



## Application Note

---

### Creating PLCopen Compliant Function Blocks in IEC 61131

---

Subject: Application Note	Product: MPiec Series Controllers	Doc#: AN.MWIEC.01
Title: Creating PLCopen Compliant Function Blocks in IEC 61131		

Introduction .....	3
Benefits .....	3
Getting Started .....	3
Function Block Models .....	4
Input Behavior .....	6
Output Behavior .....	6
Execute Function Block Model.....	7
Trigger Statements .....	8
RETURN statement .....	9
Setting the iActive Flag .....	10
Initialization .....	11
Main Code Body .....	13
Motion Blocks.....	14
Error Processing .....	16
CommandAborted Output.....	18
Busy Output .....	19
Done Output .....	20
Enable Function Block Model.....	22
Trigger Statements .....	23
RETURN statement .....	23
iActive Flag .....	23
Initialization .....	24
Main Code Body .....	24
Motion Function Blocks.....	24
Error Processing .....	24
Valid Output .....	25
Execute / Enable Model Variants .....	26
Variant #1: .....	26
Variant #2: .....	26
Recommended Interlocks.....	28
Summary.....	29
Appendix A: Logic Analyzer traces.....	30

Subject: Application Note	Product: MPiec Series Controllers	Doc#: AN.MWIEC.01
Title: Creating PLCopen Compliant Function Blocks in IEC 61131		

## Introduction

PLCopen is an organization that creates programming specifications for motion control in the automation industry. Manufacturers, OEMs, and End Users who are members of the PLCopen organization pave the way for standardized programming of motion control applications. Some users may not be familiar with the standards, and assume that manufacturers of automation products alone provide products which conform, but there's more an application programmer can do to maximize the benefits offered in the PLCopen specification. This document provides details on extending the PLCopen concept to other functions required by an application. This application note focuses on how to build upon the standards by describing code templates which have proven successful, and often refers to function blocks in Yaskawa's Toolboxes at [www.yaskawa.com/iectb](http://www.yaskawa.com/iectb).

Visit [http://www.plcopen.org/pages/tc2\\_motion\\_control/](http://www.plcopen.org/pages/tc2_motion_control/) for the complete PLCopen specification.

## Benefits

One of the best outcomes of the PLCopen specification is the definition provided for function block input and outputs. This provides a clear and concise shell as a starting point when considering the type of application level function to be created. Two main function block categories are specified: the **Execute** model and the **Enable** model.

By strictly following a few key features of the PLCopen specification, application level function blocks can provide a high degree of robustness, usability and predictability. The behavior described makes it very easy to incorporate and debug user specific functions in an application. **Errors** and **ErrorIDs** can be elevated to the calling functions. Interlocks are easier to create. Linking activities becomes easier. This is the focus of this Application Note.

## Getting Started

- 1) Decide upon the function block inputs and outputs before writing any code. Draw the function on a piece of paper first. This step will help you determine the necessary data required from an implementation perspective. It helps to imagine how the finished function block would be used by the application code.
- 2) Depending on the type of functionality to be created, consider the ideal situation in which the function block would operate. Will it require placement in a high speed cyclic task to capture data in real time? Will it also contain some intensive data processing? If so, there may be an advantage to split these two activities into separate function blocks, so that one can be executed in a high priority task, and the other executed in a lower priority task.

## Function Block Models

There are only two basic function block models as described below. These functions are shown with the minimum inputs and outputs. Notice that **Execute** pairs with **Done**, and **Enable** pairs with **Valid**. This aids the visual organization of the code structure in FBD format when contacts and coils are connected to the function block.



Figure 1: Execute and Enable function blocks in their most simple form.

Variants on these two models will be described in detail in the following sections. All implementations will include additional inputs and outputs for an actual application as shown in blue below.

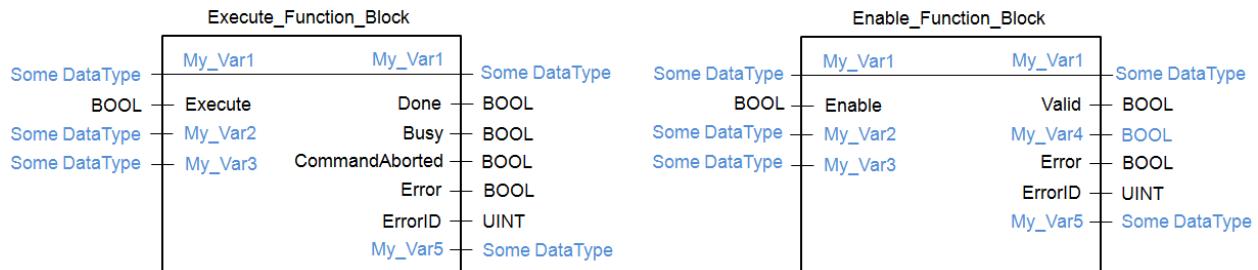


Figure 2: Execute and Enable function blocks with additional I/O.

Subject: Application Note	Product: MPiec Series Controllers	Doc#: AN.MWIEC.01
Title: Creating PLCopen Compliant Function Blocks in IEC 61131		

The main differences between the Execute and Enable models are show in the following table.

Table 1: Execute and Enable Model behavior comparison

Function Block Type	Required Behavior	Example
Execute	The action is temporary; it has a finite beginning and ending. A sequence of activities.	Homing to a limit switch, C channel, and then making an offset move.
	The action takes only one scan.	Setting a parameter.
Enable	The function can complete its job in one scan, and repeats this action each scan.	Reading a parameter.
	The action must run indefinitely.	Monitoring for new product registration latches to store them into a circular buffer.

Subject: Application Note	Product: MPiec Series Controllers	Doc#: AN.MWIEC.01
Title: Creating PLCopen Compliant Function Blocks in IEC 61131		

## Input Behavior

Execute Model	Inputs other than <b>Execute</b> are only to be read upon the rising edge of <b>Execute</b> . The concept here is that the function uses input values present when execution was initiated. This requires copying VAR_INPUTs to a VAR in the Initialize section.
Enable Model	A function block using the <b>Enable</b> model is expected to read the VAR_INPUTs every scan and if necessary, act upon changed values.

VAR\_IN\_OUT data is passed to the function block by reference, meaning that only a pointer to the original data location is copied into the function block, eliminating longer data copy times for larger data types. Because a VAR\_IN\_OUT references the original data, not a copy, changes made to the variable inside the function block can be seen immediately by the application code as well versa. Program accordingly, and use the **Done** or **Valid** output to indicate when data referenced as VAR\_IN\_OUT is valid.

## Output Behavior

A brief review of the PLCopen specification follows.

- 1) Only one PLCopen status output (**Done**, **Busy**, **CommandAborted**, **Error**) can be TRUE at one time.
- 2) Execute Function Blocks have a finite execute life which will end in one of the three ways:
  - a. **Done** – Function block has completed its task successfully.
  - b. **CommandAborted** – Another action took control away from the function block, so this function did not complete its task successfully.
  - c. **Error** – There was a problem with the VAR\_INPUTs or with other internal processing that prevented the function block from completing its task.

Once one of these three PLCopen status outputs are set, the function block can never change the outputs until the **Execute** or **Enable** input goes low and the function block restarts again.

- 3) When the **Execute** input goes low and the function is no longer **Busy**, VAR\_OUTPUTs must be set to zero. This also applies to non Boolean outputs that are part of a specific implementation. If the **Execute** input goes low while the function block is still **Busy**, one of the three outputs (**Done**, **CommandAborted**, or **Error**) will pulse for one scan when the function completes.
- 4) When the **Enable** input goes low, VAR\_OUTPUTs must be set to zero.

## Execute Function Block Model

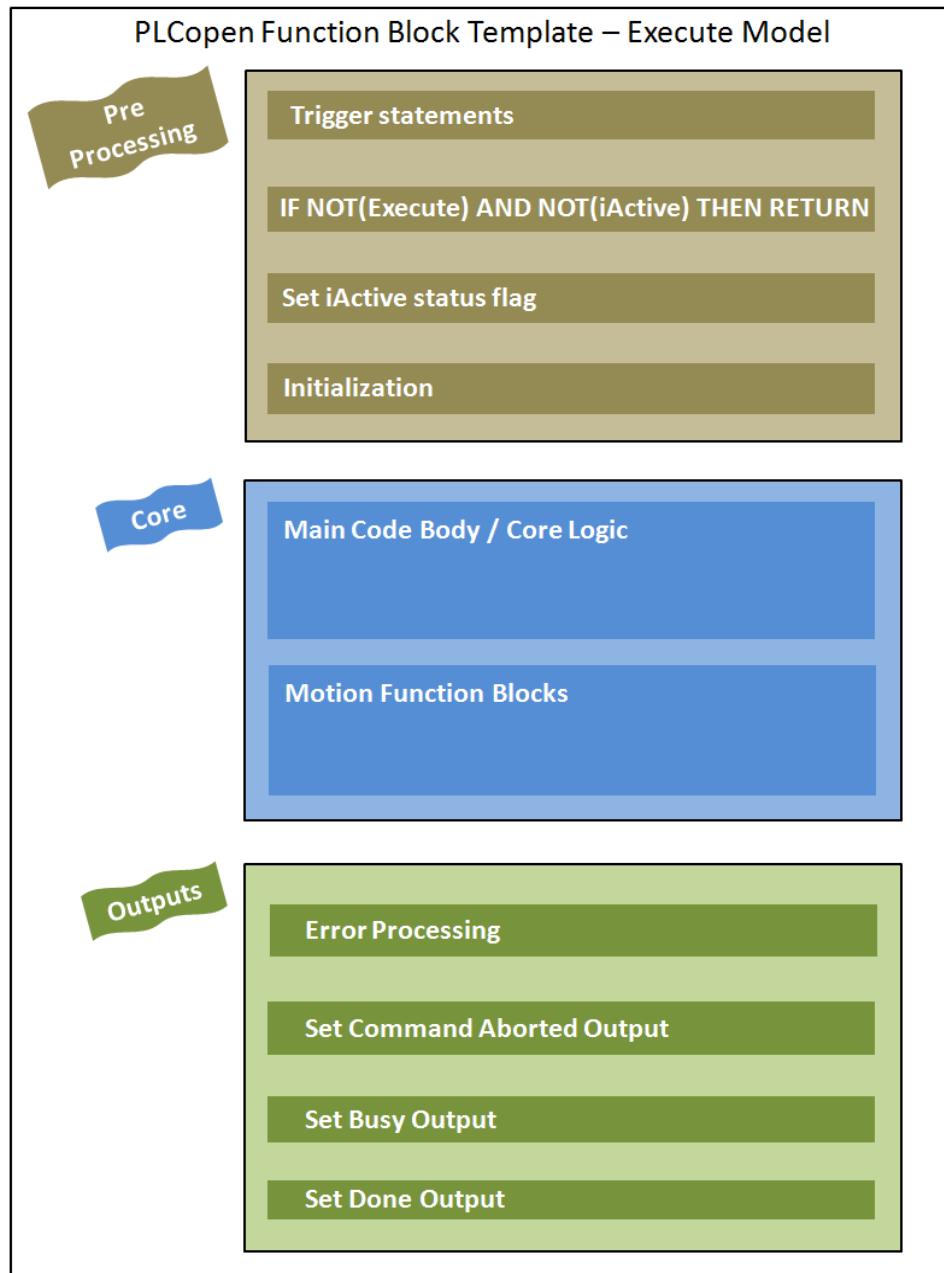


Figure 3: Execute Template in simplified form

Figure 3 is a representation of each part of the function block code. The order of operations is the same in Ladder Diagram (LD) or Structured Text (ST) programming. Code order is very important for ensuring consistent output behavior, which is essential for proper sequencing and error trapping, especially when function blocks are triggered using an R\_TRIG one shot.

What follows is a detailed explanation of the code sections depicted in Figure 3 on page 7.

### Trigger Statements

This is where most if not all R\_TRIG and F\_TRIG function blocks should be placed, especially in ST function blocks. Typically one of the trigger statements detects the rising edge of the **Execute** or **Enable** input to be used to run the initialization code on the first scan. By inserting trigger statements at the top of the POU, they are sure to run every scan (no IF / END\_IF conditions to inadvertently prevent execution.)

### LD Format

It's less important to put Trigger statements at the top of function block written in LD, and most examples are actually initialization code. This is because a LD POU does not have the potential for conditional execution, such as the IF statement in ST. Triggers can be used throughout a LD function block without concern.

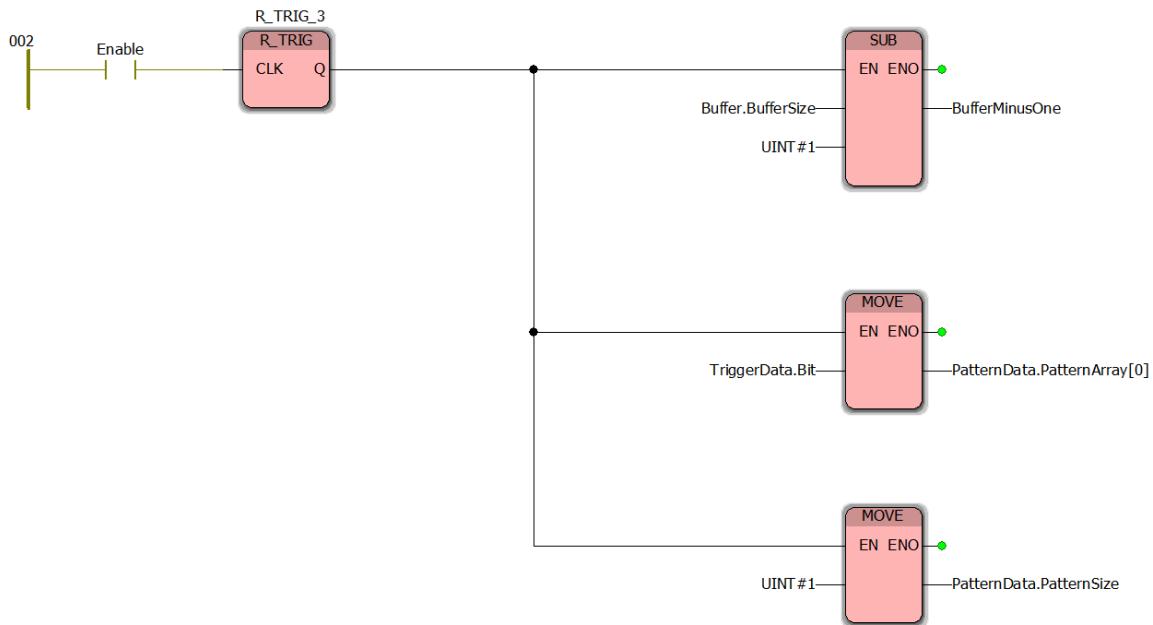


Figure 4: Execute Model – LD trigger statement example

### **ST Format**

```

1  (******)
2  (****** Event Handling Section *****)
3  (******)
4  R_TRIG_Execute(CLK:=Execute);
5  R_TRIG_ChangeVelocity(CLK:=(Velocity <> PreviousVelocity) AND NOT(R_TRIG_ChangeVelocity.Q) AND NOT(R_TRIG_Execute.Q));
6  R_TRIG_NewSegment(CLK:=(CurrSegment <> ActiveSegment) AND NOT(R_TRIG_NewSegment.Q));
7  R_TRIG_Complete(CLK:=AllSequencesDone);

```

Figure 5: Execute Model - ST trigger statements example.

### **RETURN statement**

This line is purely for efficiency, but if not used carefully it can lead to problems that are hard to debug. A RETURN causes the function block to exit without running any of the instructions below the RETURN statement. Imagine a project with over one hundred function blocks. Most likely, only a few blocks may be active at any one time while all others RETURN. This saves processing time because the controller will not have to evaluate the many IF conditions which may follow, presumably skipping most of the code because the function is not to be executed.

Note that when the RETURN takes place, the debug information shown for the remainder of the POU contains old values from the last time it ran.

### **LD Format**

In this example “Adjusting” is a variable connected to the **Busy** output of a PLCopen motion function block.

(\*If Enable and all busy outputs are off, the function exits\*)

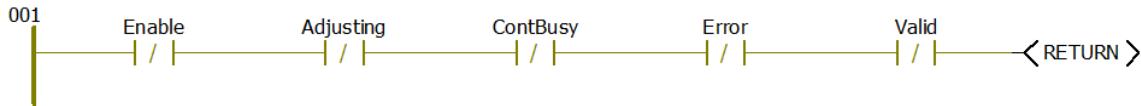


Figure 6: Execute Model – LD example RETURN statement.

### **ST Format**

```

6  (* This line causes the function block to exit if the execute, and all outputs are off. (For efficiency.) *)
7  (* IMPORTANT, be sure to include CommandAborted in the interlock logic if a motion block will be used. *)
8  IF NOT(Execute) AND NOT(Active) AND NOT(Busy) AND NOT(Done) AND NOT(Error) THEN RETURN; END_IF;

```

Figure 7: Execute Model - ST example RETURN statement.

### Setting the iActive Flag

'Internal Active' is a Boolean flag to keep the function block code running even if the **Execute** input goes false before the function block is **Done**. The **Execute** input can be pulsed, and the code will continue to execute, and the outputs remain valid until one scan after the final outcome of the function occurs. The iActive flag serves as a good way to control the execution of sub functions and other logic within your function block. It is named 'internal Active' to differentiate from the other PLCopen output **Active** which is found on PLCopen function blocks that control motion.

There is an underlying theme at work here, and it pertains to the execution behavior of a real time system such as a PLC. Basically speaking, don't let code become dormant if it's in the middle of executing. If this situation occurs, it will likely cause unexpected behavior. Pulsed or event actions such as R\_TRIG or functions with an **Execute** input work by comparing a value on a previous scan to the current scan. Execution must not be interrupted by the RETURN statement or other logic change which would prevent a function from working normally the next time it's required to execute.

### LD Format

In this example from the CamBlend function block, three separate inputs act as the **Execute** input based on the mode required. Normally just the standard **Execute** would be included instead of three.

#### (\*iActive Interlock\*)

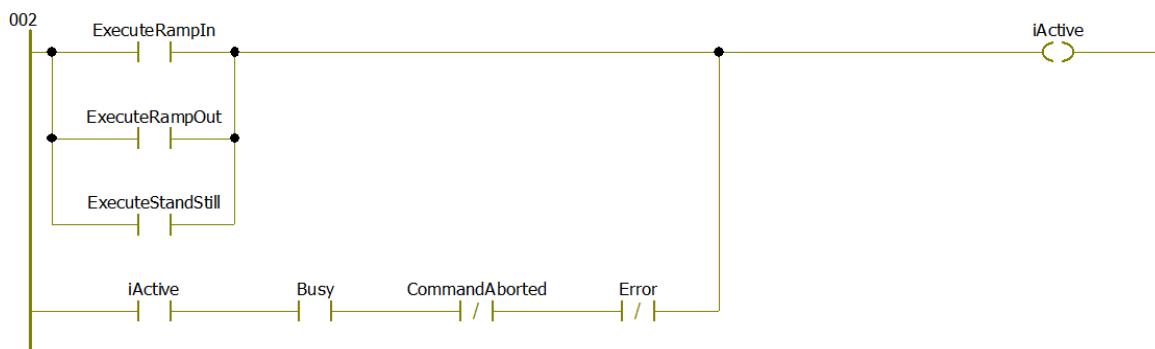


Figure 8: Execute Model - LD example for setting the iActive flag.

### ST Format

```
|14  iActive:= Execute OR (iActive AND NOT(Done) AND NOT(CommandAborted) AND NOT(Error));
```

Figure 9: Execute Model - ST example for setting the iActive flag.

## Initialization

The PLCopen specification states that VAR\_INPUTs are only read upon the rising edge of the **Execute** input. This section is very important because it's the location where the VAR\_INPUTs passed into the function block are validated and transferred to a working copy. Create a naming convention so that copied VAR\_INPUT values are obvious. If the VAR\_INPUT name is Acceleration, use iAcceleration in the Main Code Body for example.

**Tip:** During code development, make a habit of adding new variables (VAR) into the Initialization section immediately, and provide an initial value. It will save debug time by reducing the likelihood that your function block will not execute properly more than once due to variables being left in an unexpected state from a previous run. This is one of the most important things you can do to boost the reliability and consistency of function block execution.

## LD Format

It's more difficult to show a good example of initialization code in LD format. Initialization is associated with algorithms and similar processes which are typically written in ST. LD code tends to be self initializing by nature. The contact and coil approach will both set and clear a bit scan every scan. Yaskawa recommends avoiding Set and Reset coils if possible. These

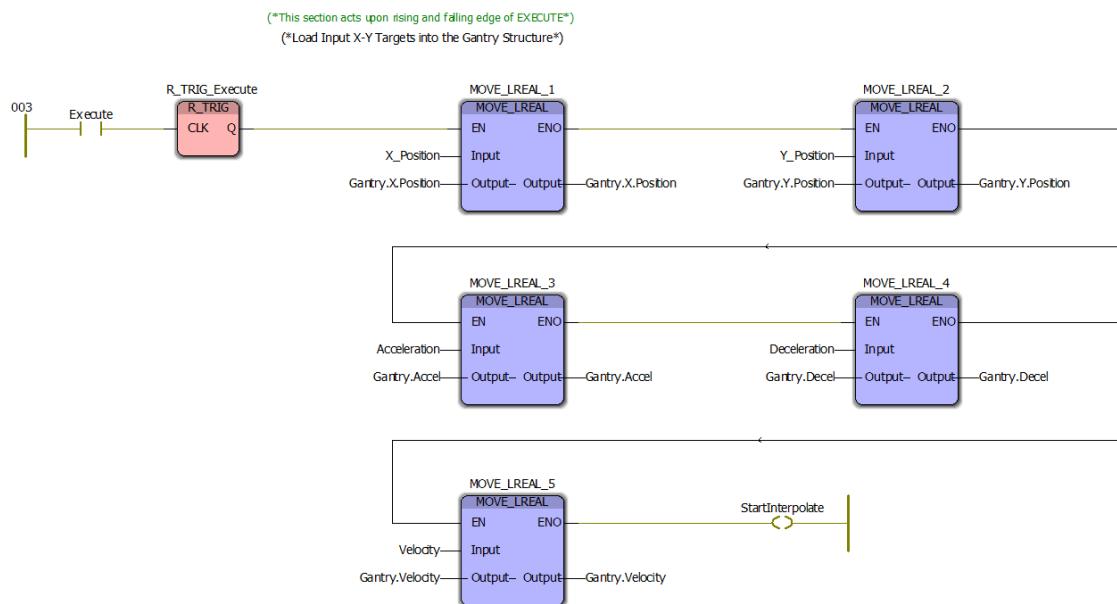


Figure 10: LD example for initializing variables.

coil types tend to cause more bugs, but if they are used, performing a reset coil for each set coil would be another great example of initialization code in LD format.

### **ST Format**

Notice the use of R\_TRIG\_Execute.Q, which is the output of the R\_TRIG function used in the initialization section.

```

18   (* **** Initialization Section ****)
19   (* ****
20   (* ****
21 IF R_TRIG_Execute.Q THEN
22   (* Sequence Initialization *)
23   (* Sequences relate to the number of individual moves that the vector
24 CurrSequence:=INT#1;
25 PrevSequence:=INT#0;
26 AllSequencesDone:=FALSE;
27
28   (* Segment Initialization *)
29 CurrSegment:=INT#1;
30 PrevSegment:=INT#0;
31 MoveSegment:=INT#0;
32 IF SegmentData.LastSegment = INT#1 THEN
33   NextSegment:=INT#1;
34 ELSE
35   NextSegment:=CurrSegment + INT#1;
36 END_IF;
37
38 PreviousVelocity:=Velocity;          (* Initialaizing PrevVelocity
39
40 XAxisConfigured:=(Gantry.X.Ref.AxisNum <> UINT#0);
41 YAxisConfigured:=(Gantry.Y.Ref.AxisNum <> UINT#0);
42 ZAxisConfigured:=(Gantry.Z.Ref.AxisNum <> UINT#0);
43 TangentAxisConfigured:=(Gantry.Tangent.Ref.AxisNum <> UINT#0);
44 TEngageData.SlaveAbsolute:=TRUE;
45 ConfiguredAxesInSync:=FALSE;
46 END_IF;

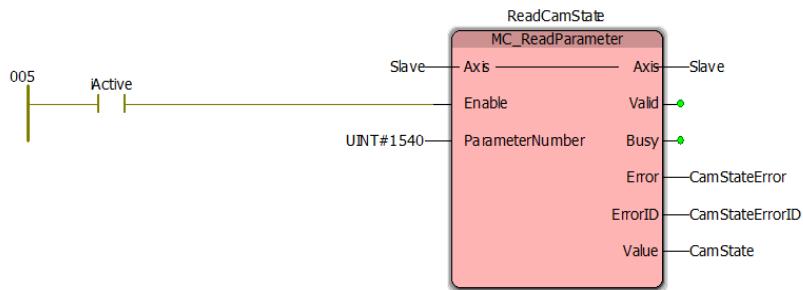
```

Figure 11: Execute Model - Example ST code for initializing variables.

## Main Code Body

This is where the core activity of the function block resides. The Main Code Body logic always includes reference to the iActive flag. This section may contain a series of other function blocks which execute in sequence, or a wide variety of other actions based on the purpose of the function you are designing. The best way to determine the appropriate code structure to implement is to refer to the many examples in Yaskawa's Toolboxes, available at [www.yaskawa.com/iectb](http://www.yaskawa.com/iectb). Recommended LD references include the Home\_LS\_Pulse function in the PLCopen Toolbox.

## LD Format



(\*On the first time a CamShift completes (falling edge) and the slave is not yet engaged (CamState=0), set the Startup output to engage.\*)

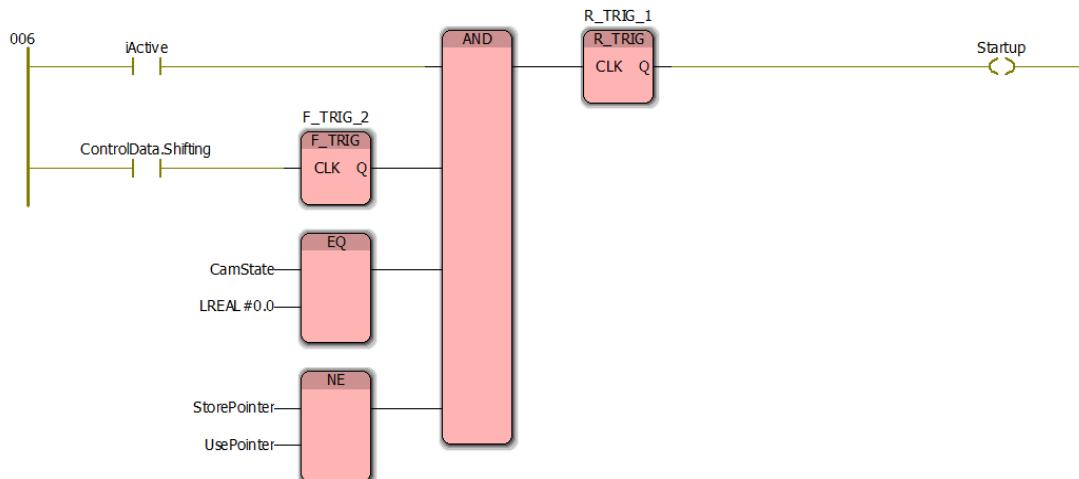


Figure 12: Execute Model – A partial example of LD code of the Main Code Body, which is controlled by the iActive flag

### ST Format

```

26  **** Main Operation Section ****
27  IF Enable AND NOT(Error) THEN
28
29      Mini:=INT#0;
30      Maxi:=SegmentData.LastSegment;
31
32      (* Make sure the VectorPosition is withing the boundaries of the total vector path *)
33      IF VectorPosition < SegmentData.Segment[0].VectorDistance THEN
34          VectorPosition:=SegmentData.Segment[0].VectorDistance;
35      ELSE
36          IF VectorPosition > SegmentData.Segment[LastSegment].VectorDistance THEN
37              VectorPosition:=SegmentData.Segment[LastSegment].VectorDistance;
38          END_IF;
39      END_IF;
40
41
42      WHILE ((Maxi - Mini) > INT#1) DO
43          SearchLoc:=(Maxi + Mini) / INT#2;
44          IF VectorPosition > SegmentData.Segment[SearchLoc].VectorDistance THEN
45              Mini:=(SearchLoc);
46          ELSIF VectorPosition < SegmentData.Segment[SearchLoc].VectorDistance THEN
47              Maxi:=(SearchLoc);
48          ELSIF VectorPosition = SegmentData.Segment[SearchLoc].VectorDistance THEN
49              Mini:=SearchLoc;
50              Maxi:=SearchLoc;
51          END_IF;
52          Counter:=Counter+INT#1;
53      END_WHILE;
54
55      IF (Maxi = Mini) THEN
56          ActiveSegment:= SegmentData.Segment[Mini].Segment;
57          OutputFlags := SegmentData.Segment[Mini].OutputFlags;
58      ELSE
59          ActiveSegment:= SegmentData.Segment[Maxi].Segment;
60          OutputFlags := SegmentData.Segment[Maxi].OutputFlags;
61      END_IF;
62
63  ELSE
64      (* Clear FB outputs when function is disabled *)
65      ActiveSegment := INT#0;
66      OutputFlags := DWORD#0;
67  END_IF;

```

Figure 13: Execute Model - ST code example show the Main Code Body.

### Motion Blocks

This section is only necessary if the function block being designed provides motion, especially if written in ST. The strategy here is to provide the ability to execute (scan) the motion function blocks with their **Execute** input set to FALSE before the RETURN statement is executed. This will ensure that motion functions used within your function block can execute correctly the next time they are required.

### **Frozen code**

One of main challenges is to avoid leaving a function block frozen (its code no longer being executed when it or any of its sub function blocks are still *Busy*.) In LD, this is easy to avoid because typically all ladder rungs are executed each scan. In ST, this quickly becomes a bigger challenge because inserting functions under IF statements can cause major problems if the logic in the IF statement changes before nested function blocks are *Done*.

### **ST Solutions**

Solve the logic required to determine if executing a particular sub function is necessary within nested IF statements, but leave the sub functions *outside* of all IF statements so they are evaluated every scan. This will result in the same logic but avoid the possibility of leaving sub function blocks in a frozen state. A portion of a function block implementing this strategy is shown below. The variables XAxisConfigured, YAxisConfigured and SetPosDone are solved inside IF conditions in the Main Code Body.

```

86  Y_CamIn_X.Master:=          Gantry.Virtual.Ref;
87  Y_CamIn_X.Slave:=          Gantry.X.Ref;
88  Y_CamIn_X.Execute:=        iActive AND ((XAxisConfigured AND SetPosDone) OR XAxisError);
89  Y_CamIn_X.CamTableID:=    PathID.XAxisTable;
90  Y_CamIn_X.EngagePosition:= LREAL#0.0;
91  Y_CamIn_X.EngageWindow:=  LREAL#0.0;
92  Y_CamIn_X.EngageData:=   EngageData;
93  Y_CamIn_X.Periodic:=    TRUE;
94  Y_CamIn_X();               (* Execute PLCopen Function Block *)
95  Gantry.Virtual.Ref:=     Y_CamIn_X.Master;
96  Gantry.X.Ref:=           Y_CamIn_X.Slave;
97  XInSync:=                Y_CamIn_X.InSync;
98  XaxisBusy:=              Y_CamIn_X.Busy;
99  XaxisError:=             Y_CamIn_X.Error;
100 XaxisErrorID:=           Y_CamIn_X.ErrorID;
101
102
103 Y_CamIn_Y.Master:=          Gantry.Virtual.Ref;
104 Y_CamIn_Y.Slave:=          Gantry.Y.Ref;
105 Y_CamIn_Y.Execute:=        iActive AND ((YAxisConfigured AND SetPosDone) OR YAxisError);
106 Y_CamIn_Y.CamTableID:=    PathID.YAxisTable;
107 Y_CamIn_Y.EngagePosition:= LREAL#0.0;
108 Y_CamIn_Y.EngageWindow:=  LREAL#0.0;
109 Y_CamIn_Y.EngageData:=   EngageData;
110 Y_CamIn_Y.Periodic:=    TRUE;
111 Y_CamIn_Y();               (* Execute PLCopen Function Block *)
112 Gantry.Virtual.Ref:=     Y_CamIn_Y.Master;
113 Gantry.Y.Ref:=           Y_CamIn_Y.Slave;
114 YInSync:=                 Y_CamIn_Y.InSync;
115 YaxisBusy:=               Y_CamIn_Y.Busy;
116 YaxisError:=              Y_CamIn_Y.Error;
117 YaxisErrorID:=            Y_CamIn_Y.ErrorID;

```

Figure 15: Execute Model - ST example of Motion Function Blocks on the main level (Not under an IF statement)

## Error Processing

### Latching Function Block Errors

A very important debugging feature provided by the PLCopen specification is the ability of each function block to latch (or freeze at the current state) when an **Error** occurs so that the code can be debugged to determine the cause of the **Error**. Without this simple feature, debugging transient errors would be exceeding difficult.

To integrate error trapping effectively, a referenced function block that generates an error must have its **Execute** input held high to insure that it continues to report its **ErrorID**. In the LD example shown below, notice that the contacts 'iActive' and 'SetPositionError' work together to hold the **Execute** high and lock on once an **Error** has occurred. When the **Error** has been acknowledged, the calling function drops the **Execute** input, which will start a chain reaction which will drop iActive, which will drop MC\_SetPosition's **Error** output, which will drop this function block's **Error** and **ErrorID** output.

**TIP:** There may be more than one **Error** present in the function. Because the IEC code executes from top to bottom, the last instruction to write to a variable will provide the resulting value. In the case of reporting **ErrorIDs**, you may want to prioritize the errors so that the most important or relevant ErrorID is output in the event of multiple errors. This means arranging them in increasing order. See Figure 17 and Figure 18 starting on page 17.

### LD Format

The first graphic shows an error being set by a sub function used within the function block. Notice the variable SetPositionError. The example shown in Figure 16 might exist in the Main Code Body. The example shown in Figure 17 demonstrates the Error processing code which comes below the Main Code Body. Recall the Execute Template overview on page 7 for reference.

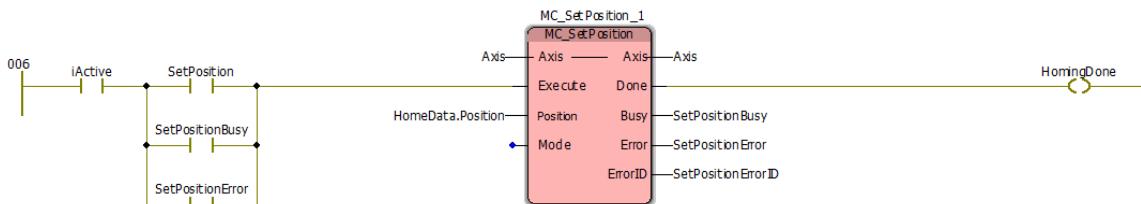


Figure 16: Execute Model – LD example showing a function block's Error output used to hold its Execute high.

Figure 17 shows the Error handing code. If simultaneous errors may occur, consider prioritizing them so the last MOVE instruction copies the ID of the most important error to the **ErrorID**.

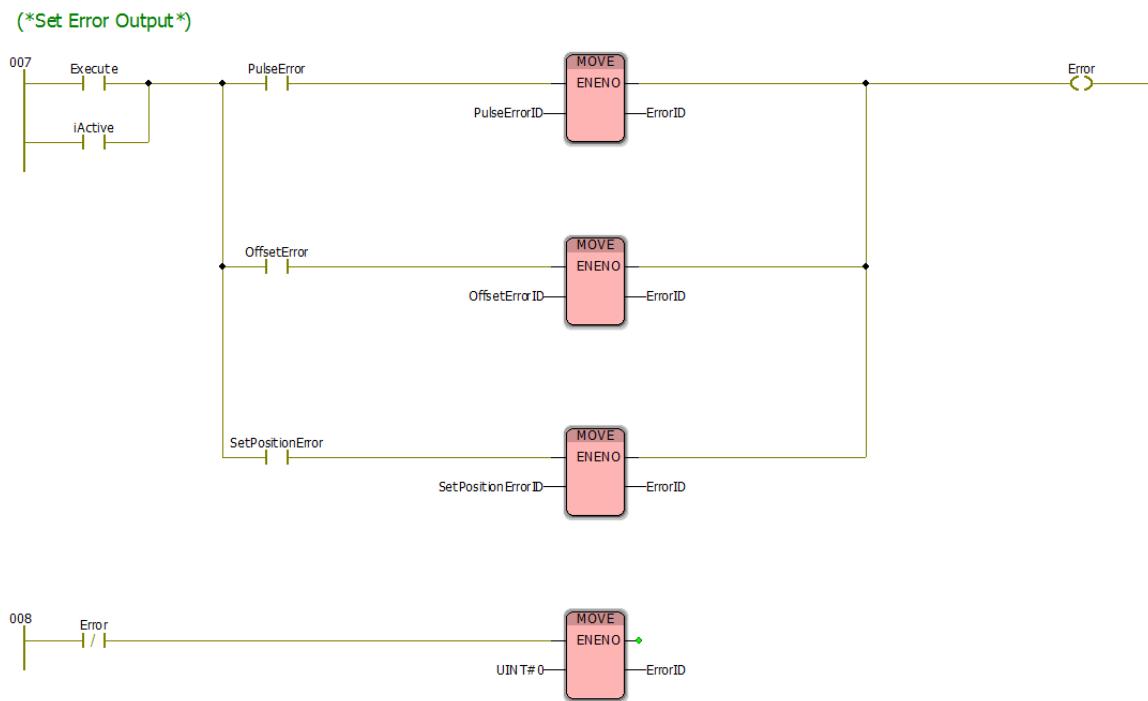


Figure 17: Execute Model - Example in LD showing the Error processing code.

### ST Format

This example shows a total of 12 possible errors being monitored. If any one of them occurs, it will lock on the **Error** output, and set the appropriate **ErrorID**. Even if the original **Error** goes away, the **Error** bit locks on in line 433, and because of the logic in lines 436 through 447, the **ErrorID** will still be set to the value that caused the initial **Error**. The more **ErrorIDs** the better to provide an adequate description of the problem that occurred to the user or machine operator. This example shows the incorporation of custom **ErrorIDs** allocated for specific errors and the propagation of errors from embedded PLCopen function blocks.

```

432 (****** Error Handling Section *****)
433 Error:=Execute AND (Error OR SegmentSizeError OR PairSizeError OR SegmentTypeError OR StandStillDurationError OR InputConditionsError OR SegmentTooShort OR
434 CircleError OR StartAngleError OR Y_CamStructSelect_X.Error OR Y_CamStructSelect_Y.Error OR Y_CamStructSelect_Z.Error OR Y_CamStructSelect_T.Error);
435 IF Error THEN
436   IF SegmentSizeError THEN ErrorID:=UINT#10038; END_IF;
437   IF PairSizeError THEN ErrorID:=UINT#10041; END_IF;
438   IF SegmentTypeError THEN ErrorID:=UINT#10054; END_IF;
439   IF SegmentTooShort THEN ErrorID:=UINT#10055; END_IF;
440   IF StandStillDurationError THEN ErrorID := UINT#10134; END_IF;
441   IF InputConditionsError THEN ErrorID := UINT#10135; END_IF;
442   IF CircleError THEN ErrorID:=UINT#10056; END_IF;
443   IF StartAngleError THEN ErrorID:=UINT#10058; END_IF;
444   IF Y_CamStructSelect_X.Error THEN ErrorID:=Y_CamStructSelect_X.ErrorID; END_IF;
445   IF Y_CamStructSelect_Y.Error THEN ErrorID:=Y_CamStructSelect_Y.ErrorID; END_IF;
446   IF Y_CamStructSelect_Z.Error THEN ErrorID:=Y_CamStructSelect_Z.ErrorID; END_IF;
447   IF Y_CamStructSelect_T.Error THEN ErrorID:=Y_CamStructSelect_T.ErrorID; END_IF;
448 ELSE
449   ErrorID:=UINT#0;
450 END_IF;

```

Figure 18: Execute Model - Example Error processing in ST.

### CommandAborted Output

This VAR\_OUTPUT is only required if encapsulating PLCopen motion function blocks that provide a **CommandAborted** output. Connect the **CommandAborted** output of all sub function blocks together in an OR format as shown below. PLCopen specification states that **Done**, **Busy**, **CommandAborted**, and **Error** must be mutually exclusive, so in case your function block is already outputting an error for some other reason, suppress the **CommandAborted** output by adding the **Error** contact to the logic as shown.

### LD Format

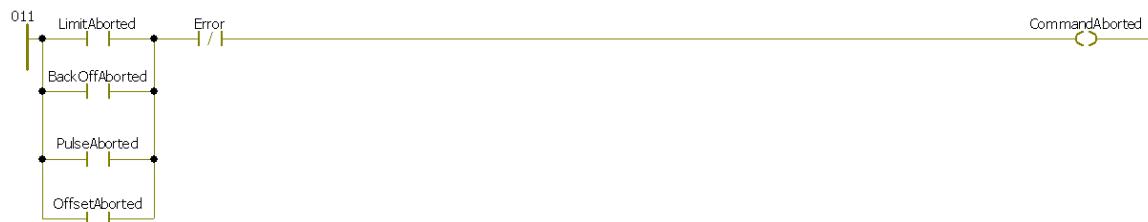


Figure 19: Execute Model - CommandAborted example in LD.

### ST Format

Sometimes **CommandAborted** is simply the same value as output from a referenced PLCopen motion function block. The following line is from the MoveRelative\_ByTime function block in the PLCopen Toolbox.

```
|43 CommandAborted:=MC_MoveRelative_1.CommandAborted AND NOT(Error);
```

Figure 20: Execute Model - ST example of setting the CommandAborted output.

Notice that the concept is the same whether writing in LD or ST. OR all **CommandAborted** outputs together with AND NOT(Error). As shown in the Execute Model graphic on page 7, the code order of **Error**, **CommandAborted**, **Busy**, and **Done** is important for the PLCopen output exclusivity concept to work correctly.

### Busy Output

Assume that your function block will take some time (multiple scans) to complete, such as homing an axis. The **Busy** output is set while your function is executing or any sub function block within your function block is Busy, thus the **Busy** outputs are ORed together. Once again, the order of operations for setting outputs is important for maintaining the PLCopen specification that only one output (**Done**, **Busy**, **CommandAborted**, or **Error**) can be on at one time. For robustness, ensure that the **Error** output has not been set just above in the error handling section. When multiple sub functions are referenced from a single function block, conditions may exist where some activities are **Busy** and others have an **Error**. Some creativity and interpretation is required to make the new function block's behavior ideal.

### Special Case

If your function block is simple and guaranteed to finish in one scan, a **Busy** output is technically not necessary, although Yaskawa recommends including the output for consistency.

### **LD Format**

This is an example from the Home\_LS\_Pulse function block in the PLCopen Toolbox. The **Busy** flags here are from the five PLCopen motion function blocks contained within.



Figure 21: Execute Model - LD example to set the Busy output.

### **ST Format**

The following example is from a function block containing four customized PLCopen function blocks.

```

459 ***** Busy Output *****
460 Busy:= (Active OR Y_CamStructSelect_X.Busy OR Y_CamStructSelect_Y.Busy OR Y_CamStructSelect_Z.Busy OR Y_CamStructSelect_T.Busy) AND NOT(Done) AND NOT(Error);
  
```

Figure 22: Execute Model - ST example to set the Busy output.

### **Done Output**

The **Done** output indicates that the function block has finished successfully and that no errors were generated. It only requires one line of code to program the **Done** output. The following graphics show code which complies with the PLCopen specification; the **Done** output will remain high as long as the **Execute** input is high. If the **Execute** input has been set low before this function completes, the **Done** output will pulse for one scan.

### **Special case**

In the LD and ST examples below, notice one includes NOT(**Error**) and the other does not. Based on the collection of sub functions used to build your function and the task they are to perform, it may be necessary to deliberately alter or suppress an output to conform to the PLCopen rule that only one output among **Busy**, **Done**, **CommandAborted**, **Error** can be on at one time. Generally, the output behavior will be more robust if they are thought of as having a hierarchy in order of importance:

- 1) Error
- 2) CommandAborted
- 3) Busy
- 4) Done

This is the same order as they are shown in the Execute function block model section starting on page 7. In summary, if any of the preceding outputs are on, the current output cannot be set on.

### **LD Format**



Figure 23: Execute Model - Example to set the Done output.

### **ST Format**

When the function block being designed will contain multiple sub actions, use another internal Boolean flag 'Complete' to indicate that all function block activities are Complete. Complete should be set in the Main Code Body as a result of the last action successfully completing. Most ST function blocks in the Yaskawa Toolboxes include this flag.

```

454 (******)
455 Done:= ((Active AND Complete) OR (Execute AND Done)) AND NOT(Error);
      Done Output *****)
  
```

Figure 24: Execute Model - ST example including the Complete flag.

## Enable Function Block Model

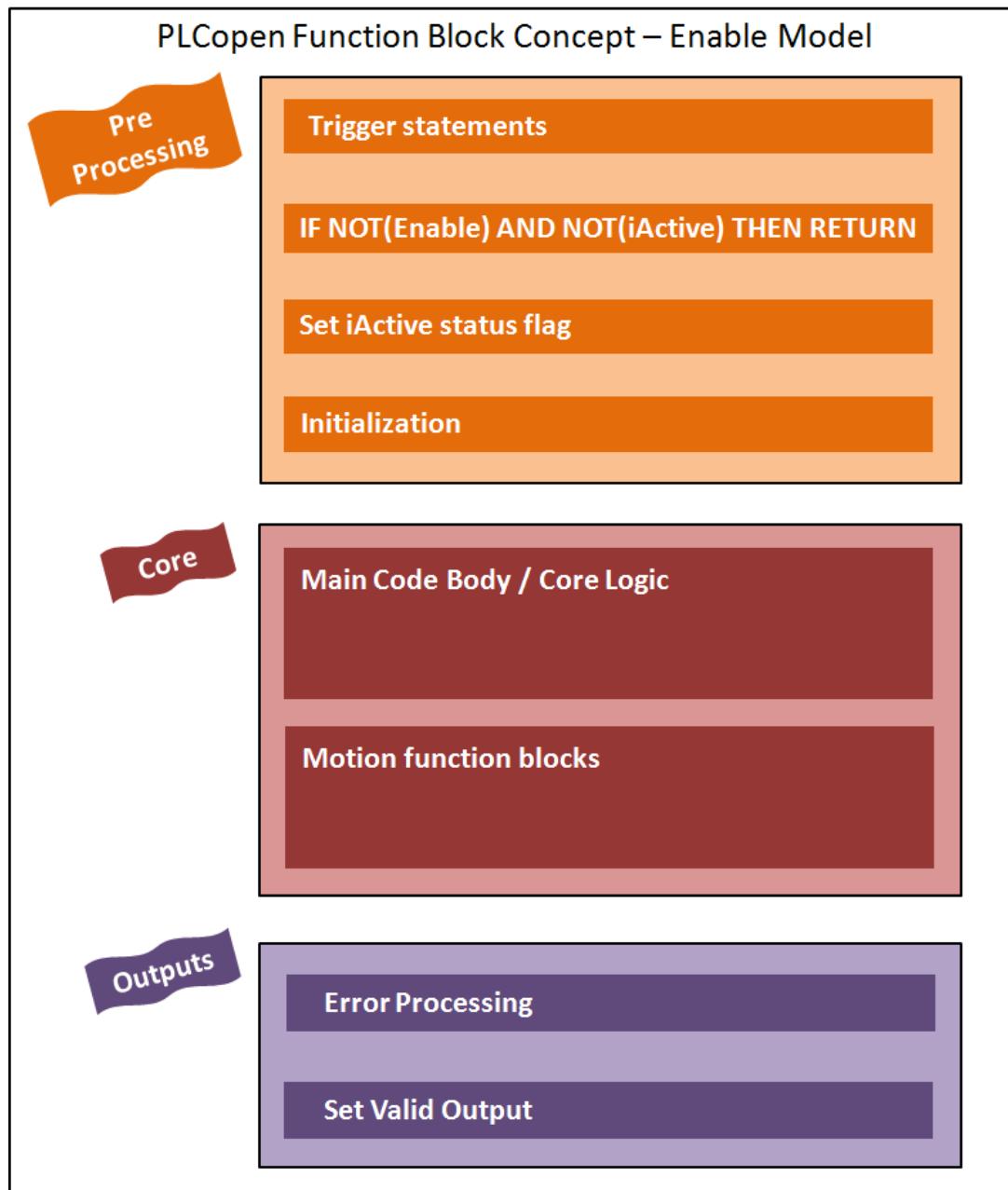


Figure 25: Enable Template in simple form.

Subject: Application Note	Product: MPiec Series Controllers	Doc#: AN.MWIEC.01
---------------------------	-----------------------------------	-------------------

Title: Creating PLCopen Compliant Function Blocks in IEC 61131
--

Many aspects of the Enable function block model are identical to Execute model, and therefore some topics are only described in the Execute Model section.

### Trigger Statements

Using R\_TRIG and F\_TRIG is the same whether the function block being created is of the Execute or Enable type. See the Trigger statements on page 8 in the Execute Model section for more details.

### RETURN statement

This line is mainly included for efficiency. It is also quite similar whether creating an Execute or Enable function block. See page 9 for more details. If a function block is not Active (dormant, and not processing anything) it makes sense to RETURN back to the calling POU instead of executing any of the code in the function block. It must be used with caution however. Using a RETURN statement improperly will cause many bugs. Only allow the RETURN to take place when all of following conditions are TRUE:

- 1) Execute or Enable = FALSE
- 2) iActive = FALSE
- 3) Busy = FALSE
- 4) Error = FALSE

By checking these conditions, it ensures that all outputs are off, and any sub function blocks are also in a dormant state.

**Tip:** Only use one RETURN per Function Block if possible. It can be confusing to debug a POU when trying to determine if code is actually being executed or if a RETURN has taken place.

### iActive Flag

The concept of 'Active' was adopted during the development of our templates as an internal way to indicate when the function block was actively processing code. This concept is necessary because no single PLCopen input or output exactly captures this condition. The **Execute** input isn't a reliable indicator because it may be pulsed by the calling code.

## Initialization

This section is the same whether creating an Execute or Enable type function block. See the Initialization section in the Execute Model section for more details. (Page 11)

## Main Code Body

The Main Code Body runs only when the **Enable** input is high and there are no **Errors**. In ST function blocks, this is typically the first IF condition of the section. In LD format, the main section does not stand out unless comments are provided.

## Motion Function Blocks

This is an optional section which is only required if the function block calls other function blocks. If the function block will be designed using ST, it's best not to include them under any IF conditions for simplicity when debugging and to avoid the trouble of coding to ensure that they are not left in a Busy state.

## Error Processing

This section is very similar whether creating an Execute or Enable type function block.

### LD Format

(\*Set Error Output\*)

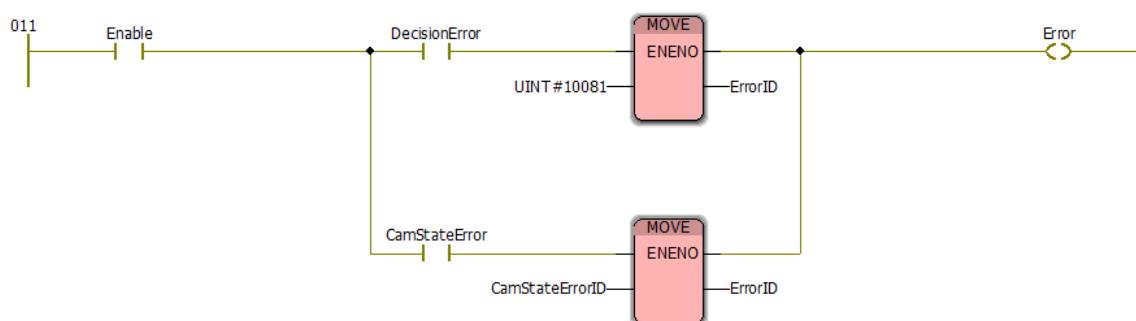


Figure 26: Enable Model - LD Error Processing example.

### ST Format

```

237  (***** Error Handling Section *****)
238  Error:=Enable AND (Error OR ModeError OR ClearError OR CamShiftError);
239  IF Error THEN
240      IF ModeError THEN ErrorID:=UINT#10082; END_IF;
241      IF Prm1511Error THEN ErrorID:=MC_ReadParameter_1511.ErrorID; END_IF;
242      IF Prm1006Error THEN ErrorID:=MC_ReadParameter_1006.ErrorID; END_IF;
243      IF Prm1512Error THEN ErrorID:=MC_ReadParameter_1512.ErrorID; END_IF;
244      IF ClearError THEN ErrorID:=Y_CamShift_Clear.ErrorID; END_IF;
245      IF CamShiftError THEN ErrorID:=Y_CamShift_1.ErrorID; END_IF;
246      IF CamShiftAbort THEN ErrorID:=UINT#7282; END_IF;
247  ELSE
248      ErrorID:=UINT#0;
249  END_IF;

```

Figure 27: Enable Model - ST Error Processing example.

### Valid Output

Setting the **Valid** output is quite simple. Only one line of code is required to operate the **Valid** Output. The outputs of the function are **Valid** when the **Enable** input is high and there are no **Errors**. According to PLCopen specification, if an error occurs, the **Enable** input must go low and high once again to re execute the function block.

### LD Format

(\*Set Valid Output\*)



Figure 28: Enable Model - LD example for setting the Valid output.

### ST Format

```

81  (***** Valid Section *****)
82  Valid:=Enable AND NOT(Error);

```

Figure 29: Enable Model - ST example for setting the Valid output.

## Execute / Enable Model Variants

Both function block models are suitable for a wide range of applications, however in practice there are many conditions where the programmer may find that neither model perfectly applies for the required behavior.

### **Variant #1:**

The function block behaves like the Enable model, but contains PLCopen function blocks with an **Execute** input (MC\_MoveVelocity and MC\_Stop). This Jog function block is an example which includes BOOL inputs for Forward and Reverse. The function block will stop jogging the axis when Forward or Reverse go low. On its own, MC\_MoveVelocity will continue to move the axis when its **Execute** input goes low.

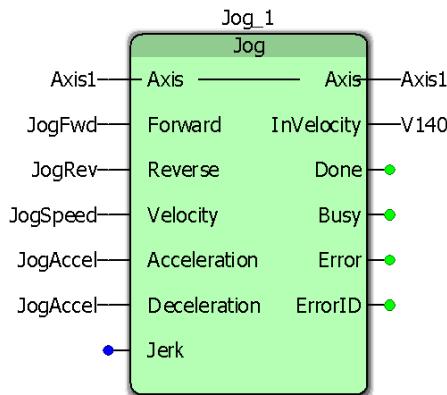


Figure 30: Function block variant of the Enable Model.

### **Variant #2:**

Some activities may require the characteristics of both function block models to perform their function well. The Feed\_To\_Length example below is designed to move an axis a default distance, but update the actual distance once an expected sensor input reports the position of the product being processed. That part of the activity can be handled nicely with an Execute model function block; all of the functions used within are Execute format. The Feed\_To\_Length function block shown here has added functionality; VAR\_INPUTS are provided so that the function block can keep track of expected registration marks, and provide **Error** outputs if a consecutive number missed marks, or MissedLatchLimit is reached. If programmed according to PLCopen, an Execute model function cannot remember how many

consecutive registration marks may have been missed, because you would have to initialize that counter when the rising edge of **Execute** goes high. **Execute model function blocks are designed to have no memory of what they were doing or how the previous execution resulted.** The function block below can perform all of its features better when designed as a hybrid function block with both **Enable** and **Execute** inputs. When the block is Enabled, the code that monitors axis motion and looks for expected registration marks at specific intervals is always running. Motion takes place when **Execute** goes high. Notice the block provides both **Valid** AND **Done** outputs.

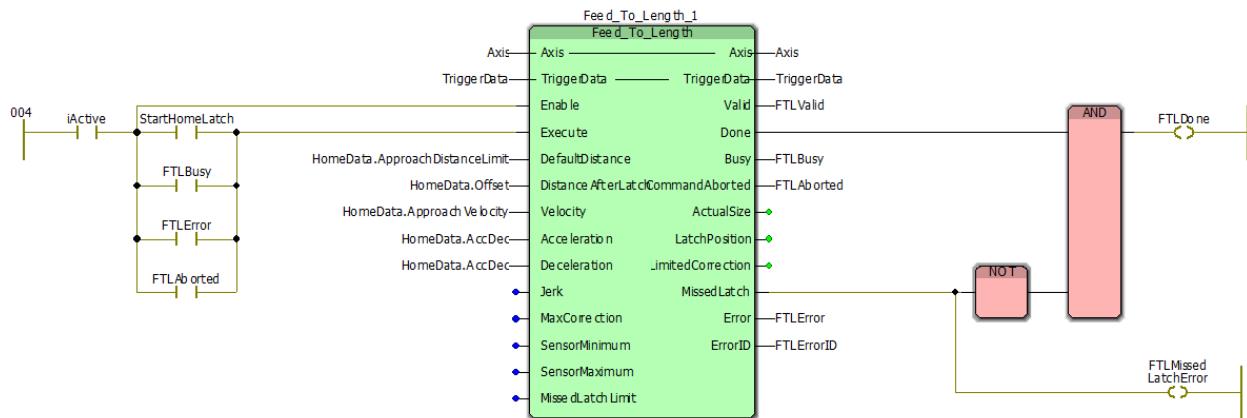


Figure 31: Function Block variant incorporating both Enable and Execute models.

## Recommended Interlocks

Make use of the status outputs of PLCopen function blocks by including them in the interlock logic which executes the function. This will ensure that the expected logic flow is occurring, and that in the event of an error, debugging is made easier because the function block which caused an error will be reporting the ErrorID. Avoid pulsing the Execute input for one scan using an R\_TRIG function. If an error occurs, the error will only be reported for one scan. Unless specific error trapping techniques are used, visual observation of the program will prove difficult to determine the source of the problem.

This logic is useful when calling sub functions within the function block you are creating, and is also a good method to apply when referencing the completed function block from other code.

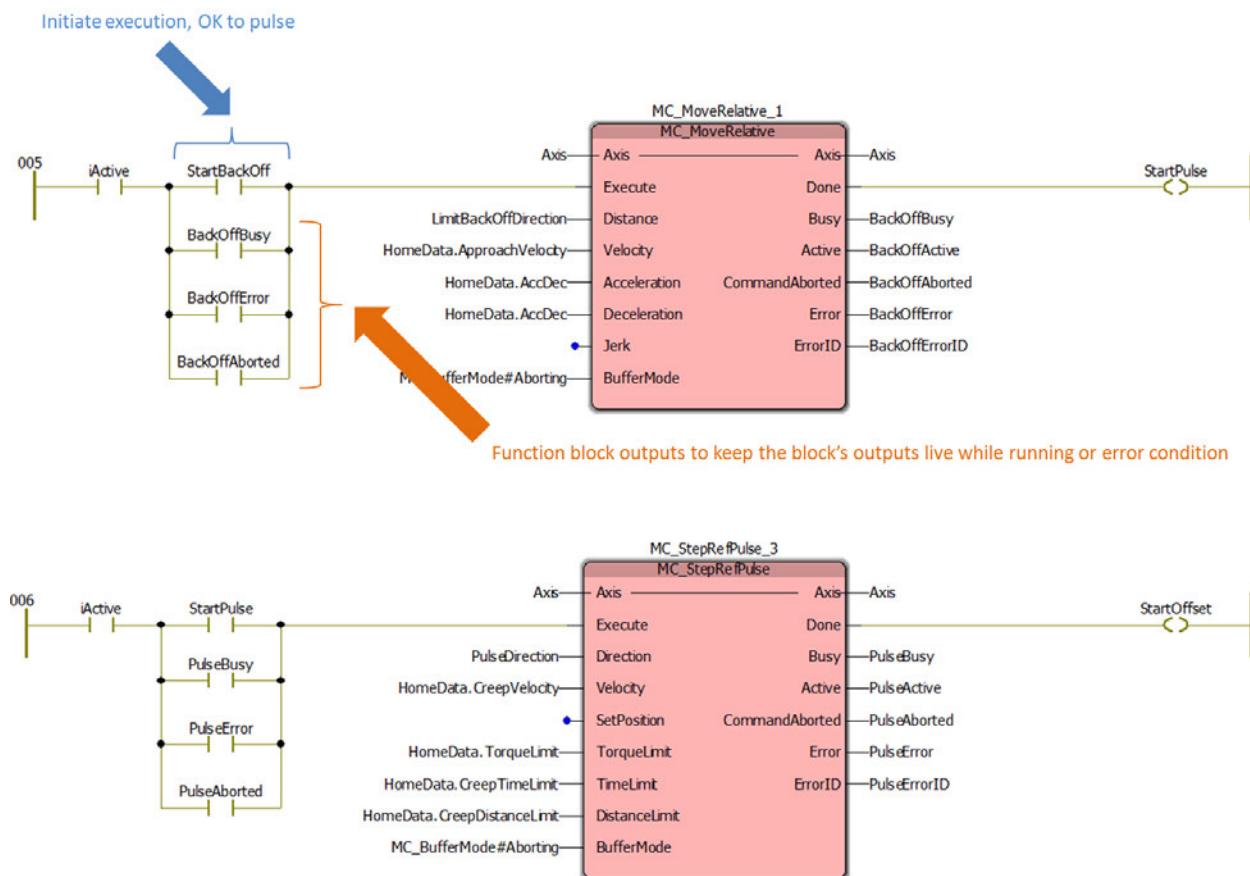


Figure 32: Recommended interlocks to improve debugging capabilities.

Subject: Application Note	Product: MPiec Series Controllers	Doc#: AN.MWIEC.01
---------------------------	-----------------------------------	-------------------

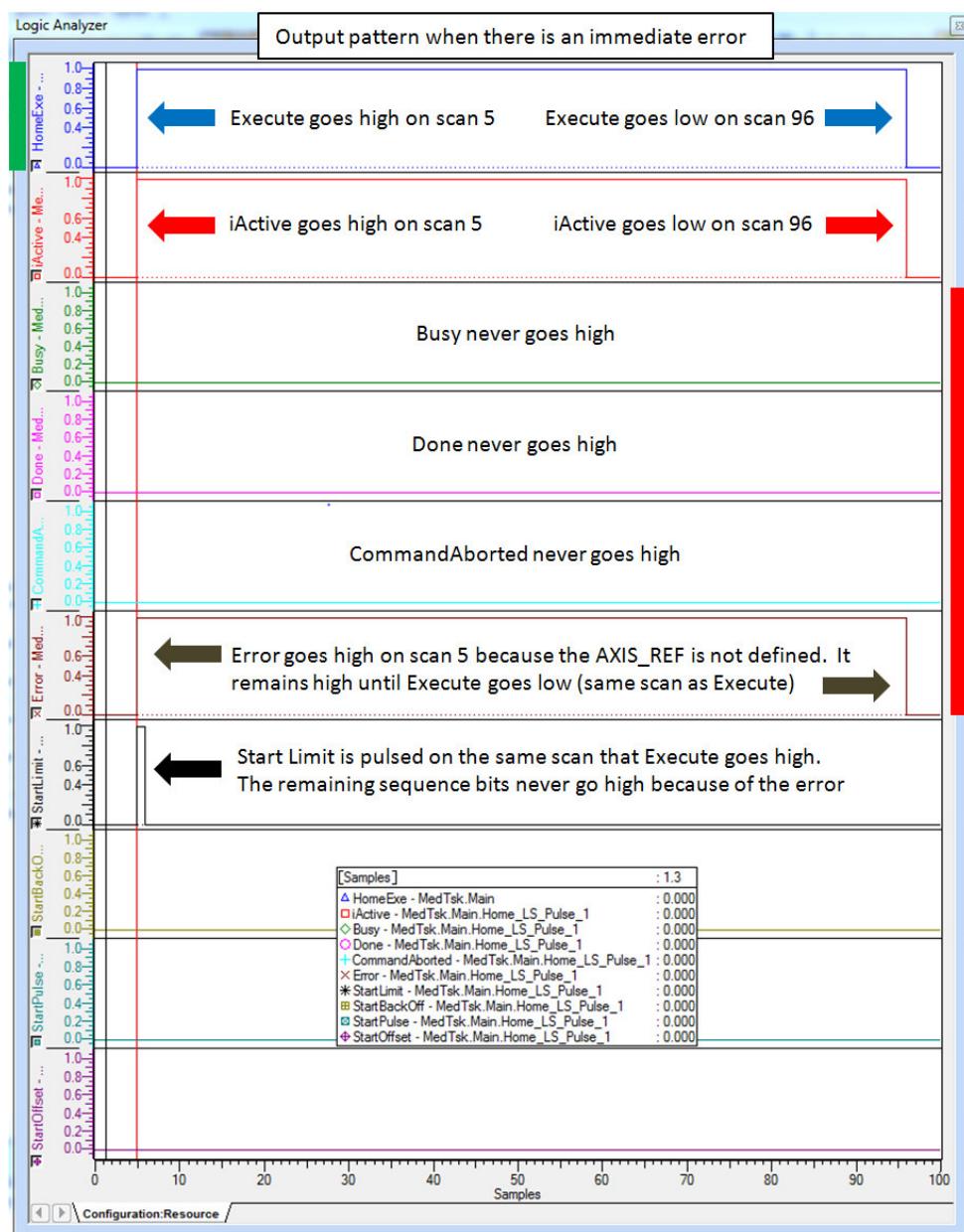
Title: Creating PLCopen Compliant Function Blocks in IEC 61131
--

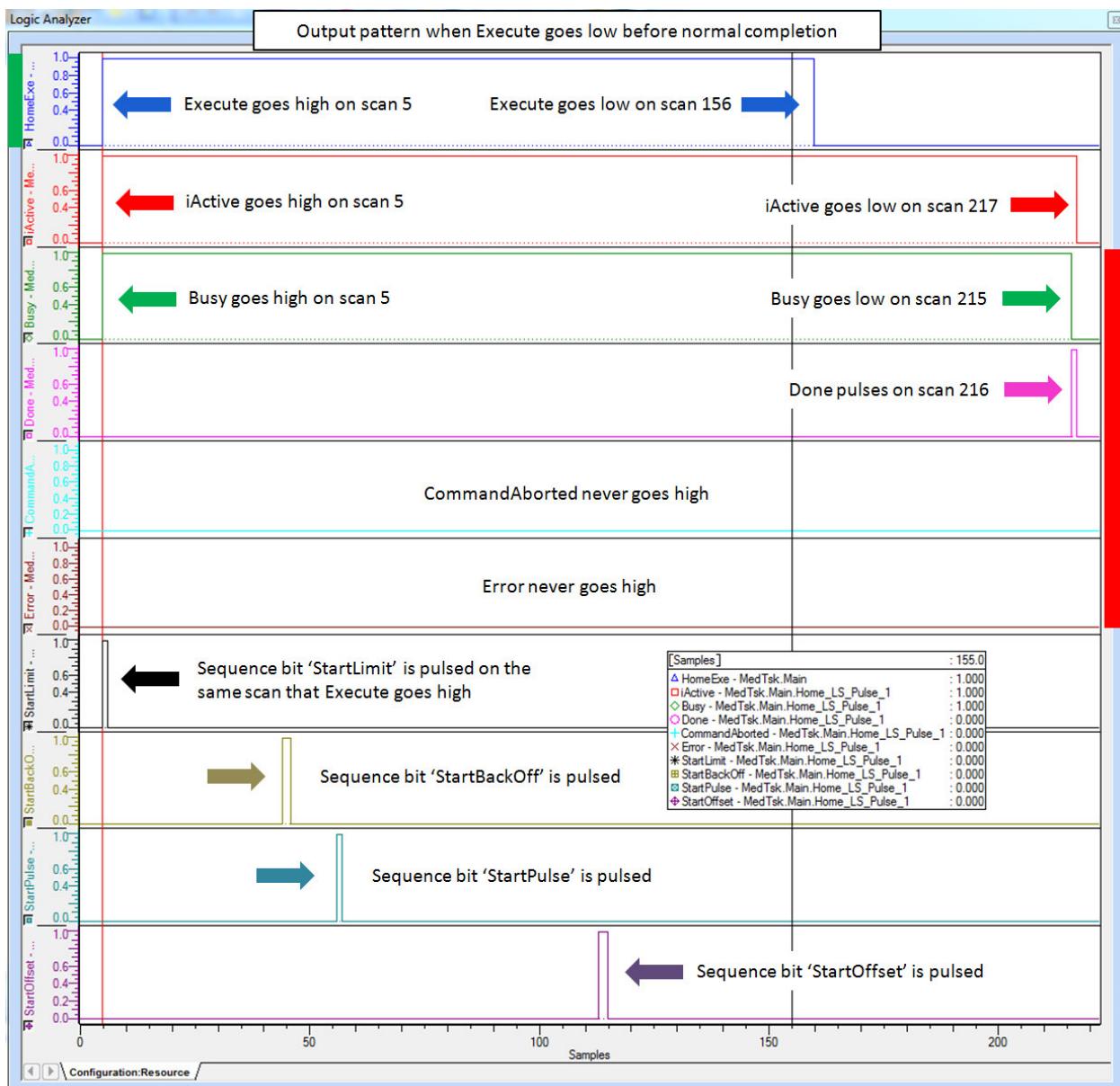
## Summary

The techniques described here are just one way to create function blocks that conform to the PLCopen specification. There are many ways to achieve the same results. Stick with a working code pattern to simplify debugging. When code is arranged in a similar structure from function block to function block, the time required to become familiar with the logic is reduced along with troubleshooting effort. Don't underestimate the value provided by the PLCopen specification; use it for increased efficiency in automation.

## Appendix A: Logic Analyzer traces

Use a recording feature provided by the IEC 61131 software to confirm proper behavior. Visually testing in debug mode is not enough to confirm that every output is behaving correctly scan by scan.





Note: iActive stays high for an additional scan to clear all status data for the next time the function block is executed.