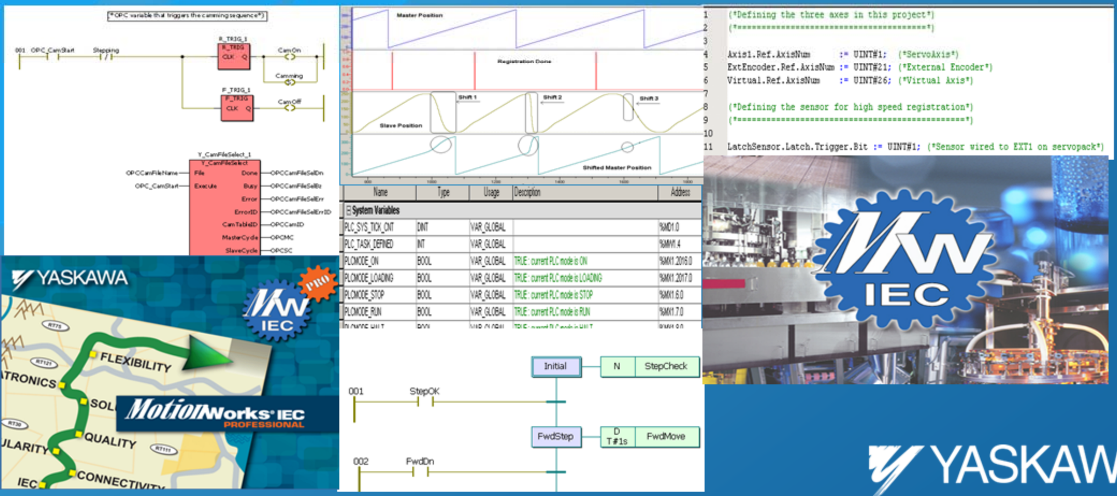
Product Application Note

Driving Value

MPiec Programming Best Practices Guideline



Applicable Product: MPiec Controllers with MotionWorks IEC

Rev 1.1

1 Application Overview

This document is a guide to implement best programming practices during the design and architecture of a MotionWorksIEC project for the MPiec series controller. The main topics discussed in this document are an overview of the design process, program architecture, and some good coding practices.

1.1 Why Use a Yaskawa Machine Controller?

General automation machine controllers are beneficial in that they offer a unique flexibility for programming the operation of functions and sequences within a machine. Whether the application warrants high performance point-to-point indexing, or complex motion profiling with on-the-fly phase shifting – all benefit from tightly coupled motion and machine sequence logic control. As factory floor production requirements increase, machine cells must also increase their own production capability, and precise control of complex motion becomes vitally important to insure production rates are met and continue high quality product yield. Many independent PLCs and Motion Controllers reach a limitation when programmers strive to achieve these new and improved line production performance benchmarks, and unfortunately result in a bottleneck when trying to increase the performance of a system. This bottleneck can often be caused by control systems latencies in data handshaking or lack of synchronization, not only from cell controller to cell controller, but also within a single controller's functional module interfacing. A good machine control platform tightly couples high performance motion control capabilities with machine sequence logic, and provides the foundation and scalability necessary to achieve performance in a changing environment.

True machine control tightly integrates the following: high and low performance features, priority setting and adjustment, determinism and predictability, synchronization of tasks, scalability without sacrificing performance, integration of peripheral components, and a focus on precise motion control. The result is smooth motion, less machine jerk, less mechanical vibration, less heating, smaller and more efficient motor sizes, higher machine reliability and longer life cycles – all with the advantage of built in future expansion.

To take advantage of the power of total machine synchronization and extend this capability across a wide range of performance requirements - from low performance, low cost machines to high performance, higher priced machines, a scalable machine control platform is offered by Yaskawa: the MPiec Series Machine Controller. To harness the power of flexibility and performance while maintaining an overall ease of use, Yaskawa introduces the MPiec Programming Best Practices Guideline.

1.2 What is in MPiec Programming Best Practices Guideline?

This document is a guideline to lead the controls engineer down a successful path that will not only increase programming productivity for any given machine, but also code reusability for an entire line of complementary machines which can be offered for different user requirements. This document helps the programmer to realize the benefits of following best practices:

1.3 Benefits of Using MPiec Best Practices:

Benefits can be realized at all levels of machine development from design to implementation to field support. In summary, all benefits work towards lowering the total cost of ownership in the machine's product lifecycle.

- *Design Level Benefits*

- *Programming Method Selection:* A flexible machine control allows several ways to program motion control. This guideline helps the programmer select the best programming method for a given application type within IEC 61131-3 standard programming environment.
- *Standardized Model:* This guideline helps the programmer develop a robust framework and controls architecture, and secures a solid platform for code development. In addition, within the guideline are several examples of standardized programming methodologies that leverage template examples.
- *Step-By-Step Basic Design:* There is a higher probability of getting it right the first time if a step-by-step approach is followed, reducing the possibility of missing important steps along the way. This guideline provides a step-by-step flowchart of machine development features.
- *Risk Reduction*: The probability of completing development projects on time, and on budget will be high through the use of MPiec Programming Best Practices.

- *Implementation Level Benefits*

- *Pre-Defined Code & Scalability:* This guideline provides pre-defined code and canned functionality recommendations, which in turn reduce development and debugging time. More efficient code development can be realized which affords more time to be spent on process related issues.
- *Optimized Performance & Robustness:* Code efficiency is dependant upon good implementation of well thought out architecture. If the code is written right, performance of the control system can be optimized resulting in lower scan times, increased motion performance, and increased performance of device interfacing. This guideline encourages optimized performance in implementation resulting in robust code that works.
- *Organization:* This guideline makes programming easier because features are compartmentalized for streamlined implementation. An organization layout is provided for variable/tag usage, alarm interlocking, program flow, axis control management, I/O usage, etc.

- *Support Level Benefits*

- *Comprehensive Training*: Using Best Practices supplements Yaskawa's product training, allowing the skills learned to become better utilized.
- *Transferable Skills*: Using best practices across multiple machine designs reduces risk by taking advantage of architecture knowledge acquired on previous designs.
- *Common Look and Feel*: Using commonly defined, standardized methods will make the system easier to support when the machine is in production. Tech support engineers can leverage the same best practices. Maintenance and troubleshooting become easier on a global basis. MPiec Best Practices is not meant to replace open standards, but rather to compliment standards suggested by PLCopen and IEC 61131-3.

2 Getting started – Using the best practices guideline

This Best Practices Guideline is designed to help the controls engineer better utilize the strengths of Yaskawa technology, allowing them to leverage the unique flexibility of functionality that the MPiec platform offers, and best align the implementation with the specifics of the application requirements. Understanding the control system's benefits and constraints early in the design process will result in a higher probability of successful machine design. This guideline will serve as a machine controller design and development aid, presenting program architecture, programming recommendations, as well as actual code examples and pre-defined templates to get started. It also provides experienced controls engineers a benchmark to compare against, allowing them to make more informed decisions between known practical solutions, and new, unique ideas for a better method.

First time users will find this MPiec Best Practices guideline a useful tool to guide them through a recommended step-by-step procedure for controls development on for an automated machine with tightly integrated motion control. More experienced users will find this MPiec Best Practices guideline as a useful reference tool for specific techniques and code modules that can be readily used. This guideline is formatted to provide the specific techniques, with a Step-by-Step framework.

The following diagram illustrates the recommended step-by-step procedure when developing a performance machine control system.

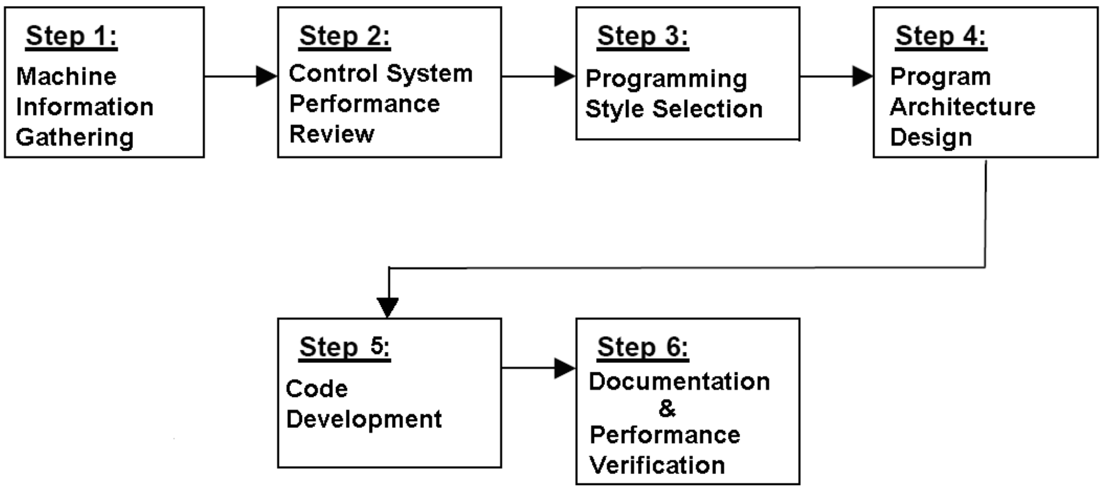

Step 1:
Machine Information Gathering
Step 2:
Control System Performance Review
Step 3:
Programming Style Selection
Step 4:
Program Architecture Design
Step 5:
Code Development
Step 6:
Documentation & Performance Verification

Figure 1: Steps for development of machine control code

3 Step 1 – Machine Information Gathering

In order to implement a structured, reliable and efficient machine control system, the machine requirements should be gathered thoroughly in a formal manner to ensure that details of the design and functionality are not overlooked. The sequences of steps in machine requirement gathering are shown in Figure 2.

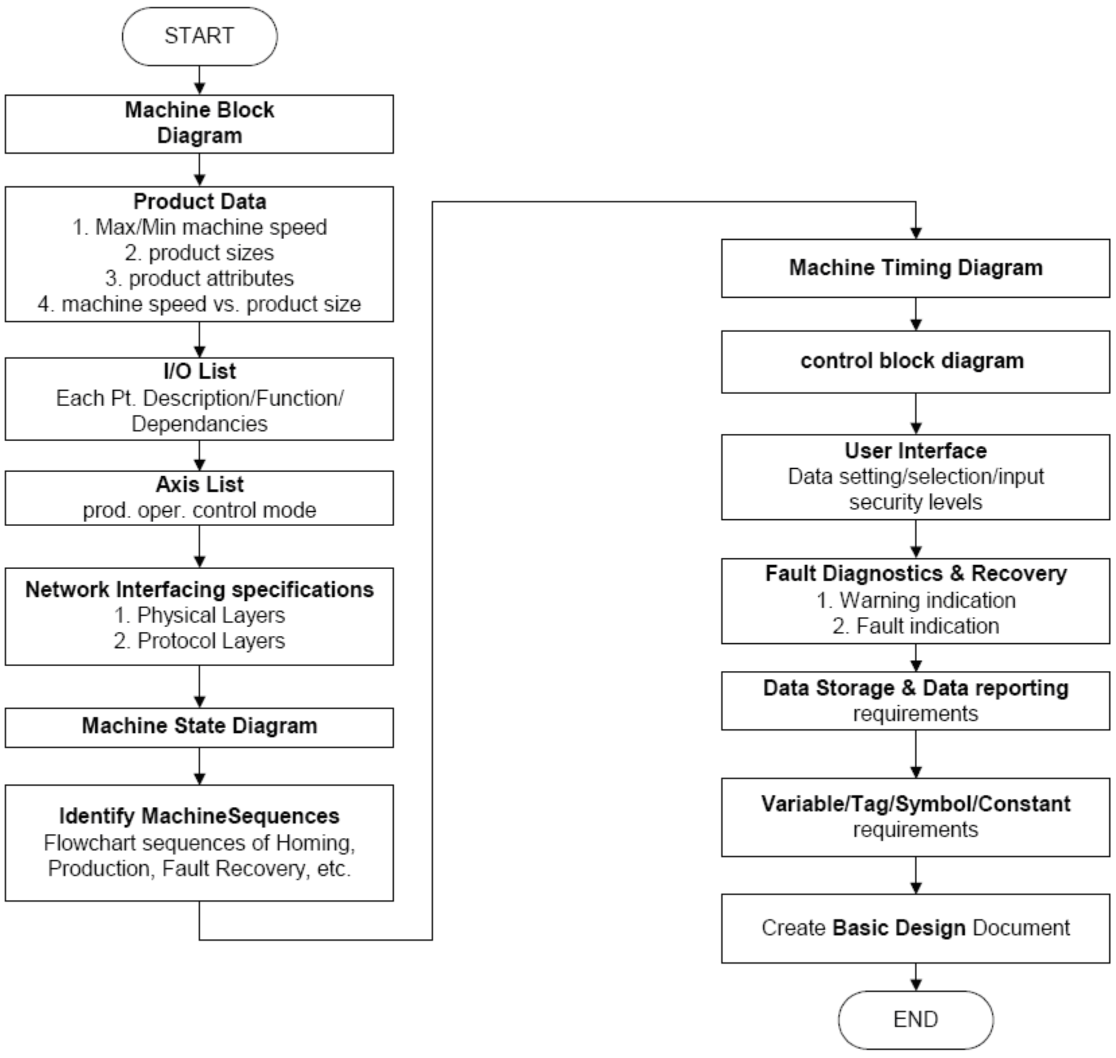

START
Machine Block Diagram
Product Data
1. Max/Min machine speed
2. product sizes
3. product attributes
4. machine speed vs. product size
I/O List
Each Pt. Description/Function/ Dependancies
Axis List
prod. oper. control mode
Network Interfacing specifications
1. Physical Layers
2. Protocol Layers
Machine State Diagram
Identify MachineSequences
Flowchart sequences of Homing, Production, Fault Recovery, etc.
Machine Timing Diagram
control block diagram
User Interface
Data setting/selection/input security levels
Fault Diagnostics & Recovery
1. Warning indication
2. Fault indication
Data Storage & Data reporting requirements
Variable/Tag/Symbol/Constant requirements
Create Basic Design Document
END

Figure 2: Basic Design process

Gathering the above mentioned data thoroughly helps in many ways. A few advantages are:

1) Easy to design machine control algorithm and timing
2) Code can be optimized
3) Easy to startup and validate as Key Performance Indicators (KPI) are clearly defined
4) Easy to understand machine operation and control at a later stage

A worksheet that can be used to start gathering machine requirements is available at:
http://partner.yaskawa.com/site/dmcontrol.nsf/(DocID)/BFAR-6SSUHB?opendocument&login
(You must be logged in to view this document, as it is secure. If you are not logged in and you click the link, you will be asked to login first. Once logged in the file will appear.)

A basic design document serves as a blueprint for all things related to controlling the machine. The design document contains topics like IO timing diagrams, conceptual machine sketch, control diagrams, move timings, BOM, HMI interface variables, machine cycle sequences etc. A template of the basic design document is available at:
http://partner.yaskawa.com/site/dmcontrol.nsf/(DocID)/AHAR-6BGQJ7?opendocument&login
(You must be logged in to view this document, as it is secure. If you are not logged in and you click the link, you will be asked to login first. Once logged in the file will appear.)

4 Step 2 – Control System Performance Review

Once the requirements and functionality of the machine are documented, the controls programmer can evaluate the optimum methods for implementing machine and motion control. The machine block diagram, sequences, timing diagram and motion kernel control block diagram will be key in evaluating the programming methods and application scan intervals to use in the application project.

4.1 Key questions to address before starting program design and development:

1. What is the rate of production you wish to achieve on this machine?
2. Given the rate of production, what is the cycle time?
3. What are the inputs and outputs that are part of a cycle?
4. What rate should the inputs and outputs achieve to keep up with production rate?
5. Should the IO be tied to the MECHATROLINK network, or on the controller back plane or on Ethernet with one of the built in communication protocols (OPC, MODBUS TCP, EtherNet/IP)
6. What update rates are required for communication from a PC or HMI?
7. If motion demands synchronous motion like camming, what rate of controller interpolation is required? (MECHATROLINK/DPRAM update)
8. Does motion or I/O demand higher priority? Do IO and motion have the same priority?
9. Is MotionWorksIEC Pro or MotionWorksIEC Express required for this application? MotionWorks IEC Pro allows the user to assign different scan intervals and priorities for up to 16 individual tasks. With multiple tasks fast updates can be achieved for critical tasks by assigning slow update rates for non critical tasks. This is not achievable through MotionWorksIEC Express which runs with only one task.

4.2 Timing chart

Once the rate and update intervals are known, the various I/O devices and motion functionality can be assigned to tasks. Task intervals and individual task priorities can be set. A sample timing chart showing time allocation of CPU resources for an application with three tasks (fast, medium and slow) is shown in Figure 3a.

Execution sequence and Timing Facts:

1. MECHATROLINK commands get highest priority (user does not have to do anything to set this)
2. User should assign the task requiring the shortest time interval the highest priority (0).
3. If the highest priority task uses almost all of the time allotted to it, there will be less time for lower priority tasks to execute. If the lower priority tasks do not have enough execution time, PLC watchdog time outs will occur.

4. Inputs are read at the beginning of the task they are associated with and outputs are written when the task they are assigned to finishes execution. This means that the outputs can get written in unequal intervals depending on how long it took the particular task to execute. For network I/O (MECHATROLINK, MODBUS TCP, Ethernet/IP), the outputs will be updated on the next data exchange, so those outputs will be synchronous with the network scan.
5. Default or background tasks may or may not run depending on whether other tasks complete executing within their allotted times. Default/Background tasks execute only after all other tasks complete executing. They run in the time interval between the completion of the slowest task and the next interrupt of the highest priority task. This can be seen from Figure 3a.

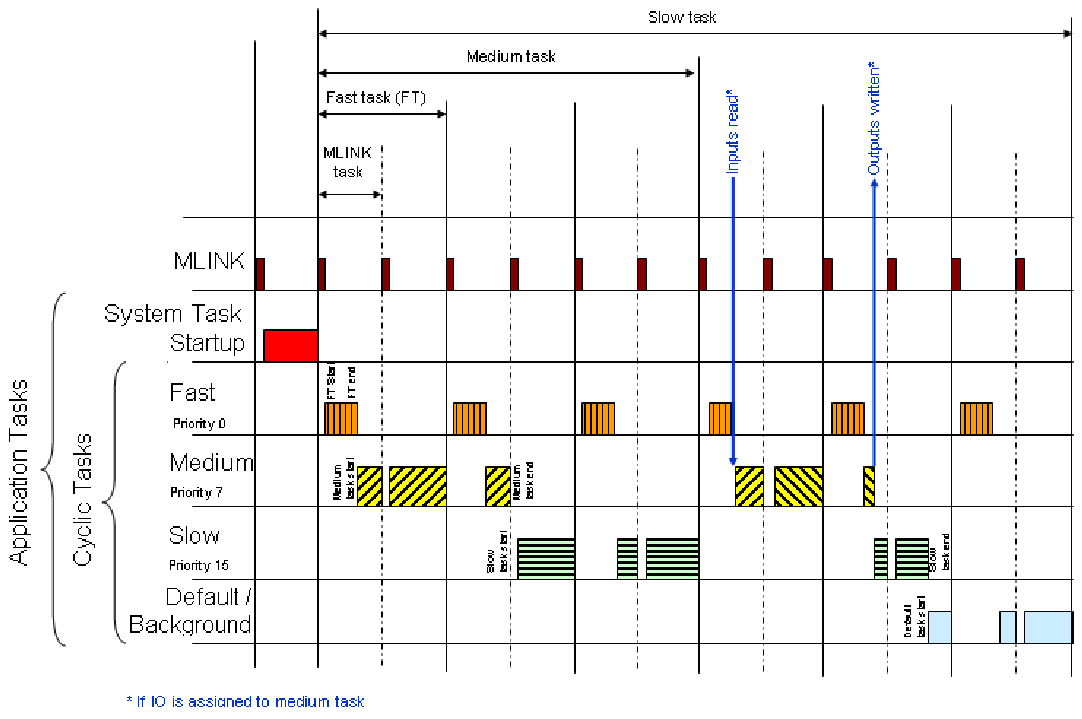


Figure 3a: Task sequence and order of operations

It is important to assign I/O drivers to the appropriate task in order to ensure that the I/O signals are updated in the application program at the desired rates. Along with the IO drivers being associated to a task, the Program Organization Unit (POU) with logic that uses the IO should also be associated with the same task such that appropriate updates are seen at the application level. The most common devices that require task assignment for drivers are:

1. Controller I/O
2. Servo I/O
3. Communication I/O (MODBUS and EtherNet/IP)

IO driver updates can be tied to a particular task in the hardware configuration tool. There is a difference between poll period (set when the MPiec controller is a client or scanner) and task update. Poll period is the rate at which the client or scanner requests data from a server or adapter. The task update is the rate at which data gets refreshed in the application level (in the project).

4.3 Sample Application: Planning tasks intervals and priorities

Consider the rotary knife application detailed in Figure 3b. The hardware consists of an external encoder (master axis), a servo axis coupled to the tool, an LIO_01 card, a high speed sensor, and an EtherNet/IP bar code reader. The part arrives along the conveyor and gets scanned by the reader. The bar code is read and data is passed over to the MPiec controller over EtherNet/IP. The bar code reader is the adapter in this protocol. 50 ms after the part passes the bar code reader, the part is detected by the latch sensor which is used to latch the position of the conveyor with the encoder position read in through the LIO card. Once a part is detected, the controller has 50 ms to get the servo ready to synchronize with the master axis such that the tool makes contact with the leading edge of the part. Once the cam cycle for a part is complete, a digital output is fired based on the bar code reader. It is important that the digital output be fired within 10 ms of the cam cycle completion for the product. The gap between successive products is 50 ms with a tolerance of 10 ms.

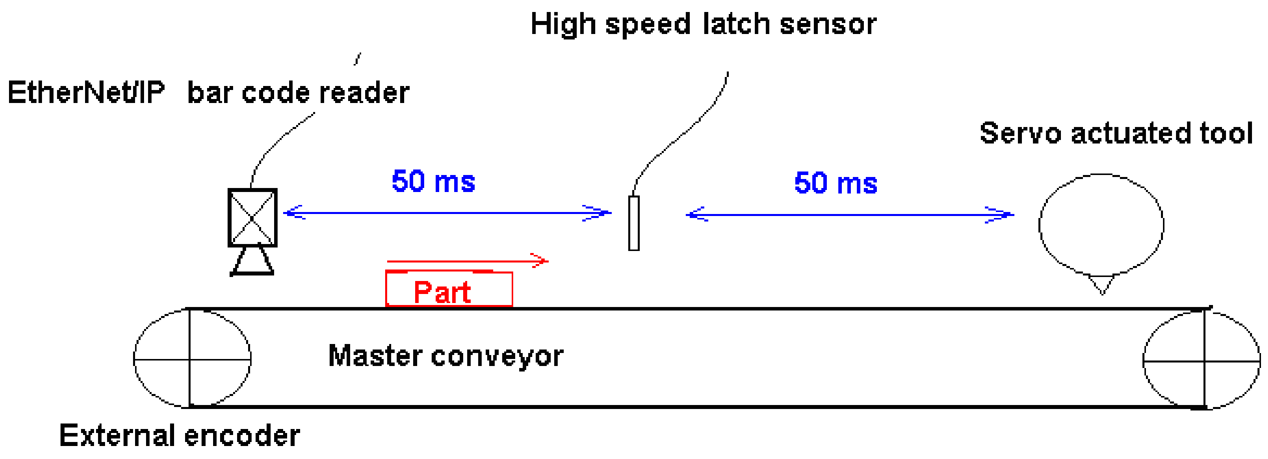


Figure 3b: Sample rotary knife application schematic

Sample rotary knife application planning:

The first step of control system performance review is to answer the 9 questions from subsection 4.1.

1. What is the rate of production you wish to achieve on this machine?
   1800 ppm
2. Given the rate of production, what is the cycle time?
   50 ms
3. What are the inputs and outputs that are part of a cycle?
   INPUTS: EtherNet/IP data, encoder master position tied into LIO card.
   OUTPUTS: Digital outputs
4. What rate should the inputs and outputs achieve to keep up with production rate?
   Specification says that 'the digital output be fired within 10 ms of the cam cycle completion'. Associate the Digital IO to a task updating at 4 ms such that outputs are fired as soon as the cam cycle is done. This will ensure that the outputs go out within 10 ms of the cam profile. Since new bar code data comes every 50 ms, the EtherNet/IP driver needs to update data at 20 ms such that new data is ready in the controller for logic to work on. The poll period can be set to 20 ms as well.
5. Should the IO be tied to the MECHATROLINK network, or on the controller back plane or on Ethernet with one of the built in communication protocols (OPC, MODBUS TCP, EtherNet/IP)
   Since no special requests were made regarding the location (remote or local) and number of digital IO, IO on the controller option slot with the LIO card is assumed to be sufficient. The outputs can be updated at faster than 10 ms on the LIO card.
6. What update rates are required for communication updates from a PC or HMI?
   Since new bar code data comes every 50 ms, a 20 ms poll period for EtherNet/IP will be more than sufficient.
7. If motion demands synchronous motion like camming, what level of controller interpolation is desired? (MECHATROLINK/DPRAM update)
   Not specified in application spec. 2 ms default MLINK update will be used
8. Does motion or I/O require higher priority? Do IO and motion have the same priority?
   Both have same priority
9. Should MotionWorksIEC Pro be used as opposed to Express?
   MotionWorks IEC Pro is required in this application since the impetus is on for fast motion and fast IO. With MotionWorks IEC Pro, task intervals can be adjusted individually. Non critical tasks can be assigned slow update rates. This allows critical tasks that include motion and IO to be updated at a higher rate without taxing the controller more.

Some of the main motion functions required in this sample application are:
1) MC_TouchProbe
2) Y_CamShift
3) Y_CamIn
4) Calculation to decide amount of shift for Y_CamShift

I/O components critical for the application are:
1) Digital IO
2) EtherNet/IP data

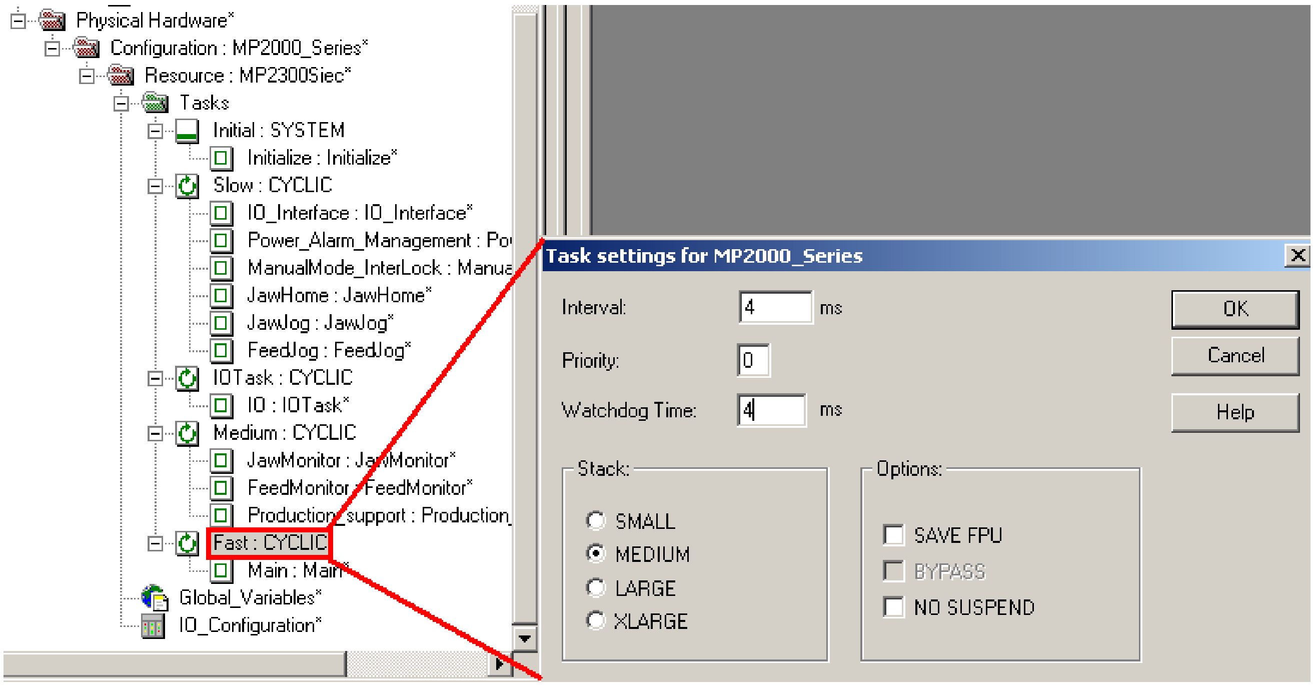
The various functions can be associated to tasks as shown in the table below

Table 1: Application scan timing for sample application

Task	Interval (ms)*	Priority*	Functionality	Reason
Fastest	4	0	MC_TouchProbe and cam shift calculation	MC_TouchProbe.Done will trigger cam shift calculation
			Digital IO	Output needs to be fired within 10 ms of cam cycle
			Y_CamIn	Y_CamIn.EndOfProfile is used to fire the digital output
Fast	10	1	Y_CamShift	Every part is 50 ms apart and the shift is only 10 ms worth. Within range mode can be used for shifting the cam profile
Medium	20	2	EtherNet/IP	New data is obtained every 50 ms
Slow	50	3	AxisControl	Application scan time is not critical for these functions
			Jog	
			Homing	

* Set by the user

A screen shot of the task tree from a project is shown below in figure 3c. The fastest scan that includes critical motion functions (like MC_TouchProbe, Y_CamIn and digital IO in the rotary knife application) is set to 4 ms.


```
Physical Hardware*
  Configuration : MP2000_Series*
    Resource : MP2300Siec*
      Tasks
        Initial : SYSTEM
          Initialize : Initialize*
        Slow : CYCLIC
          IO_Interface : IO_Interface*
          Power_Alarm_Management : Po
          ManualMode_InterLock : Manua
          JawHome : JawHome*
          JawJog : JawJog*
          FeedJog : FeedJog*
        IOTask : CYCLIC
          IO : IOTask*
        Medium : CYCLIC
          JawMonitor : JawMonitor*
          FeedMonitor : FeedMonitor*
          Production_support : Production
        Fast : CYCLIC
          Main : Main*
      Global_Variables*
      IO_Configuration*
```

Task settings for MP2000_Series

Interval: 4 ms

Priority: 0

Watchdog Time: 4 ms

Stack: ( ) SMALL [✓ MEDIUM] ( ) LARGE ( ) XLARGE

Options: ☐ SAVE FPU ☐ BYPASS ☐ NO SUSPEND

Figure 3c: Task tree from a project

5 Step 3 – Programming Style Selection

A key feature of IEC 61131-3 based programming languages is flexibility in language selection. IEC 61131-3 specifies four programming languages and an organizational tool (sometimes defined as a fifth programming language). The four languages are Structured Text (ST), Function Block Diagram (FBD), Ladder Diagram (LD), and Instruction List (IL). Sequential Function Charts (SFCs) can be used to organize programs for machine sequences.

Unlike structured text or ladder, Sequential Function Charts (SFCs) are not a programming method by itself. SFCs offer a framework where different entities programmed in structured text or ladder can be encompassed in one flowchart like structure. This can be used for step and sequence based machines (Pick and Place) or to control processes with a state machine. Programming in structured text versus function block diagram is also a matter of personal preference. In table 2 below, the qualifications OK, good and best have been made based on ease of debugging and readability by a second person.

Table 2: Program functionality vs. Programming method

	Structured Text	**Function block and ladder**	**Sequential function chart**
Point To Point	OK	Good	Good
Math Calculations and conditional programming	Best	OK	-
Multi axis synchronized motion	OK	Best	-
State based sequential process control	-	-	Best
Maintenance	-	OK (for small code)	Best
Bit I/O Interfacing	-	Best	-
Speed of Execution	Best	-	-

OK	: Can debug and read to understand logic. (Can be time consuming though)
Good	: Easy to debug code and easy to understand logic
Best	: Easiest to debug and understand logic

6 Step 4 - Program Architecture Framework

An overview of a recommended program structure is given below (Table 3).

SYSTEM TASK:	Startup Task	
		Program and axes constant definitions
HIGH SPEED TASK:	Automatic Production	
		High Speed Motion (Camming gearing, point to point motion etc)
MEDIUM SPEED TASK:	Control (optional): To monitor machine state	
	IO and communication interface (To verify IO and HMI global variables)	
	Manual Mode	
		Jogging
		Homing
LOW SPEED TASK:	System Monitoring and Maintenance	
		Power up, alarm management
		E-Stop management
		Parameter saving/writing, absolute encoder position management

It is recommended to have projects built with the methods shown below. These basic functions can be elaborated upon depending on the machine and application requirements.

Table 3: Program architecture

POU Name	**Language**	**Reason for Language Selection**
Initialization	Structured Text	Easy to read
Control POU	Structured Text	Easy to debug and troubleshoot
IO Interface	Ladder diagram	Easy to debug
Communication interface	Structured text	Easy to transfer variables
Power up , alarm management and E-Stop recovery	Function Block diagram	Use of pre-defined PLCopen Toolbox
Axis Monitoring and Status	Function Block diagram	Use of pre-defined PLCopen Toolbox
Jogging	Function Block diagram	Use of pre-defined PLCopen Toolbox
Homing	Function Block diagram	Use of PLCopen function blocks
Automatic Production	Function Block diagram	Easy to debug and troubleshoot
Maintenance and upgrade	Function Block diagram	Use of PLCopen function blocks

Make sure that the necessary Yaskawa user libraries are included in the project. It is good practice to save user libraries in one specific location. When a project that contains user libraries is extracted, the user libraries get saved to different locations on Windows XP and Windows 7.

XP: `C:>Documents and Settings> All Users> Documents> MotionWorks IEC Pro> Libraries> User Library Name.mwt.`

Win 7: `c:>Users>Public>Public Documents> MotionWorks IEC 2 Pro> Libraries> User Library Name.mwt`

This folder is a good candidate to store all user libraries

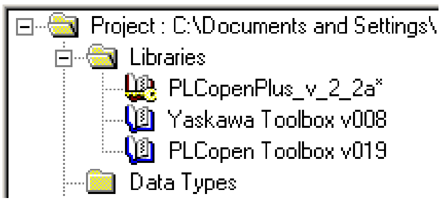


Figure 4: User libraries in project tree

7 Step 5 – Code Development

Once the program architecture and programming methods have been designed, the user can start building the project by developing code. The following sub sections will elaborate on all the elements mentioned in the recommended program architecture framework. The basic building block of a project is a Program Organization Unit (POU). This is the code where the logic and commands are created. Once POU instances are created, they need to be placed in tasks under the resource (MPiec controller). The code in a POU will get executed only if they are associated with a task. The code gets executed at the scan interval set by the user for the task in which the code is placed. A screen shot portraying the association of a POU to a task is shown below.

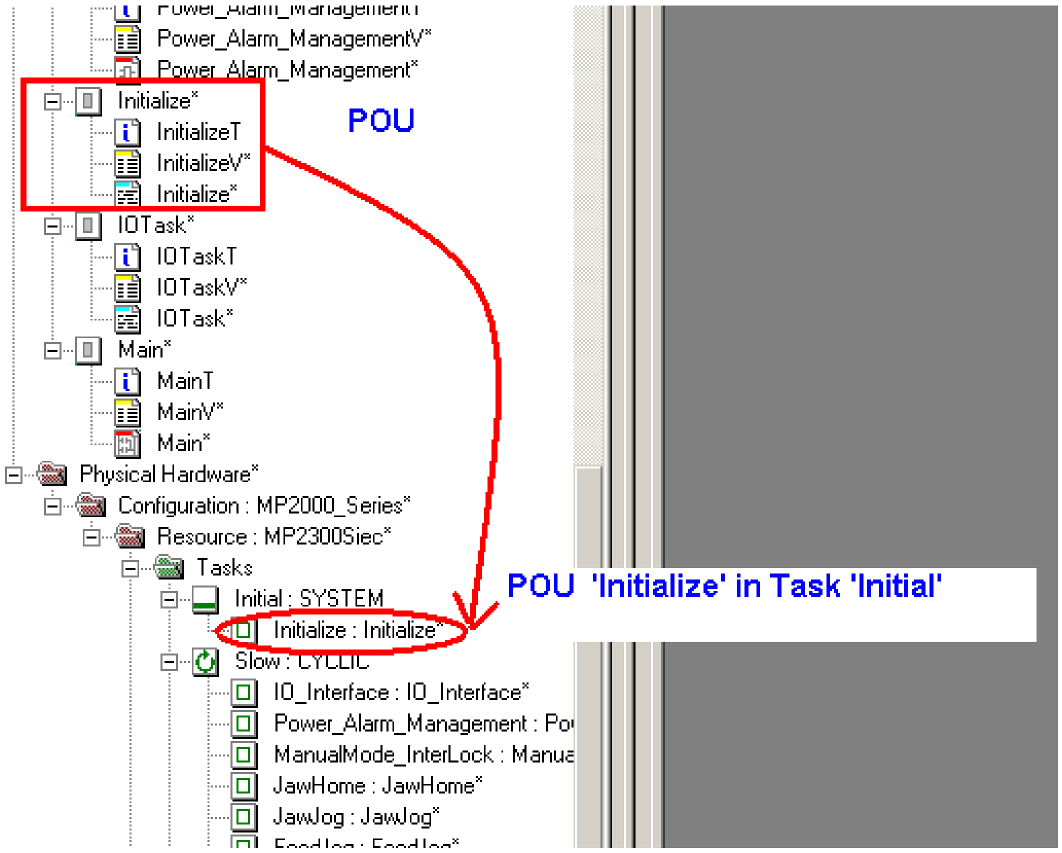
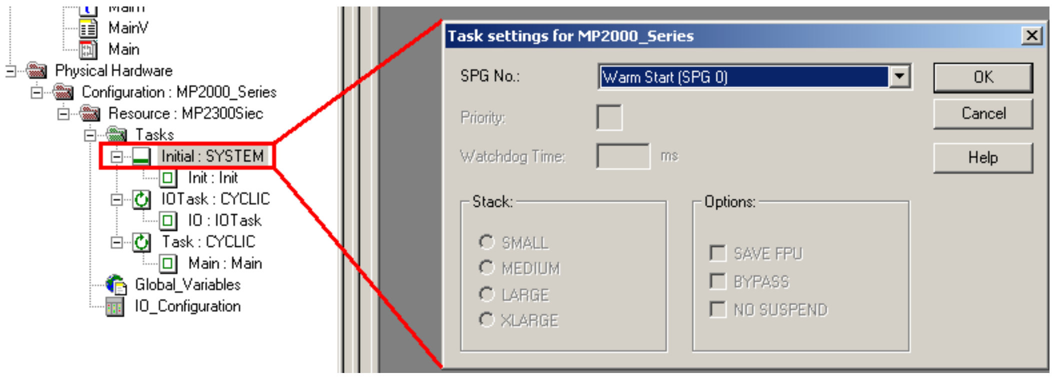


Figure 5a: POU associated with a task

7.1 Start up Task (Initialization)

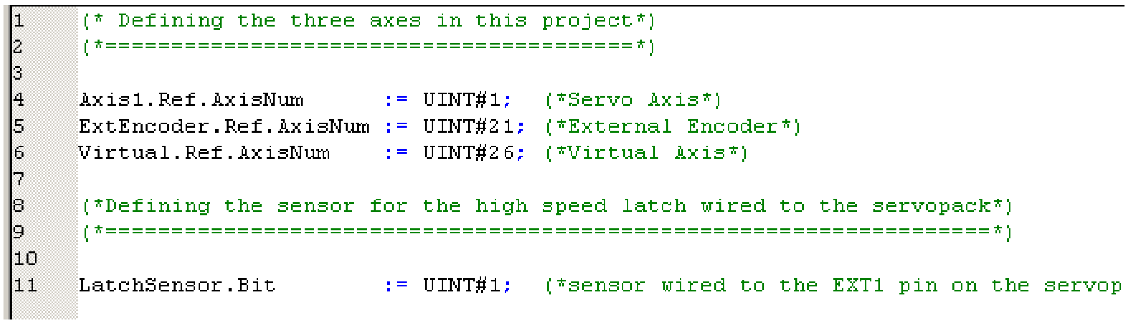
In MotionWorksIEC Pro, any POU that needs to be executed only once can be defined to be executed within a system task that is associated with an event or a system program. Examples of system programs are warm start, cold start, PLC watchdog etc. Any initialization POU containing axes and/or variable definition can be associated with a system task tied to a warm start system program. A controller starts in warm start mode on boot up.



SPG No.: Warm Start (SPG 0)

Figure 5b: System Task

A few examples of items that can be included in startup system tasks are given below.



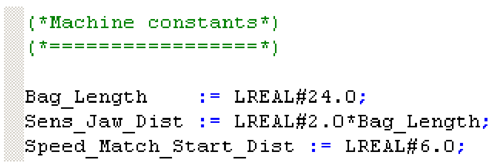
```
(* Defining the three axes in this project*)
(*========================================*)

Axis1.Ref.AxisNum       := UINT#1;  (*Servo Axis*)
ExtEncoder.Ref.AxisNum := UINT#21; (*External Encoder*)
Virtual.Ref.AxisNum    := UINT#26; (*Virtual Axis*)

(*Defining the sensor for the high speed latch wired to the servopack*)
(*===================================================================*)

LatchSensor.Bit         := UINT#1;  (*sensor wired to the EXT1 pin on the servop
```

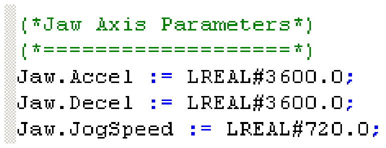
Figure 6: ID definitions



```
(*Machine constants*)
(*=================*)

Bag_Length    := LREAL#24.0;
Sens_Jaw_Dist := LREAL#2.0*Bag_Length;
Speed_Match_Start_Dist := LREAL#6.0;
```

Figure 7: Constant declarations

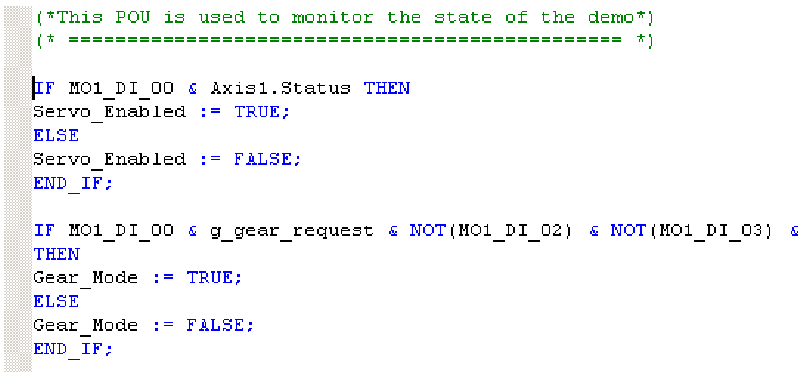


```
(*Jaw Axis Parameters*)
(*===================*)
Jaw.Accel := LREAL#3600.0;
Jaw.Decel := LREAL#3600.0;
Jaw.JogSpeed := LREAL#720.0;
```

Figure 8: Constant parameter definitions

7.2 Control POU (status)

A control POU may act as a centralized location for the operator to determine what state the machine is in at any given point of time. Proper interlocking of HMI requests and internal variables will help in trouble shooting if the machine fails to respond to commands in a particular mode. A central location like the one shown in the example below will help in locating trouble spots in program execution. It is recommended to split up machine operations into mutually exclusive modes like manual, automatic, maintenance, etc. with clear interlocks. This will prevent unwanted motion or unforeseen logic execution and hasten troubleshooting.



```
(*This POU is used to monitor the state of the demo*)
(* =============================================== *)

IF M01_DI_00 & Axis1.Status THEN
Servo_Enabled := TRUE;
ELSE
Servo_Enabled := FALSE;
END_IF;

IF M01_DI_00 & g_gear_request & NOT(M01_DI_02) & NOT(M01_DI_03) &
THEN
Gear_Mode := TRUE;
ELSE
Gear_Mode := FALSE;
END_IF;
```

Figure 9: Interlocks for Control POU

7.3 IO Interface

The controller, servopack or MECHATROLINK inputs (global variables) should be introduced into the project through one interface POU. This will help in troubleshooting and testing during installation. In the same way, output variables can be pushed out of the project through this IO interface POU. A recommended flow chart for input/output variable usage is shown below. It should be noted that in addition to associating IO drivers to particular tasks, the program units in which the IO variables are used must also be associated to tasks that have scan rates that match the IO update requirements of the application.

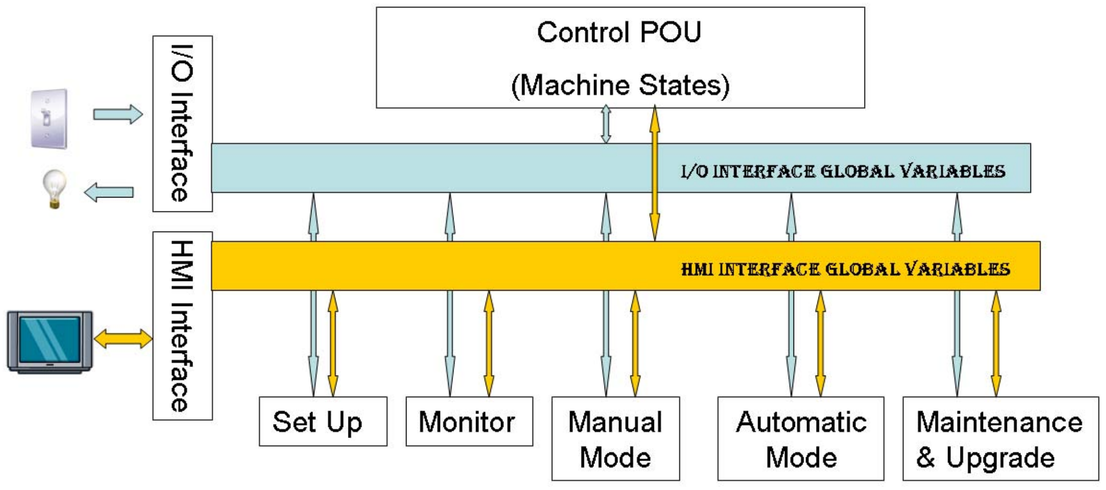


Figure 10: IO interface channels

An example of an IO interface POU is given below.

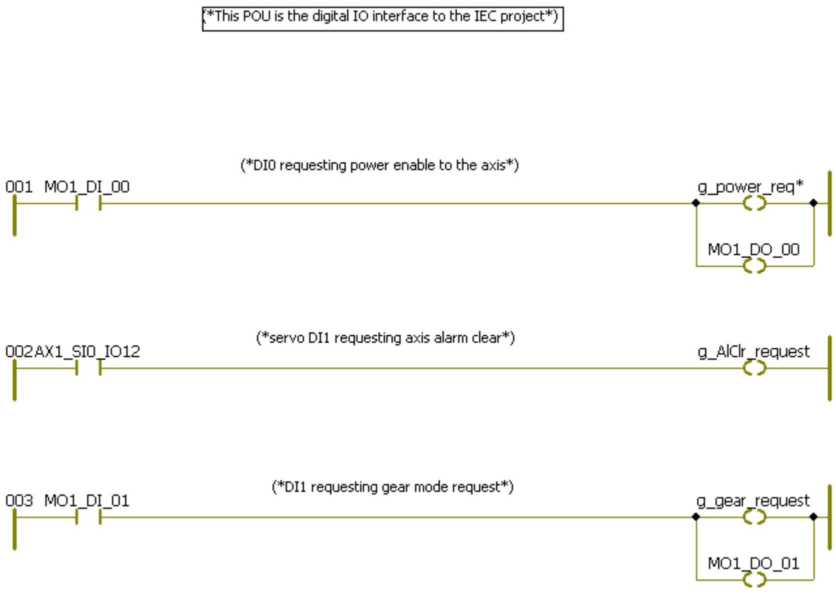
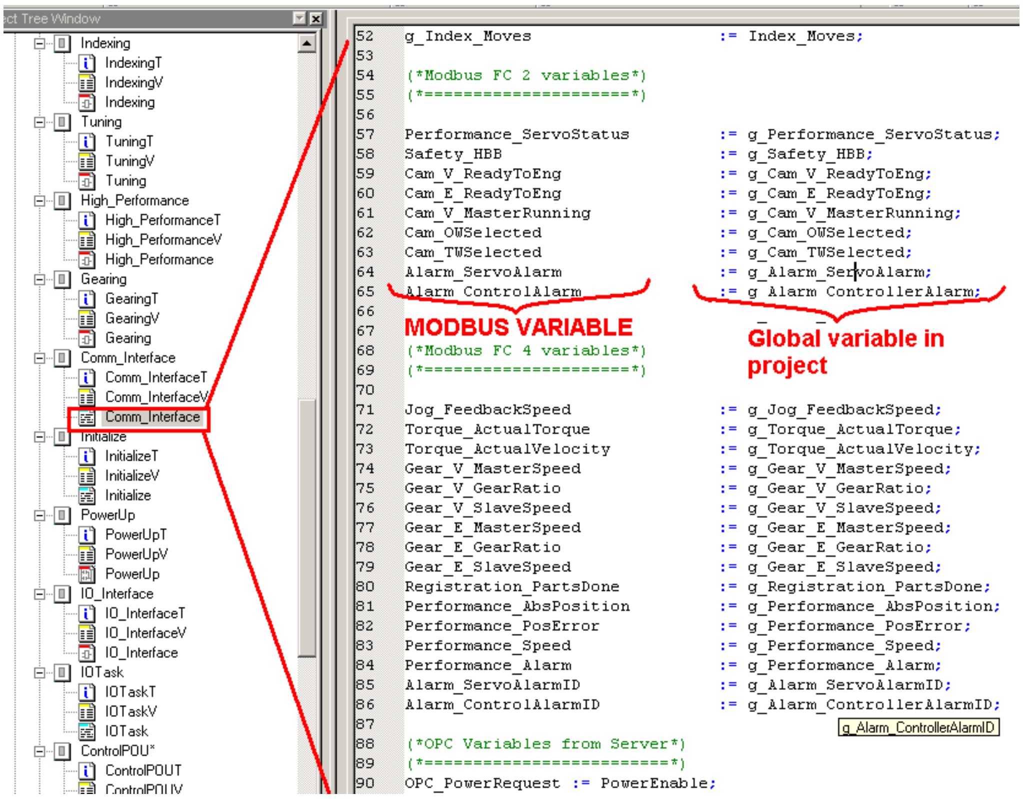

(*This POU is the digital IO interface to the IEC project*)
(*DI0 requesting power enable to the axis*)
001 MO1_DI_00
g_power_req*
MO1_DO_00
(*servo DI1 requesting axis alarm clear*)
002AX1_SI0_IO12
g_AlClr_request
(*DI1 requesting gear mode request*)
003 MO1_DI_01
g_gear_request
MO1_DO_01

Figure 11: IO interface POU

7.4 Communication Interface

Similar to having a POU for IO interfacing, a POU for communication or network variable interfacing is very important to help trouble shoot, test and install communication between the MPiec controller and the network device. With such an interface POU in place, testing for proper communication can be done in a centralized location. Shown below is an example of transferring HMI variables to global variables that can be used in the project.



```
g_Index_Moves                    := Index_Moves;

(*Modbus FC 2 variables*)
(*=====================*)

Performance_ServoStatus          := g_Performance_ServoStatus;
Safety_HBB                       := g_Safety_HBB;
Cam_V_ReadyToEng                 := g_Cam_V_ReadyToEng;
Cam_E_ReadyToEng                 := g_Cam_E_ReadyToEng;
Cam_V_MasterRunning              := g_Cam_V_MasterRunning;
Cam_OWSelected                   := g_Cam_OWSelected;
Cam_TWSelected                   := g_Cam_TWSelected;
Alarm_ServoAlarm                 := g_Alarm_ServoAlarm;
Alarm_ControlAlarm               := g_Alarm_ControllerAlarm;


(*Modbus FC 4 variables*)
(*=====================*)

Jog_FeedbackSpeed                := g_Jog_FeedbackSpeed;
Torque_ActualTorque              := g_Torque_ActualTorque;
Torque_ActualVelocity            := g_Torque_ActualVelocity;
Gear_V_MasterSpeed               := g_Gear_V_MasterSpeed;
Gear_V_GearRatio                 := g_Gear_V_GearRatio;
Gear_V_SlaveSpeed                := g_Gear_V_SlaveSpeed;
Gear_E_MasterSpeed               := g_Gear_E_MasterSpeed;
Gear_E_GearRatio                 := g_Gear_E_GearRatio;
Gear_E_SlaveSpeed                := g_Gear_E_SlaveSpeed;
Registration_PartsDone           := g_Registration_PartsDone;
Performance_AbsPosition          := g_Performance_AbsPosition;
Performance_PosError             := g_Performance_PosError;
Performance_Speed                := g_Performance_Speed;
Performance_Alarm                := g_Performance_Alarm;
Alarm_ServoAlarmID               := g_Alarm_ServoAlarmID;
Alarm_ControlAlarmID             := g_Alarm_ControllerAlarmID;

(*OPC Variables from Server*)
(*=========================*)
OPC_PowerRequest := PowerEnable;
```

Figure 12: Communication interface POU

All command bits from an HMI to the controller should be programmed as latched momentary signals. This provides safety in the event of E-Stop or power off conditions. An example of an HMI input ‘g_HMI_Start’ triggering motion with ‘Motion_Start’ is shown below. Variable ‘System_OK’ indicates that the axis is ready and there are no alarms on the axis or on the controller.

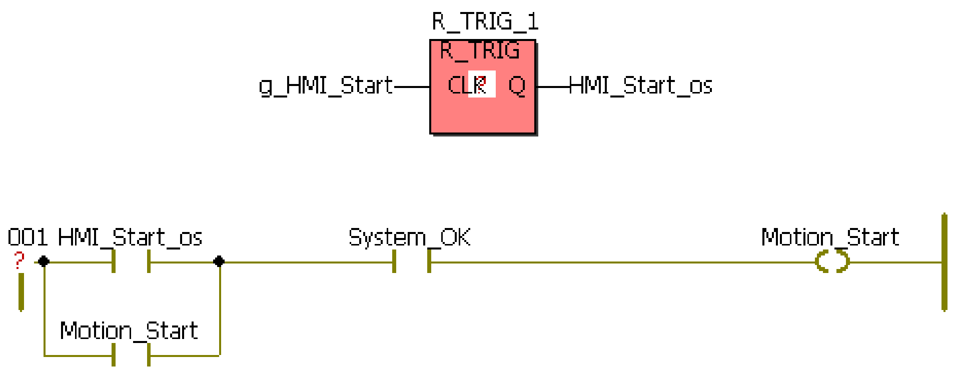


Figure 13: Rising edge signal from HMI

To avoid damage to eh system, limit ranges of motion parameters like position, velocity, acceleration that are commanded from an HMI.

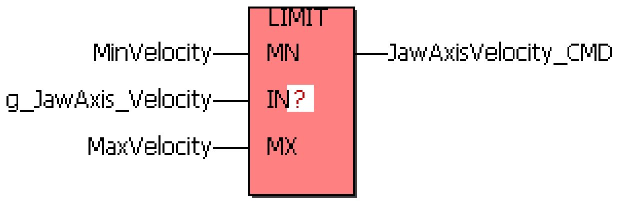


Figure 14: Safety limits on user input variables

7.5 Power up, alarm management, E-Stop recovery

In the power up sequence to enable an axis, it is important to confirm that:

- The controller is ready without alarms
- The servopack is ready without alarms
- The servopack is in base block state and main power to the servopack is on

The 'AxisControl' function block from the PLCopen Toolbox helps in consolidating all axis related warnings and alarms that affect the servopack and controller. Any controller alarms (not axis related) can be obtained using the 'ControllerAlarm' block.

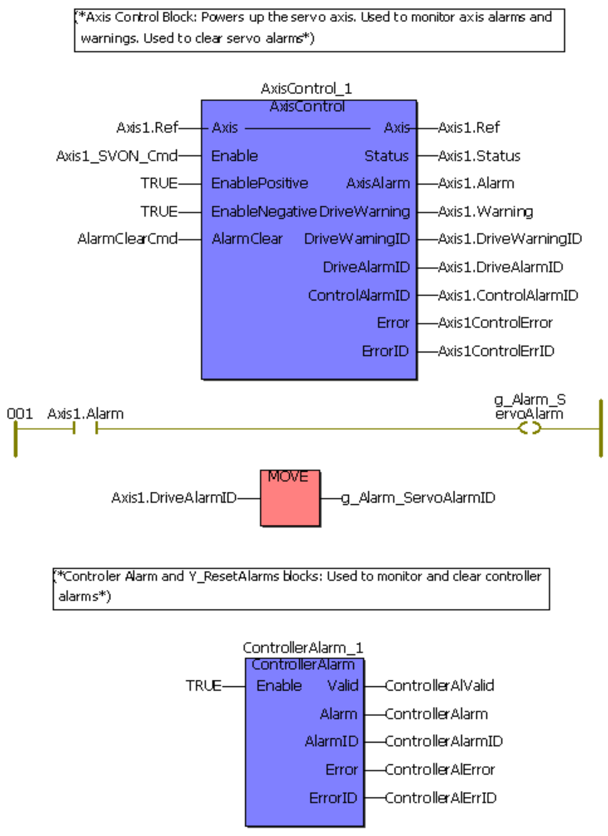


Figure 15: Power up and alarm monitoring

In the code example shown below, ‘AxisNormal’ and ‘ControllerNormal’ are conditions required to enable the servo. Main power loss, hardware base block (HBB), communication errors or other axis related alarms as well as controller specific alarms will disable the axis. Also, the user input is in series with a rising edge one shot variable to prevent rapid toggling of the servo on request signal.

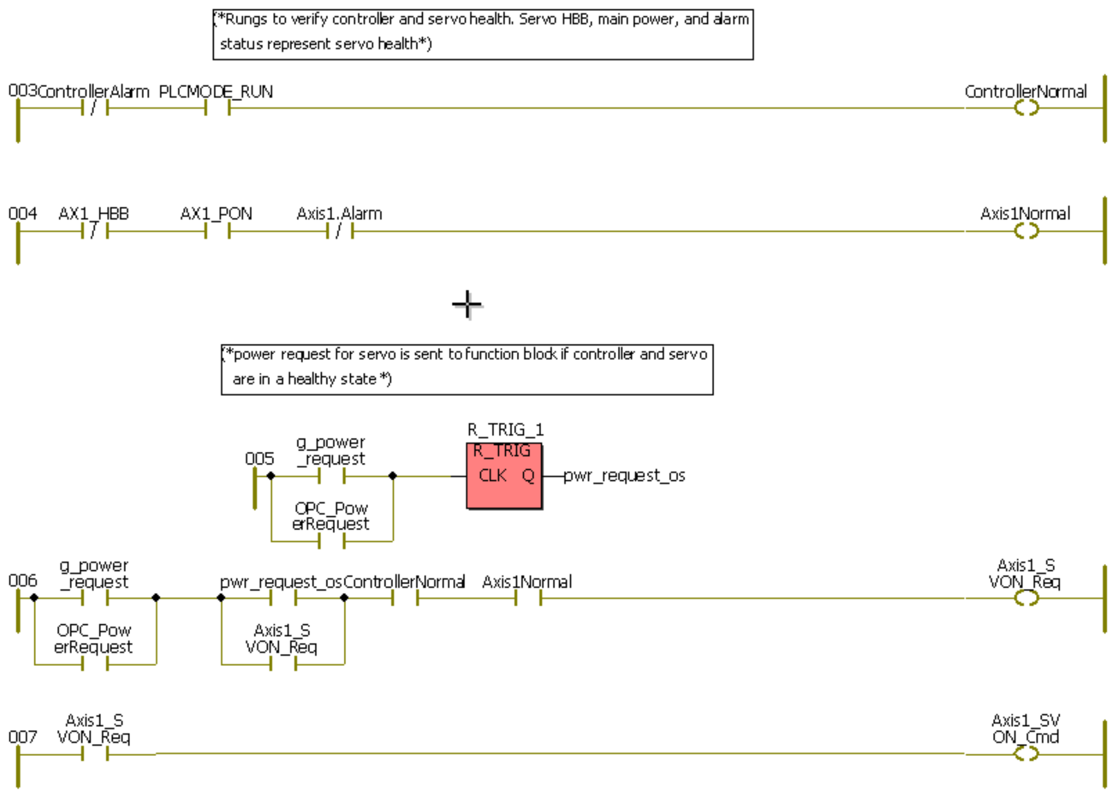

(*Rungs to verify controller and servo health. Servo HBB, main power, and alarm status represent servo health*)
003 ControllerAlarm PLCMODE_RUN ControllerNormal
004 AX1_HBB AX1_PON Axis1.Alarm Axis1Normal
(*power request for servo is sent to function block if controller and servo are in a healthy state*)
005 g_power_request OPC_PowerRequest
R_TRIG_1 R_TRIG CLK Q pwr_request_os
006 g_power_request OPC_PowerRequest pwr_request_os ControllerNormal Axis1Normal Axis1_SVON_Req Axis1_SVON_Req
007 Axis1_SVON_Req Axis1_SVON_Cmd

Figure 16: Servo enable command if controller and servopack are normal

Alarm clearing:

Alarms or errors can be from three sources. They are:

1) Servopack: The controller can display servopack alarms and warnings that appear on the servopack

display. The controller can also display servopack specific alarms that the controller encountered. These may not be displayed on the servopack. An example is an invalid watchdog code from a servopack because of lost MECHATROLINK communication. These alarms and warnings are clearly distinguished and thay can be cleared if the user uses the axis control block from the PLCopen toolbox.

2) Controller: Any controller specific alarms can be accessed using the Controller alarm function block (figure 15) from the PLCopen toolbox. YClearAlarms can be used to clear controller specific alarms

3) Function block errors: function block errors can be cleared only if the condition causing the error is cleared either in the machine operation or in the code.

Sample code that can be followed to initiate an alarm clear request is shown below. AlarmClearCmd commands a controller alarm clear on ‘Y_ClearAlarms’ and an axis alarm clear through ‘AxisControl’ shown in figure 15.

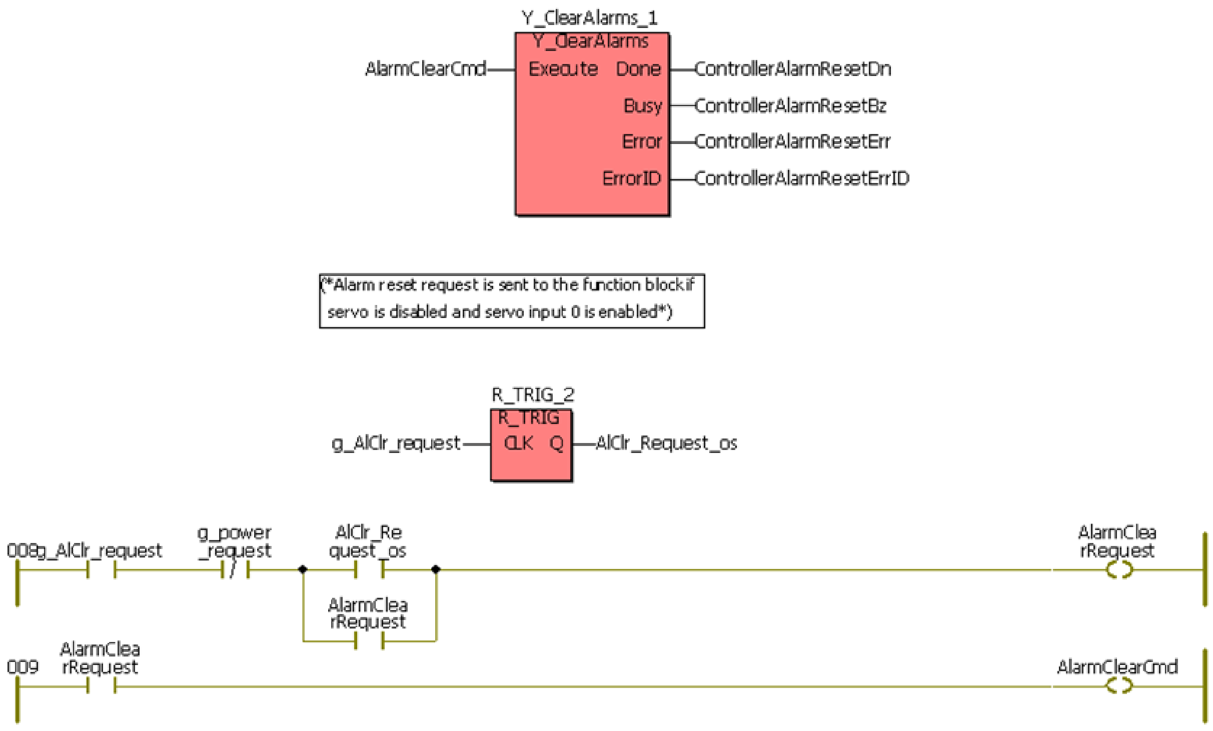


Figure 17: Alarm clearing command

E-Stop Recovery

It is recommended that an E-Stop recovery (either using HBB or using main power on the servopack) be carried out by clearing all alarms, then clearing HBB or servopack main power loss conditions. It is recommended that a new servo on request be sent (with operator intervention) to enable the axis after an E-Stop. This is the recommended sequence for machines that are brought to a hardware base block condition using light curtains as well. Such a sequence can be accomplished by the logic shown in the power on sequence above (Figure 17).

7.6 Axis Monitoring and status

The ReadAxisParameters function block in the PLCopen Toolbox lists all the axis parameters in a structure format. Individual parameters can be extracted as shown below. The axis state can also be determined using MC_ReadStatus.

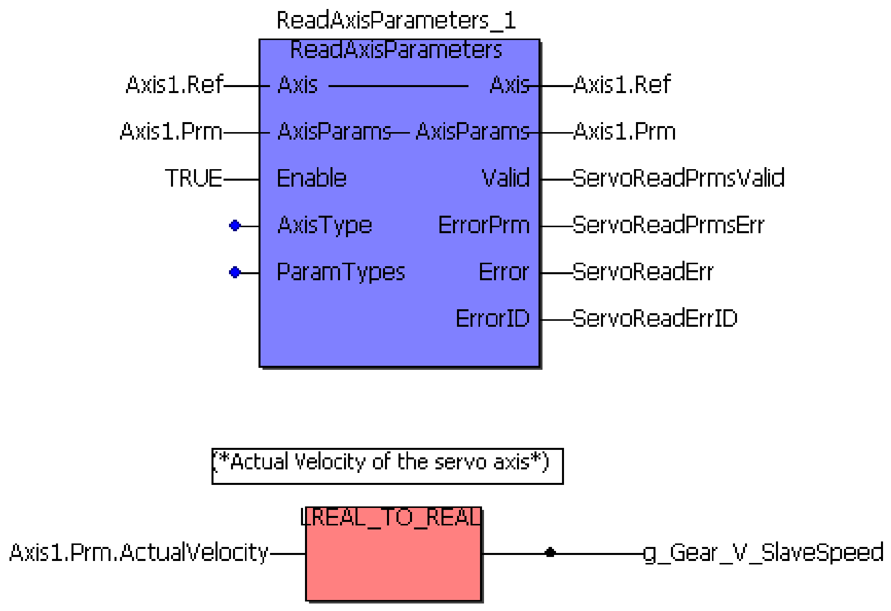
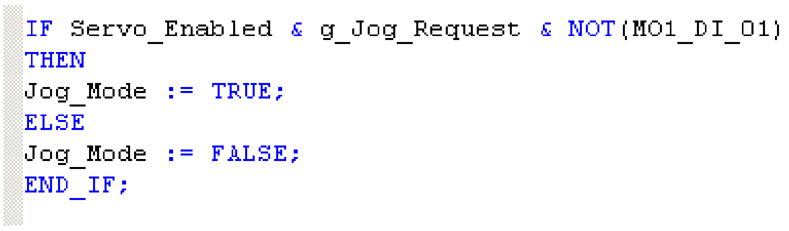


Figure 18: Axis parameter monitoring

7.7 Jogging

The interlocks for jogging typically involve ensuring that the axis is enabled and the machine is in manual mode. The Jog function block in the PLCopen Toolbox ensures that the forward and reverse jog commands are not sent to the axis simultaneously.



```
IF Servo_Enabled & g_Jog_Request & NOT(MO1_DI_01)
THEN
Jog_Mode := TRUE;
ELSE
Jog_Mode := FALSE;
END_IF;
```

Figure 19: Jogging interlock

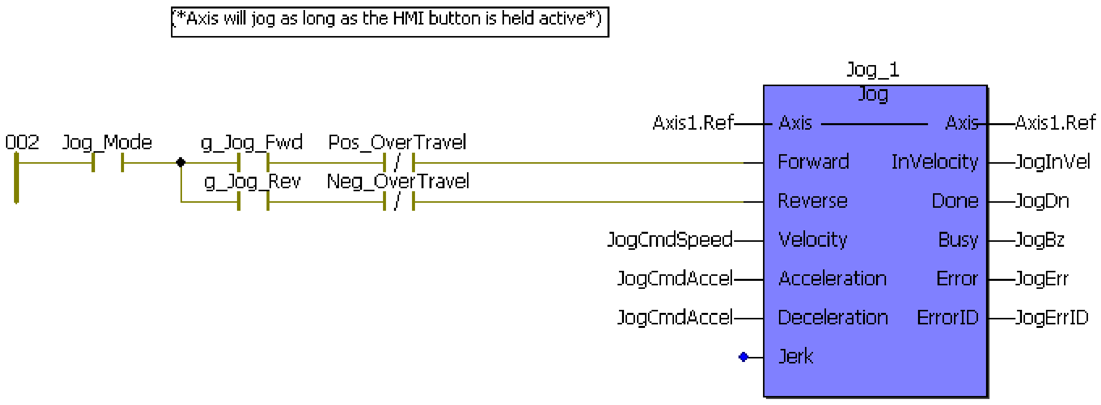


Figure 20: Jogging logic

7.8 Homing

A few recommended interlocks for homing are: servo enabled condition, servo in standstill state, Axis not E-Stopped, servopack and controller normal without alarms. 'HomeFBReady' makes sure that the function block used for homing is ready to start the routine. A one shot from the HMI variable should start the homing routine if the axis is ready for homing. Once the homing routine starts, busy or error output bits used as contacts to energize the execute input will hold at least one output high during the execution of the homing routine. Please refer to the application design guideline for supported PLCopen homing methods. There are two PLCopen specified homing methods currently supported by MotionWorksIEC
http://partner.yaskawa.com/site/dmcontrol.nsf/(DocID)/BFAR-7Q3PY5?opendocument&login
(You must be logged in to view this document, as it is secure. If you are not logged in and you click the link, you will be asked to login first. Once logged in the file will appear.)

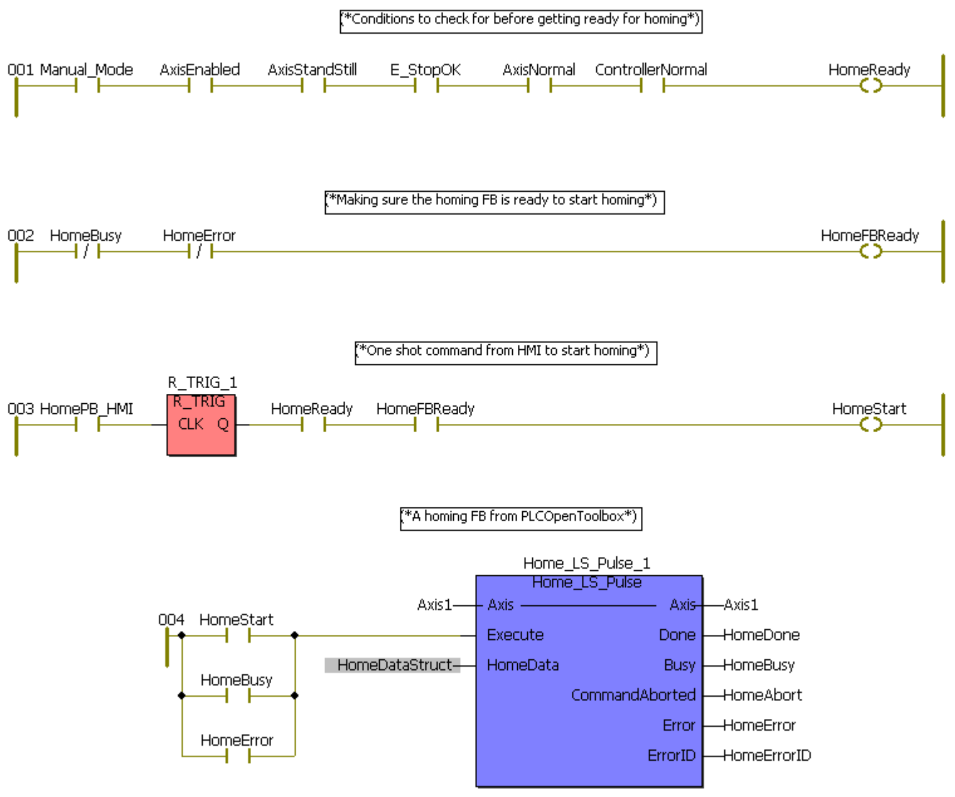


Figure 21: Home logic

7.9 Automatic production

A recommended interlock logic sequence for automatic production is shown below.

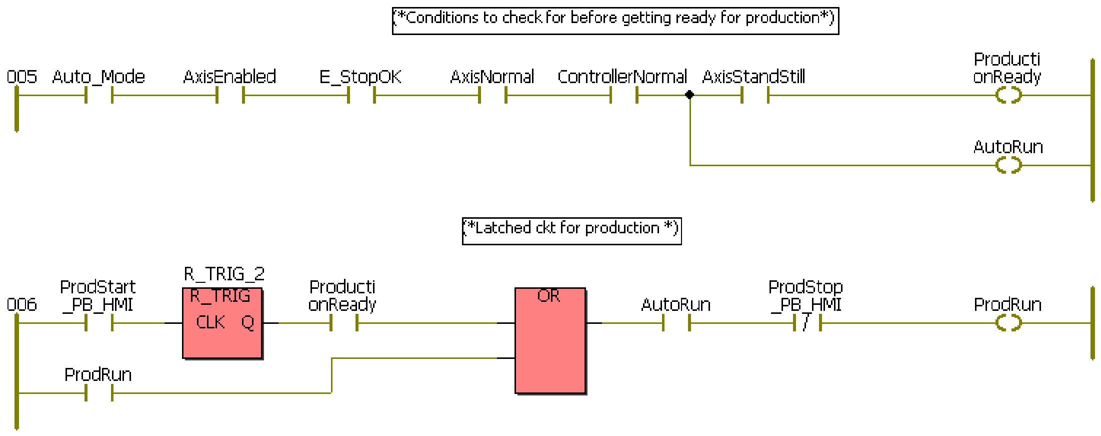


Figure 22: Automatic production interlocks

The assignment of motion POUs (with motion function blocks) to particular tasks has to be carried out prudently. It is important to understand how function blocks in MotionWorksIEC interact with the motion engine and motion is executed by the motion engine in MPiec series controllers.

Function blocks like MC_MoveAbsolute, MC_MoveVelocity, and Y_CamIn are used to configure the motion profile and prepare the motion engine to calculate the profile which is then commanded. Once the motion engine is configured to execute a motion profile calculations are carried out every MECHATROLINK cycle and commands are sent out to the servopack at the MECHATROLINK rate. Therefore, contrary to intuition, associating motion POUs to fast high priority tasks is not a requirement to obtain accurate deterministic motion at the servo. This is because motion profile calculations are carried out at the MECHATROLINK update rate irrespective of what task interval the POU or motion function block is placed under.

Assigning manual mode motion operations like jogging, HMI communication, homing, power up sequence etc to low priority slower tasks gives the controller more resources to be utilized for fast tasks that require frequent updates if and when required. Placing all motion POUs in fast tasks will end up causing CPU watchdog timeouts and does not improve precision for the servo axis.

7.9.1 Point to Point motion:

Typical point to point motion blocks can be used in POUs assigned to tasks which have medium priority. The cases where motion blocks need to be updated at fast rates are:

a) If moves are being aborted based on a high speed input and the profile change needs to be immediate
b) If a certain frequency or cycle time needs to be achieved for throughput. The task will have to update faster than the move frequency.

7.9.2 Synchronous motion:

The cam engage block Y_CamIn is used to prepare the motion engine with the slave and master axes IDs, the cam table to use and when to engage the slave to the master. The actual engage and cam table interpolation calculation are performed at the MECHATROLINK update rate. This function block needs to be updated only with medium priority. The only case where high update rates are required are:

a) If a cam adjustment needs to be carried out immediately based on a high speed input. For example, if a shift needs to be carried out based on an input and the shift needs to be carried out as soon as possible, the shift block will have to be in a high priority task.

7.10 Maintenance and upgrade

A few function blocks and lines of logic in the project on a machine can ensure easy replacement of amplifiers and motors if needed without having to worry about having to go thorough difficult machine start up and configuration steps. Please consult the documents from the links given below for recommendations on programming for amplifier and motor replacement at maintenance.

7.10.1 Amplifier replacement:

http://www.yaskawa.com/site/dmcontrol.nsf/(DocID)/JSOT-7ZXTNM?opendocument

7.10.2 Servomotor replacement:

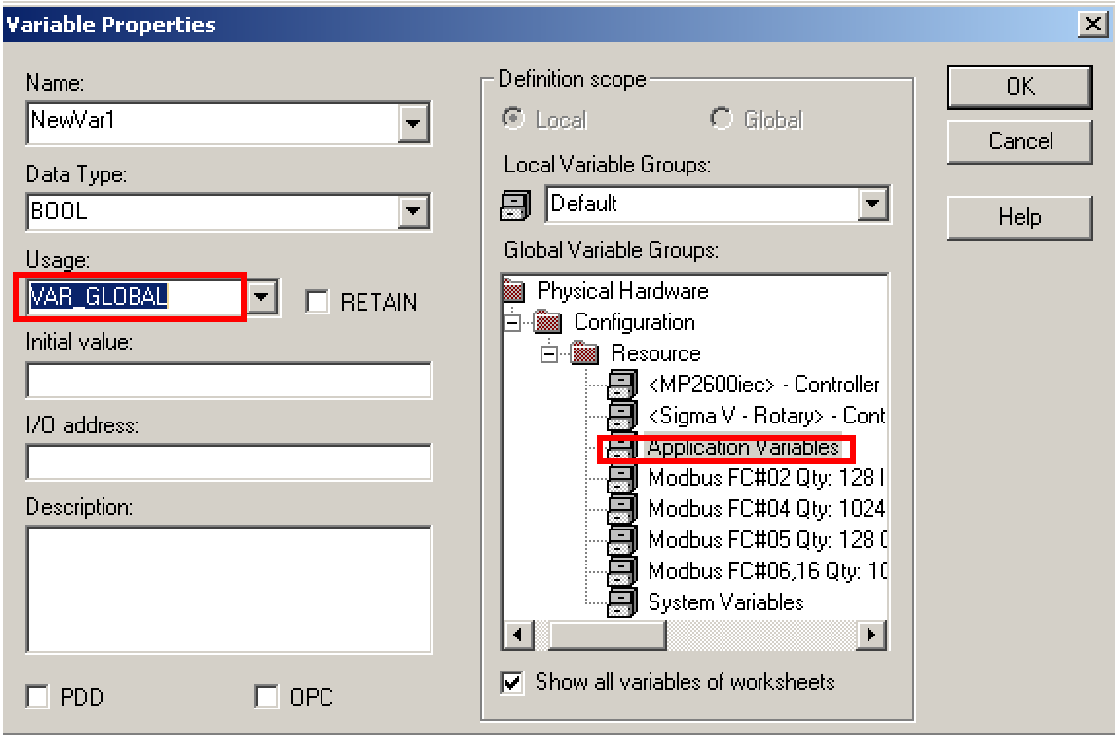
http://www.yaskawa.com/site/dmcontrol.nsf/(DocID)/MPER-83UM8U?opendocument

8 Coding Best Practices

8.1 Assigning variable names and groups

If a variable does not have to be a global variable keep it local. One may use a distinguishing feature when naming global variables. (eg: g_newvariable)

For every new global variable created, assign the group it needs to go into in the global variable worksheet



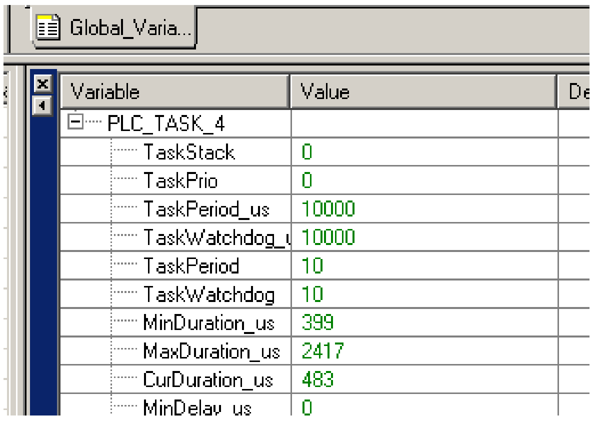
Name: NewVar1
Data Type: BOOL
Usage: VAR_GLOBAL
RETAIN
Initial value:
I/O address:
Description:
PDD
OPC
Local Variable Groups: Default
Show all variables of worksheets

Figure 23: Variable declaration

8.2 Use of Task time monitoring in order to optimize project

The MP2600iec and MP23xxiec controllers have different methods for reading the high resolution task times.

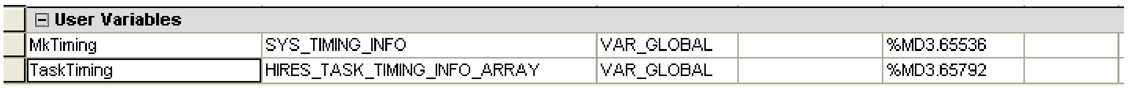
For the MP2600iec controller, the task interval can be monitored with microsecond precision using variables in the global variable list. MaxDuration_us is the maximum time that particular task took to execute. CurDuration is the time that task is taking to run in real time. These variables can be used to decide on whether a task requires to be given more time to run or if some other non urgent tasks can be moved to lower priority levels.



Variable	Value	De
PLC_TASK_4		
TaskStack	0	
TaskPrio	0	
TaskPeriod_us	10000	
TaskWatchdog_u	10000	
TaskPeriod	10	
TaskWatchdog	10	
MinDuration_us	399	
MaxDuration_us	2417	
CurDuration_us	483	
MinDelay_us	0	

Figure 24: Task time real time monitoring for task 4

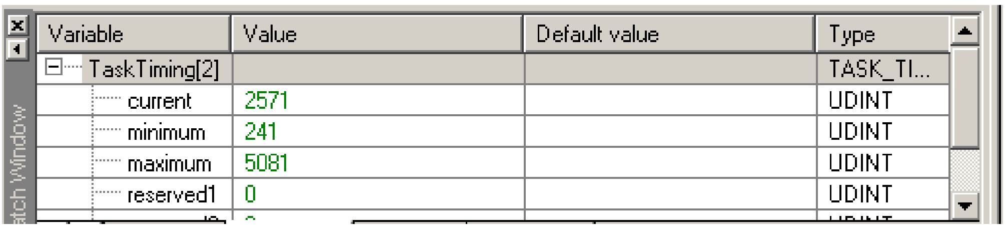
For the the MP2300iec and the MP2310iec controllers can be obtained as follows: Add two variables (one for MECHATROLINK timing and one for task timing) with the data types and addresses in the global variable list of the project as shown in Figure 24a below



User Variables					
MkTiming	SYS_TIMING_INFO	VAR_GLOBAL		%MD3.65536	
TaskTiming	HIRES_TASK_TIMING_INFO_ARRAY	VAR_GLOBAL		%MD3.65792	

Figure 24a

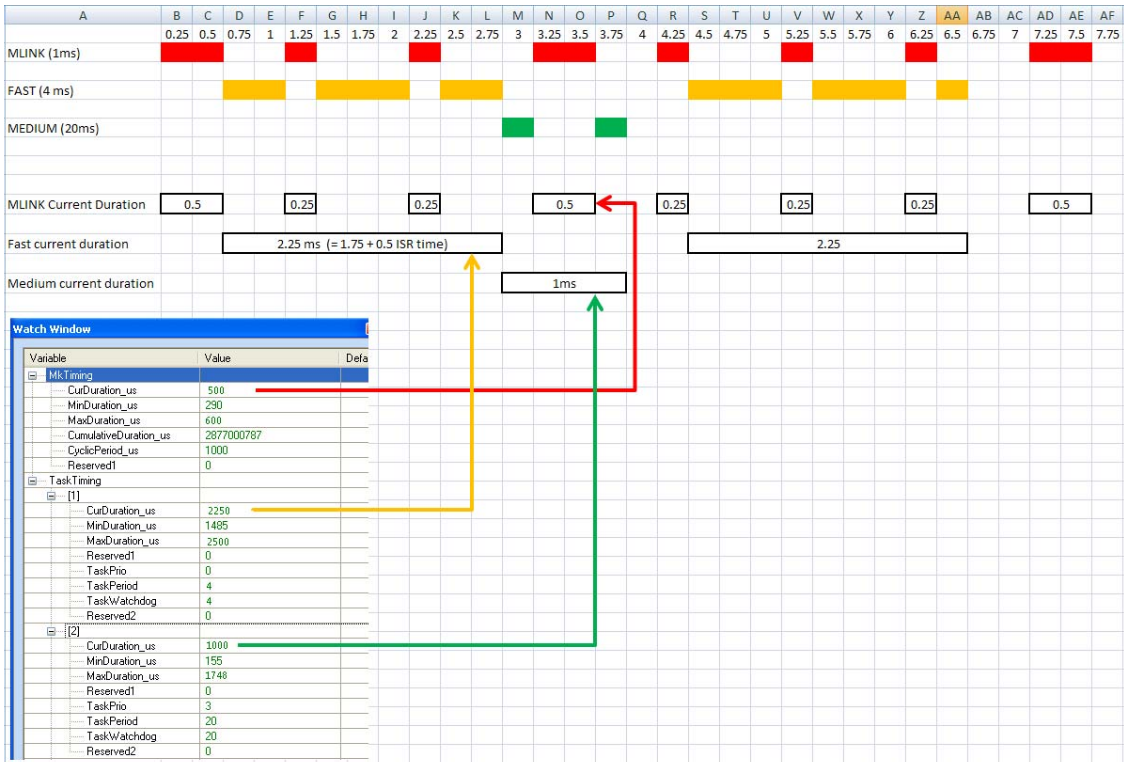
Add the two variables to the watch window to view timings of individual tasks in the user program.



Variable	Value	Default value	Type
TaskTiming[2]			TASK_TI...
current	2571		UDINT
minimum	241		UDINT
maximum	5081		UDINT
reserved1	0		UDINT

Figure 24b

An example of how to interpret data for a project with MECHATROLINK update of 1 ms, fast task update of 4 ms and a medium task update of 20 ms is shown below.



Variable	Value	Defa
MkTiming		
CurDuration_us	500	
MinDuration_us	290	
MaxDuration_us	600	
CumulativeDuration_us	2877000787	
CyclicPeriod_us	1000	
Reserved1	0	
TaskTiming		
[1]		
CurDuration_us	2250	
MinDuration_us	1485	
MaxDuration_us	2500	
Reserved1	0	
TaskPrio	0	
TaskPeriod	4	
TaskWatchdog	4	
Reserved2	0	
[2]		
CurDuration_us	1000	
MinDuration_us	155	
MaxDuration_us	1748	
Reserved1	0	
TaskPrio	3	
TaskPeriod	20	
TaskWatchdog	20	
Reserved2	0	

Fig 24c: Interpretation of task timing data

8.3 Use of PLCopenToolBox

This toolbox is built from standard basic PLCopen motion function blocks resulting in easy to-use higher-level function blocks. A few examples are:

AxisControl: Used to enable an axis and monitor the state of the axis to verify if there are axis alarms or warnings. Axis Alarms can be cleared from this function block as well

ReadAxisParameters: All axis parameters have been listed with appropriate data types such that the user can use this one function block and get a structure with all parameters as an output. This saves the user from having to use multiple read parameter blocks for multiple axis parameters

Jog: Built from the PLCopen MC_MoveVelocity. This function blocks enables the user to switch commanded velocity as well as direction on the fly.

Please refer to
http://yaskawa.com/site/products.nsf/ProductDetailPages/Multi-Axis%20Motion%20Controllers~MP2000iec%20Series~MP2000iec_Application_Toolboxes.html
for details about the function blocks in the toolbox including the manual explaining functionality of all the function blocks

8.4 User created function blocks

It is recommended that users create user defined function blocks and store it in a separate library that can be reused. Most programmers program machines that have similar core motion functionality and logic. The advantage of creating one's own function block is that the code can be reused for machines that have similar logic or motion functionality. It is beneficial to create function blocks adhering to PLCopen standards. General rules for inputs and outputs for function blocks can be obtained at
plcopen.org

There are general rules about the way function blocks execute like output exclusivity, behavior of the output bits when the execute bit turns off before the function block is done etc. If such rules are followed during code development, such code will be scalable and easy to understand. The behavior of user defined function blocks and Yaskawa's motion function blocks will be similar. This makes code readable and easy to understand. A good practice can be seen in the function blocks in the PLCopen Toolbox.

Always include comments about the working of user defined function block in the source code. If the function block has complex motion and complex structures being input or output from the block, a short manual explaining the inputs, outputs and functionality along with a timing diagram of the interacting signals is recommended.

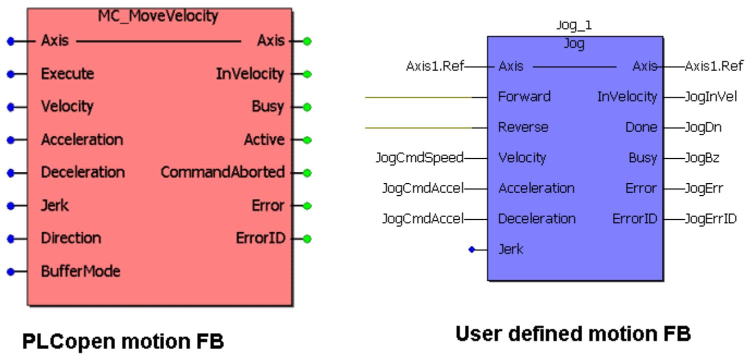


Figure 25: User created function block: Jog

8.5 All FB outputs should be connected for easy trouble shooting

If outputs of function blocks are not mapped to variables, it will be difficult to trouble shoot motion or machine functionality. In the case of the example provided below, without the outputs to the MC_TorqueControl block connected, it would be difficult to trouble shoot torque mode motion if the axis does not behave as expected. It is not possible to say if the function block is executing or not since the outputs are missing. This is equivalent to not connecting the feedback loop in a closed loop circuit.

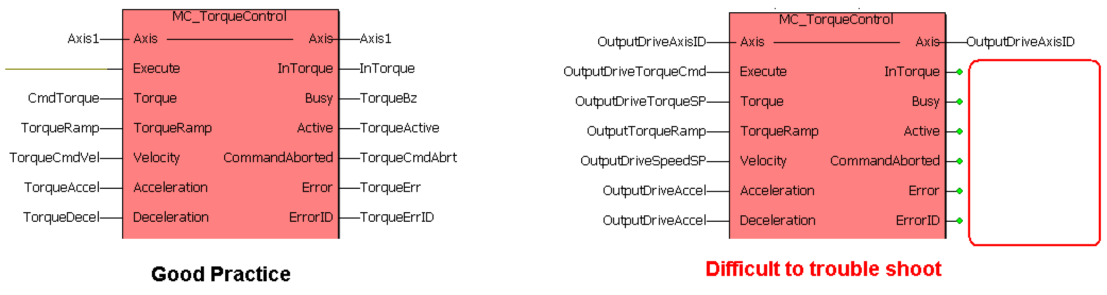


Figure 26: Function block outputs should be connected

8.6 Latching execute bits for FB execution

In case a function is being executed by a momentary bit, it is recommended to have the execute input latched with outputs bits of the same function block that can demonstrate the status of that function blocks impact on the axis. The PLCopen specification states that if the execute bit is held high, one of the following output bits must be high (only one at a time). The specified outputs bits are done, busy, aborted and error. Having these outputs bits latch the execute bit will help the programmer monitor the status of the function block as it controls the axis even if it gets activated by a momentary contact.

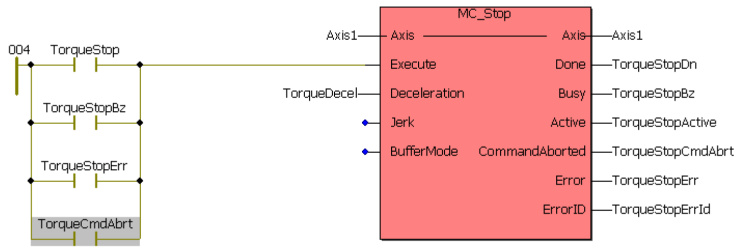


Figure 27: Latching execute bits

8.7 Use of local vs. global variables

It is recommended to keep the number of global variables to the bare minimum. Try to decide on the variables that need to be shared between POUs before starting to program. This way programming errors due to data overwriting can be kept to a minimum. If a variable can be kept local, it should be.

8.8 Function Block Copy and Paste

Caution should be exercised while copying and pasting function blocks. Instance names should be checked in the case of copying function blocks. The figure shown below can lead to bad logic and may be difficult to trouble shoot if care is not taken while programming itself. It is recommended to avoid copying and pasting without planning and monitoring what is being copied.

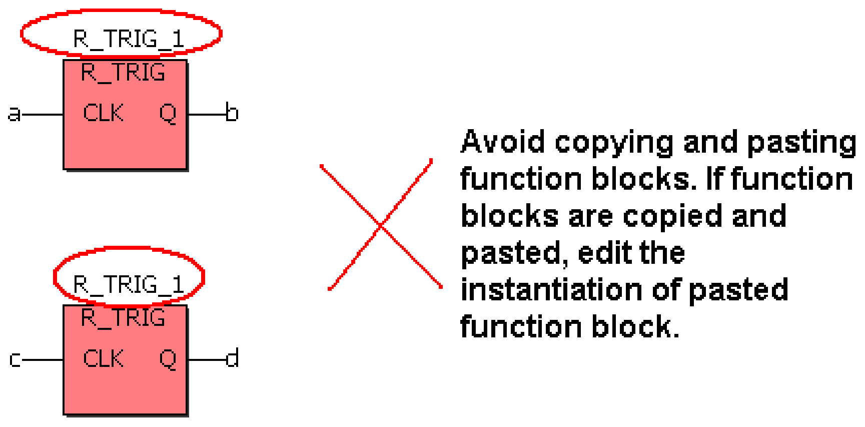

R_TRIG_1
R_TRIG
CLK
Q
a
b
R_TRIG_1
R_TRIG
CLK
Q
c
d
Avoid copying and pasting function blocks. If function blocks are copied and pasted, edit the instantiation of pasted function block.

Figure 28: Rename instances if function blocks are copied and pasted

8.9 Use of structures for axis motion parameters

The use of structures will make the project scalable and easy to manage. Some examples of structures used to manage different modes of machine operation are shown in the PLCopen Toolbox data types file. Structures for homing, cams, point to point motion etc are provided in the tool box. The programmer will benefit from using structures if the project needs to be scaled to use multiple axes.

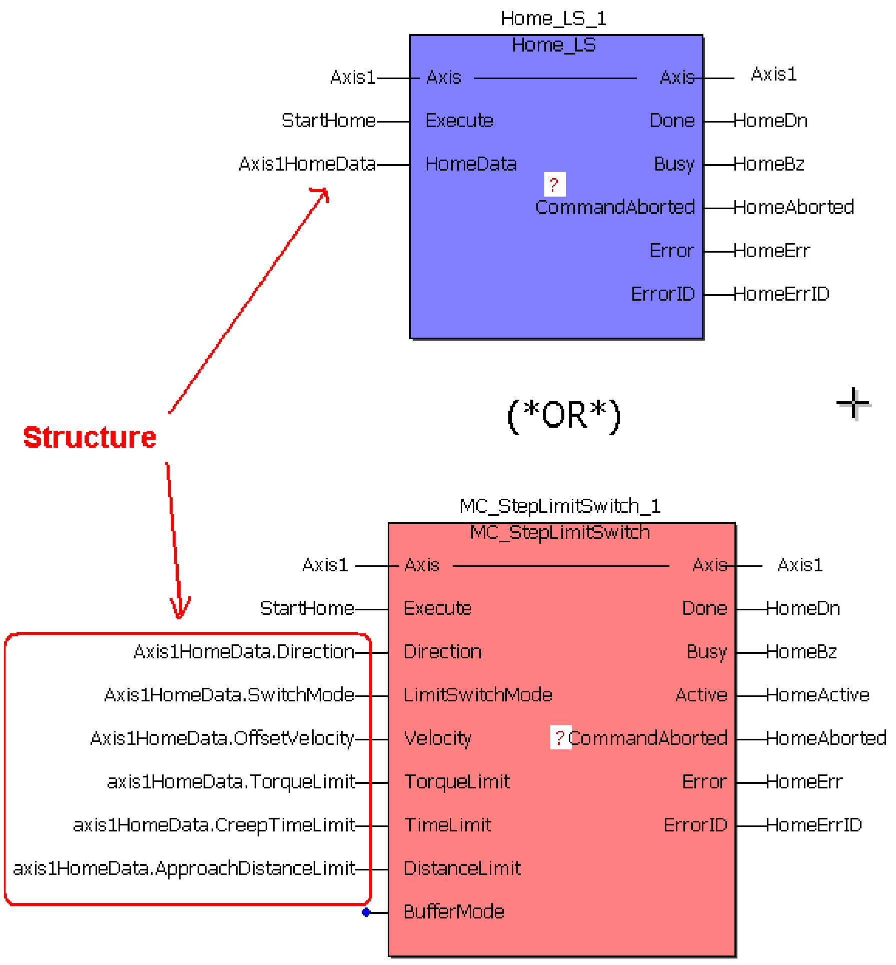


Figure 29: Use of structures for scalability

8.10 Write a coil at only one location in the program

It is recommended that a coil or a register be written only at one instance in a program. It can get difficult to track the value of the coil or register if it is written to in multiple places through out the program. This is true for MOVE blocks as well. If a variable gets values at various locations in the program, it will be difficult to track which value is being used in the controller execution at specific scans. Use cross referencing to check if there are multiple instances where a coil or a register is getting written to.

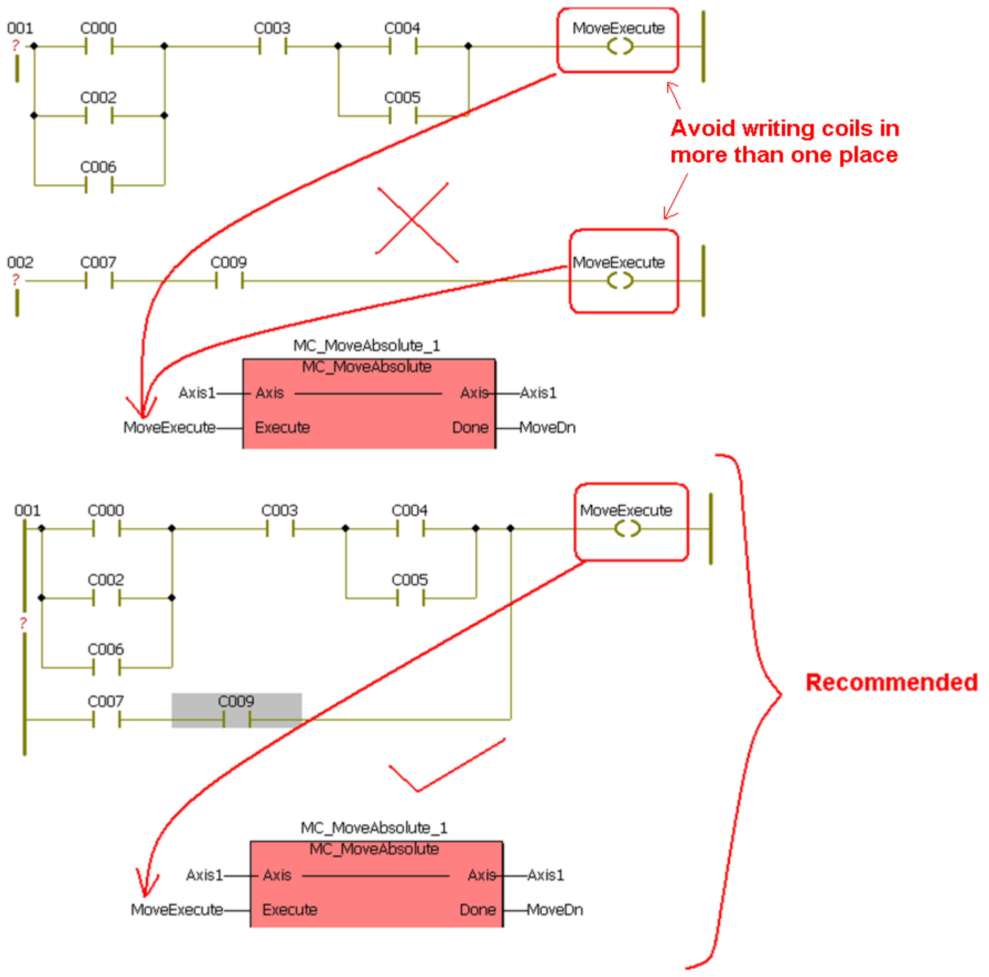


Figure 30: Write to a coil only in one place in a program

8.11 Keep the number of set-reset combinations as low as possible

It is recommended to keep set-reset combinations as low as possible. If possible, use input latching to avoid set-reset combinations. If unavoidable, use set-reset coils physically close in a POU such that they can be verified in one debug window. Do not use only a set coil. Make sure every coil that is set has a corresponding reset.

8.12 Tuning and Performance

It is possible to set sub-interpolation at the servopack level by setting controller parameters and amplifier parameters. For details on the specific parameters, refer subsection 2.1.4 of the MP2000iec Application Design Guideline:
http://partner.yaskawa.com/site/dmcontrol.nsf/(DocID)/BFAR-7Q3PY5?opendocument&login
(You must be logged in to view this document, as it is secure. If you are not logged in and you click the link, you will be asked to login first. Once logged in the file will appear.)

9 Code Inspirationals and FAQs

9.1 Implementing Enable logic for function blocks

The following figure illustrates the recommended technique to implement logic based bit operations. In this example, a Greater Than (GT) function block is enabled based on logic.

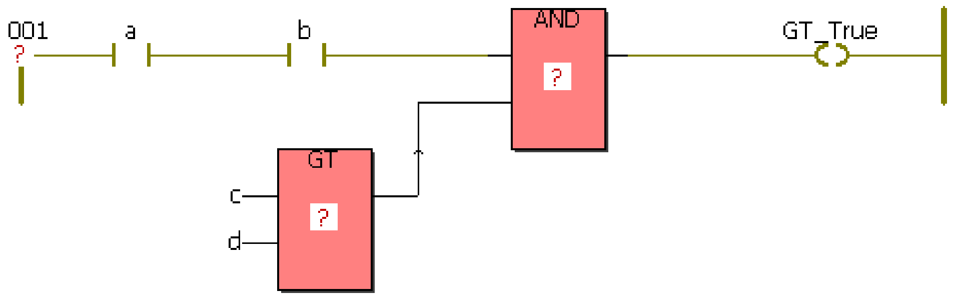
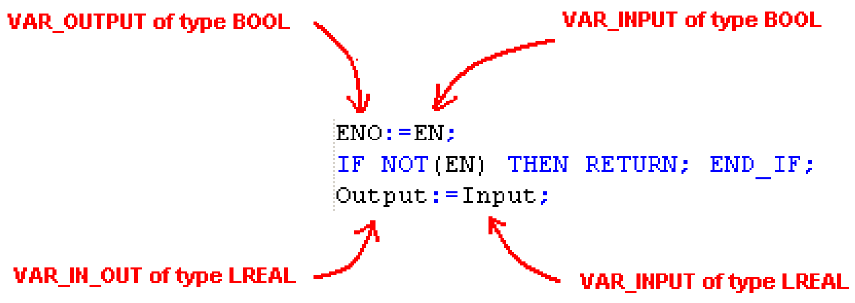


Figure 31: Boolean operations using enable logic

The following code is recommended for non bit data type operations which require enable logic. The first example is the code for an LREAL MOVE function block.



```
ENO:=EN;
IF NOT(EN) THEN RETURN; END_IF;
Output:=Input;
```

Figure 32: Floating point operations using enable logic

The LREAL MOVE function block will appear as shown below.

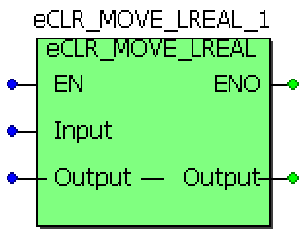
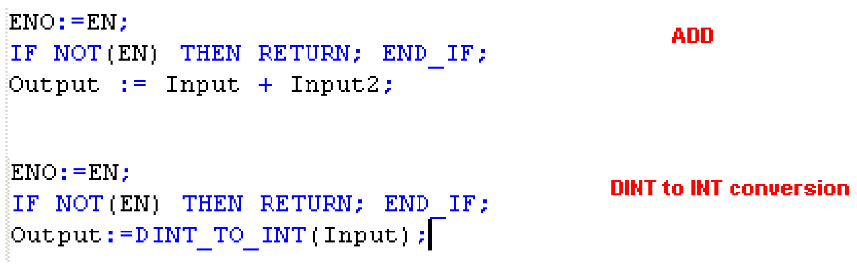


Figure 33: LREAL move with enable logic

It should be noted that an exclusive function block will have to be made for each data type to implement logically enabled MOVE. Examples of adding and data type conversion are shown below.



```
ENO:=EN;
IF NOT(EN) THEN RETURN; END_IF;
Output := Input + Input2;
```

ADD

```
ENO:=EN;
IF NOT(EN) THEN RETURN; END_IF;
Output:=DINT_TO_INT(Input);
```

DINT to INT conversion

Figure33: Code for INT add and DINT to INT conversion with enable logic

9.2 Are there limits on POUs or variables in a project?

Yes there are. 2000 POUs and 15000 global variables per project. Please search ‘Project limits for ProConOS targets’ in PLC help.

9.3 What is the effect of using different power rails for ladder rungs?

The effect of different power rail is shown in the example shown below.

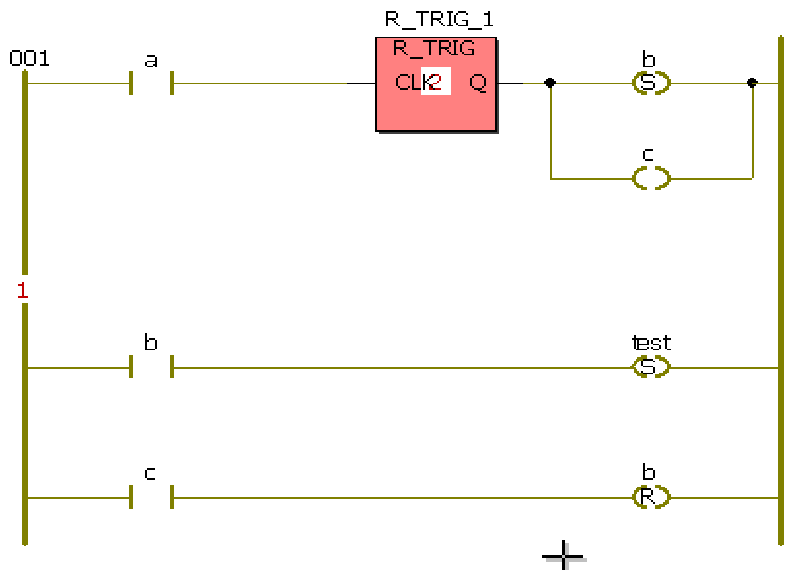


Figure 34: Coils on the same power rung

The coils b and c which go high in scan 1, act as contacts only in scan 2 if they are used in the same power rail as shown from the plot below. Note that there is only one power rail in the above logic.

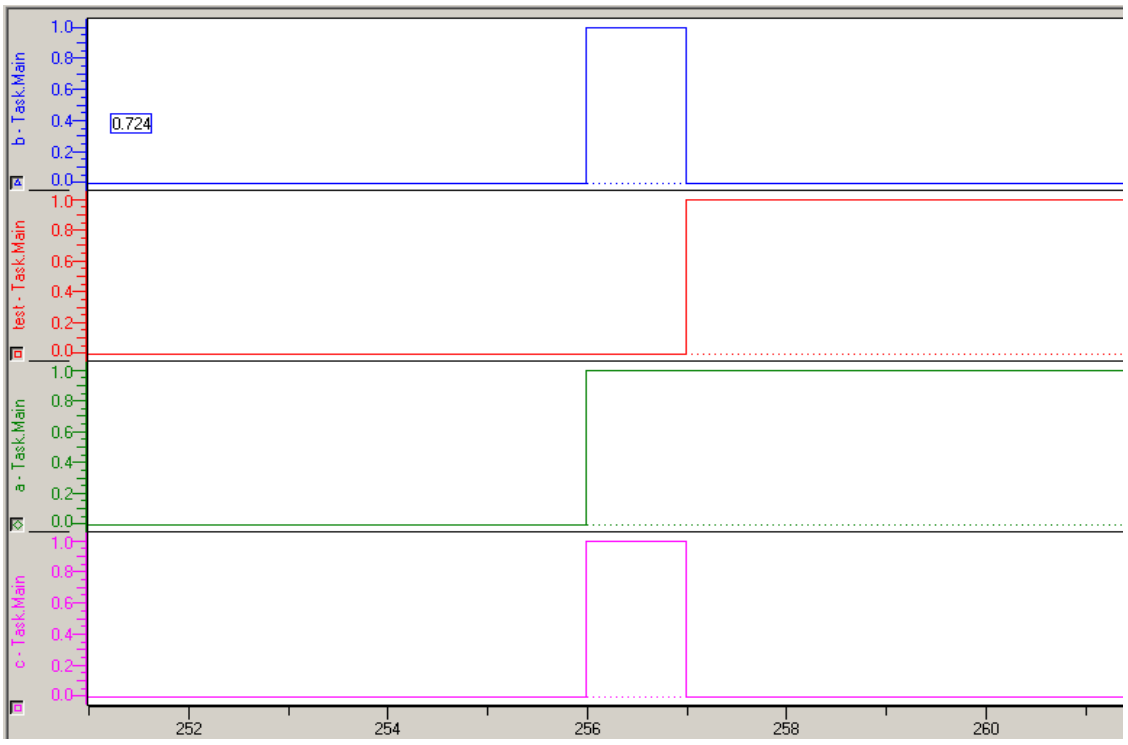


Figure 35: Results of coils on same power rung

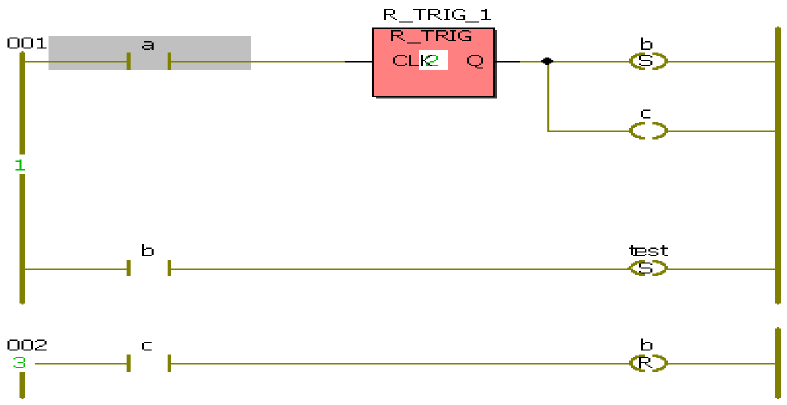


Figure 36: Coils on separate power rungs

Note that there are two power rails in this above logic. In the above scenario, coil c which goes high in scan 1 acts as a contact in scan 1 to reset b in scan 1 (last rung). Coil b is used as a contact in the same power rail. So, b would get a chance to be a high contact only in the next scan (scan 2). Since b is reset by c in scan 1, b never becomes high in scan 1 or 2. As b never becomes high, test never goes high either.

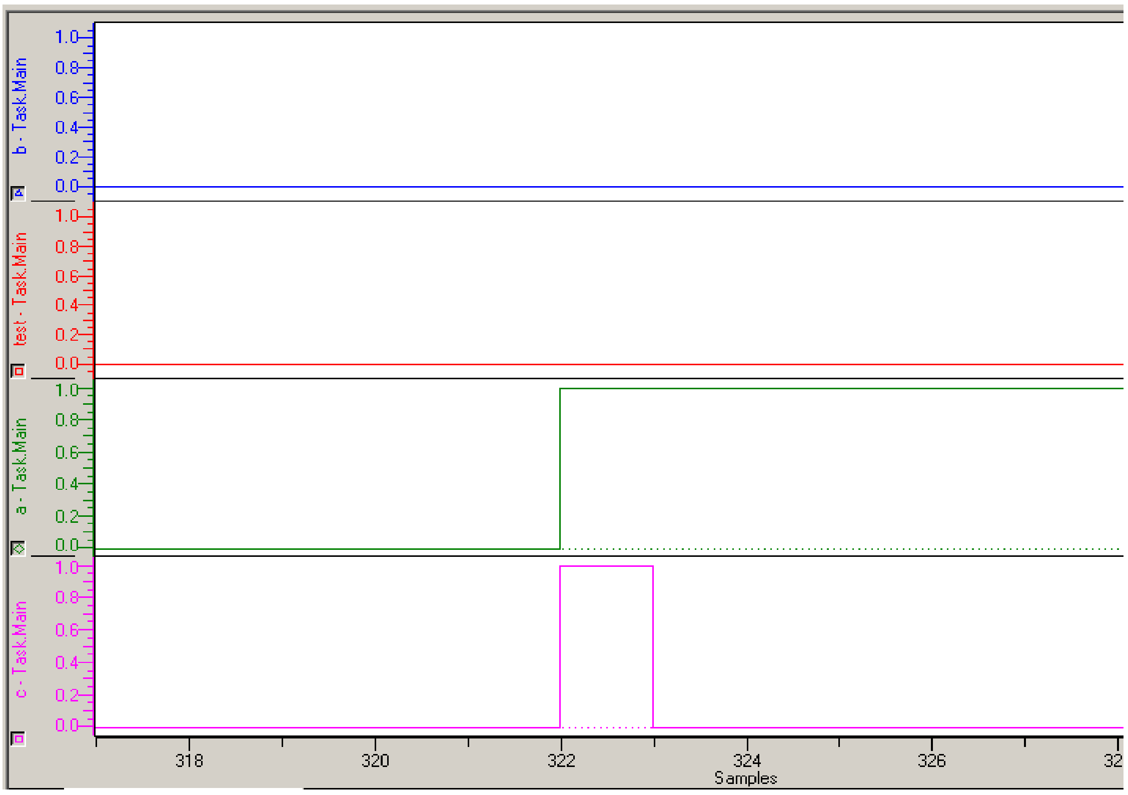


Figure 37: Results of coils on separate power rungs

9.4 How does RETURN work in MotionWorksIEC programming

A return object in a ladder rung in a POU will make the scan control to exit from the particular POU and go to the next POU in the task. For example in a project where the POU order of execution is as shown in the following figure, POU ‘first’ gets executed first and ‘second’ gets executed second.

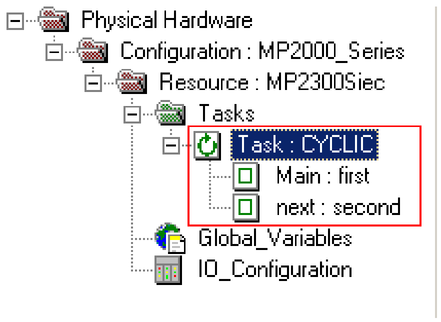


Figure 38: Sequence of execution

A return encountered in ‘first’ will make the scan ignore the ladder rungs and logic that is waiting to be scanned in ‘first’ and go to ‘second’. This is shown using an example below. POU ‘first’ has the following logic. (All variables are of type BOOL)

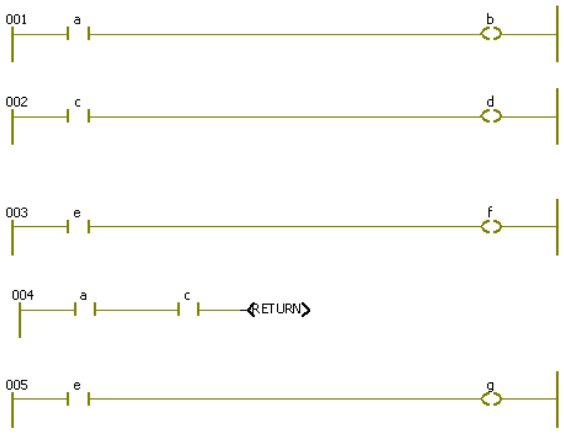


Figure 39: Return in logic in POU ‘first’

If ‘a’ and ‘c’ are TRUE, ‘g’ will not be TRUE even if ‘e’ is true because the return interrupts the scan of POU ‘first’ and transfers the scan to ‘second whose logic is as follows.

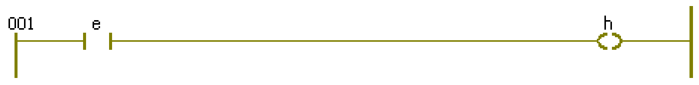


Figure 40: Logic in POU ‘second’

Variable ‘h’ in POU ‘second’ will become TRUE since ‘e’ (a global variable) is TRUE.

The result of running the project is shown below.

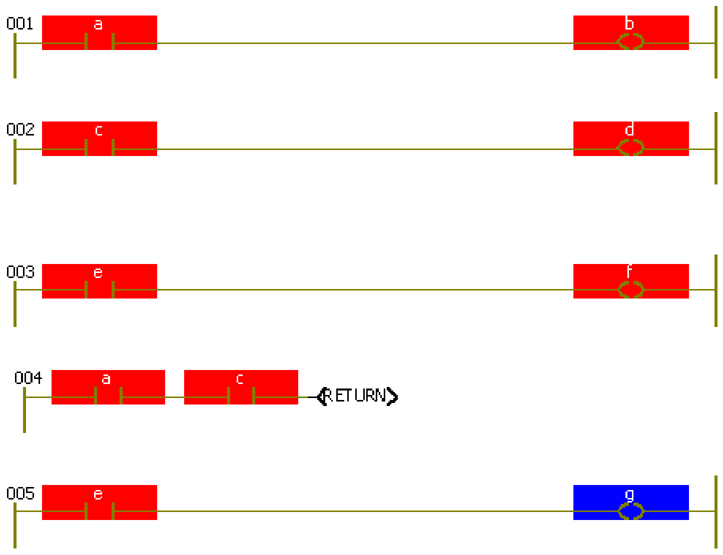


Figure 41: POU ‘first’ in debug mode

It can be seen that ‘g’ did not get triggered because the scan jumped to the next POU ‘second’ after line 4.

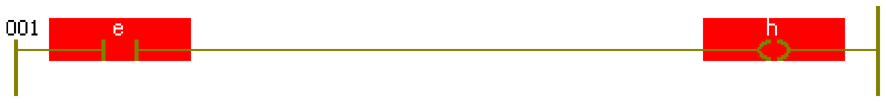


Figure 42: POU ‘second’ in debug mode

The sequence of bits is shown in the following figure.

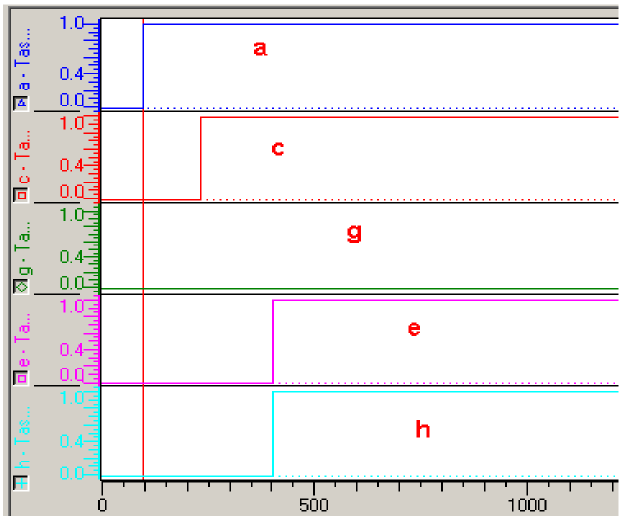
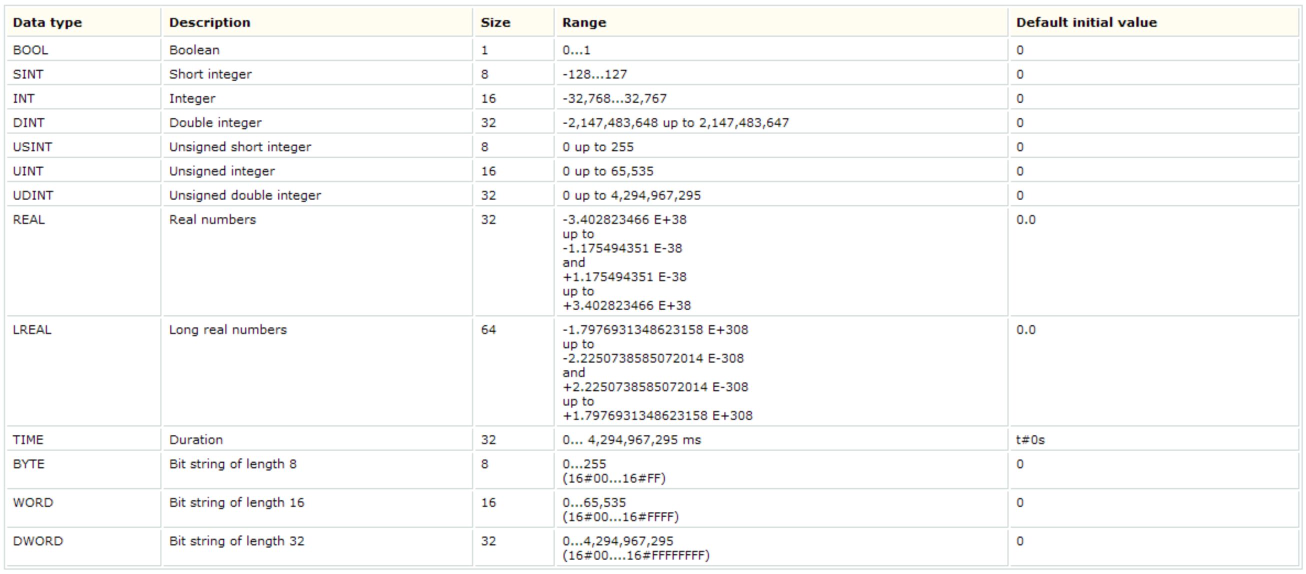


Figure 43: Results of RETURN

9.5 Rollover in MotionWorks IEC

Integers defined in MotionWorks IEC automatically rollover
The rollover limit is dependent on the data type of the variable being used. For example, a variable defined as an Integer (INT: 16 bit) in MotionWorks IEC has an operating range of -32768 to 32767. Other limits can be seen in the table provided below

Table 4: Elementary data types and limits



Data type	Description	Size	Range	Default initial value
BOOL	Boolean	1	0...1	0
SINT	Short integer	8	-128...127	0
INT	Integer	16	-32,768...32,767	0
DINT	Double integer	32	-2,147,483,648 up to 2,147,483,647	0
USINT	Unsigned short integer	8	0 up to 255	0
UINT	Unsigned integer	16	0 up to 65,535	0
UDINT	Unsigned double integer	32	0 up to 4,294,967,295	0
REAL	Real numbers	32	-3.402823466 E+38 up to -1.175494351 E-38 and +1.175494351 E-38 up to +3.402823466 E+38	0.0
LREAL	Long real numbers	64	-1.7976931348623158 E+308 up to -2.2250738585072014 E-308 and +2.2250738585072014 E-308 up to +1.7976931348623158 E+308	0.0
TIME	Duration	32	0... 4,294,967,295 ms	t#0s
BYTE	Bit string of length 8	8	0...255 (16#00...16#FF)	0
WORD	Bit string of length 16	16	0...65,535 (16#00...16#FFFF)	0
DWORD	Bit string of length 32	32	0...4,294,967,295 (16#00....16#FFFFFFFF)	0

Real numbers do not rollover
A variable defined as an LREAL (64 bit) in MotionWorks IEC has a maximum operating range of -1.7E308 to 1.7E308. The variable will have to be reset before these limits are reached. If not, the value of the variable beyond these limits goes to – Inf or + Inf depending on how the count was progressing. Realizing these limits on LREAL variables that command position should not be of any practical concern to users because the MPiec controller series uses double precision floating point format for position definition and calculations.

Consider an application configured with a feed constant of 6 mm/rev. If the axis runs at 6000 rpm (peak speed) starting from position 0.0, the maximum distance it will travel without losing precision of 1 mm is 999,999,999,999,999 mm. It will take the axis 52849 years to reach that position running at peak speed. In short it will take the axis at least 528 years to lose precision by 0.01 mm.

9.6 Can variable descriptions be included in the editor window

The description option can be selected by going to Extras>Options> ToolTips as shown below:

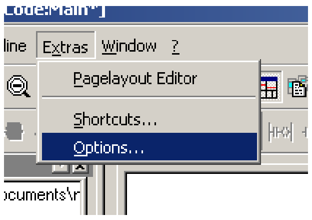
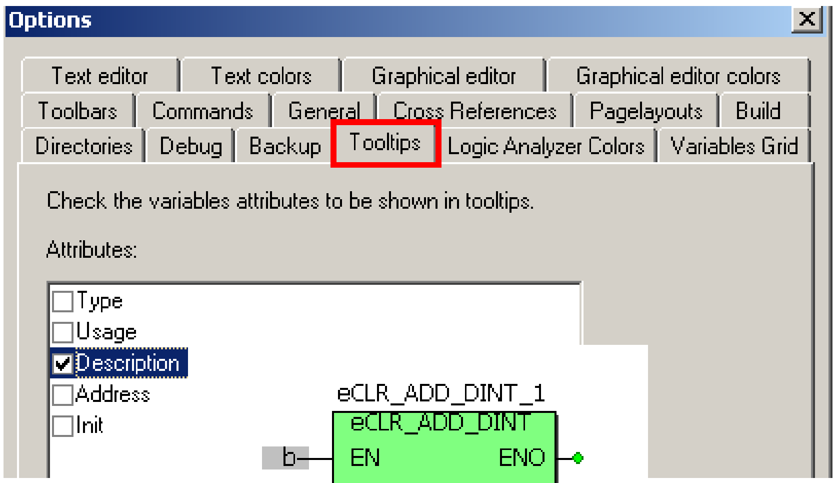


Figure 44: Editor Options



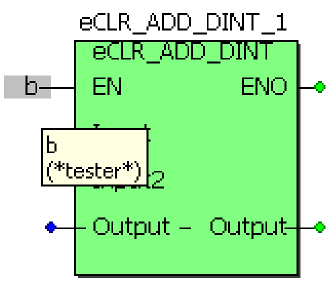
Check the variables attributes to be shown in tooltips.

Attributes:

- [ ] Type
- [ ] Usage
- [✓] Description
- [ ] Address
- [ ] Init

Fig ow

The description will not be displayed ɜ tool tip is hovered around the variable in question as shown below.

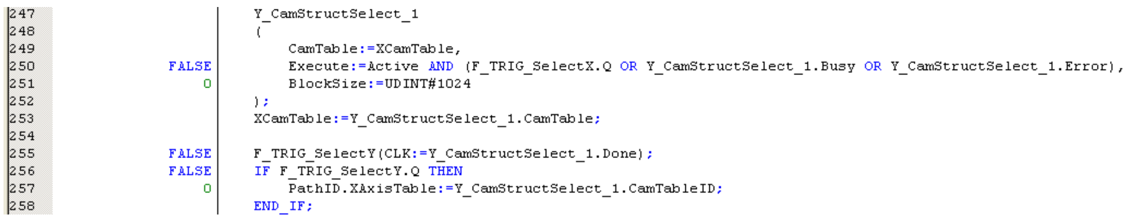


9.7 Best practices in structured text programming

Care should be exercised if a function block is enabled inside an IF ELSE structure. The state transition may never be scanned if the logic does not sweep through the IF ELSE structure.

Care should be exercised while implementing any one shot trigger based logic inside an IF-ELSE loop in structured text programming.

Put each input on its own line, so more debug information will be available.



```
      | Y_CamStructSelect_1
      | (
      |     CamTable:=XCamTable,
FALSE |     Execute:=Active AND (F_TRIG_SelectX.Q OR Y_CamStructSelect_1.Busy OR Y_CamStructSelect_1.Error),
    0 |     BlockSize:=UDINT#1024
      | );
      | XCamTable:=Y_CamStructSelect_1.CamTable;
      |
FALSE | F_TRIG_SelectY(CLK:=Y_CamStructSelect_1.Done);
FALSE | IF F_TRIG_SelectY.Q THEN
    0 |     PathID.XAxisTable:=Y_CamStructSelect_1.CamTableID;
      | END_IF;
```

Figure 46: Structured text practices

9.8 Best practices in Sequential Function Chart (SFC) programming

Use of the ‘Step.X’ bit in action instances is recommended for SFC programming. This will ensure that the action is not held in a state of limbo if thes step is not active. Use of the Step.X bit (‘FwdStep.X’ in this example) will ensure that when FwdStep stops being active, the contact FwdStep.X in the action block will be low and will prevent that particular rung from being stuck in one state. If another variable were being used in place of FwdStep.X, there is a possibility of that variable being in a stuck state if it did not change state while FwdStep was active.

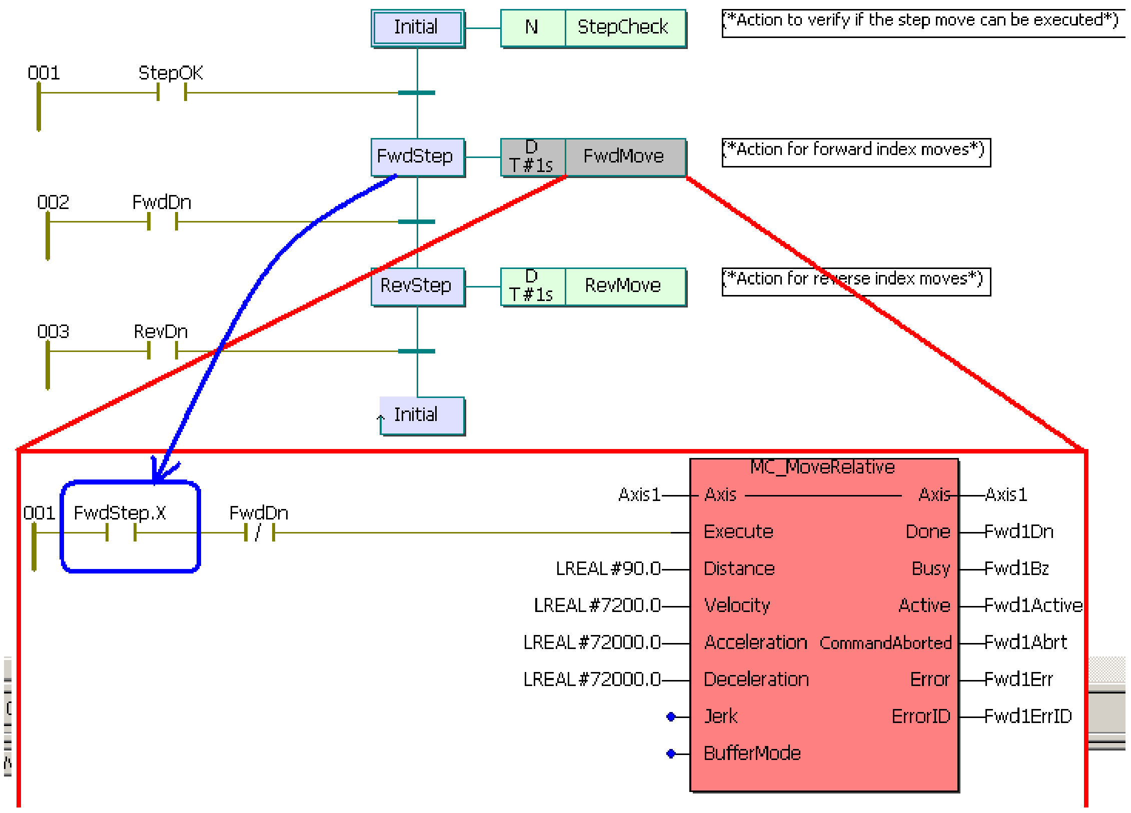


Figure 47: SFC practices

9.9 Best practices in Ladder Diagram (LD) programming

1) Please follow the following practice when coding parallel rungs in logic when there is a function block on at least one rung.

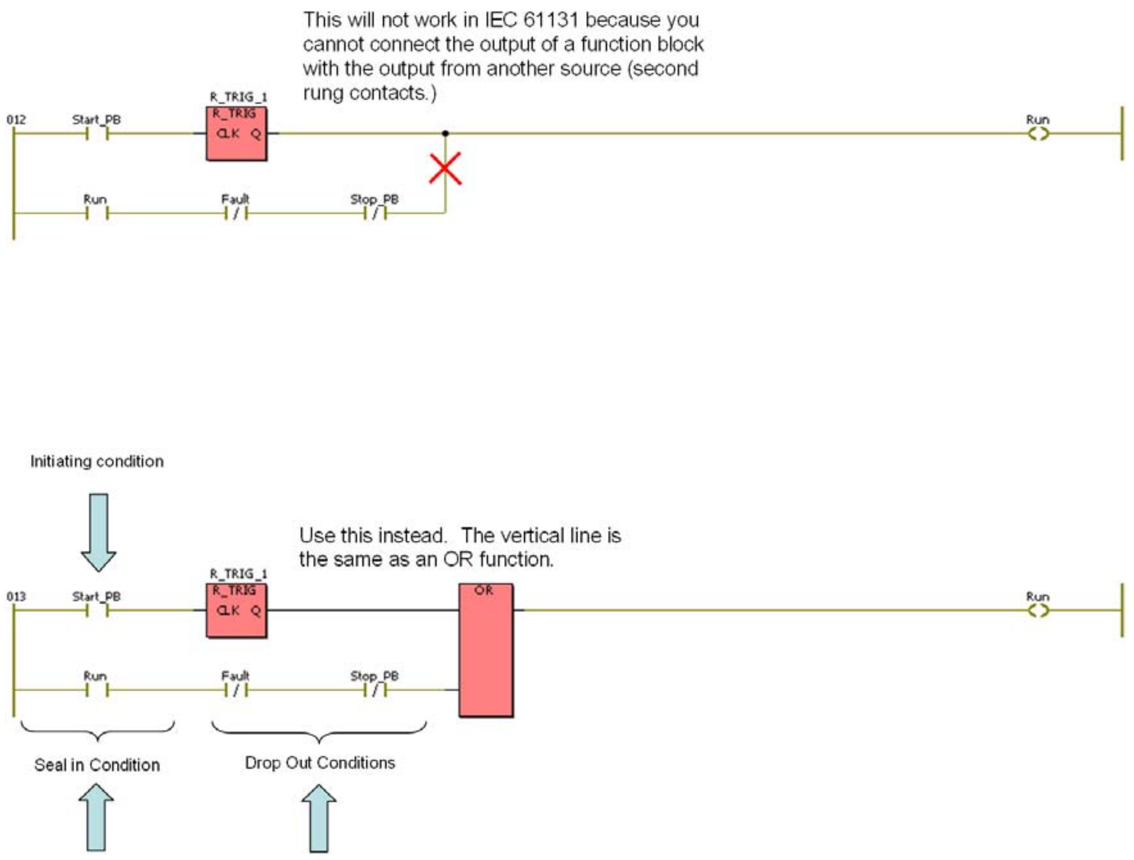
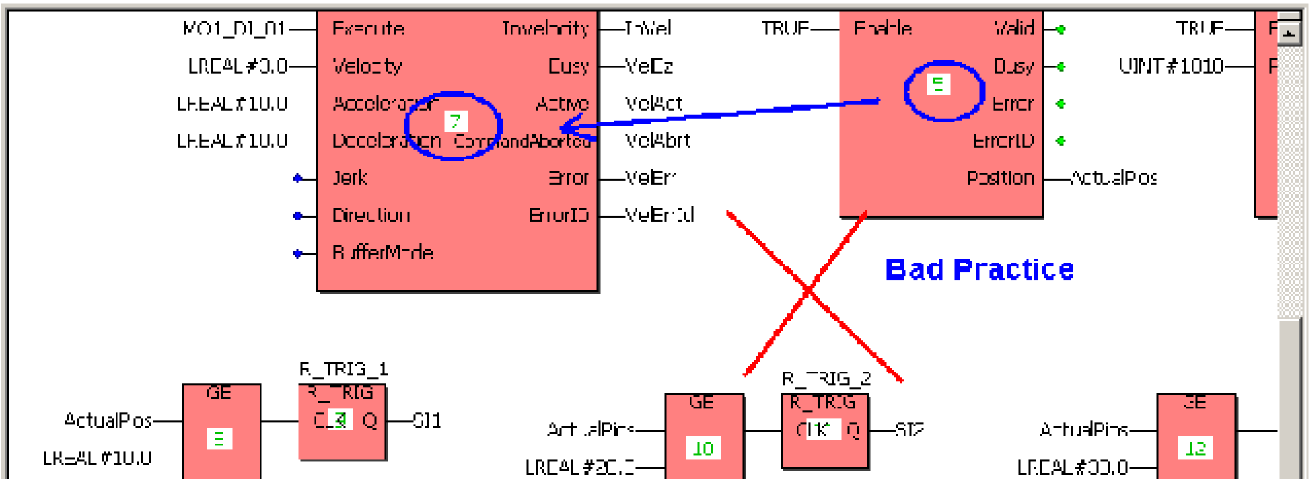
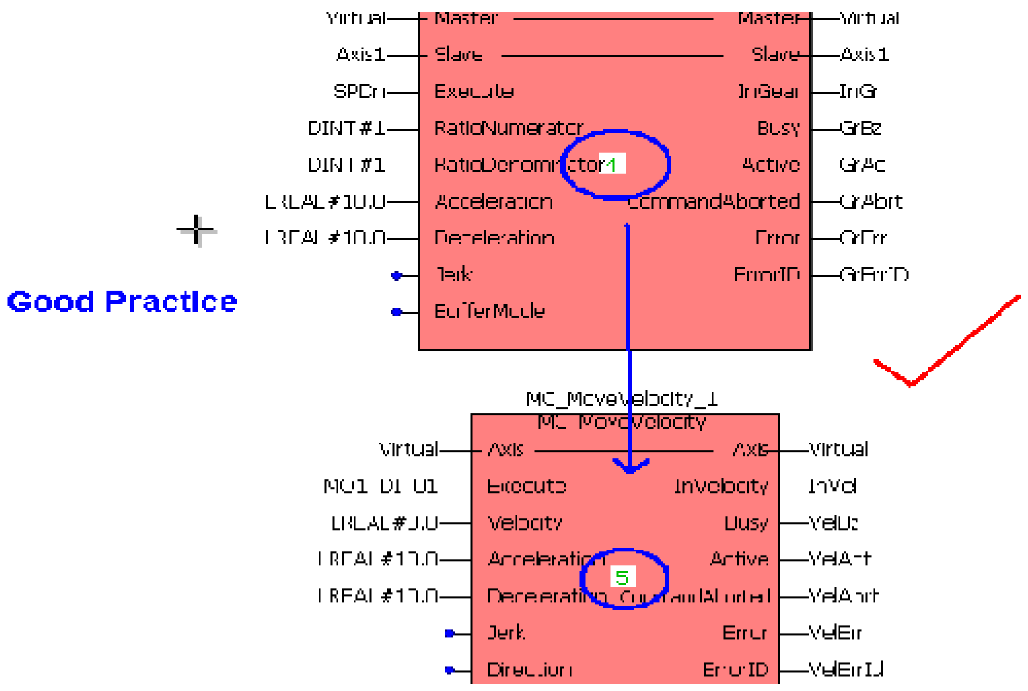

This will not work in IEC 61131 because you cannot connect the output of a function block with the output from another source (second rung contacts.)
012
Start_PB
R_TRIG_1
R_TRIG
CLK Q
Run
Run
Fault
Stop_PB
Initiating condition
Use this instead. The vertical line is the same as an OR function.
013
Start_PB
R_TRIG_1
R_TRIG
CLK Q
OR
Run
Run
Fault
Stop_PB
Seal in Condition
Drop Out Conditions

Figure 48: LD practices

2) Try to write code in a top – down format such that it is readable without having to scroll (on the monitor) from side to side. Code is easier to trouble shoot this way. This also gives the developer a better feel for execution sequence.



Bad Practice
Execute
Velocity
Acceleration
Deceleration
Jerk
Direction
BufferMode
InVelocity
Busy
Active
CommandAborted
Error
ErrorID
Enable
Valid
Position
ActualPos
R_TRIG_1
R_TRIG_2
GE



Good Practice
Master
Slave
Execute
RatioNumerator
RatioDenominator
Acceleration
Deceleration
Jerk
BufferMode
InGear
Busy
Active
CommandAborted
Error
ErrorID
MC_MoveVelocity_1
MC_MoveVelocity
Axis
Virtual
Velocity
InVelocity