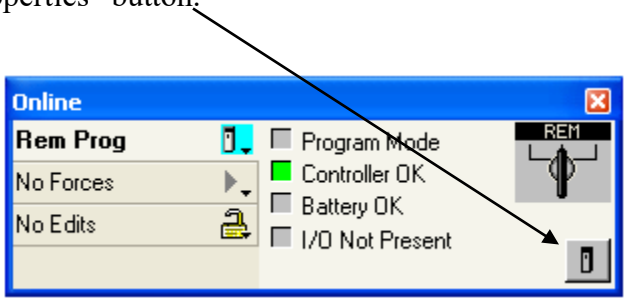
Use the application we made earlier for the FIFO. In RSView we loaded a stack with data. Modify this data by multiplying it by 2 and copy it to another stack. Show that stack also in RSView. Execute the instruction so that 2 words are send at a time.

LAB 2

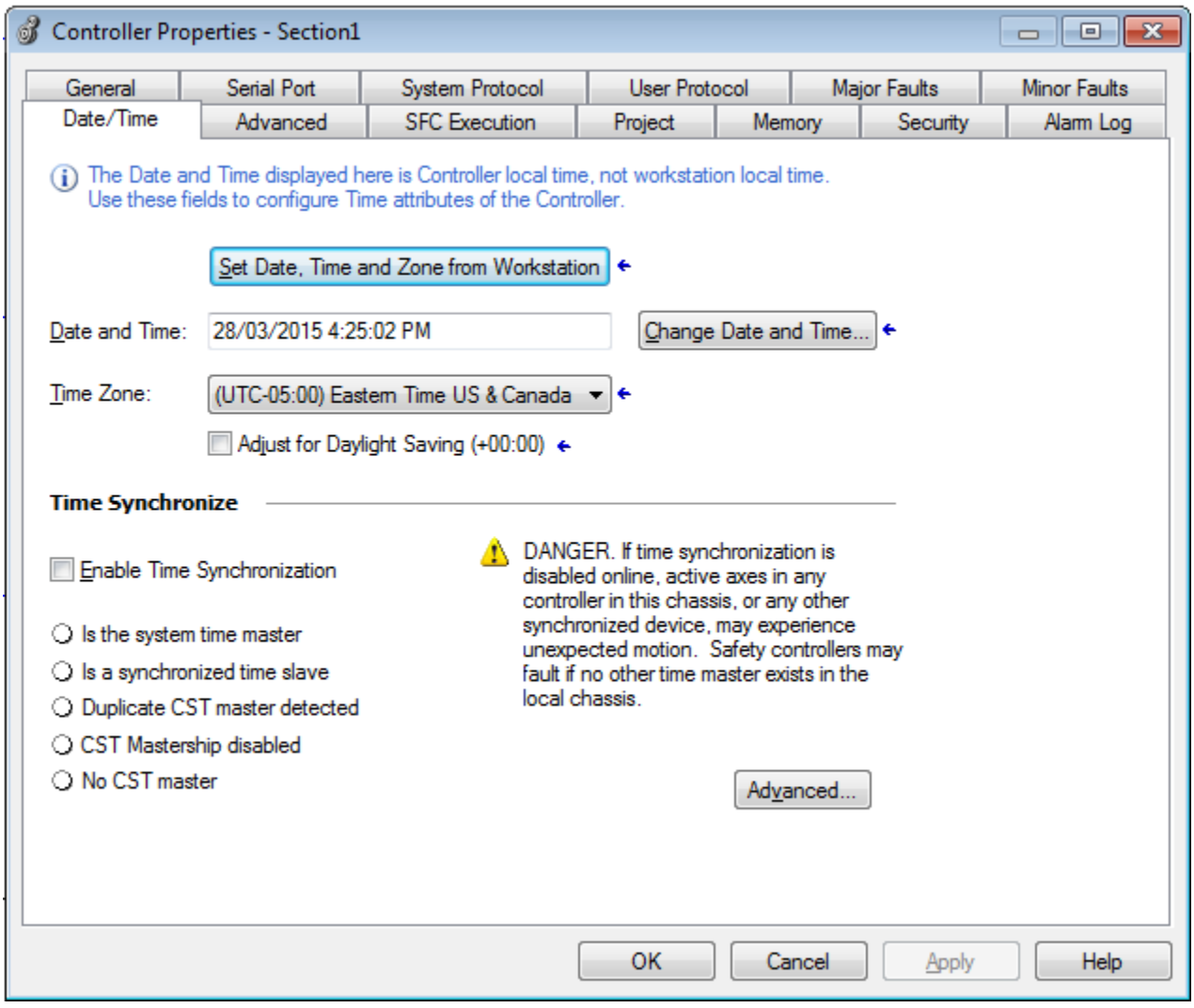
Take the value from your Analog channel 0 and your analog channel 1 and using the FAL instruction translate that into decimal and show the values in RSView.

In PLC5 and SLC you have a status file for status of the IO, internal Clock, Scan time or processor faults. All that information is also available in Logix but you need some instructions to retrieve this information.

In a logix controller you can for example set the time in the controller very easy using the “Controller Properties” button.



On the “Controller Properties” window you can select the “Date/Time” tab and just use the “Set Date, Time and Zone from Workstation” button to set the time to the same time as your computer.

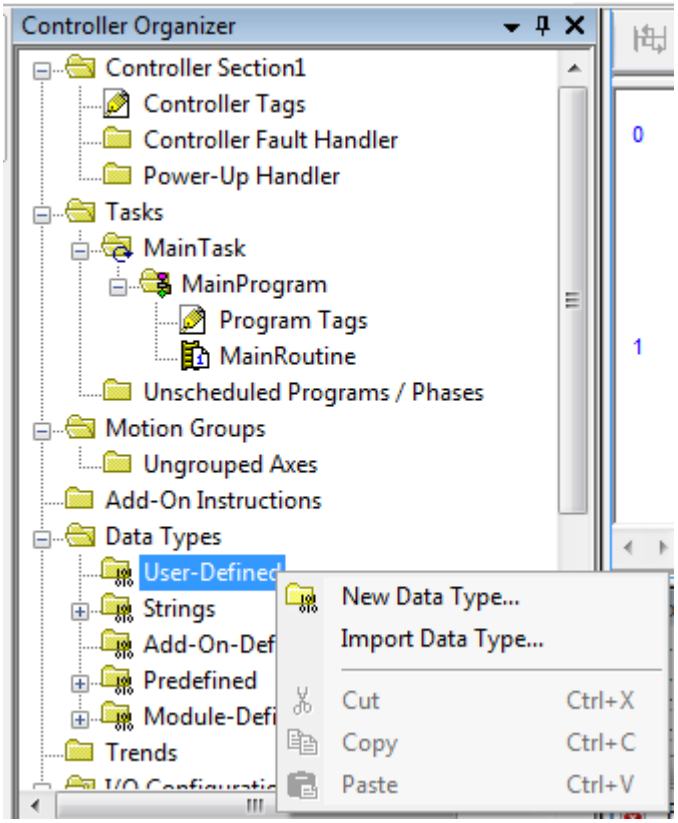


So it's easy to set the time and date in the PLC but if you want to use this data in your PLC program you need to access this information. In a PLC5 you address the time and date using the S2 file. For example S23 is used for the seconds and S22 for minutes.
In Logix you need to transfer this data from the system file into a tag before you can use it in your program. To do this you have 2 instruction available, the SSV and GSV instructions. The SSV(Set System Variable) is used to write into the system file and the GSV(Get System Value) is used to read from the system file. You have to be very careful writing to the system file as there are many items accessible that affect the processor in different ways.

In the following lab you are going to get the time and date from the system file and make it available to be used in a program.

LAB : GSV – System Time and Date

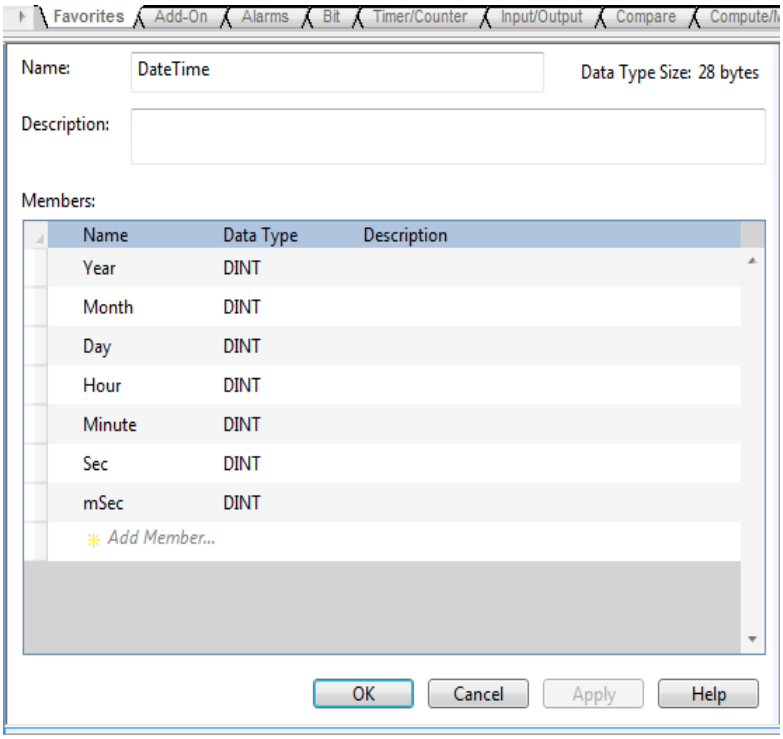
First you need to create a "User Defined Tag" to store the date and time from the controller.



In the Controller Organizer Right Click 

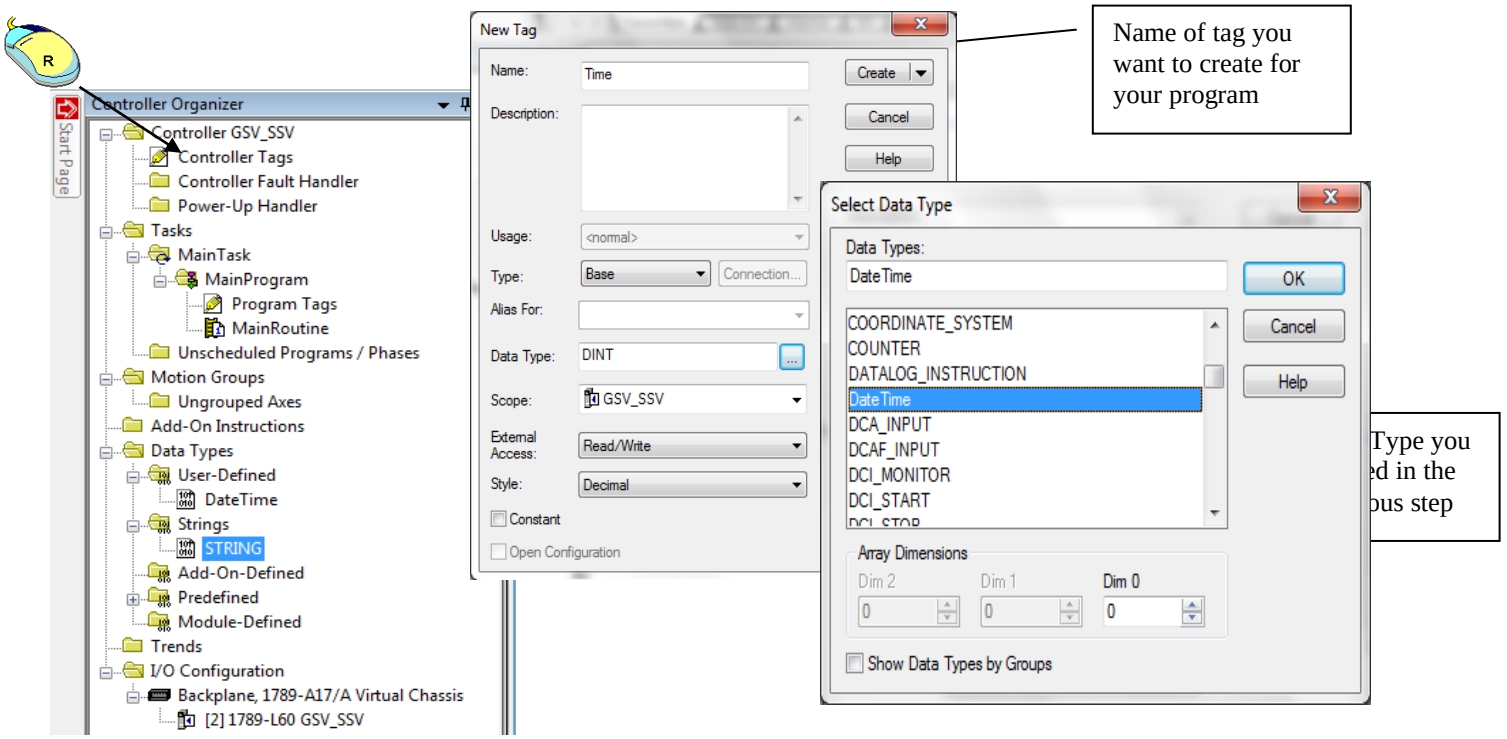
on the "User-Defined" folder and select "New Data Type"

In the “Data Type” window enter a name for your User Defined Tag, I used “DateTime” for this, and create the tags as you see in the picture below.



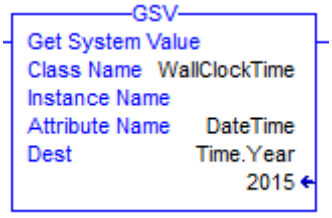
You have now created a UDT that can hold the date from the system time.

Next step is to create a tag based on the UDT we just created for our ladder program.

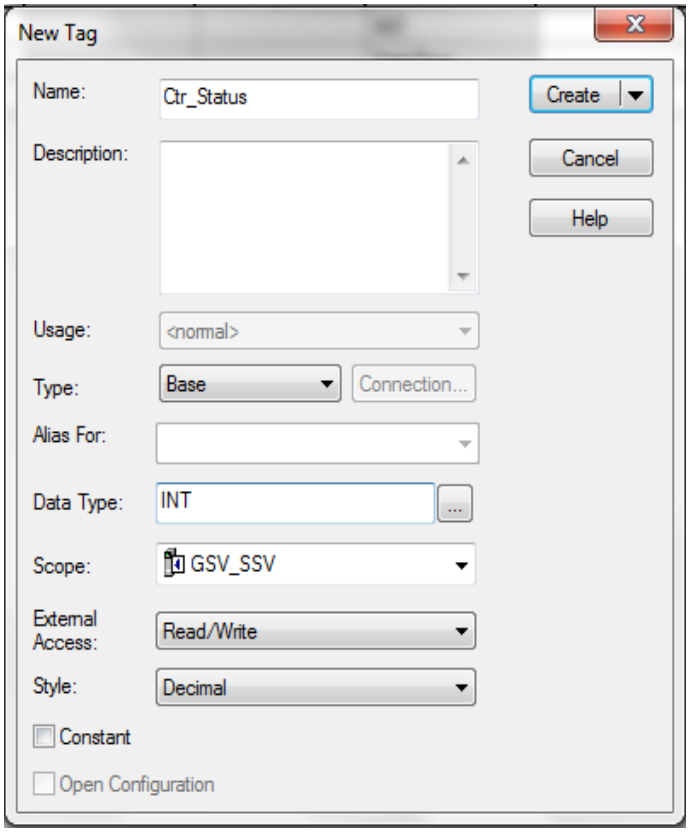


Now you just need to enter the ladder code to Get The System value and populate your “Time” tag with this data. The DateTime UDT is an array of 7 Dint’s, you need to address the first DINT in the array in your instruction as you can see below.

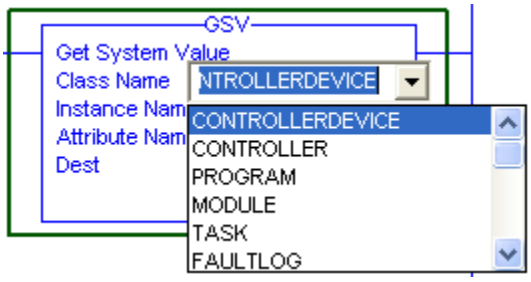
Now if you put the controller in Run state and double click Controller Tags and go to Monitor Tags section
You can see the current Date and Time of the controller

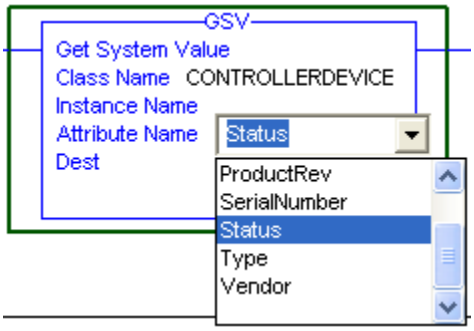


The next thing you can access from the System value is the processor status. This information comes from the “CONTROLLERDEVICE” object selectable from the “Class Name” drop down in the CSV instruction. First create a tag to store the data. This tag is an Integer as you see in the picture below.



Next enter a GSV instruction on a new rung and select the “ControllerDevice” object from the drop down box.





For the “Attribute” select status from the drop down box.

Select the tag you created earlier for the destination.

The 16 bits written to your status word “Ctr_Status” and the meaning of the bits are explained in the columns below. Bits 0-3 are reserved.

Bits 7-4	**Meaning**
0000	reserved
0001	flash update in progress
0010	reserved
0011	reserved
0100	flash is bad
0101	faulted
0110	run
0111	program

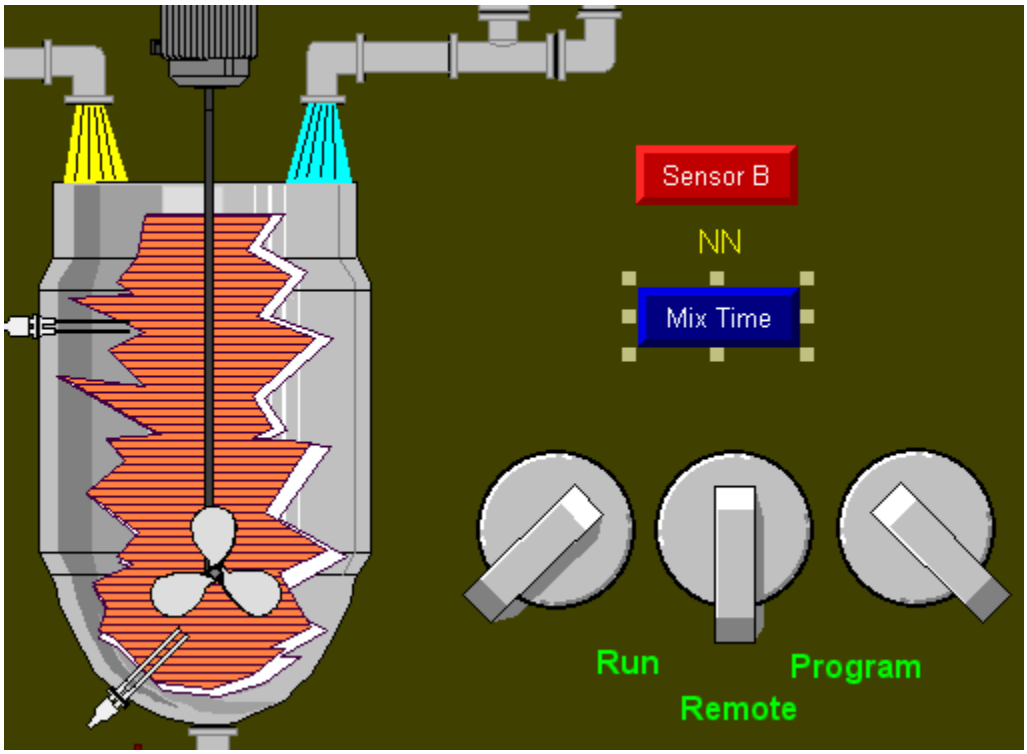
Bits 11-8	**Meaning**
0001	recoverable minor fault
0010	unrecoverable minor fault
0100	recoverable major fault
1000	unrecoverable major fault

Bits 13-12	**Meaning**
01	keyswitch in run
10	keyswitch in program
11	keyswitch in remote

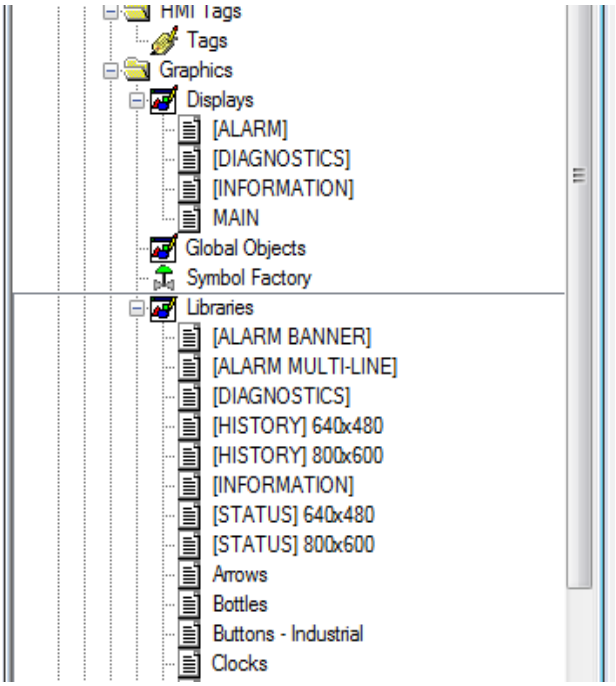
Bits 15-14	**Meaning**
01	controller is changing modes
10	debug mode if controller is in Run mode

Let’s use some of these bits and show how they could be used. Start RSView Studio and open one of the applications you created earlier or start a new one. I used the Batch application from level 1. Add the “GSV” instruction described above and add it to your existing code. Download it to your controller and open RSView Studio.

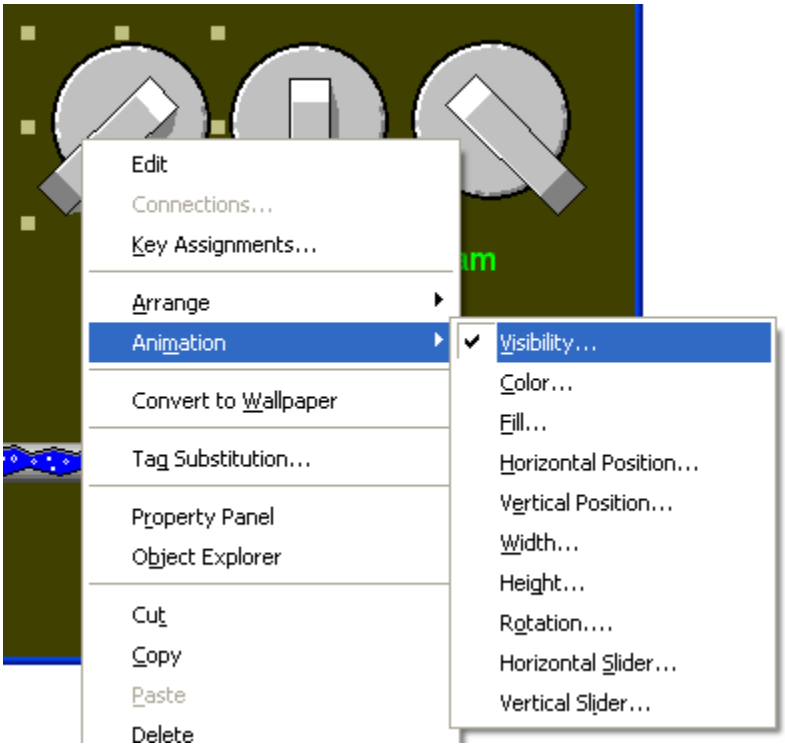
As you can see I used the “Batch” application but you can use whatever you have or make a new RSView application. If you do make a new one, make sure you have a topic created that points to you controller.



From the graphics library, add the 3 graphics that resemble a selector switch to your display. You can find them in the library under “Buttons – Industrial”. Make sure you do not save changes when you close the library screen again.



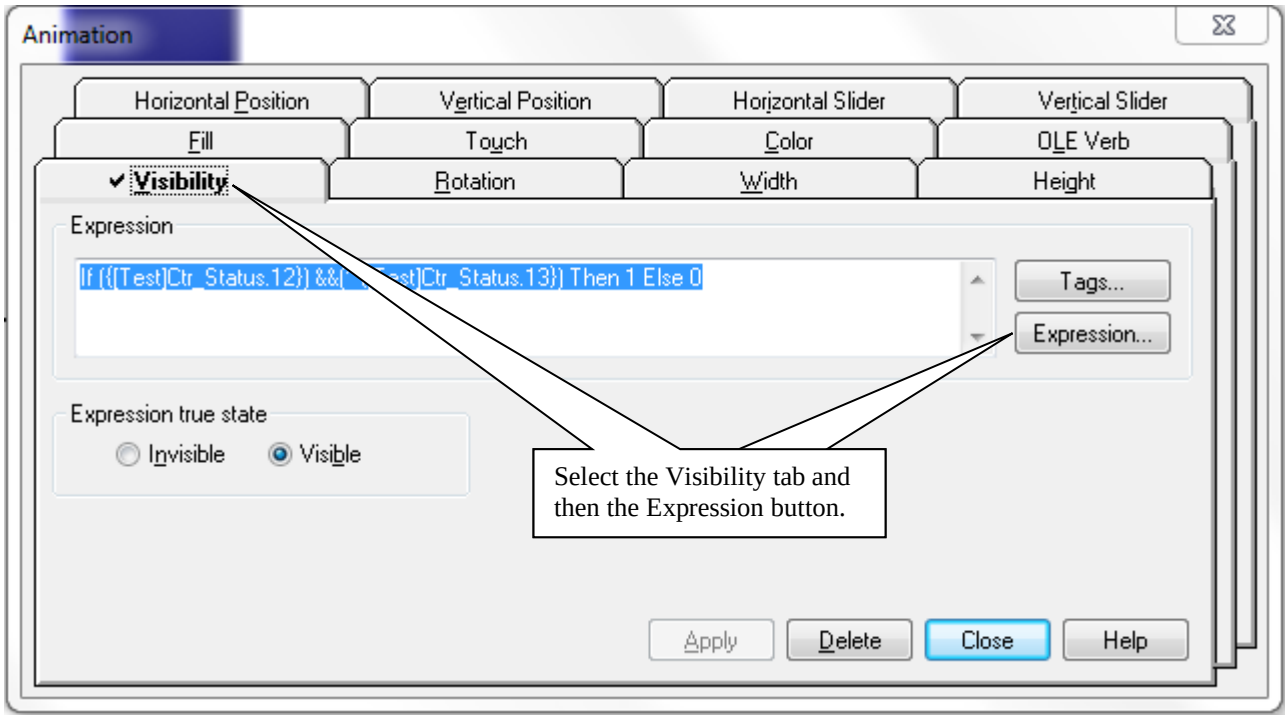
Right Click  on the first button and from the menu select “Animation” and then “Visibility”. We are going to show/hide each graphics depending on the bit pattern of bits 12 and 13 of the status word.



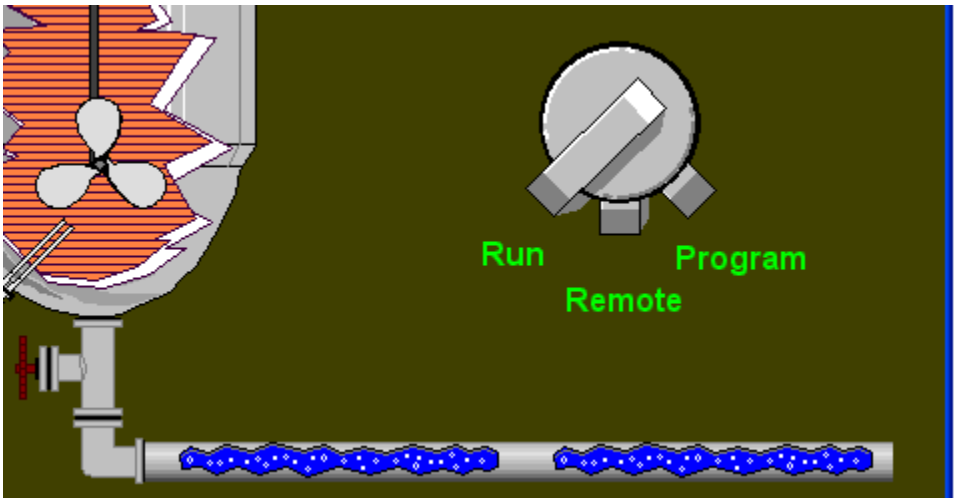
In the “Animation” window, click the “Expression” button and fill in the expression you see below.

If ({[Test]Ctr_Status.12}==1) &&({[Test]Ctr_Status.13}==0) Then 1 Else 0

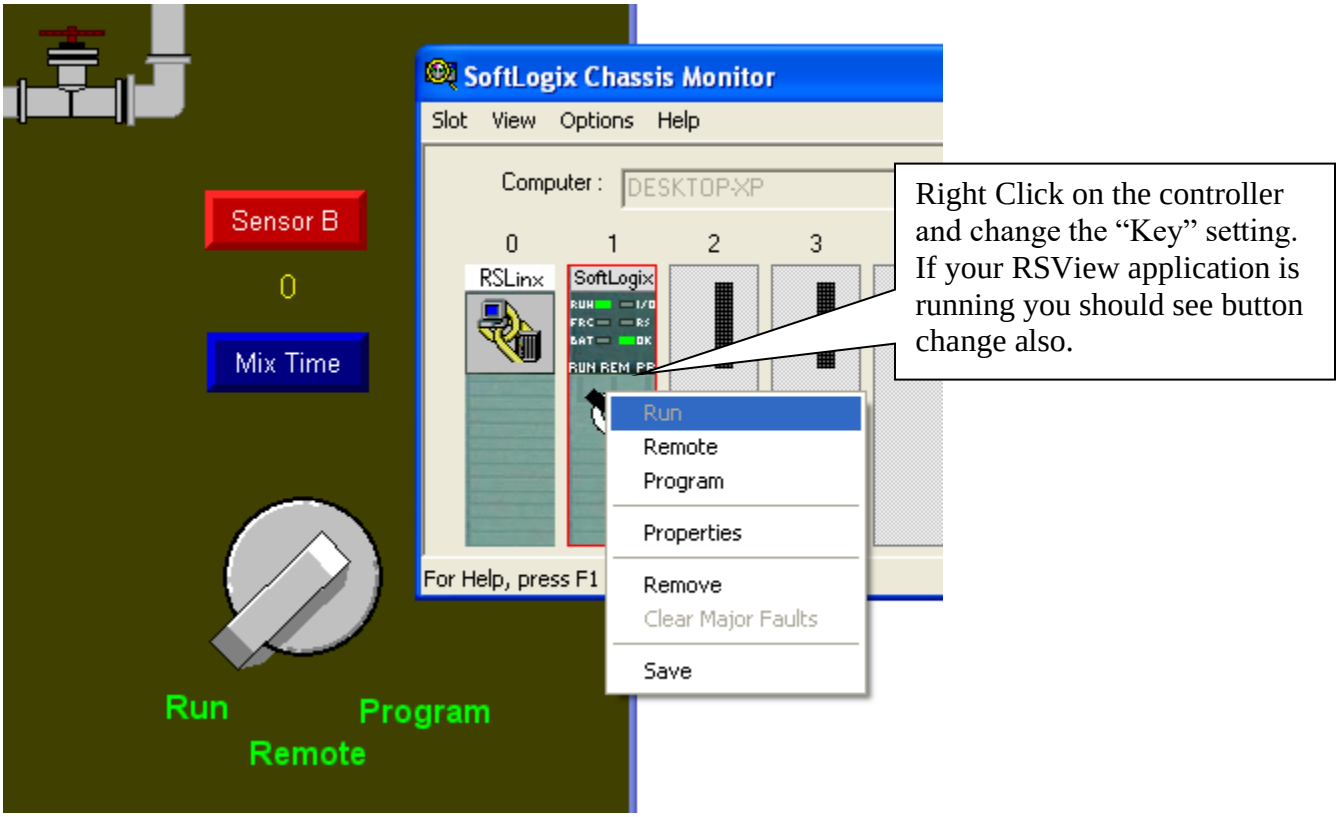
In this example you can see that the tag in the controller is called **“Ctr_status”** and the communication shortcut that points to the controller is **“Test”**. Make sure you change that to match your tag and shortcut.



The above status bits (12 = 1 and 13 = 0) resemble the key in the “Run” position. Do the same thing for the other 2 buttons and modify bits 12 and 13 using the list on the 2 pages back. At run time you don’t want to see the buttons jump from left to right so place the buttons all 3 on top of each other. Your display should look like the example below.



At run time the display will show only the button for which the expression is true. You can test your display by pulling your chassis monitor on top and change the key switch on the controller.



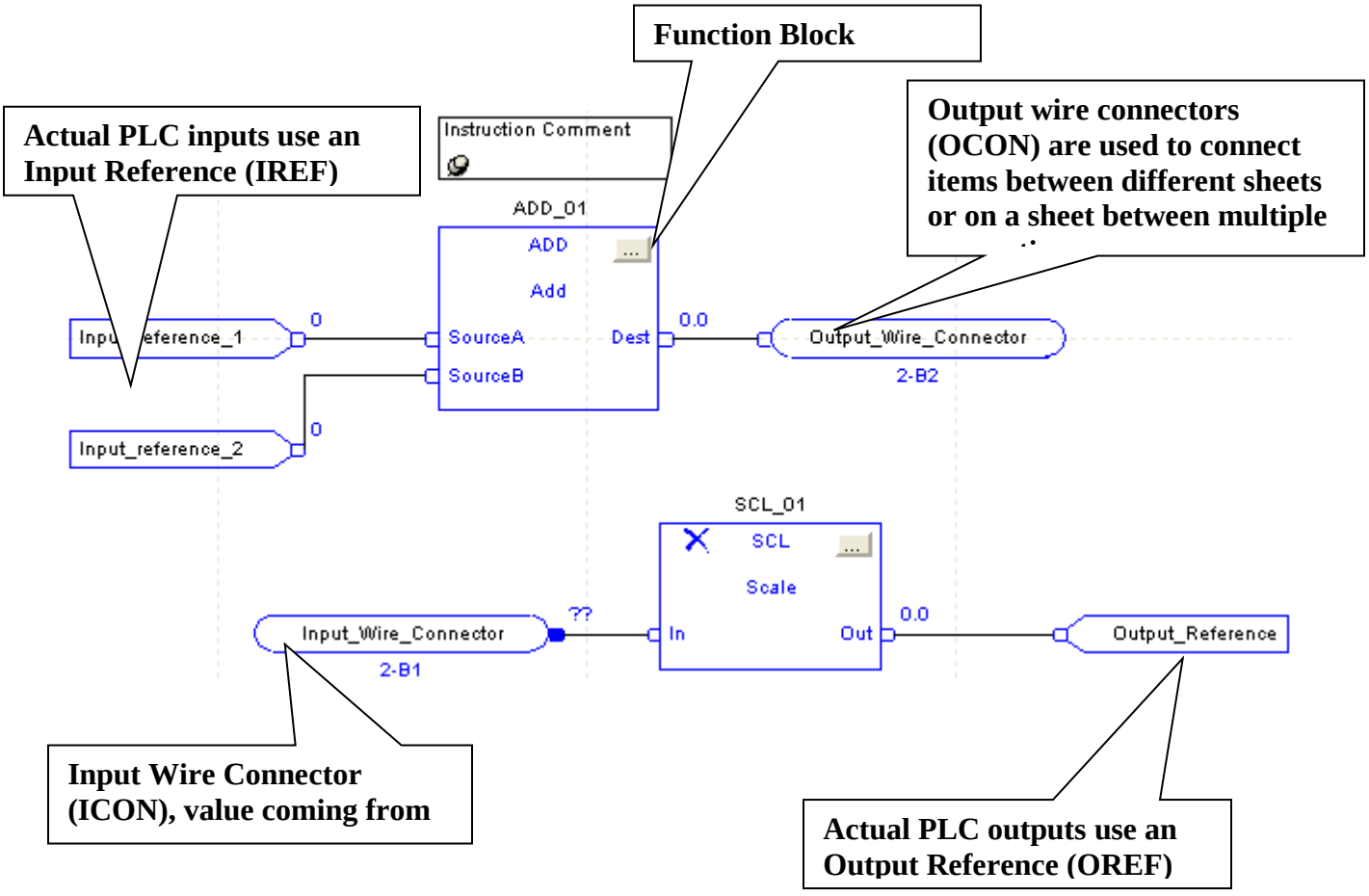
Now you can see what the position of the key switch is on the controller but you can actually see the mode of the controller. I could have the Key Switch in "Remote" and use Studio 5000 to change the mode to "Program".

If you want and there is time left, you can use the above information to show on the display what the actual mode of the controller is.

Introduction to Function Block Programming

Ladder logic is by far the most popular language to program PLC's. In certain situations though, it will be easier to use another language. In this section you will be introduced to Function Block programming. Function Block is ideal for applications that use a lot of analog control functions. In the process industry, where you will find analog for temperature control, flow control, level control, etc. Function Block might be the better choice for programming.

Let's start with some naming conventions associated with Function Block programming. When you create a new routine in a program and double click on the routine to open it, the workspace that opens in the routine window is called a sheet. On a Function Block sheet you work with the following components:

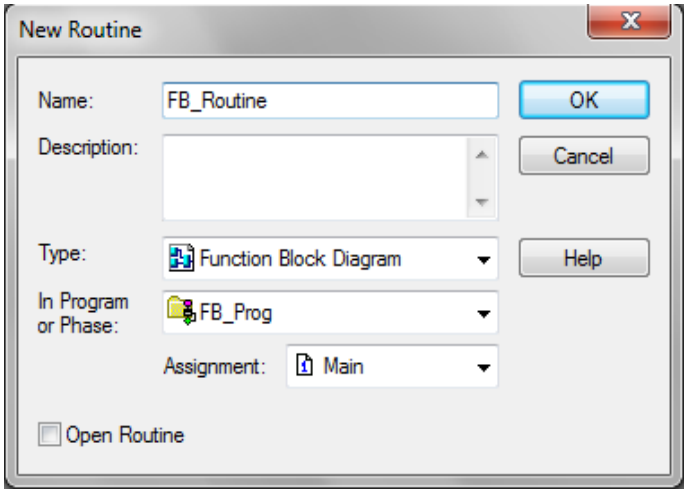


Let's work on the following lab to become more familiar with Function Block Programming and use some "ActiveX" components that are available from the Studio 5000 CD. They should be installed on your computer, if not let your instructor know.

Create the Function Block routine and schedule it to run

A ‘Routine’ is the basic container of your program. It can be made up of any of the supported languages of the controller; ladder diagram, function block, structured text or sequential function chart. Many times the choice of language at this level is driven by the actual functionality of the specific routine and the suitability of one of the supported languages to accomplish the task by virtue of its feature set.

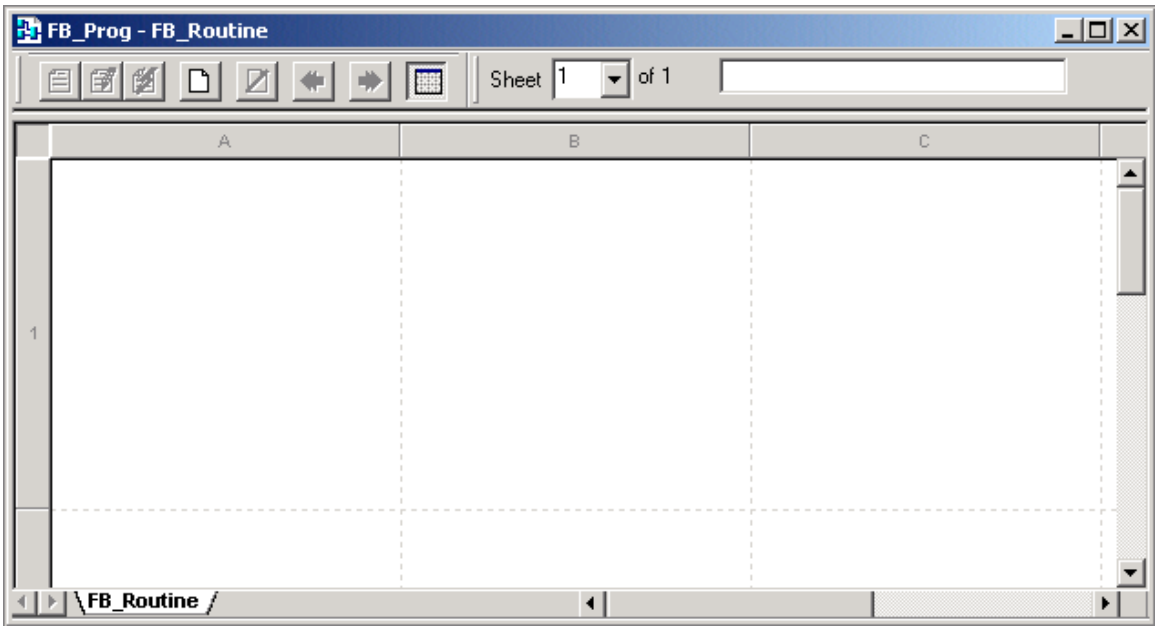
1. Create a periodic Task FB_Task. Create a Program inside the Task FB_Prog.
2. Right click on your program “FB_Prog” and choose ‘New Routine …’.
3. Create your new routine as follows and click ‘OK’ when finished:



Note that the Type is a Function Block Diagram

The project tree should now appear as follows:

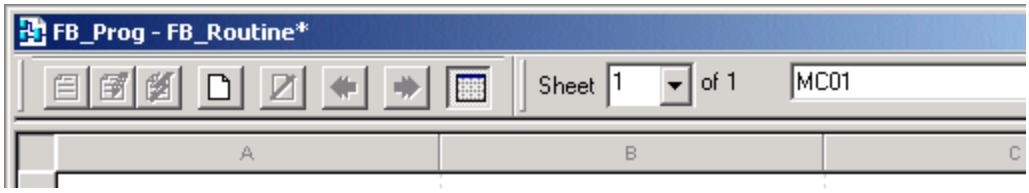
1. In your project tree, double click on your routine “FB_Routine”, a blank sheet (sheet 1) should open in the workspace.



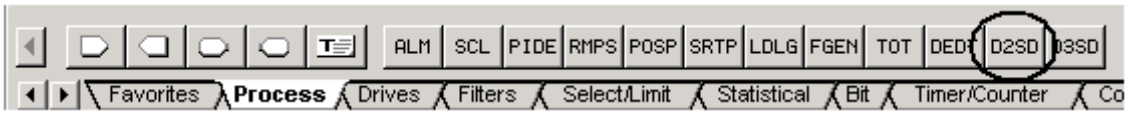
Function block routines are comprised of ‘sheets.’ A function block routine must have one sheet but can have as many as is required by the programmer. Sheets give the programmer the ability to segment the program organizationally where it is desired. Sheets are for organizational purposes only and have no influence over how the program executes.

The first block to add to the diagram is the 2 State Device Driver Block (D2SD) to control a simulated motor.

2. Name this sheet MC01 in the namespace edit box.



3. On the “Process” tab on the toolbar, click on the “D2SD” function.

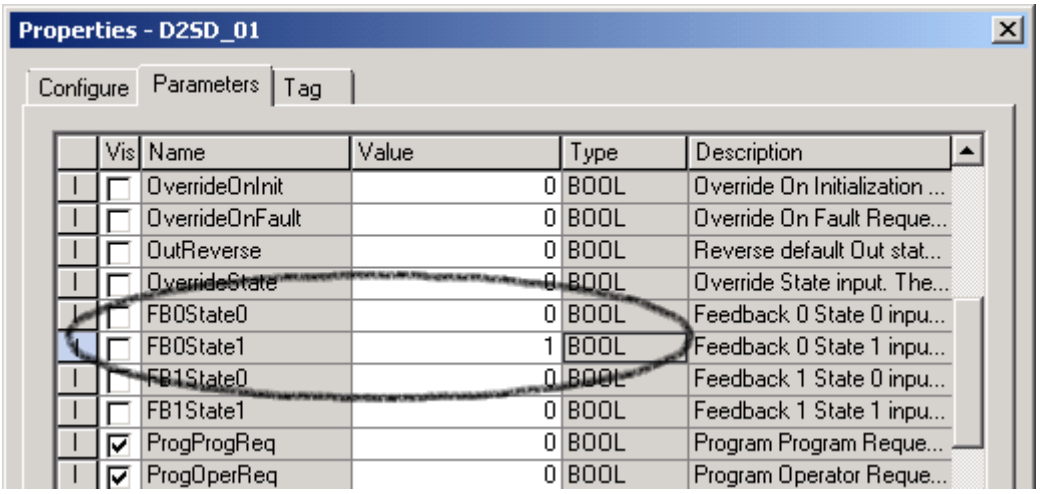


The D2SD block should now appear on the diagram.

Click on the properties button for this block  and take a minute to view all of the available parameter.

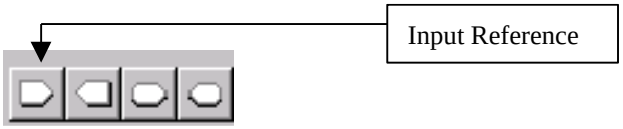
Note that the first column tells whether the parameter is an input or an output to the block. Also, you can use the checkbox in the second column (the column labeled ‘Vis’) to expose or hide parameter pins on the block itself. This is convenient to hide parameters which are not required by the program.

4. In the properties, scroll down to and change ‘FB0State1’ to ‘1’. This configuration tells the device driver that when FB0=0 it is in state ‘0.’ Likewise, when FB0=1 and it is in state ‘1.’ Change this value by clicking on the respective value field and editing the entry.

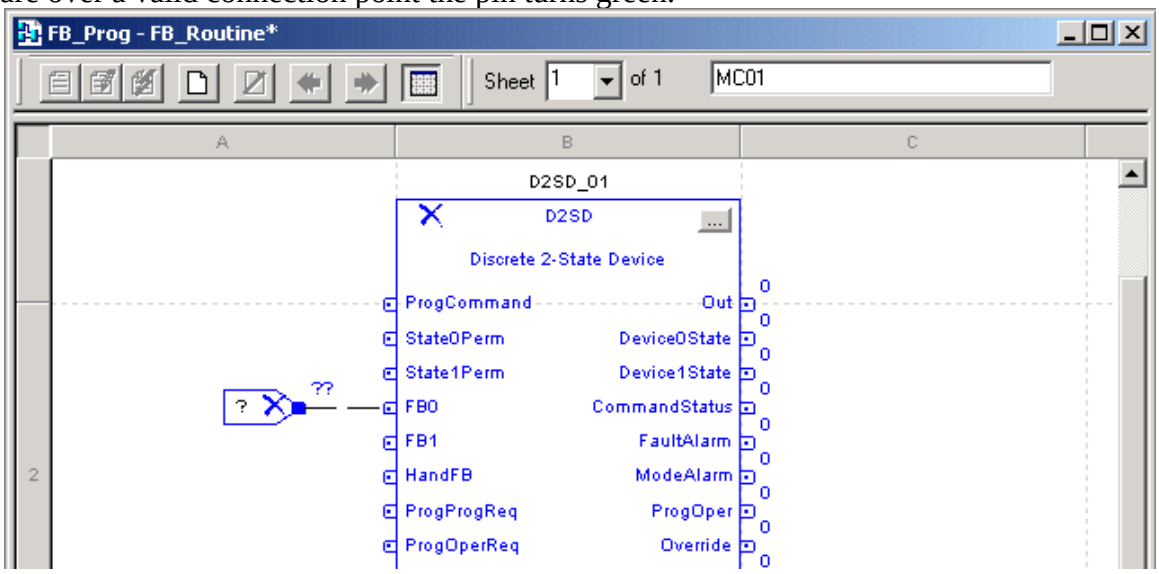


Click ‘Apply’ to apply the change and ‘OK’ to close the D2SD properties dialog.

5. Choose the input reference object from the toolbar.

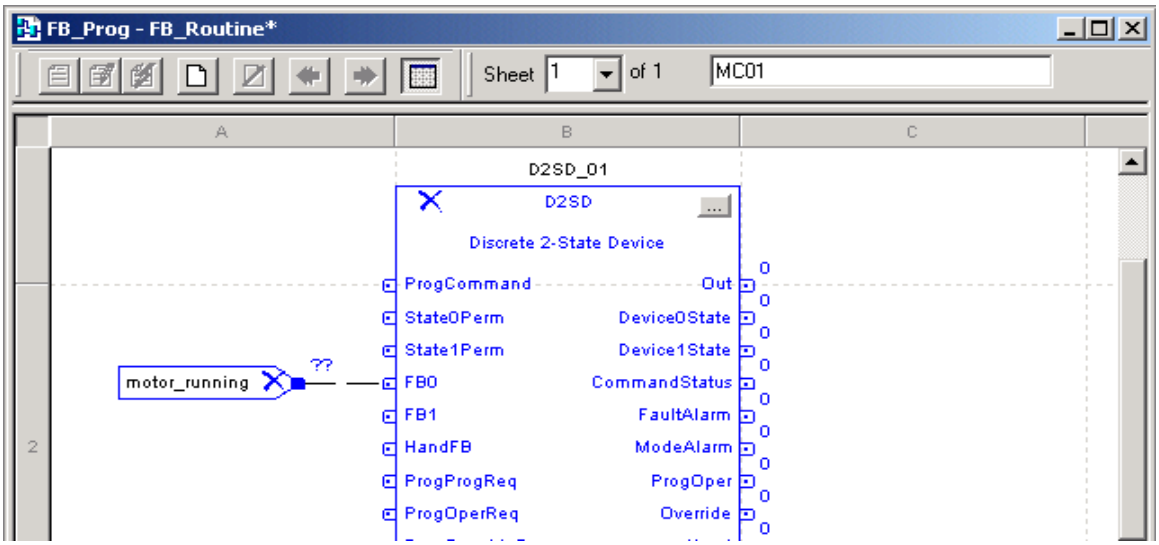


Move the input reference (by dragging) to the input (left) side of your D2SD block and connect it to the FB0 point by clicking once on the input reference output pin and once on the D2SD FB0 input pin. Note that if you are over a valid connection point the pin turns green.

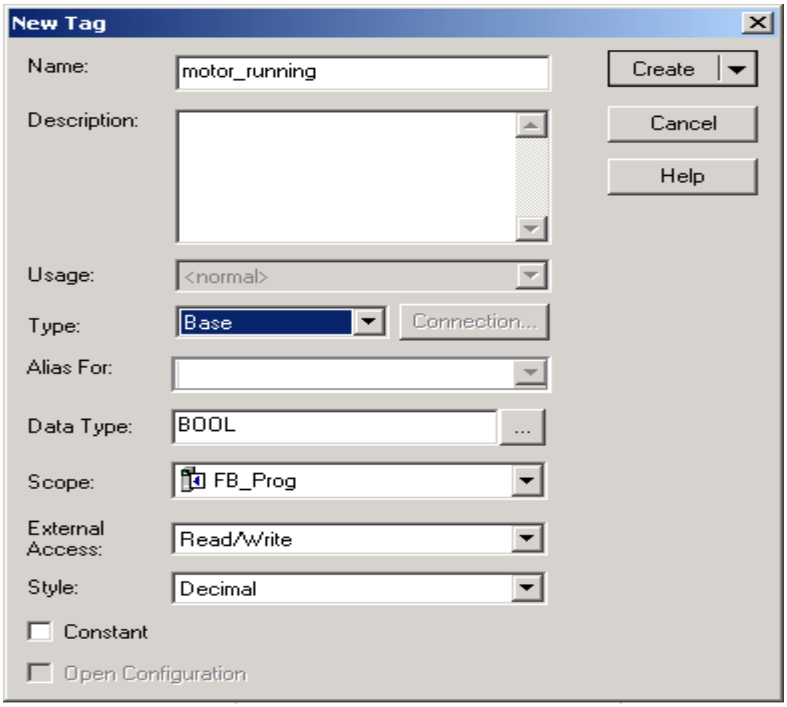


Input and Output References are the blocks used to interface tags in the tag database to the function block program. Input Reference blocks use the current value of the tag to pass to the rest of the program, where-ever it is connected. Likewise, Output Reference blocks take the result or value that they are given by the program and set the value of the referenced tag.

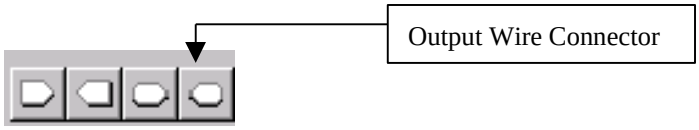
6. Create the input tag by double clicking on the tag reference (currently a question mark) in the tag reference block and type “motor_running”. Accept this by pressing return.



7. Create the input tag by right clicking on the tag reference and selecting ‘New “motor_running” ‘. Make this tag a BOOL data type of controller scope. Note that this tag could have been any tag in the controller database: a discrete input, a calculated result, local or global tags, etc.

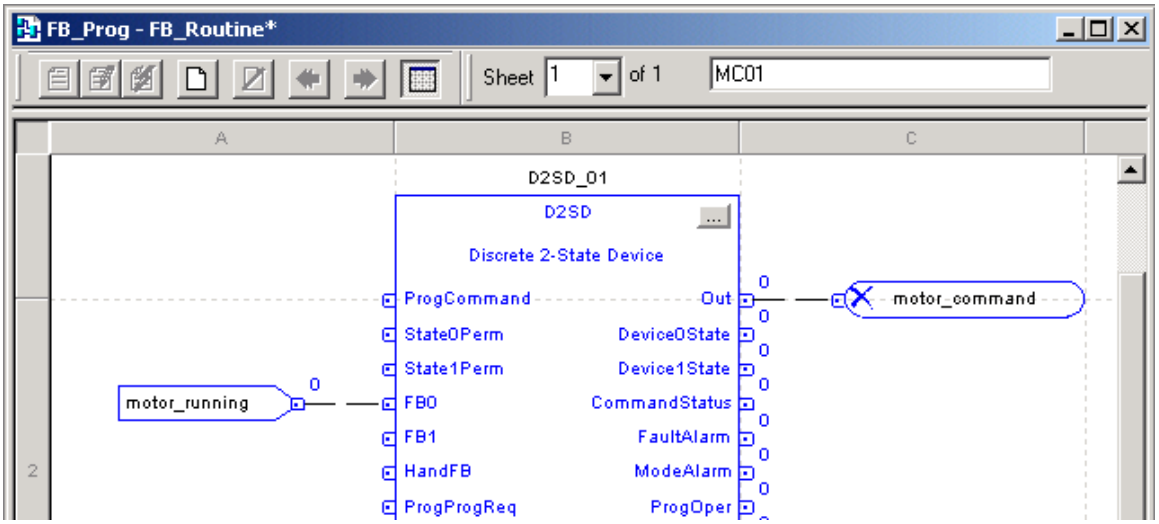


8. Choose the Output Wire Connector object from the toolbar.



Move the Output Wire Connector (by dragging) to the output side of your D2SD block and connect it to your Out point by clicking once on the D2SD Out pin and once on the Output Wire Connector input pin.

9. Double click on the wire connector reference and type “motor_command”. Pressing return to accept.



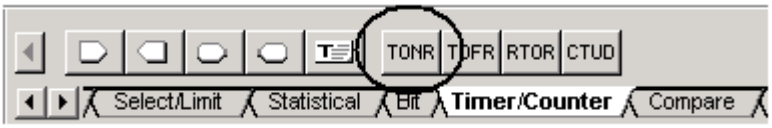
Input and Output Wire Connectors are the blocks used to interface values between sheets in the same routine.

Edit the motor simulation sheet

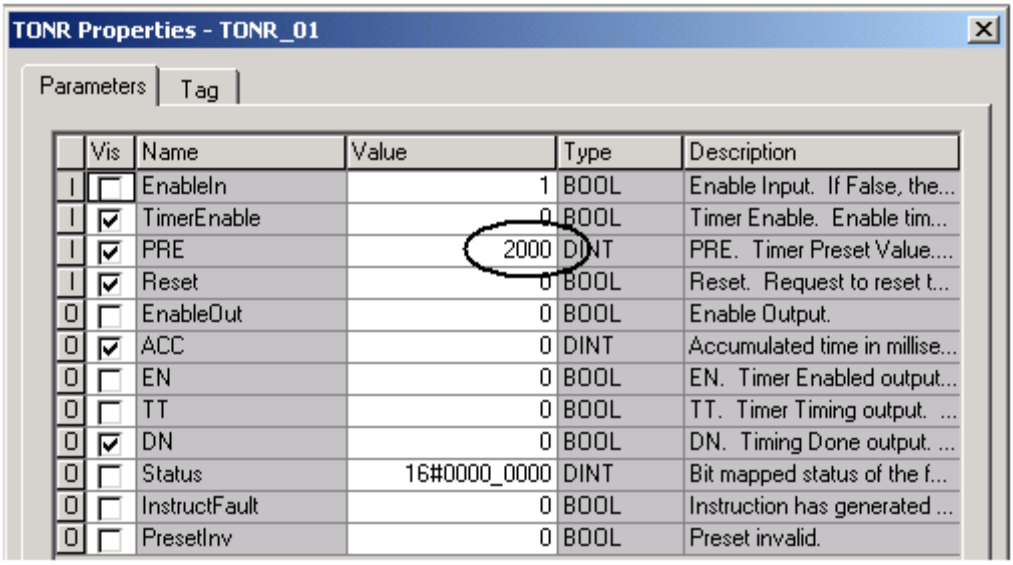
1. Create a new sheet for the simulation elements by clicking once on the new sheet button.

You should now be on a ‘clean’ sheet designated sheet 2 of 2. This sheet will contain the simulation. Name the sheet ‘Simulation’.

2. From the “Timer/Counter” tab on the toolbar,



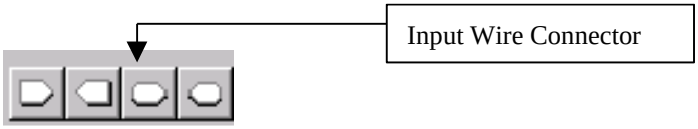
Select and place a Timer On Delay with Reset (TONR) block onto sheet 2. Open the TONR parameters (by clicking on the ellipsis) and configure a 2 second Preset.



	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Enable Input. If False, the...
I	☑	TimerEnable	0	BOOL	Timer Enable. Enable tim...
I	☑	PRE	2000	DINT	PRE. Timer Preset Value....
I	☑	Reset	0	BOOL	Reset. Request to reset t...
O	☐	EnableOut	0	BOOL	Enable Output.
O	☑	ACC	0	DINT	Accumulated time in millise...
O	☐	EN	0	BOOL	EN. Timer Enabled output...
O	☐	TT	0	BOOL	TT. Timer Timing output. ...
O	☑	DN	0	BOOL	DN. Timing Done output. ...
O	☐	Status	16#0000_0000	DINT	Bit mapped status of the f...
O	☐	InstructFault	0	BOOL	Instruction has generated ...
O	☐	PresetInv	0	BOOL	Preset invalid.

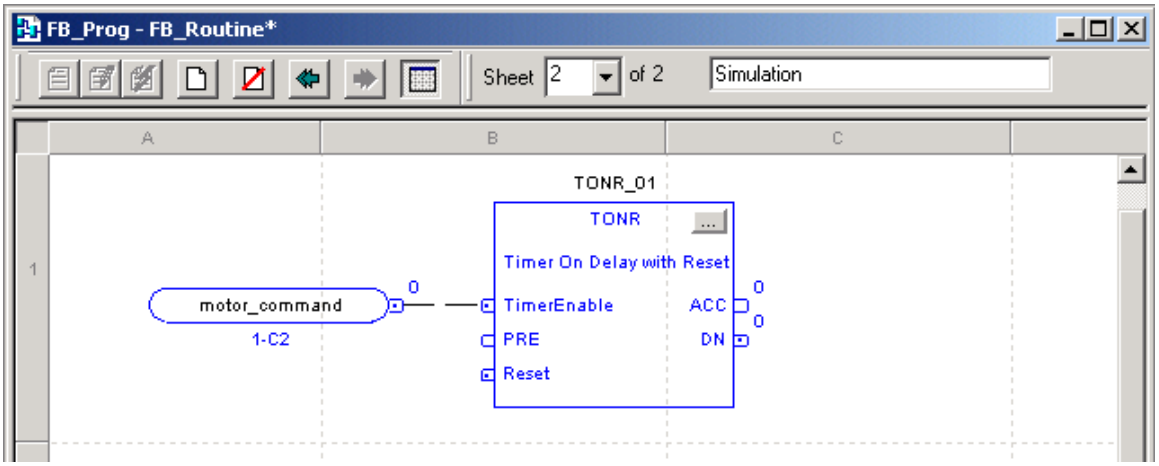
Apply the change and click ‘OK’.

3. Choose an Input Wire Connector from the toolbar and connect it to the TimerEnable input of the TONR block.



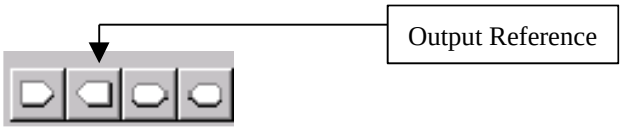
4. Double click on the wire connector reference. Click the arrow for a drop down list of available connector references. Choose “motor_command” and press return to accept.

   The drop-down that appears when clicking inside of an Input Wire Connector will display a list of all available references from Output Wire Connectors in the current routine. Likewise, the drop-down that appears when clicking inside of an Output Wire Connector will display a list of all available references from Input Wire Connectors in the current routine.

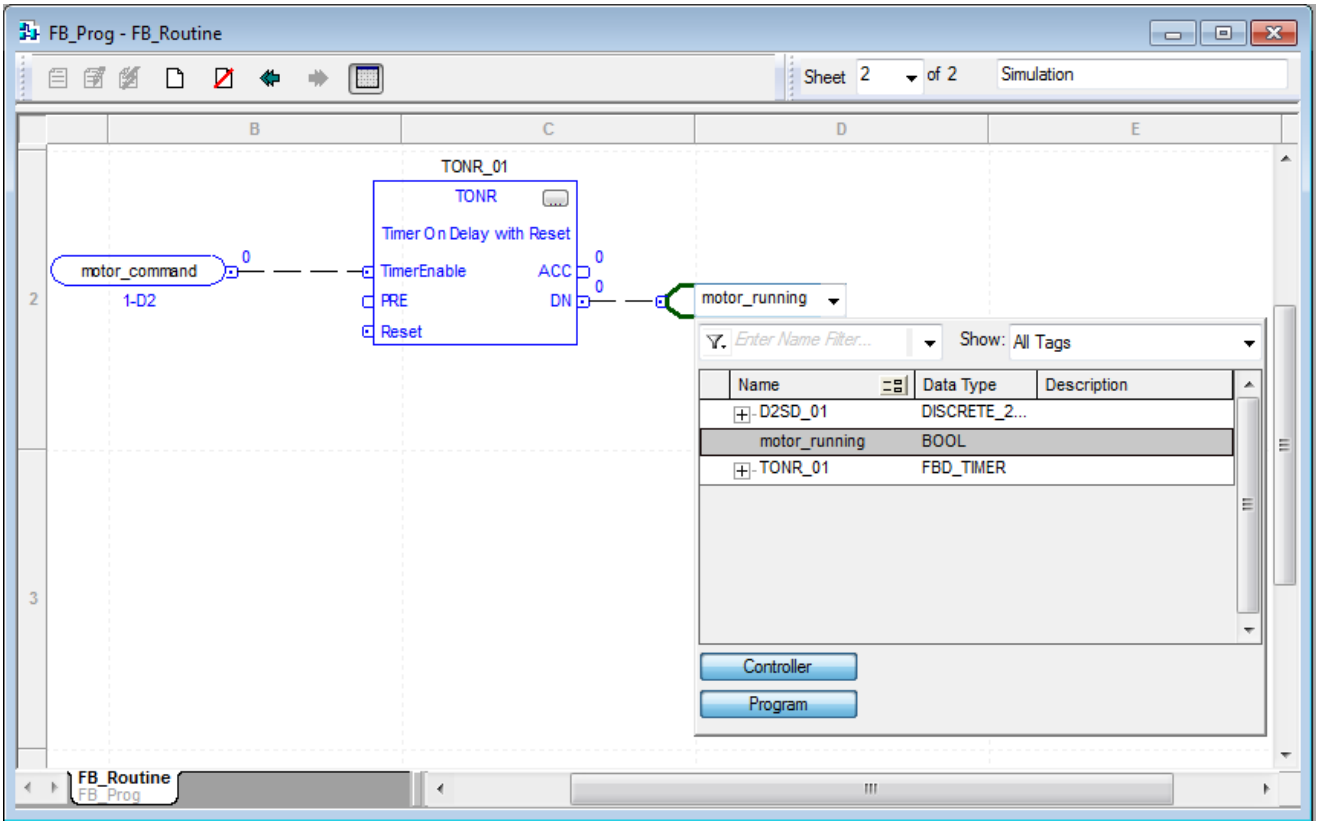


Note the reference which appears below the completed Input Wire Reference (appearing above as '1-C2'). This reference points to the 'source' for this Input Wire Reference as 'Sheet 1 – Grid Location C2. In this way Wire Connectors can reference their sources and destinations so that a programmer can easily navigate to the corresponding sections of the program by clicking on the reference. When you do so the function block environment will immediately take you to the corresponding location in the program.

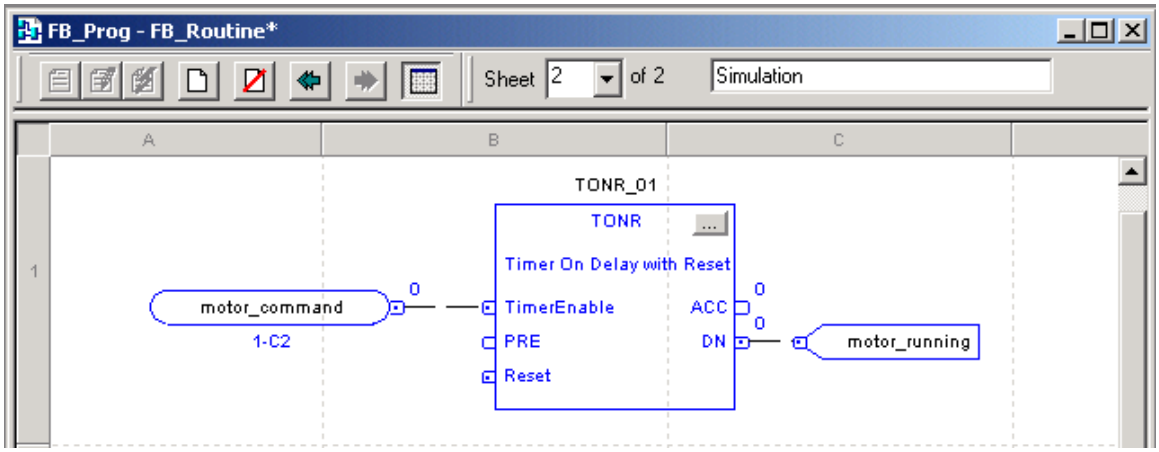
5. Drop an Output Reference onto the sheet and connect it to the DN output of the TONR block.



Double click on the Output Reference. Click the arrow for a drop down list of available tags. Choose the controller scoped tag "motor_running" and press return to accept.



You should have a diagram for sheet 2 similar to this:



6. Verify the entire project to this point by clicking on the ‘Verify Controller’ icon

If no errors are detected, the device control and motor simulation are finished

7. Download your project to the controller and change the controller to Run mode.
8. At this point you have created a FB project and we will communicate with that program from FTA View SE.

Active X Faceplates

Studio 5000 provides Fourteen Active X faceplates which can be used in RSView32 or any Active X container.

We will create a faceplate in Factory Talk View Site Edition (Local Station) for our operator (The Active X control which we are going to use is not available in Machine Edition of Factory Talk View).

First we need to set up an OPC topic in RSLinx which the faceplate can use to communicate with the controller. The other solution is to create one RSLinks Enterprise server inside FTA View SE.

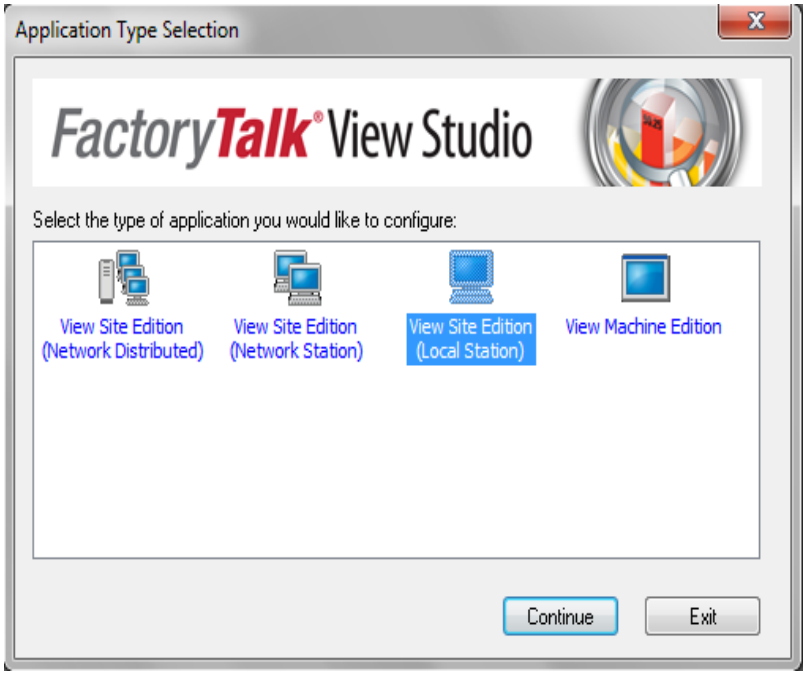
1. Go to RSLinx and select "DDE/OPC" and then "Topic Configuration". Select the "New" button and create a topic called "FBTopic".(**RSLinx must run in application mode**)

2. Find the appropriate driver and processor as assigned by the lab instructor and select it. **If you don't know which processor is yours, please ask the instructor.**

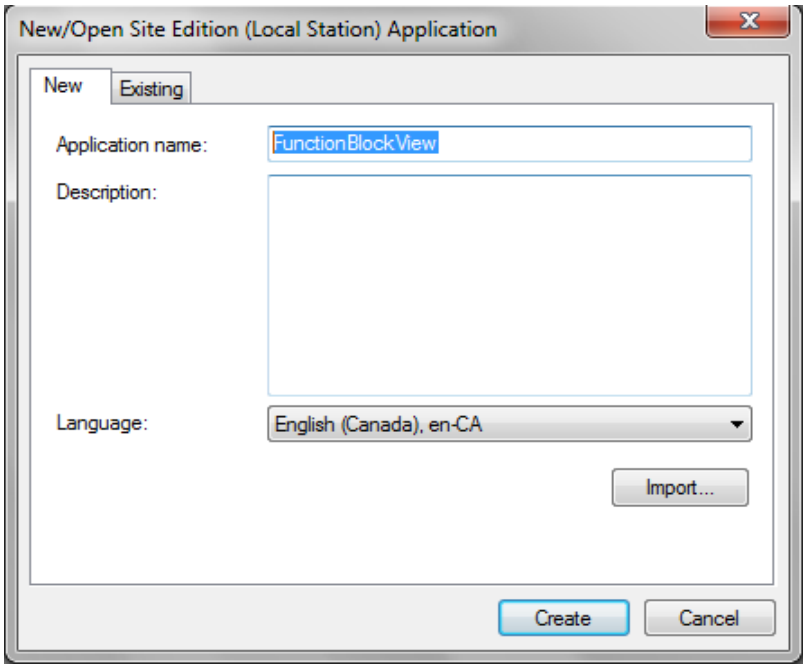
3. Under the 'Data Collection' tab verify that the 'Polled Messages' selection is checked and the 'Unsolicited Messages' selection is unchecked. Choose Apply and then Done when finished.

Now we can start Factory Talk SE and place a D2SD faceplate on a worksheet.

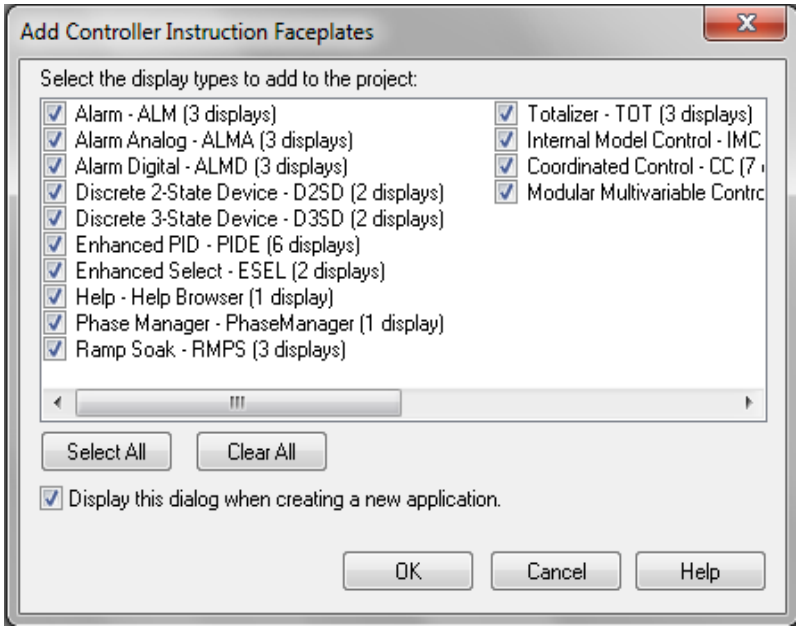
4. Start Factory Talk View Studio and in the Welcome Page choose "**View Site Edition(Local Station)"** and click Continue.



5. In the New/Open menu choose New and give the application a name. For example" **FunctionBlockView**" and click Create.

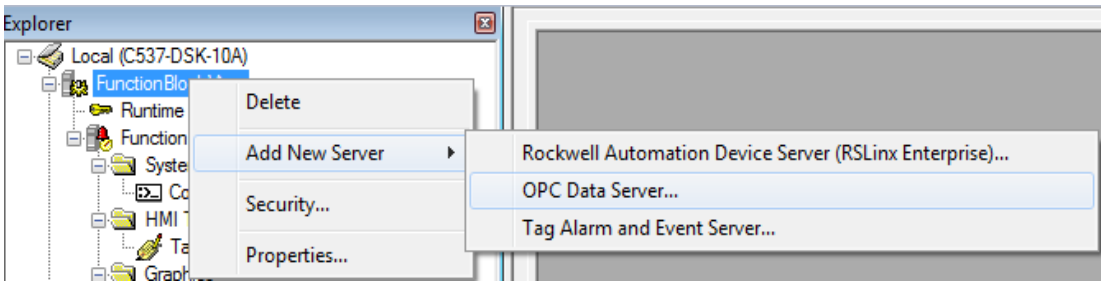


6. In the "**Add Controller Instruction Faceplates**" page Select All and click OK to add all Fourteen Faceplates to the project.

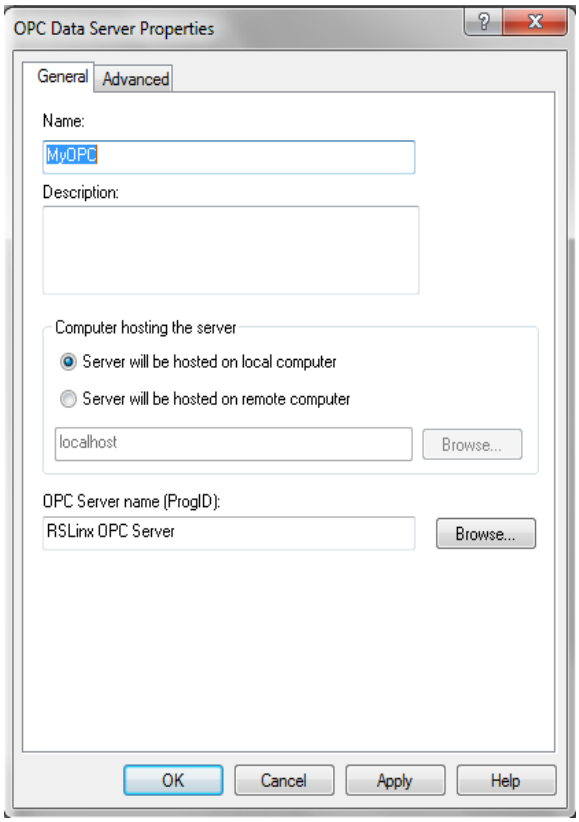


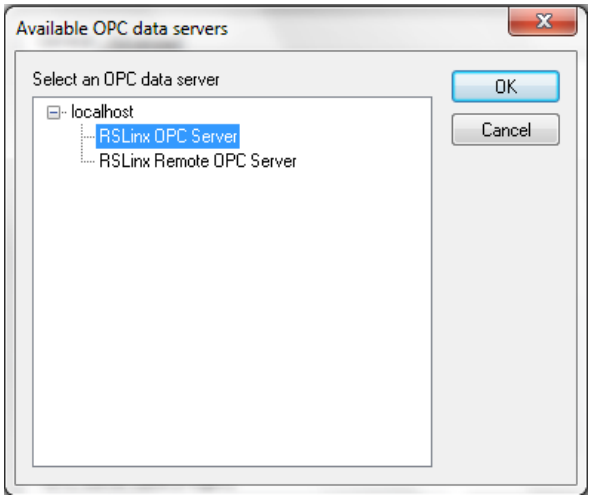
7. The project is created and we need a device driver to communicate with PLC(in this project SoftLogix). To do this we are going to use the OPC driver we created in RSLinks.

Goto Explorer in your project and write click on the name of project in main tree. Then Add New Server and OPC Data Server.



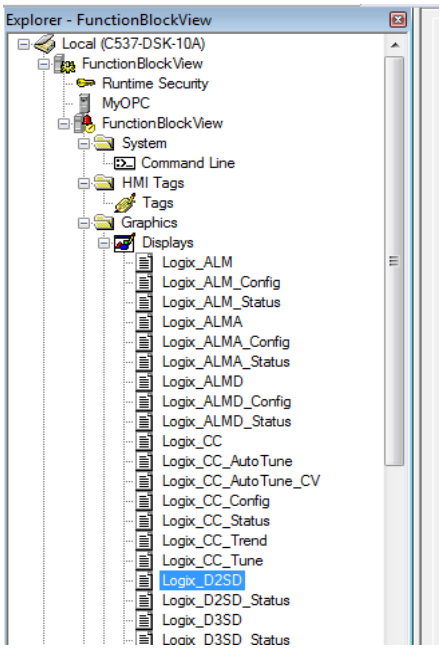
8. On the General Tab give a name to OPC server and Browse and select "**RSLinx OPC Server**" then OK and OK again.

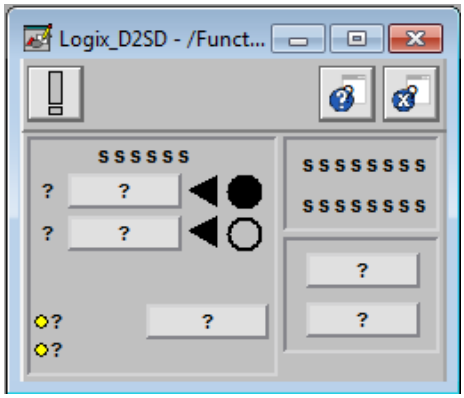




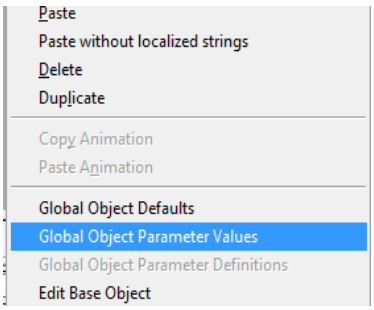
9. On the Explorer tab open the display section and

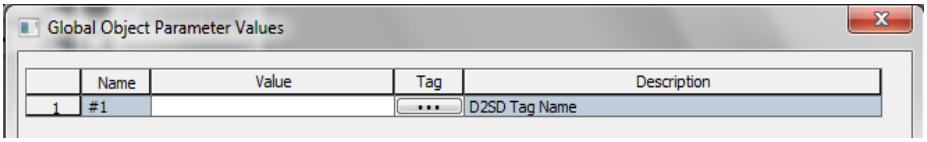
   double click Logix-D2SD. The following window appears.



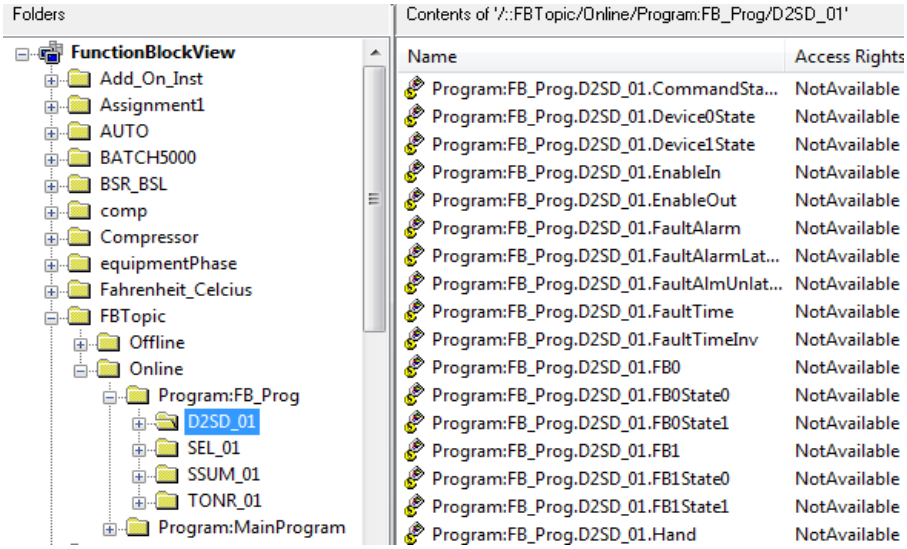


10. Right click on this window and select "**Global Object Parameter Values**".

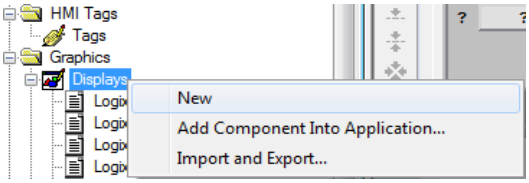


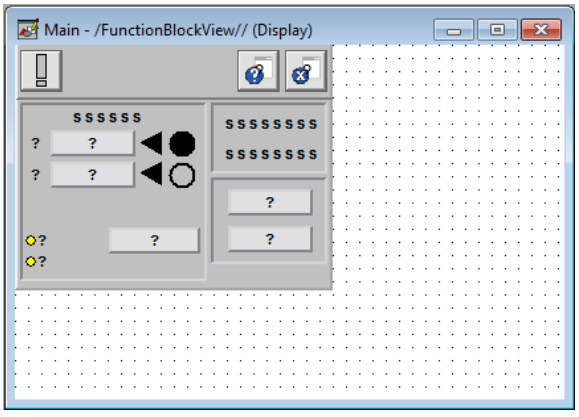


11. Click the ellipse box under tag and select D2SD_01 from the topic "**FBTopic**". This is the topic you have created in RSLinx before. If this topic is not shown in the Folders tree press "Refresh All Folders". Press OK and OK again.

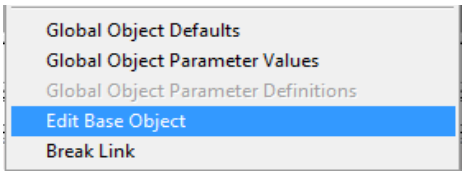


12. On the Logix_D2SD display window right click and select copy. Goto Explorer tree and right click on Display and select New. A new display will be created. Right click on an empty area and click "Paste". The D2SD Faceplate will be pasted on this screen. Save the display and give this page a name. Let's call it "**Main**".

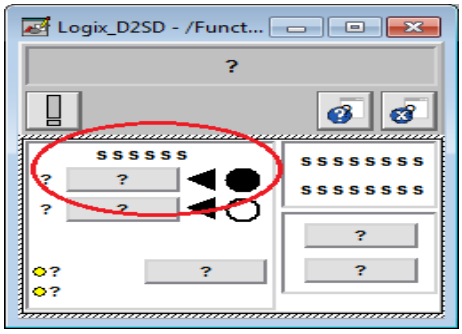




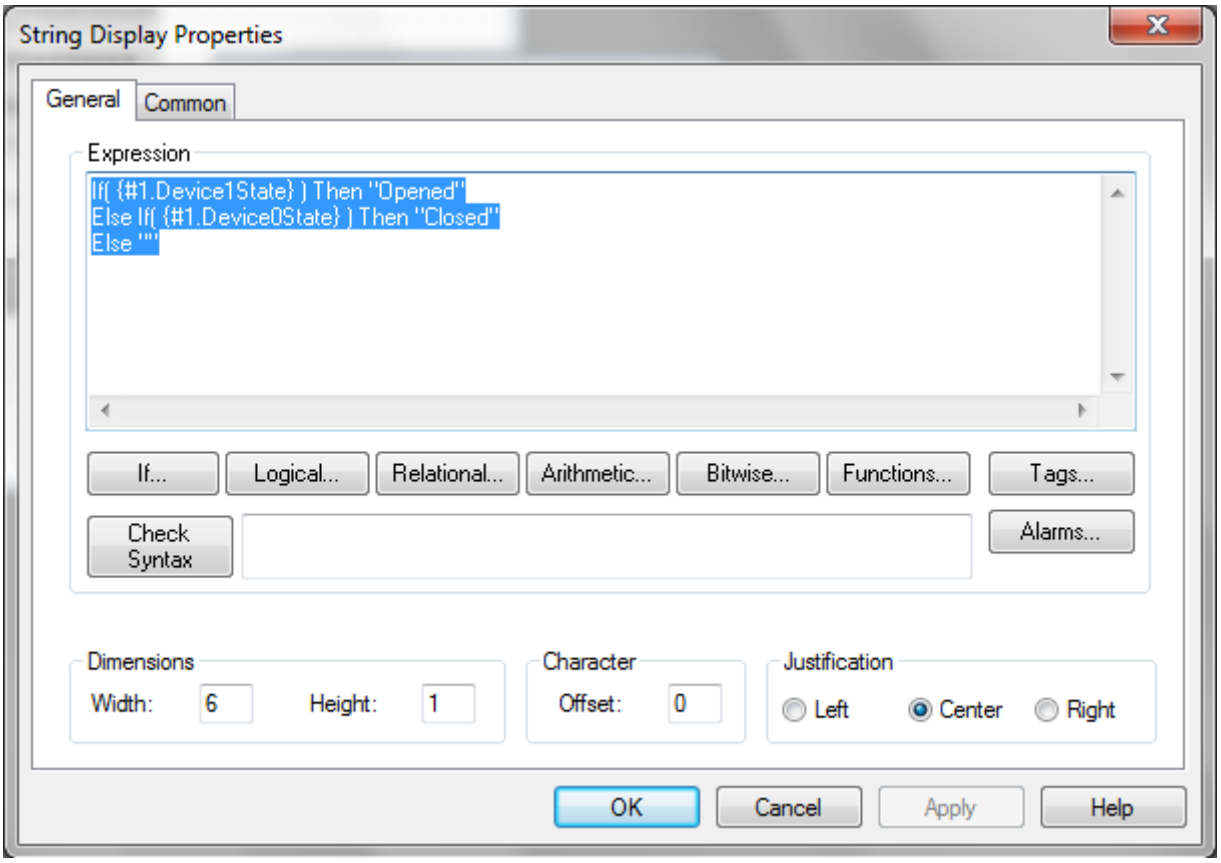
13. On the Main display right click D2SD Faceplate and select "**Edit Base Object**". The Faceplate will be opened in a new window.



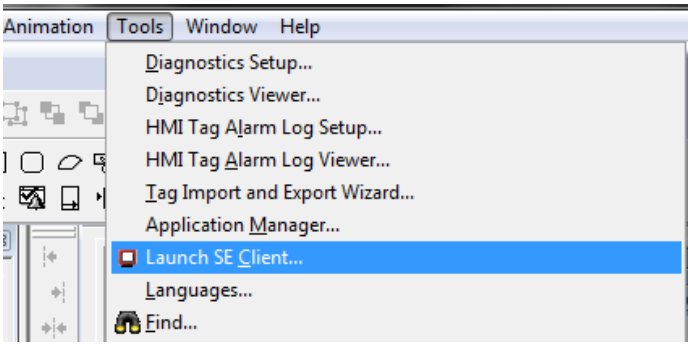
14. Now we are going to customize this object and changing the display string. Point your mouse on the "**ssssss**" and keep double clicking it. With each double click you are getting closer to select the "**ssssss**".



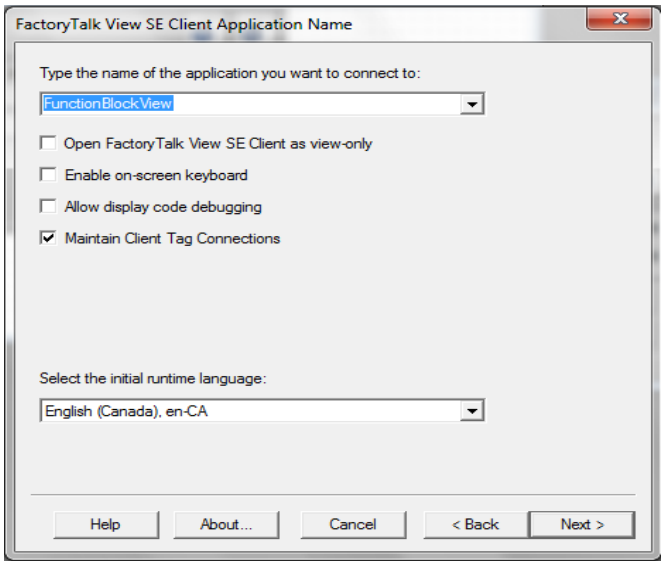
15. After three double click you will get the String Display Properties. In that screen and in the Expression section change the "**Opened"** with "**Started**" and "**Closed**" with "**Stopped**".



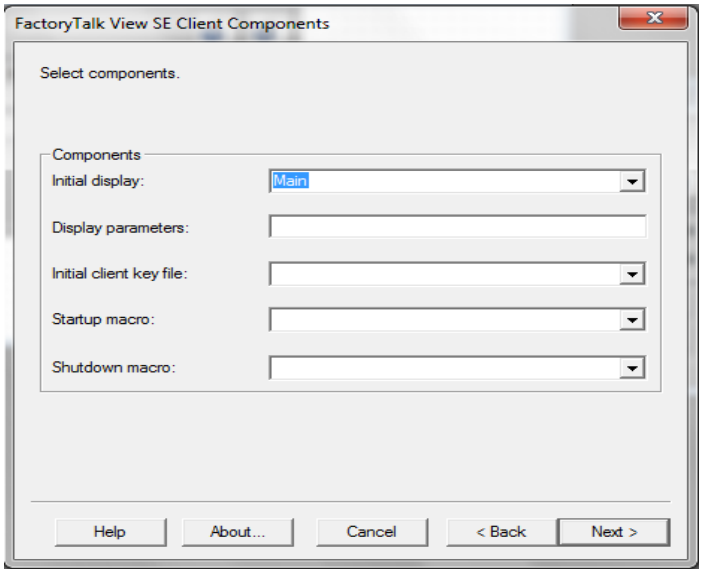
16. Save the Project and goto Tools Menu and select "**Launch SE Client"**.

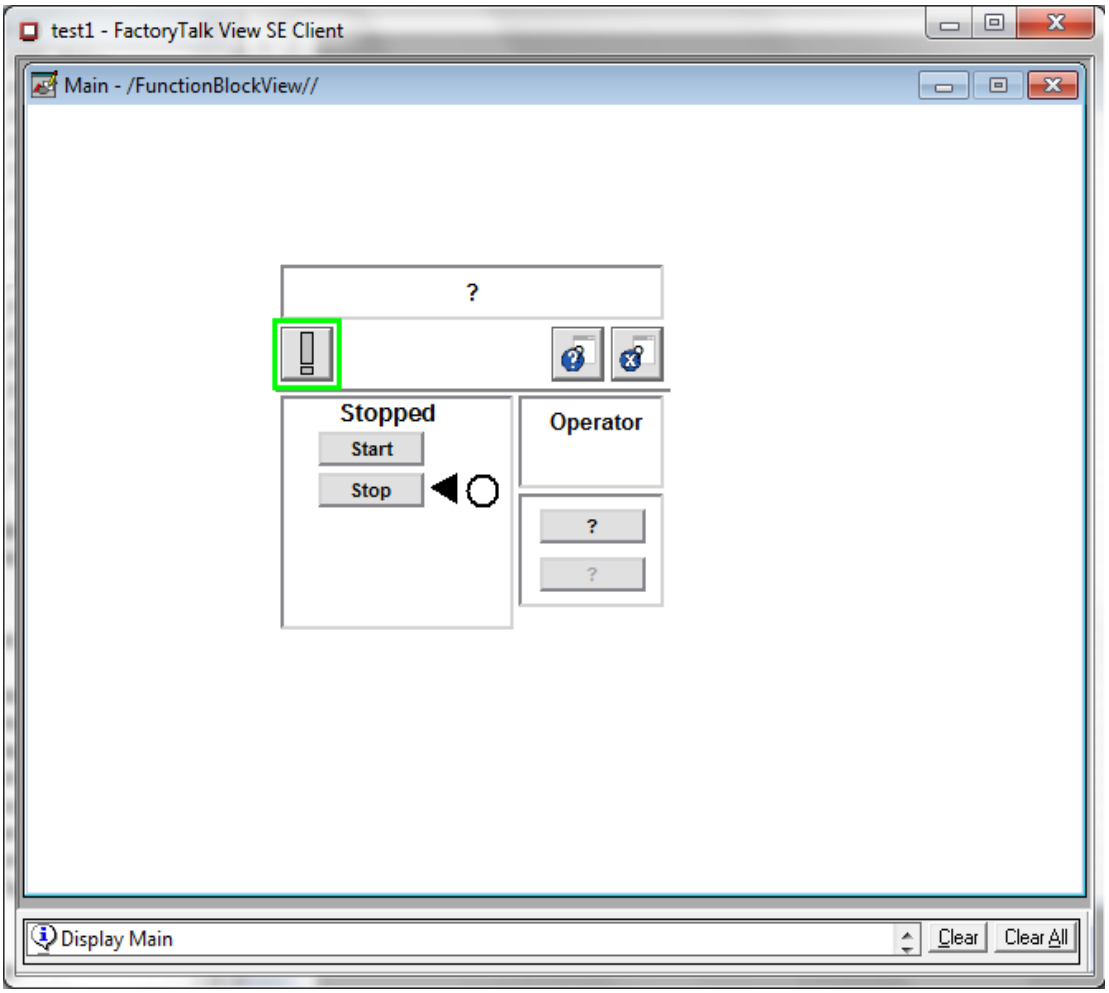


In the welcome page select "**New**". On the next page give the Se Client a name nad choose the path to save it. Then click next. Select local Station and click next. On the next page select the project that you have created in Factory talk SE from drop down menu. Click next.



On the next page choose the Main display (the display which we have created in Factory Talk) from drop down list box and click next. On the ongoing pages accept the defaults and click next and finally **Finish**.





17. Click the ‘Start’ button on the faceplate to command the simulated motor to start. Note that the black arrow will shift upward next to the ‘Start’ button and the text ‘Started’ appears two seconds later. The movement of the arrow indicates that the command to start has been issued. The appearance of the text is confirmation that the motor has indeed started. The two second delay is the delay that was configured in the TONR instruction in the function block program.

18. Click the ‘Stop’ button on the faceplate to command the simulated motor to stop. Note that there is no delay between the arrow (the command) and the text (the confirmation).

Add Motor Speed Control with Function Block On-Line Edit

In this section you will add a speed reference to the motor which is selectable between a ‘Line_Speed_Reference’ and a ‘Diagnostic_Speed_Reference’. This selection will also be contingent upon the motor being commanded to ‘Run.’

This section of the program will be added on-line (while the original code is running).

The on-line editing functionality in the function block environment is procedurally identical to that of ladder routines:

Start edits on the selected routine.

The original routine is running while edits are made on a copy.

Make the desired edits on the copy.

Verify (Accept) the edits in the copy.

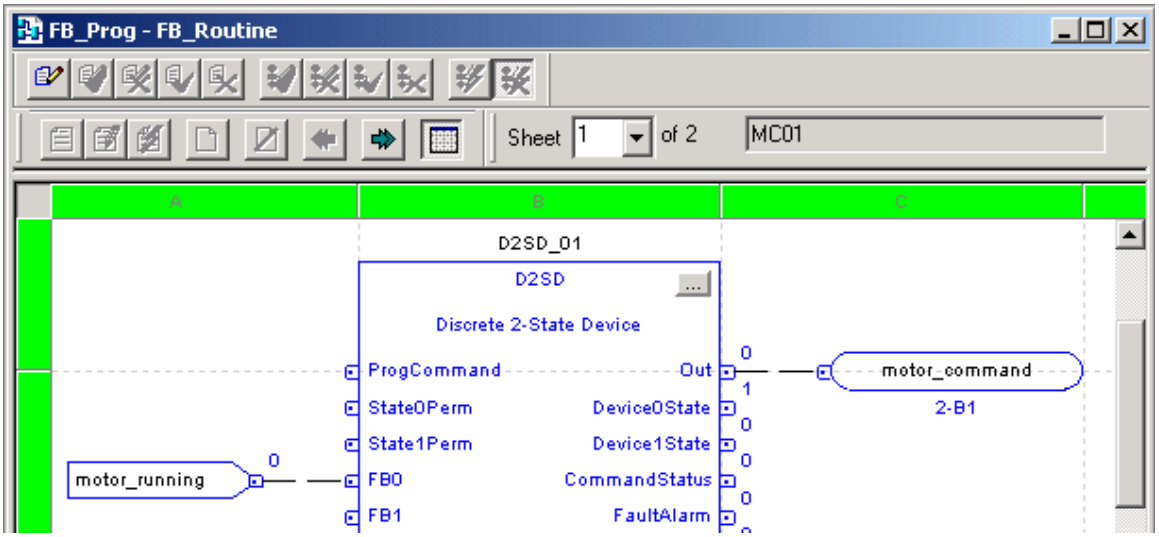
Test the edited copy (swap the original and the edited version in the execution thread).

Optionally Untest to return execution to the original.

Assemble the edited copy to replace the original.

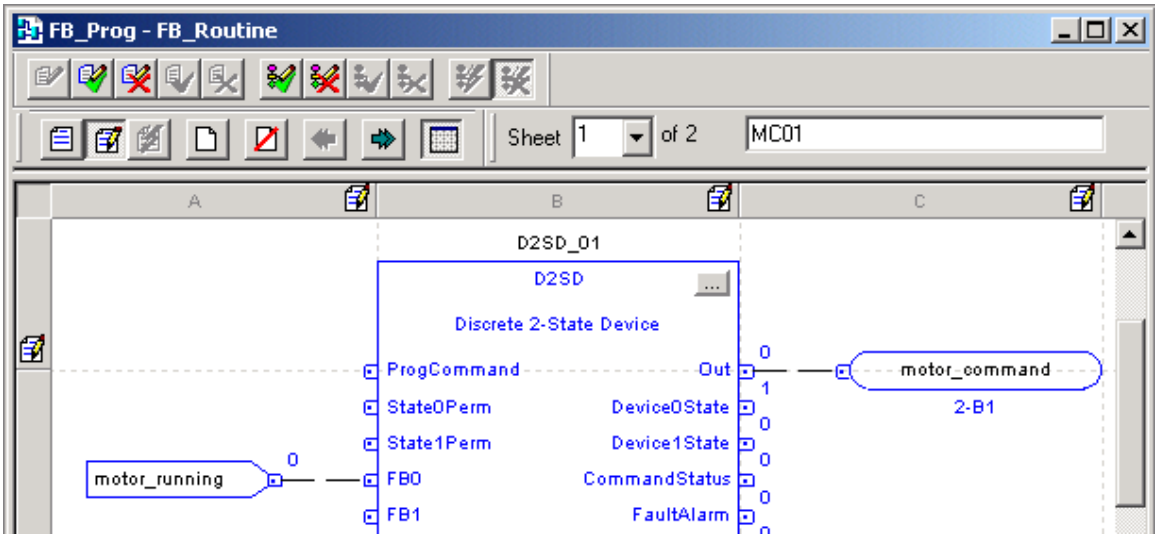
In the Function Block editor the edit 'zone' is the entire routine upon which edits are being performed. So two copies of the routine being edited are held in memory from the time that the edits are verified until the edits are either cancelled or assembled.

1. Bring Studio 5000 to the foreground (if it is not currently) and go on-line with the processor.
2. Open the routine FB_Routine so that sheet 1 is in the editor workspace.

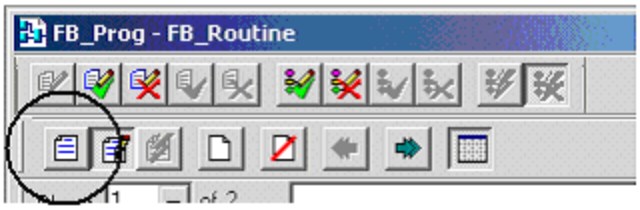


3. Click once on the 'Start Pending Routine Edits' button in the function block toolbar.

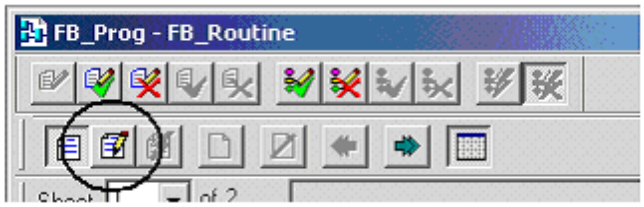
You should now be in the 'Pending Edits' view.



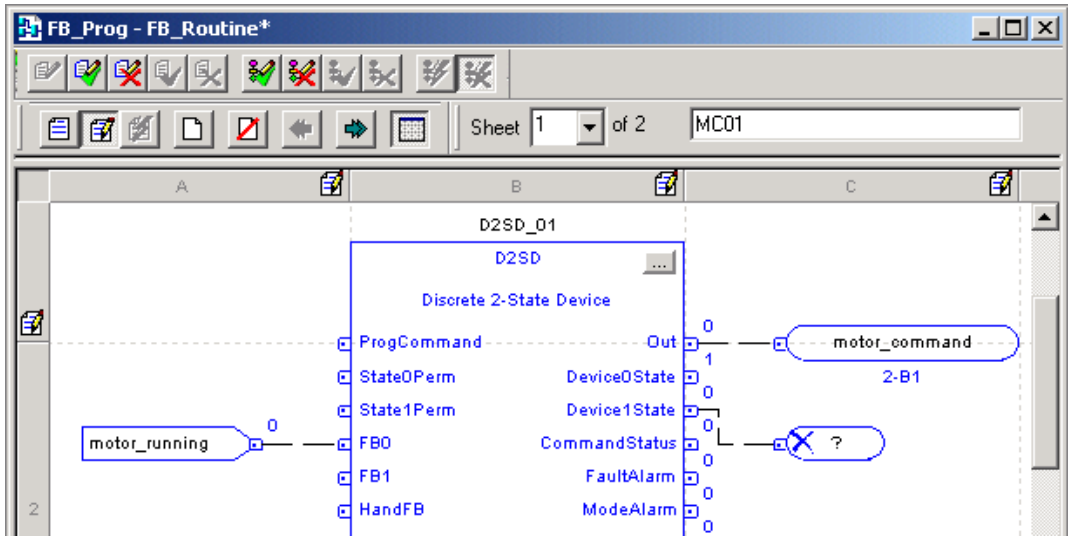
Note that the border around the function block diagram is no longer green. This is because the program is not running the routine being edited. It is still running the original. In order to view the original, click once on the 'Original View' button.



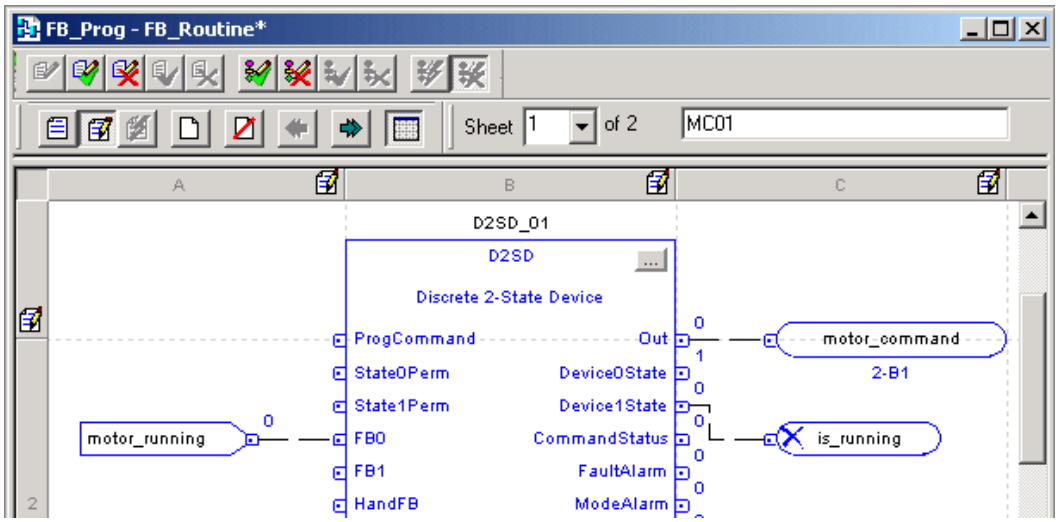
To return to the 'Pending Edits View', click once on the 'Pending Edits View' button.



4. Add an Output Wire Connector to the D2SD Device1State output pin.

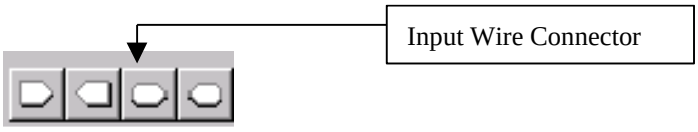


5. In the output wire connector type "**is_running"** as the reference.



6. Create a new sheet for the new functionality by clicking once on the new sheet button.
7. In this new sheet 2 add the following function blocks:

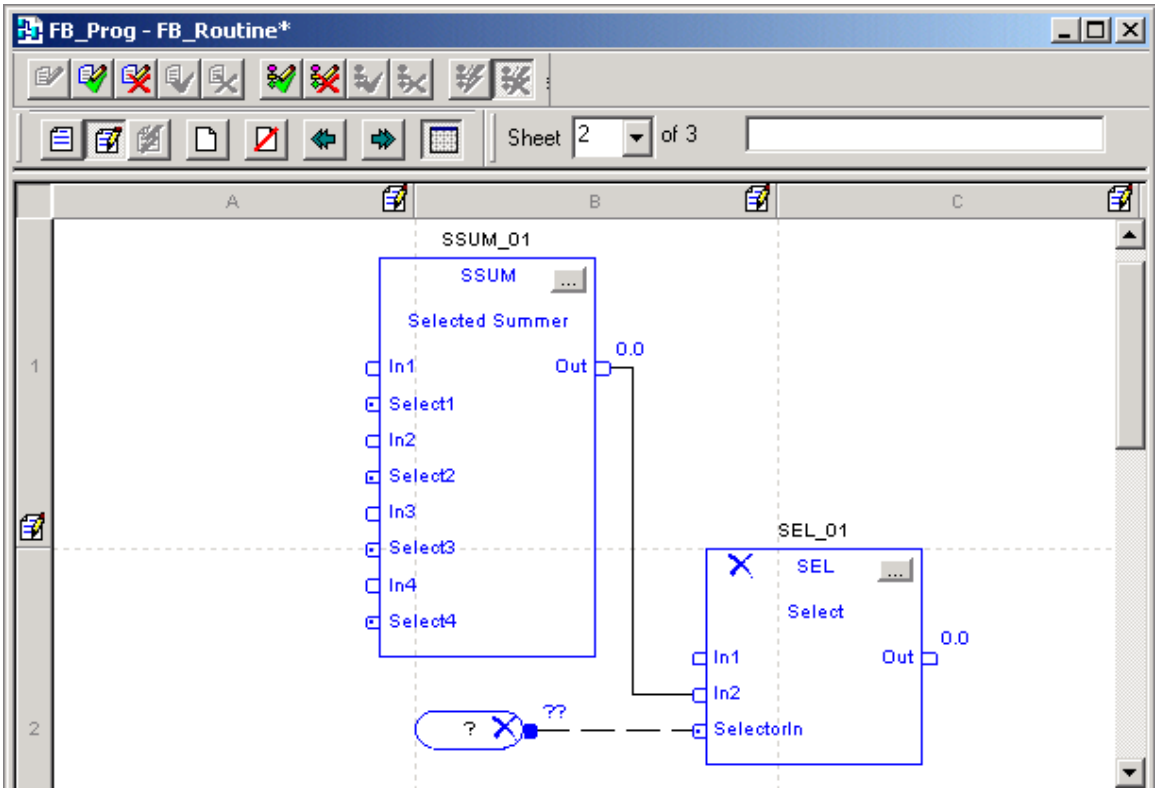
An input wire connector



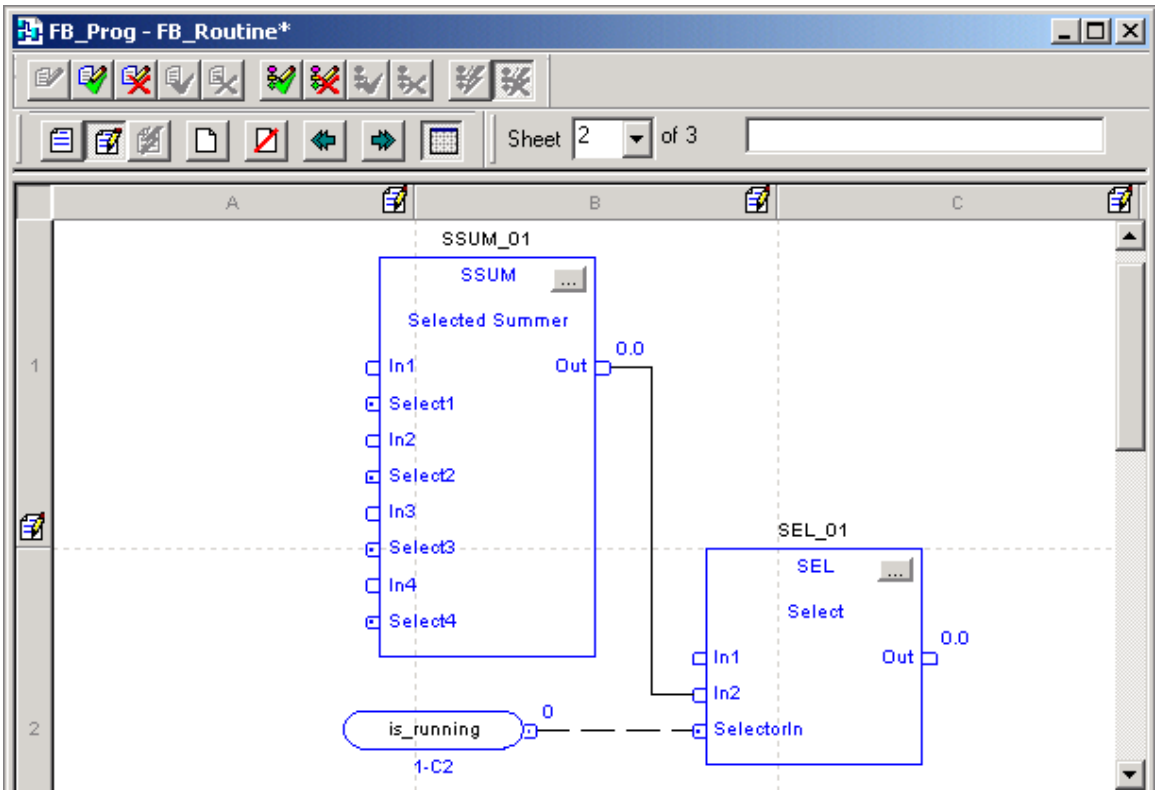
A Selected Summer (SSUM)

A Select (SEL)

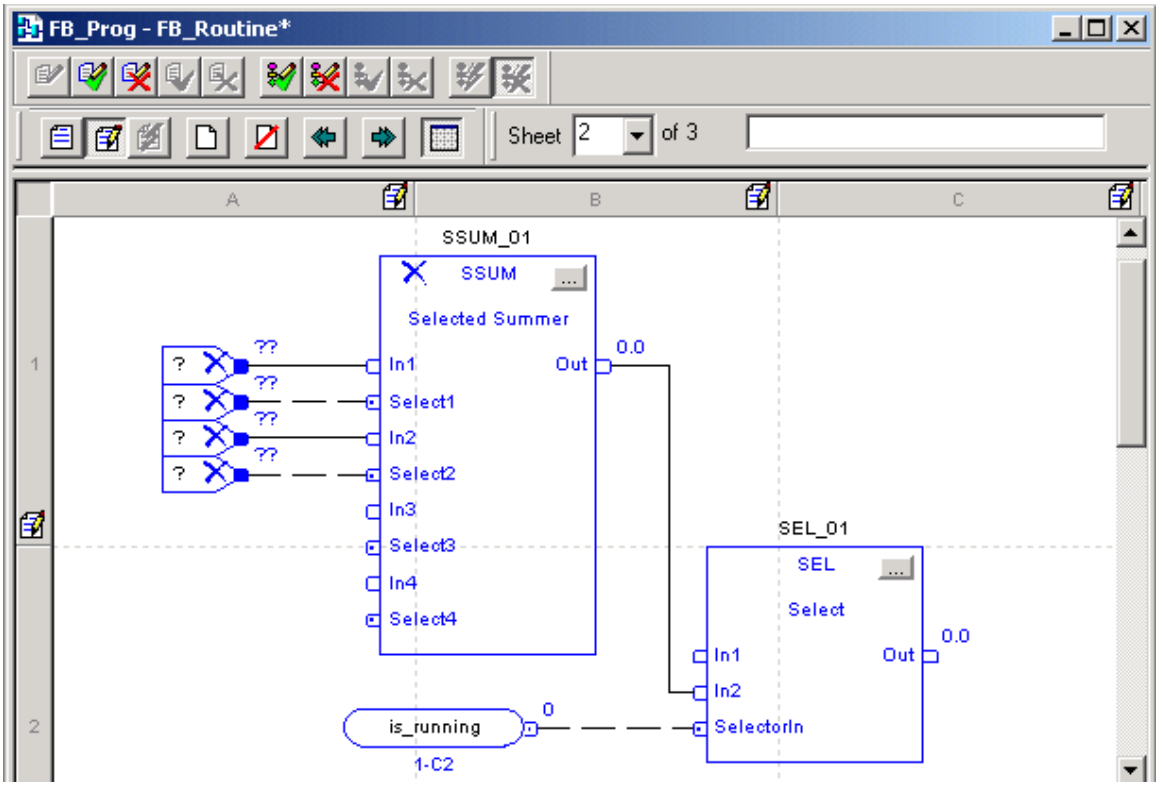
Arrange and connect them as follows.



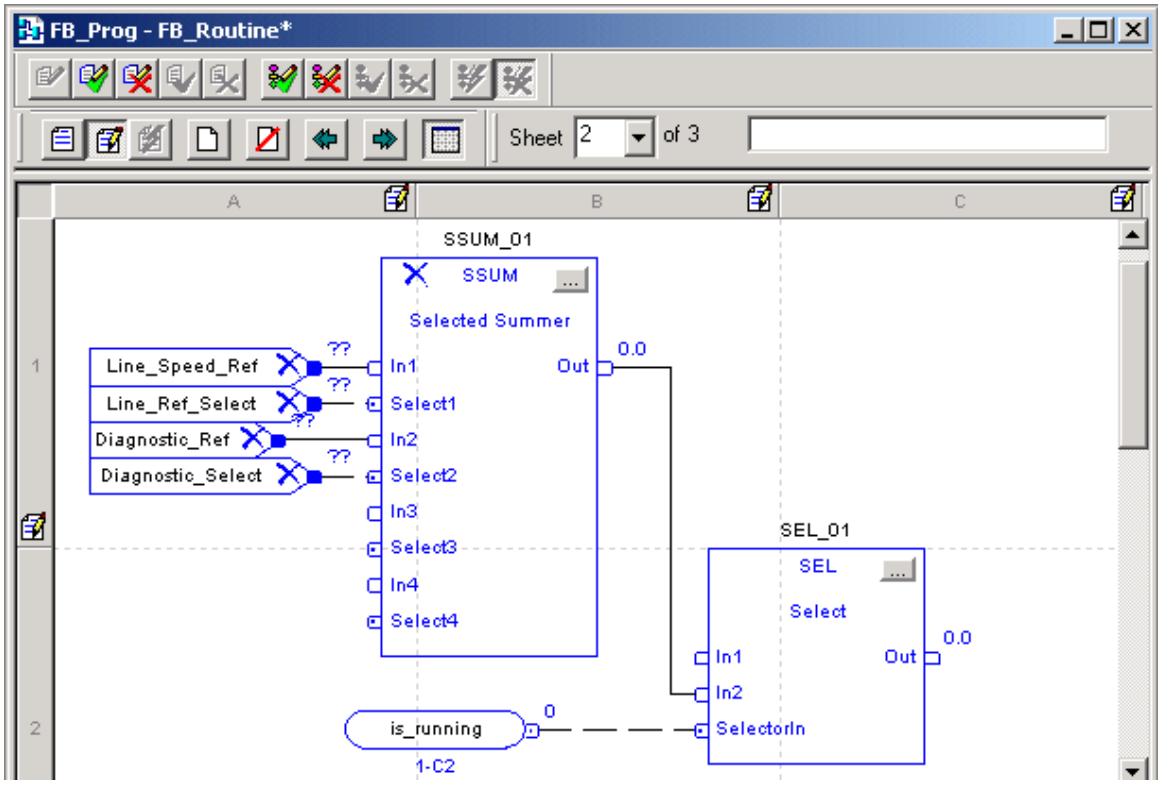
8. Double click on the '?' in the Input Wire Connector and choose 'is_running' from the drop-down.



9. Place four Input References in front of the Selected Summer and wire them as shown.



10. Create the input tags shown in the following by double clicking on each of the tag references (currently question marks) in the tag reference block.



11. Create the input tags by right clicking on each of the tag references and selecting ‘New Tag.” Complete the information in the New Tag Dialog for each tag as follows:

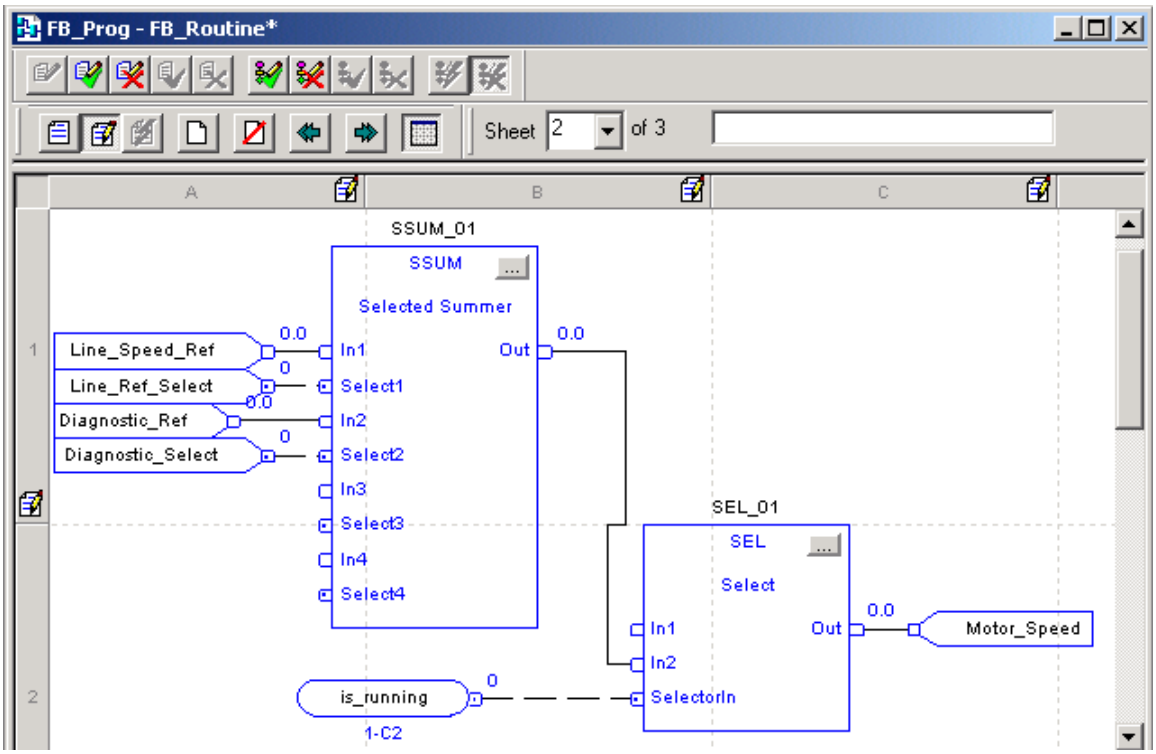
    Line_Speed_Ref REAL

    Line_Ref_Select BOOL

    Diagnostic_Ref REAL

    Diagnostic_Select BOOL

12. Use the same techniques from the last three steps to place an Output Reference, connect it to the output of the Select block, and create the output tag ‘Motor_Speed’ (of type REAL) for the output reference. You should have something similar to the following.

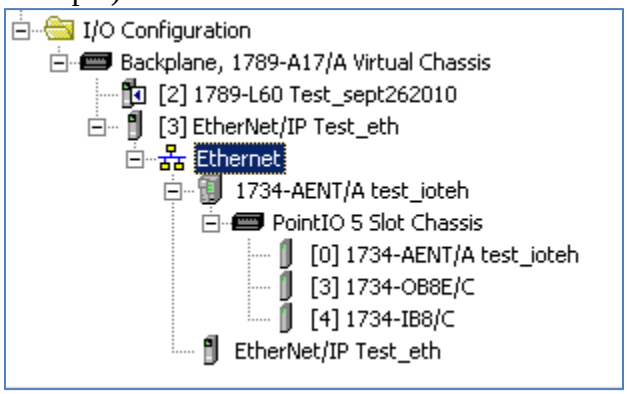


Now, if the motor is being commanded to run (Device1State -> is_running = 1) then the Motor_Speed variable is set to either Line_Speed_Ref or Diagnostic_Ref depending upon which mode is currently selected (Line_Ref_Select or Diagnostic Select).

13. Click once on the ‘Finalize All Edits in Program’ button to verify and accept the edits.

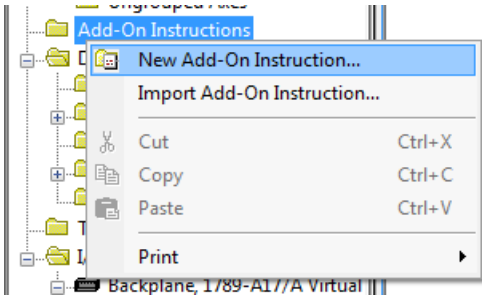
Add-on instructions

Add-ons Instruction (AOI) are use to create commonly use set of instruction and being able to apply them to many components without having to re-write them over and over again in a same program.
The Lab will show you for example how to start and stop 4 motor (simulated with the button and light on your control panel)
We will use the same button to start and stop the motor, a timer to simulate the starting time and one output to indicate that the motor is running.
First set-up your Softlogix controller and the Point I/O Digital input and output (Analog are not used in this example)

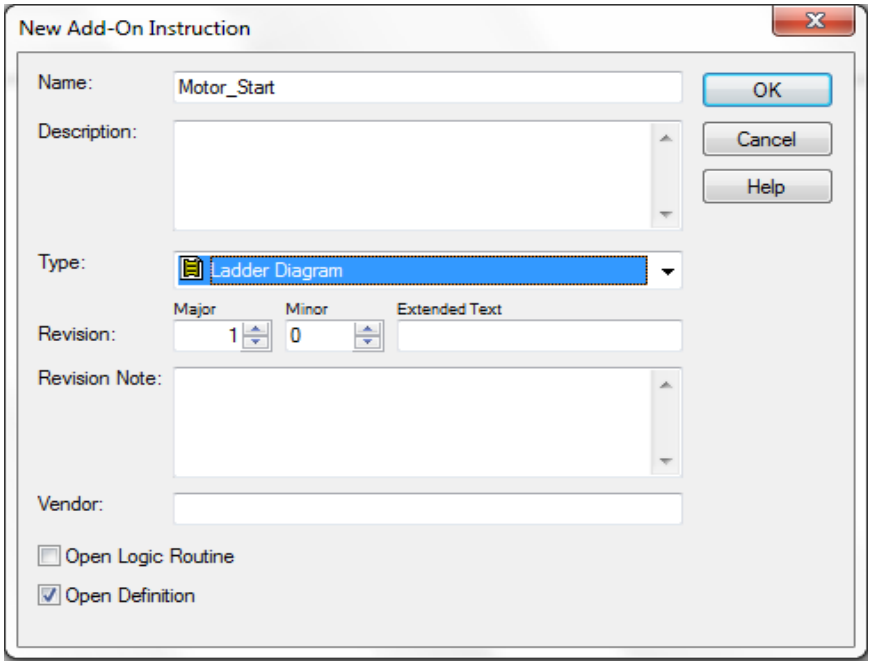


Create the Add-ons program

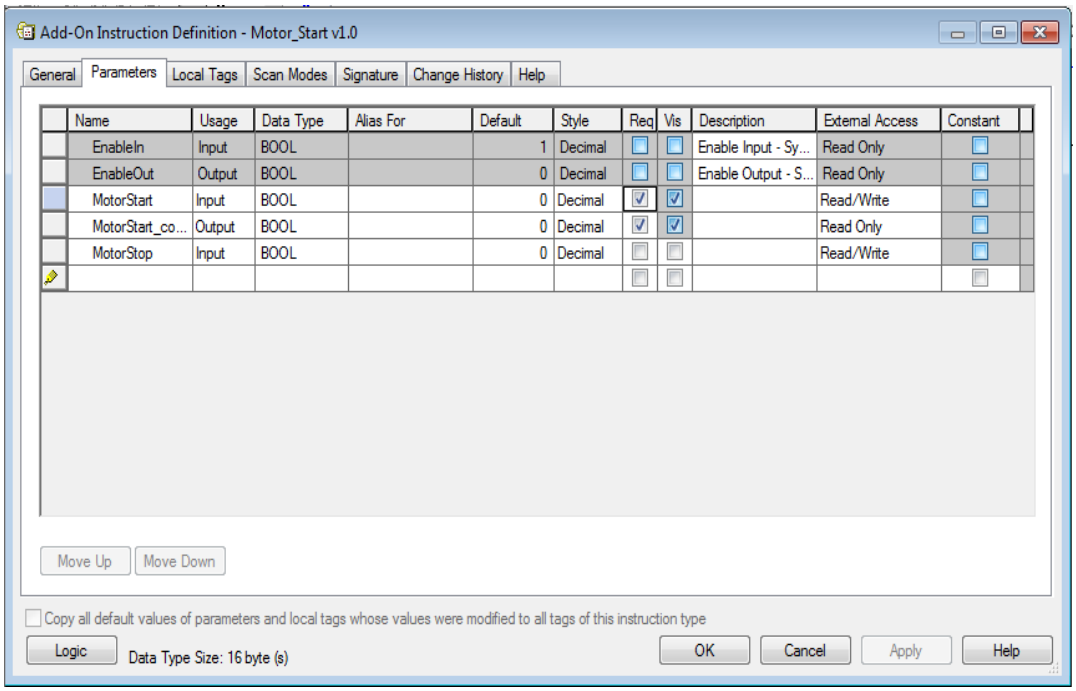
1. Right-click on the **Add-on instruction** folder
2. Select **New Add-on Instruction**



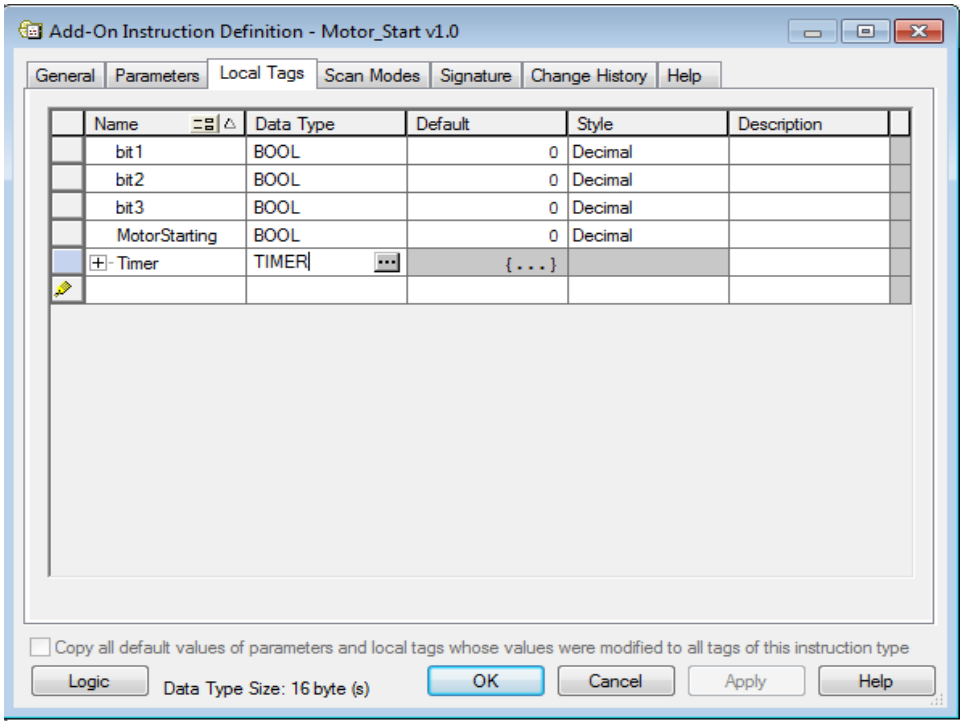
3. Give a name to your Add-on instruction, **Motor_start** for example
4. Make sure the type is set to **Ladder Diagram** for our example but you can use Structure text or Sequential function chart if you want.
5. Make sure that you check the **Open definition** box



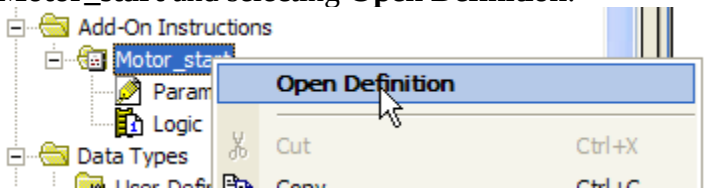
After you press OK the definition window will pop up.
Click on the Parameters tab. You will see two defaults tags, the **EnableIn** and **EnableOut** they can be used but not modified
Create the tags we will need in our example as shown below.
The **Req** column is to tell the controller which tags are to be used in other routine
The **Vis** column is to tell the controller which tags are to be visible in other routine



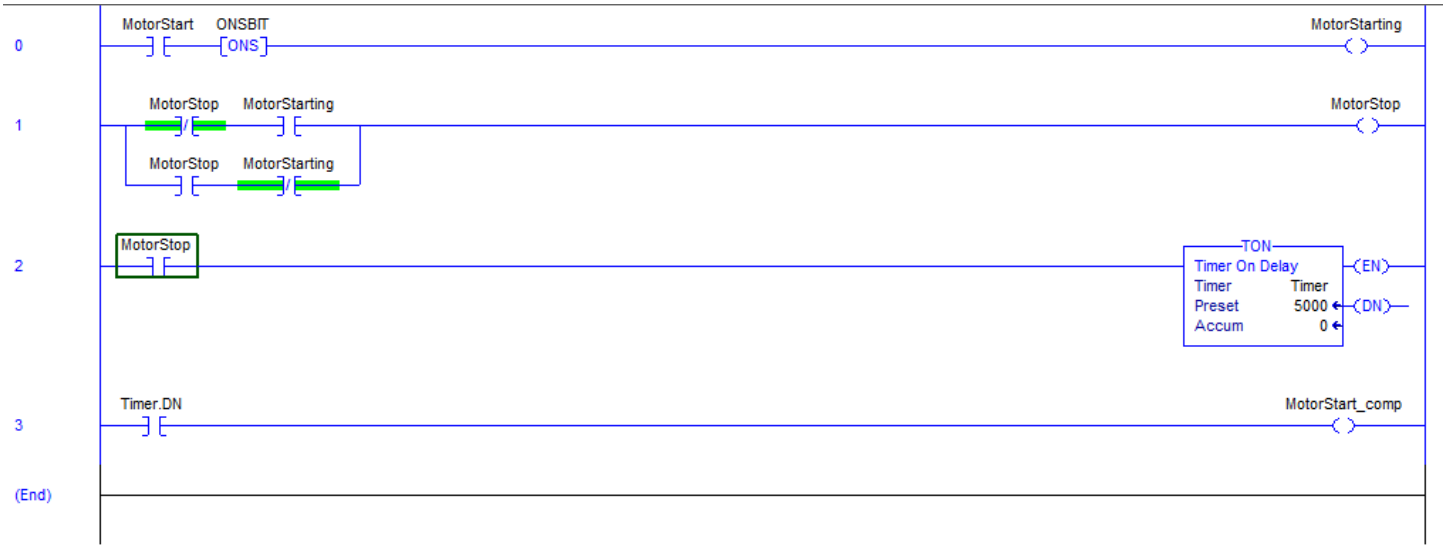
Then click on **Local Tags** tab, this section is used to create tags that are only use in the AOI and don't need to be used or seen in other routine
Create the tags as shown below.



In case you need to go back to the definition to make some modification you can by right-clicking on **Motor_start** and selecting **Open Definition**.

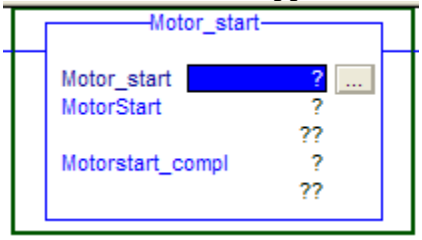


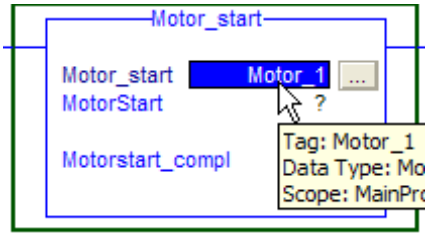
Now that the tags are created we will start to write the code in the AOI, we will use one of the buttons on the point I/O board to turn on the corresponding light after 5 second which will simulate the starting of the motor. Below is the code to perform this task



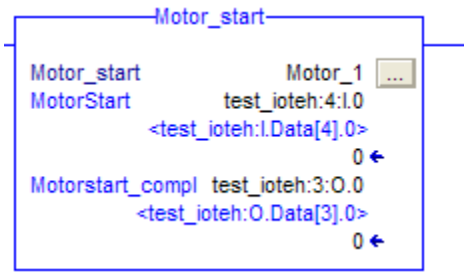
Now that we have created the AOI we need to call it up in the main routine and assign the button and the light. Double-click on the **MainRoutine** and create the program as shown below. You will find your AOI by clicking on the **Add-On** tab selecting **Motor_start** in the instruction toolbar.

The instruction will appear on the selected rung



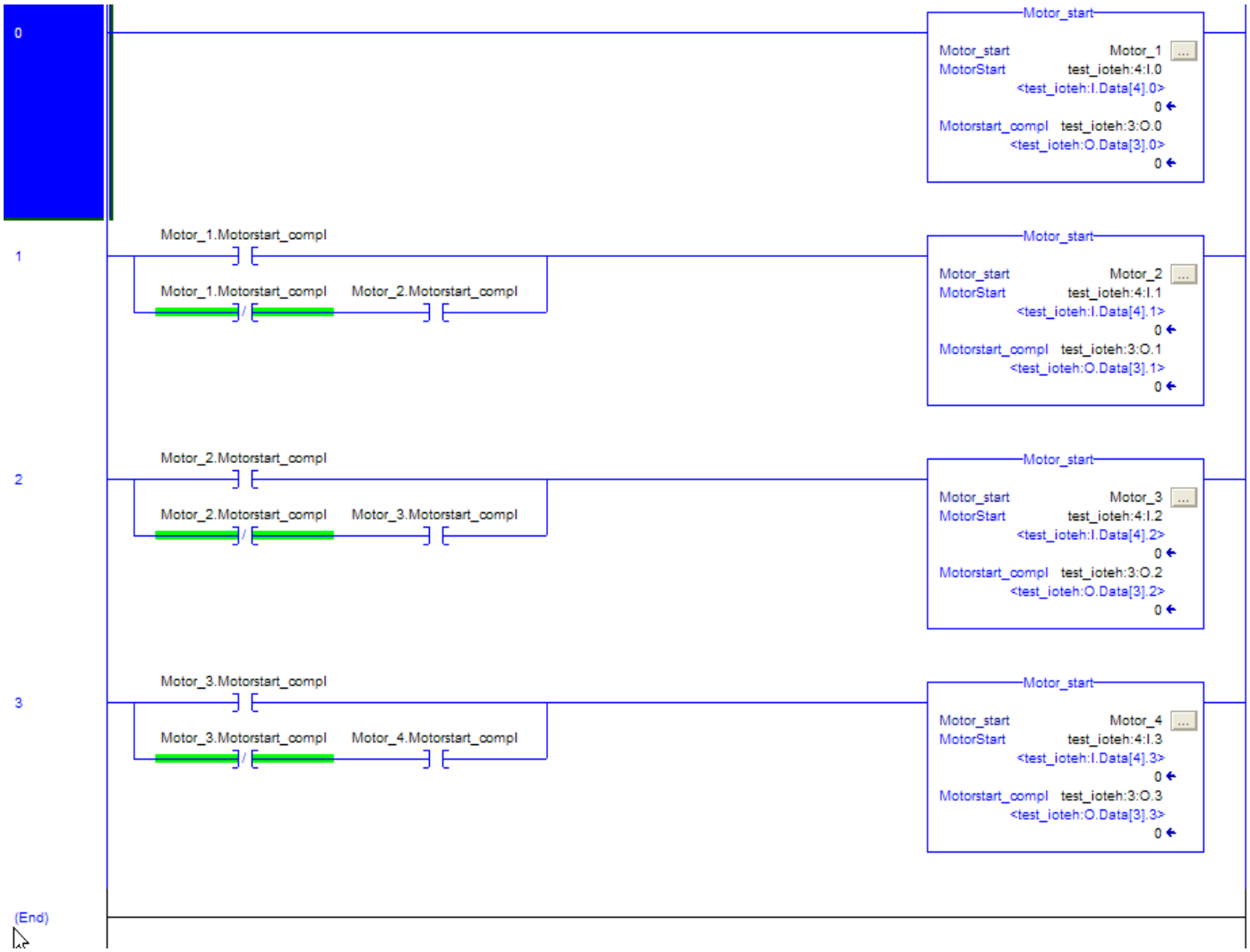


Give a name to your instruction like **Motor_1**
Right click on Motor_1 and create new alias of type Motor_Start.
To assign the button to start and stop the AOI click on the **?** for **MotorStart**
We will use the first button on your I/O board
We will assign as well the first light of your point I/O board to Motorstart_compl by click on the **?**
Your instruction should appear as shown below

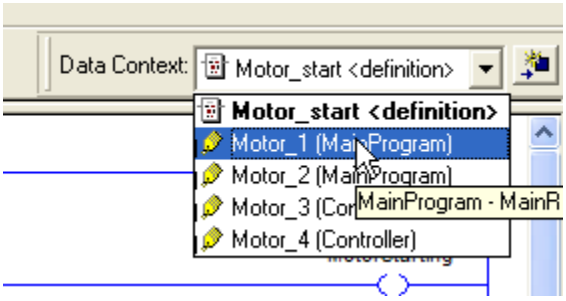


Using the same procedure create one instruction for each button and corresponding light on your point I/O board.
As we will add few instructions in front of some AOI so “motor 2” can only start if “motor 1” has completed its start-up, and so on…

Your routine should appear as shown below



Test your program, by pressing the first button the first light should come up after 5 second, if you press any other button before the light come on the AOI will not start.
To monitor your AOI, go back to your AOI routine and click on the down arrow on the **Data Context** menu



You can choose which AOI you want to monitor

Extra task: Create a parameter to set a different preset time in your AOI. Set the first timer at 1 sec, the second one at 3 second, the third on at 5 second and the last one at 2.5 second, use the **MOV** instruction to help you.

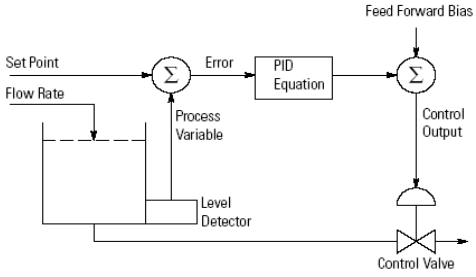
Proportional/Integral/Derivative Control (PIDE)

The PID instruction is an output instruction that controls physical properties such as temperature, pressure, liquid level, or flow rate using process loops.

PID Concept

- The PID instruction normally controls a closed loop using inputs from an analog input module and providing an output to an analog output module. For temperature control, you can convert the analog output to a time proportioning on/off output for driving a heater or cooling unit.

PID closed loop control holds a process variable at a desired set point. A flow rate/fluid level example is shown below.

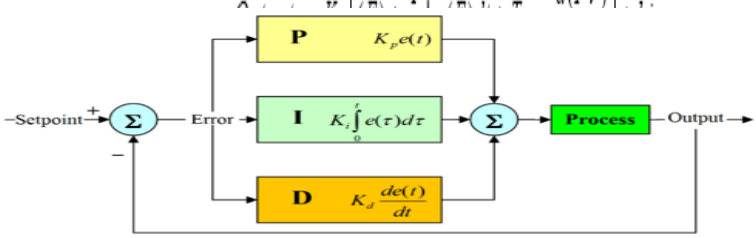


PID Control

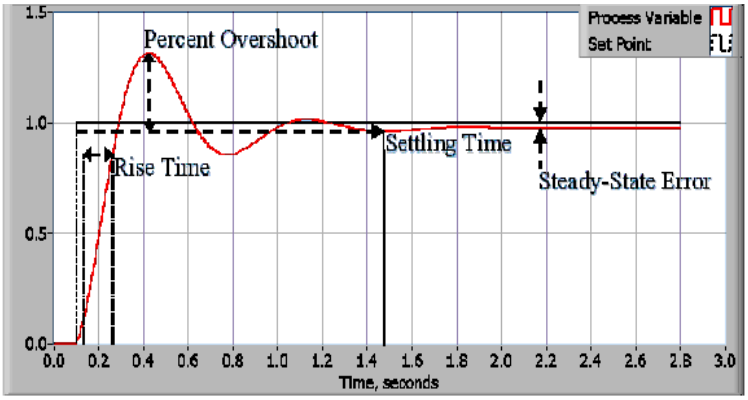
- The PID equation controls the process by sending an output signal to the control valve. The greater the error between the setpoint and process variable input, the greater the output signal. Alternately, the smaller the error, the smaller the output signals. An additional value (feed forward or bias) can be added to the control output as an offset. The PID result (control variable) drives the process variable toward the set point.

The PID instruction uses the following algorithm:

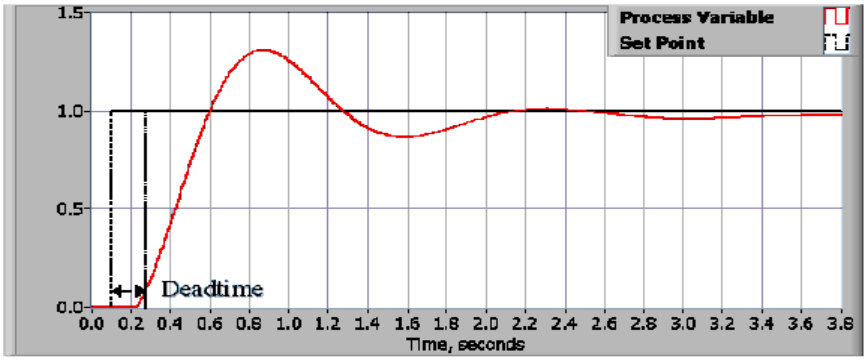
Standard equation with dependent gains:



Control system performance is often measured by applying a step function as the set point command variable, and then measuring the response of the process variable. Commonly, the response is quantified by measuring defined waveform characteristics. Rise Time is the amount of time the system takes to go from 10% to 90% of the steady-state, or final, value. Percent Overshoot is the amount that the process variable overshoots the final value, expressed as a percentage of the final value. Settling time is the time required for the process variable to settle to within a certain percentage (commonly 5%) of the final value. Steady-State Error is the final difference between the process variable and set point.



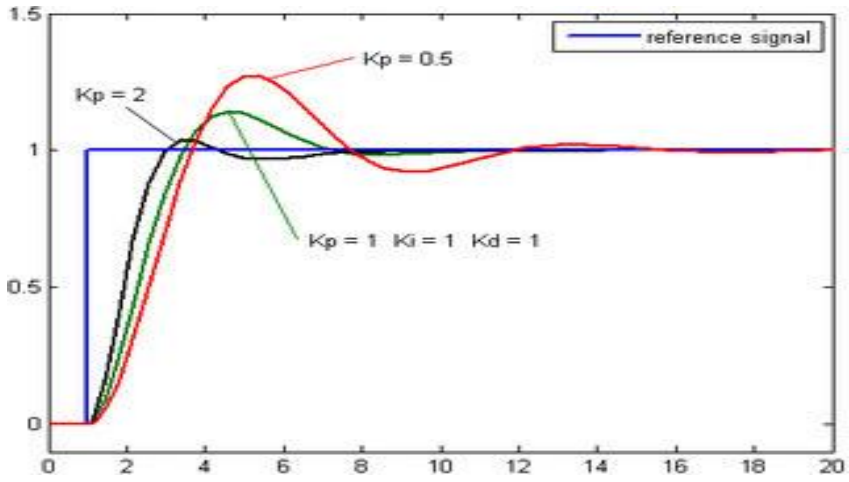
Some systems exhibit an undesirable behavior called *Deadtime*. Deadtime is a delay between when a process variable changes, and when that change can be observed. For instance, if a temperature sensor is placed far away from a cold water fluid inlet valve, it will not measure a change in temperature immediately if the valve is opened or closed. Deadtime can also be caused by a system or output actuator that is slow to respond to the control command, for instance, a valve that is slow to open or close. A common source of deadtime in chemical plants is the delay caused by the flow of fluid through pipes.



Proportional Response

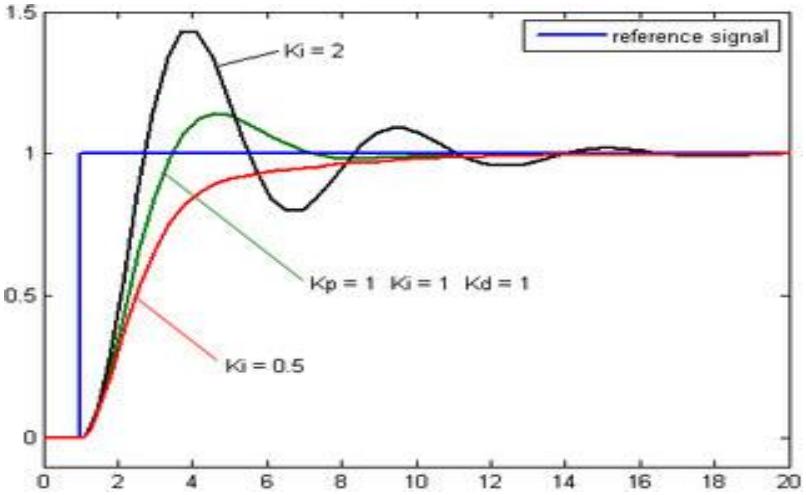
The proportional component depends only on the difference between the set point and the process variable. This difference is referred to as the Error term. The *proportional gain (Kc)* determines the ratio of output response to the error signal. For instance, if the error term has a magnitude of 10, a proportional gain of 5 would produce a proportional response of 50. In general, increasing the proportional gain will increase the speed of the control

system response. However, if the proportional gain is too large, the process variable will begin to oscillate. If Kc is increased further, the oscillations will become larger and the system will become unstable and may even oscillate out of control.



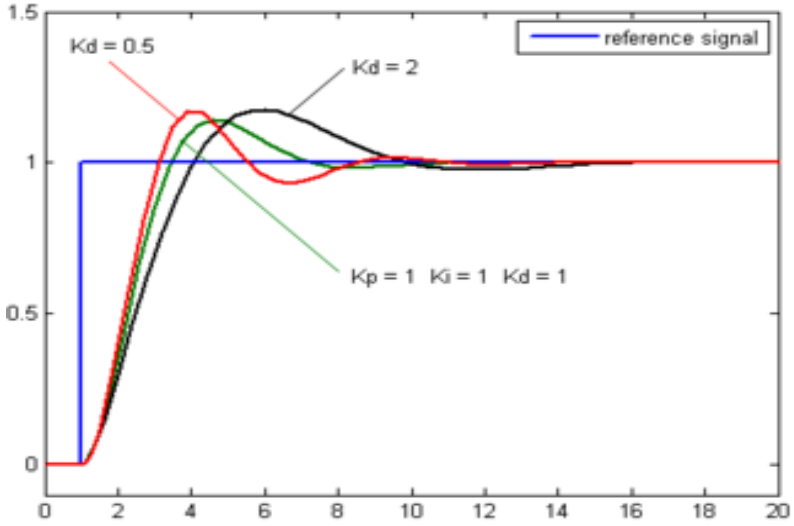
Integral Response

The integral component sums the error term over time. The result is that even a small error term will cause the integral component to increase slowly. The integral response will continually increase over time unless the error is zero, so the effect is to drive the Steady-State error to zero. Steady-State error is the final difference between the process variable and set point. A phenomenon called integral windup results when integral action saturates a controller without the controller driving the error signal toward zero.



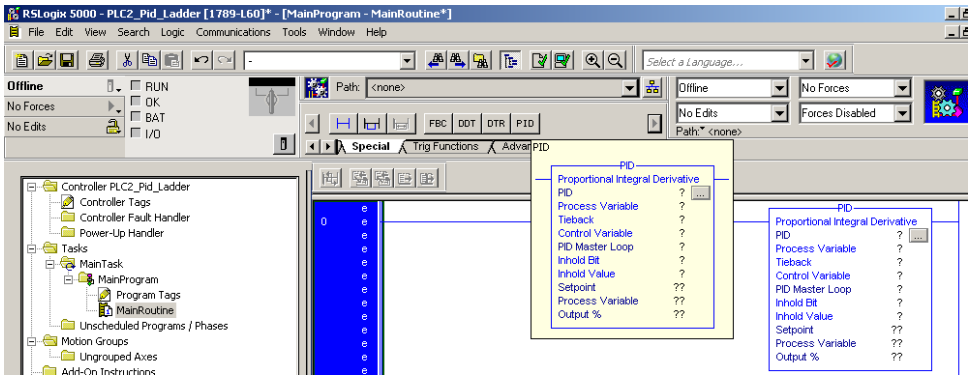
Derivative Response

The derivative component causes the output to decrease if the process variable is increasing rapidly. The derivative response is proportional to the rate of change of the process variable. Increasing the *derivative time (Td)* parameter will cause the control system to react more strongly to changes in the error term and will increase the speed of the overall control system response. Most practical control systems use very small derivative time (Td), because the Derivative Response is highly sensitive to noise in the process variable signal. If the sensor feedback signal is noisy or if the control loop rate is too slow, the derivative response can make the control system unstable

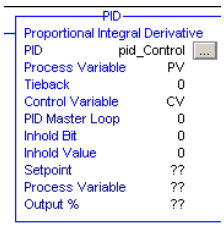


Configuring and Manually Tuning the Ladder Logic PID Instruction

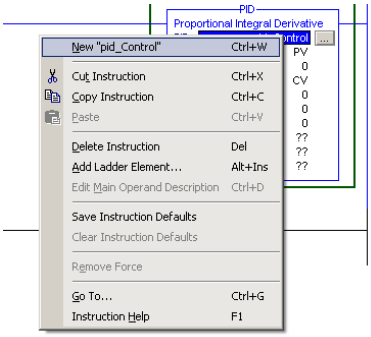
Open a Studio 5000 project, drag and Drop a PID instruction from the Special instruction tab into a rung in the main routine.



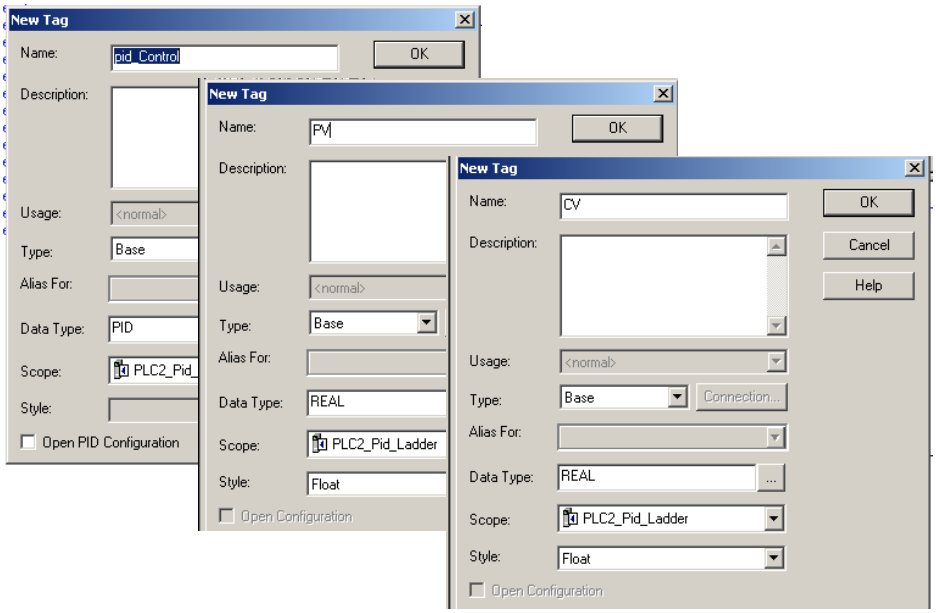
Enter “PID_Control” as tag for the PID Parameter. Enter the”PV” and “CV” tag in Process Variable and Control Variable Parameters.



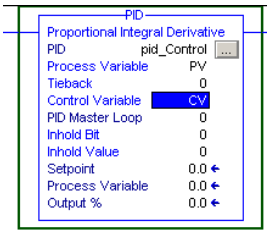
Right click PID_Control and initialize the tag by selecting new PID_Control



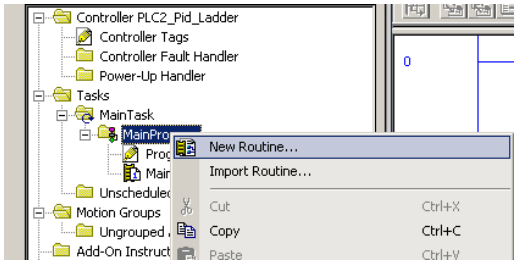
Notice the Data Type for PID_Contol Is PID. Initialize the CV and PV tags As real data type, if not the instruction will not work.



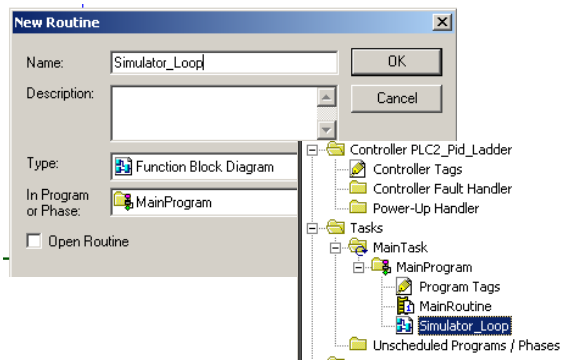
The tieback, PID Master Loop Inhold Bit, and Inhold Value Parameters should all have 0 values



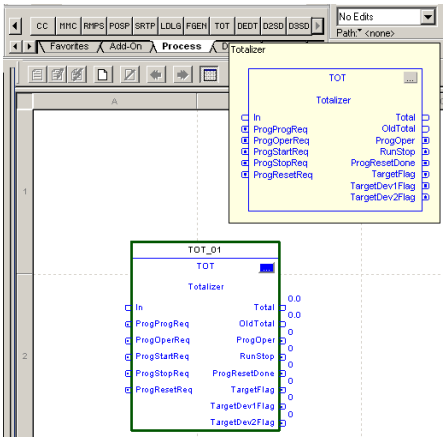
Right click the Main Program folder and select new routine



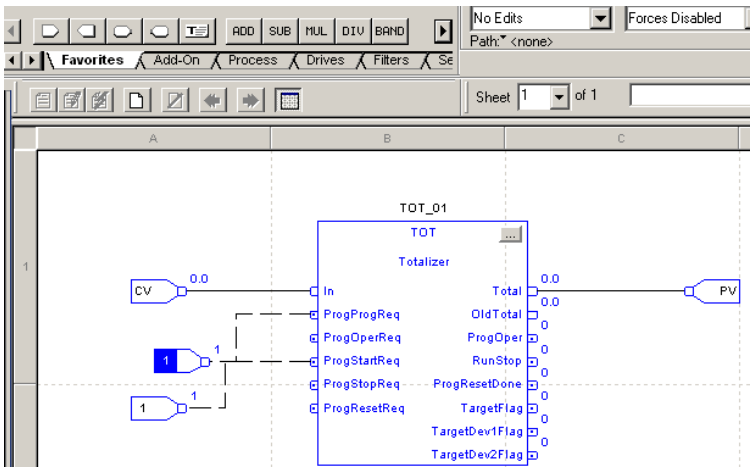
Enter Simulator Loop as the routine name. Click on The Type drop down arrow, and Select Function Block Diagram as the programming language you will use for this routine



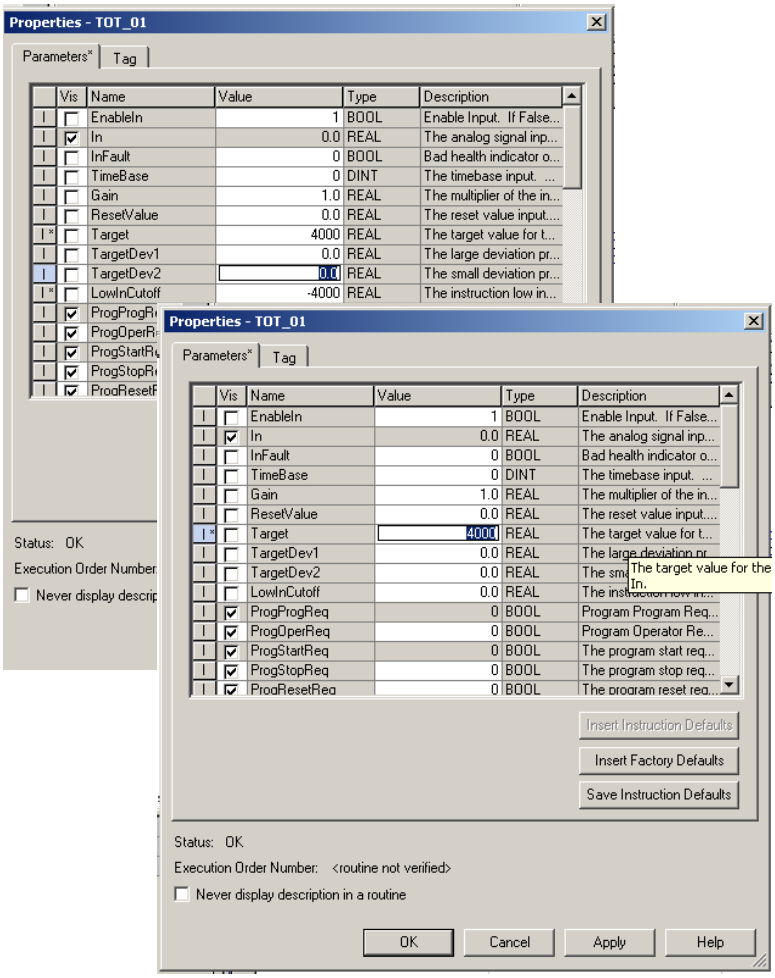
From the Process Tab drag and drop TOT (Totalizer) Instruction into the routine Sheet.



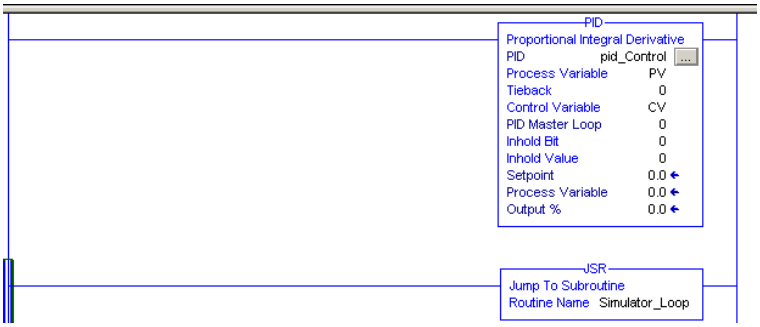
Create a cv input reference connected to In on The totalizer and create a PV output reference connected to total on the toalizer instruction. Create 2 input references with the value of on connected to the ProgProgReq and ProgStartReq.



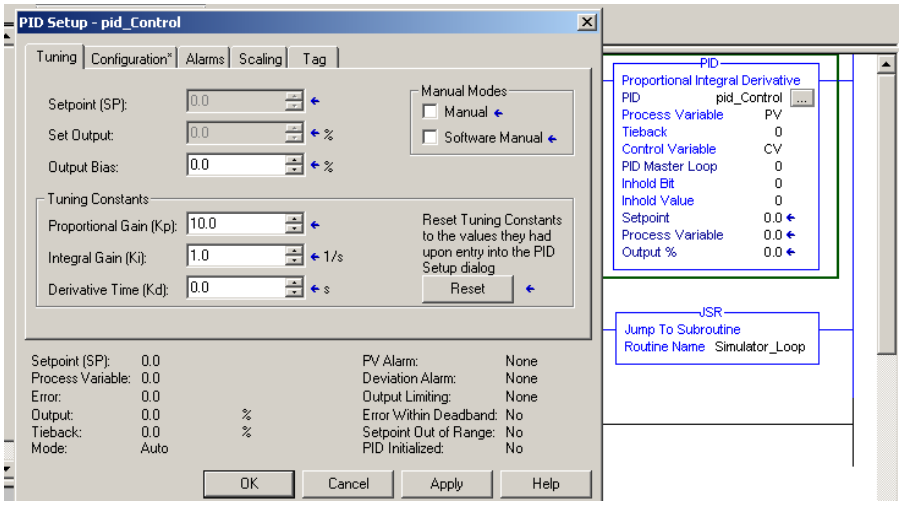
Double click on totolizer elispe on upper right Hand corner of the Totalizer instruction.
In the Parametrs tab ,scroll down and Change the LowInCutoff value to -4000 and the Target value to 4000.



Go back to the main routine and add in a JSR Instruction. From drop down list under Routine Name Select Simulator_Loop.

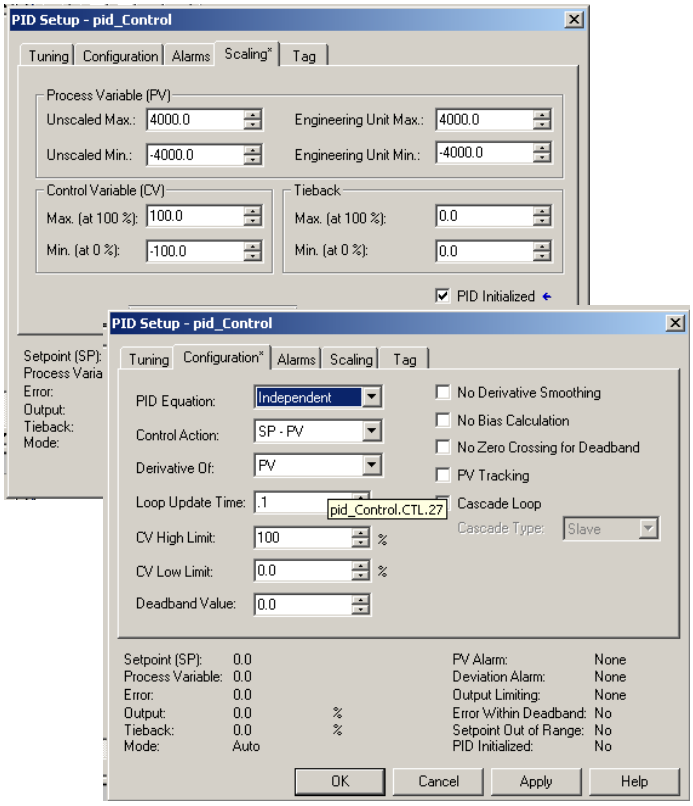


Double click on the ellipse in the upper Right corner of PID instruction. Go the Tuning tab enter the following
Proportion Gain (Kp) = 10
Integral Gain (Ki) =1

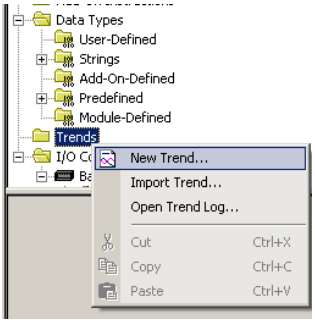


Derivative tTme (Kd) =0
Go the Scaling tab enter the following:
Unscaled / Eng Max PV = 4000
Unscaled / Eng Min PV = -4000
CV Max at 100% = 100
CV Max at 0% = -100
What would 50% be? ____

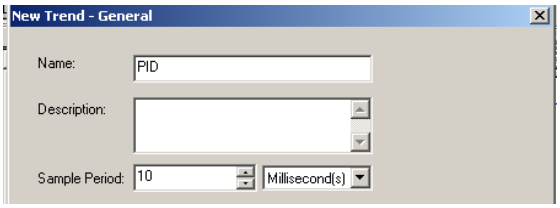
Go the Configuration tab enter the following:
CV high Limit (PID Max not CV max) = 100
CV Low Limit = -100
Loop Update = 0.1



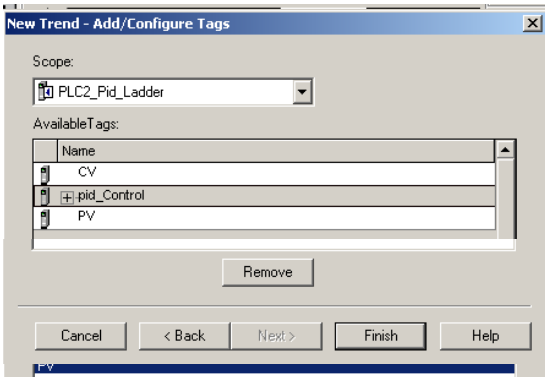
In the Project Explorer Right click the trend folder and select New Trend.



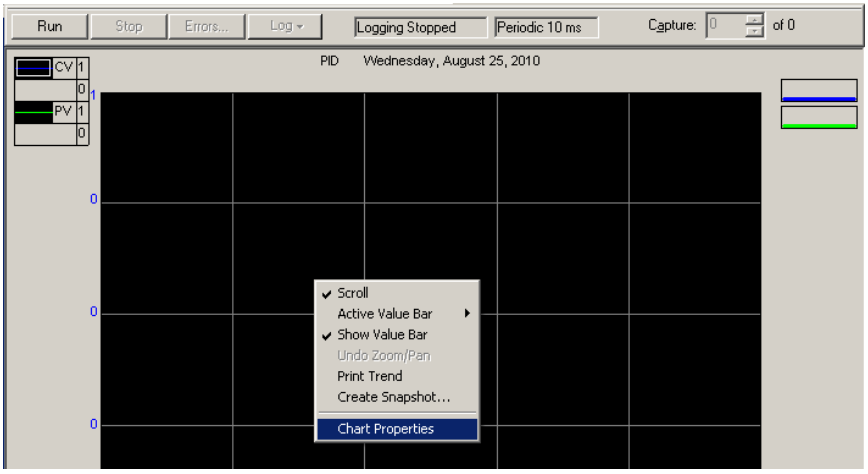
Enter “PID” for the Trend Name. Click next.



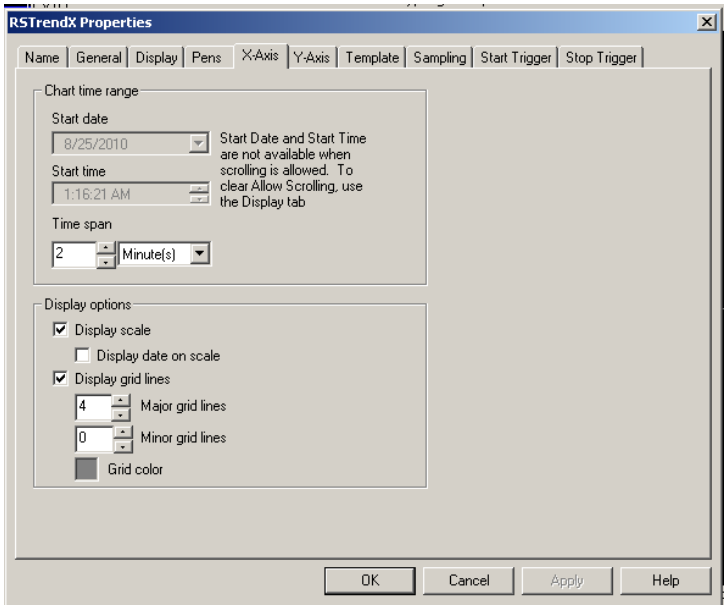
In the Avaialble tag section Select CV and click on Add Button. Do this for PV as well. Press the finish button.



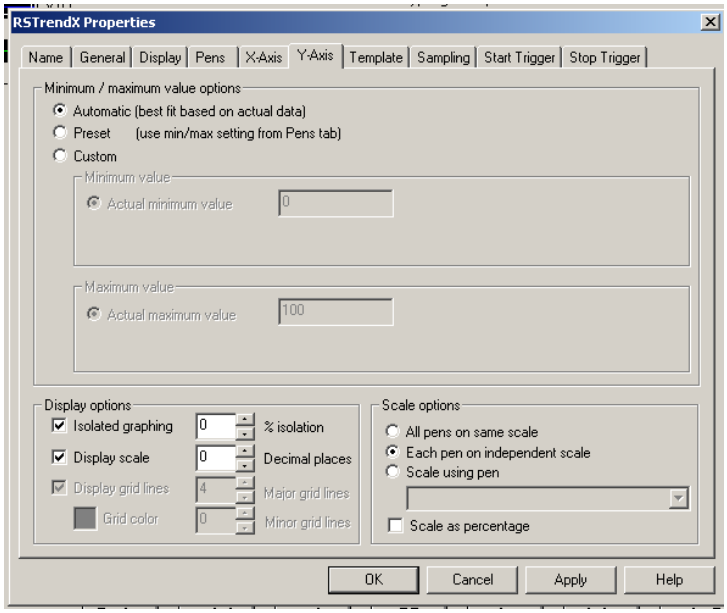
Right Click in an empty space in the Trend And Select Chart Properties.



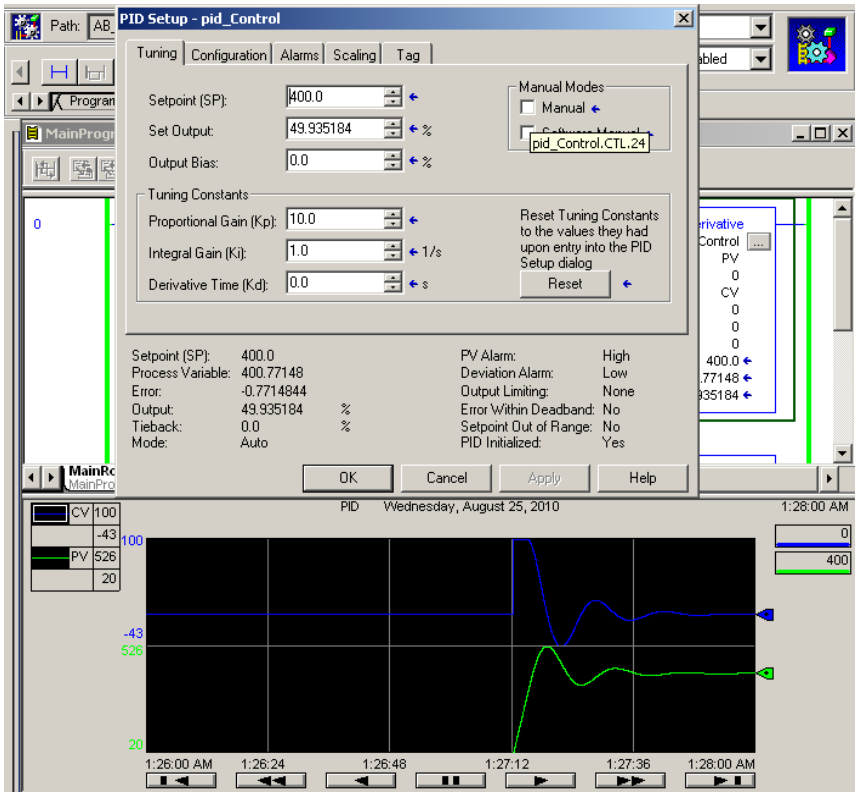
Select the X-Axis Tab. Change the Time Span To 2 minutes



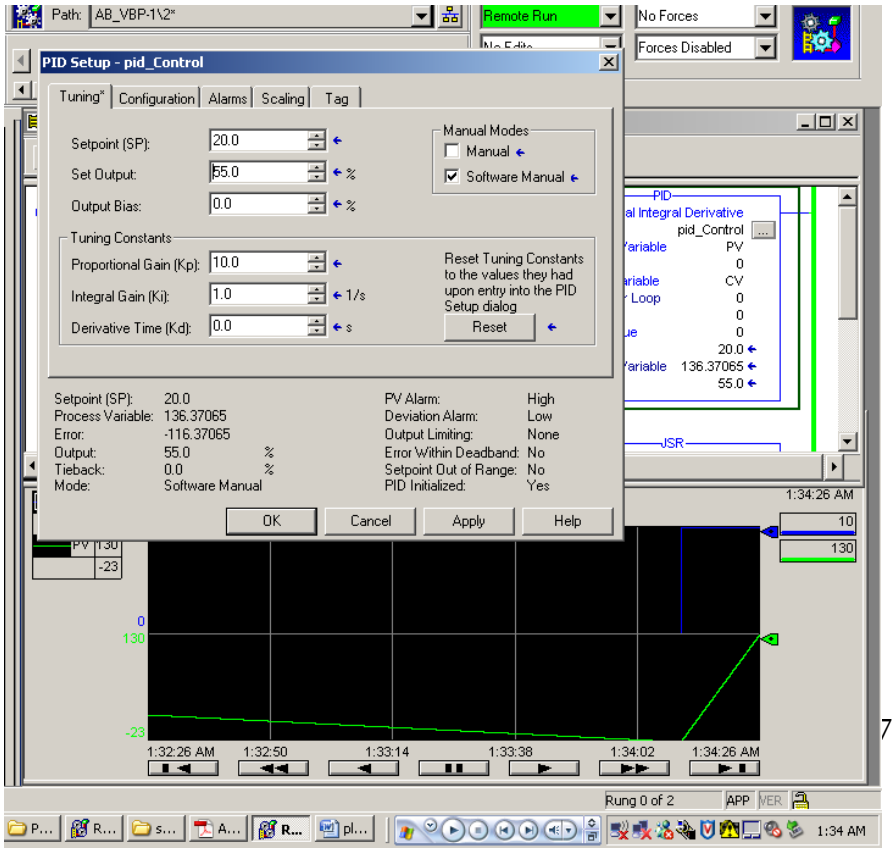
Select the Y-Axis. Under the Display Options Put a checkmark in the isolated graphing box



Down load the program. Goto Ladder Program an click ellipse on the PID Instruction to open the PID set screen. Change the setpoint to 400. Observe response.
The other way to change the setpoint is to double click the SP value on the PID instruction, change the value and press enter.

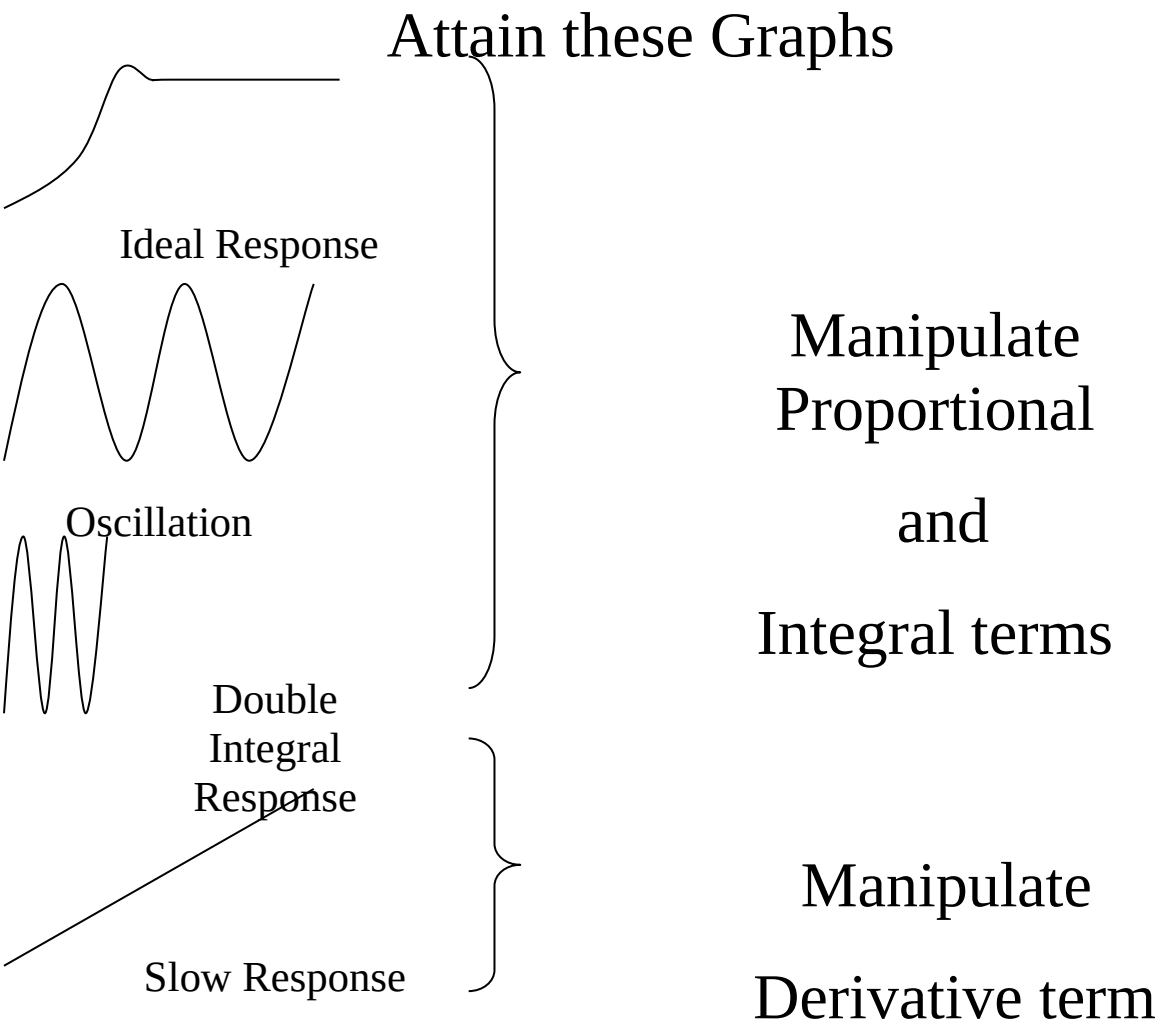


In the manual Modes section put a Check mark in the software manual box. On the Set Output parameter Use the up and down to get above 50%. This will allow you to manually Control CV from the Software.



On-line trial tuning

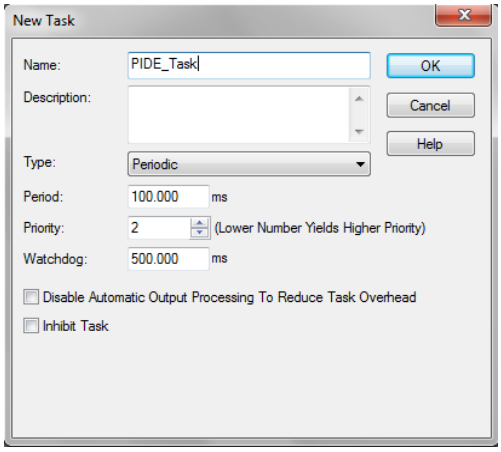
1. Enter an initial set of tuning constants from experience. A conservative setting would be a gain of 1 or less and a reset of less than 0.1.
2. Put loop in automatic with process "lined out".
3. Make step changes (about 5%) in setpoint.
4. Compare response with diagrams and adjust.



Write down the Proportional, Integral and derivate term for each response.

Configuring Function block PIDE with autotune

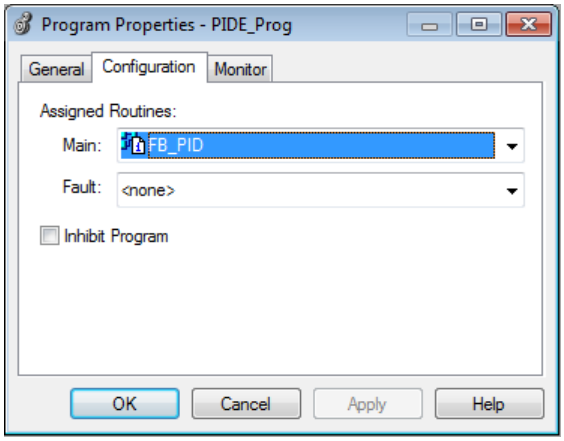
Open up Studio 5000 and create an application and name it PIDE.
Right click the Tasks and select New Task.



In the new Task window name the Task “PIDE_Task” and pull the type drop down menu and select periodic. Enter 100ms for the period and press OK.

Right click on newly created task, Create a new Program name it “PIDE_Prog” and inside this Program’s tree create a new routine and name it “FB_PID”

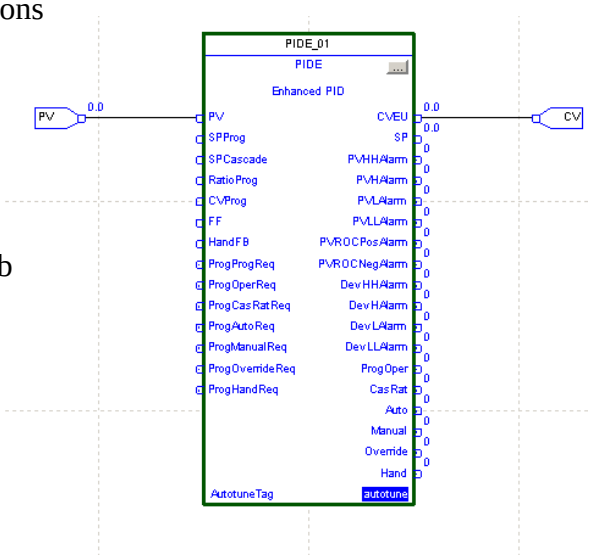
Right click the “PIDE_Prog” Program folder and select Properties. Under Configuration select FB_PID as the Main routine.



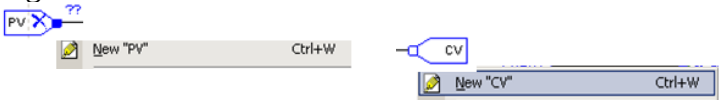
ON sheet 1 of the FB_PID routine insert the following instructions

The PIDE instruction can be found in the process tab

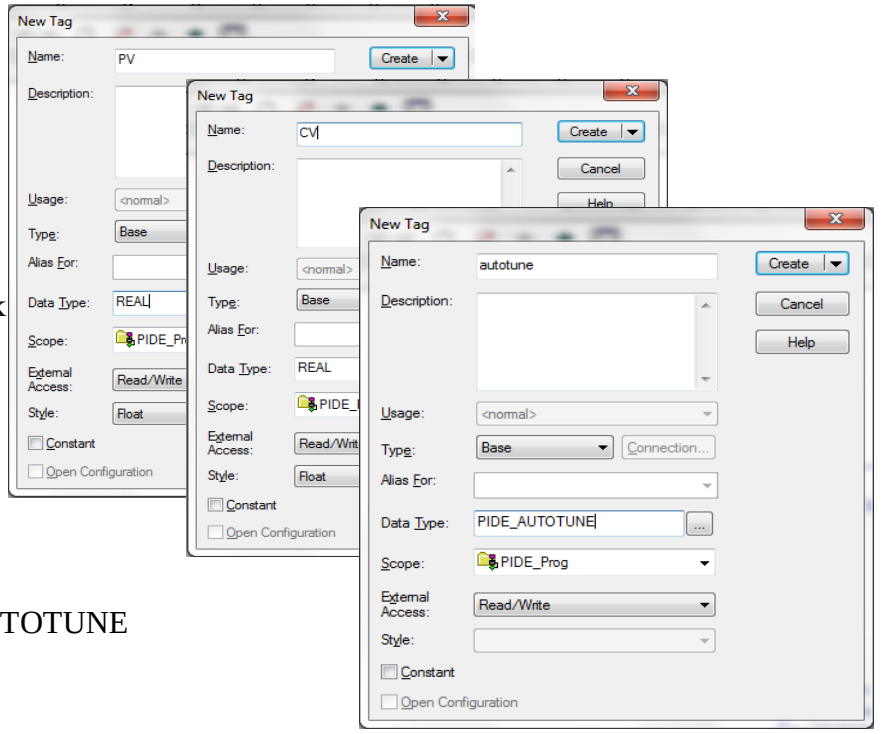
and The Input and output reference can be found in favorites tab



Right Click the CV and PV select New PV and CV.



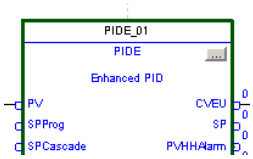
Both PC and CV should be of the Real Data Type.



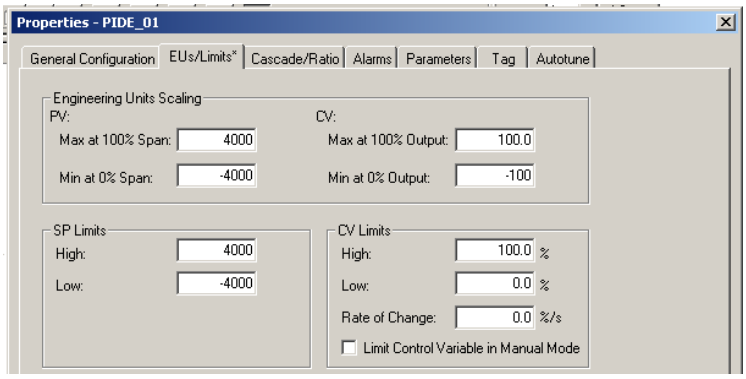
In the bottom right hand corner of PIDE instruction opposite the Autotune Tag Label insert a tag called ”autotune” .Right Click Autune and select New Autotune.

The tag should be of the data type of PIDE_AUTOTUNE

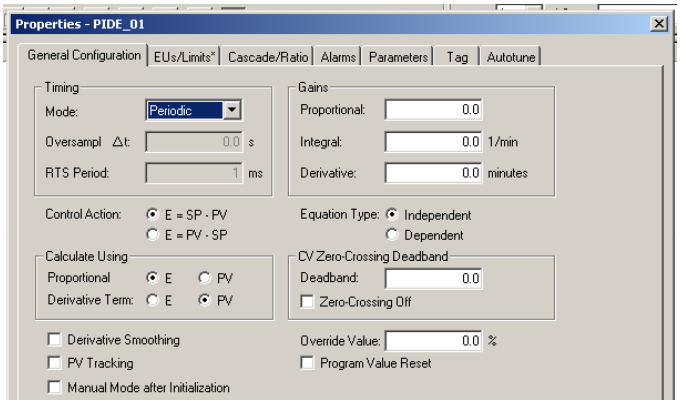
Click the ellipse in the top right hand corner in order To Set the PIDE Properties



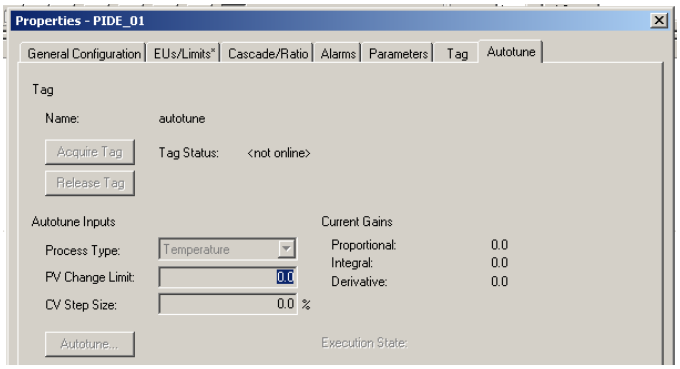
On the EUs/Limits tap enter the following values.



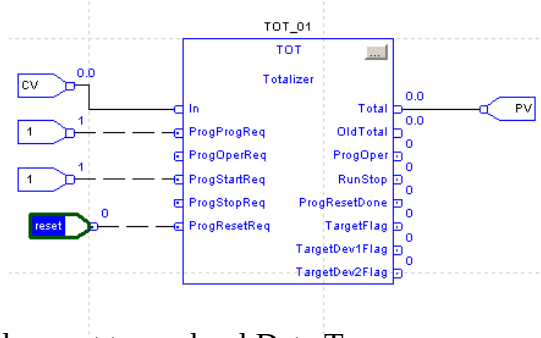
The General Configuration tab will allow you To manually set the Gains and various other Parameters.

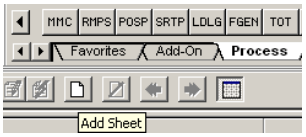


One of the most important issues in any control system is tuning the controller's parameters. Considering different parameters and the permutation of values, there is unlimited number of states. There are several algorithms to determine optimal values for these parameters.
The Autotune tab allows the instruction to tune your process loop. After completing the program this tab will be used to tune the P,I and D parameters.

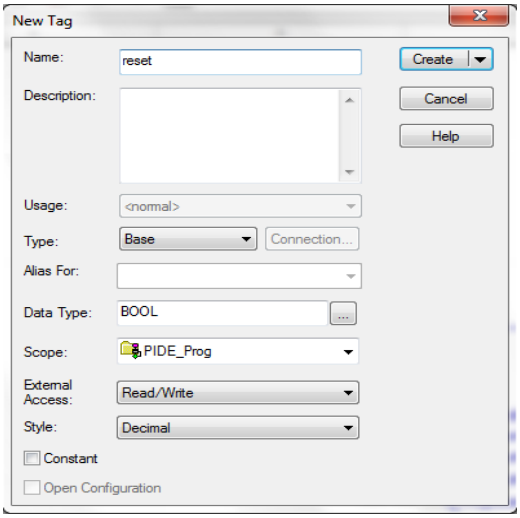


Create a new sheet and from the process menu add a TOT (Totalizer) instruction as shown below.

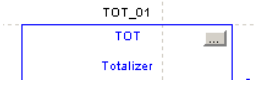




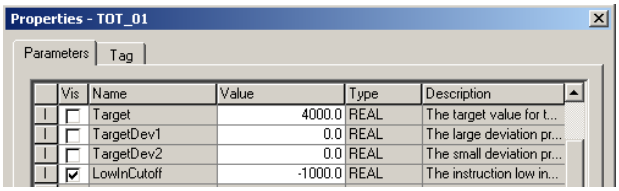
Initialize the reset tag as bool Data Type



Click the ellipse in the top right hand corner

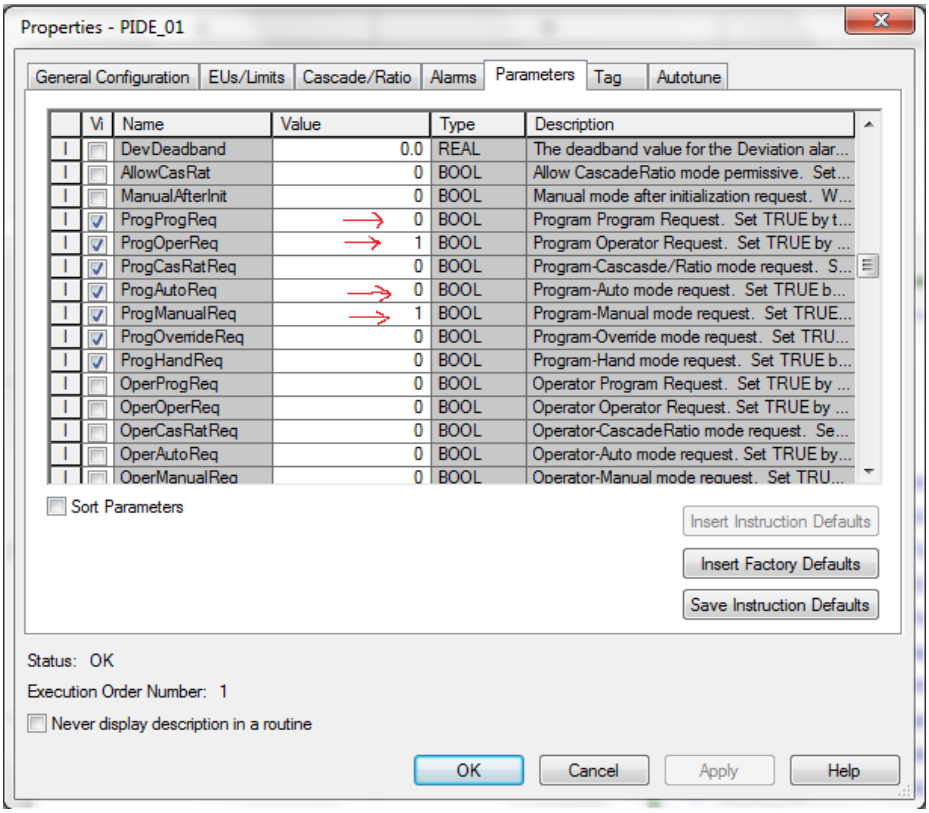


Set the Target parameter to 4000 and the LowinCutoff -1000



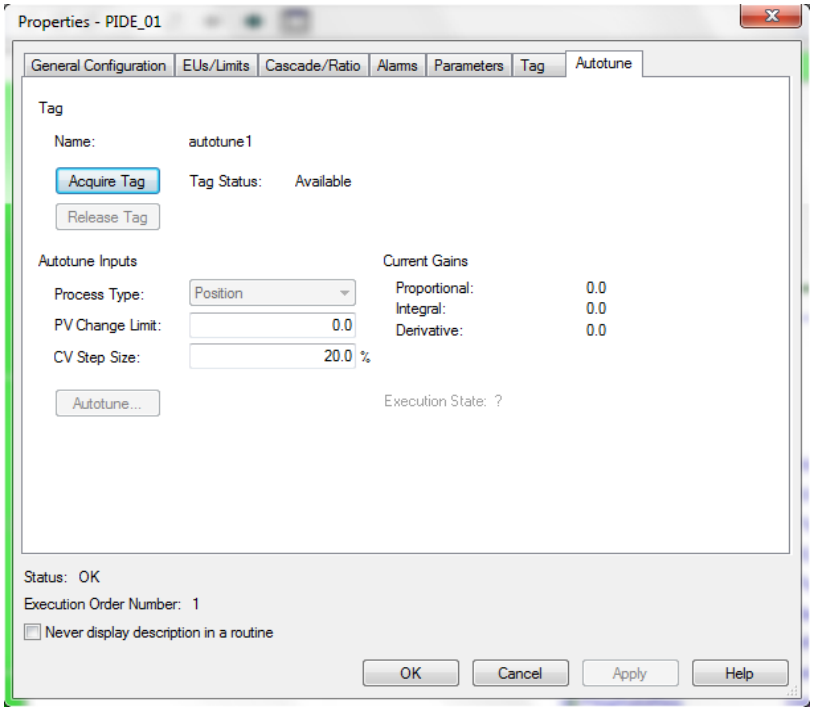
In order to run the autune the PIDE instruction must be in "**Operator Mode"** and "**Manual".** Otherwise when the autune is run there will be a error message which shows wrong mode is selected.

Click on ellipse on the right top of PIDE instruction and goto parameters tab. Set the following parameters according to the following snapshot.

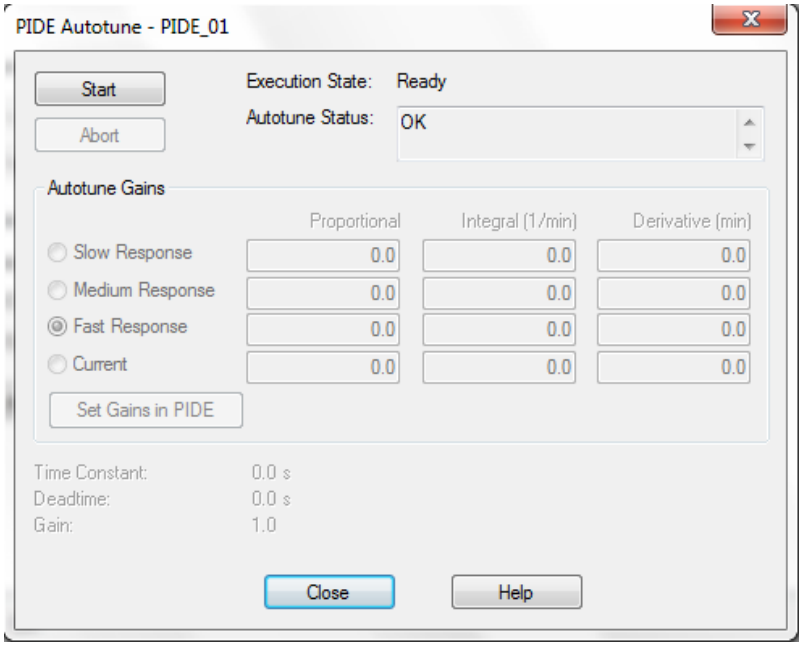


Download the program into softlogix(Or any PLC in the LAB) and set the PLC into RUN mode.(At this stage you must be able to do it).
Get the PIDE propertise dialog on the screen and goto Autotune tab.

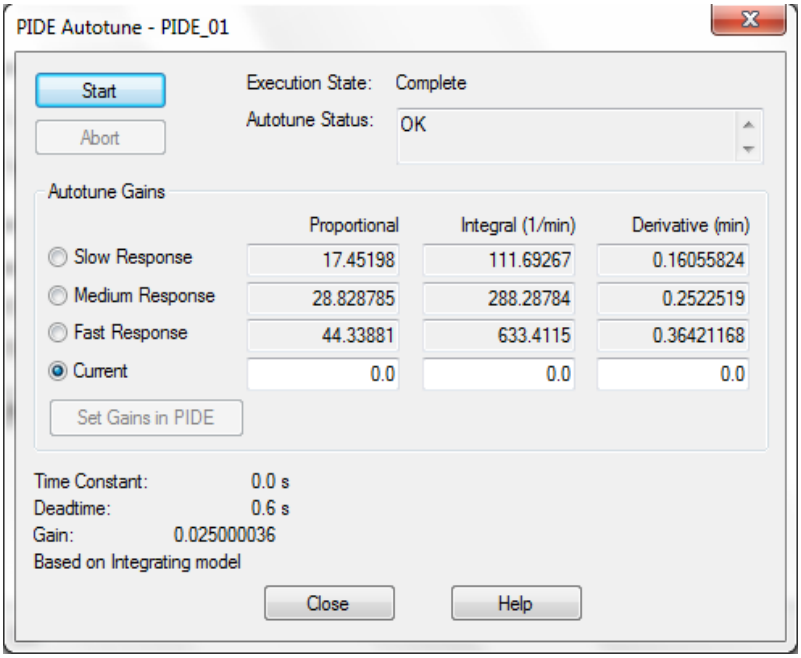
Press "**Acquire Tag".** For the process Type Select Position(You can select other Types but depending on the algorythm that Autotuner selects it will take a long time to tune the parameters). Finally click the **"Autotune"** button.



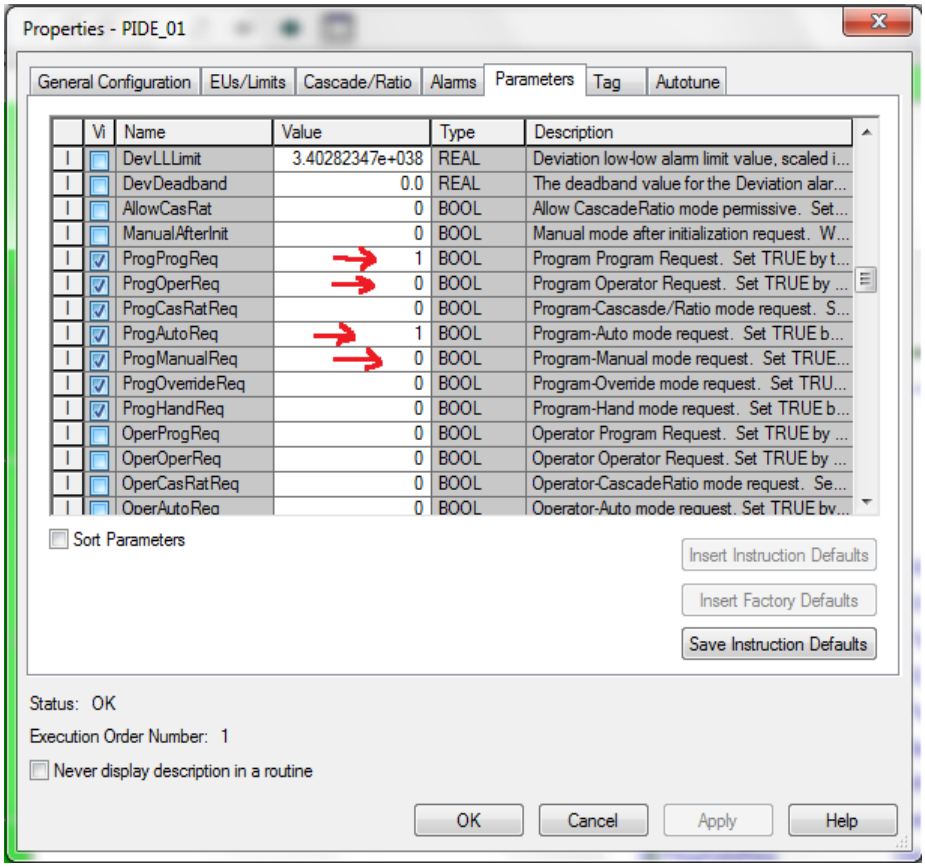
Press Start and wait until the **Complete** is displayed in front of **Execution State**.



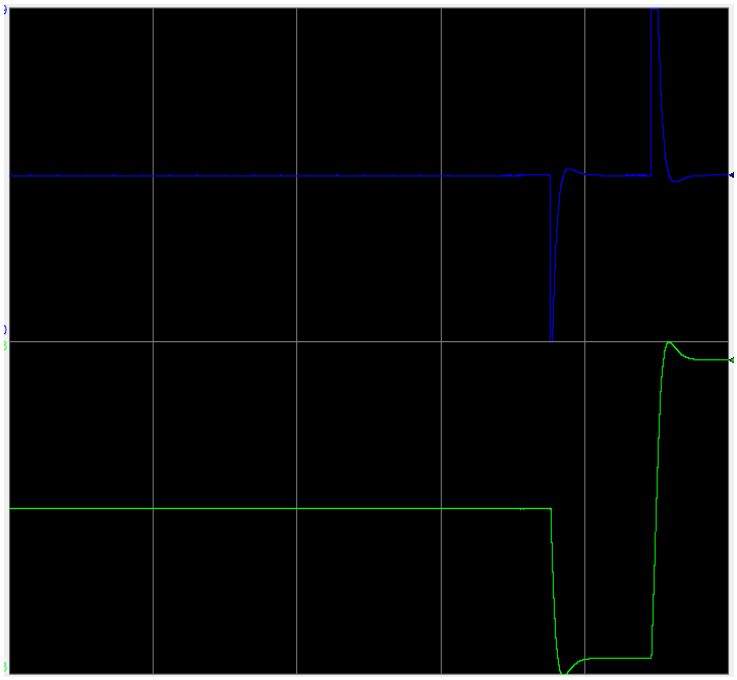
As you see the Autotuner has calculated Proportional, Integral and Derivative coefficients. There are three choises: Slow, Medium and Fast responses that you can select. Select Slow Response and press "**Set Gains in PIDE**". By doing that the calculated values will be substituted in place of Current values. Close the PIDE Autotune dialog and goto Parameters tab.



Enter 200 for SpProg and set the PIDE into Auto Prog mode according to the following snapshot.



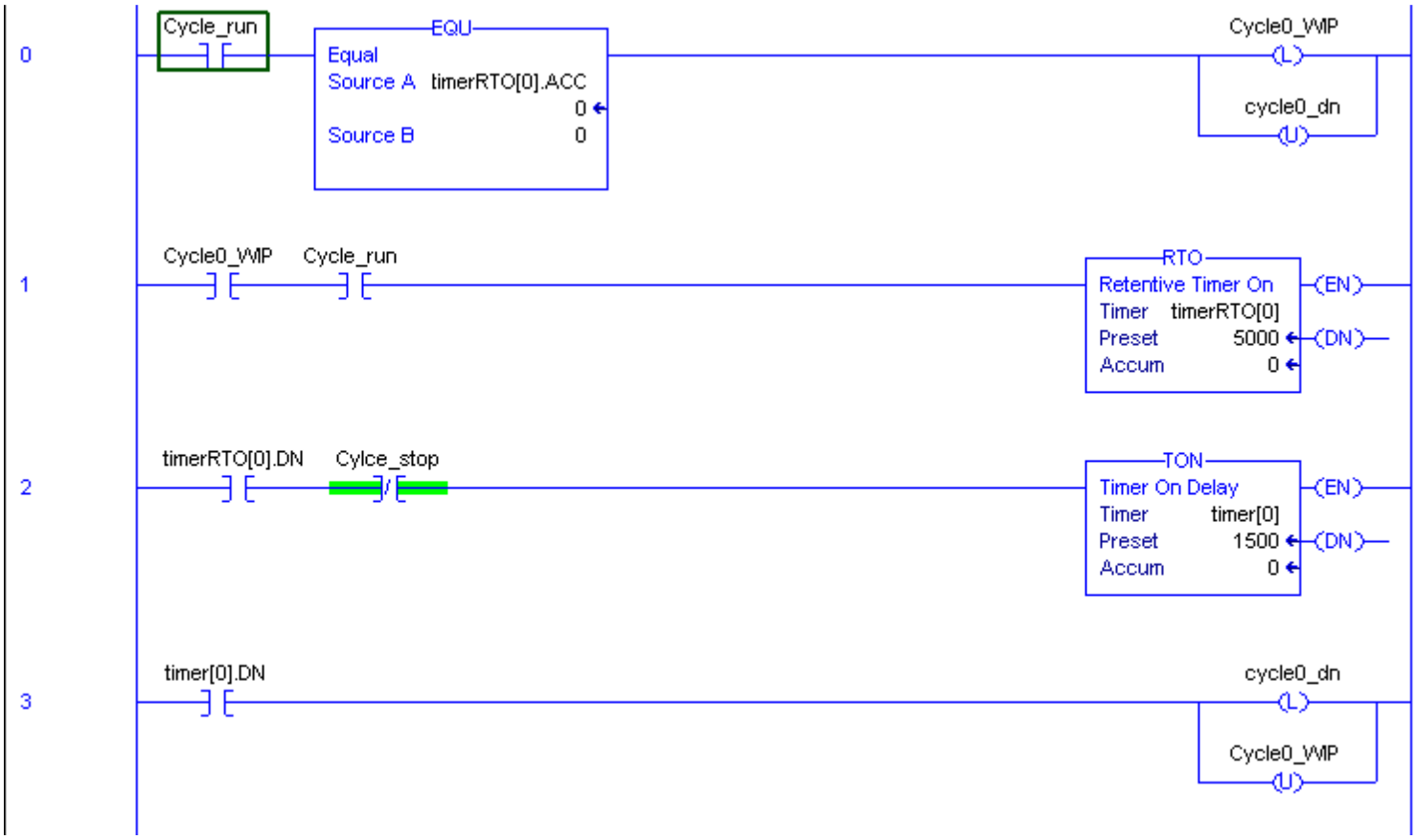
Now press ok and watch the behaviour of controller. To have a better visualization and underestanding of different phases of control system add a trend to your PLC program(To get the instructions about how to add Trends into your PLC program refer to the PID section of this manual).
Now you can change the SPProg and see the behaviour of system. Further you can go to Autune section of PIDE instruction and select other P,I,D values which are calculated and compare the response for different Set points by drwing different curves in created Trend.

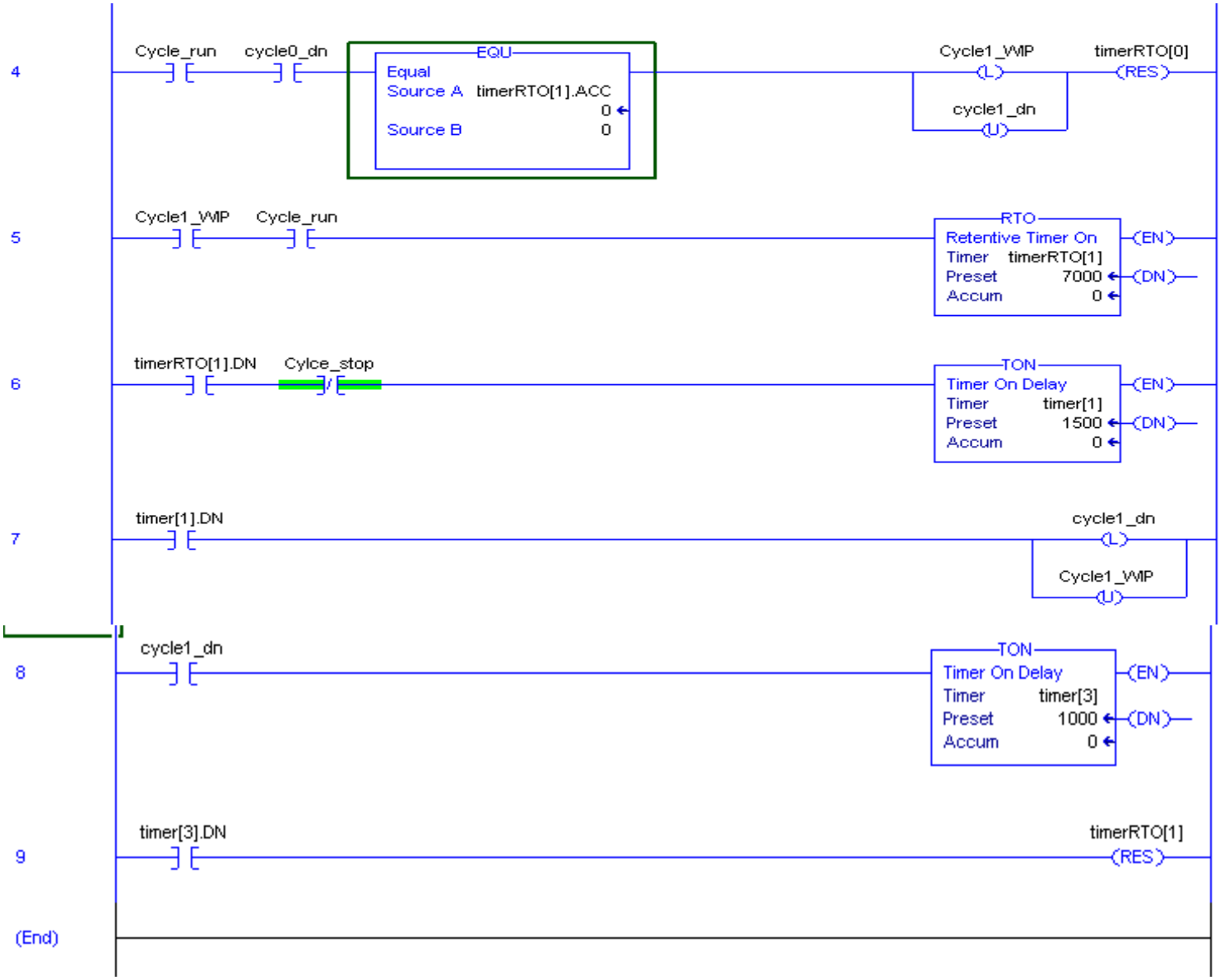


Equipment Phase

Phase Manager allows you to model your machinery around the S-88 Industry Standard for State Machine Control. This Lab covers how to add a Phase managed system to your existing program.

Before we start setting up the Phase manager, we will create a simple routine that we will run in cycle. Quickly copy the code below in a new program using Studio 5000.





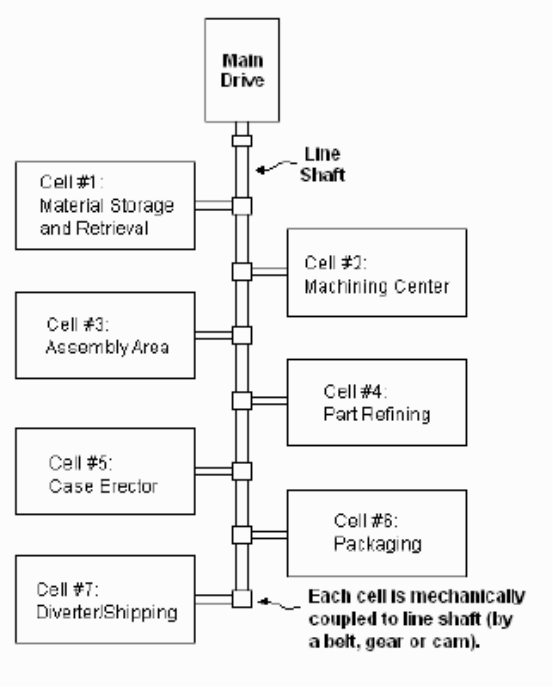
Test your application, toggle the bit "Cycle_run" the program will run as long as you don't toggle it off or toggle the bit "Cycle_stop"

Reviewing Useful Background Information

In the past, many manufacturing facilities had a line shaft that ran throughout the plant. From raw material storage and retrieval through manufacturing, packaging and even shipping, this line shaft provided the mechanical power to each "cell" or "function" along the path in order to synchronize them all together and make products correctly.

One (often large) Main Drive or motor ran the line shaft, and therefore the entire facility relied on it. As variable speed drives began to be used, the plant could now control the manufacturing speed by increasing or decreasing the line shaft speed. Since all of the other cells were mechanically connected, their associated speed increased or decreased and the plant stayed in synch.

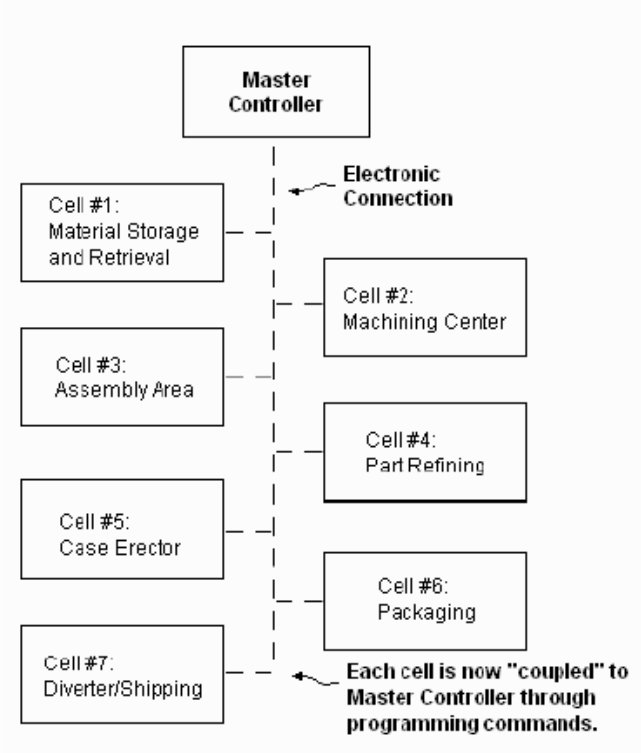
Because these line shafts were often fitted with many rotating pieces, belts, gears and cams, they were extremely high maintenance and when they went down, the entire plant was down and manufacturing stopped.



As technology progressed, the line shaft was eventually replaced by a Master Controller, and the mechanical couplings with programming commands, such as electronic gearing or camming.

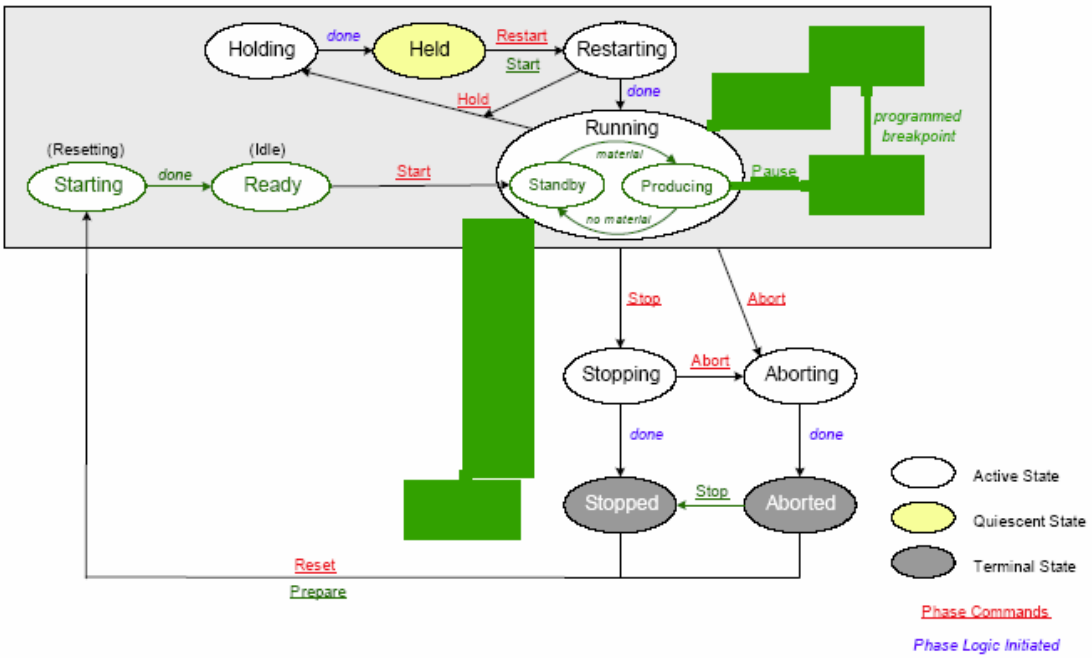
Since these cells often come from various machine builders and include a variety of controllers, it isn't always easy to synchronize them. The S88 state model was designed to help standardize all of these machines within your plant and allow them to function together, all controlled by the Master Controller.

This concept relies on standard commands being issued by the Master Controller and standard conditions being sent back to it from each cell. The format of these commands and conditions enables multiple brands of controllers to look and behave the same within your plant.



Reviewing the S88 State Model

This model shows the standard states and allowable transition that will be used to model your machine's functionality. The commands issued by the Master Controller are shown as underlined above.



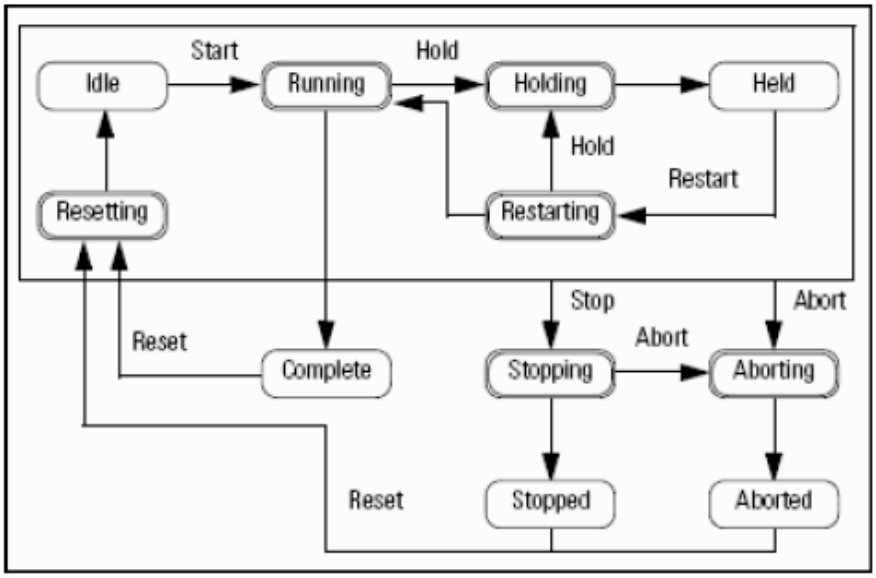
In its simplest form, PhaseManager can be used to guide your machine through all of the steps required to get ready to make products, make the products and then shut down in an orderly manner. On a much larger scale, PhaseManager can be applied for synchronizing all of the machines within your entire enterprise.

In addition to the S88 State Model, you can further standardize your systems by following tag naming conventions and other programming standards covered by Rockwell Automation's ***Power Programming*** guidelines. For more information, see the following section or visit the following website: www.ab.com/powerprogramming.

Now, let's plan out the functionality that we want each of the ***Phase States*** to contain for our machine. You can bookmark this page for the duration of the lab, if you like.

Remember, the states will simply command the machine through the functionality that already exists in our sample motion code. The work is done in those existing routines; we just have to link the state's commands to the routine's functionality.

Our Rockwell Automation representative provided us with the State Model and helpful guideline (both from the PhaseManager User Manual) to help us determine what should happen in each the Phase States. Here it is:

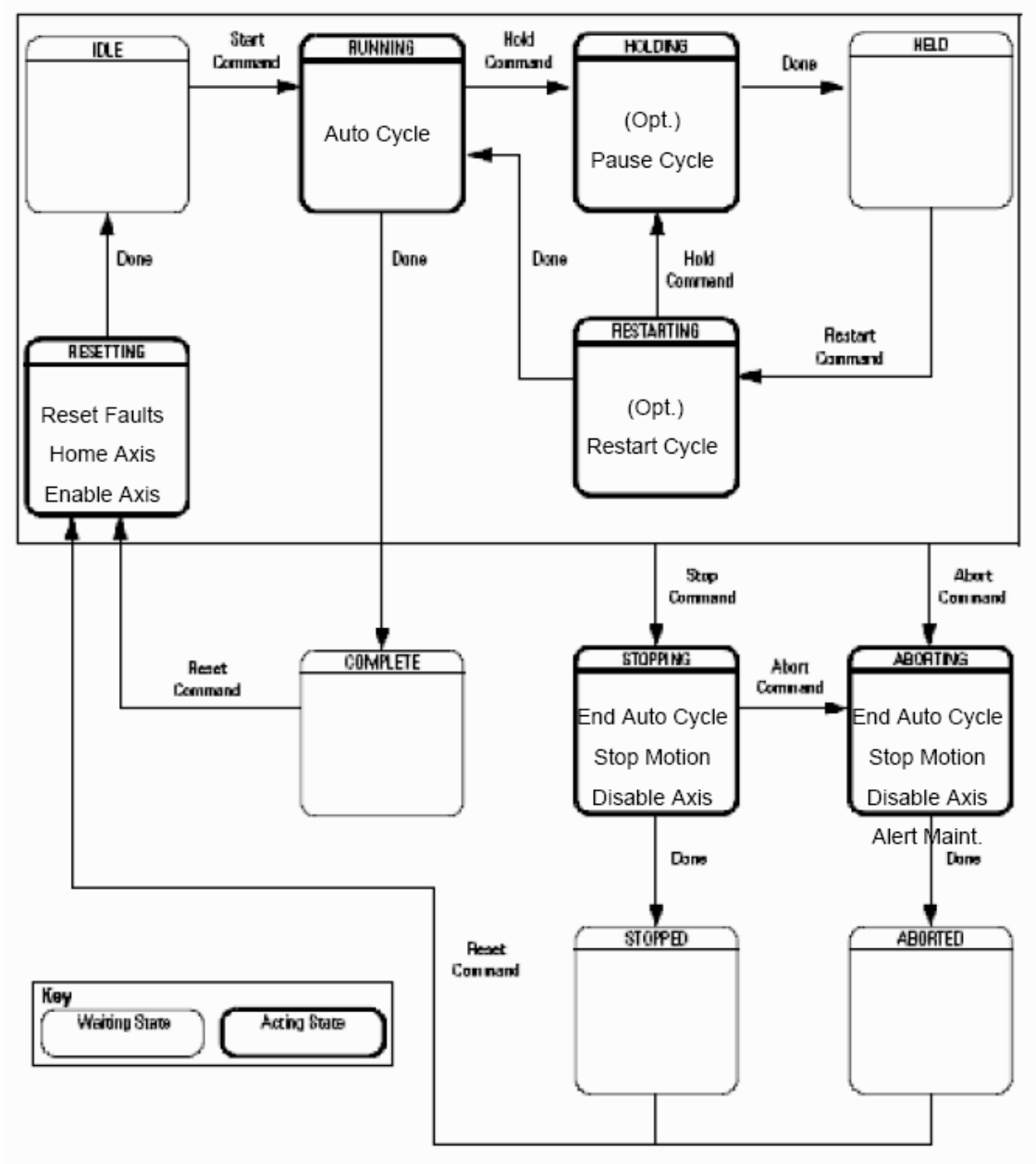


For this State:	Ask:
Stopped	What happens when you turn on power?
Resetting	How does the equipment get ready to run?
Idle	How do you tell that the equipment is ready to run?
Running	What does the equipment do to make product?
Holding	How does the equipment pause without making scrap?
Held	How do you tell if the equipment is safely paused?
Restarting	How does the equipment resume production after a pause?
Complete	How do you tell when the equipment is done with what it had to do?
Stopping	What happens during an normal shutdown?
Aborting	How does the equipment shutdown if a fault or failure happens?
Aborted	How do you tell if the equipment is safely shutdown?

You've thought about this, and filled in each of these state commands on the State Model worksheet (also from the User Manual), on the next page.

State Model Worksheet

In this section of the lab, you will review the state model for your project.



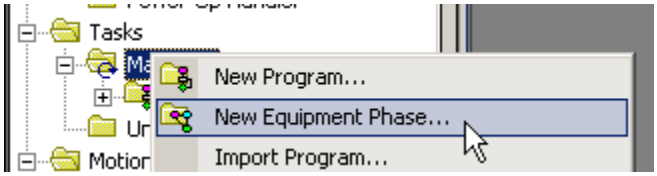
Referring to the diagram on the previous page, the following is what this State Model says:

- Our machine will start in the Stopped state, just to be safe.
- if the operator issues the **Reset** command, we will go through the **Resetting** state:
  - In our case it will reset the RTO accumulator, but this stage is usually used to **Clear faults & Enable drive**
- After seeing the **Start** command, the machine proceeds to the **Running** state
  -Enter the machine's Auto cycle, which in our case will run the program we created at the beginning of the Lab.
- If a **Hold** command is issue, in this case it will stop the RTO only
  - From this point if the operator **Restart** the program will resume
  - on the other hand if the operator decide to **Stop** the program goes through the **Stopping** stage and get back to the beginning.
- In our case we will not use **Abort** but it will act as a **Stop** because no extra command will be set.

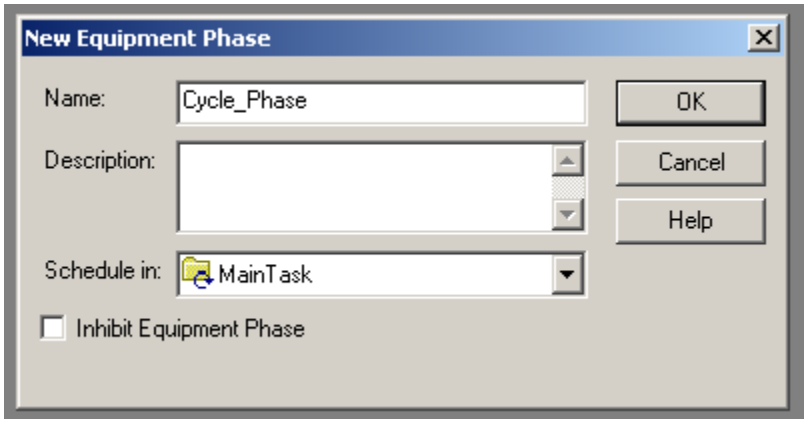
Let's begin by creating this Equipment Phase which will contain our action states.

Creating an Equipment Phase

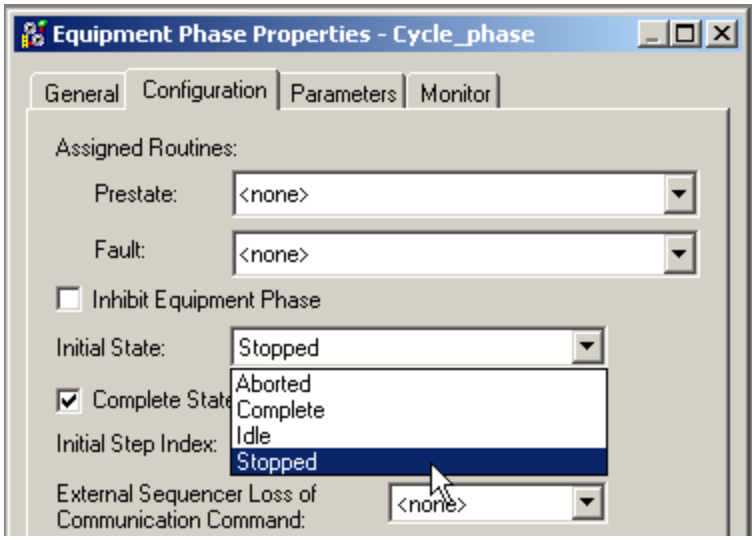
1. From the controller Organizer, right click on the **Main task** and select **New equipment Phase**



2. Name the Phase **Cycle_phase** and press **OK** to complete

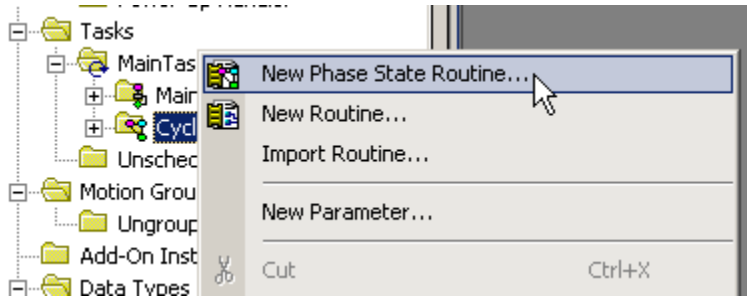


3.From the controller Organizer, Right click on the **Cycle_Phase** and select **Properties.**
4. Click on the **Configuration** tab, select **Stopped** from the **Initial State** drop down menu, then click **OK** to accept your change.
This tell our phase to start in a Stopped state.

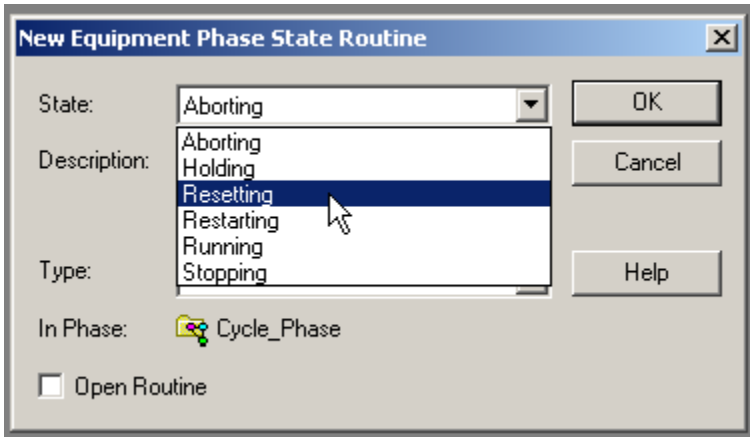


Now we have to add our Phase states.

5. Right click on the **Cycle_Phase** and select **New phase State Routine.**



6. Add the **Resetting** state as a ladder diagram and press **OK.**



7. Repeat step 5 & 6 to add **Stopping, and Running**
8. Verify that your Controller Organizer appears as follow:

*Any phase states you don't create are simply bypassed. if your machine initial state is **Stopped** and you have not created a **Resetting** state, the phase would proceed right through to the **Idle** state upon seeing the **Reset** command.*

*In our machine, not creating the **Aborting** state causes the machine to proceed directly to the **Aborted** state without commanding any actions.*

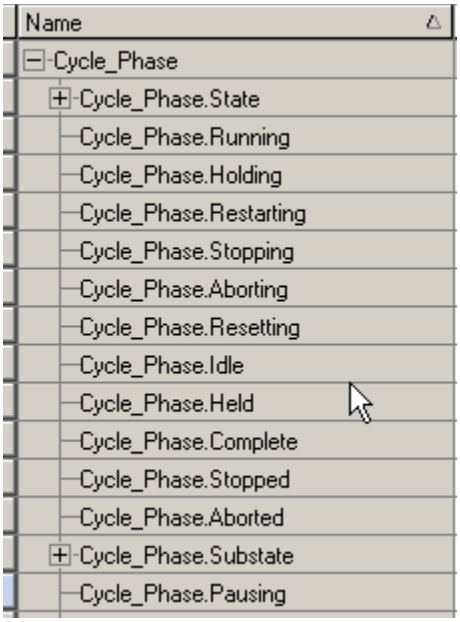
9. From the toolbar menu, **Select File> Save as**

10. Save your job in your **working folder**

At this point we need to add commands in our Phase States to eventually call the main routine we already created in the beginning of the Lab. Make sure that you have the State Model worksheet handy.

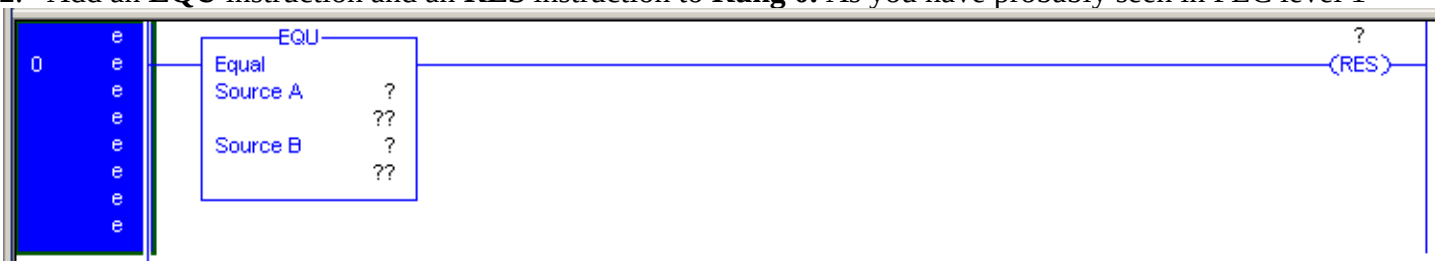
Writing Execution code for the Resetting State
Here we will add the command that will reset the RTO, on a real machine this would be use to reset drives faults, home axis, and enable axis,…

One of the many benefits of a Phase managed system is that backing tags are created for you when you create the Equipment Phase. These backing tags contains elements that indicate what State we are in, if there are any faults because a state didn't complete properly and many other helpful tags that can be used within our program. You can explore these tags in the controller tags section of the Controller organizer, if you like. Simply locate and expand the tag Cylce_Phase. See the snapshot below for a few sample tags

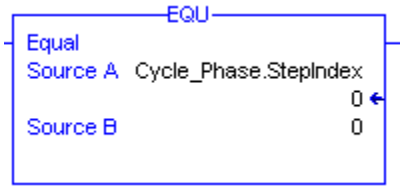


One of the tags provided in the Phase could be used as a step sequencer (Cycle_Phase.StepIndex) to force us through the steps of this state order, for instance.

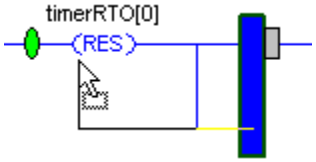
1. From the Controller Organizer, double click on the **Resetting** state in the **Cycle_Phase**
2. Add an **EQU** instruction and an **RES** instruction to **Rung 0.** As you have probably seen in PLC level 1



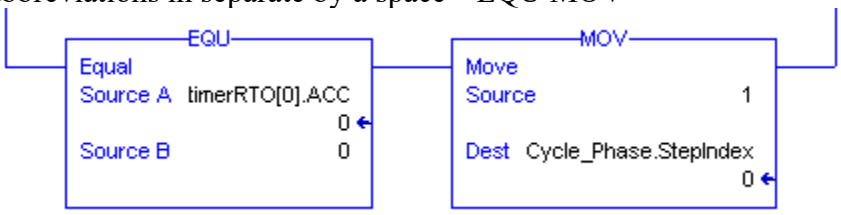
   a. We will use Cycle_Phase.StepIndex sequencer as mentioned above
3. In the **EQU** instruction, double click on the **?** for the **source A** value.
4. Browse through the tags and select **Cycle_Phase.StepIndex** tag.
5. Next, set the **Source B** value to **0** by simply typing **0** over the **?** symbol.
6. Verify that your instruction appears as shown below.



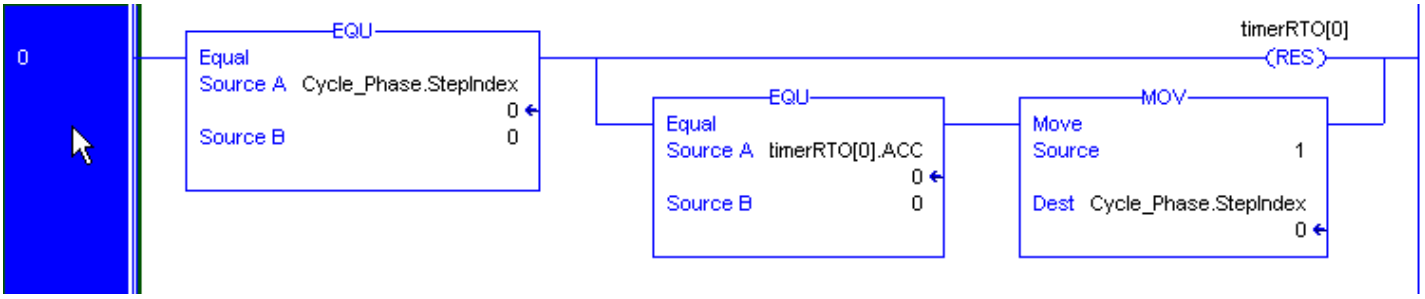
7. Click on the **?** above the **RES** instruction and type timerRTO[0], which is the first RTO for our main routine we have created at the beginning of the Lab.
8. We have to increment the StepIndex sequencer after the reset command is completed. Here is how:
9. Click on the **RES** instruction and press the **branch**  button at the top of the screen
10. Drag the branch over to the left of the **RES** instruction



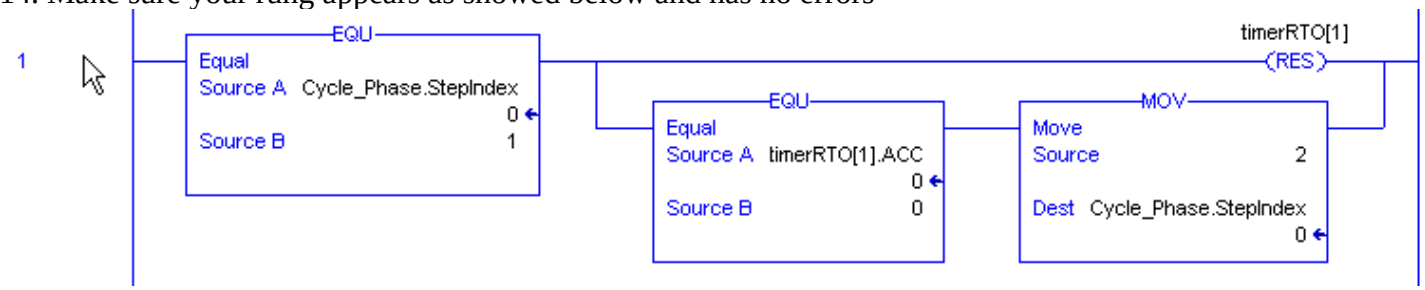
11. On the Branch you have just created, add an **EQU** and a **MOV** instruction
    a. You could also double click on the left hand side of the branch and type the instruction abbreviations in separate by a space " EQU MOV"



    b. We want to check to see that the **RTO** is rested
12. Make sure your rung appears as showed below and has no errors



13. Add an other rung with the same instruction to reset the RTO[1] when StepIndex = 1
14. Make sure your rung appears as showed below and has no errors



15. The last rung we will be added to automatically exit the Resetting phase using the **PSC** instruction once StepIndex =2
16. Make sure your rung appears as showed below and has no errors

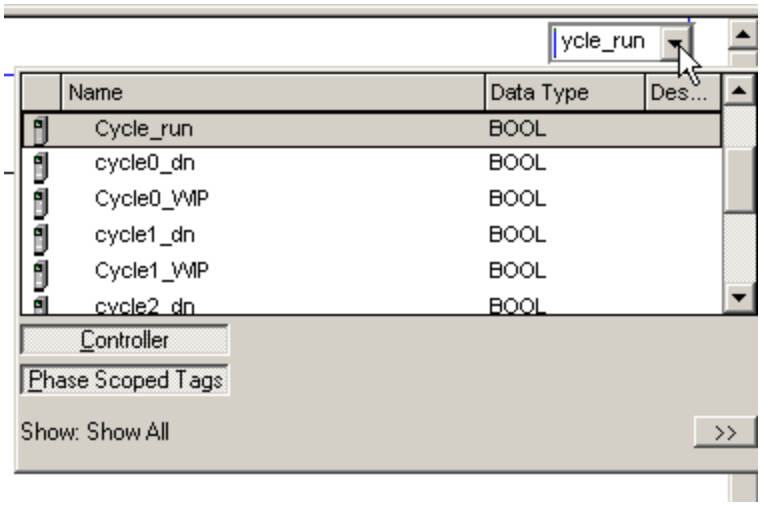
That's all we need for the **Resetting** State. The **StepIndex** value walks us through the three commands that we defined for this state and tells the Phase that it is complete. The Phase by definition now moves to the **Idle** State, waiting for a **Start** command to be issued (see the **State Model**)

Let's move to the **Running** state.

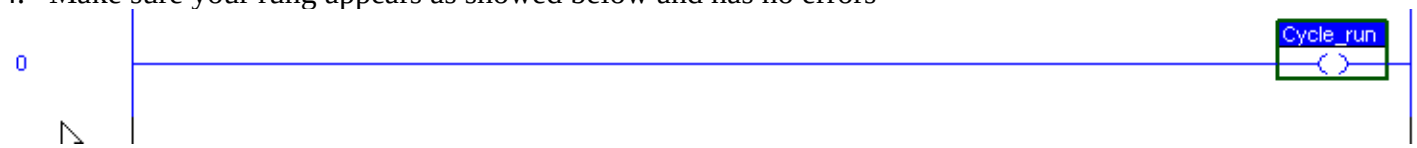
Writing Execution Code for the Running State

1. From the Controller Organizer, double click on the **Running** state in the **Cycle_Phase**
2. Add an **OTE** instruction to **Rung 0**

3. In the tag browser select the Cycle_Run that we created for the main routine



4. Make sure your rung appears as showed below and has no errors



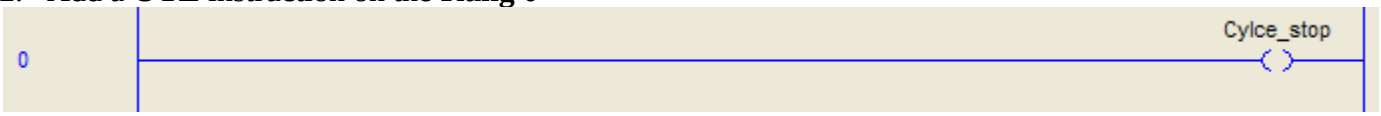
That's all we need for the **Running** State. There is no need for a **PSC** command as we don't want to exit the **Running** phase unless there is a **Stop** or **Hold** command issued

Let's move to **Holding**

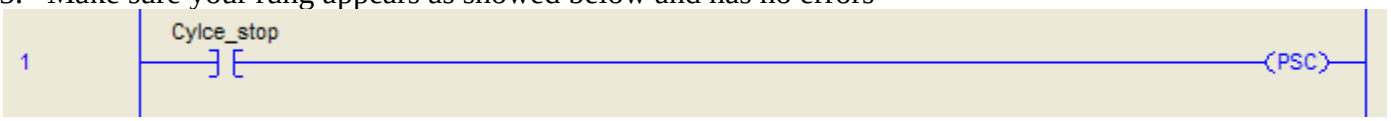
Writing Execution Code for the Stopping State

The difference between the **Holding** and the **Stopping** state is that the **Holding** state will act like a pause in the man routine. You will see that when you set the program on Hold the RTO only will stop counting (if the hold is set while the TON are counting they will finish their count then the program will hold, this is to simulate the completion of a homing after the task is complete for instance). Once you are in **Held** Sate you can either **Restart** or **Stop,** if you stop then the program will have to go through the **Resetting** state (see the **State Model**).

1. From the Controller Organizer, double click on the **Stopping** state in the **Cycle_Phase**
2. Add a **OTE** instruction on the **Rung 0**



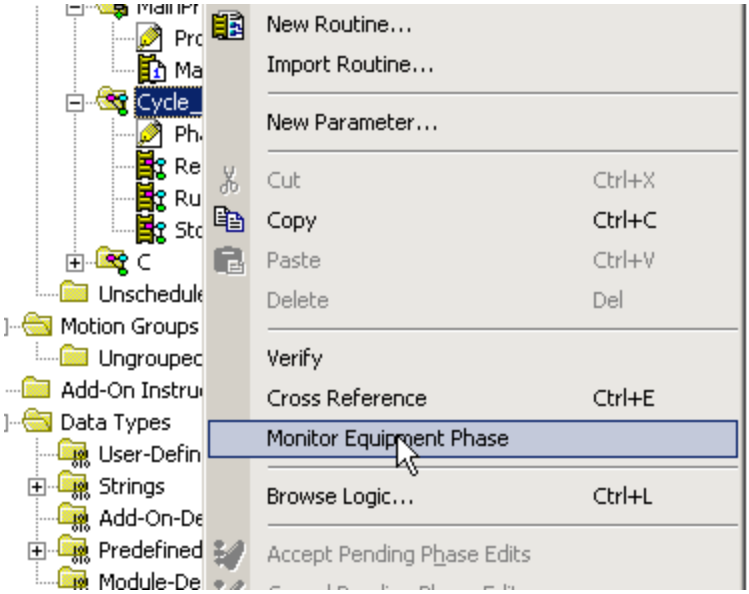
3. Assign tag as shown above
4. Add a Rung then add a **XIC** Cycle_stop and a **PSC** instruction
5. Make sure your rung appears as showed below and has no errors



Test your work

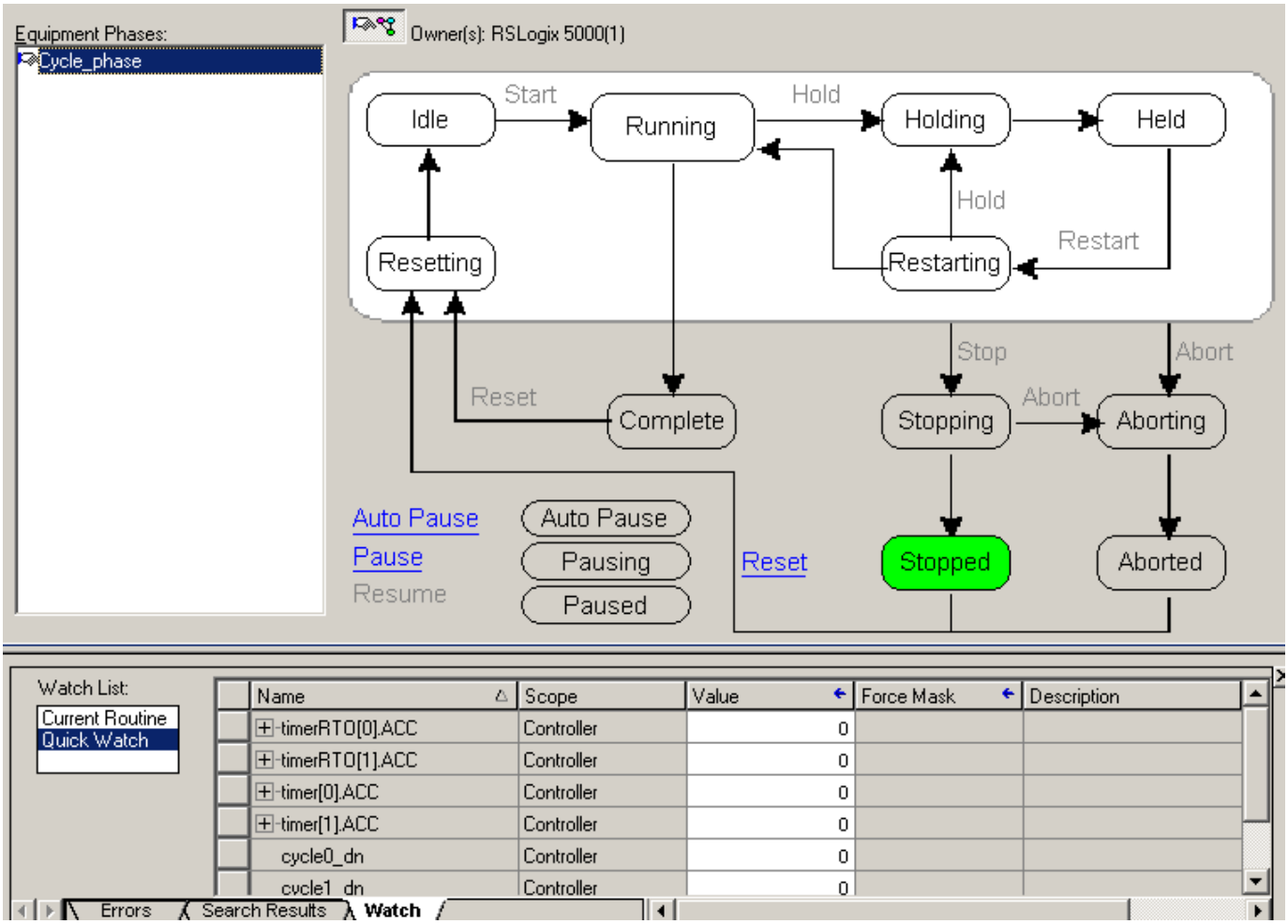
Download all your changes to the controller, go online and turn on Run mode

To open the **Phase monitor** tool, right click on the **Cycle_Phase** folder and select **Monitor Equipment Phase**



Highlight your Phase on the left pane, then click the **Take Ownership** button at the top, and answer **Yes** to the confirmation dialog that will appear.

Notice that the **Equipment Phase** has come up in the **Stopped** state as expected.
Set up the quick watch view so you can see the **RTO** and **TON** accumulator value as well as **Cycle0_dn** and **Cycle1_dn**



Now you are ready to work with the **Equipment Phase Monitoring** and being able to see what is happening in the main routine.
As we mentioned above you are now in the **Stopped** state (Green Box), the available command are shown in Blue and underlined
Press on the **Reset** command, once the **Resetting** program that we created earlier is completed the **Idle** box become green which is the actual state you are in.
Press on the **Start** command, the **Running** box become green and you can see RTO[0] start to count. Let the main routine run through a few cycle to make sure everything is fine, then try the **Hold, Restart, and Stop** commands in any different way you want and see what happen it the main routine.

Extra task: Creating the Remaining Phase states
Using Factory talk create control screen to monitor and control the Equipment Phase, use the backing tags to help you out

Structured Text and Sequential Function Chart Programming

Structured Text and Sequential Function Chart programming formats are supported in the Studio 5000 programming environment and are configured at the Routine level. Similar to the ladder format these two formats are based on the IEC-61131 standard for PLC programming. Any program can make use of any combination of these PLC routine formats. In some cases such as the SFC format we can make use of both of these programming methods within the same code structure.

The decision to use one programming format over another must be considered against the following criteria,

- Types of variables being manipulated, i.e. mostly bit logic verses processing math or calculus functions.
- Type of process being controlled, batch processing, sequential processing, continuous, intermittent or event driven processing.
- How well the PLC project conveys the operational design of the control. For instance ladder logic can quickly become unwieldy and look overly complicated while SFC presents more compact structure and a well defined orde of execution.
- How likely is the program going to be used to examine problems in process events and possible alterations for improvement or containment of electro – mechanical issues?
- What is the PLC program reading proficiency of the people who will be expected to maintain and troubleshoot the process? The original PLC specification created by GM required that all aspects of the program had to be easily understood by a line technician. The reality is this can no longer be expected given the level of sophistication of the processes that are now controlled by PLC programs.

Advantages of Structured Text programming.

- Can create a very compressed formulation of the program routine.
- Allows a reasonable means of conveying on a line by line base the intent of the software code.
- Allows the programmer to leverage powerful constructs similar to high level programming languages such as Pascal and C. This allows the implementation of OOP designed program solutions.
- Efficient representation of multiple condition checking.
- Free style text programming with adherence to a small set of delimitation rules.
- Allows for very large variable identifiers that would otherwise generate very largely spaced program elements in Ladder or SFC routines.
- Particular good for high level math calculations.
- Allows enumeration of program variables for better decision making algorithms.
- Allows for strong data typing so as to minimize error due to rounding off or poor conversions.

Disadvantages of Structured Text programming

- Because the compiler must convert from textual to machine code this process can take longer than IL or Ladder program conversion.
- Because this format can allow for a high degree of abstraction as is the case in OOP algorithms, this can also result in longer compilation times. The disadvantage can really be noticed in the last 5% to 10% of the project development activity.
- Requires some discipline in naming conventions and how variable names representing conditions as created.
  For this purpose there are standardized guidelines on text layout and naming notation available.

Structured Text Example

****

```
(*The following Structured Text code in this routine is the Traffic Light control logic making use of
one timer function driving the lamp outputs using comparison statements on the timer's accumulator variable.*)

// Note the use of two different comment delimiters, // and (* .... *)

TONR(Timer1);                  // Create an instance of the timer function
Timer1.pre := 14000;  // Set the preset variable to 14000ms
If Timer1.DN Then     // Enable and Reset the timer based on its Done bit.
        Timer1.TimerEnable := 0;
Else
        Timer1.TimerEnable := 1;
End_if;
Interval := Timer1.acc;          // Assign the timer's accumulated variable to a
                        // separate "Interval" variable for comparison.
```

****

The editor text is colour coded to help in distinguishing the different parts of the code. The colour assignments can be changed in the Options menu of the Studio 5000 editor but the default colouring is Green for comments which are ignored by the compiler, Blue for key words such as functions and black for variable names and operators. The function name is followed by open and closed brackets which contain the required arguments that are passed into the function. Variables values are assigned with the " : = " operator. Statements must be delimited with the " ; " operator so that the compiler can determine the beginning and end of an instruction. White space and comments are ignored. Unlike C and C++ programming, structured text is not case sensitive.

Advantages of Sequence Function Chart.

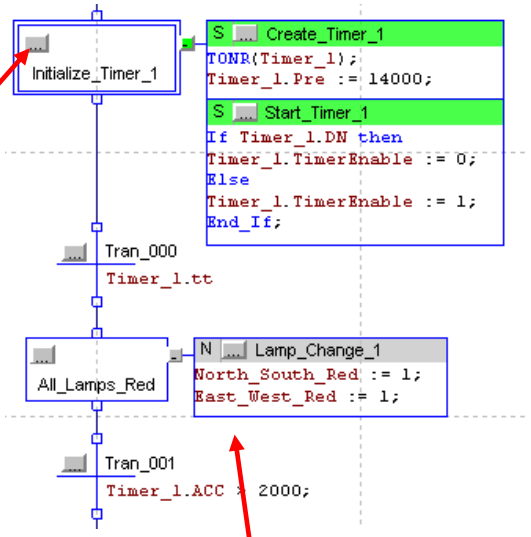
- Allows for the almost one for one transfer of the program algorithm flow chart to PLC project code resulting in graphical representation of the progression of the program execution.
- Clear indication of the tasks being executed at any given instant and the conditions required to progress through the process.
- Ideal for processes that execute sequentially with little or no variation in process path.
- Gives clear indication on the type of action that is performed within a step by the use of single or two letter qualifiers listed by each action.
- Allows for strategic placement of labeling and commenting text adjacent to the program flow elements.
- Action and transition code can be input in ST and other SFC routines or any other program format by using a Jump Sub Routine (JSR) function.
- Makes for a very efficient representation of the processing progression when there are a limited numbers (5 or less) alternate processing paths or parallel processing paths.
- Possibly the best programming format to be used as a graphical troubleshooting aid. For this reason SFC program format should be considered for the main calling routine of most PLC programs.
- SFC processor scans are generally much more efficient (faster) as there is less overhead as only code in the current step is scanned.

Disadvantages of Sequential Function Chart

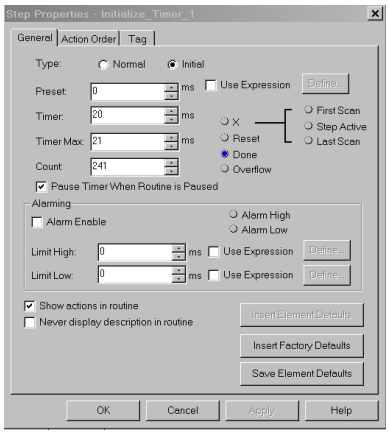
- Applications containing many simultaneous processes that operate intermittently or in different combinations can be difficult to represent and coordinate in a SFC routine.
- Hard coded into the execution processor is the rule that on entering a step, all associated actions will be processed before the transition condition is examined, even if it is already being met.

Structured Text Example

The SFC editor allows the insertion of a series of STEP boxes and TRANSITION nodes. They can be inserted together of separately.
Each STEP has an individual name and a control that allows access to a form where the step execution can be adapted to the process requirements.



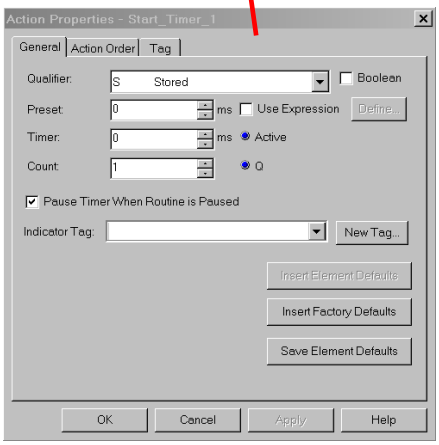
The STEP box has a small control on the right side which opens up the ACTION box which is where the individual actions are defined for the current step. A step remains active until the following transition condition is met.



Graphic 1

Graphic 2

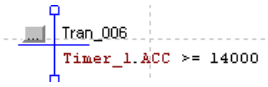
Each Action box has a small control in the upper left side that allows you to set the behavior of the action such as delaying the action for some time period or allowing the action to continue after the processing has transitioned on to the next Step. In the example to the right the Stored designation S means that the action command is stored in memory and will continue to execute after the processing has moved to the subsequent routine steps.



Graphic 3

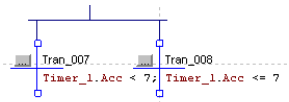
Steps must be separated by transition nodes. There are three basic types of transition.

1. Single Transition from previous to next step.



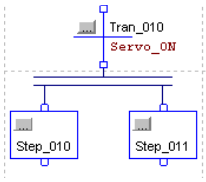
Graphic 4

2. Selection Branch Diverge.



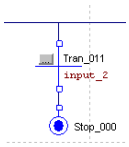
Graphic 5

3. Simultaneous Branch Diverge



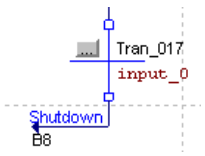
Graphic 6

Action box instructions and the transition box condition can be entered in Structured Text as in the examples above. Alternately, the JSR function can be used to run routines in other program language formats to process these actions and detect the transition condition.
Branches in a SFC can either come to a final end point or they can loop forward or backward to some other point in the routine. Termination is indicated by the Stop element as shown in n Graphic 8.



Graphic 7

Process flow that loops to another location in the SFC will have a reference to that point shown at the redirection element connected to the transition similar to that shown in Graphic 9. In this case process flow is directed to chart location B8.



Graphic 8

Structured Text and Sequential Function Chart programming Lab.

Review the Traffic Light application which makes use of a single timer function to control all lights at an intersection.

North-South Red	**X**	**X**	**X**	**X**			**0 - 9 seconds**
North-South Yellow						**X**	**12 - 14 seconds**
North-South Green					**X**		**9 - 12 seconds**
East-West Red	**X**			**X**	**X**	**X**	**0 - 2 and 7 - 14 seconds**
East-West Yellow			**X**				**5 - 7 seconds**
East-West Green		**X**					**2 - 5 seconds**
	0s	**2s**	**5s**	**7s**	**9s**	**12s**	**14s**

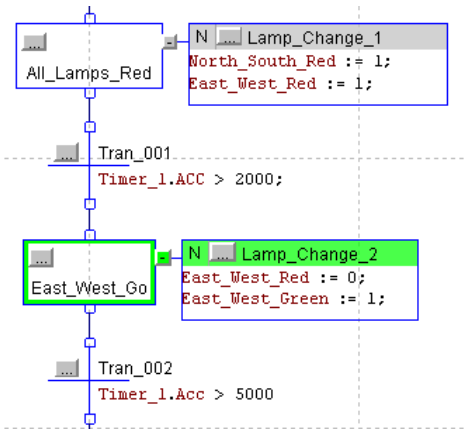
Table 1

Use a structure similar to that show in the examples in 10 and 11 to create the Traffic Light application in Structure Text and Sequential Function Chart programming format.

```
//North South Red Traffic Light Control.

If Interval > 0 and Interval <= 9000 Then
    North_South_Red_Light := 1;
Else
    North_South_Red_Light := 0;
End_if;
```

Graphic 9



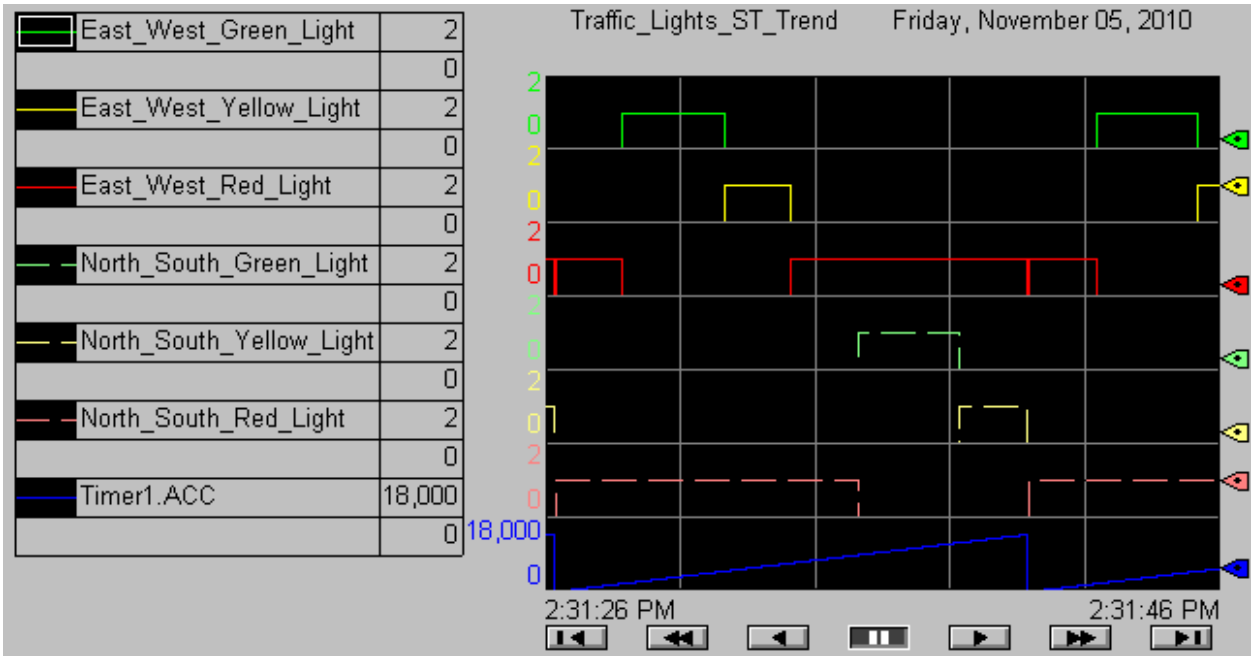
Graphic 10

Graphics switching

Do one of the following to confirm that the light sequence and timing are correct.

1. Map the light outputs to the Remote I/O module at your station.
2. Link the project tags to a HMI project running the Traffic Light animation.
3. Create a trend in Studio 5000 to track the switching of the lights.

Trend of Traffic Light application


East_West_Green_Light
2
0
East_West_Yellow_Light
2
0
East_West_Red_Light
2
0
North_South_Green_Light
2
0
North_South_Yellow_Light
2
0
North_South_Red_Light
2
0
Timer1.ACC
18,000
0
Traffic_Lights_ST_Trend
Friday, November 05, 2010
18,000
2:31:26 PM
2:31:46 PM

Graphic 11