SMLC Tutorial

An introduction to programming ORMEC's

SMLC IEC-61131-3

Soft Motion and Logic Controller

February 15, 2005

Mick Oakley,
Principal Applications Engineer
ORMEC Systems Corp
19 Linden Park
Rochester NY 14625

(585) 385-3520

www.ormec.com

Table of Contents

1. Introduction **1**

2. Basics **3**

3. Motion Control with IEC-61131-3 .... **5**

4. Application Description **7**

4.1. Program Organization 7

5. Motor/Drive Configuration **9**

6. Starting the Project **11**

6.1. Opening a New Project 12

6.2. Add the Supervisor State 13

6.3. Add the Estop State 15

6.4. Add the Manual, Homing and Auto States 15

6.5. Setting up the Transitions 17

6.6. Initialize the Jog Speed 22

6.7. Opening Drive Communications 23

6.8. Setting the Next State 26

6.9. Leaving the Init State 26

6.10. Programming the Estop State 27

6.11. Closing The Bus Contactor 27

6.12. Enabling The Drive 28

6.13. Estop Entry Action 29

6.14. Ladder Diagram Comments 30

7. Testing Your Work So Far **32**

7.1. Compilation Problems 33

7.2. Unused Variables 33

7.3. Running the Program 33

8. The Supervisor State **38**

9. Manual State **40**

9.1. Homing State 41

9.2. Adding the Homing State Entry Action 42

10. Adding the EveryScan Action **44**

10.1. Creating the IsAtHome Function 44

11. Auto State **48**

Table of Contents

11.1. Alternative Auto State 49

12. Error and Fault Handling 52

12.1. Error and Fault Annunciation 52

12.2. Creating The HMI Task 53

13. I/O Configuration 58

13.1. Associated I/O Task 61

13.2. I/O Cycle Time 62

14. Appendix A – Variable Prefixes 64

Table of Figures

Figure 1, Application State Diagram 7

Figure 2, User Unit Setup for Tutorial 9

Figure 3, Main Screen Layout 11

Figure 4, New POU Dialog 12

Figure 5, New Sequential Function Chart 12

Figure 6, Step and Transition Selection Points 13

Figure 7, Init and Supervisor States 13

Figure 8, Init, Supervisor and Estop States 15

Figure 9, Completed SFC Diagram Template 16

Figure 10, Initial State ENUM Definitions 18

Figure 11, Completed States ENUM Definitions 18

Figure 12, Global Variables 18

Figure 13, Global Variables Window 19

Figure 14, Help Manager 19

Figure 15, Initial SFC Diagram 21

Figure 16, Help Manger Window 23

Figure 17, Init Action 24

Figure 18, Opening an Axis Using Ladder Diagram 25

Figure 19, Init State Actions 26

Figure 20, Ladder Diagram Editor 27

Figure 21, Closing The Bus Contactor 28

Figure 22, Powering Up the Drive 28

Figure 23, Completed Estop state action 29

Table of Contents

Figure 24, Estop Entry Action 30
Figure 25, Adding a Comment to a Ladder Network 30
Figure 26, Completed Ladder Network Comment 31
Figure 27, Online 33
Figure 28, Program Running 34
Figure 29, Network with program stopped on a break point 35
Figure 30, Manual State Action 40
Figure 31, Homing Entry Action 42
Figure 32, New POU Window 44
Figure 33. IsAtHome Function 45
Figure 34, EveryScan Action 46
Figure 35, Auto State 48
Figure 36, Alternative Auto State using Enqueued Moves 49
Figure 37, ErrorCheck Action 52
Figure 38, HMI POU Configuration 53
Figure 39, Task Configuration – PLC_PRG 55
Figure 40, Task Configuration 55
Figure 41, HMI Task 56
Figure 42, Control Panel Visualization 57
Figure 43, Error Log Visualization 57
Figure 44, PLC Configuration 58
Figure 45, HMI String Registers 59
Figure 46, HMI Integer Registers 60
Figure 47, Typical I/O Configuration 62

Table of Contents

SMLC Tutorial

1. Introduction

This tutorial is designed as an introduction to the motion features of ORMEC's SMLC IEC-61131-3 programming language. It will walk you through the process of creating a simple program that will allow you to enable an axis and move it. It will continue to build on this framework to add a homing function and simple automatic cut-off system.

This is intended as a Quick Start guide, not a complete training manual. If you want a more thorough discussion of programming with IEC-61131-3, we suggest you visit the PLC Open website at http://www.plcopen.org where you can check out the links to Training and Education and/or Publications & Press - IEC Books.

You may also consider attending one of ORMEC's PLCopen certified training classes.

2. Basics

IEC-61131-3 is an international standard that defines programming languages for machine and motion control applications. ORMEC's implementation of these languages complies with IEC-61131-3 and provides enhancements to them as permitted by the standard.

The standard defines 3 graphical languages and 2 text based languages. The ORMEC implementation adds one additional graphical language.

Standard Languages:

- Sequential Function Chart (SFC) — Provides a graphical overview of an application based on a state machine model. ORMEC recommends you use SFC as the starting point for any application rather than a monolithic Ladder diagram since it provides structure and keeps each machine state self-contained. ORMEC's implementation supports the full IEC version of SFC along with a simplified and easier to use version. This easier version is the one featured in this tutorial.
- Ladder Diagram (LD) — Similar to the graphical languages used by most Programmable Logic Controllers (PLCs), Ladder diagrams are an ideal way to manipulate inputs and outputs as well as logical decisions. Ladder diagrams are typically used to implement SFC actions within a SFC program.
- Structured Text (ST) — A textual language with many similarities to Pascal. ST is ideal for SFC Actions or user-defined functions that require mathematical, data or text manipulation.
- Instruction List (IL) — A textual language with many similarities to Assembly Language, IL is often used to implement user-defined functions and function blocks that are called by other languages. Because of its low level syntax it is usually not a good choice for portions of the program to which plant maintenance personnel are exposed.
- Function Block Diagram (FBD) — Function Block Diagrams are very common in the process industry but less often used in general automation. They express the behavior of functions, function blocks and programs as a set of interconnected graphical blocks, like in electronic circuit diagrams. They look at a system in terms of the flow of signals between processing elements.

Additional Languages:

- Continuous Function Chart (CFC) — A variation on FDB that allows free placement and interconnection of function blocks including feedback connections.

ORMEC recommends that programs start with a Sequential Function Chart and then use either Ladder or Structured Text to implement actions and transitions. Where plant maintenance personnel may need to trouble-shoot a step, Ladder is probably the best choice. Some tasks such as data manipulation and extensive calculations are usually best accomplished using Structured Text.

3. Motion Control with IEC-61131-3

The PLCopen – TC2 – Task Force for Motion Control has proposed standards for IEC-61131-3 Function Blocks for motion control applications. ORMEC's motion control function blocks have been certified as complying with the standard. ORMEC Systems Corporation is a voting member of PLCopen.

4. Application Description

We'll start by creating a small application that resembles a rotary knife. It will consist of one axis, which we will call the Knife axis. When the program starts, it will set up and assign values to some variables then open communications with a drive. The program will then enter the Estop state.

When the Reset input is activated, the drive will be powered up and the program will enter the Manual state.

In the Manual state, you will be able to jog the motor forward or reverse and home the motor to its once-per-revolution marker.

Providing the motor is at its home position, the Auto switch will cause the motor continuously make one-revolution indexes with a dwell between each one until either the Auto switch or Estop input is turned off.

If a fault occurs, the system will stop any motion in progress and return to the Estop state.

4.1. Program Organization

The program will always be in one of five following states:

1. Initializing
2. Estop
3. Manual
4. Homing
5. Auto

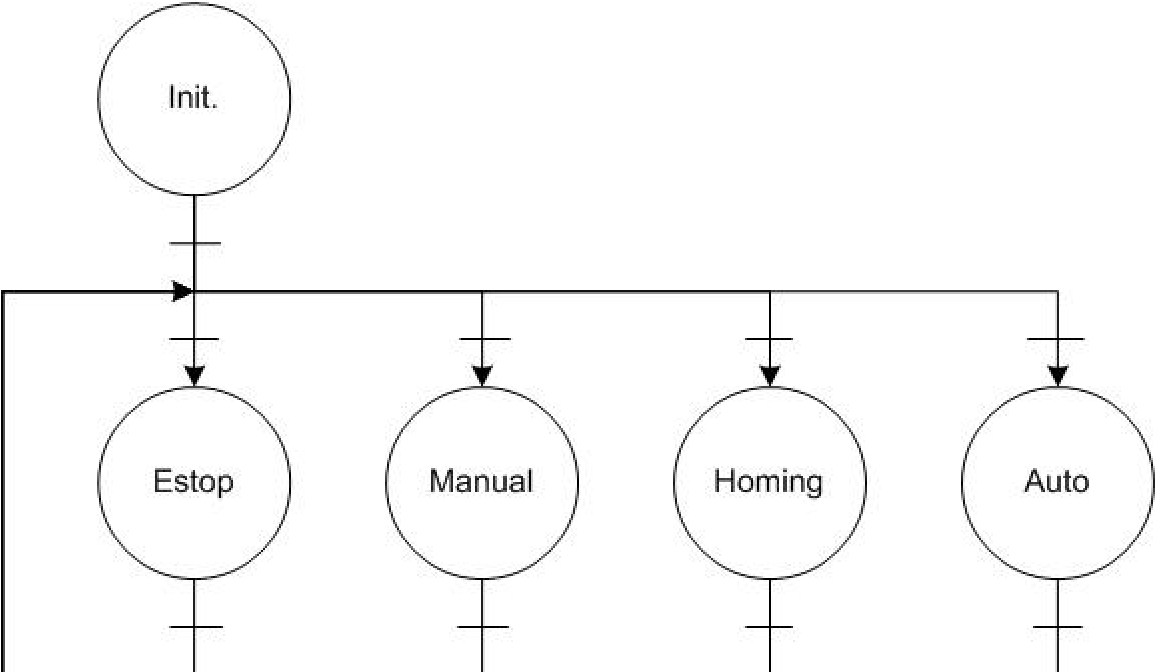


Figure 1, Application State Diagram

In addition, we will want code that executes regardless of what state the program is in to check for faults and errors, as well as updating information for display on an HMI.

5. Motor/Drive Configuration

This application requires a single servomotor. This is configured using ORMEC's ServoWire Pro software. The exact configuration will depend on the specific motor and drive available to you. The example in this tutorial uses a MAC-G005A1/M2 motor and SAC-SWM203/E drive. Information on how to use ServoWire Pro is the subject of another tutorial and is not covered here. For those already familiar with ServoWire Pro, Figure 2 shows the user units setup for the tutorial. The filename used for the complete configuration is 'Tutorial.SwSetup'.

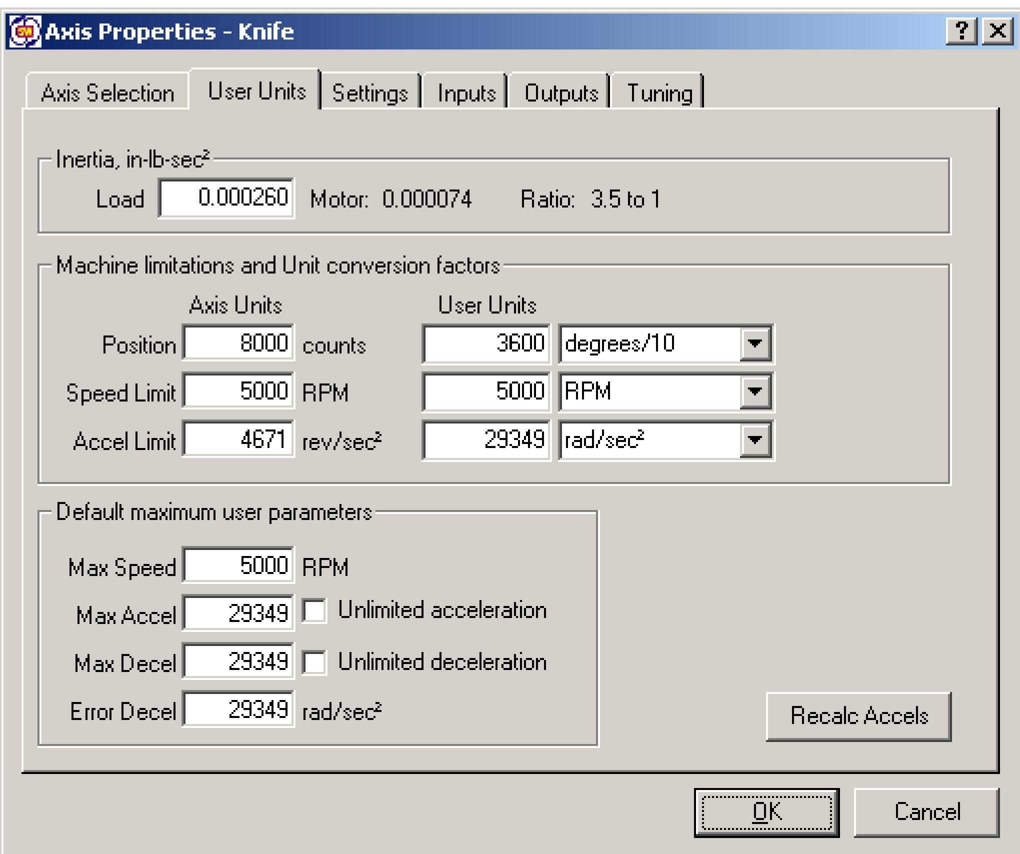


Figure 2, User Unit Setup for Tutorial

6. Starting the Project

The first steps in developing the program are to create the Sequential Function Chart that provides an overview of the program.

- Open the SMLC program editor by selecting Start Menu - Programs - 3S Software - CoDeSys V2.3 - CoDeSys V2.3.

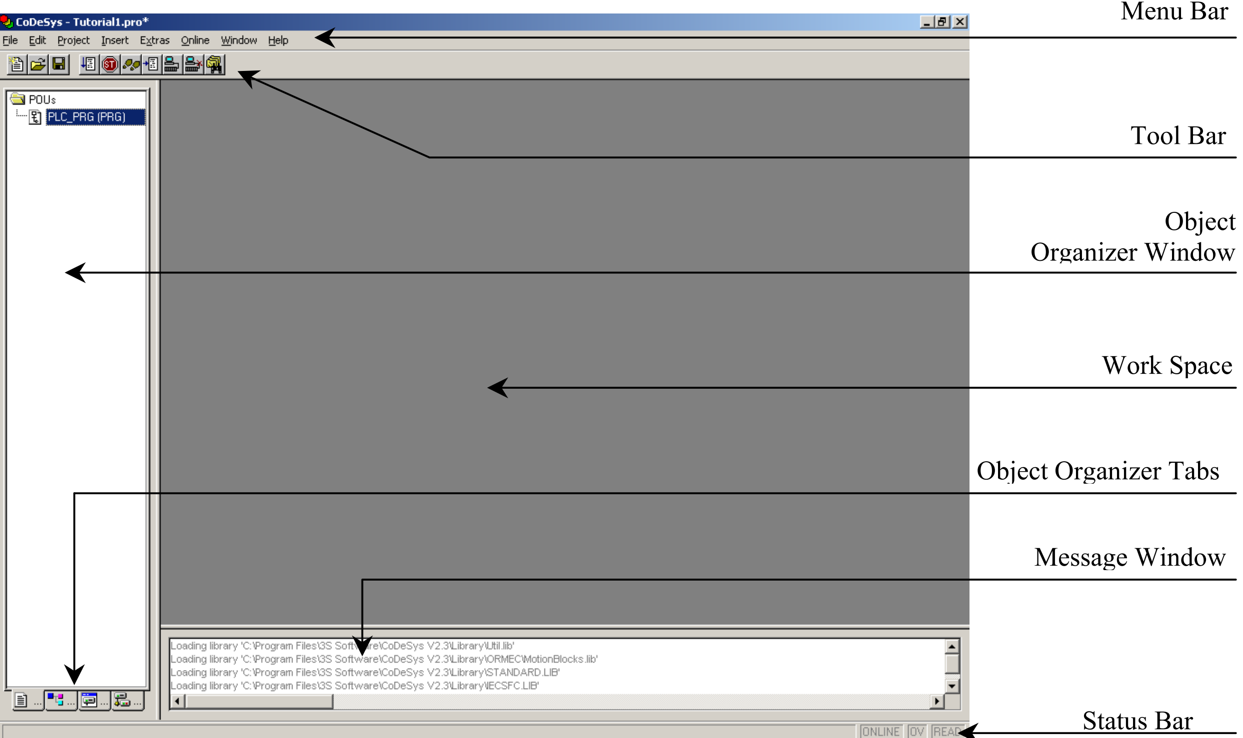


Figure 3, Main Screen Layout

<u>Tutorial Conventions</u>

- Bulleted paragraphs like this one indicate actions you must take if you are creating the application as you read the tutorial.

Normal paragraphs give an explanation of how things work or what comes next.

> Text within shaded boxes like this, provide background information that is useful but may be skipped if you are in a hurry to finish the tutorial.

The early sections of the tutorial provide step-by-step instructions on how to enter the information in you your project. To encourage you to apply what you learn as the tutorial progresses, the step-by-step instructions become less and less detailed until they are minimal.

The tutorial uses variable names that comply with the recommended prefixes shown in Appendix A.

6.1. Opening a New Project

This section tells you how to open a new project and start writing your program.

- Click on the **New** icon in the toolbar . When the Target Settings dialog pops up, select “CoDeSys SP v2.3 for ORMEC SMLC”. If you don’t have an SMLC controller, you will still be able to create and compile the program but you’ll only be able to run it in simulation mode.
- The New POU dialog will appear

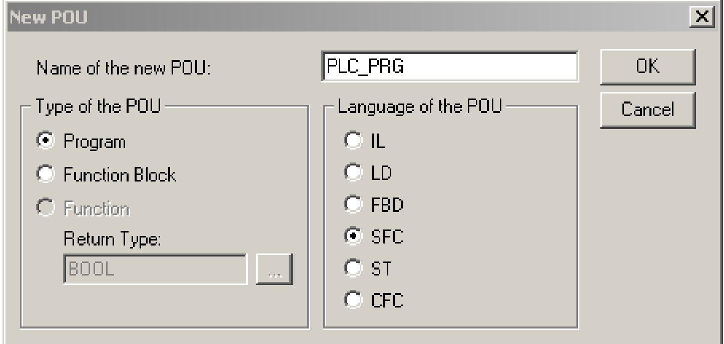


Figure 4, New POU Dialog

- Set the following values:

Name of the POU:	PLC_PRG
Type of the POU:	Program
Language of the POU:	SFC

We’ll use SFC because this is the best language to implement a state diagram such as that shown in Figure 1

- Click **OK** to continue.

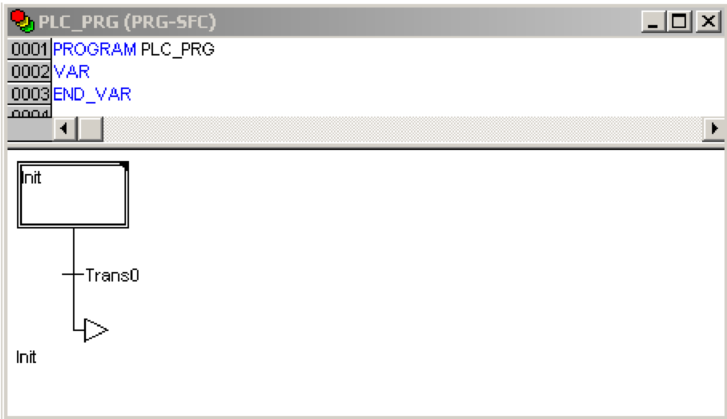


Figure 5, New Sequential Function Chart

When we run the program, it will start with any code we add to the Init state. We’ll use this to set the value of some variables and to open communication with the Knife drive.

Trans0 represents a condition that must become true for the program to stop looping through the code in the Init step and move on to whatever step we add next.

About POUs

Programs, function blocks and functions are all called Program Organization Units or POUs. They represent the blocks from which a program is created.

POUs can either be *standard* or *user-defined*.

Standard POUs are supplied with the libraries you include in your project using the **Window..Library Manager** selection from the menu bar.

User-defined POUs are those you create yourself or copy from other projects.

Every project must have one POU named PLC_PRG. This is the starting point for the program. It can be completely self-contained or it may call other POU’s.

A POU can be written in any of the IEC-61131-3 languages:

- Instruction List (IL)
- Ladder Diagram (LD)
- Function Block Diagram (FBD)
- Sequential Function Chart (ST)
- Structured Text (ST)
- Continuous Function Chart (CFC)

Each POU you create may be written in a different language if you choose.

6.2. Add the Supervisor State

The Supervisor state is where we will check for faults and errors as well as update information for the HMI.

- Click on the Trans0 transition as shown below, a dotted box should appear around the Trans0 transition.

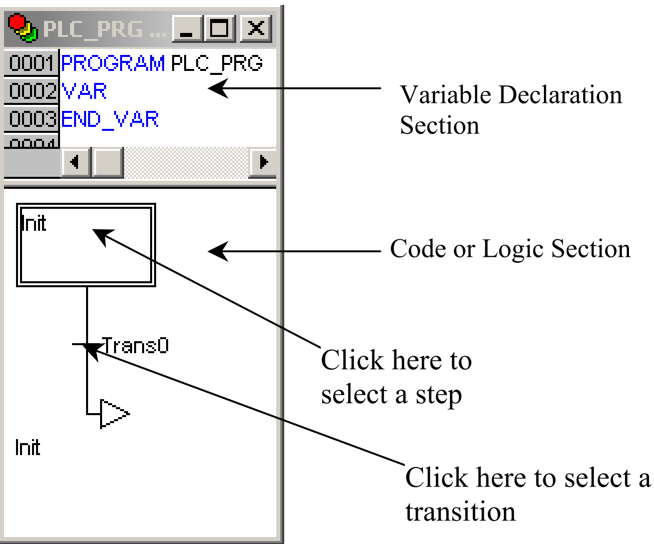


Figure 6, Step and Transition Selection Points

The POU Window

The POU window shown to the left has two sections.

The upper section is the Variable Declaration section, where as the name suggests, you define your program variables.

The lower section is the Code or Logic section where you write the code for your application. The word code is used loosely since the code may be Structured Text, Ladder Diagrams or any of the other IEC-61131-3 languages.

- Now click on the **Use IEC-Steps** icon on the tool bar, this will allow us to use a slightly different type of step that provides code that executes on every scan regardless of what state the program is in.
- Now click on the **Step-Transition (after)** icon on the tool bar or press **Ctrl+E**.

Step2 and transition Trans1 should appear below the Trans0 transition.

- Click on the name Step2 and change it to _Supervisor.

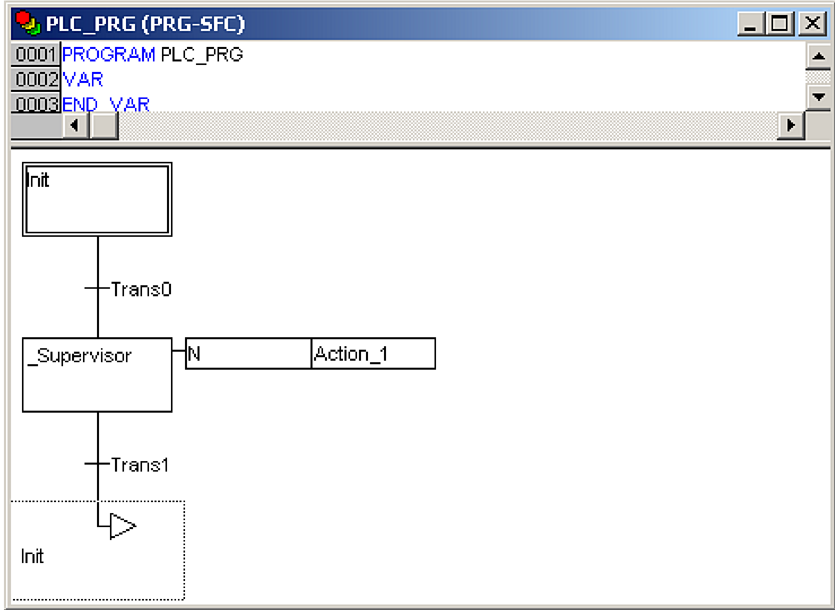


Figure 7, Init and Supervisor States

SFC Diagrams

SFC Diagrams consist of steps (sometimes referred to as states) separated by transitions. The code or logic within a step is known as an action. The program continues to loop through the action code or logic until the transition below it becomes true.

Steps can lead to multiple branches, in which case the transitions must be mutually exclusive. The program will loop within the step until one of the transitions becomes true.

The program will then drop into one of the alternate steps depending on which transition becomes true first.

IEC-Steps

IEC-Steps in an SFC provide additional capabilities over normal CoDeSys steps. You can associate up to 9 different types of "Action" programs with each IEC-Step. The different types relate to when and how the "Action" program executes. The types are:

N	**N**on-stored	The action program runs as long as the step is active
R	overriding **R**eset	Deactivates a specific action program
S	**S**et (**S**tored)	Continues to run, even after the step is no longer active, until an "**R**" action stops it
L	time **L**imited	Runs for a specific time or until the step is no longer active, whichever comes first
D	time **D**elayed	Starts running a specific time after the step becomes active (if the step is still active) and continues to run until the step is no longer active
P	**P**ulse	Runs just one time when the step becomes active
SD	**S**tored and time **D**elayed	Starts running a specific time after the step becomes active and continues to run after the step is no longer active until an "**R**" action stops it
DS	time **D**elayed and **S**tored	Starts running a specific time after the step becomes active as long as the step is still active. It continues to run after the step is no longer active until an "**R**" action stops it
SL	**S**tored and time **L**imited	Starts running as soon as the step becomes active and continues to run for a specific time, even if the step is no longer active. It can be stopped before that time by an "R" action

Note: When an action program is stopped it will execute one more time. This means that each action at least is executed twice. This applies to all action types, even a Pulse "**P**" action.

Non IEC-Steps

Non IEC Steps can contain only three types of action program. One that runs for as long as the step is active, an **E** or **E**ntry action that runs once when the step first becomes active and an **X** or e**X**it action that runs once when the step stops.

Deleting an IEC Step

Simply right-click in the associated state and select, "Clear Action/Transition" from the pop-up menu. Then select the step (or transition) you want to delete from the list.

Now we will add another action to the Supervisor state.

- Right-click inside the Supervisor state and select Associate Action.
- Change the "N" to "S" and change the action name to "ErrorCheck".

We will use this action to implement code that must run once every scan regardless of what state we are in.

Now we'll add another action to update the HMI.

- Right-click on the Supervisor state and select **Associate Action**. (**Use IEC Steps** should still be checked).

- Rename the action "EveryScan" and change the "N" to "S".
- Since we will not be using IEC Steps for the remainder of the program, deactivate IEC-Steps by clicking again on the **Use IEC-Steps** icon.

6.3. Add the Estop State

- Deactivate IEC-Steps by clicking again on the **Use IEC-Steps** icon
- Using the same process as 6.2, add another step and transition below the Trans1 transition and rename it to _Estop.

The result should be as shown in Figure 8.

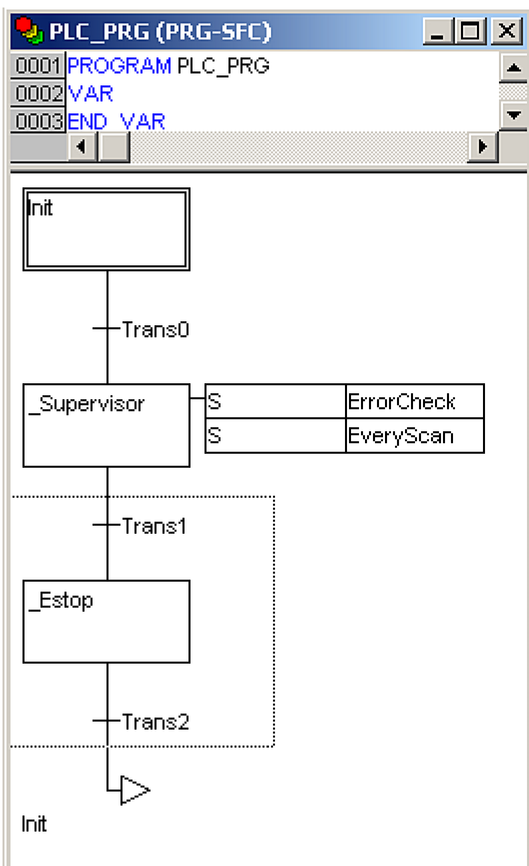


Figure 8, Init, Supervisor and Estop States

Step and Transition Names

When you create an SFC step, you also create a Boolean variable with the same name as the step. For example, if you create a step named "Manual" a variable named "Manual" is automatically created. This variable will be true while the step is active and setting false deactivates the step. If you inadvertently create and use your own variable named "Manual" you can cause unexpected program behavior, which can be very hard to troubleshoot.

To avoid this we recommend you always rename your SFC steps using an underscore "_" prefix and of course, never name your own variables beginning with an underscore.

Likewise, we recommend transition be renamed with a prefix of "trn". Specific suggestions on how to name the transitions for this application are given in Figure 15 on page 20.

These conventions have the added advantage of making printed documentation easier to follow.

6.4. Add the Manual, Homing and Auto States

Select Trans1, then while pressing the shift key, select _Estop. Now while still holding the shift key, select Trans2.

Trans1, _Estop and Trans2 should now be enclosed in a dotted box as shown in. Figure 8.

- Click on the **Alternative Branch (right)** icon or press **Ctrl+A**. This will insert a branch to the right of the Manual state.
- Now select Trans3 and insert a **Step Transition (below)**.

- Rename Step4 to _Manual.
- Repeat the process and add a Branch and Step/Transition to the right of _Manual and rename it to _Homing.
- Repeat the process above to add a new branch and Step/Transition to the right of _Homing and rename it to _Auto.
- The symbol to the right is a jump that tells the program to jump to the Init State after exiting _Estop, _Manual, _Homing or _Auto. Change the target of the jump "Init", to _Supervisor.

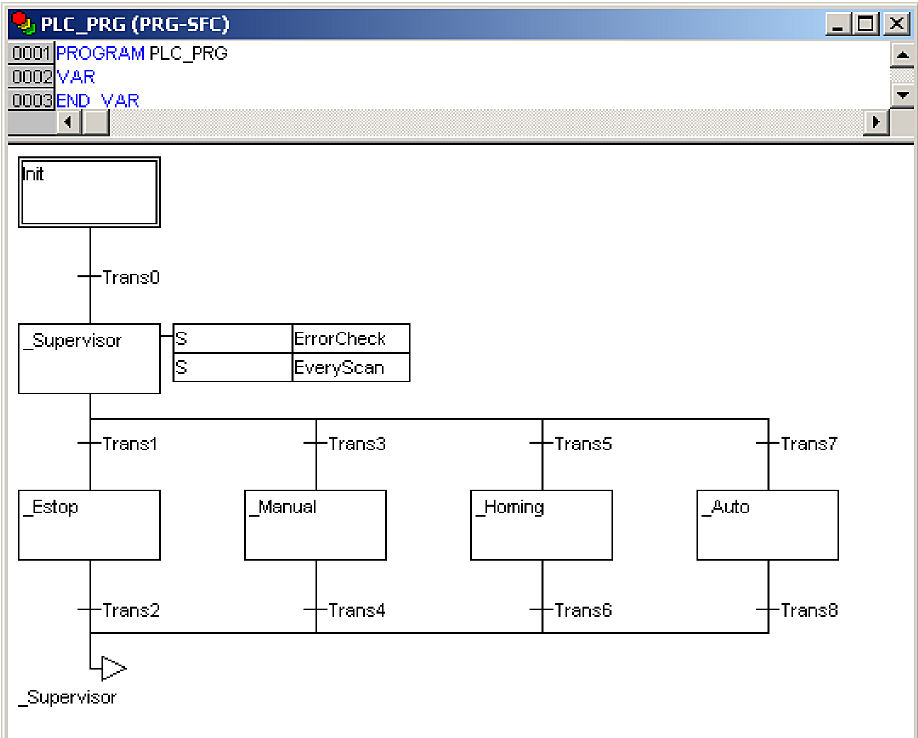


Figure 9, Completed SFC Diagram Template

The SFC template is now complete and should look as shown in Figure 9.

We call it a template because we still have to add the code for the states and the transitions.

Now you can begin to see how Sequential Function charts make it easier to understand how programs flow.

The program starts in the Init block where variables are set up and communication with the drive is started.

It then moves to the Supervisor state where logic will determine whether the program should move to the _Estop, _Manual, _Homing _Auto state.

When the program exits any of these four states it will jump back to the _Supervisor state.

The next thing we will do is set up the logic for the transitions Trans0 through Trans8. These transitions control when the program may move from one state to another.

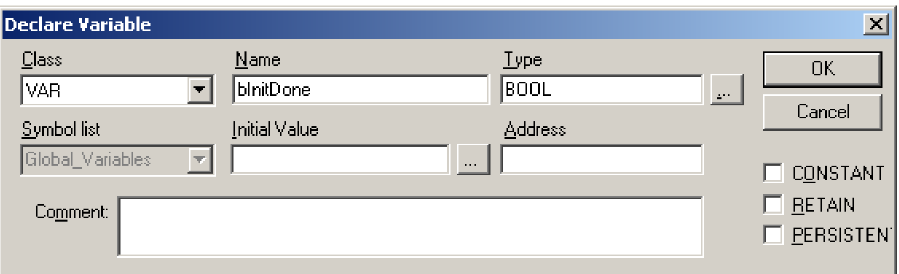
6.5. Setting up the Transitions

To set up the transitions, we'll write the conditional expressions for each transition and create variables to use in them. We'll start by setting up the condition for Trans0.

- Double-click on the Trans0 transition, where the vertical and horizontal lines cross. This will open the New Transition dialog, which asks what language you want to use. Structured Text (ST) or Ladder (LD) are usually the best choices. For very simple transitions, Structured Text is the easiest and that is what we'll use here.
- When the Structured Text editor opens, type the following:

```
bInitDone = TRUE
```

  As soon as you hit the **Enter** key, the system recognizes that bInitDone is a variable that has not been used in the program before. Since all variables must be declared, you are given the opportunity to declare it. The Declare Variables dialog defaults the variable type to BOOL which stands for Boolean or true/false. This happens to be the type we need here.



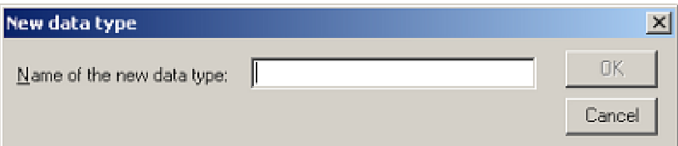
- Set the following values: Class VAR Name bInitDone Type BOOL
- Click OK to continue.

When you close the Structured Text editor, you'll notice Trans0 now has a black triangle indicating the transition has now been programmed. Trans0

- Change the name of Trans0 to trnSupervisor. This will make the program documentation easier to follow.

The next variable we need to declare will be for the state of the machine. Since there are only a discrete number of states the machine can be in, we want to make sure a programming error can't put the machine into some invalid state. The best way to do this is to create a user defined Enum type. An Enum (**Enum**erated) variable can only have certain pre-defined values.

- To create a new Data Type, click on the tab at the bottom of the Object Organizer window to open the Data Type tab.
- Right-click anywhere in the Data Type window and select Add Object. The New data type window will open.



About variables

Variables names can be up to 32 alpha-numeric characters long, they must start with a letter, may not include spaces and may include single underscores (_). The names are not case sensitive but may not be the same as any keyword.

All variables must be declared before they are used. They can be local to a POU or can be declared globally (see the resource tab).

We recommend variable names begin with a prefix that describes the scope and type of any variable. A list of suggested prefixes is included in appendix A.

Any time you uses a new variable in your program, the Declare Variable dialog will open automatically (this is sometimes a clue that you may have misspelled a previously declared variable).

You may also edit the declaration in the Declaration Window at the top of each POU.

You can re-open the Declare Variable dialog by selecting any variable in your program and selecting **Edit..Auto Declare** from the menu bar, or by pressing Shift-F2.

If you open the Declare Variable dialog without first selecting a variable, you'll be able to declare a new variable.

- Enter eSTATES for the name of the new type and then click OK. A Data Type editor window will open.

We now need to add code to define the allowable values for variables of this type. We will need variables for the Estop and Manual states, but while we're here, we'll also create values for the Homing and Auto states that we'll need later in the tutorial.

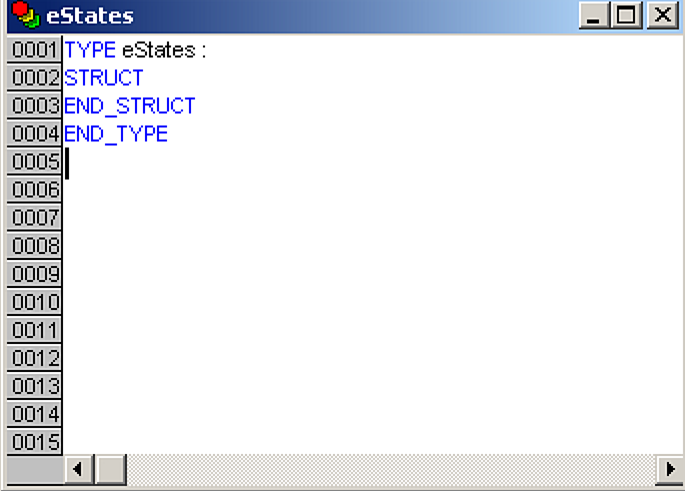


Figure 10, Initial eState ENUM Definitions

- Delete lines 2 and 3 (STRUCT and END_STUCT).
- Change the section between TYPE eSTATES: and END_TYPE as shown in Figure 11 below.

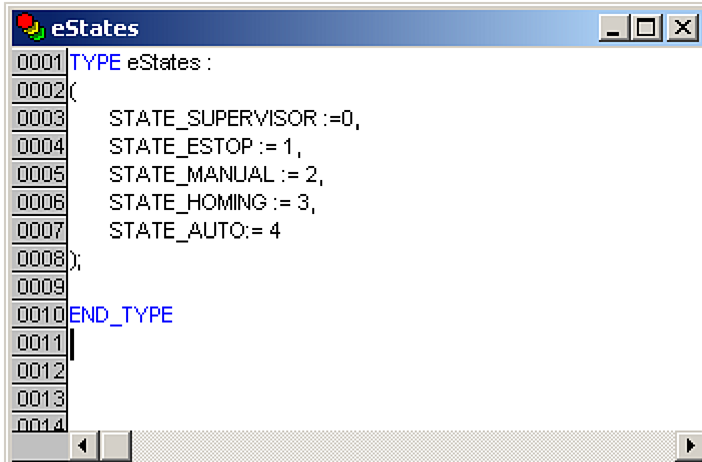


Figure 11, Completed eStates ENUM Definitions

We will want some global variables, to hold things like the name of the Knife axis and the current state of the system. By making them global, any task we create can access them.

- Click on the Resources icon in the Object Organizer Tabs.
- Expand the Global Variables folder as shown in Figure 12.

Figure 12, Global Variables

➢ Now double-click on Global Variables to open up the global variables.

➢ Now type in the variable definitions as shown in Figure 13.

Figure 13, Global Variables Window

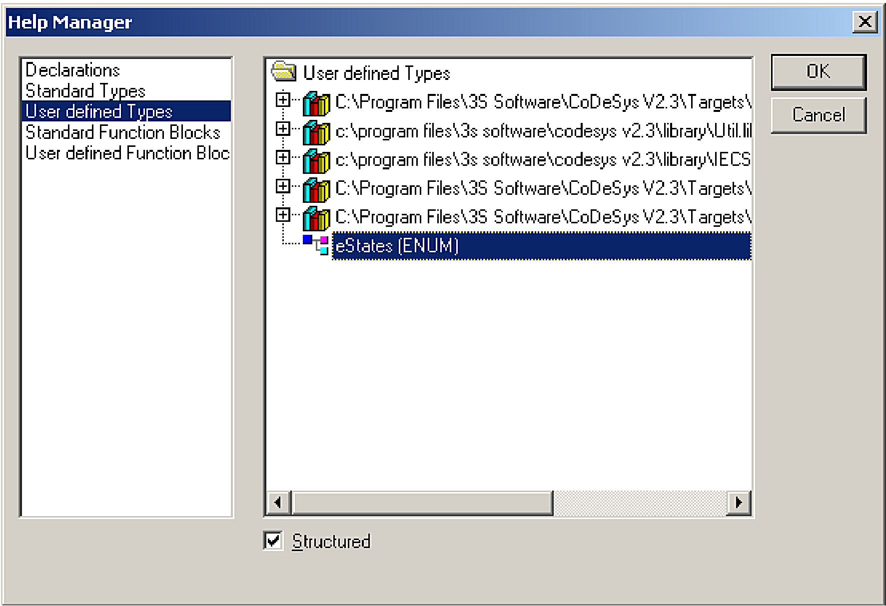


Figure 14, Help Manager

Instead of typing AXIS_REF and eSTATES, you can pick them off the list of available variables types. Simply type:

g_State:

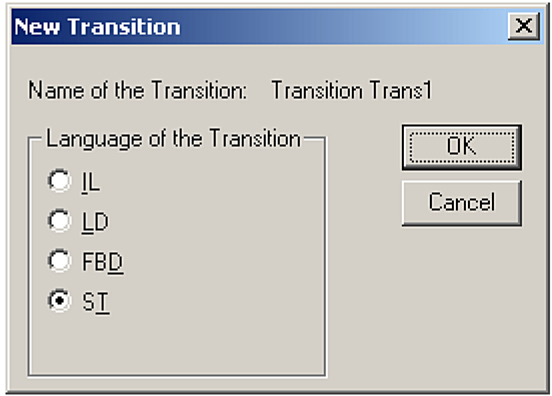
Then press F2, this will open the help manager. Shown in Figure 14.

Both of these types will be in the list of User Defined Types.

Select the one you want, in this case eSTATES, and press Enter.

Now we can use the g_State variable to set up additional transitions.

➢ Double-click on the Trans1 transition. The New Transition dialog will open.



- Set the Language of the Transition to ST and click **OK**.

The Structured Text Editor window appears.

- On line 0001 type: g_State = STATE_ESTOP
- Change the name of Trans1 to trnEstop
- Repeat the Process for Trans2 defining the transition as: g_State <> STATE_ESTOP
- Rename Trans2 to trnXEstop.
- Define the Trans3 transition as: g_State = STATE_MANUAL
- Rename Trans3 to trnManual.
- Define the Trans4 transition as: g_State <> STATE_MANUAL
- Rename Trans4 to trnXManual.
- Define the Trans5 transition as: g_State = STATE_HOMING
- Rename Trans5 to trnHoming.
- Define the Trans6 transition as: g_State <> STATE_HOMING
- Rename Trans6 to trnXHoming.
- Define the Trans7 transition as: g_State = STATE_AUTO
- Rename Trans7 to trnAuto.
- Define the Trans8 transition as: g_State <> STATE_AUTO
- Rename Trans8 to trnXAuto.

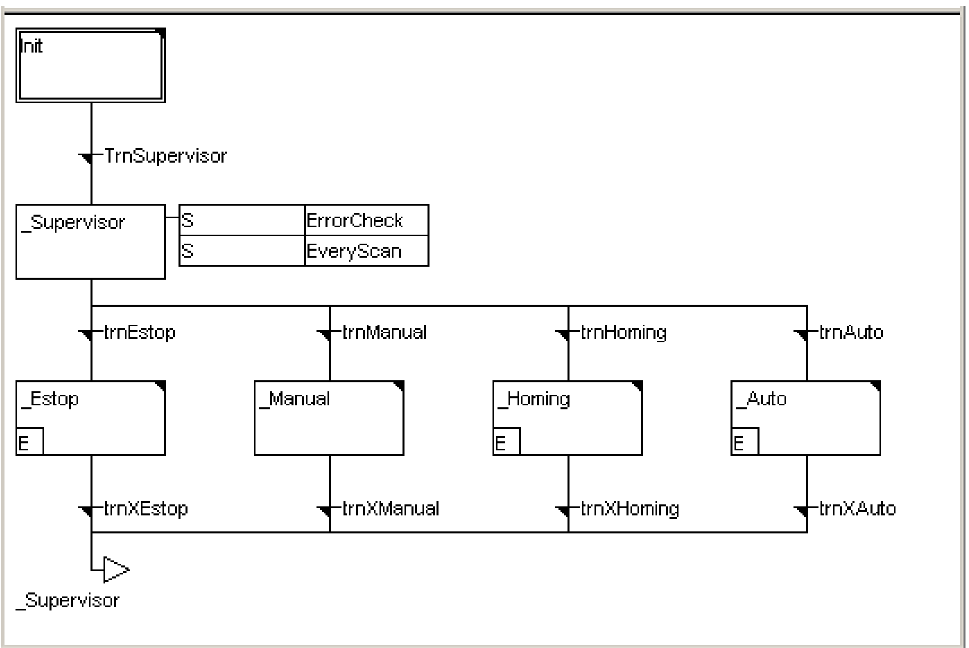


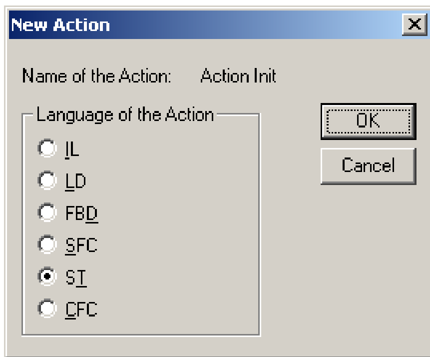
Figure 15, Initial SFC Diagram

The end result should look like Figure 15. You'll note in the transitions where we added ST code, there is now a black triangle. This symbol is used in any state or transition to indicate there is code that can be viewed by double-clicking.

Next we will set up some variables in the Init state and open communication with a drive.

6.6. Initialize the Jog Speed

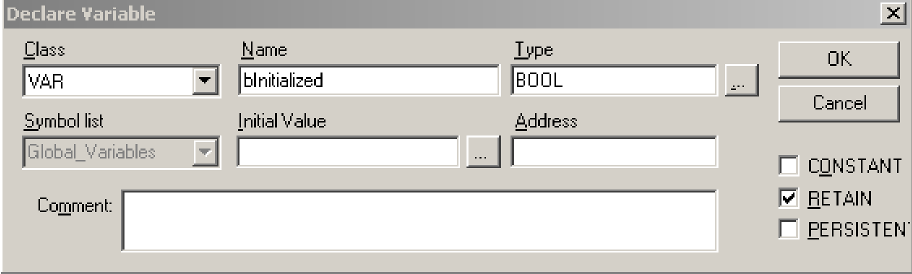
- Double click within the Init State and when the New Action dialog appears, select ST (Structured Text) and click OK.



Before we start to enter code, it's important to understand that any code you add, regardless of which language you're using, will execute over and over again until the exit condition, in this case when bInitDone becomes true. If there is any code you only want to execute once, you must add a condition that will only allow it to execute on the first pass.

- Type the following:

```
(* Initialize some variables *)
If NOT bInitialized THEN
```



- When the Declare Variables dialog pops up, set the values to:

  Class VAR Name bInitialized Type BOOL RETAIN checked

By checking the RETAIN box, the value of the variable will be remembered even when the SMLC is turned off and back on again.

We now need to set a value for the Jog speed.

- Add the following:

```
(* Initialize some variables *)
IF NOT bInitialized THEN
    diJogSpeed:= 20;
    bInitialized:= TRUE;
END_IF;
```

- This time, when the Declare Variables window pops-up, click the ... and select DINT (Double Integer) for the type.

  Class VAR Name diJogSpeed Type DINT RETAIN checked

> IEC-61131-3 is a strongly typed language. This means every variable must be declared and assigned a type before it can be used.
>
> To help identify the type of each variable, we recommend you prefix the variable names with characters that identify the type. We suggest "di" for double integer and "b" for Boolean.
>
> There is a complete list of the recommended type prefixes in Appendix A.

The IF statement makes sure we only set the value of diJogSpeed once when the program is first downloaded and run. If the value is changed while the program is running (using some kind of HMI), the new value will be remembered even when the SMLC is turned off. When it is turned on again, the retained value of bInitialized will be true so diJogSpeed will not be set back to the default value. This is how we can initialize variables when a program is first loaded but use the most recent value when the SMLC is power cycled. Any variables that are not RETAINed must be initialized any time the Init routine runs and should therefore be outside the IF / END_IF block.

- Now see if you can add three more variables inside the IF / END_IF block. The variables we need are diKnifeHomePosition, diMoveTime and tDwellTime. diKnifeHomePosition and diMoveTime will be of type DINT and tDwellTime of type TIME, all three variables should be RETAINed.
- To set the values for diKnifeHomePosition, diMoveTime and tDwellTime use:

```
diKnifeHomePosition:= 0;
diMoveTime:= 200;
tDwellTime:= t#1s;
```

6.7. Opening Drive Communications

Next we'll open communication with a ServoWire drive.

- Place the cursor on the next line in the ST editor and type:

```
(* open communication with the Knife drive*)
```

- Then with the cursor at the start of the next line, click **Insert…Function Block** on the menu bar.

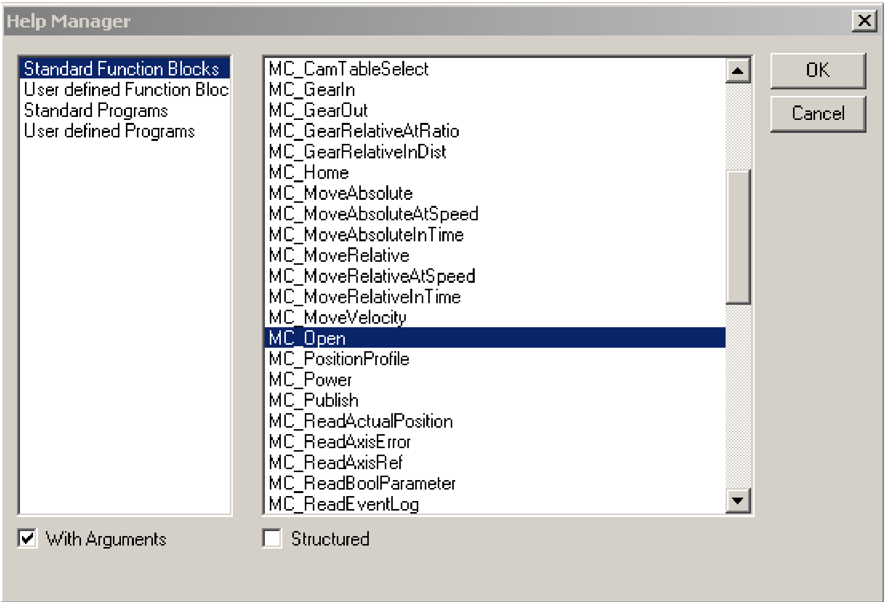


Figure 16, Help Manger Window

Function Blocks, Instances vs. Types

When a function block is first inserted, it is a variable of the type of the function block. Before you can compile and run the program, you must convert it to an ***instance*** of the function block. You do this by renaming it and assigning it the same type as the function block.

If you have several of the same type of function block in your program, you need to pay attention to whether they are the same or different ***instances***.

Function blocks with identical names are all the same ***instance*** of the function block. If the names are different, they are different ***instances***.

When a function block executes, the input and output variables are stored in a structure specific to that ***instance*** of the block.

The next time that type of function block executes, if it is the same ***instance***, any inputs will initially have the same as in the most recent occurrence of that ***instance***.

For example, with a MoveRelative function block, the motion is triggered by a change in the Execute input from false to true. So if your program executes one MoveRelative followed by another of the same ***instance*** or name, the Execute input will already be true so it will not see a transition from false to true and the motion will not occur.

- When the Help Manager dialog opens, make sure that Standard Function Blocks is selected from the list on the left and then clear the "Structured" checkbox at the bottom of the dialog.

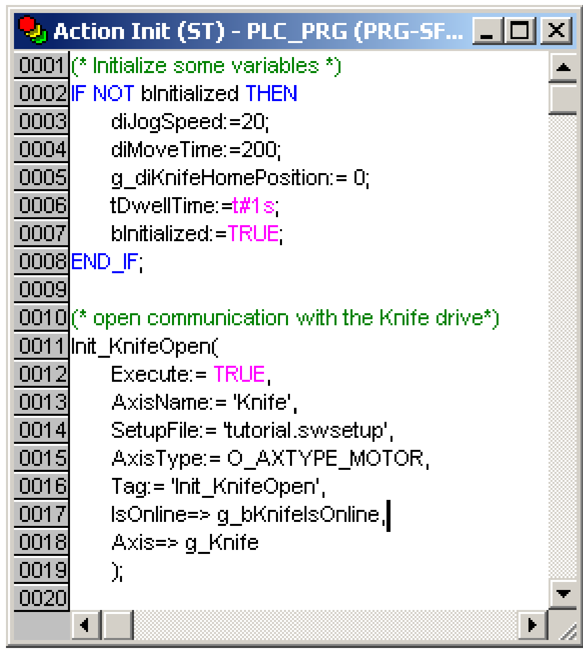
- Scroll through the list of available Standard Function Blocks on the right until you find MC_Open. Select it and then click OK. This will place the following line in your program:

```
MC_Open(
      Execute:= ,
      AxisName:= ,
      SetupFile:= ,
      AxisType:= ,
      Tag:= ,
      IsOnline=> ,
      Error=> ,
      ErrorID=> ,
      Axis=> );
```

The := and => symbols in the function block show where you will add the input and output variables.

The name MC_Open is the type of the Function Block; we must change the name to make it an instance of function block type MC_Open.

- Change the name of the function block from MC_Open to Init_KnifeOpen and click on the next line. We recommend you name function block instances in such a way as to identify the location of the instance in your program.
- When the Declare Variable dialog opens, click the ... icon, select Standard Function Blocks from the list on the left and then MC_Open in the box on the right.
- Click OK to continue.



Function Block Tags

Every ORMEC Motion Control function block has an input variable named ‘Tag’. Whenever execution of one of these function blocks generates an error, information about the error is automatically logged in the Error Log. The ‘Tag’ variable is used to identify which function block generated the error. You can set the Tag input to any text string you want. ORMEC’s recommendation is that you use the name and location of the function block instance, this will help locate the problem when trouble shooting.

Figure 17, Init Action

- Add input/output variables to the Init_KnifeMC_Open instance of the MC_Open function block as shown in Figure 17. Delete those inputs and outputs that are not used. As you add variables and constants, some of them will need to be declared:
- Declare IsOnline=> bKnifeIsOnline as:

  Class VAR Name bKnifeIsOnline Type BOOL

- Since we will not use the Error=> or ErrorID=> outputs, delete them.

Figure 17 shows how to open communications using Structured Text. The same operation using Ladder Diagram is shown in Figure 18.

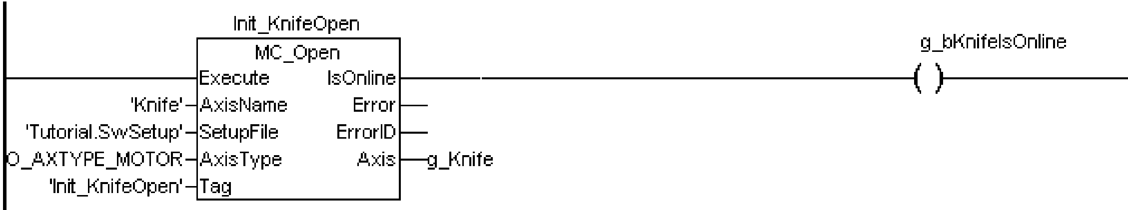


Figure 18, Opening an Axis Using Ladder Diagram

Axis References

The MC_Open function block returns an Axis Reference from the Axis output. The Axis Reference is the identifier used to specify an axis in any other motion or axis related function once the axis has been opened. The AxisName string is only used in the MC_OPEN function block.

6.8. Setting the Next State

After we have finished initializing everything we will want the program to go into the Estop state.

➢ On the next lines, add:

```
(* Force the system into the Estop state *)
g_State:=STATE_ESTOP;
```

6.9. Leaving the Init State

We have opened communication with the Knife drive but it may take a few moments to come online. We must wait until it comes on line and then we can set the transition variable, bInitDone and continue to the next state.

➢ On the next line, add:

```
(* Exit once the Knife drive comes on line *)
bInitDone:= g_bKnifeIsOnline;
```

The finished version of the Init state code should look like Figure 19.

```
Action Init (ST) - PLC_PRG (PRG-S...
0001 (* Initialize some variables *)
0002 IF NOT bInitialized THEN
0003     diJogSpeed:=20;
0004     diMoveTime:=200;
0005     g_diKnifeHomePosition:= 0;
0006     tDwellTime:=t#1s;
0007     bInitialized:=TRUE;
0008 END_IF;
0009
0010 (* open communication with the Knife drive*)
0011 Init_KnifeOpen(
0012     Execute:= TRUE,
0013     AxisName:= 'Knife',
0014     SetupFile:= 'tutorial.swsetup',
0015     AxisType:= O_AXTYPE_MOTOR,
0016     Tag:= 'Init_KnifeOpen',
0017     IsOnline=> g_bKnifeIsOnline,
0018     Axis=> g_Knife
0019     );
0020
0021 (* Force the system into the Estop state *)
0022 g_State := STATE_ESTOP;
0023
0024 (* Exit once the Knife drive comes on line *)
0025 bInitDone :=  g_bKnifeIsOnline;
0026
0027
```

Figure 19, Init State Actions

Help with variable names -The . key

When using previously defined variables, you can get help so you don't have to remember the exact spelling of each one.

For example, when you need to enter STATE_ESTOP, type State:=. and a window will pop up.

Continue typing what you remember of the variable, g_STATE_. The list of variables will scroll down to the first entry that matches what you have typed so far.

When you get close to the variable you need, use the up/down arrow keys or the scroll bar to find the exact one.

Once you've got it selected, double-click or press the TAB or ENTER key and it will be placed in your program.

Caution: Having entered a variable this way, you must go back and delete the ".". Forgetting to do this usually results in an "Expected Expression" error when you compile.

6.10. Programming the Estop State

We will program the Estop State using Ladder Diagram.

➢ Double-click on the Estop state, select LD and click OK.

The Ladder Diagram Editor will appear and the tool bar will change to display the Ladder Diagram icons.

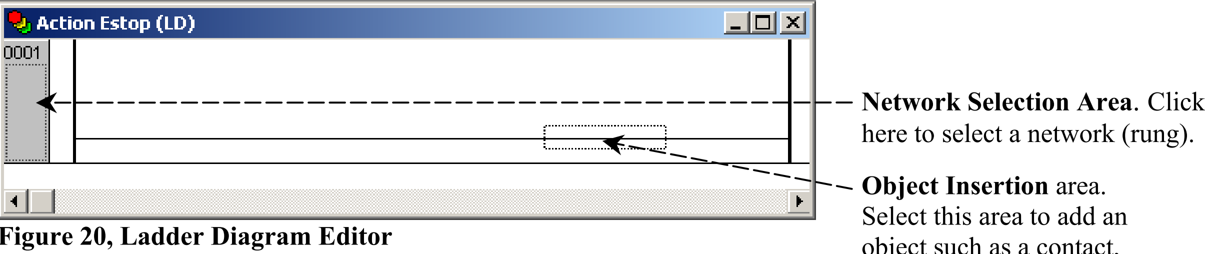


Figure 20, Ladder Diagram Editor

Within the Estop State we'll need to do the following:

1. Check the Reset push-button and when it is pressed, providing the Estop input is on, turn on a latched output to close the drive bus contactor.
2. When the bus contactor auxiliary contact input turns on, enable the drive after a ½ second delay to give the drive bus voltage time to stabilize.
3. If the drive is successfully enabled, change the program state to Manual.

6.11. Closing The Bus Contactor

Most real applications will have a main Bus Contactor that supplies power to the drives. We will need to close this contactor before we can enable the drives.

➢ With the Object Insertion Area selected, click on the Contact (input) icon in the tool bar.

A contact will appear at the left side of the rung and it will be given the temporary name ???.

➢ Rename the contact to hmi_bResetPB and declare it as a global Boolean variable.

Class VAR_GLOBAL Name hmi_bResetPB Type BOOL

We use the prefix 'hmi' to indicate the eventual source of the input will be an HMI screen.

➢ Select the Object Insertion Area again and insert another contact. This time rename it io_bESTOP_IN as a global Boolean variable. You should have already declared this variable on page 19.

Class VAR_GLOBAL Name io_bESTOP_IN Type BOOL

➢ Select the Object Insertion Area and click the icon to insert a coil (output).

➢ Change the name of the coil to g_bMC1_OUT. You should have already declared this variable on page 19.

Class VAR_GLOBAL Name io_bMC1_OUT Type BOOL

➢ Right-click on the g_bMC1_OUT coil and select S/R. This will turn the coil into a Set coil. The coil will be set true if there is power flow to it but will remain true when the power flow is cut off. This output would normally be connected in series with the machine's Emergency Stop circuit to control the drive Bus Power.

Set/Reset Coils

These coils are latched. After you set the coil with a (S) it will remain on, regardless of the power flow on that rung until the program executes a reset (R) of the same coil. This is usually done in a different rung or maybe a different SFC action.

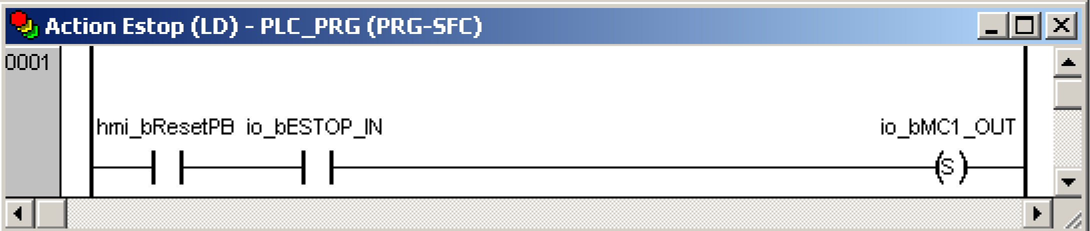


Figure 21, Closing The Bus Contactor

6.12. Enabling The Drive

When the main contactor (MC1) has closed, we'll want to power up (enable) the Knife drive.

- Add a new network with an io_bMC1_OUT contact.
- Right-click on the Object Insertion Area and select "Function Block". Select "User Defined Function Blocks", then "Utility Functions", "Initialization Function" and finally "OrmMultiAxisPower". If you don't see these options, you need to install the ORMEC Utility Functions Package, which should be on your installation CD. Read the box below to see how get and install them.

Figure 22, Powering Up the Drive

- Rename the function block Estop_Enable and add the inputs shown in Figure 22.
- Add the output coil hmi_bBusPwrLT as a global Boolean variable

> **Note:**
>
> In a real application you would use an auxiliary contact from the MC1 contactor in the Enable input of the Estop_Enable function block to ensure the contactor is closed before attempting to power up the drive.

The Estop_Enable function block will wait the time specified on the Delay input and then clear any faults and enable each axis with valid references on the Axis1 through Axis8 inputs. Next we need to change the state of the machine to Manual.

Utility Functions

The PLCopen standard for IEC-61131 motion control function blocks defines a minimum set of motion control function blocks that should be provided. This set includes the basic functionality needed for effective motion control. However because the PLCopen set provides primitive functions it can take several networks to accomplish relatively simple things like enabling drives and jogging motors.

ORMEC provides a collection of Utility Functions and Function Blocks, built using the PLCopen FBs, so these operations can be programmed as a single network. The ORMEC Utility Functions are include on the installation CD

You may also download these libraries from ORMEC's User Web site at http://www.ormec.com/user/. Go to the "Download Area" and click on "Download Area" for SMLC" products. Then select "SMLC Utility Functions". Save the libraries in your \Program Files\3S Software\CoDeSys\Targets\ORMEC SMLC\ folder.

- Right-click on the network 0003 Object Insertion Area and select "Box with EN". This will add an AND function in parallel with the hmi_bBusPwrLT coil.
- Change the AND to MOVE.
- Now make the input to the MOVE block STATE_MANUAL and the output g_State. When the Done output from the Estop_Enable function block becomes true, the value of g_State will be set to STATE_MANUAL.

Figure 23, Completed Estop state action

6.13. Estop Entry Action

We have one thing left to do to complete the Estop state. We need to make sure that any time the program enters the Estop state; the MC1 bus contactor output is turned off and the Drive is disabled.

To do this we'll use a feature of SFC that allows us to create code or networks that run only once when we first enter the state.

- Go to the Object Organizer window and double-click on PLC_PRG to re-open the SFC diagram.
- Right-click anywhere in the Estop state and select Add Entry-Action. For the language, select LD.
- Add a contact and name it FALSE, This will then be an always OFF contact.
- Add an OrmSingleAxisPower function block and name it Estop_Enable. This will make it the same instance of the OrmSingleAxisPower block we used in the main Estop action ladder diagram.

- Add a coil and name it bJunk (we don't need the Status output for anything, but an output coil is required).
- Add a network after
- Now add a coil and name it io_bMC1_OUT.
- Select the coil and click twice on the [S R] icon in the toolbar.

The result should look like this:

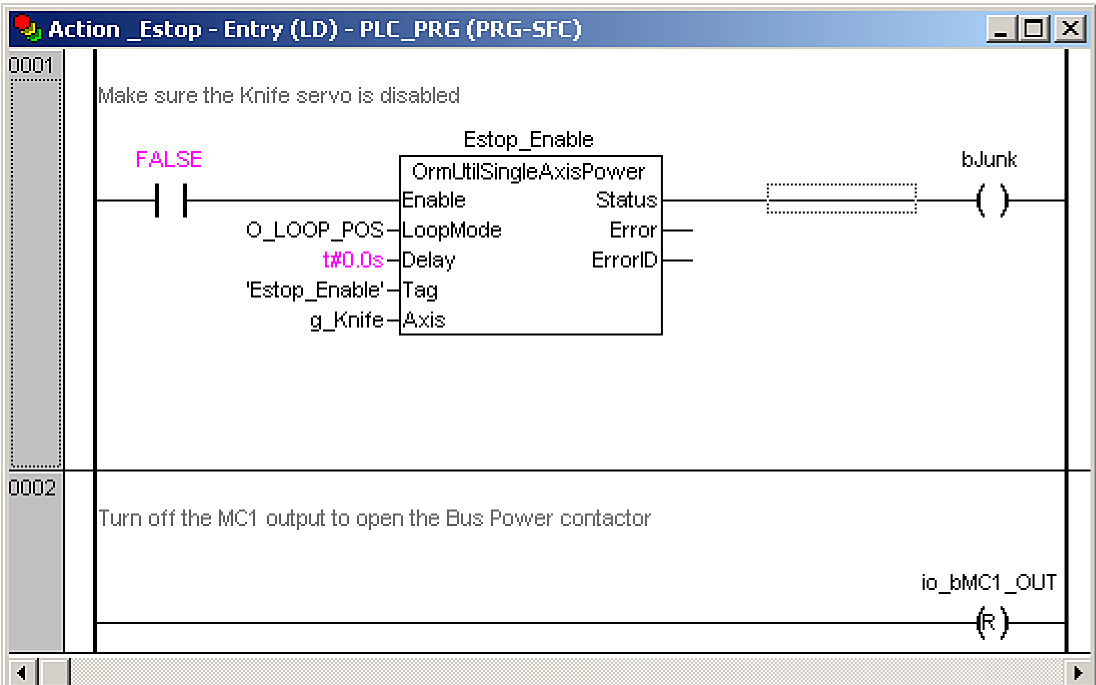


Figure 24, Estop Entry Action

When the program enters the Estop state it will make one pass through this network and will disable the Knife servo and turn the Bus Power contactor off.

At first sight you might think you could place the io_bMC1_OUT reset coil in the status output of the Estop_Enable function block. While this would work, it would also make opening the bus contactor dependent on the success of that block. Given the safety implications, it is probably not a good idea.

6.14. Ladder Diagram Comments

If you open the pre-completed tutorial project and look through the Ladder Diagrams, you will see that many, if not all, of the rungs have comments that describe what the rung does. Thorough commenting is always a good idea regardless of which of the IEC-61131-3 languages you're using.

To add a comment to a rung of a Ladder Diagram program do the following:

- Right-click in the rung selection area and select comment.

The word Comment will appear in the top left corner of the network.

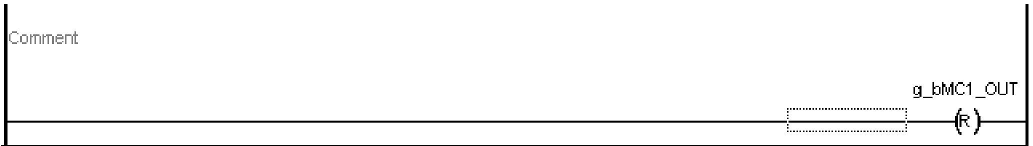


Figure 25, Adding a Comment to a Ladder Network

- Select the word Comment and change it to whatever you want to say.

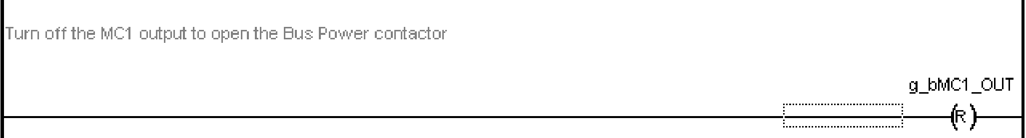


Figure 26, Completed Ladder Network Comment

- Go through the Estop and Estop Entry action programs and add appropriate comments.

Comments and Linebreaks

You can control how many lines are available for rung comments by selecting Extras the Options from the menubar.

You can also add comments on individual contacts and coils this can be useful for identifying terminal and wire numbers associated with the coil or contact. Be wary of using long multi-line comments on coils or contacts since your program can rapidly become very cluttered.

Finally you can select "Networks with linebreaks". This causes rungs that are too long to fit in the window to be wrapped around on multiple lines.

7. Testing Your Work So Far

Every now and then, you'll want to test your work to find and correct any structural or syntactical errors you may have made. To do this press F11 to compile you program. The Message Window (see Figure 3) will show you if there are any errors.

If you don't see the Message Window on your screen, click Window on the Menu bar and make sure that Messages is checked.

If there are no errors in your program, the message window will look like this:

```
Hardware-Configuration
0 Error(s), 0 Warning(s)
```

If there are problems, the message window will look something like this:

```
Interface of POU 'PLC_PRG'
Data allocation
Check task configuration
Implementation of POU 'PLC_PRG'
Error 4053:    Action _Estop - Entry (1): 'Estop_Enable' must be a declared instance of function block 'OrmUtilSingleAxisPower'
    Action Init
    Action _Estop
Error 4053:    Action _Estop (2): 'Estop_Enable' must be a declared instance of function block 'OrmUtilSingleAxisPower'
    Action _Manual
    Action _Homing
    Action _Auto
Implementation of POU 'PLC_PRG.EveryScan'
Implementation of task 'HMI_PRG'
Implementation of task 'PLC_PRG'
Check of the parameter configuration
Hardware-Configuration
2 Error(s), 0 Warning(s).
```

This message is telling us there is an error in network 1 of the _Estop-Entry action and network 3 of _Estop ladder diagrams. For some reason, probably an invalid declaration[1], the compiler doesn't recognize OrmSingleAxisPower as a valid function block.

Pressing F4 will take you directly to the location of the first error in your program. Pressing F4 again will then take you to the next error.

Two cautions:

1. While the compiler may detect an error in a particular location, the actual error may be elsewhere. In the above example, the compiler detected a problem in rung 1 of the _Estop Entry action. The actual error however, was in the PLC_PRG Variable Declaration section which set the incorrect type for a function block used in the above rung.
2. If after compiling, you correct an error in the program by adding or deleting a line, subsequent use of F4 may not take you to the exact location of the next error. This is due to the fact the compiler stores the error locations when you compile. When you delete or add a line, it does not adjust where it thinks the next error is. Re-compile and use F4 again.

---

[1] This error was created by incorrectly declaring *_Estop_Enable* as type 'BOOL' instead of type 'OrmMultiAxisPower'. Incorrectly declaring a variable or function block as type 'BOOL' is a common mistake since BOOL is the default type.

7.1. Compilation Problems

Normally when you compile, the system only re-compiles those POU's it believes have changed since the last compile. Occasionally, some types of programming errors or computer crashes may cause it to become confused as to which POU's have changed. If your program is behaving strangely, select "Project – Clean All" and then "Project – Rebuild All" from the menu bar. This will delete all the previously compiled files and intermediate files and then recompile all of them.

7.2. Unused Variables

As you work though your program and correct errors, you may end up with variables you declared but do not actually use in the program. Clicking "Project – Check – Unused Variables" will display a list of such variables in the Message Window. Double-clicking on any variable in the list will take you to the place where it is declared so you can change or delete it. If you double-click on an unused variable then delete it, the compiler will not know where the next unused variable is. If you aren't paying attention, you may delete the wrong variable.

Tip!

To avoid the problem described here, delete unused variable declarations using the backspace leaving the end -of -line carriage return.

7.3. Running the Program

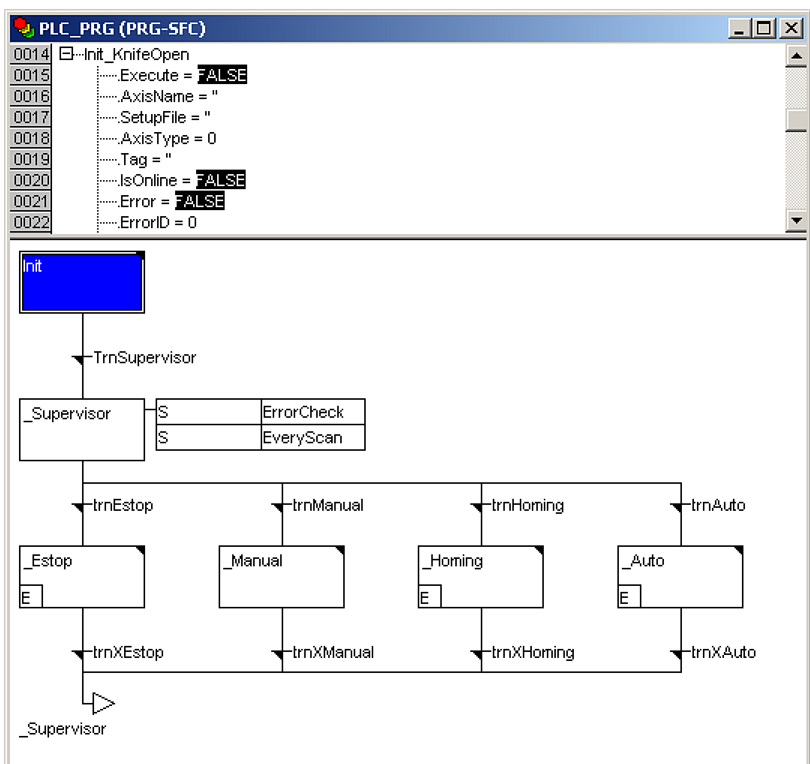


Figure 27, Online

Tip!

The variables show up in the order they appear in the Variable Declaration window. You can make it easier to find a specific variable by arranging them in alphabetical order before compiling. An easy way to do this is to copy and past them all into a spreadsheet program, use the spreadsheet Data - Sort function to sort them then paste them back into the Declaration window.

You can run the program even if you don't have an SMLC controller. After you've successfully compiled your program without any errors or warnings, double-click on PLC_PRG in the Object Organizer window to show your SFC program.

If you don't have an SMLC, go to the "Online" menu and make sure "Simulation Mode" is checked. Click on the Online – Login item on the Menu Bar. If you have an SMLC, this will log your computer on to the controller. If

you don't have an SMLC, it will log you into the simulator. You can also press Alt-F8 or the icon instead of using the Menu Bar. The view of your program will change to look like this:

The upper part of the window shows a list of all of the variables and function blocks in your program, along with their values. The function blocks have a box with a plus sign. If you click on the plus sign, it will expand the list to show all the inputs and outputs to the block.

At the present time, your program is not running so all the variables will be in their un-initialized states. In the bottom part of the window, the solid blue box shows where the program is in its execution.

- Now click on the icon (or press F5) to start the program. You should see the solid blue block move down to Supervisor and then to the Estop state. The ErrorCheck and EveryScan actions will also turn blue indicating they are active.

Now let's try enabling the drive.

- First, press Shift-Esc to close the Message Window.
- Then double-click on the Estop state to view the Action Estop(LD) Ladder Diagram.
- Drag and size the windows until your screen looks something like that shown in Figure 28.

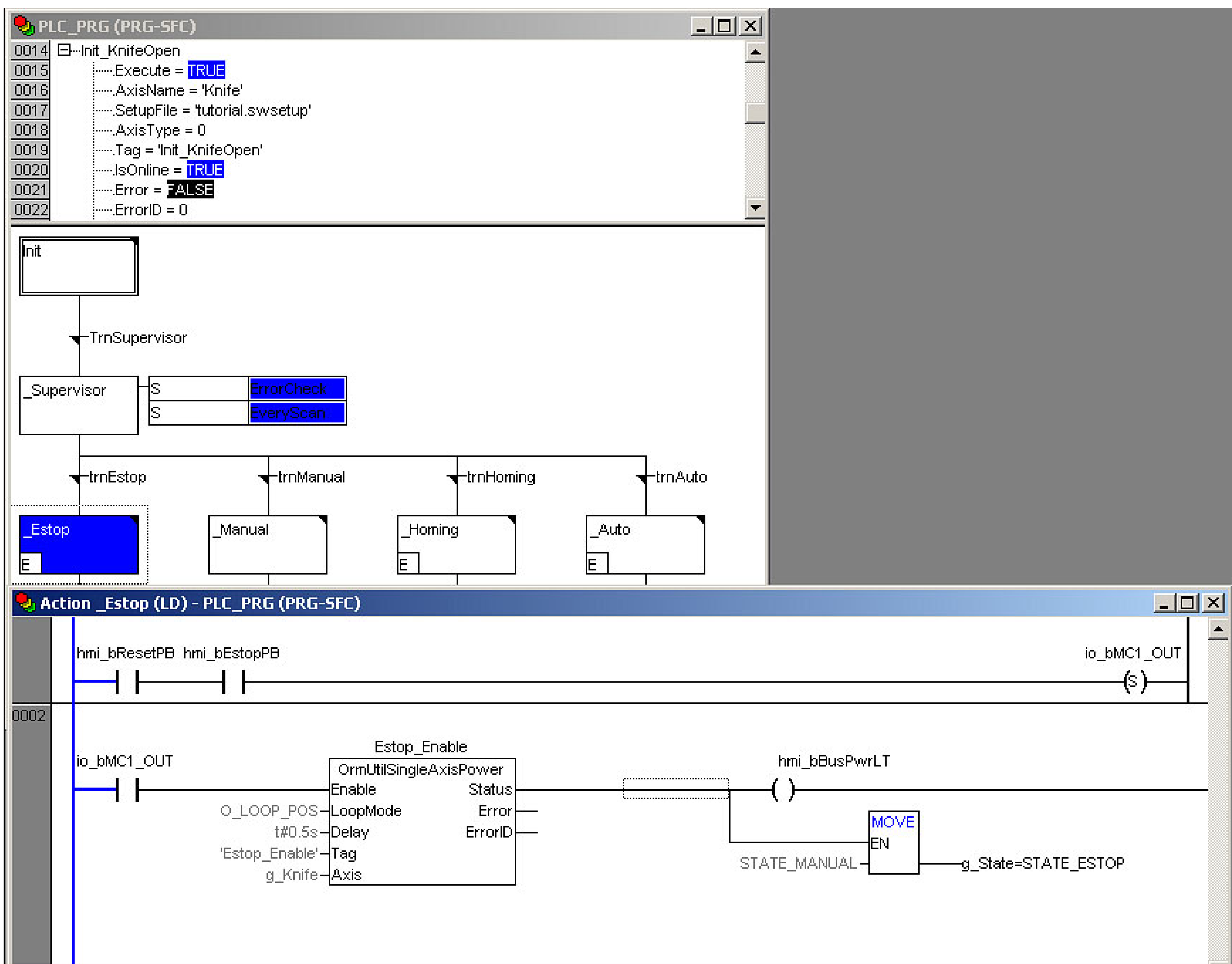
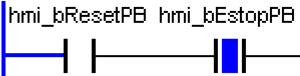


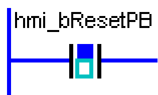
Figure 28, Program Running

- Double-click on the io_bEstopPB contact on the first rung of the Action Estop window, the display should change as shown below. The dual tone blue square indicates you are preparing to write true value to the contact:

- Watch the screen carefully, and press Ctrl-F7 to accept the new value. The contact now looks like this indicating the contact is true:
- You should see the power flow change and the State on rung 2 change to STATE_MANUAL. The contact now looks like this indicating the contact is true:



- Now repeat the process using the hmi_bResetPB contact. When you press Ctrl-F7, watch the ladder diagram closely. You should see the power flow change and the State on rung 2 change to STATE_MANUAL:
- Look at the PLC_PRG(PRG-SFC) window, you should see the Manual state is now solid blue indicating it is the active state.
- Double-click on hmi_bResetPB again. The empty blue square indicates you have written false to the contact:



Writing and Forcing Values

You can write or Force values anywhere the variable appears in a code or declaration window.

If you implement the written value using Ctrl-F7, the new value is temporary and will be changed any time the program, a physical input or HMI writes a new value.

If you implement it with an F7, the new value is "Forced" and will remain in place until you remove the Force.

- Press Ctrl-F7 to implement the write.
- Stop the program using the red stop sign on the toolbar and then, click on the Online – Reset item in the menu. This will reset the program to its starting point. Run the program again.

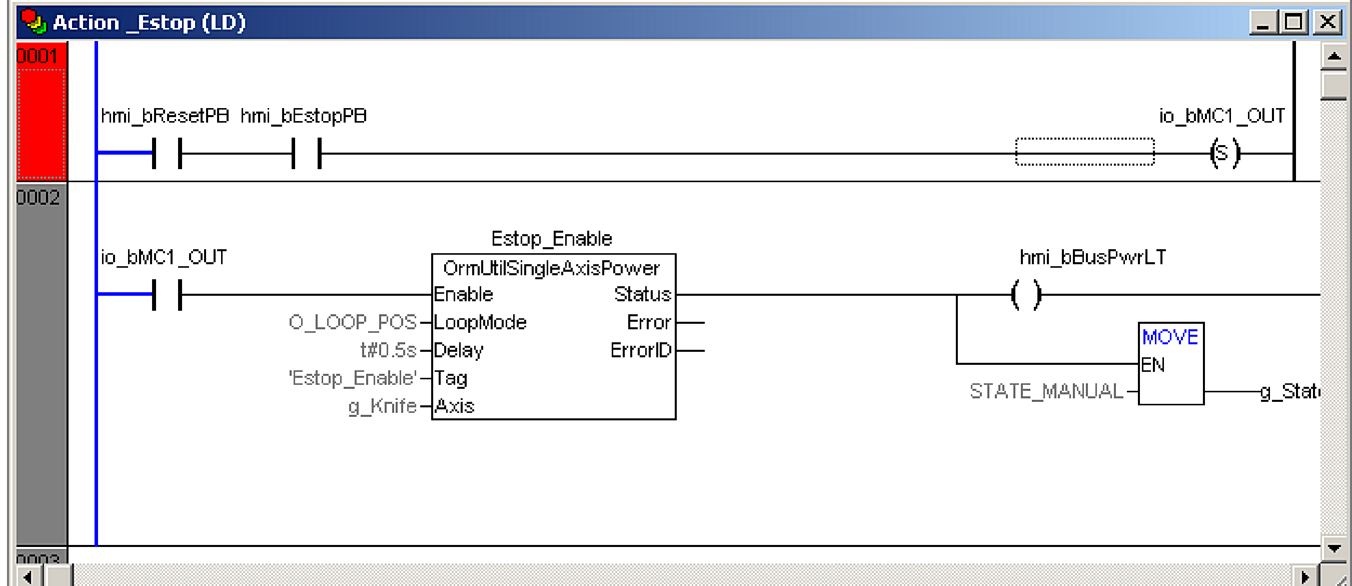


Figure 29, Network with program stopped on a break point

- Click just below the 0001 to set a break point at that network .
- The margin should turn red indicating the program has reached the break point and stopped before executing the rung.
- Set the hmi_bResetPB and io_Estop_PB variables true again.
- It may take at least two scans through the networks (due to the ½ second delay in Estop_Enable) before the State changes to STATE_MANUAL.
- F5 will cause the program to quit stepping and run (until the next breakpoint).
- To remove a breakpoint, just click below the network number again or click “Online” and “Breakpoint Dialog”.
- Click on the icon (or press Ctrl-F8) to logout and return to offline mode.

8. The Supervisor State

The Supervisor state is a transitional state; the program will enter this state and immediately move on to another state. There is no need for code in the Supervisor State.

9. Manual State

From this point on, I'll assume you are able to look at a ladder diagram using contacts, coils and function blocks and figure out how to create it.

➢ Add the ladder diagram shown in Figure 30.

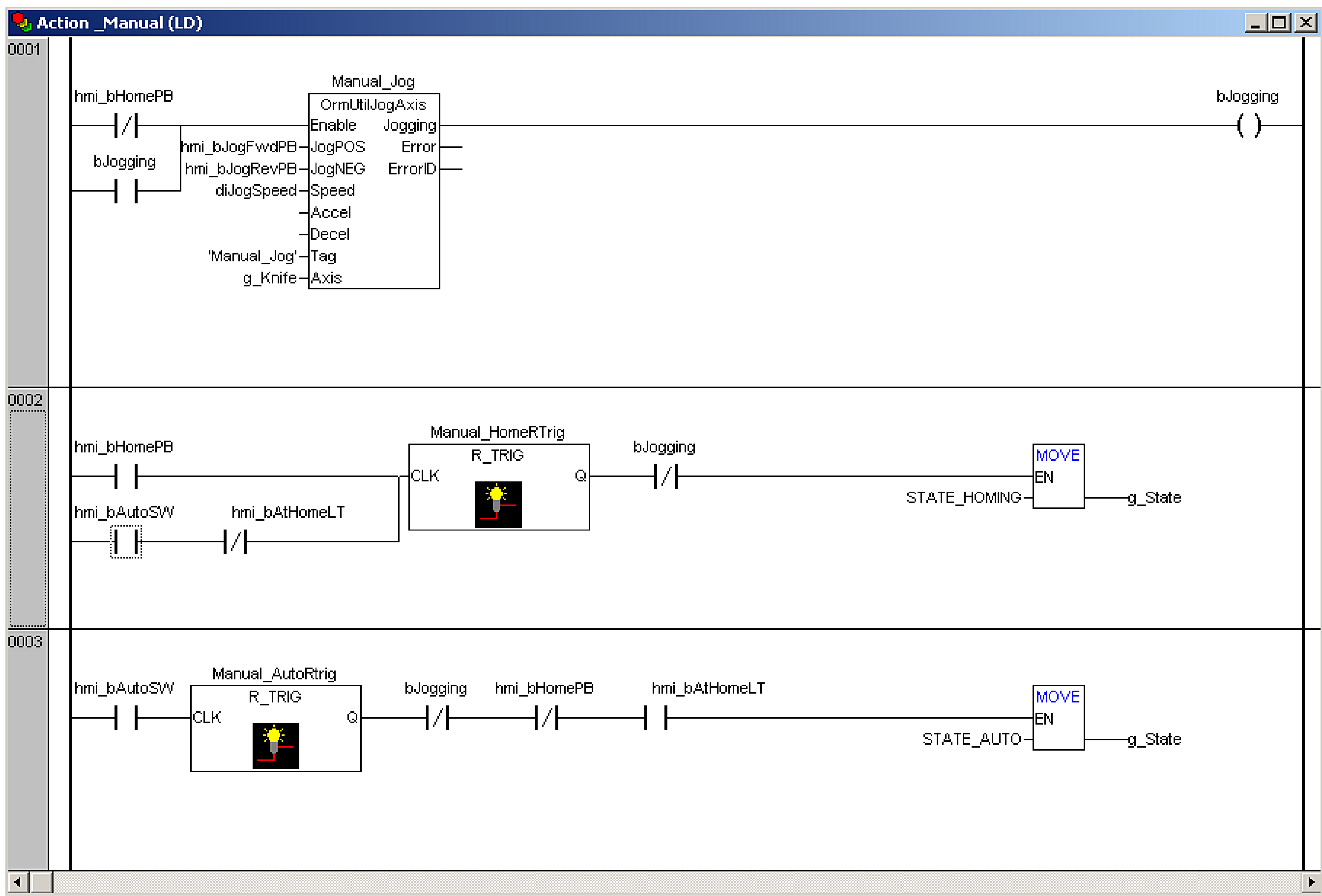


Figure 30, Manual State Action

The Normally Closed (N/C) hmi_bHomePB contact on the Enable of the Manual_Jog function block will prevent jogging if the Home push button is pressed. When either Jog input sees a rising edge, the axis will jog in the appropriate direction and the Jogging output will be set true. When the Jog (or Enable) inputs return false, the axis will stop and the Jogging output will become false.

The Normally Open (N/O) bJogging contact seals the Enable input true while the axis is actually jogging. The bJogging N/C Jog contacts in rungs 2 and 3 prevent state changes while the axis is jogging.

There are two ways to home the knife, both require bJogging to be false. One way is if hmi_bHomePB becomes true, the other is if hmi_bAutoSW becomes true while the knife is not already at its home position. This allows the knife to automatically home when the auto switch is turned on.

The Rising Edge Trigger (R_Trig) prevents the system from entering the homing state if either of these conditions become true while jogging and the operator releases the jog button while the conditions are still true. It must see a rising edge while not jogging.

If the hmi_bHomePB is off, the Knife is not jogging and is at its home position; a true value on the hmi_AutoSW will send the program to the Auto state.

Once you've added the Manual ladder diagram and successfully compiled it, you can go online and test it. If you have an SMLC controller and drive, you should be able to make motors move by writing to the jog pushbuttons using Ctr-F7 as we did in Section 7.3. You should note that if you are able to get the program into the Homing or Auto states, until we add code there, you will have to write to the g_State variable to get out.

9.1. Homing State

When a servo drive is powered up, it does not know its current position[2]. Before the system can enter the Auto state we need to execute the OrmUtilHomeToSensor function block so the motor can calibrate its exact position. OrmUtilHomeToSensor moves the motor at a set speed until the Home Sensor input is asserted. The Home Sensor input could be an external drive input or the motor's once-per-rev marker. For the tutorial, we'll use the motor's once-per-rev marker. Within 10 micro-seconds of the input's assertion, the drive will store the current position of the motor. It will then stop the motor using the set deceleration. Once the motor has stopped, the function block waits the set time delay to allow the motor and its load to settle. It then sets the position of the motor to the value set by the SensorPosition input using the captured position to compensate for the distance traveled during the deceleration. Finally it moves the motor to the position set by the HomePosition input. After OrmUtilHomeToSensor has completed successfully, the drive knows the correct position of the motor to an accuracy equal to the distance the motor travels in 10 micro-seconds[3] at the speed you use for homing, and is at the Home Position.

➢ Add the ladder diagram shown in Figure 31 to Homing State action.

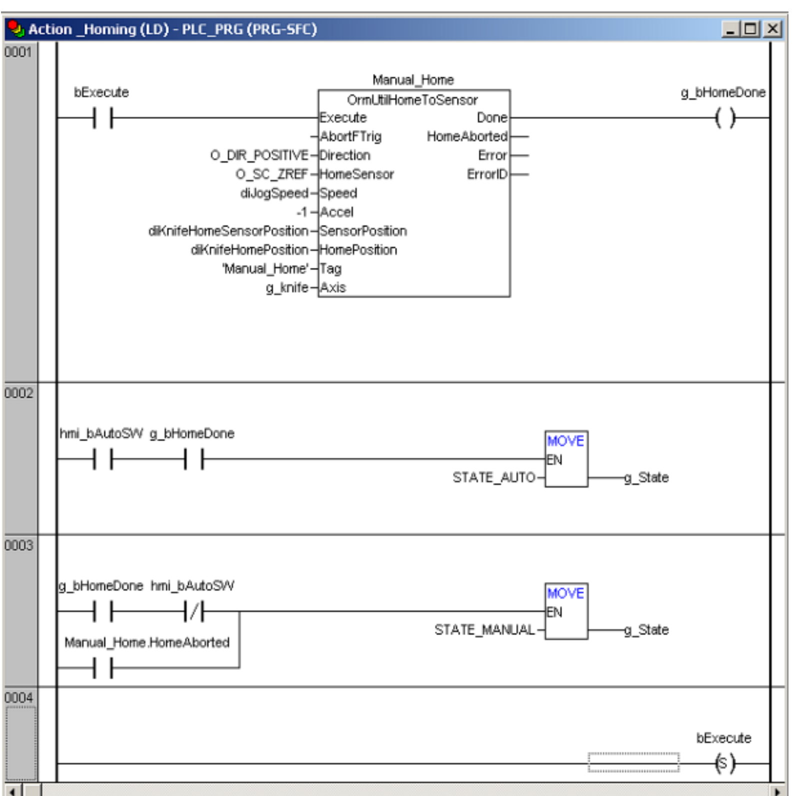


Figure 31, Homing State Action

Here's how the code works. On the first pass through the code, bExecute is false so the Manual_Home FB is not executed. bExecute is set true at the bottom of the diagram so, on the next pass, the Manual_Home FB will see the rising edge it needs on its Execute input in order to start. When it finishes, its Done output is turned on.

> **AbortFTrig input**
>
> If you want to force the operator to hold his finger on the hmi_HomePB throughout the homing process, you can add hmi_HomePB to the AbortFTrig input. When AbortFTrig sees a falling edge while Manual_Home is executing, the motor will stop and the HomeAborted output will be set true. So if the operator takes his finger of the Home button while homing, the motor will stop. If you don't want this to happen, i.e. once the Home button is pressed, you want it to complete the homing sequence regardless of whether the operator continues to press the button, simply leave the AbortFTrig input blank.

[2] Unless it is equipped with ORMEC's absolute encoder option.

[3] There may be additional input latency depending on the type of sensor used for the stop input.

In the Manual state code, the rising edge trigger after the hmi_HomePB contact prevents the system from re-entering the Homing state if the operator still has his finger on the Home button when the system returns to the Manual state.

9.2. Adding the Homing State Entry Action

We will need to add some code to make sure bExecute is false when we first enter the Homing State. To do this we will add an Entry action that is only execute once when we first enter the Homing State.

- Right-click on the Homing State and select "Add Entry-Action".
- Set the language to LD and then enter the network shown in Figure 32 below.

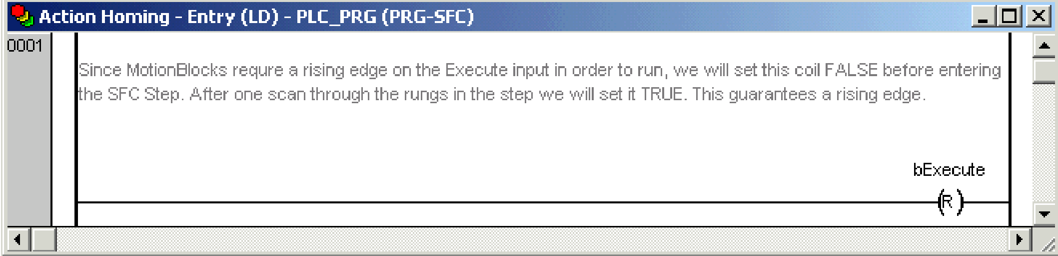


Figure 32, Homing Entry Action

You should note that the Homing Step now has a box containing an "E" in its bottom right-hand corner indicating that an entry action has been programmed. You can edit an entry action by double clicking on the "E".

10. Adding the EveryScan Action

We will need a way to tell whether the knife has completed the homing sequence and is actually at its home position before we allow it to enter the Auto state. Since we would also like to be able to constantly update and indicator on the operator's HMI whenever the knife is at its Home position, the best place to do it is in the EveryScan action.

To simply the EveryScan action code, we'll create a User Defined function in structured text and then simply call it in the EveryScan action.

10.1. Creating the IsAtHome Function

We could create this function using any of the supported languages, since most of the program has been written in Ladder Diagram so far, we'll use Structured Text to give you an idea of how to use it.

➢ Right-click on POU's in the Object Organizer window and select "Add Object"

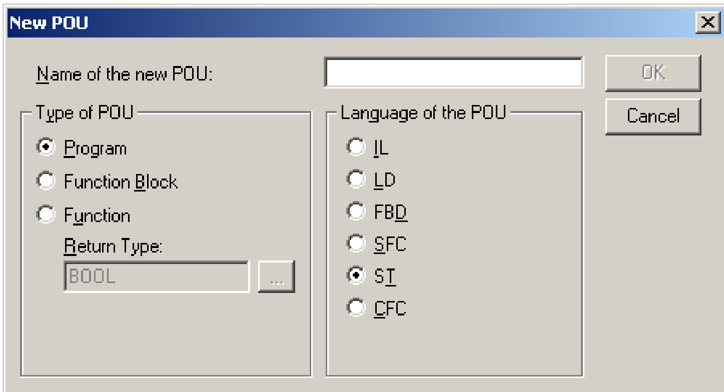


Figure 33, New POU Window

➢ In the "New POU" dialog, give it the name "IsAtHome". Set the type as "Function" and the Language as "ST". Leave the Return Type as "BOOL".

The function will need to check several things.

1. Is the axis moving?
2. Is the axis enabled?
3. What position is the axis being commanded to?
4. What is the axis position lag alarm setting?
5. Is the axis position lag less than the lag alarm setting?
6. Has the axis been homed?

If the axis is:

- Stopped
- Enabled
- Being commanded to the home position
- Has a position error (lag) less than its lag alarm setting, and;
- Has been homed

The axis is stopped at its home position.

We'll make Axis, Homed and Home Position inputs to the function rather than use global variables. This way, the function becomes re-useable and can be used for other axes in this project or in other projects.

➢ In the VAR_INPUT section of the function, create the following variables:

```
VAR_INPUT
    Axis: AXIS_REF;
    Homed: BOOL;
    HomePosition: DINT;
END_VAR
```

```
VAR_INPUT
    Axis:AXIS_REF;
    Homed: BOOL;
    HomePosition: DINT;
END_VAR
VAR
    IAH_InMotion: MC_ReadBoolParameter;
    IAH_Lag: MC_ReadParameter;
    IAH_LagLimit: MC_ReadParameter;
    IAH_Mode: MC_ReadParameter;
    IAH_PosCmd: MC_ReadParameter;
END_VAR
(* Is the axis moving? *)
IAH_InMotion(
    Enable:= TRUE,
    ParameterNumber:= OP_INMOT,
    Tag:= 'IAH_InMot',
    Axis:= Axis,
    );
(* In what  mode is the axis? *)
IAH_Mode(
    Enable:= TRUE,
    ParameterNumber:= OP_LOOP_MD,
    Tag:= 'IAH_MODE',
    Axis:= Axis,
    );
(* Where Is the axis supposed to be? *)
IAH_PosCmd(
    Enable:= TRUE,
    ParameterNumber:= OP_POS_CMD,
    Tag:= 'IAS_PosCmd',
    Axis:= Axis,
    Done=> ,
    );
(* Read Lag Alarm Limit setting *)
IAH_LagLimit(
    Enable:= TRUE,
    ParameterNumber:= OP_LAG_ALARM,
    Tag:= 'IAH_LagLimit',
    Axis:= Axis,
    );
(* Read the position lag *)
IAH_Lag(
    Enable:= TRUE,
    ParameterNumber:= OP_LAG,
    Tag:= 'IAH_Lag',
    Axis:= Axis,
    );
(* Now set the return value based on the results from each FB *)
```

Figure 34. IsAtHome Function

➢ In the code area, add an MC_ReadBoolParameter. To do this press F2 select "Standard Function Blocks" in the left window and navigate to the SMLC Admistrative Blocks in the right window

```
MC_ReadBoolParameter(
                Enable:= ,
                ParameterNumber:= ,
                Index:= ,
                Tag:= ,
                Axis:= ,
                Done=> ,
                Error=> ,
                ErrorID=> ,
                Value=> )
```

➢ Change the name of the Function Block to IAH_InMotion and add the variable names as shown below. Remember to delete the ones we're not using. Make sure you select a variable type of "MC_ReadBoolParameter" from the Help Manager (F4) when you rename the function block.

```
IAH_InMotion(
  Enable:= TRUE,
  ParameterNumber:= OP_INMOT,
  Tag:= 'IAH_InMot',
  Axis:= Axis,
  );
```

➢ Continue to build the code until it looks like the example in Figure 34.

Accessing Function Outputs

Note the way IsAtHome accesses the results of the various function blocks it uses. Rather than assigning the results to variables as it execute each Function Block, it simply executes them and then refers to the results at the end of the code. For example IAH_InMotion, returns a Boolean result in its Value output. The IF statement at the end of IsAtHome refers to this as IAH_InMotion.Value.

Next we need to add code to the EveryScan action to call IsAtHome. By adding it to EveryScan, it will be executed at the start of each program scan. This eliminates the need to call it every time we need it. However, we must remember to protect the program from things that might change the IsAtHome status during a scan. For example, at the start of the scan, the Knife may be at home, but if a Jog button is pressed, we don't want to change the state to auto. This is why we include a bJogging contact in rung 3 of the Manual state.

- Right-click on PLC_PRG (PRG) in the Object Organizer window and select "Add Action" from the menu. Type the name EveryScan and select LD for the Language.
- Add the rung shown in Figure 35.

Functions vs. Function Blocks

There are several differences between functions and function blocks. Which you use when you need to create a user defined operation, will depend on your needs.

Functions

- Return only a single value.
- Have no memory, for a given set of inputs, they always return the same result
- Should be inserted as a "Box with EN".

Function Blocks

- May return multiple values.
- Have internal memory. When called, the internal variables will have retained their values from the previous execution. So the result may vary depending on previous executions even if the inputs don't change.
- The code for a function block is executed regardless of the state of the Enable or Execute input. Your code must handle the case when these inputs are false.

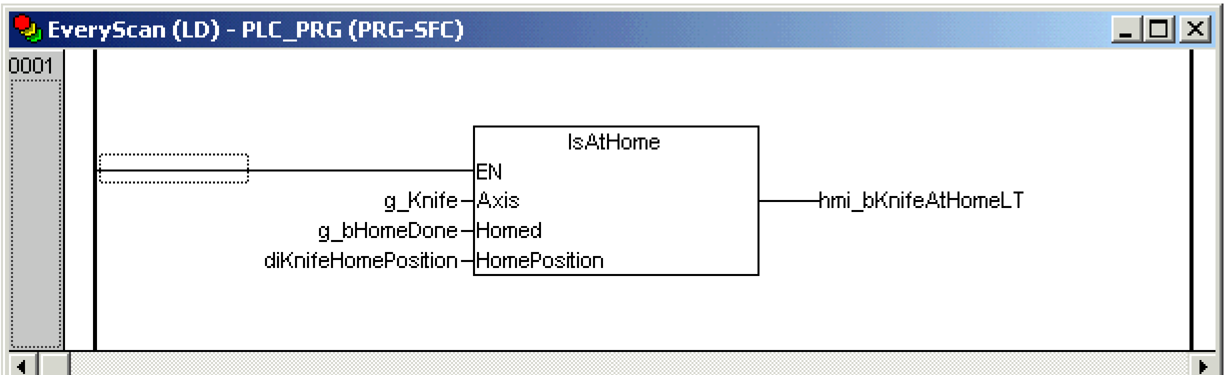


Figure 35, EveryScan Action

Hint

To add the IsAtHome function, first insert a "Box with EN". Then change the "AND" inside the box to "IsAtHome".

11. Auto State

In the Auto state, we will execute one revolution moves with a timed dwell between them. We will use a motion function block that allows us to cause the move to occur is a specific time rather than at desired speed. This is much easier than doing the calculations yourself to produce the correct speed and acceleration/deceleration rates for the move to happen in the desired time. As with the Manual and Homing states, Auto will require an Entry action to reset bExecute.

➢ Add the rungs shown in Figure 36 to the Auto State.

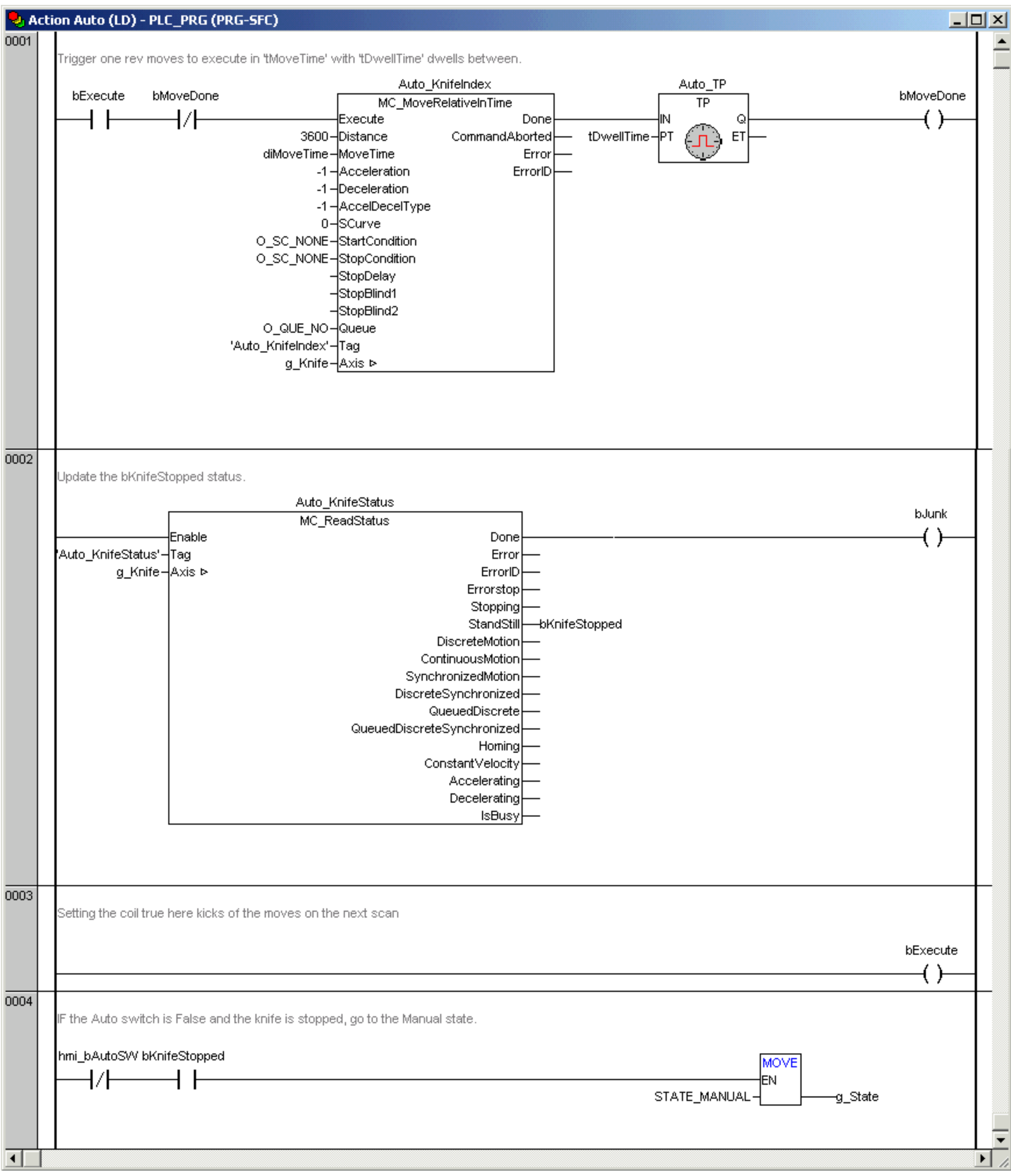


Figure 36, Auto State

11.1. Alternative Auto State

In the example shown in Figure 36 the exact duration of the dwell could be influenced by the scan time of your program. With a simple program like the one shown this is unlikely to be a problem. However with much larger programs and applications with very critical timing requirements this could become a problem. ORMEC's Motion Libraries provide a way to avoid this.

Moves and dwells can be placed in a queue that can be repeated indefinitely. By placing them in a queue, the timing of their execution is handled in the background by the Real Time Operating System and is not affected by your program.

In place of the program shown in Figure 36, you would use the one shown in Figure 37

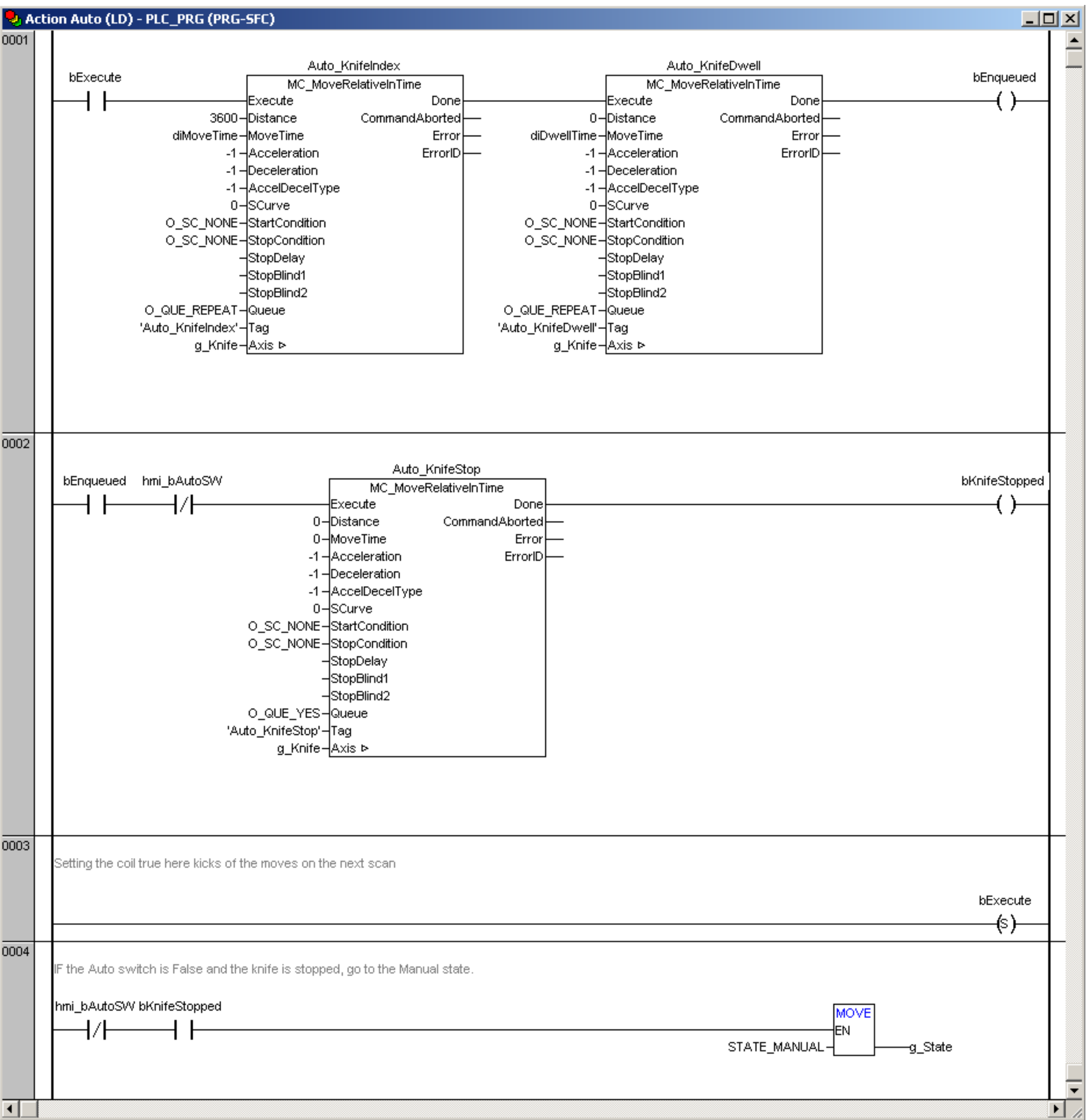


Figure 37, Alternative Auto State using Enqueued Moves

Once bExecute becomes true, the move and dwell will be enqueued and will execute repeatedly from the motion queue with no intervention from the program. All the Auto State will need to do is monitor the Auto switch and stop the motion queue if it is switched to manual.

bEnqueued will become true as soon as the index and dwell have been placed in the queue. If the Auto switch goes false after bEnqueued becomes true, a non-repeated queued move for zero distance in zero time will be added to the end of the existing motion queue. This will cause the Knife to complete whatever index and dwell it is working on and then terminate the queue.

Since the MC_MoveRelativeInTime function block takes its time input from a DINT variable rather than a TIME variable, you would have to change the declaration of tDwellTime from TIME to DINT and rename it diDwellTime. You would also have to change its initial value from t#1.0s to 1000.

12. Error and Fault Handling

Whenever a fault or error occurs, you will need to safely stop the machine. The easiest way to do this is with the ErrorCheck action.

- Right-click on PLC_PRG in the Object Organizer window.
- Select "Add Action".
- Name the action "ErrorCheck", and select LD for the language.
- Enter the code as shown in Figure 38.

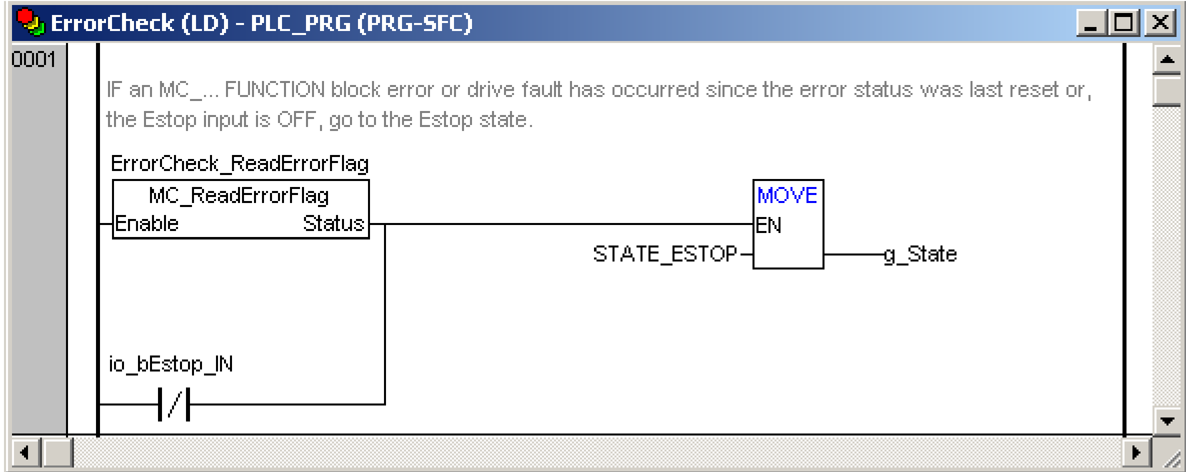


Figure 38, ErrorCheck Action

Anytime an MC… function block generates an error or a drive fault occurs, the SMLC's Error Flag is set true. In Figure 38, the Error Flag is read and if it is true, the state is set to Estop. Likewise, if the Estop input opens, the state is also set to Estop.

12.1. Error and Fault Annunciation

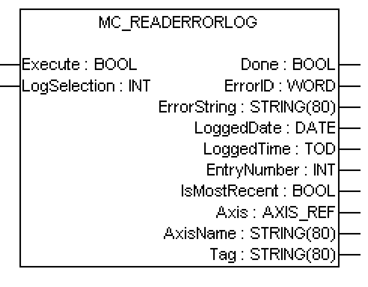
You always need to let the operator know if any kind of Error or Fault has occurred. If your application doesn't have any kind of HMI display, about all you can do is turn on a light whenever MC_ReadErrorStatus returns true. If you do have an HMI display, in addition to turning on light you can also tell the operator the type of Error or Fault that occurred.

The PLCopen specification for motion control requires every motion related function block have an Error output that is set if the function block fails. There is also an ErrorID output that gives the ID of the error that occurred. To detect drive faults, you can use the MC_ReadStatus function block to see if a drive is in the ErrorStop state. This would tell you the drive has experienced a fault. You could then use the MC_ReadAxisError function block to find out what type of fault.

Using the above approach requires you to use a different variable for each function block Error and ErrorID output. You then have to poll each of the Error outputs and if it is true, display the associated ErrorID. This takes a lot of programming and adds a great deal of complexity to your program. Fortunately, the SMLC provides an easier and more effective way of telling the operator what went wrong.

The SMLC maintains a date and time stamped history of every Error or Fault that happens.

The MC_ReadEventLog function block provides outputs you can send to a display unit. You set the Selection input to one of the following:



- The most recent Error/Fault.
- The next Error/Fault.
- The previous Error/Fault.
- A specific Error/Fault.
- Clear the Error/Fault history.

You can add a new task that runs concurrently with your main program, PLC_PRG, to handle the error log. This will allow the operator to browse the recent errors and faults regardless of the machine state. Inputs from the HMI can be used drive the Selection input so the operator can select which error he wants to review. The outputs would be connected to string variables that are displayed on the HMI.

12.2. Creating The HMI Task

First we need to create a new program to handle the HMI task. Right-click anywhere in the Object Organizer window and select AddObject. Configure the new POU as shown in Figure 39.

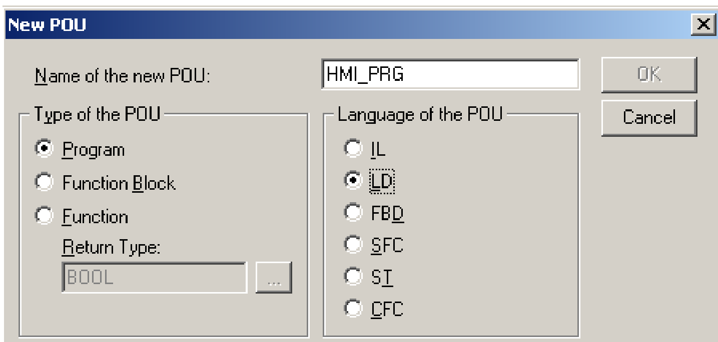


Figure 39, HMI POU Configuration

Up to this point, we've only had one task running in our program, PLC_PRG. The system knows which one to run because there is only one. When we have more than one task we need to explicitly tell the system which programs to run.

Task Types and Priorities

There are three main types of task, Freewheeling (runs as often as it can), Cyclic (runs every N ms) and Event (runs after a specific event occurs). It is best to avoid freewheeling tasks since they can have unexpected effects on task execution. Since Event tasks are beyond the scope of this Tutorial, we will discuss only Cyclic tasks.

Task Intervals

The Task Interval setting determines how much time elapses between when a task starts to execute and when it starts to execute again. As such it must be set longer than the time the task takes to execute. If the Interval is set to less than the task execution time, depending on its priority, it can lock out other tasks.

Task Priorities

Tasks with a lower priority are given preference over tasks with a higher number. While the available range is 1 to 31, there are in fact only 10 priority levels. This means 0, 1 and 2 have the same (highest) priority 3,4 and 5 have the next priority and so on.

While this may seem counter-intuitive, when you have two tasks, one of which needs to run more frequently than the other, you should give the lower frequency task greater priority (lower priority number) than the more frequent task. For example, the main PLC_PRG task that runs the machine might be set up with a priority number of 10 with a time interval of 5ms. An HMI task that is less critical should have a priority of 1 with an interval of 10 ms. This will cause the HMI task to run once while the PLC_PRG task runs twice. The higher priority on the HMI task guarantees that the HMI task will get the opportunity to run and will not be preempted by PLC_PRG should it take longer than 5ms to execute. However, you must also make sure you don't add any programming to HMI_PRG that will unduly delay or prevent execution of PLC_PRG.

Task Watchdog

Each task has a watchdog that will stop the task if its scan time exceeds its watchdog time. Since having a single task stop due to a watchdog fault could have all kinds of unforeseen consequences, we recommend you do not activate the task watchdogs.

- Click on the Resources icon in the Object Organizer Tabs.
- Double-click on Task Configuration. This will bring up the Task Configuration window.
- Right-click on Task Configuration in the left-hand window and select Append Task, this will create a new task and bring up the Task Attributes window.
- In the right-hand window, Type PLC_PRG for the name and set the type to Cyclic as shown in Figure 40.
- Set the priority to 10 and the time interval to T#5ms.
- Right-click in the left-hand pane on the icon to the left of PLC_PRG and select Append Program Call.
- In the Program Call window, click on … and select PLC_PRG().

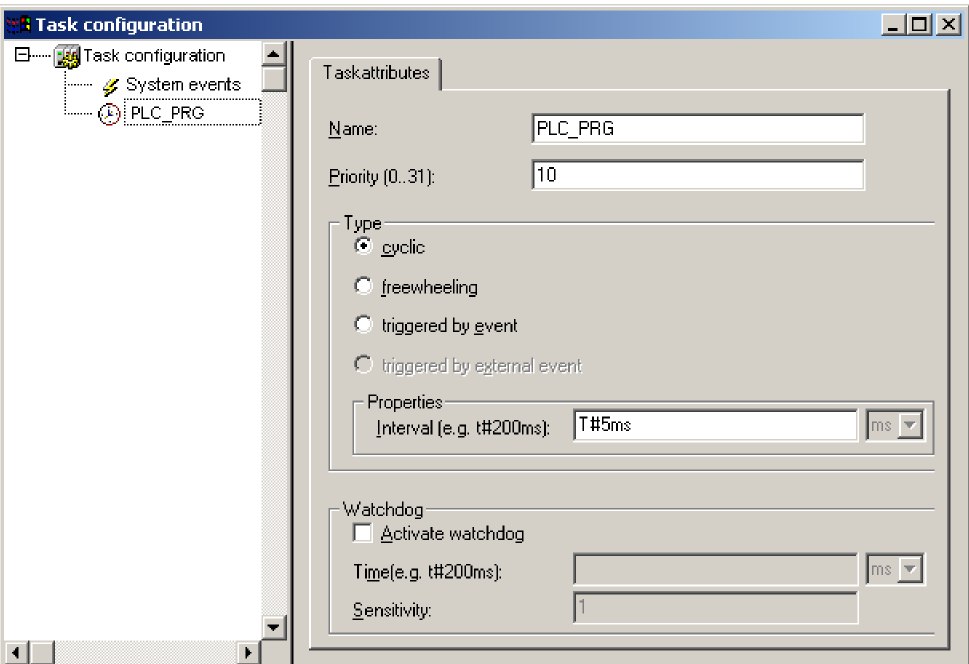


Figure 40, Task Configuration – PLC_PRG

Now we'll create a second task to run the HMI:

- Right-click on Task Configuration in the left-hand window and select Append Task again.
- This time set the new task name to HMI_PRG and set the type to Cyclic.
- Set the priority to 1 and the Interval to T#50ms
- Append a program call to HMI_PRG.
- The Task Attributes window will change to look like Figure 41.

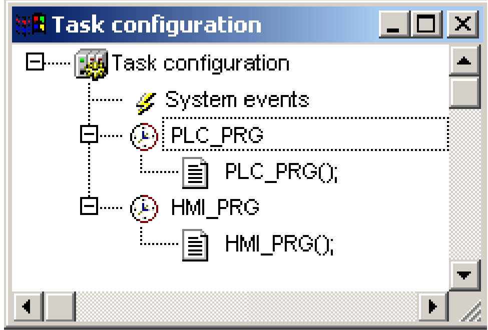


Figure 41, Task Configuration

Priorities

It may seem counter-intuitive to give the less important HMI task a higher priority than PLC_PRG.

If PLC_PRG had a higher priority than HMI_PRG and it were to take longer than 5ms, HMI_PRG would never have the opportunity to run. The higher priority of HMI_PRG guarantees it gets the opportunity to run even if PLC_PRG overruns its interval time.

The result of these two task configurations is that PLC_PRG will run scan every 5ms. HMI_PRG will run one scan every 50ms.

Now we must write the code for HMI_PRG.

- In the Object Organizer window, double-click on HMI_PRG to open the Ladder Diagram editor.
- Now enter the Ladder code shown in Figure 42.

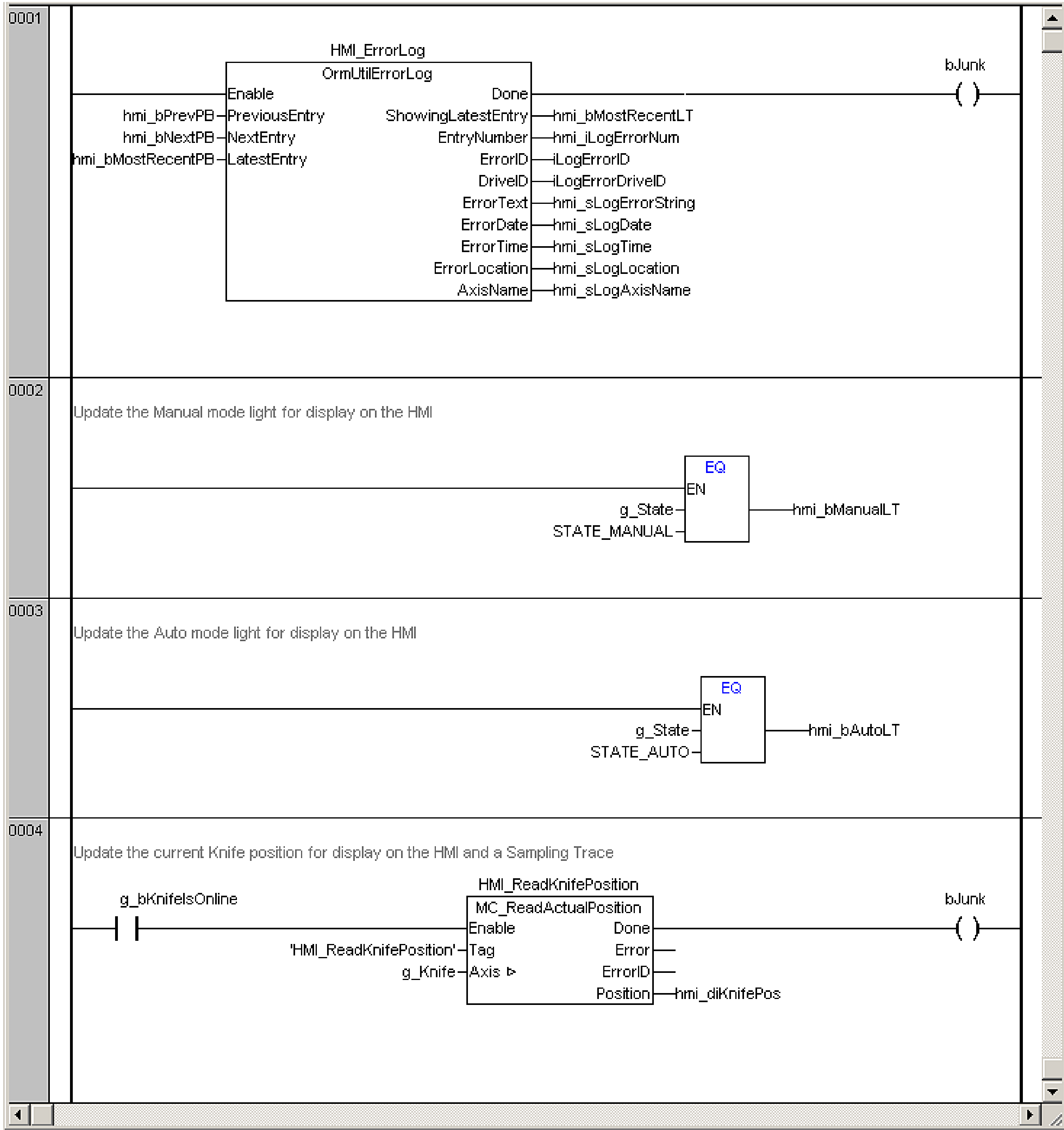


Figure 42, HMI Task

- Rung 1 checks the error log display push-buttons retrieves the text to display on the HMI error log screen.
- Rungs 2 and 3 set the Auto and Manual status indicators on the HMI.
- Rung 4 reads the Knife position, which is displayed on the HMI and used by the sampling trace.

These variables are displayed on one of two the two Visualization screens shown in Figure 43 and Figure 44.

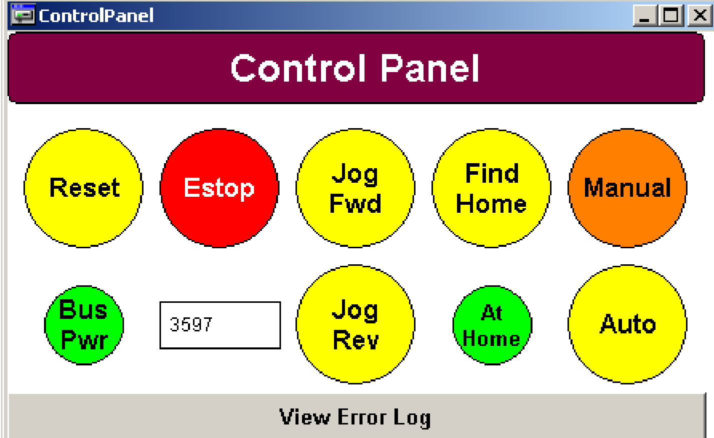


Figure 43, Control Panel Visualization

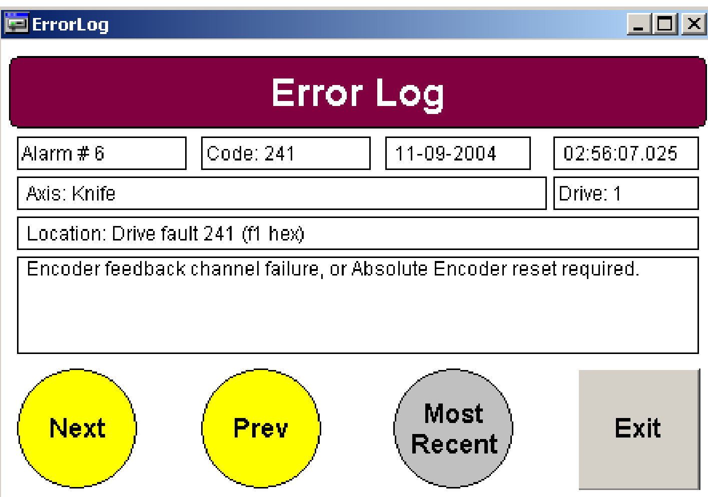


Figure 44, Error Log Visualization

To configure the elements of the visualizations, right-click on any element and select “Configure”.

Text Tab: This tab allows you to define text that will appear in the element.

Colors: Allows you to select the background color for the element, The Alarm Color sets the color when the “Change Color” variable associated with the element is true.

Variables: Is where you set the Change Color variable.

Inputs: Is where you can assign variables that will be changed when you click on the element. Choices include Toggle, Tap (momentary on), Tap False (momentary off), Zoom to another visualization, Execute a program and enter a value.

13. I/O Configuration

While the program described in this tutorial does not use any physical I/O, most real applications will. This section describes how to configure your SMLC to use physical I/O and how to add names to the individual I/O points.

This tutorial describes the way you configure Ethernet I/O. You can configure FireWire (IEEE-1394) and Profibus I/O in a very similar manner.

- Click on the Resources icon in the Object Organizer Tabs, and then select PLC Configuration.

This is what you should see.

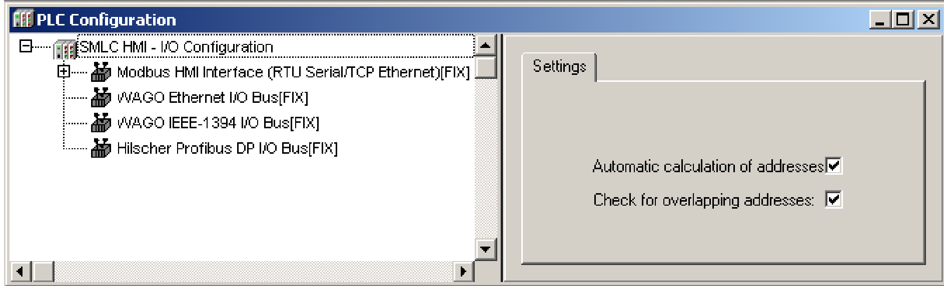


Figure 45, PLC Configuration

If you don't see the Ethernet and IEEE-1394 items, try clicking on the box to the left of the SMLC I/O Configuration. If that doesn't work, it may be that you have inadvertently selected a different PLC Configuration. To correct this, open the PLC configuration window and then select the Standard Configuration from the "Extras" menu.

First we will add a register to allow a remote HMI to display the Knife Position.

- Right-Click on Modbus HMI Interface and select "Append Subelement - add 20 DINT variable".
- Expand the resulting block of registers by clicking on the "+" sign to the left of them and then click just to the left of the word "AT" on the first register.
- Then type hmi_diKnifePos.
- Collapse, the DINT registers by clicking on the "-" sign and then add 40 INT registers which we will use to hold the push-buttons, indicator lights and integer values.
- Expand the block of 40 integers and add the name "hmi_PBs_SWs" to the first integer.
- Expand the first integer, and add the following names:

  hmi_bResetPB
  hmi_bRestPB
  hmi_bJogFwdPB
  hmi_bJogRevPB
  hmi_bHomePB
  hmi_bAutoSW
  hmi_bNextPB
  hmi_bPrevPB
  hmi_bMostRecentPB

When you are done, it should look something like this:

- In a similar manner, name the second integer register "hmi_LTs" and add the following variables:

  hmi_bBusPwrLT
  hmi_bAutoLT
  hmi_bManualLT
  hmi_bKnifeAtHomeLT
  hmi_bMostRecentLT

- Next we need to use full integer registers (as opposed to bits within registers) to hold the following variables:

  hmi_iLogErrorNum
  hmi_iLogErrorDriveID
  hmi_iLogErrorID

The result should look like Figure 47 on the next page.

- Finally, we need to add 5 String registers to hold the following variables:

  hmi_sLogErrorString
  hmi_sLogAxisName
  hmi_sLogLogTime
  hmi_sLogLocation
  hmi_sLogDate

The result should look like Figure 46 below.

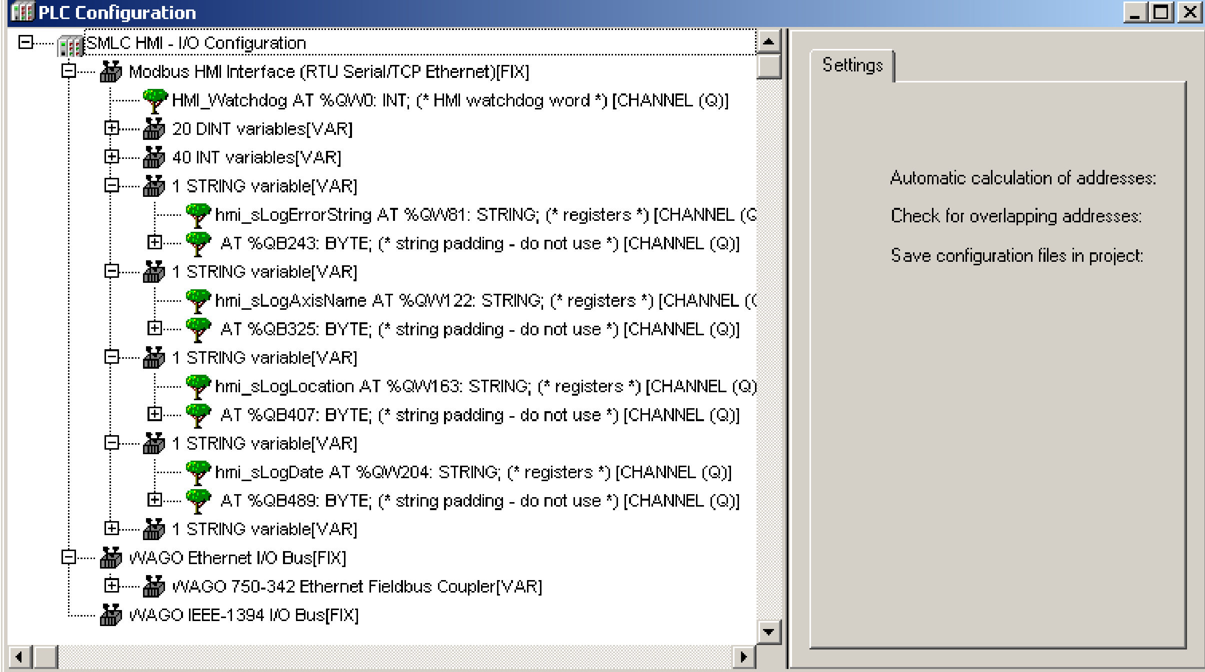


Figure 46, HMI String Registers

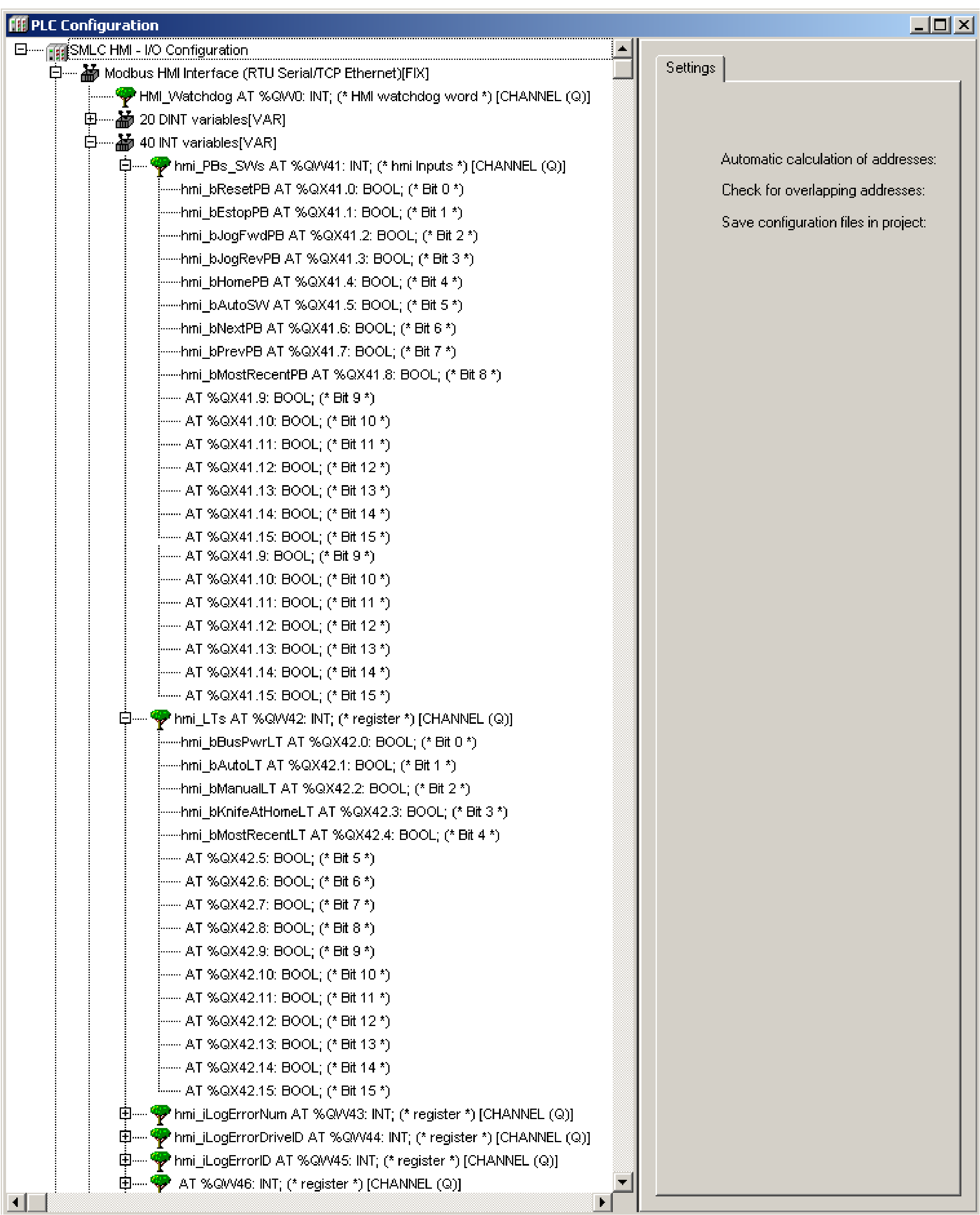
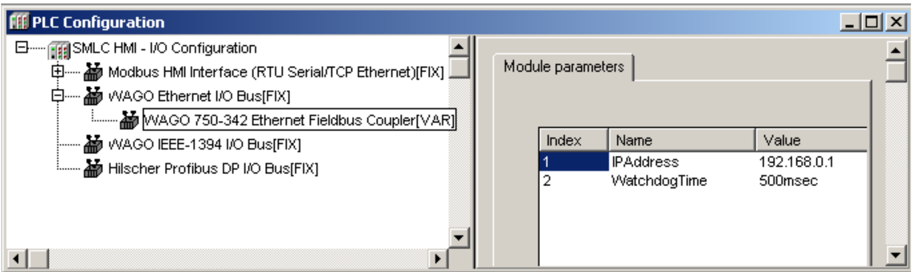


Figure 47, HMI Integer Registers

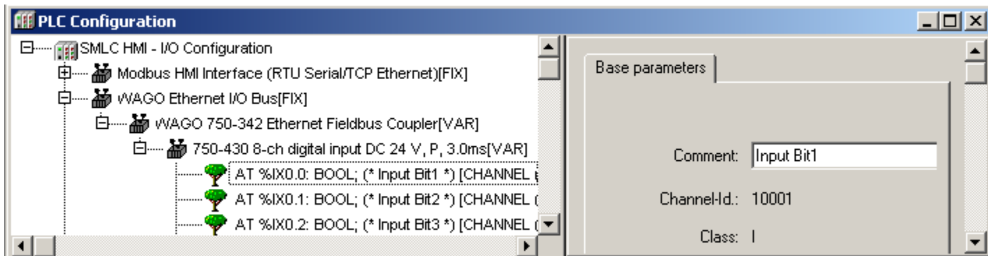
> Assigning registers to the variables automatically declares them as global variables. Since we had previously declared these variables, we must remove the original declarations to avoid duplicate declarations. The result of not doing this would be the originally declared variables would be local to your program and the new ones declared in the PLC Configuration, with the same names, would be global. This would result in untold confusion!
>
> If the original variables were declared as global variables failure to remove them will result in an error message when you compile "Several declarations with the same identifier".

- Right-click on the "Ethernet I/O Bus[FIX]" item and select "Append Wago 750-342 Ethernet Fieldbus Coupler".
- Left-click on the resulting "Wago 750-342 Ethernet Fieldbus Coupler [VAR]" item.



In the right-hand pane you should now see a table in which you can set the IP address of your Ethernet FieldBus Coupler and the I/O watchdog time

- Right-click on the "Wago 750-342 Ethernet Fieldbus Coupler [VAR]" item and move the cursor over the "Append Subelement" item. Hold it their for a few seconds and a long list of available I/O modules will appear. Select "750-430 8 ch DC input…" or whatever input module you have installed.



You may then select each individual I/O point and type a short comment describing what it is for. If you left-click over the "AT", a box will open up into which you may type a variable name which will become associated with that I/O point. This variable is automatically declared as a global variable. If you use the name of a previously defined variable, you must delete the original declaration otherwise you will get an "Error 3703: Error PLC Configuration (IX0.0): Several declarations with the same identifier" compilation error. The indicated address (IX0.0) corresponds to the I/O point address in the PLC Configuration and will allow you to figure out which variable is causing the problem.

13.1. Associated I/O Task

The I/O automatically becomes associated with the program task that has the shortest Interval setting. If there is more than one with the same setting, the I/O will be associated with the one that appears higher in the list. Whenever the associated task executes, the physical inputs will be read into memory, the logic solved and then the physical outputs set based on the memory image. All other tasks will use the same memory image that existed when the I/O associated task completes.

13.2. I/O Cycle Time

With Ethernet I/O, reading and writing the physical I/O adds about 3ms per FieldBus coupler to the associated tasks execution time. This should be taken into account when setting task Intervals. IEEE-1394 I/O has a cycle time of about 2.5 ms and Profibus I/O 0.2ms at 12 mbaud.

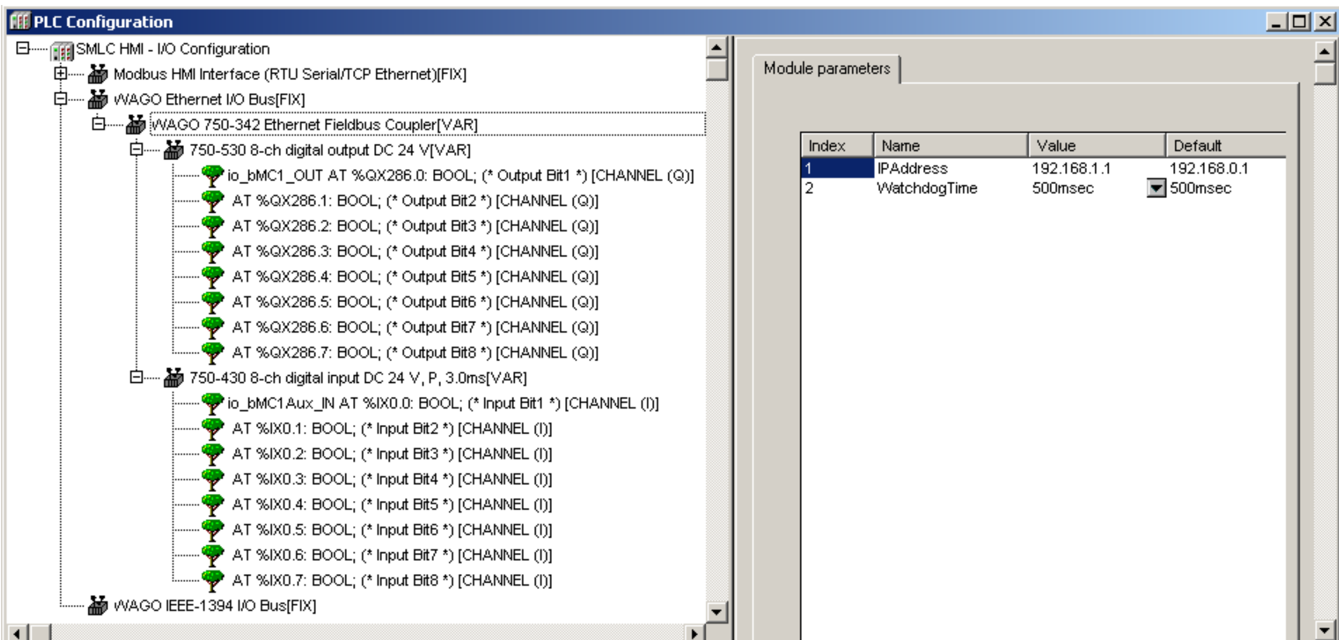


Figure 48, Typical I/O Configuration

14. Appendix A – Variable Prefixes

Prefix	Type		Lower limit	Upper limit	Memory space
b	BOOL	Boolean	0	1	8 Bit
by	BYTE	Byte	0	255	8 Bit
w	WORD	Word	0	65535	16 Bit
dw	DWORD	Double word	0	4294967295	32 Bit
si	SINT:	Short integer	-128	127	8 Bit
usi	USINT:	Unsigned short integer	0	255	8 Bit
i	INT:	Integer	-32768	32767	16 Bit
ui	UINT:	Unsigned integer	0	65535	16 Bit
di	DINT:	Double integer	-2147483648	2147483647	32 Bit
udi	UDINT:	Unsigned double integer	0	4294967295	32 Bit
r	REAL	Real (float)	$\pm 3.4 \times 10^{-38}$	$\pm 3.4 \times 10^{38}$	32 Bit
lr	LREAL	Long real (float)	$\pm 1.07 \times 10^{-308}$	$\pm 1.07 \times 10^{308}$	64 Bit
t	TIME	Time			
d	DATE	Date			
tod	TIME_OF_DAY	Time of day			
dt	DATE_AND_TIME	Date and time			
s	STRING	String			

- Global variables should have the prefix g_
  g_bAuto
- Variables declared in the MODBUS register section of the PLC Configuration, should begin hmi_
  hmi_bJogPB
- Where appropriate, HMI variables should have a suffix:
  PB – Push Buttons
  SW – Switches
  LT – Indicator Lights
- Variables declared in the I/O section of the PLC Configuration, should be prefixed, io_ have a type prefix and a suffix indicating whether they are inputs or outputs.
  io_bMC1Aux_IN for Boolean I/O or io_iSpeed_OUT for analog I/O.
- Constants should be in upper case:
  TWO_PI
  g_PI_BY_TWO
- Names for SFC steps should begin with an underscore _
- Names for SFC transitions should begin with the prefix trn
- Names used in Type declarations for enumerated variables should begin with the prefix e