Technical Paper
PLCopen Technical Committee 2

Function Blocks for Motion Control: Part 3 - User Guidelines

PLCopen Document, Published as Version 2.0.

Date: Feb. 21, 2013

Total number of pages: 94

Function Blocks for Motion Control

The following paper is a document created by the PLCopen Task Force Motion Control. As such it is an addition to other documents of the PLCopen Task Force Motion Control, such as Part 1 – Function Blocks for Motion Control, and Part 2 – Extensions. As such it is released on an on-going basis, if new examples become available.

It summarizes the results of the PLCopen Task Force Motion Control, containing contributions of all its members.
The present specification was written thanks to the following members of the Task Force:

Istvan Ulvros	Tetrapak
Hilmar Panzer	3S Smart Software Solutions
Joachim Unfried	B & R Automation
Klaus Bernzen	Beckhoff
Roland Schaumburg	Danfoss
Djafar Hadiouche	GE
Harald Buchgeher	KEBA
Johannes Kühn	Lenze
Candido Ferrio	Omron
Carlos Ruiz	Omron
Willi Gagsteiger	Siemens
Günter Neumann	Siemens
Kevin Hull	Yaskawa
Eelco van der Wal	PLCopen

Change Status List:

Version number	Date	Change comment
V 0.1	September, 04 2003	Fist version – result of the decision on the meeting of July 2003, to separate this part from the overall V0.8
V 0.2	December 8, 2003	Second version – includes all information in conjunction with the release of part 2 – extensions
V 0.3	April 16, 2004	First draft for release
V 0.3a	May 24, 2006	As result of the meeting in Sitges, Spain and previous feedback, spec. OMAC Packaging Workgroup related.
V 0.3b	September 21, 2006	As result of meeting Hamburg – Enable vs. Execute added
V 0.4	April 18, 2008	Several edits. Released draft version
V 0.41	May 19, 2010	Start of an update of the document due to V 2.0 of Part 1 & 2 Result of meeting in Kempten, Germany
V 0.42	April 29, 2011	Further in-house update after release of Version 2.0 of Part 1 and 2
V 0.43	December 12, 2011	Added example on synchronized motion and published to group
V 0.5	September 12, 2012	As result of the meeting in July in the vicinity of Amsterdam as well as extended editing of the document by PLCopen
V 0.51	October 11, 2012	As a result of the webmeeting
V 0.52	October 25, 2012	Minor editing done throughout the document. Last open issues resolved. New drawings added at Camming. In parallel are 2 examples from Yaskawa.
V 0.53	December 10, 2012	Basis for webmeeting. FBs Jog to Position and Axes Interlock added.
V 0.54	December 14, 2012	Result webmeeting.
V 0.55	January 18, 2012	Result of the webmeeting that week. MC_Jog, MC_Inch and MC_AxesInterlock changed, and some minor editorial issues
V 0.56	February 21, 2013	As a result of the webmeeting on Jan. 29 and last changes included.
V 2.0	February 21, 2013	Version 0.56 finalized for publication as V 2.0

Table of Contents

Table of Figures

1. General

This document is the third part of a set of documents of the PLCopen Task Force Motion Control. Currently, the set consists of 6 documents:

- Part 1 – Basics Function Blocks for Motion Control
- Part 2 – Extensions. Included in Part 1 since V 2.0
- Part 3 – User Guidelines
- Part 4 – Coordinated Motion
- Part 5 – Homing Procedures
- Part 6 – Fluid Power

With the publication of the first two parts of the motion control specification, it became clear that there was a need for application examples. For this reason this part was created.
This document is released on an on-going basis. With every release new examples are added. The first release was as version 0.3, in April 2004. The second release in April 2008 as Version 0.4. This version is the 3rd release and with the update and integration of part 1 and 2 it was decided to make the release in line with the other specifications.

1.1. Objectives

The objective of this document is to define a set of examples and clarifications for users of the other PLCopen documents on Motion Control. The examples presented here are for explanations only, are not seen as the only way to solve the application, or the only correct way to do this.
This document is not intended as an IEC 61131-3 tutorial – basic understanding of IEC 61131-3 is seen as a pre-requisite.
These examples are to be seen as examples only – they have not been tested in practice on real implementations. Also they are shown in somewhat different representation and languages, matching small differences in look and feel of implementations.

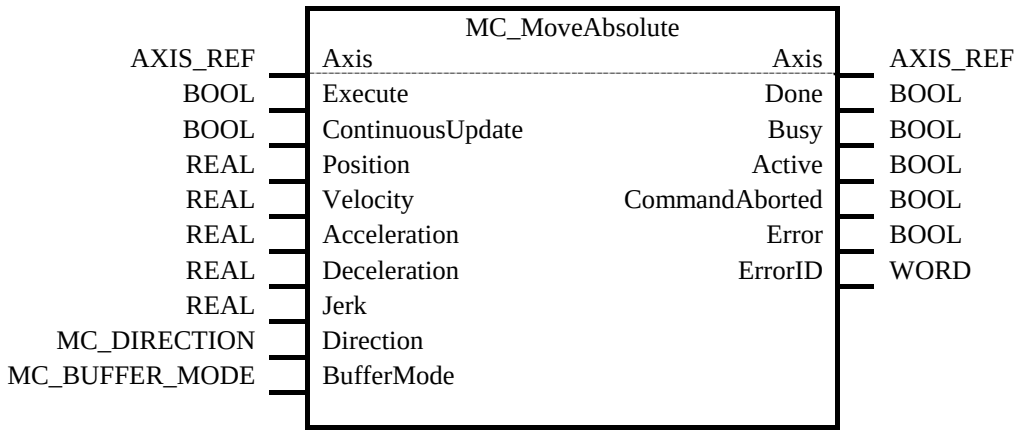
1.2. User Derived Function Blocks

The IEC 61131-3 defines Program Organization Units, POUs, consisting of Functions, Function Blocks and Programs. Within this concept, a user can generate own libraries of POUs. Of particular interest are the User Derived Libraries of Function Blocks. Within such a library, a user identifies the reusable parts of an application program, created with the standard available POUs, as well as with the PLCopen Motion Control Function Blocks.
This document shows how users can generate their own library, dedicated to their own application area. By creating such a library, and making it available throughout their organization, one can save a tremendous amount of time in the next project. Moreover, the usage of own libraries enhance the readability and transparency of the application programs generated.

1.3. Graphical versus textual representation

In the existing documents of the PLCopen Motion Control Function Blocks, there is a graphical representation used for clarification of the Function Blocks. Of course, this also can be represented in a textual representation. The following example shows how this can be done.

MC_MoveAbsolute
Graphical representation



Textual representation

```
FUNCTION_BLOCK MC_MoveAbsolute
    VAR_INPUT
        Execute :          BOOL;
        ContinuousUpdate : BOOL;
        Position :         REAL;
        Velocity :         REAL;
        Acceleration :     REAL;
        Deceleration :     REAL;
        Jerk :             REAL;
        Direction :        MC_DIRECTION
        BufferMode :       MC_BUFFER_MODE
    END_VAR

    VAR_IN_OUT
        Axis :             AXIS_REF;
    END_VAR

    VAR_OUTPUT
        Done :             BOOL;
        Busy :             BOOL;
        Active :           BOOL;
        CommandAborted :   BOOL;
        Error :            BOOL;
        ErrorID :          WORD;
    END_VAR

    VAR
        (* define local variables here *)
    END_VAR

    (* define the internal code here *)

END_FUNCTION_BLOCK
```

1.4. History

Not all history is included here – just after release of version 0.4 in 2008.
Version 0.42 – included example on ‘cut-to-length’ and update of functionality at MC_Jog
Version 0.43 – Added example on synchronized motion and preparation for the face2face meeting
Version 0.5 – general overhaul, update to Version 2.0 of Part 1 and 2, and new examples added: label machine, and warehousing, and split of three segment CAM and Cut-to-Length.
Version 2.0 – a longer introduction added and making it in line with Version 2.0 of Part 1. For this reason (input ContinuousUpdate) this document was also released as Version 2.0.

2. Application of MC FB

In the following subchapters several application examples are explained. The intent is just to show how the PLCopen Motion Control Function Blocks can be used in practical applications. This does not mean that these examples will exactly fit any particular application: they are listed for clarifications only, are not tested in practice on any system, and adaptations can be required.

2.1. Getting started

This will show the simple startup procedure by using MC_Power, MC_Home and MC_MoveAbsolute.

The startup of a motion control axis goes via issuing the following FBs:

- One MC_Power per axis to enable the axis
- For absolute positioning MC_Home is needed to define the home position. (Note: MC_Home is not always needed, for instance with MC_MoveVelocity and a rotating axis)
- Now it is possible to position the axis, for instance via issuing MC_MoveAbsolute

The picture below shows the graphical representation of this combined functionality.

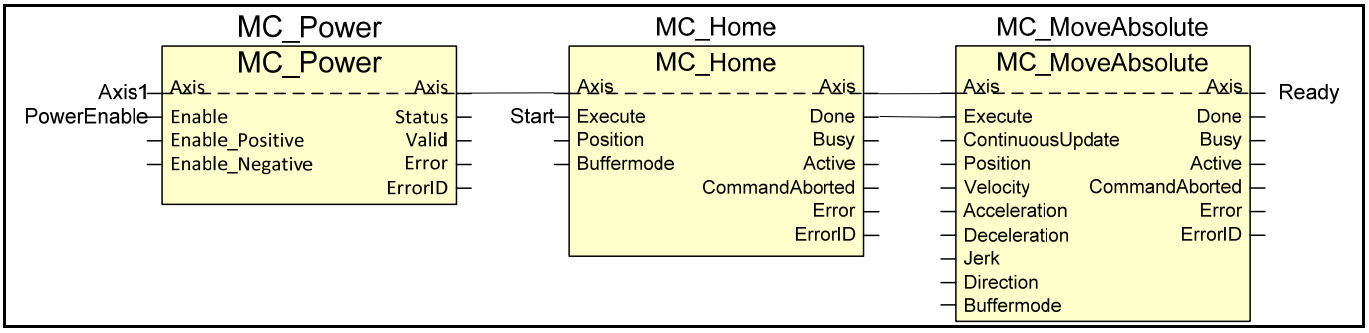


Figure 1 - Initialization program

Note: for clarity sake not all inputs are connected to a value

The referred axis is called here Axis1, and is linked to all 3 Function Blocks. MC_Power is enabled via setting 'PowerEnable'. Thereafter 'Start' is set to execute MC_Home. After the homing position is reached the 'Done' output of MC_Home is set. This executes the MC_MoveAbsolute Function Block which moves to the set position. The output 'Ready' will be set when the position is reached.
This basic functionality can be extended upfront with the use of the MC_ReadAxisInfo Function Block. In this way one can check if the communication to the drive is established, and can use the output 'ReadyForPowerOn' to enable MC_Power. Feedback is given via the output 'PowerOn' which can be the trigger to start MC_Home. This is than reflected in the output 'IsHomed' which can then be used to start the absolute movement.

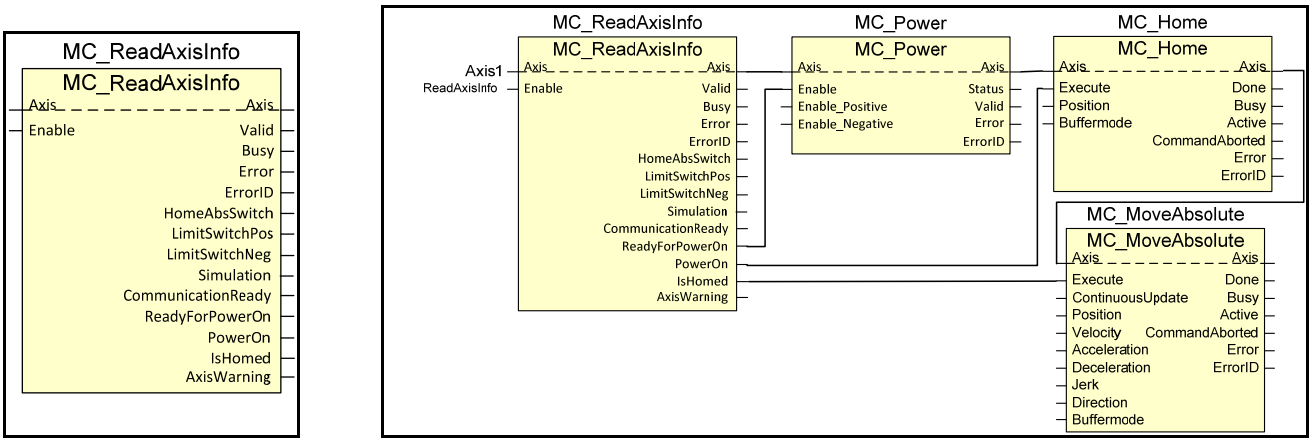


Figure 2 - Extended Initialization program

2.2. Label machine

2.2.1. Application description

The task is to place a label at a particular position on a product.
The application has two drives, one to feed the product via a conveyor belt, the other to feed the labels and to place the labels on the products. The labeling process is triggered by a position detection sensor. From the detection of the product to the start of the label movement there is a delay depending on the velocity of the conveyor, the position of the sensor and the position of the label on the product.

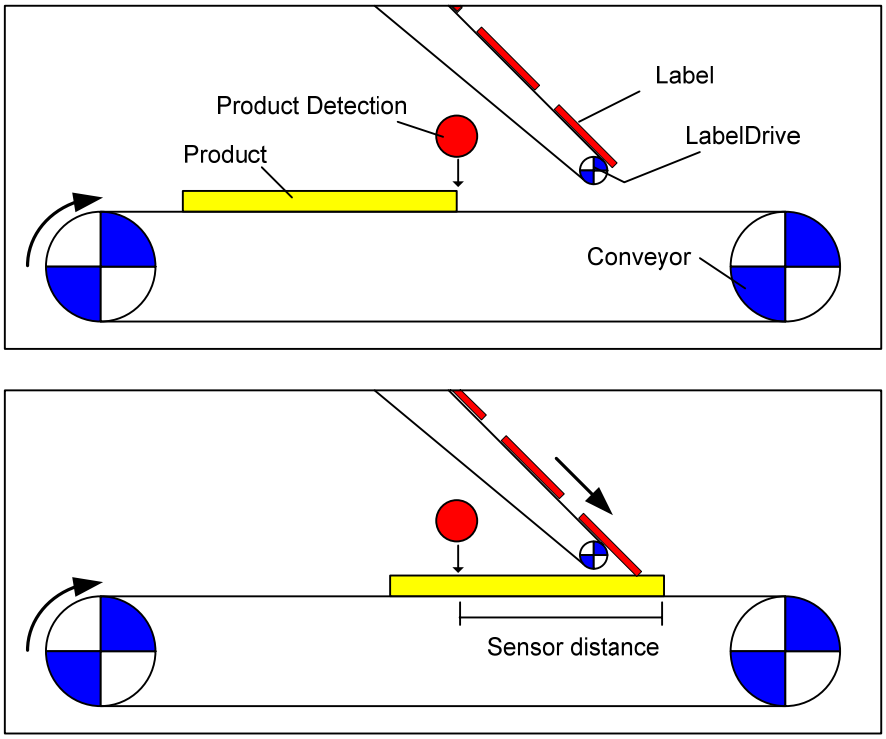


Figure 3 - Labeling machine

2.2.2. Programming example

This example shows a way to solve this task.
Both axes move with the same velocity setpoint. The delay for TON is calculated from the sensor distance and the velocity. After a labeling step the LabelDrive stops again and waits for the next trigger, while the conveyor continuously moves.

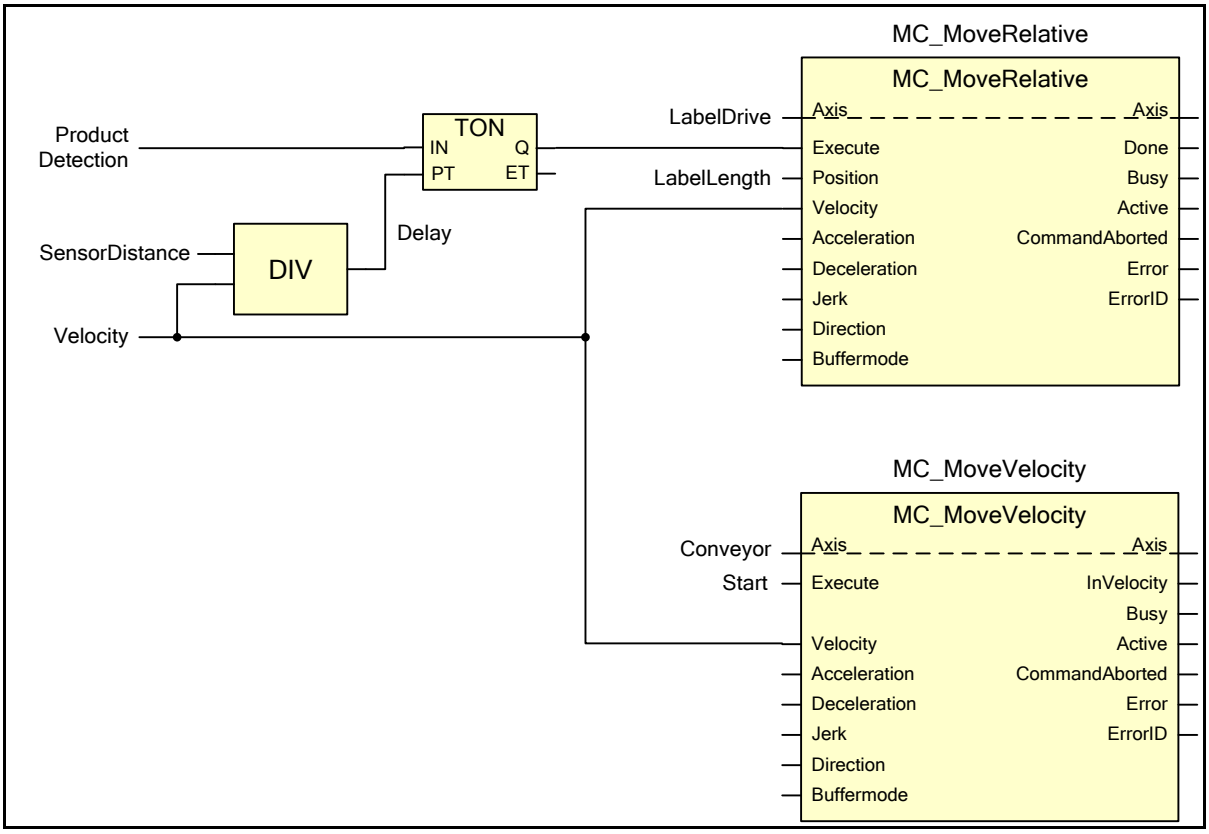


Figure 4 - Program example for the labeling machine

2.2.3. Possible Improvements

Although this principle is working, there are some possibilities to improve the functionalities and performance to achieve faster and more precise machines. Ways to do this are:

- Compensate for the drift of the label position as a result of the sum of incremental errors.
- A fast touch-probe input to detect the start position of the product more precisely
- A MC_CamIn or MC_GearInPos function to synchronize the label and product position in order to position the label more exact on the product. The conveyor should be the Master axis and the LabelDrive the Slave axis. In this way a mismatch caused by acceleration of the conveyor during labeling can be avoided.
- If the product is smaller than the sensor distance a kind of FIFO for product tracking can be necessary. See e.g. 2.18 FIFO Function Block.

2.3. Warehousing example

2.3.1. Application description

The purpose of this application is to automatically retrieve goods from a storage cabinet with shelves. The goods are stored in pallets that can be retrieved with a fork system.

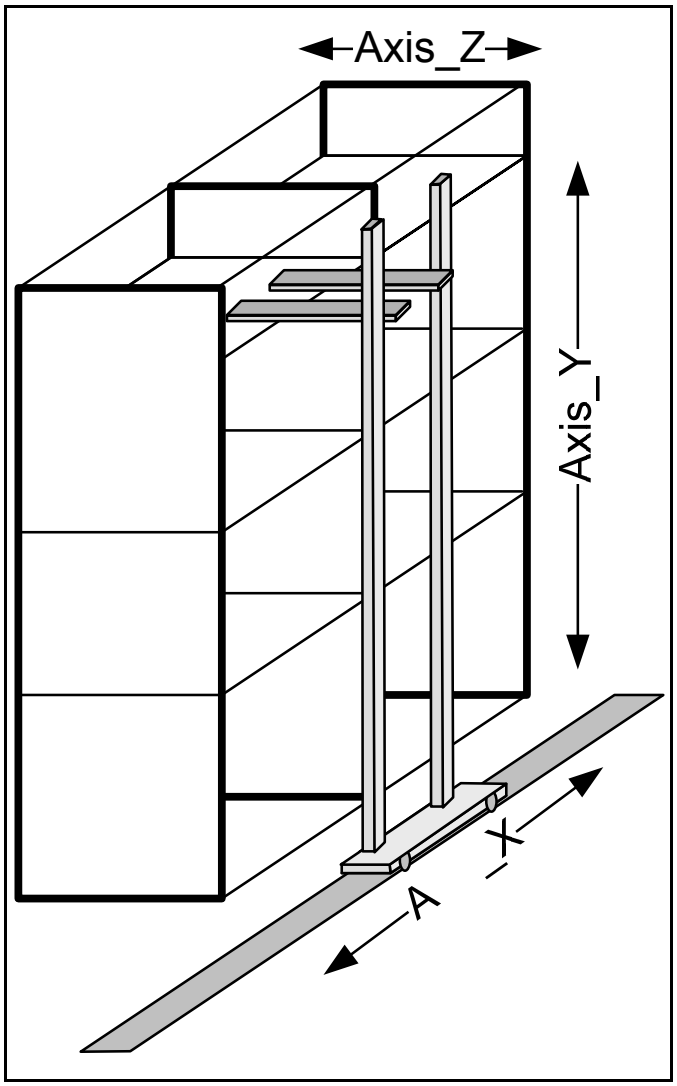


Figure 5 - Warehousing example

The warehouse task is to move the fork with three axes to place or take the pallet:

- Axis X moves along the floor;
- Axis Y moves to the needed height;
- Axis Z moves the fork into the shelf to fetch the pallet.

The sequence is to move the axes X and Y to the requested position. As soon as both axes have reached this position, the Z axis moves into the shelf under the pallet, in this example for 1000 mm. Then the Y axis lifts the pallet for another 100 mm to lift the pallet from the shelf, so it can be moved out of the shelf and to the required position to deliver it.

This example can be implemented in different ways. A straightforward approach is to use Part 1 Function Blocks.
Alternatively, a XYZ group could be defined in controllers supporting PLCopen Part 4, Coordinated Motion, which can simplify and optimize the movements.

2.3.2. First programming example (using Part 1)

This version could be implemented in the following way by only using Function Blocks from Part 1.

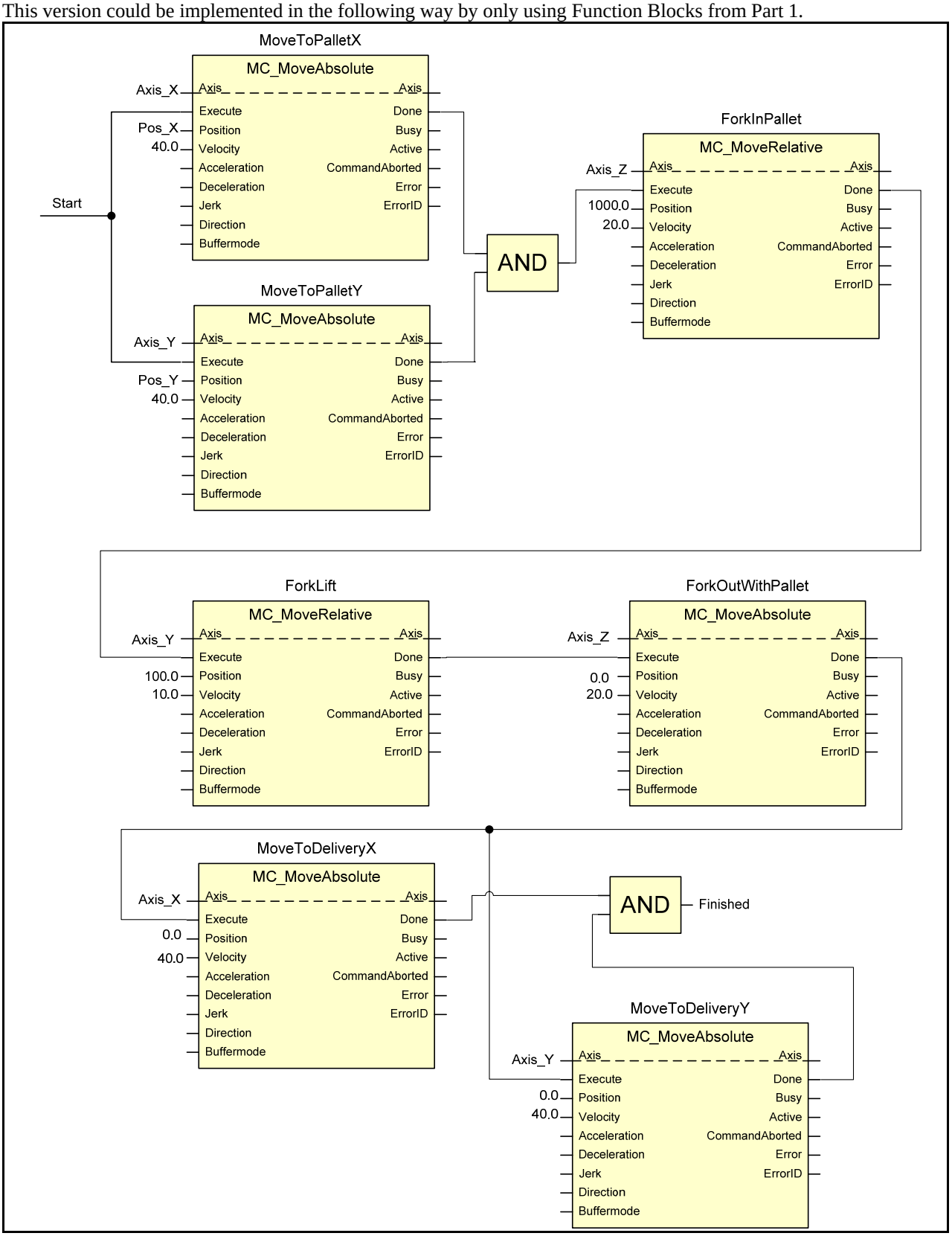


Figure 6 - First Program for warehousing example

Note: not all the specified inputs are shown in FBs above.

2.3.3. Timing diagram

The following graphics shows the sequence to fetch a pallet from the storage system.

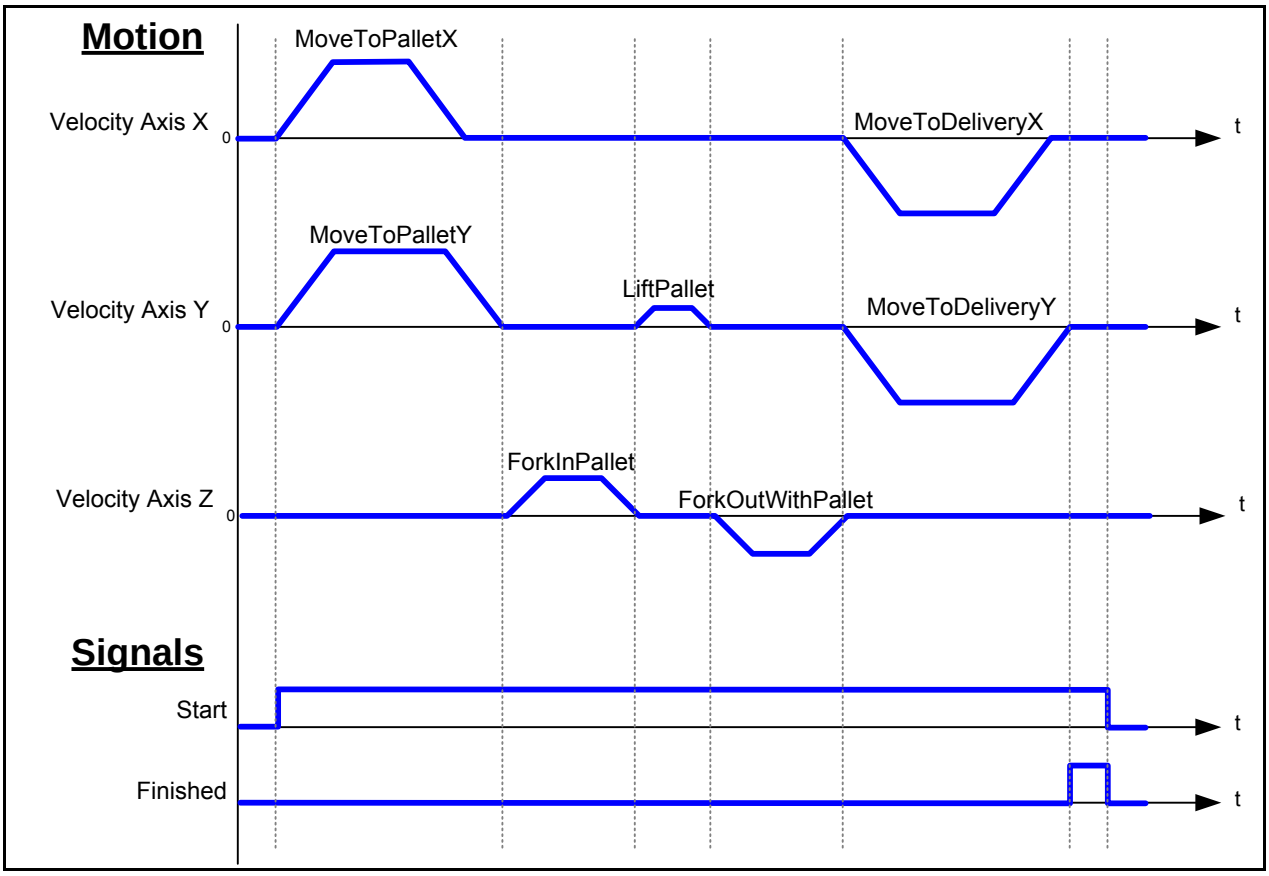


Figure 7 - Timing diagram for warehousing example

2.3.4. Second programming example (using Part 4)

This version is implemented using coordinated motion commands from PLCopen Part 4, Coordinated Motion. For information on how to create Axis Groups and enable them for coordinated operation, refer to chapter 4.1 of Part 4 specification. In this example, the group XYZLifter is made up from Axis_X, Axis_Y and Axis_Z. Blending is used to optimize the approach time to the end positions (The motion of the fork Axis_Z does not have to wait for completion of the movement of the Axis_X and Axis_Y to enter the pallet). This has to be done with a "TMCornerDistance" method to avoid collision with the shelf (in this particular case, we assume the distance from fork to shelf is bigger than 100). For this cornering to become effective, the 'Busy' output of precedent Function Blocks is triggering the 'Execute' of the buffered movement.

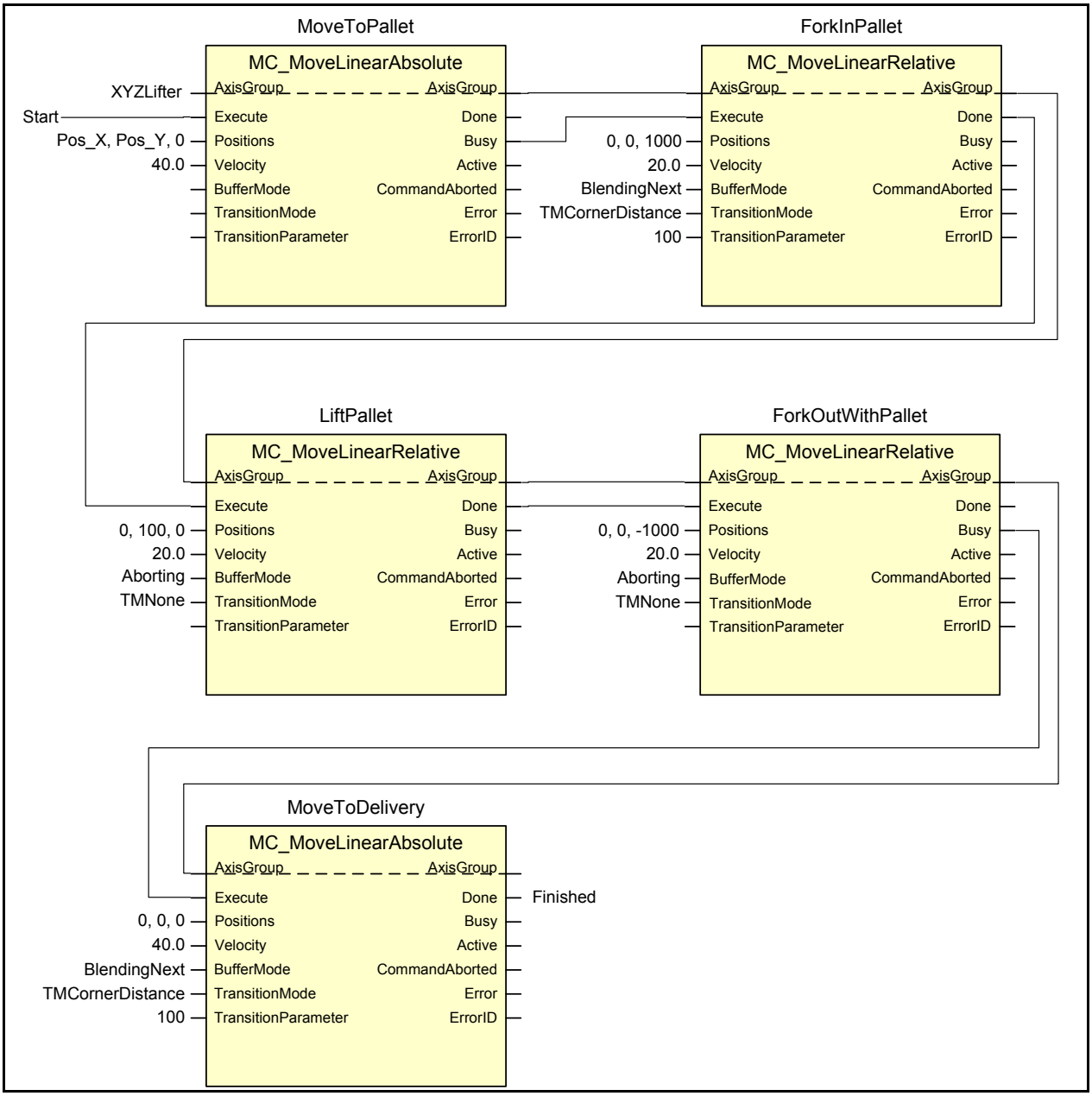
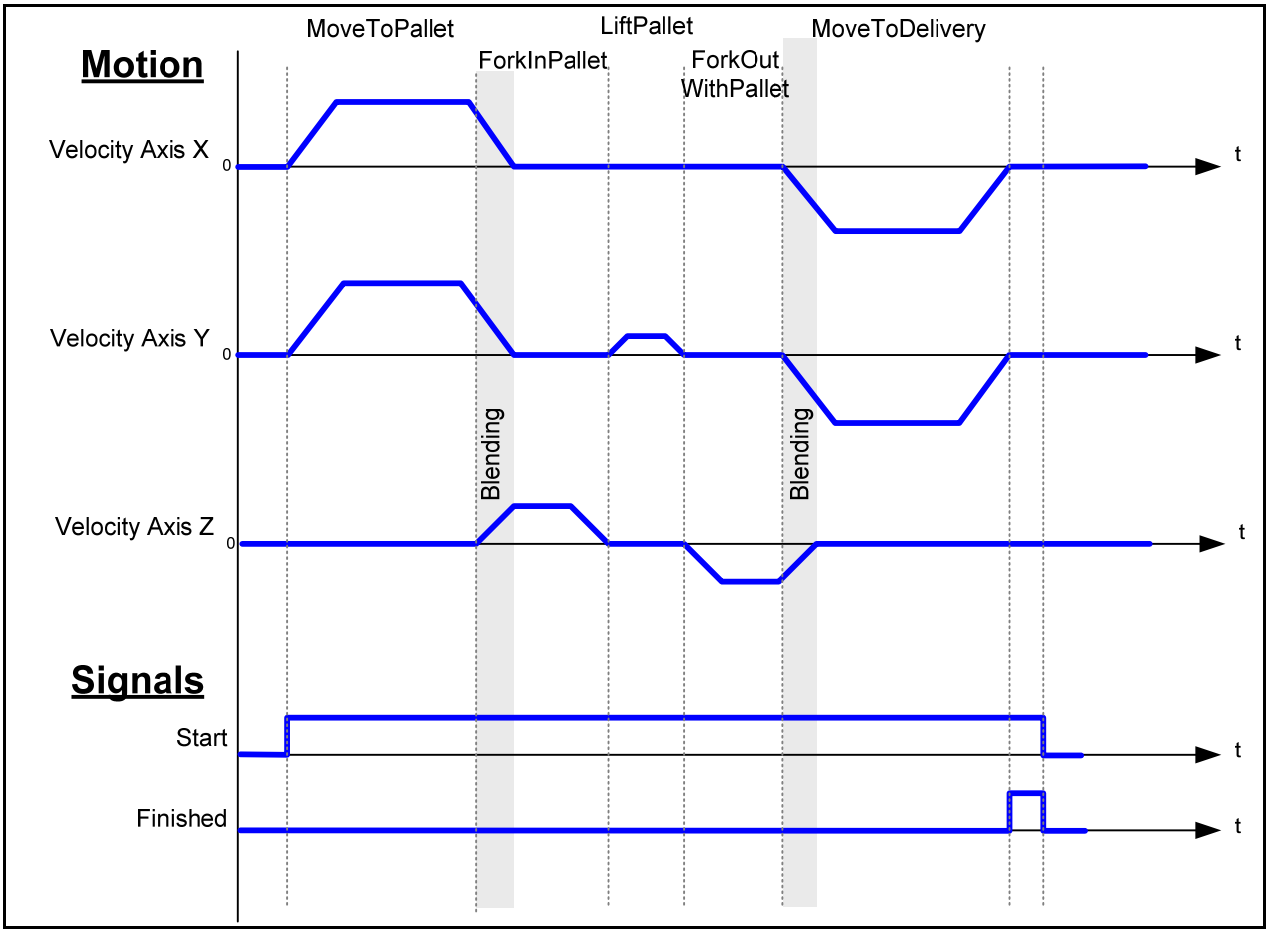


Figure 8 - Second program example for warehousing

2.3.5. Timing diagram

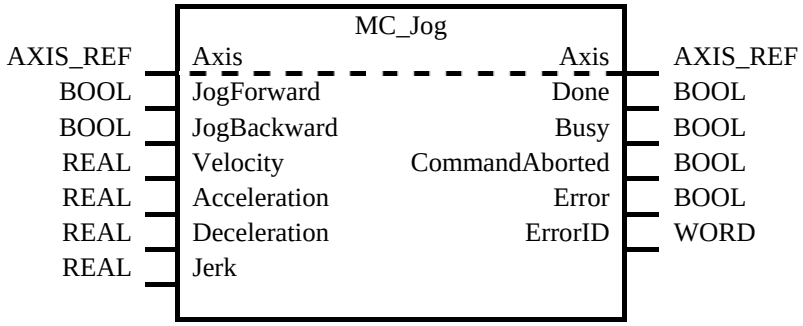
The following figure shows the sequence with the use of Part 4 Function Blocks. Blending between movements provides optimization.



2.4. Jogging

The following shows an example of a User Derived Function Block with the usage of MC_MoveVelocity and MC_Halt FBs internally and an Enable behavior via the combination of the inputs JogForward and JogBackward.

FB-Name	**MC_Jog**	
This User Derived Function Block commands a jogged movement to an axis as long as the input 'JogForward' or 'JogBackward' is SET.		
VAR_IN_OUT		
Axis	AXIS_REF	Reference to the axis
VAR_INPUT		
JogForward	BOOL	Start the jogged motion in positive direction
JogBackward	BOOL	Start the jogged motion in negative direction
Velocity	REAL	Value of the maximum 'Velocity' (not necessarily reached) [u/s].
Acceleration	REAL	Value of the 'Acceleration' (always positive) (increasing energy of the motor) [u/s^2]
Deceleration	REAL	Value of the 'Deceleration' (always positive) (decreasing energy of the motor) [u/s^2]
Jerk	REAL	Value of the 'Jerk' [u/s^3] (always positive)
VAR_OUTPUT		
Done	BOOL	This output is set for 1 cycle when MC_Halt is 'Done'
Busy	BOOL	The FB is not finished and new output values are to be expected
CommandAborted	BOOL	'Command' is aborted by another command
Error	BOOL	Signals that an error has occurred within the Function Block
ErrorID	WORD	Error identification
Note:		



Program written in Structured Text, using MC_Halt and MC_MoveVelocity:

```
FUNCTION_BLOCK MC_Jog
    VAR_IN_OUT
        Axis :           AXIS_REF;
    END_VAR

    VAR_INPUT
        JogForward :     BOOL;
        JogBackward :    BOOL;
        Velocity :       REAL;
        Acceleration :   REAL;
        Deceleration :   REAL;
        Jerk :           REAL;
    END_VAR

    VAR_OUTPUT
        Done :           BOOL;
        Busy :           BOOL;
        CommandAborted : BOOL;
        Error :          BOOL;
        ErrorID :        WORD;
    END_VAR

    VAR
        fbHalt :         MC_Halt;
        fbMoveVelocity : MC_MoveVelocity;
        iState :         INT;
    END_VAR

    CASE iState OF
        0:  (* wait for start conditions *)

            Done := FALSE;

            (* error conditions *)
            IF Error OR CommandAborted THEN
                IF NOT JogForward AND NOT JogBackward THEN
                    Error := FALSE;
                    CommandAborted := FALSE;
                    ErrorID := 0;
                END_IF
            END_IF

            IF JogForward AND NOT JogBackward THEN
                iState := 10;
            END_IF

            IF JogBackward AND NOT JogForward THEN
                iState := 20;
            END_IF

            IF iState <> 0 THEN
                fbMoveVelocity(Execute := FALSE,
                               Velocity := Velocity,
                               Acceleration := Acceleration,
                               Deceleration := Deceleration,
                               Jerk := Jerk,
                               Axis := Axis);

                Busy := TRUE;
            ELSE
                Busy := FALSE;
            END_IF

        10: (* move forward *)
            fbMoveVelocity(Execute := TRUE,
                           Direction := mcPositiveDirection,
                           Axis := Axis);

            IF fbMoveVelocity.Error THEN
                Error := TRUE;
                ErrorID := fbMoveVelocity.ErrorID;
                iState := 0;
            ELSIF fbMoveVelocity.CommandAborted THEN
                CommandAborted := TRUE;
                iState := 0;
            ELSIF NOT JogForward OR JogBackward THEN
                iState := 30;
```

```
                fbHalt(Execute := FALSE,
                       Deceleration := Deceleration,
                       Axis := Axis);
            END_IF

        20: (* move backwards *)
            fbMoveVelocity(Execute := TRUE,
                           Direction := mcNegativeDirection,
                           Axis := Axis);

            IF fbMoveVelocity.Error THEN
                Error := TRUE;
                ErrorID := fbMoveVelocity.ErrorID;
                iState := 0;
            ELSIF fbMoveVelocity.CommandAborted THEN
                CommandAborted := TRUE;
                iState := 0;
            ELSIF NOT JogBackward OR JogForward THEN
                iState := 30;

                fbHalt(Execute := FALSE,
                       Deceleration := Deceleration,
                       Axis := Axis);
            END_IF

        30: (* halt *)
            fbHalt(Execute := TRUE,
                   Axis := Axis);

            IF fbHalt.Error THEN
                Error := TRUE;
                ErrorID := fbHalt.ErrorID;
                iState := 0;
            ELSIF fbHalt.CommandAborted THEN
                CommandAborted := TRUE;
                iState := 0;
            ELSIF fbHalt.Done THEN
                Done := TRUE;
                iState := 0;
            END_IF
    END_CASE

END_FUNCTION_BLOCK
```

2.4.1. Short Explanation

The program code above shows, how the FB MC_Jog can be programmed using the basic PLCopen FBs MC_MoveVelocity and MC_Halt. A state machine is applied that makes sure, that the FBs are controlled in the correct way:

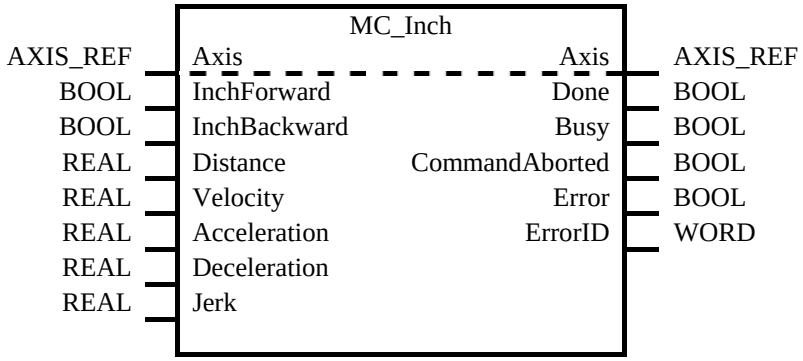
In state 0 the FB generally is idle. It first checks, if its Error or CommandAborted output (that might have been occurred during previous operations) can be reset. Then, when one of the move commands in positive or negative direction is TRUE, it calls the MC_MoveVelocity instance with Execute=FALSE to reset it and switches to the moving state 10 or 20.

In state 10 (or 20) the MC_MoveVelocity instance is started. When it shows an error or is aborted, the corresponding signals are set in the MC_Jog FB and the FB immediately goes back to state 0. When the commanding input signal (JogBackward or JogForward) becomes FALSE, the MC_Halt instance is reset (being called with Execute = FALSE) and the state is switched to 30.
In state 30 the MC_Halt FB is executed, until an error occurs, the FB is aborted or until the axis has been halted successfully. Then, the state machine is reset into state 0 and waits for new commands.

2.5. Inching

This FB is similar to MC_Jog. In contrast to it, the distance that is travelled by the axis is limited by a certain distance value, at which the axis stops the latest (meaning it can stop also before, when the Boolean inputs are released earlier). To travel more than this distance, the user must retrigger the input.

FB-Name	**MC_Inch**	
This User Derived Function Block commands an inching movement to an axis as long as the moving distance is not reached and the input 'InchForward' or 'InchBackward' is set.		
VAR_IN_OUT		
Axis	AXIS_REF	Reference to the axis
VAR_INPUT		
InchForward	BOOL	Start the inched motion in positive direction
InchBackward	BOOL	Start the inched motion in negative direction
Distance	REAL	Maximum moving distance
Velocity	REAL	Value of the maximum 'Velocity' (not necessarily reached) [u/s].
Acceleration	REAL	Value of the 'Acceleration' (always positive) (increasing energy of the motor) [u/s^2]
Deceleration	REAL	Value of the 'Deceleration' (always positive) (decreasing energy of the motor) [u/s^2]
Jerk	REAL	Value of the 'Jerk' [u/s^3] (always positive)
VAR_OUTPUT		
Done	BOOL	The 'Done' output is TRUE when MC_Halt is 'Done' OR MC_MoveRelative is 'Done'
Busy	BOOL	The FB is not finished and new output values are to be expected
CommandAborted	BOOL	'Command' is aborted by another command
Error	BOOL	Signals that an error has occurred within the Function Block
ErrorID	WORD	Error identification
Note:		



ST Code of MC_Inch is very similar to the code at MC_Jog above, with the difference that MC_MoveRelative is used versus MC_MoveVelocity to create the inching behavior:

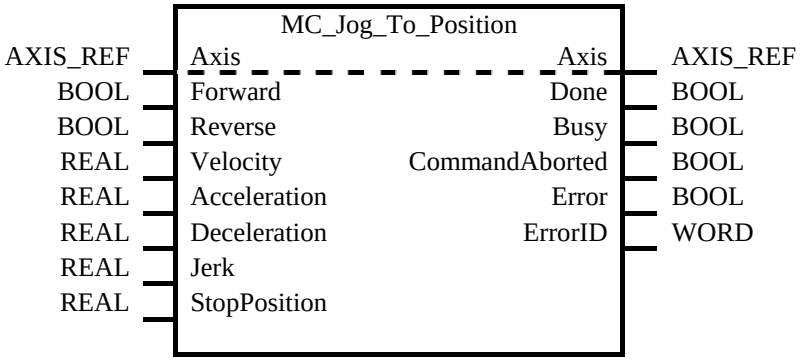
```
FUNCTION_BLOCK MC_Inch
    VAR_IN_OUT
        Axis :          AXIS_REF;
    END_VAR

    VAR_INPUT
        InchForward :   BOOL;
        InchBackward :  BOOL;
        Distance:       REAL;
        Velocity :      REAL;
        Acceleration :  REAL;
        Deceleration :  REAL;
        Jerk :          REAL;
```

```
    END_VAR

    VAR_OUTPUT
        Done :           BOOL;
        Busy :           BOOL;
        CommandAborted : BOOL;
        Error :          BOOL;
        ErrorID :        WORD;
    END_VAR

    VAR
        fbHalt :         MC_Halt;
        fbMoveRelative : MC_MoveRelative;
        iState :         INT;
    END_VAR

  CASE iState OF
    0:  (* wait for start conditions *)

        Done := FALSE;
        (* error conditions *)
        IF Error OR CommandAborted THEN
            IF NOT InchForward AND NOT InchBackward THEN
                Error := FALSE;
                CommandAborted := FALSE;
                ErrorID := 0;
            END_IF
        END_IF

        IF InchForward AND NOT InchBackward THEN
            iState := 10;
           fbMoveRelative.Distance := Distance;
        END_IF

        IF InchBackward AND NOT InchForward THEN
            iState := 20;
            fbMoveRelative.Distance := -Distance;
        END_IF

        IF iState <> 0 THEN
            fbMoveRelative(Execute := FALSE,
                           Velocity := Velocity,
                           Acceleration := Acceleration,
                           Deceleration := Deceleration,
                           Jerk := Jerk,
                           Axis := Axis);

            Busy := TRUE;
        ELSE
            Busy := FALSE;
        END_IF

    10: (* move forwards *)
        fbMoveRelative(Execute := TRUE,
                       Axis := Axis);

        IF fbMoveRelative.Error THEN
            Error := TRUE;
            ErrorID := fbMoveRelative.ErrorID;
            iState := 0;
        ELSIF fbMoveRelative.CommandAborted THEN
            CommandAborted := TRUE;
            iState := 0;
        ELSIF fbMoveRelative.Done THEN
            Done := TRUE;
        ELSIF NOT InchForward OR InchBackward THEN
            iState := 30;

            fbHalt(Execute := FALSE,
                   Deceleration := Deceleration,
                   Jerk := Jerk,
                   Axis := Axis);
        END_IF

    20: (* move backwards *)
        fbMoveRelative(Execute := TRUE,
                       Axis := Axis);

        IF fbMoveRelative.Error THEN
            Error := TRUE;
```

```
            ErrorID := fbMoveRelative.ErrorID;
            iState := 0;
        ELSIF fbMoveRelative.CommandAborted THEN
            CommandAborted := TRUE;
            iState := 0;
        ELSIF fbMoveRelative.Done THEN
            Done := TRUE;
        ELSIF NOT InchBackward OR InchForward THEN
            iState := 30;
            fbHalt(Execute := FALSE,
                   Deceleration := Deceleration,
                   Jerk := Jerk,
                   Axis := Axis);
        END_IF

    30: (* halt *)
        fbHalt(Execute := TRUE,
               Axis := Axis);

        IF fbHalt.Error THEN
            Error := TRUE;
            ErrorID := fbHalt.ErrorID;
            iState := 0;
        ELSIF fbHalt.CommandAborted THEN
            CommandAborted := TRUE;
            iState := 0;
        ELSIF fbHalt.Done THEN
            Done := TRUE;
            iState := 0;
        END_IF
    END_CASE

END_FUNCTION_BLOCK
```

2.6. Jog to Position

FB-Name	**MC_JogToPosition**	
This User Derived Function Block commands a jogged movement to an axis as long as the input 'Forward' or 'Reverse' is SET. Once they are removed, the axis will start a deceleration path to stop at a specified position. This function block is designed for use with rotary axes that may require several rotations at the specified deceleration rate before coming to a complete stop.		
VAR_IN_OUT		
Axis	AXIS_REF	Reference to the axis
VAR_INPUT		
Forward	BOOL	Start the jogged motion in positive direction
Reverse	BOOL	Start the jogged motion in negative direction
Velocity	REAL	Value of the maximum 'Velocity' (not necessarily reached) [u/s].
Acceleration	REAL	Value of the 'Acceleration' (always positive) (increasing energy of the motor) [u/s^2]
Deceleration	REAL	Value of the 'Deceleration' (always positive) (decreasing energy of the motor) [u/s^2]
Jerk	REAL	Value of the 'Jerk' [u/s^3] (always positive)
StopPosition	REAL	Final stop position in a rotary axis system.
VAR_OUTPUT		
Done	BOOL	This function block will pulse the 'Done' bit for one scan only at the completion of the deceleration to the StopPosition
Busy	BOOL	The FB is not finished and new output values are to be expected
CommandAborted	BOOL	'Command' is aborted by another command
Error	BOOL	Signals that error has occurred within Function Block
ErrorID	WORD	Error identification
Note:		

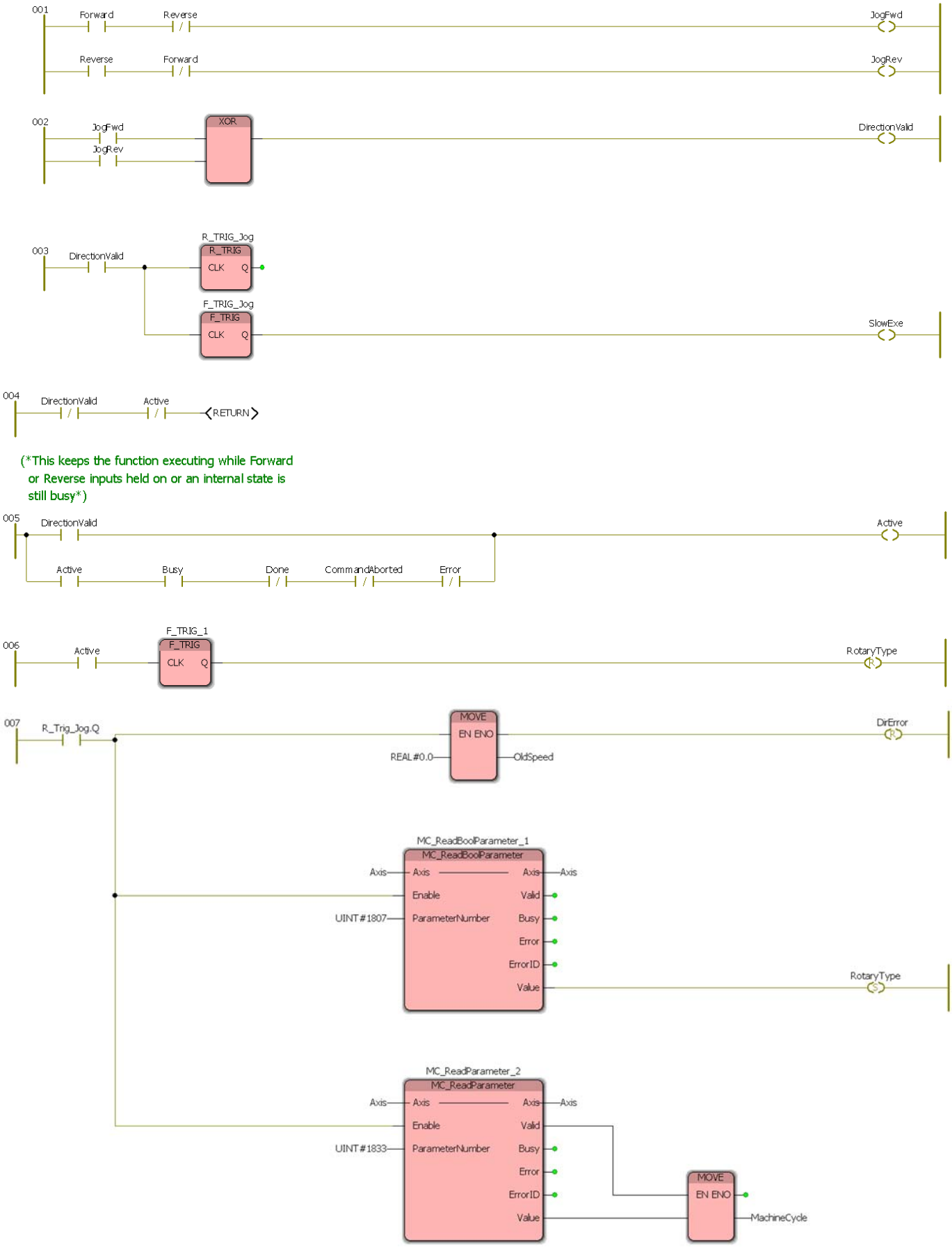


Variable Declaration:

```
FUNCTION_BLOCK Jog_To_Position
    VAR_IN_OUT
        Axis :              AXIS_REF;
    END_VAR

    VAR_INPUT
        Forward :           BOOL;
        Reverse :           BOOL;
        Velocity :          REAL;
        Acceleration :      REAL;
        Deceleration :      REAL;
        Jerk :              REAL;
        StopPosition:       REAL;
    END_VAR

    VAR_OUTPUT
        Done :              BOOL;
        Busy:               BOOL;
```

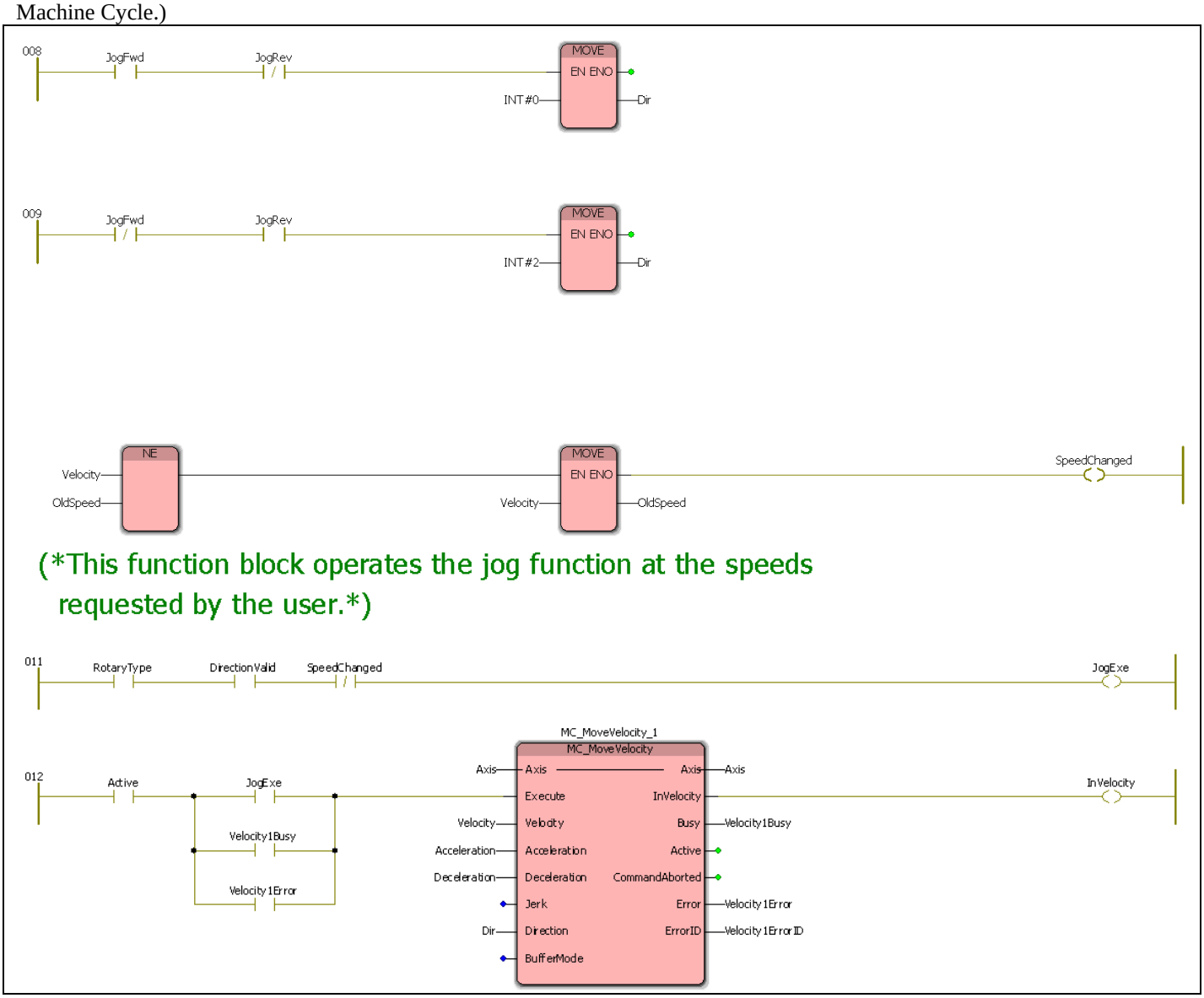
```
        CommandAborted :     BOOL;
        Error :              BOOL;
        ErrorID :            WORD;
    END_VAR

    VAR
     (* define the internal variables here *)
        Active :                     BOOL
        CommandedPositionCyclic :    REAL
        DecNotZero :                 BOOL
        Dir :                        INT
        DirectionValid :             BOOL
        DirError :                   BOOL
        DistanceToStop :             REAL
        F_TRIG_1 :                   F_TRIG
        F_TRIG_Jog :                 F_TRIG
        IdealDecelPosition :         REAL
        JogExe :                     BOOL
        JogFwd :                     BOOL
        JogRev :                     BOOL
        MachineCycle :               REAL
        MC_MoveAbsolute_1 :          MC_MoveAbsolute
        MC_MoveVelocity_1 :          MC_MoveVelocity
        MC_MoveVelocity_2 :          MC_MoveVelocity
        MC_ReadBoolParameter_1 :     MC_ReadBoolParameter
        MC_ReadParameter_1 :         MC_ReadParameter
        MC_ReadParameter_2 :         MC_ReadParameter
        MC_ReadParameter_3 :         MC_ReadParameter
        MOVE_INT_1 :                 MOVE_INT
        MOVE_INT_2 :                 MOVE_INT
        MOVE_REAL_1 :                MOVE_REAL
        MOVE_REAL_2 :                MOVE_REAL
        MOVE_REAL_3 :                MOVE_REAL
        MOVE_UINT_1 :                MOVE_UINT
        MOVE_UINT_2 :                MOVE_UINT
        MOVE_UINT_3 :                MOVE_UINT
        MOVE_UINT_4 :                MOVE_UINT
        MOVE_UINT_5 :                MOVE_UINT
        MoveAbsBusy :                BOOL
        MoveAbsError :               BOOL
        MoveAbsErrorID :             UINT
        OldSpeed :                   REAL
        OneCycleToStop :             BOOL
        R_TRIG_1 :                   R_TRIG
        R_TRIG_2 :                   R_TRIG
        R_TRIG_Jog :                 R_TRIG
        RotaryType :                 BOOL
        SlowBusy :                   BOOL
        SlowDone :                   BOOL
        SlowExe :                    BOOL
        SlowNow :                    BOOL
        SpeedChanged :               BOOL
        StopAtPositionExe :          BOOL
        StopDone :                   BOOL
        vel :                        REAL
        Velocify2Error :             BOOL
        Velocity1Busy :              BOOL
        Velocity1Error :             BOOL
        Velocity1ErrorID :           UINT
        Velocity2Error :             BOOL
        Velocity2ErrorID :           UINT
    END_VAR

END_FUNCTION_BLOCK;
```

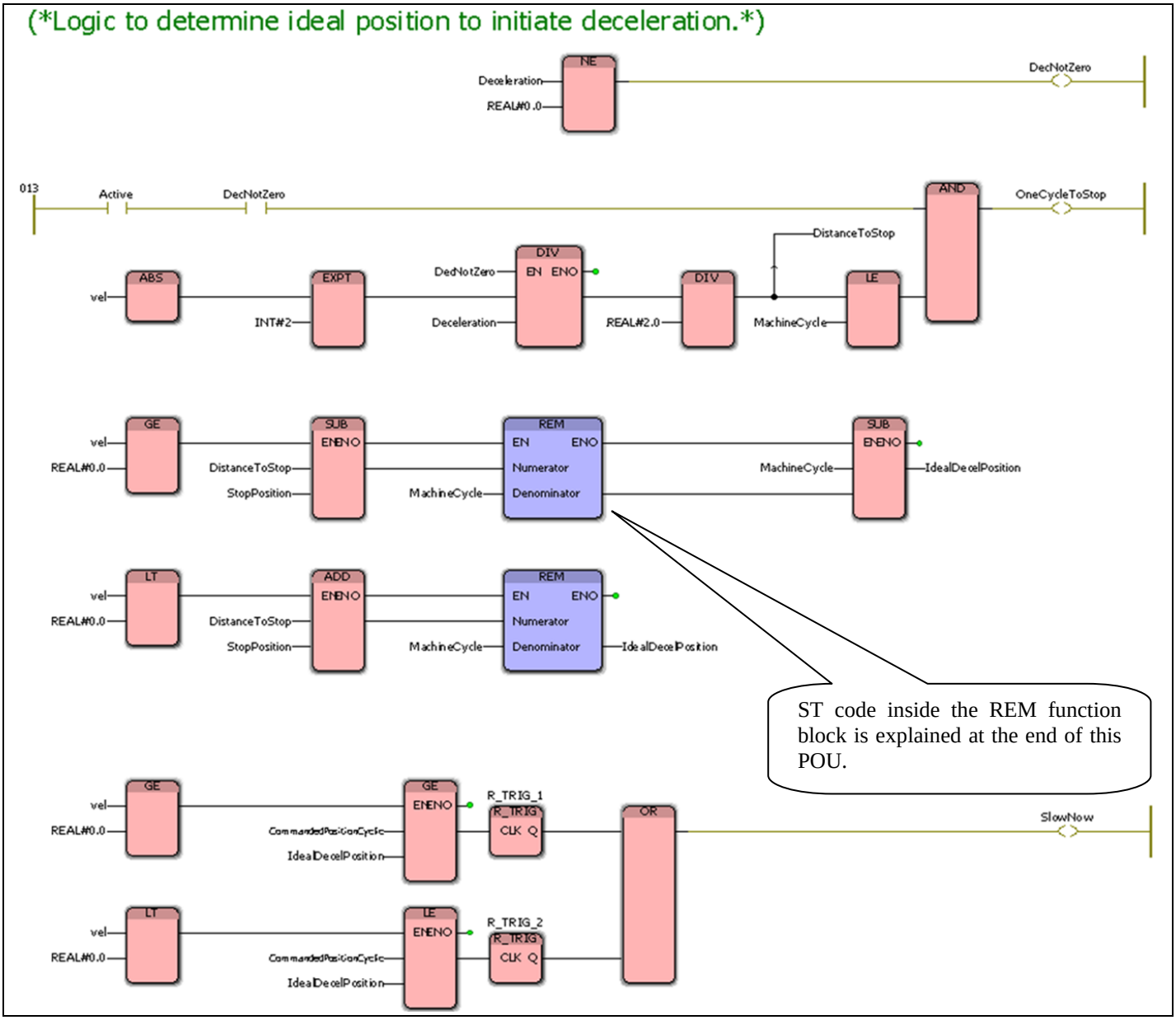
Internal code in Ladder Diagram:

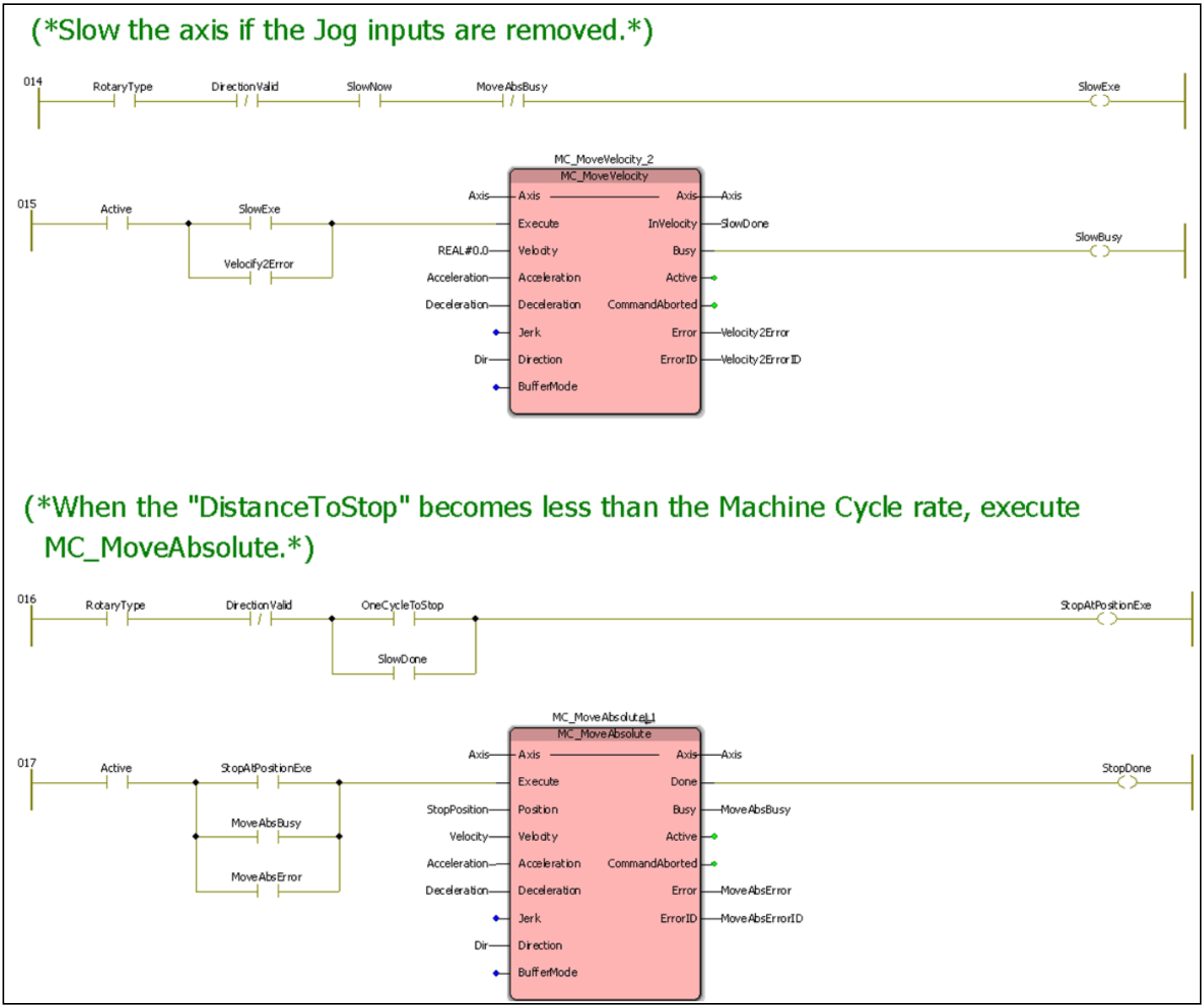

001
Forward
Reverse
JogFwd
Reverse
Forward
JogRev
002
JogFwd
JogRev
XOR
DirectionValid
003
DirectionValid
R_TRIG_Jog
R_TRIG
CLK
Q
F_TRIG_Jog
F_TRIG
CLK
Q
SlowExe
004
DirectionValid
Active
RETURN
(*This keeps the function executing while Forward or Reverse inputs held on or an internal state is still busy*)
005
DirectionValid
Active
Active
Busy
Done
CommandAborted
Error
006
Active
F_TRIG_1
F_TRIG
CLK
Q
RotaryType
007
R_Trig_Jog.Q
MOVE
EN ENO
REAL#0.0
OldSpeed
DirError
MC_ReadBoolParameter_1
MC_ReadBoolParameter
Axis
Axis
Axis
Axis
Enable
Valid
UINT#1807
ParameterNumber
Busy
Error
ErrorID
Value
RotaryType
MC_ReadParameter_2
MC_ReadParameter
Axis
Axis
Axis
Axis
Enable
Valid
UINT#1833
ParameterNumber
Busy
Error
ErrorID
Value
MOVE
EN ENO
MachineCycle

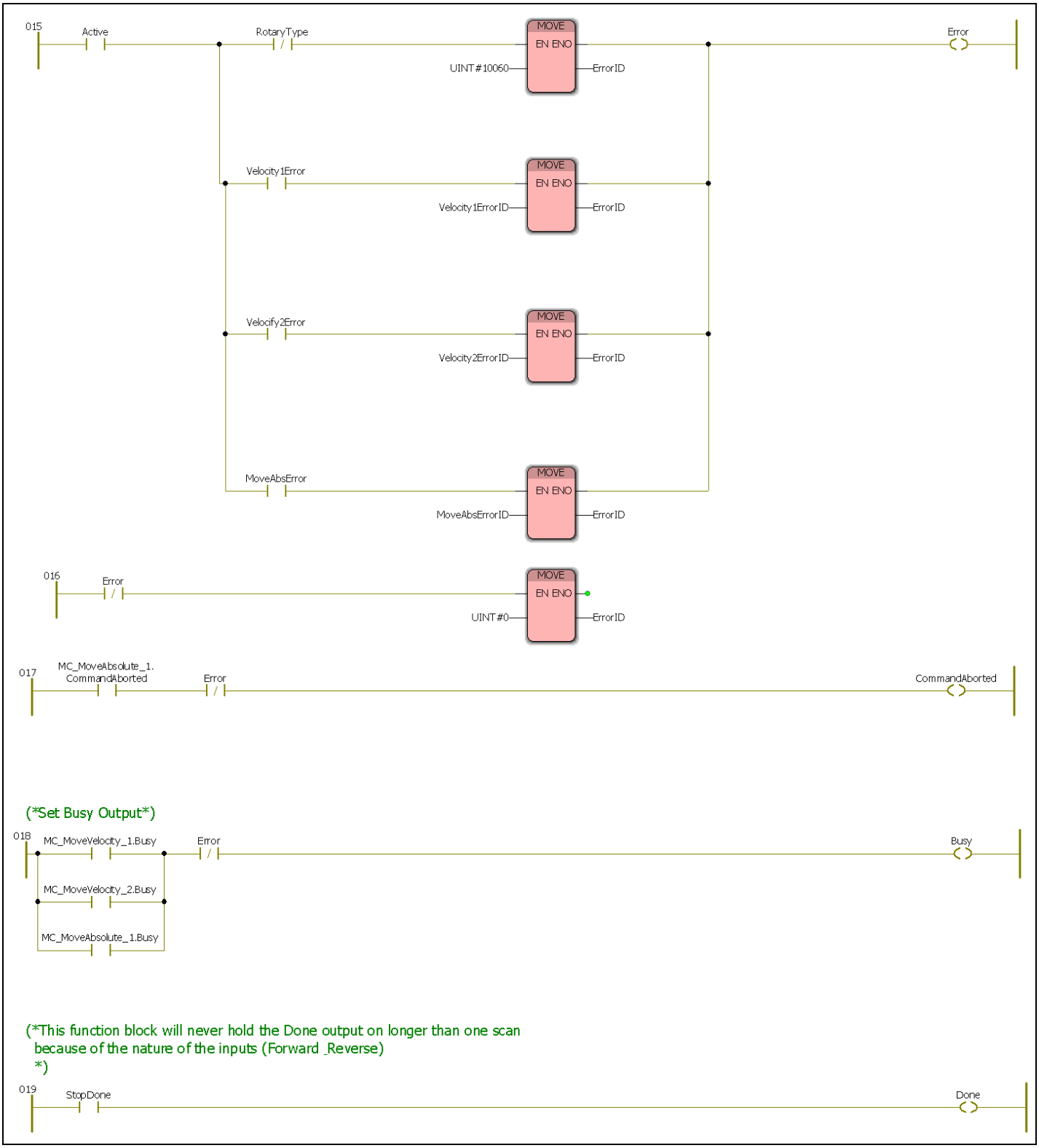
Optional: Use a vendor specific method to obtain the axis configuration. (In this example, Parameter UINT#1807 returns 0=linear, 1=Rotary.)

Optional: Use a vendor specific method to obtain the Machine Cycle of the axis. (In this example, Parameter UINT#1833 is the Machine Cycle.)




(*Logic to determine ideal position to initiate deceleration.*)
NE
Deceleration
REAL#0.0
DecNotZero
013
Active
DecNotZero
AND
OneCycleToStop
DistanceToStop
ABS
EXPT
DIV
DecNotZero
Deceleration
INT#2
DIV
REAL#2.0
LE
MachineCycle
vel
GE
REAL#0.0
SUB
DistanceToStop
StopPosition
REM
EN
ENO
Numerator
Denominator
MachineCycle
SUB
MachineCycle
IdealDecelPosition
LT
ADD
DistanceToStop
StopPosition
REM
MachineCycle
IdealDecelPosition
ST code inside the REM function block is explained at the end of this POU.
GE
LE
CommandedPositionCyclic
IdealDecelPosition
R_TRIG_1
R_TRIG_2
R_TRIG
CLK
Q
OR
SlowNow

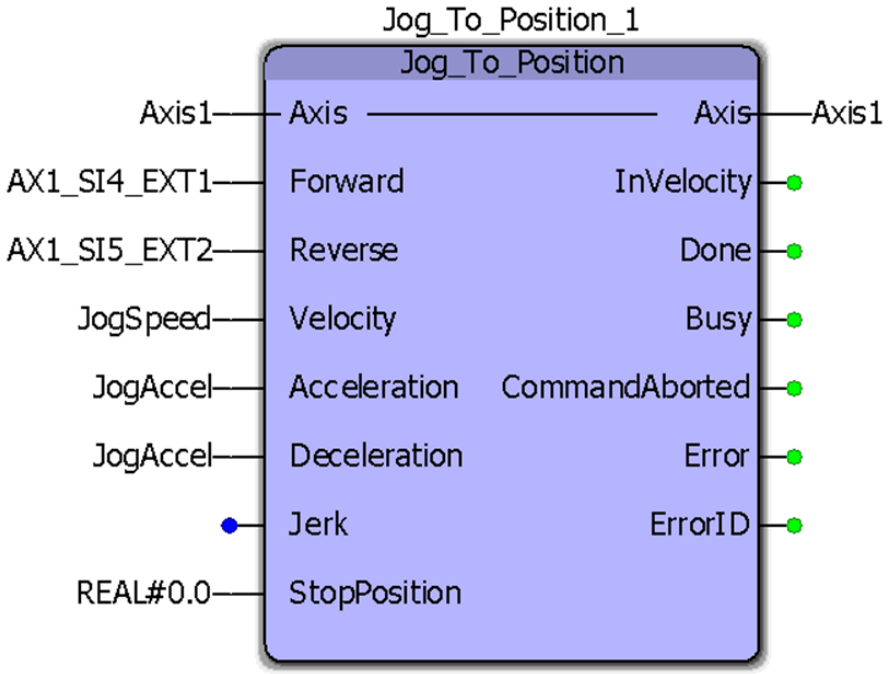


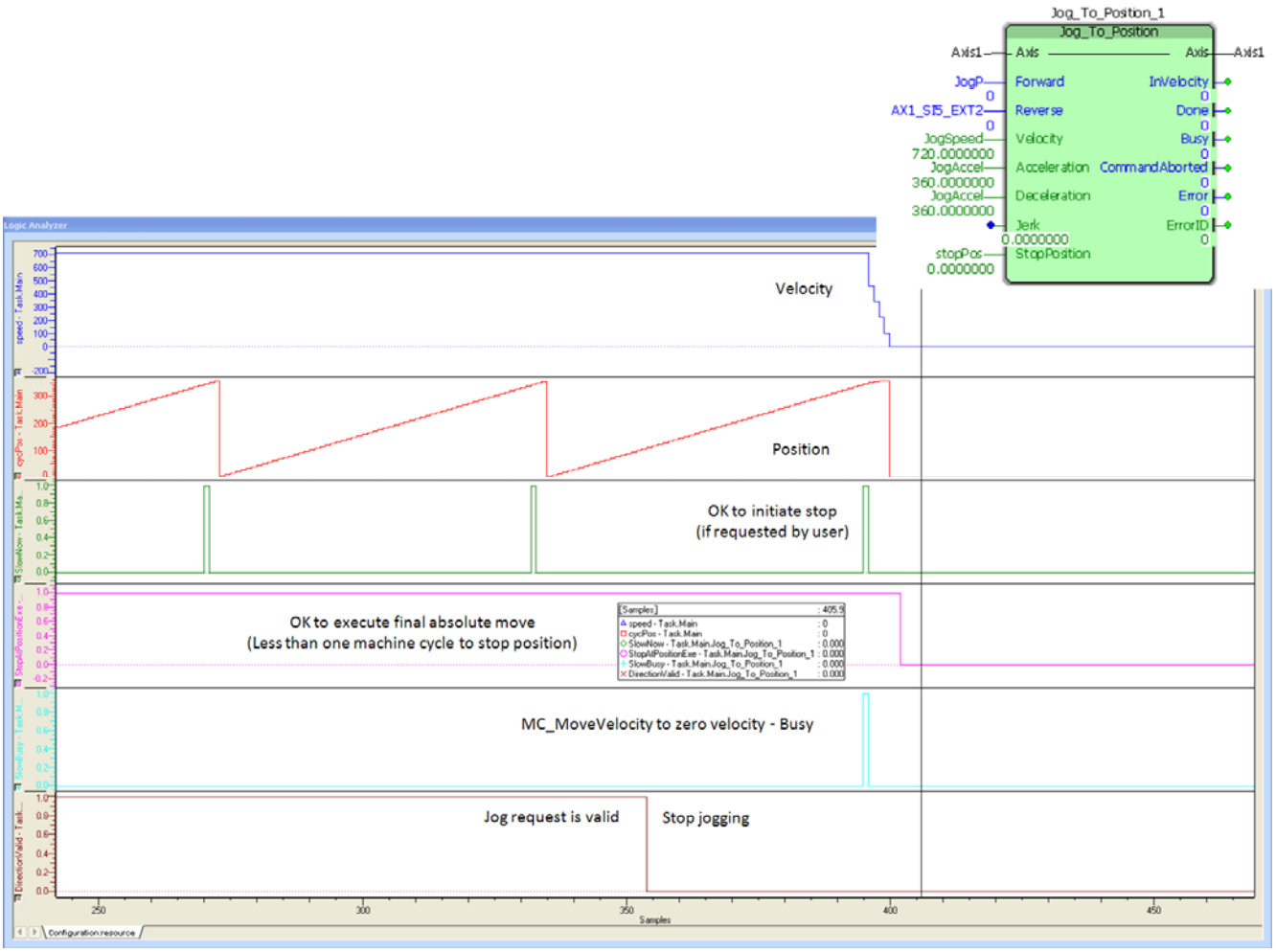

015
Active
RotaryType
MOVE
EN ENO
UINT#10060
ErrorID
Error
Velocity1Error
Velocity1ErrorID
Velocity2Error
Velocity2ErrorID
MoveAbsError
MoveAbsErrorID
016
017
MC_MoveAbsolute_1.
CommandAborted
CommandAborted
(*Set Busy Output*)
018
MC_MoveVelocity_1.Busy
MC_MoveVelocity_2.Busy
MC_MoveAbsolute_1.Busy
Busy
(*This function block will never hold the Done output on longer than one scan
because of the nature of the inputs (Forward _Reverse)
*)
019
StopDone
Done

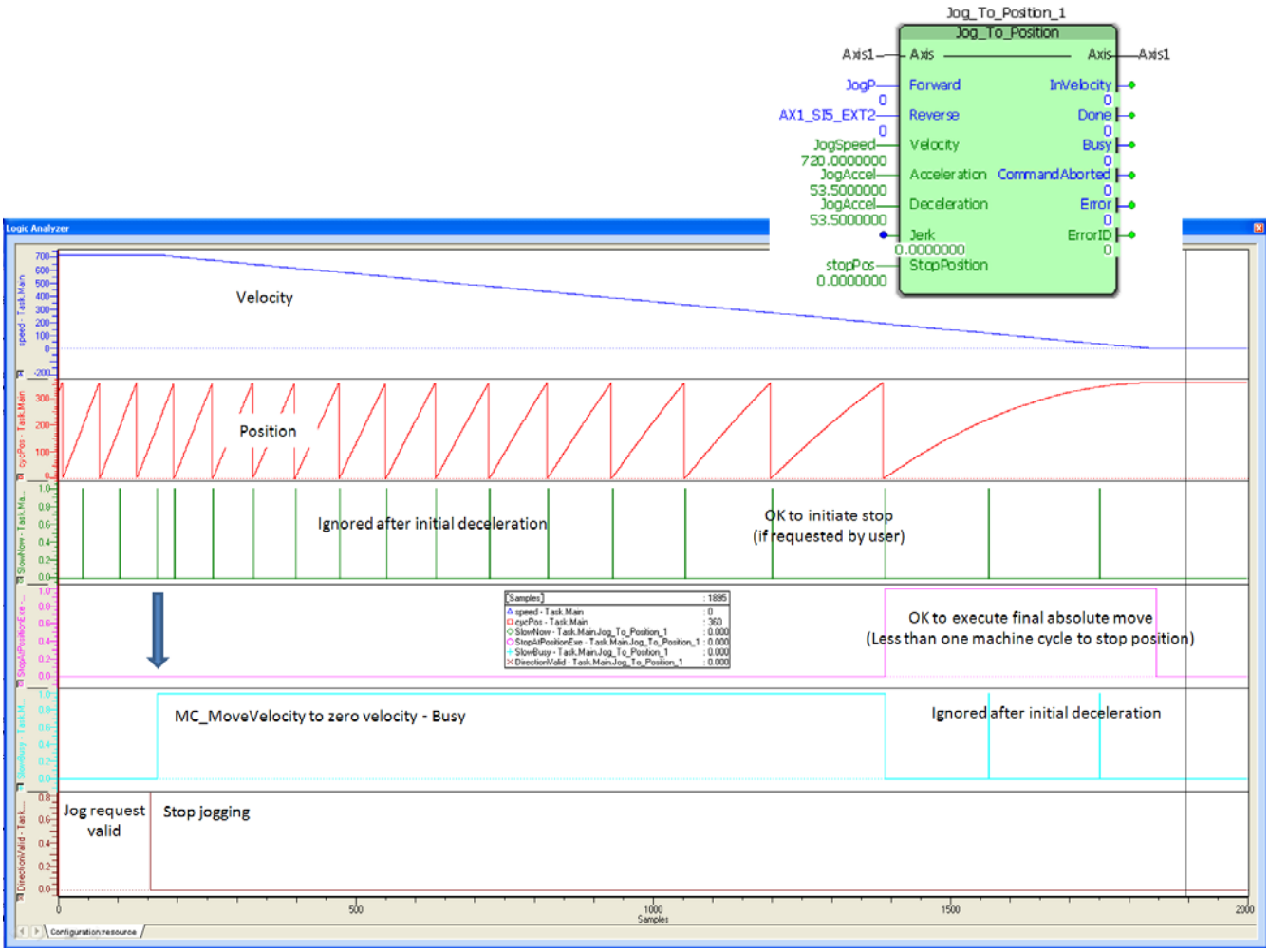
```
(* The purpose of the REM function block is to act like a MOD function, but for floating point values.*)

FUNCTION_BLOCK REM
    VAR_INPUT
        Denominator :        REAL;
        Numerator :          REAL;
    END_VAR

    VAR_OUTPUT
        REM :                REAL;
    END_VAR

    VAR
        IntegerResult :      DINT;
    END_VAR

    IF Denominator <> REAL#0.0 THEN
        IntegerResult := TRUNC_DINT(Numerator / Denominator);
        REM := Numerator - (Denominator * DINT_TO_REAL(IntegerResult));
    ELSE
        REM := REAL#0.0;
    END_IF;

END_FUNCTION_BLOCK;
```

2.6.1. Application Example using Jog_To_Positon

This function block allows a rotary axis to jog indefinitely, then to be decelerated and stopped at a specific angular position.

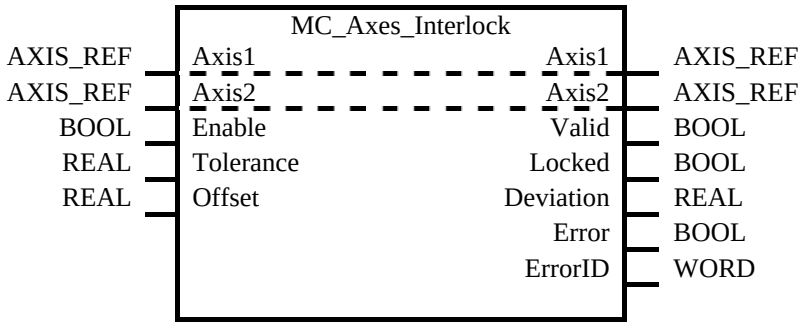



Jog_To_Position_1
Jog_To_Position
Axis1
Axis
Axis
Axis1
JogP
0
Forward
InVelocity
0
AX1_SI5_EXT2
0
Reverse
Done
0
JogSpeed
720.0000000
Velocity
Busy
0
JogAccel
360.0000000
Acceleration
CommandAborted
0
JogAccel
360.0000000
Deceleration
Error
0
Jerk
0.0000000
ErrorID
0
stopPos
0.0000000
StopPosition
Velocity
Position
OK to initiate stop
(if requested by user)
OK to execute final absolute move
(Less than one machine cycle to stop position)
MC_MoveVelocity to zero velocity - Busy
Jog request is valid
Stop jogging
Samples


Jog_To_Position_1
Jog_To_Position
Axis1
Axis
Axis
Axis1
JogP
0
Forward
InVelocity
0
AX1_SI5_EXT2
0
Reverse
Done
0
JogSpeed
720.0000000
Velocity
Busy
0
JogAccel
53.5000000
Acceleration
CommandAborted
0
JogAccel
53.5000000
Deceleration
Error
0
Jerk
0.0000000
ErrorID
0
stopPos
0.0000000
StopPosition
Logic Analyzer
Velocity
Position
Ignored after initial deceleration
OK to initiate stop
(if requested by user)
OK to execute final absolute move
(Less than one machine cycle to stop position)
MC_MoveVelocity to zero velocity - Busy
Ignored after initial deceleration
Jog request
valid
Stop jogging
Samples

2.7. Axes Interlock

FB-Name	**MC_Axes_Interlock**	
This User Derived Function Block monitors two axes which are mechanically linked to ensure that the position of both axes is within specified tolerances, and that both axes are free from alarms. The Locked output provides a way to confirm these conditions.		
VAR_IN_OUT		
Axis1	AXIS_REF	Reference to the axis
Axis2	AXIS_REF	Reference to the axis
VAR_INPUT		
Enable	BOOL	The function block will continuously monitor the two axes and provide output while enable is held high.
Tolerance	REAL	Specifies the maximum allowable position difference between the two axes before the ‘ Locked’ output should become FALSE.
Offset	REAL	Specify any intentional position offset to be ignored when comparing the position of Axis1 and Axis2
VAR_OUTPUT		
Valid	BOOL	Indicates that the function is enabled and there are no internal errors preventing it from providing its output.
Locked	BOOL	Indicates that both axes are within the specified position Tolerance and that neither axis has an alarm.
Deviation	REAL	The positional difference between the two axes.
Error	BOOL	Signals that an error has occurred within the Function Block
ErrorID	UINT	Error identification
Note:		



Textual representation

Variable Declaration:

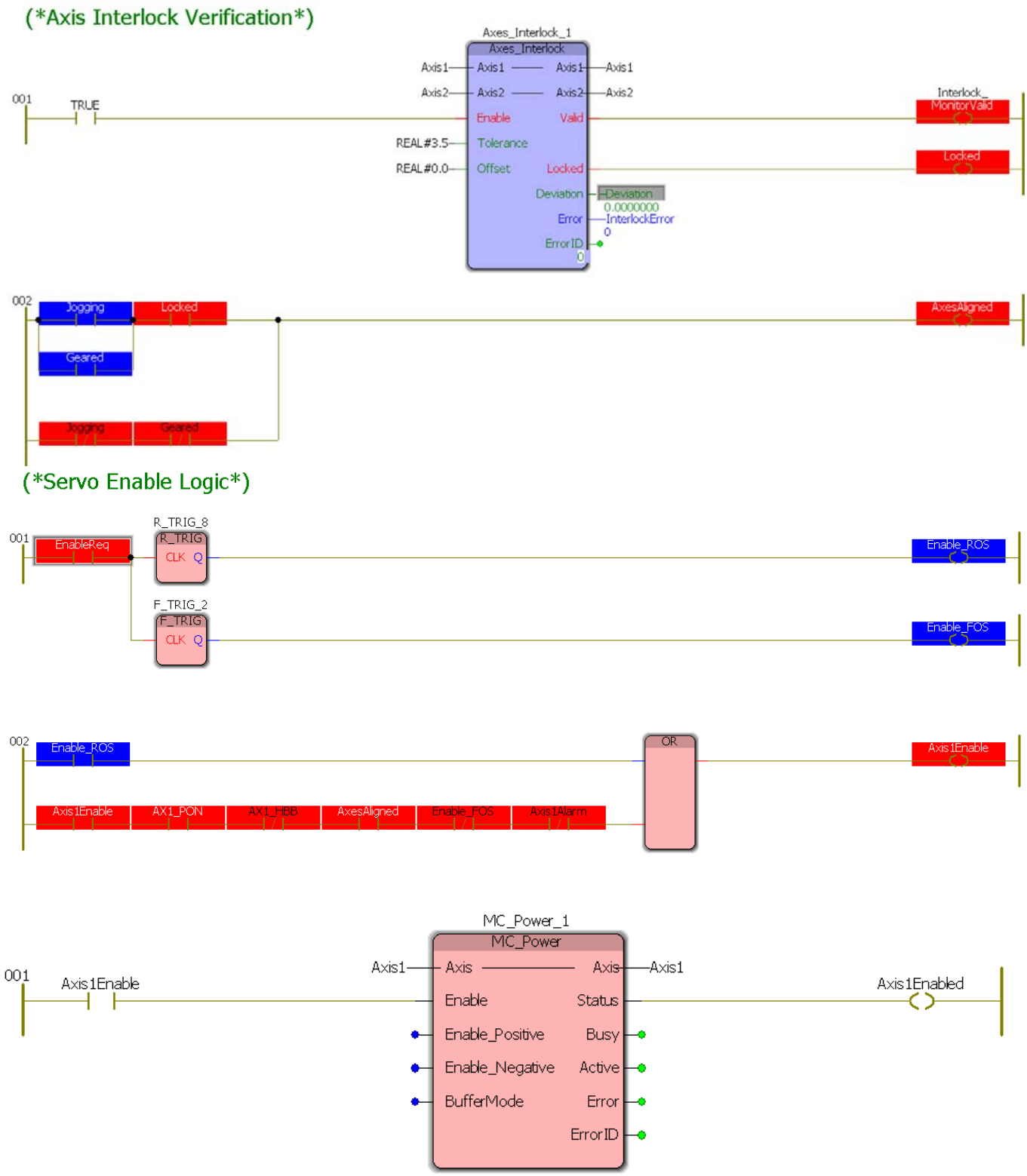
```
FUNCTION_BLOCK Axes_Interlock
    VAR_INPUT
        Enable :                BOOL;
        Tolerance :             REAL;
        Offset :                REAL;
    END_VAR

    VAR_IN_OUT
        Axis1 :                 AXIS_REF;
        Axis2 :                 AXIS_REF;
    END_VAR

    VAR_OUTPUT
        Valid :                 BOOL;
        Locked :                BOOL;
        Deviation :             REAL;
        Error :                 BOOL;
        ErrorID :               WORD;
    END_VAR

    VAR
        Active :                BOOL
```

```
        Axis1Error :                BOOL
        Axis2Error :                BOOL
        AxisConfigError :           BOOL
        AxisDefError :              BOOL
        CycleError :                BOOL
        DrivesOK :                  BOOL
        MachineCycle :              REAL
        MC_ReadActualPositionNC_1 : MC_ReadParameter
        MC_ReadActualPositionNC_2 : MC_ReadParameter
        MC_ReadAxisError_1 :        MC_ReadAxisError
        MC_ReadAxisError_2 :        MC_ReadAxisError
        MC_ReadBoolParameter_1 :    MC_ReadBoolParameter
        MC_ReadBoolParameter_2 :    MC_ReadBoolParameter
        MC_ReadParameter_1 :        MC_ReadParameter
        MC_ReadParameter_2 :        MC_ReadParameter
        PositionOK :                BOOL
        R_TRIG_Enable :             R_TRIG
        Read1Error :                BOOL
        Read2Error :                BOOL
        ReadBoolPrmError :          BOOL
        ReadPrmError :              BOOL
        RotaryAxis :                BOOL
    END_VAR


(* define the internal code here *)

(* This function was designed to monitor axes that are mechanically coupled together or otherwise must move
   together within critical tolerances. It compares the actual position deviation to a tolerance input.
(*  It also monitors MC_ReadAxisAlarm of each axis specified. This block simply supplies an output to
   indicate if the axes are with the specified tolerance and neither servo has an alarm.

   The application program must then use the output to interlock the servos from enabling.

   A note about the offset input: a positive value means it's expected that Axis1 is AHEAD of Axis2 *)


(*******************************       Event Handling Section     *******************************)
(* Never put R_TRIG / F_TRIG functions under conditional logic such as IF statements *)

R_TRIG_Enable(CLK:=Enable);                         (*  For capturing the rising edge of Enable  *)


(* This line causes the function block to exit if the execute, and all outputs are off. (For efficiency.)
   IMPORTANT, be sure to include CommandAborted in the interlock logic if a motion block will be used. *)

IF NOT(Enable) AND NOT(Active) THEN RETURN;
END_IF;


Active:= Enable OR Valid OR Error;
(* This keeps the function executing while enable held on or an internal state is still busy  *)


(*******************************       Initialization Section     *******************************)
IF R_TRIG_Enable.Q THEN
        AxisConfigError := FALSE;
        ReadPrmError := FALSE;
        CycleError := FALSE;
        RotaryAxis := FALSE;
        MachineCycle := REAL#1.0;

        (*  Axes must both be defined  *)
        AxisDefError := (Axis1.AxisNum = UINT#0) OR (Axis2.AxisNum = UINT#0);

        IF NOT AxisDefError THEN

                (*  Determine axis configuration for Axis1  (Rotary or Linear?  *)

                (*  Add other error checking code here  *)
                (*  For example, if one axis is configured as a rotary type, verify both axes have the same
                    configuration and Machine Cycle  *)

                (*  Add other initialization code here  *)

                (*  For example, if rotary axes, read the machine cycle as a parameter or create another
                    function block input  *)

        END_IF;
```

```
        (*  Set Error flag here to prevent the main section from running even once is there is a problem  *)
        (*  "OR" any other errors generated in the initialization section  *)
        Error:=AxisDefError;
END_IF;


(*******************************       Main Operation Section     *******************************)
IF Enable AND NOT Error THEN

    (****  Compare Positions of the two mechanically interlocked axes  ****)
    MC_ReadActualPosition_1(Axis := Axis1, Enable := TRUE);
    Read1Error := MC_ReadActualPosition_1.Error;

    MC_ReadActualPosition_2(Axis := Axis2, Enable := TRUE);
    Read2Error := MC_ReadActualPosition_2.Error;

    Deviation := ((MC_ReadActualPosition_1.Value - Offset) - MC_ReadActualPosition_2.Value);
    PositionOK := ABS(Deviation) <= Tolerance;

    (****  Check for alarms on the two mechanically interlocked axes  ****)
    MC_ReadAxisError_1(Axis := Axis1, Enable := TRUE);
    Axis1Error := MC_ReadAxisError_1.Error;

    MC_ReadAxisError_2(Axis := Axis2, Enable := TRUE);
    Axis2Error := MC_ReadAxisError_2.Error;

    DrivesOK := (MC_ReadAxisError_1.ErrorClass = UINT#0) AND (MC_ReadAxisError_2.ErrorClass = UINT#0);

    Locked := PositionOK AND DrivesOK;
ELSE
    Locked := FALSE;
    Deviation := REAL#0.0;
END_IF;

(*******************************       Error Handling Section     *******************************)
Error := Enable AND (Error OR AxisDefError OR ReadBoolPrmError OR Read1Error OR
                     Read2Error OR Axis1Error OR Axis2Error);
IF Error THEN
        IF AxisDefError THEN ErrorID := UINT#4625; END_IF;
        IF Read1Error THEN ErrorID := MC_ReadActualPosition_1.ErrorID; END_IF;
        IF Read2Error THEN ErrorID := MC_ReadActualPosition_2.ErrorID; END_IF;
        IF Axis1Error THEN ErrorID := MC_ReadAxisError_1.ErrorID; END_IF;
        IF Axis2Error THEN ErrorID := MC_ReadAxisError_2.ErrorID; END_IF;
        (*  Add other errors if necessary  *)
ELSE
        ErrorID := UINT#0;
END_IF;

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

2.7.1. Application Example using Axes Interlock.

This function block is for use when two servos are operating the same mechanical load, and must remain in operation simultaneously. The Axes_Interlock.Locked output is used as part of the logic sequence for each axes MC_Power function block.



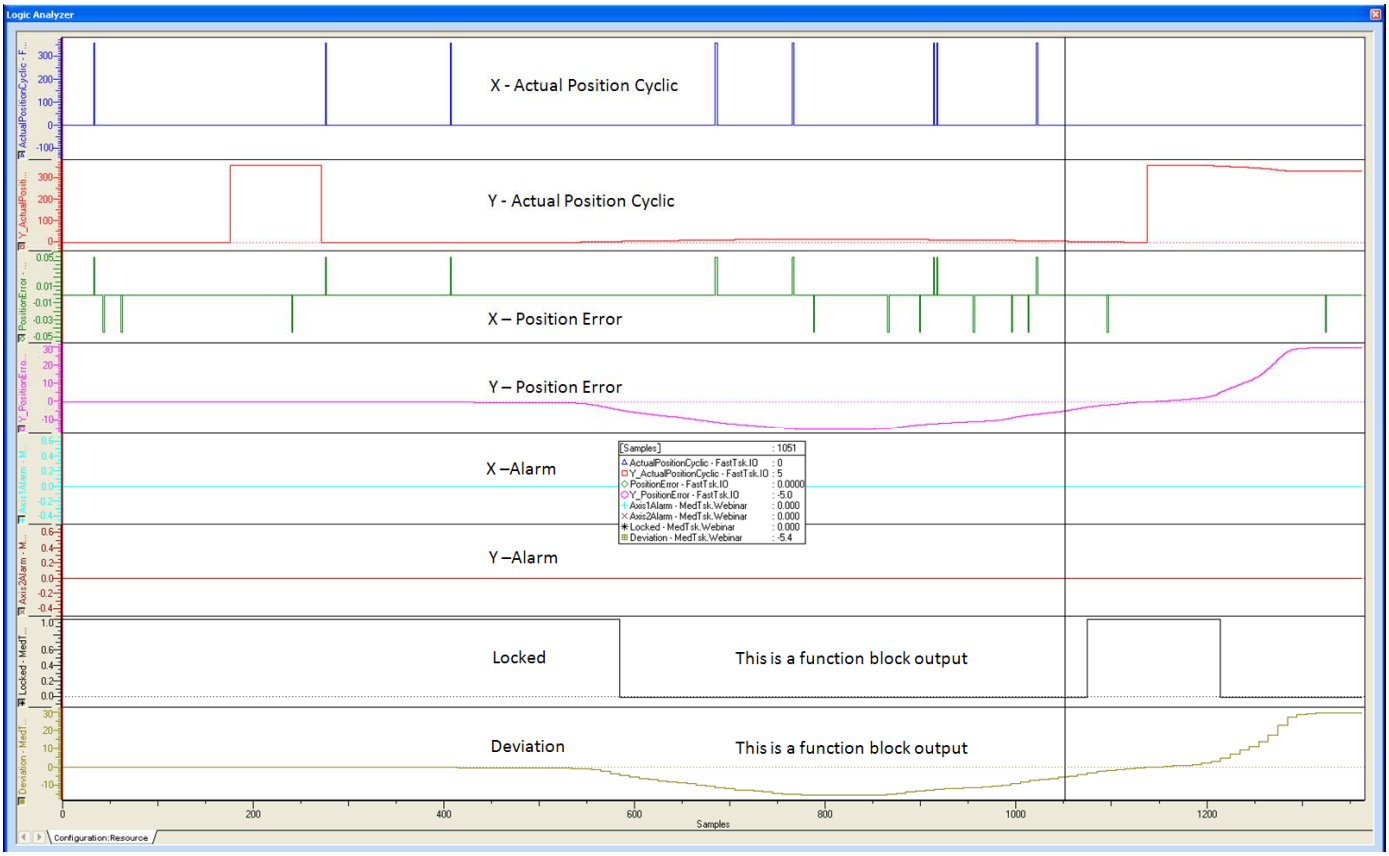
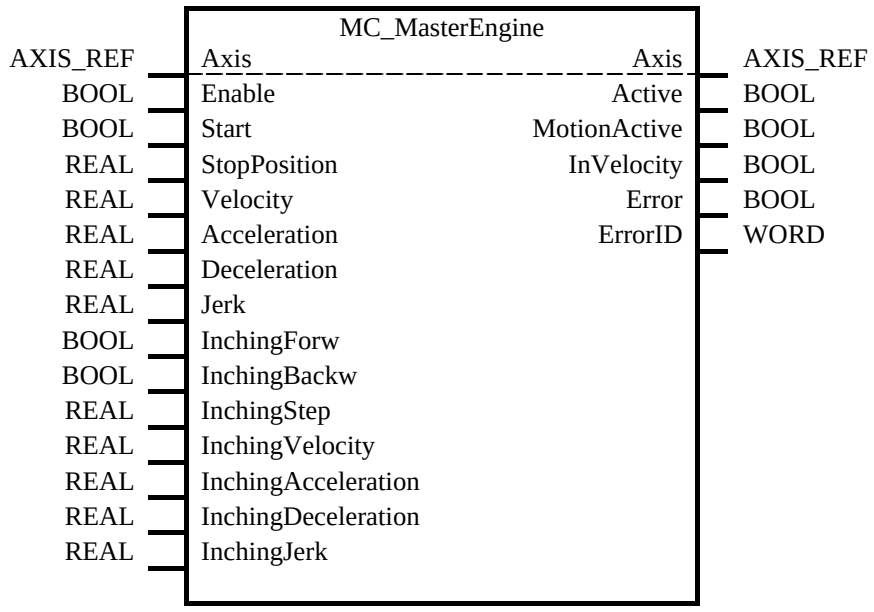


Figure 9 - Programming example of Axes Interlock

2.8. Master Engine

It can be practical to create a Function Block, using the PLCopen Motion Control Function Blocks (FB's) and IEC 61131-3 FB's, representing the classical mechanically coupled master axis in a machine in a virtual way. The other axes can be coupled to this virtual master axis for synchronization. This example consists of multiple parts, starting with the representation of the Function Block itself.

FB-Name	MC_MasterEngine	
The FB is used for driving the virtual Master Axis in packaging machine. It behaves like a real mechanical main shaft. It runs at a predetermined velocity, can be stopped at certain positions, it has an inching mode for startup or run in, and serves as a Master for the Slave Axes which are connected to Cam and Gear functionalities.		
VAR_IN_OUT		
Axis	AXIS_REF	Reference to the axis
VAR_INPUT		
Enable	BOOL	Enables and initializes the FB
Start	BOOL	Start the motion at rising edge, Stop on falling edge at 'StopPosition'
StopPosition	REAL	Stop position [u]
Velocity	REAL	Velocity for continuous motion [u/s]
Acceleration	REAL	Acceleration for continuous motion [u/s^2]
Deceleration	REAL	Deceleration for continuous motion [u/s^2]
Jerk	REAL	Jerk for continuous motion [u/s^3]
InchingForw	BOOL	Input to start Inching in the forward direction (Axis stops, if input reset before reaching 'InchingStep')
InchingBackw	BOOL	Input to start Inching in the backward direction (Axis stops, if input reset before reaching 'InchingStep')
InchingStep	REAL	Maximum distance for inching [u]
InchingVelocity	REAL	Velocity for inching [u/s]
InchingAcceleration	REAL	Acceleration for inching [u/s^2]
InchingDeceleration	REAL	Deceleration for inching [u/s^2]
InchingJerk	REAL	Jerk for inching [u/s^3]
VAR_OUTPUT		
Active	BOOL	FB enabled
MotionActive	BOOL	Motion active
InVelocity	BOOL	Set Velocity reached (continuous motion)
Error	BOOL	Signals that an error has occurred within the Function Block
ErrorID	WORD	Error identification
Notes: The inching mode is a manual controlled mode with a movement over a pre-defined distance ('InchingStep'). It is stopped immediately when the related input is reset.		



Below a graphical example of the 'Start' – 'Stop' behavior:

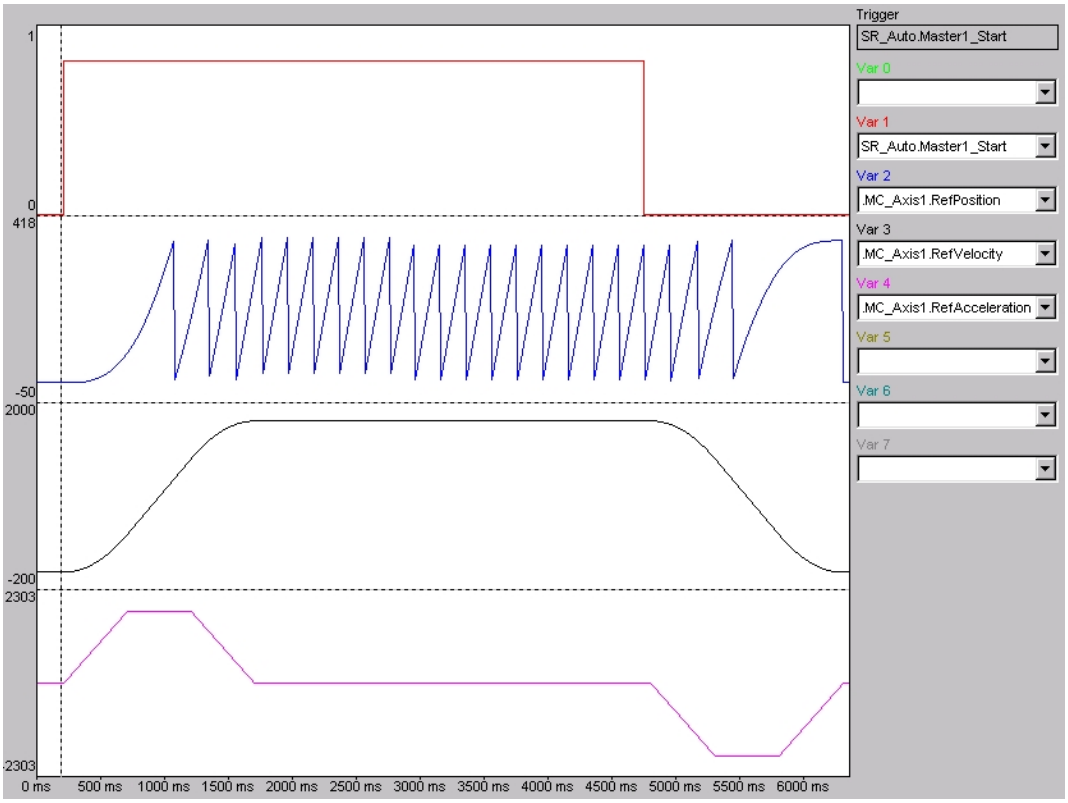


Figure 10 - 'Start'-'Stop' behavior of MC_MasterEngine

Explanation:

- The first channel (on top) shows the 'Start' input.
- The second channel shows the position of the axis
- Channel 3 shows the velocity of the axis
- Channel 4 shows the acceleration of the axis

This second example deals with inching (slowly moving) over a complete ‘InchingStep’

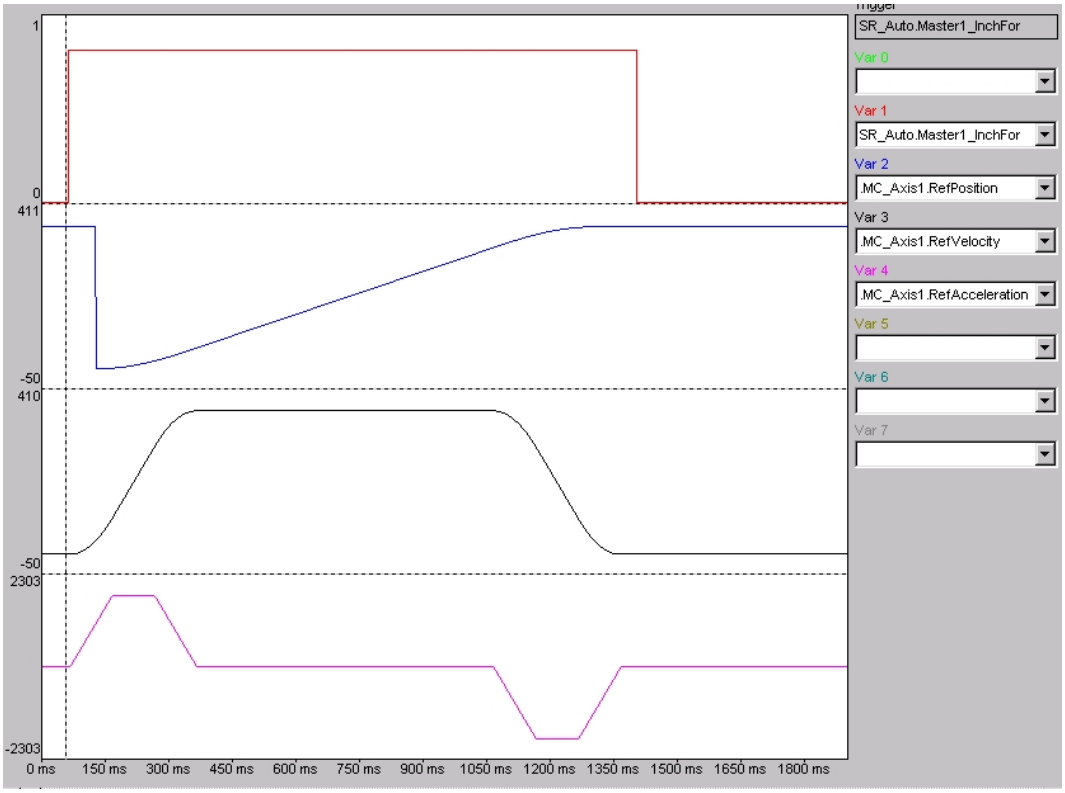


Figure 11 - Inching with a complete ‘InchingStep’

Explanation:

- The first channel (on top) shows the ‘InchingForw’ input.
- The second channel shows the position of the axis
- Channel3 shows the velocity of the axis
- Channel4 shows the acceleration of the axis

2.8.1. Program example for the use of MC_MasterEngine

Axis1 is the virtual Master; Axis5 is coupled via Gear to the Master.

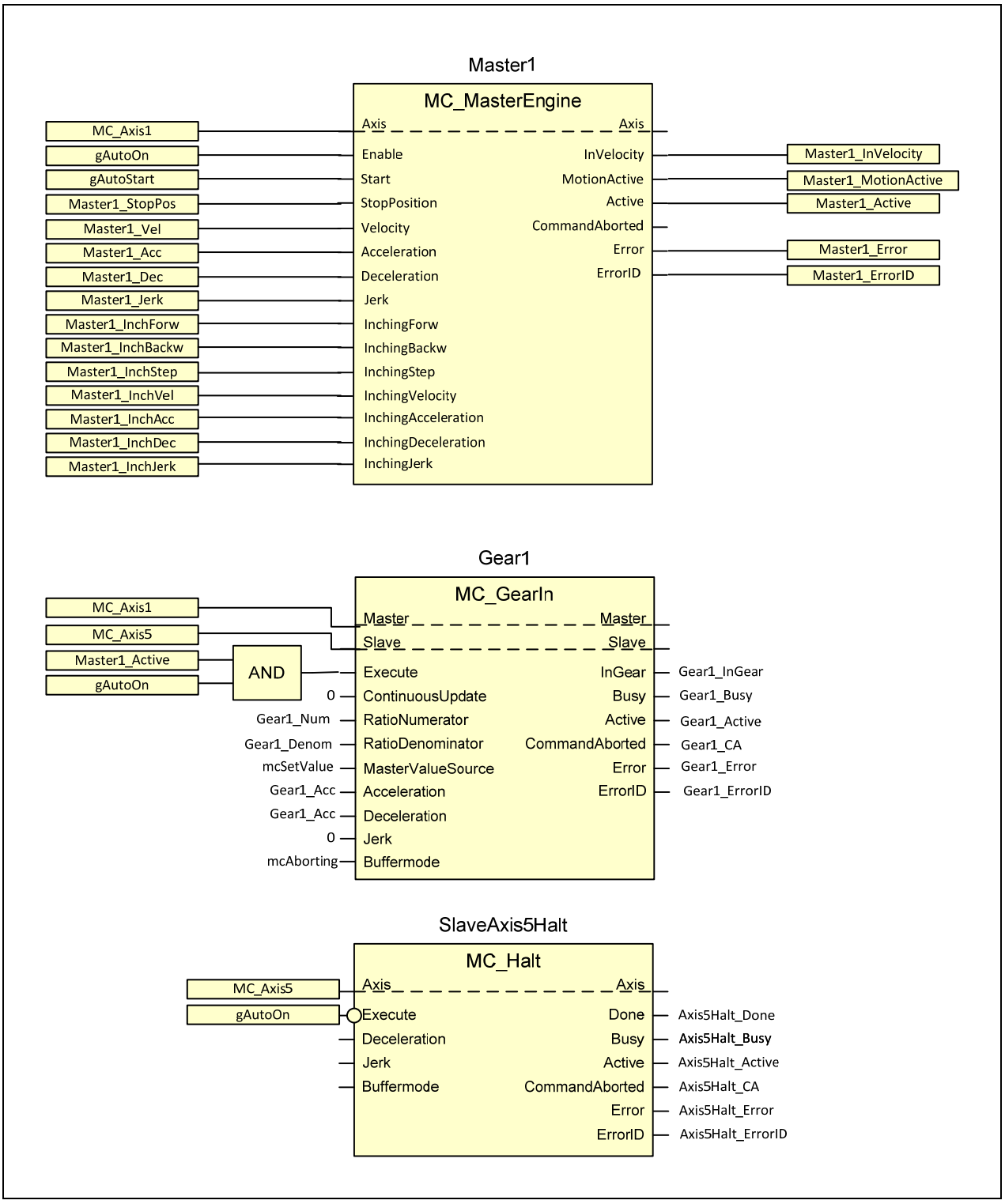


Figure 12 - Program example for the use of MC_MasterEngine

2.8.2. The inside of the Function Block MC_MasterEngine

The content of the User derived FB MC_MasterEngine consists of two sections as described here.

The first part of the FB generates the continuous motion and calculates the 'StopPosition' in a way, that the axis always

maintains its ‘Deceleration’.

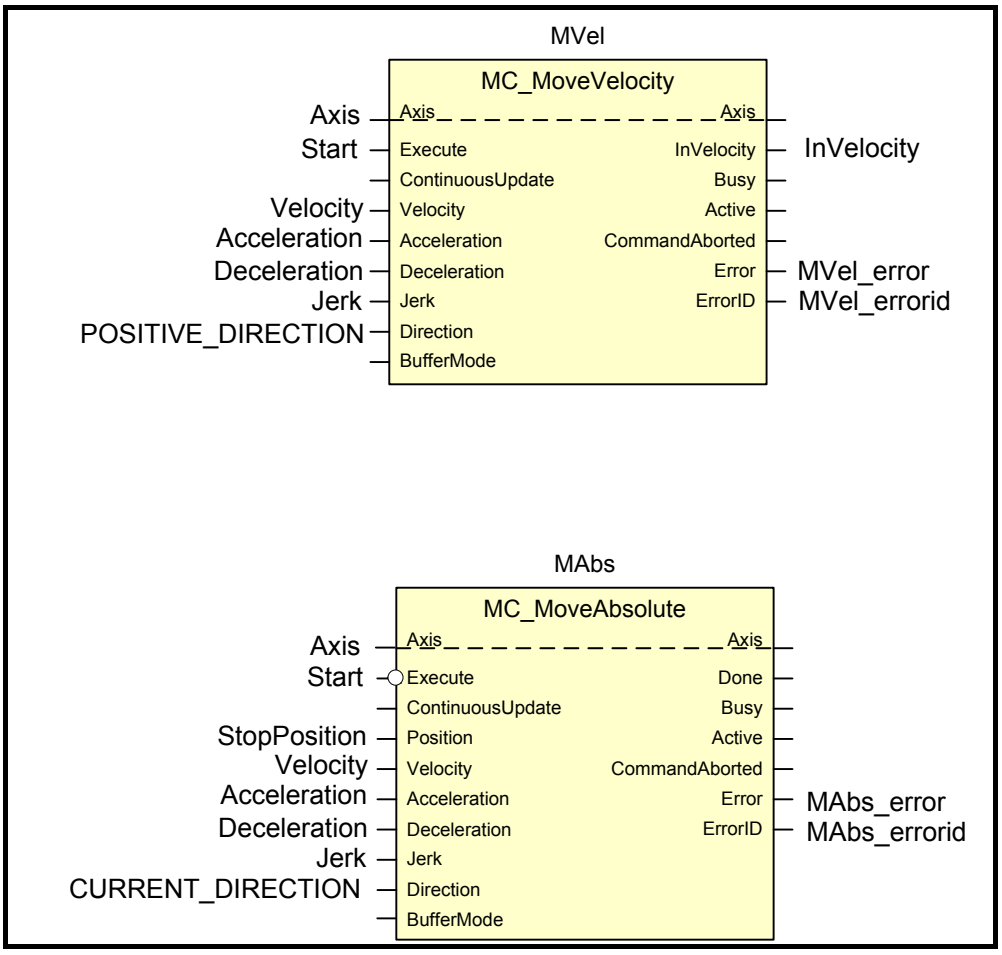


Figure 13 - The first part of the FB MC_MasterEngine

This next part is dedicated to the inching mode.

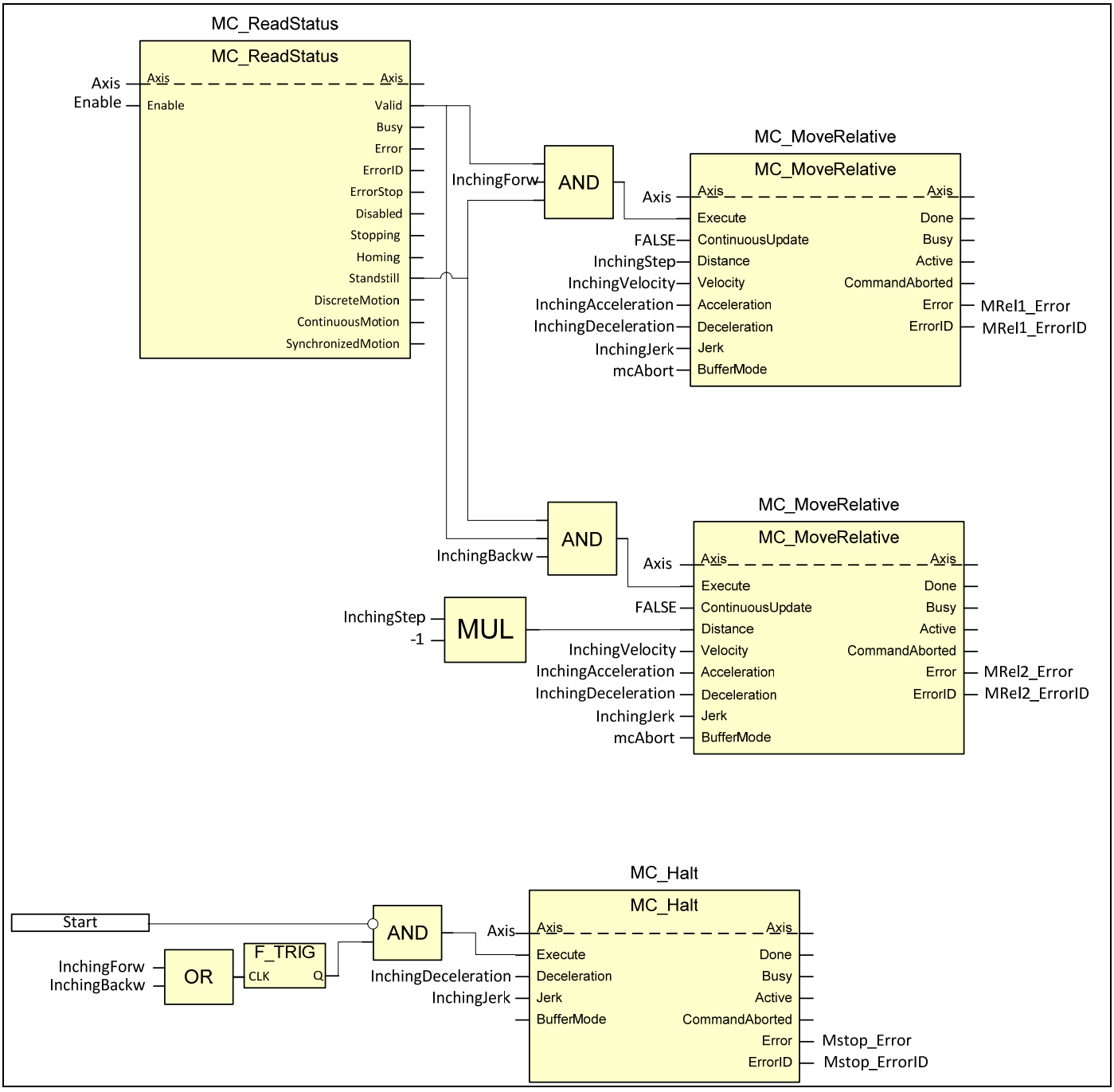


Figure 14 - The second part of the FB MC_MasterEngine for inching

2.9. Explanation of Camming in combination with MC_MasterEngine

In Part 1, the following Function Blocks are defined for the Camming functionality:

1. MC_CamTableSelect
2. MC_CamIn
3. MC_CamOut

The principle Tasks of MC_CamTableSelect:

- Initialization of one ‘CamTable’ or a whole set of CAM tables for MC_CAM_REF. For this operation a reference to the master or slave axis is not always necessary;
- Selection of one curve within MC_CAM_REF for MC_CamIn. For this operation a reference to the master and slave axis is not always necessary;
- The whole process of preparing the ‘CamTable’ (ID) switch (selection). For this operation a reference to the master and slave axis is necessary.

The generation of the ‘CamTable’ can be done via an independent software tool in a preparation phase, or in real time by the controller. The table is generated in the MC_CAM_REF structure. The ‘CamTable’ to be processed is identified with the ‘CamTableID’ output. The combination of it all is shown in the figures below, for basic use and extended use.

2.9.1. The Basic Use of MC_CamTableSelect

The call of MC_CamTableSelect and MC_CamIn are done in 2 separate tasks due to possible limitations of the system.

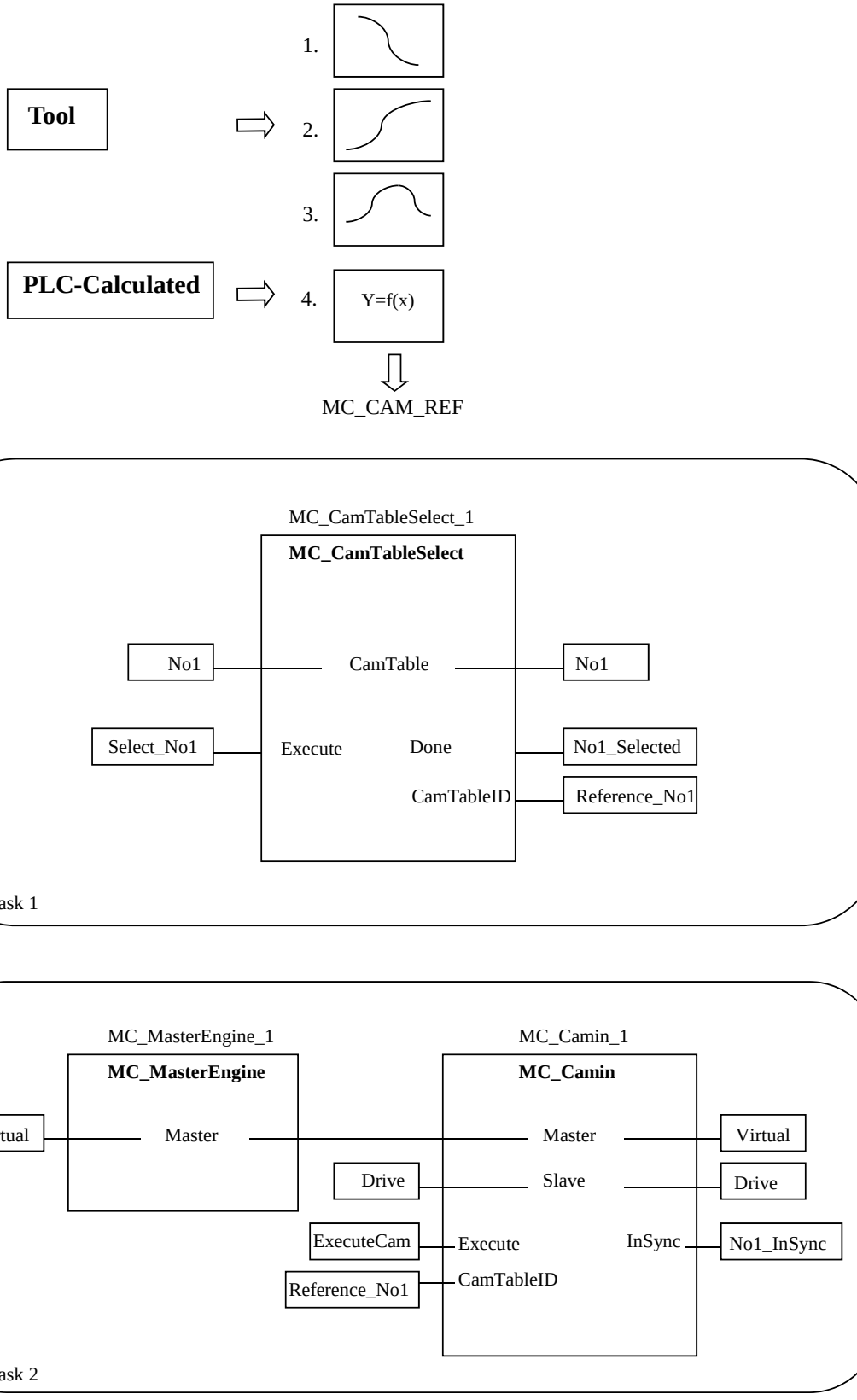


Figure 15 - Basic use of MC_CamTableSelect

2.9.2. The Extended Use of MC_CamTableSelect

In this example both the MC_CamTableSelect and MC_CamIn are called in the same task and cycle.

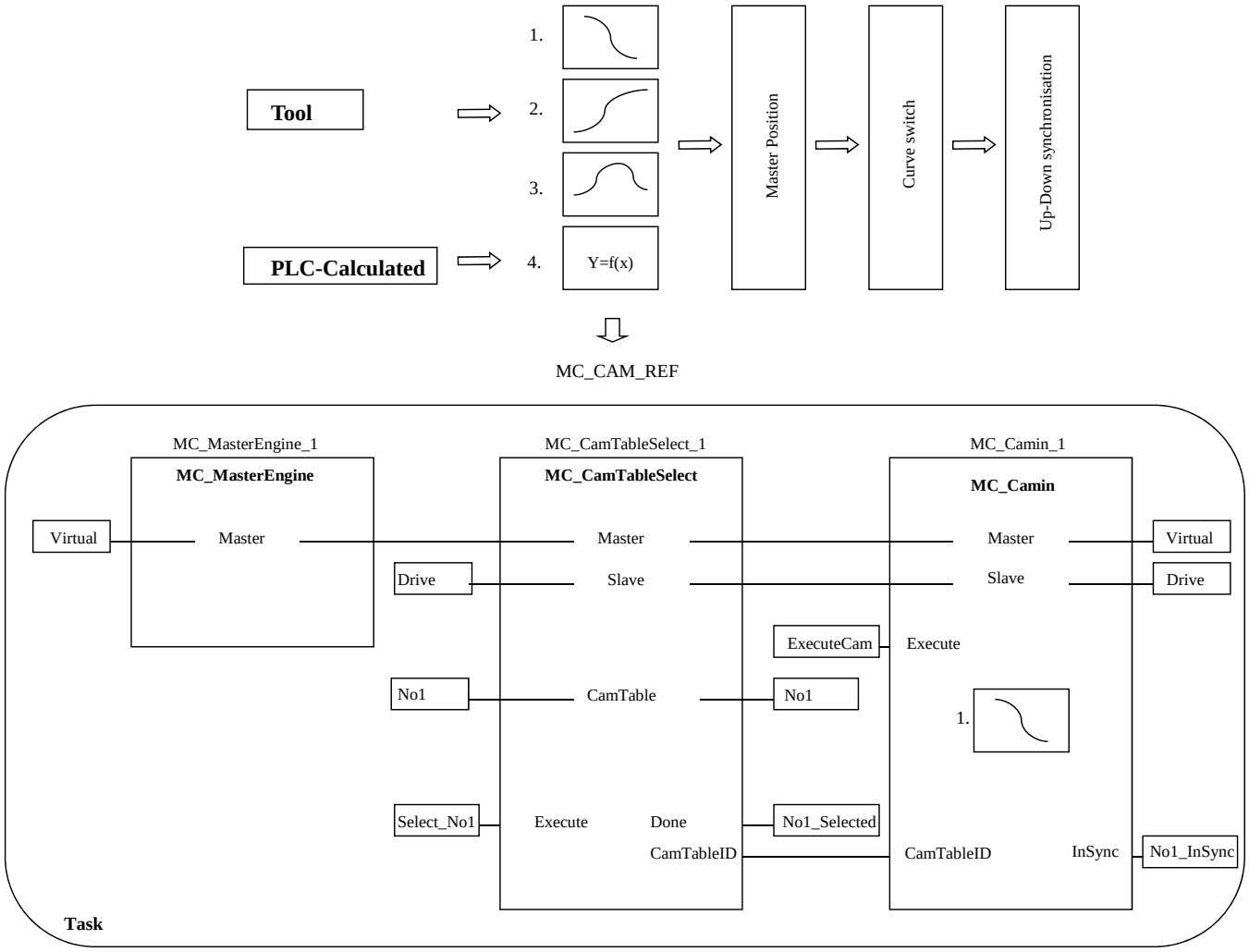


Figure 16 - Extended use of MC_CamTable_Select

2.10. Using three segments CAM profile

Many machine operations, or even sub operations, have a start sequence, a repetitive normal operation sequence, and a stopping sequence. For this reason it makes sense to combine these three phases in one function.
The basis consists of a virtual axis to which the other axes are linked via a CAM profile. This means that such a system has no time base, but works via position-position basis between the virtual master and the slave(s). In case of multiple slave axes, the sequences run in parallel. The position-based diagram describing the synchronized parallel execution of these functions is called the synchronization diagram (see Figure 17 - General three-segment cam profile for one coupled axis). The top graphics shows the virtual master axis position on the vertical axis, with a modulo 360 degrees. This is the common reference to all slave axes, servos and Digital Cam Switches (DCS). Because all elements are synchronized to the same master, they are all synchronized together. When the master travels a full cycle in the second CAM segment, like the distance [0-360), one production cycle has been produced. The distance [0-360) represents one production cycle.

2.10.1. General User-Derived Function Block (UDFB) – Three-segment Cam profile

Looking at the axis position profile on the synchronization diagram, we can see that the axis executes a profile that is composed of three “segments”. The connecting points between those segments are not fixed, i.e. their position may vary depending on the machine production parameters. Therefore each of these Cam profile can be a subset of a general “three-segments Cam profile”, which therefore makes sense to create. Each segment is a separate Cam profile, and the application program will need to take care that these cam profiles are properly buffered and executed one after the other, and continuously.
The following figures show the position profile of a general three-segment Cam profile, and an example of the corresponding logic. This logic can be embedded into a UDFB that be re-used and specialized for different axes.

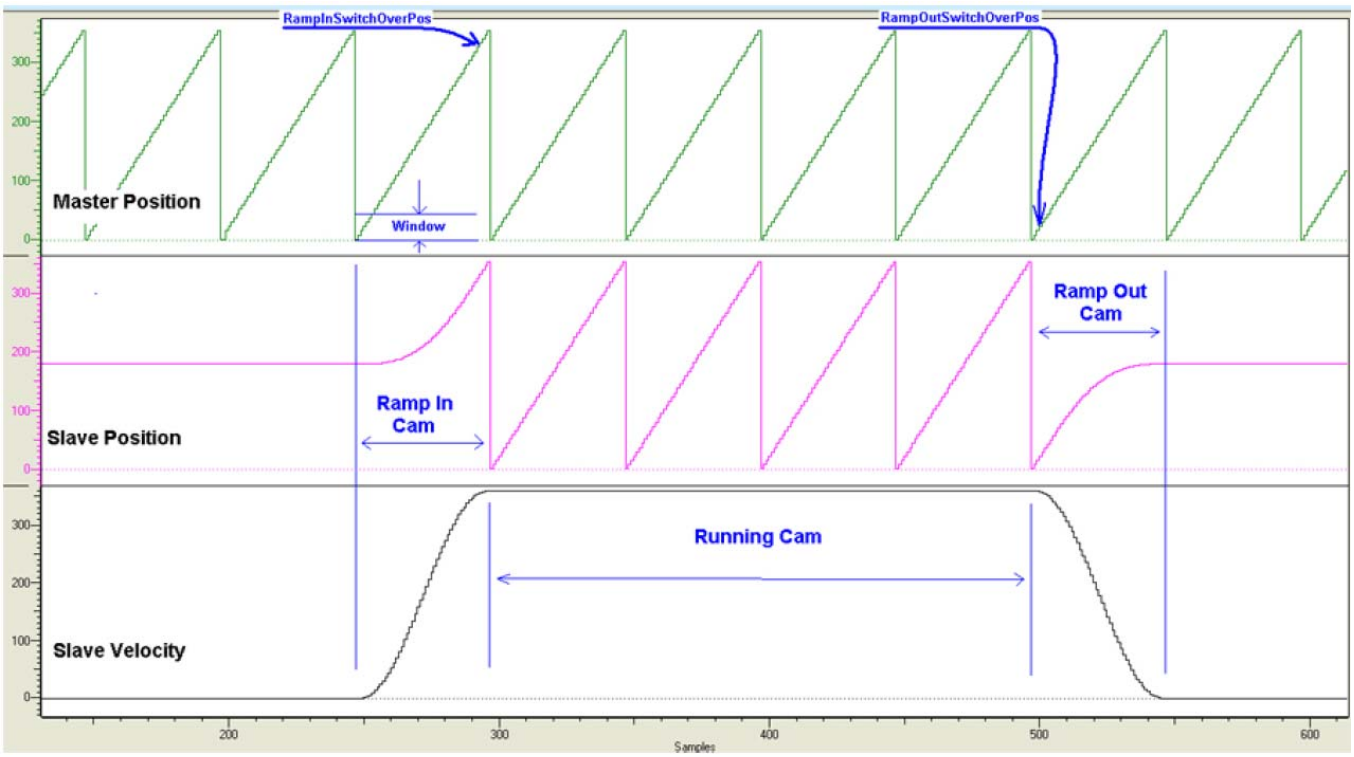


Figure 17 - General three-segment cam profile

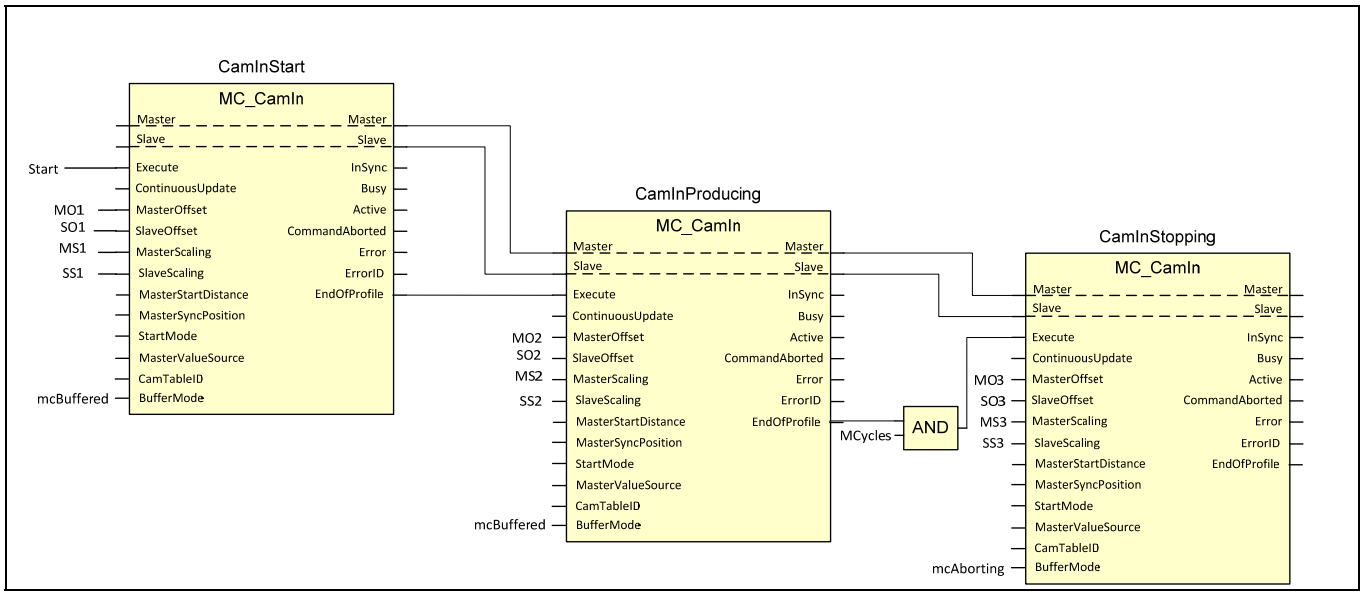


Figure 18 - Logic example for a general three-segment Cam profile

The parameter MCycles should be set in a different part of the program when the required number of production cycles is done. This is combined with the EndOfProfile to abort the CamInProducing FB with the CamInStopping FB.

These kind of segmented profiles are now used in the example on the Cut-to-Length example in Chapter 2.11 hereunder although the cyclic behavior is different.

2.11. Cut to length example

The following application example is material cut-to-length (e.g. plastic tube here) combined with an assembly round table. The plastic tube is un-winded, i.e. a certain length is pulled off, then cut and finally assembled in a round table. The figure below shows a simplified view of the machine. In this example also a 'three segment CAM profile' is used. However in this case the repetitive mode of the second CAM is only one cycle, and all three segments are combined in every production cycle.

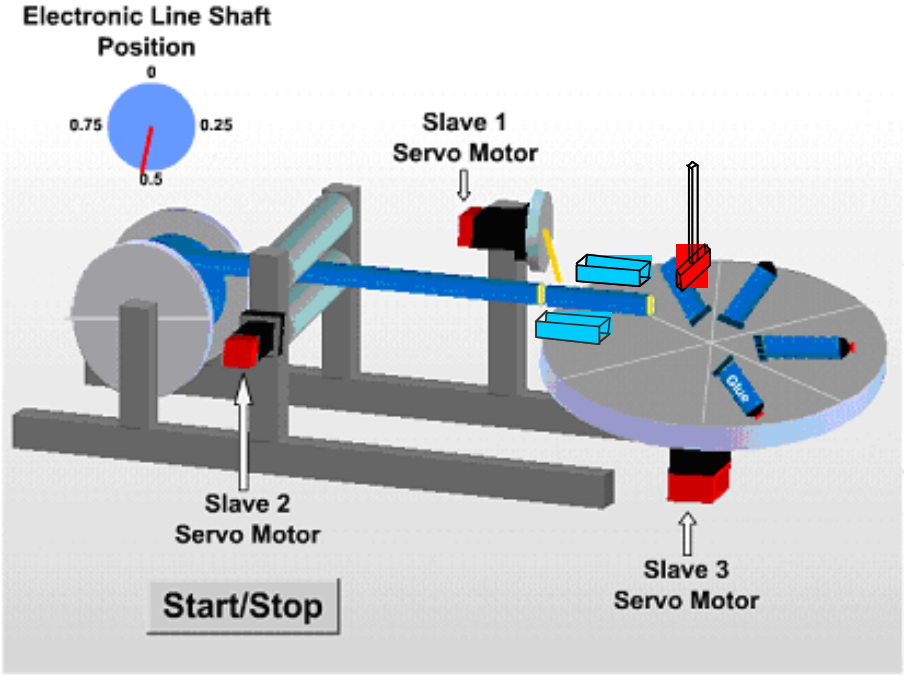


Figure 19 - Cut-to-length, round table machine

The different sequences of operation within a machine cycle are the following:

- Un-wind plastic tube, i.e. pull-off a certain tube length, and then stop
- Clamp the tube
- Cut the tube
- Un-clamp the tube (free fall on the round table)
- Turn the round table to move the tube to the first tool
- Run the first tool, e.g. to seal one extremity of the tube
- Turn the round table to move the tube to the second tool
- Run the second tool
- Further possible processing steps (not shown here)
- Product is finished

Three motion functions done with servomotors can be identified:

- Pulling axis
- Cutting axis
- Round-table axis

Several other functions, driven by digital outputs, can be identified. Each function corresponds to a tool in the assembly round table:

- Clamping / un-clamping
- Sealing
- Filling
- Capping
- Further possible processing steps (not shown here)

For the sake of simplicity we will only consider the first two ones.

The figure below shows a more detailed view of each element and breaks down the machine functionalities.

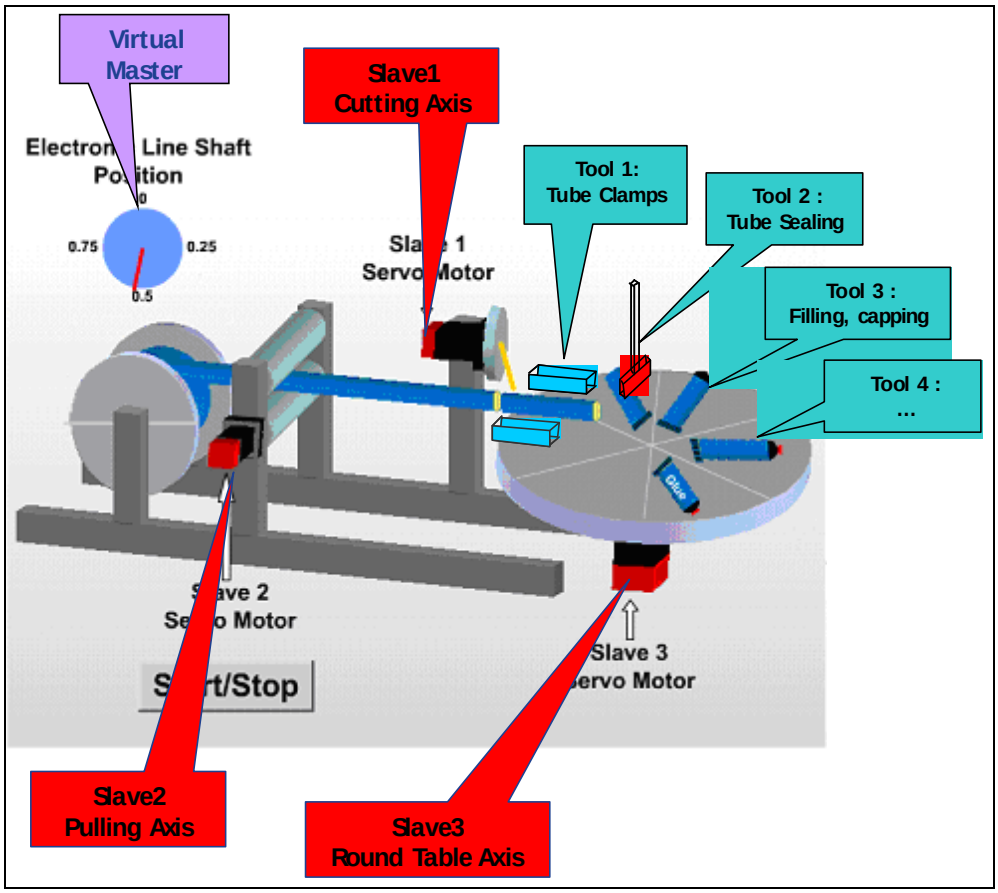


Figure 20 - Breakdown of machine functionalities

All the functions / steps described above could be run sequentially, i.e. respond to events or time, executed one after the other. However, in order to obtain higher machine throughputs, it is possible to run these functions partially in parallel, whenever possible. To do so, the functions cannot be run based on time or events, but on position. The position-based diagram describing the synchronized parallel execution of these functions is called the synchronization diagram.
A simplified synchronization diagram for the cut-to-length round table application is shown in the figure below:

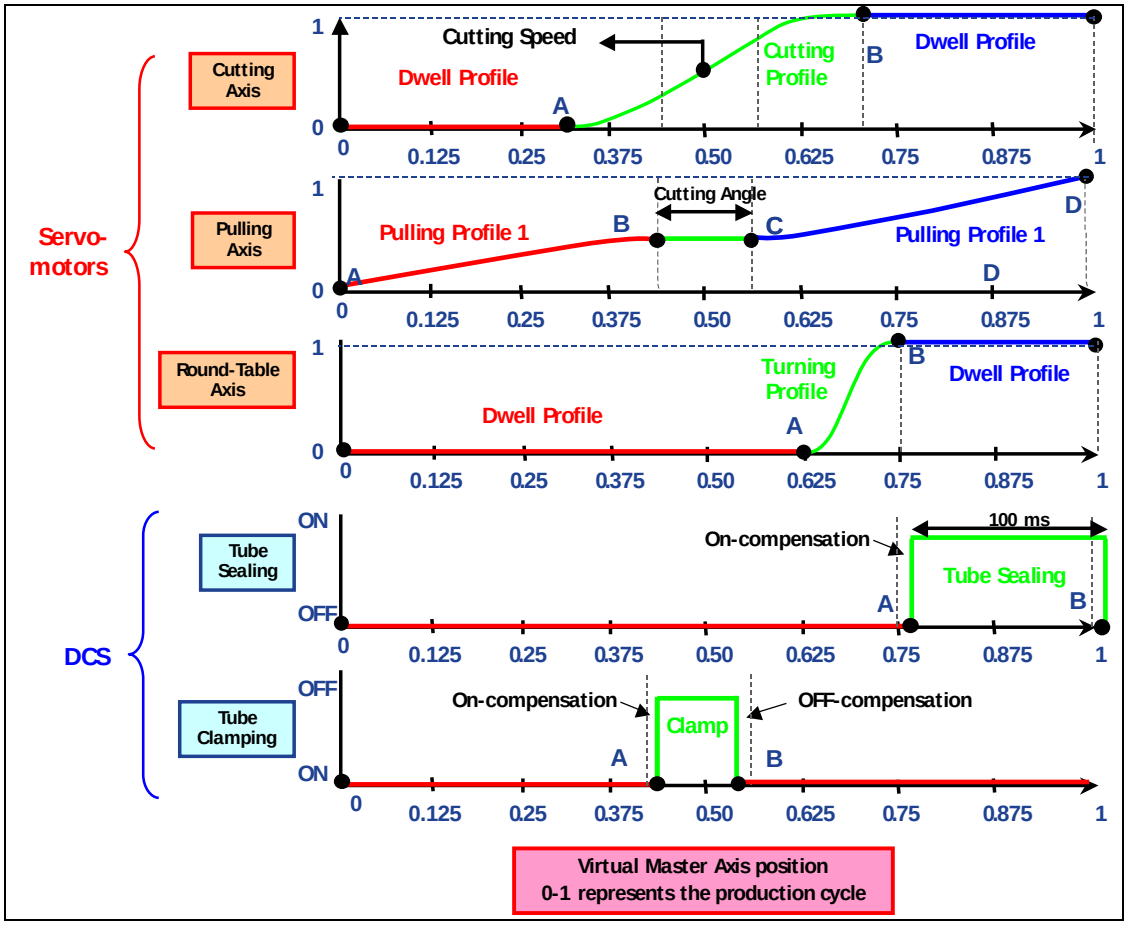


Figure 21 - Synchronization diagram cut-to-length round table machine

The three first plots describe the position of the servo-axes. The vertical coordinates are dimensionless, i.e. (0-1) represents the default distance the axes have to travel:

- Cutting axis: 1 represents one revolution of the knife
- Pulling axis: 1 represents the default tube length, e.g. 100mm
- Round-Table axis: 1 represents the default angle, e.g. 60 Deg.

The next two plots describe the state of the Digital Cam Switches (DCS) outputs, controlling the clamping tool and the sealing tool.

The horizontal coordinate is the virtual master axis position. This is the common reference to all five slave elements, servos and DCS. Because all elements are synchronized to the same master, they are all synchronized together.
When the master travels the distance (0-1), one product has been produced.
The distance (0-1) represents one production cycle.

2.11.1. Specialized User-Derived Function Block (UDFB) – Cutting axis Cam profile

With the general three-segment cam UDFB been created in Chapter 2.10 Using three segments CAM profile, it can be-re-used and specialized to fit to different axes. For example, to obtain the position profile of the cutting axis the specialization consists in setting the following MC_CamIn input:

- MO1 := 0 ; MO2 := 0 ; MO3 := 0 ; SO1 := 0 ; SO2 := 0 ; SO2 := 0 ;
- MS2 := Constant * MachineSpeed / CuttingSpeed ;
- MS1 := (1 – MS2)/2 ; MS3 := (1 – MS2)/2
- SS1 : = 0 ; SS1 : = 1 ; SS1 : = 0 ;

The Constant above depends on the rising Cam profile used the middle segment.

We have now a UDFB with reduced number of input parameters, and now directly linked to machine parameters.

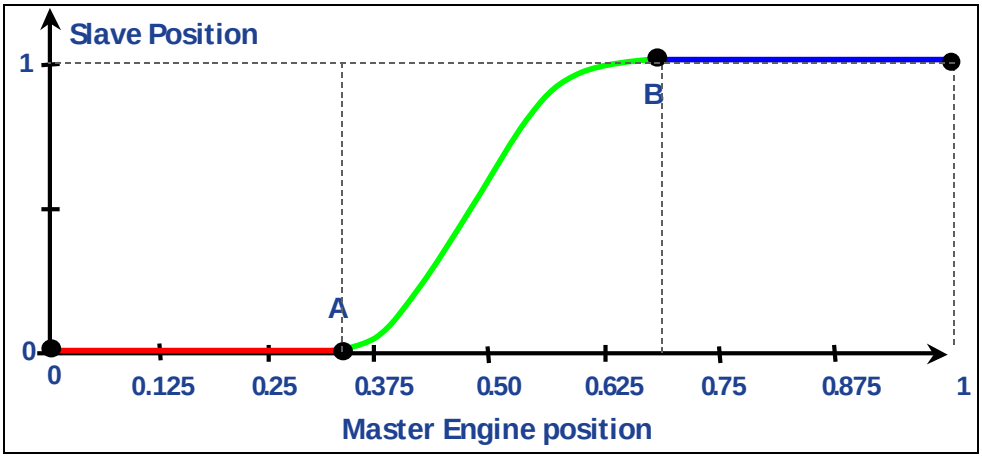


Figure 22 - Cutting axis three-segment cam profile

The following figure shows a Ladder logic view of the general three-segment cam profile UDFB and the specialized cutting-axis three-segment cam profile UDFB. This view shows well the principle of UDFB nesting.

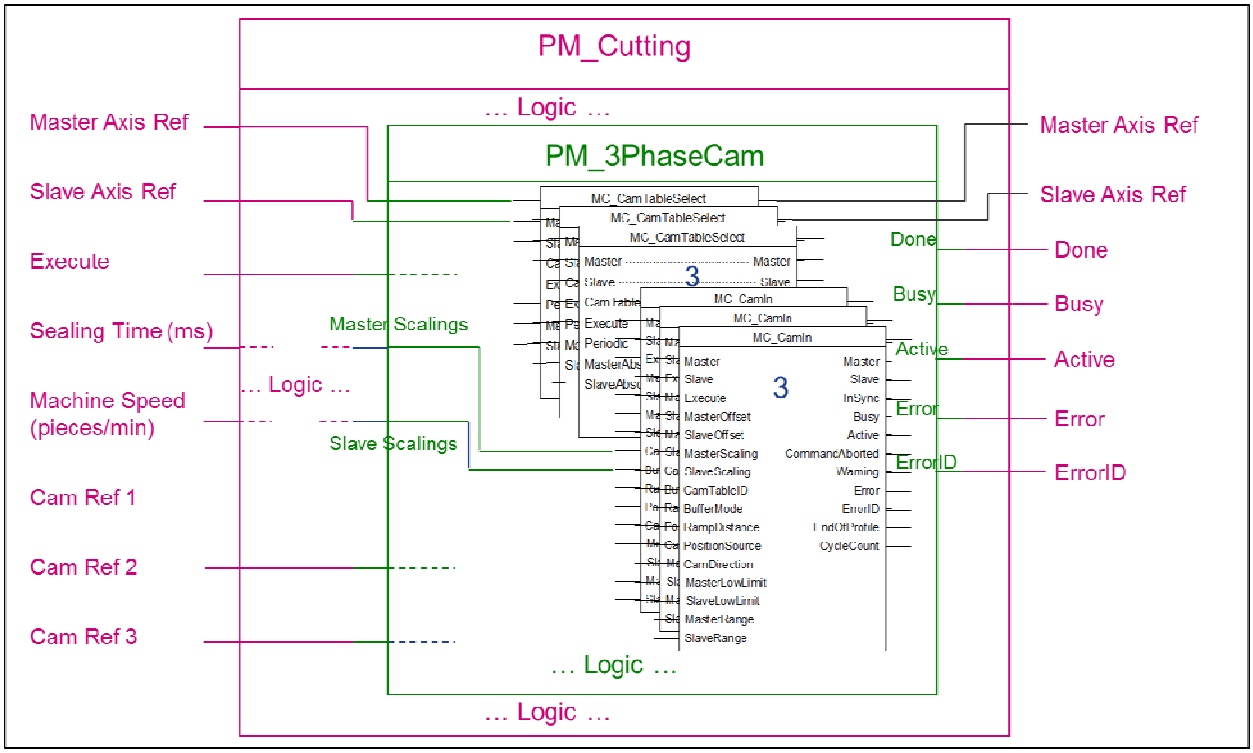


Figure 23 - View of the nested UDFB 'ThreePhaseCam' in the new UDFB 'Cutting'

2.12. Registration function using MC_TouchProbe and MC_Phasing

2.12.1. Introduction into web handling and registration

When looking at products that are bought off the shelf today, almost all of them are packaged. In many cases, the product is actually packaged and wrapped more than once. For example, candy bars come with wrappers around each bar and all of the bars are either in a bag or a box. When shipping the individual bags or boxes to the store, the product is usually placed into a corrugated cardboard box.
Without considering how the product itself is made, let us consider how the packaging that holds the product is made and then placed around the product. The materials used in most packages are plastic, paper or cardboard. Many times the final product is enclosed in all of these types of packaging. All of these products begin as a raw product and are extruded into continuous sheets of material referred to as a web.
As the web material is moved and converted into the final packaging, the product will change shape and color many times before enclosing around the final product. There are many factors that influence the quality of the final product. By using automated control systems it is possible to gather these factors together to allow production of a consistent quality product.
Web processing can be broken down into a variety of different industries. Many of them require the use of registration within the web processing. Examples of some of these industries are listed below:

- Elastic webs - Polycarbonates and Plastics
- Non-elastic Webs - Paper, cardboard and steel
- Sheeters - Cut to print
- Die Cutters - Intermittent and continuous material feed
- Printing - Print to print, Print to cut, Glue to print

Web materials are sensitive to many controllable factors such as temperature, humidity, tension and pressure. Since these elements can not always be perfectly controlled, variations are created which may need to be compensated for. One form of compensation is registration.

2.12.2. Registration functionality

These diverse industries require different types of registration techniques. Some examples of the different registration variations include:

1. Clear lane registration
2. Print registration
3. Product registration

Clear lane and print registration are very similar. Both typically use length as the component that is important to registration.
Clear lane registration is the most common type of registration used in the industry. A lane of material is reserved solely for registration purposes. The only marks that will trigger the fast input will be the registration marks. Print registration involves picking out a distinct distance between marks, which is unique. Print registration is used when it is not practical to use clear lane.
Product registration uses cycle position registration. The important relationship is the position of the product in the cycle.

Registration is most frequently used in master/slave applications. When used with master/slave movements, it has the additional ability of compensating for errors that may occur. The end result is a system that remains synchronized with no accumulated error. Repeatable accuracy throughout a process can be maintained.

In many closed-loop servo systems, it is often necessary to maintain synchronization and accurate positioning repeatability throughout a process. This can be difficult when the product or process itself is inconsistent. Using registration allows you to overcome this difficulty.
Typically, when using registration, sensors are used to detect the position of the product. With non-rigid materials which may stretch or shrink, a photo-eye can detect registration marks on the material. With rigid products, a proximity switch could detect material spacing.

The fast input on the feedback module allows a position at a registration event to be captured. When this occurs, the system generates the referenced axis position.
This is important in applications such as packaging or converting where the process must be precisely coordinated and any non-rigid material cannot be depended upon to retain dimensional relationships. These applications usually involve master/slave moves. The fast input signals can be used as reference to which the master and all subsequent slaves synchronize.

In order to use the registration functionality, a generic block called MC_TouchProbe has been defined within the PLCopen Function Blocks for (extensions). This Function Block provides a very fast recording of the position of a specified axis at the trigger event. The examples below use this to create the registration functionality.

2.12.3. Example of registration

In this example, a foil is continuously fed from a feeder into a cutter. A print mark on the foil is detected from a touch probe sensor, TP-Sensor, which is connected in the software to the MC_TouchProbe.

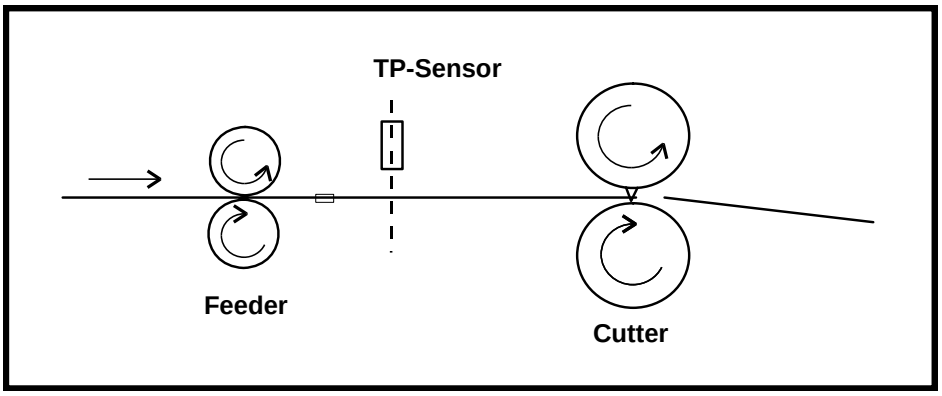


Figure 24 - First Application Example Registration

Principle of operation:

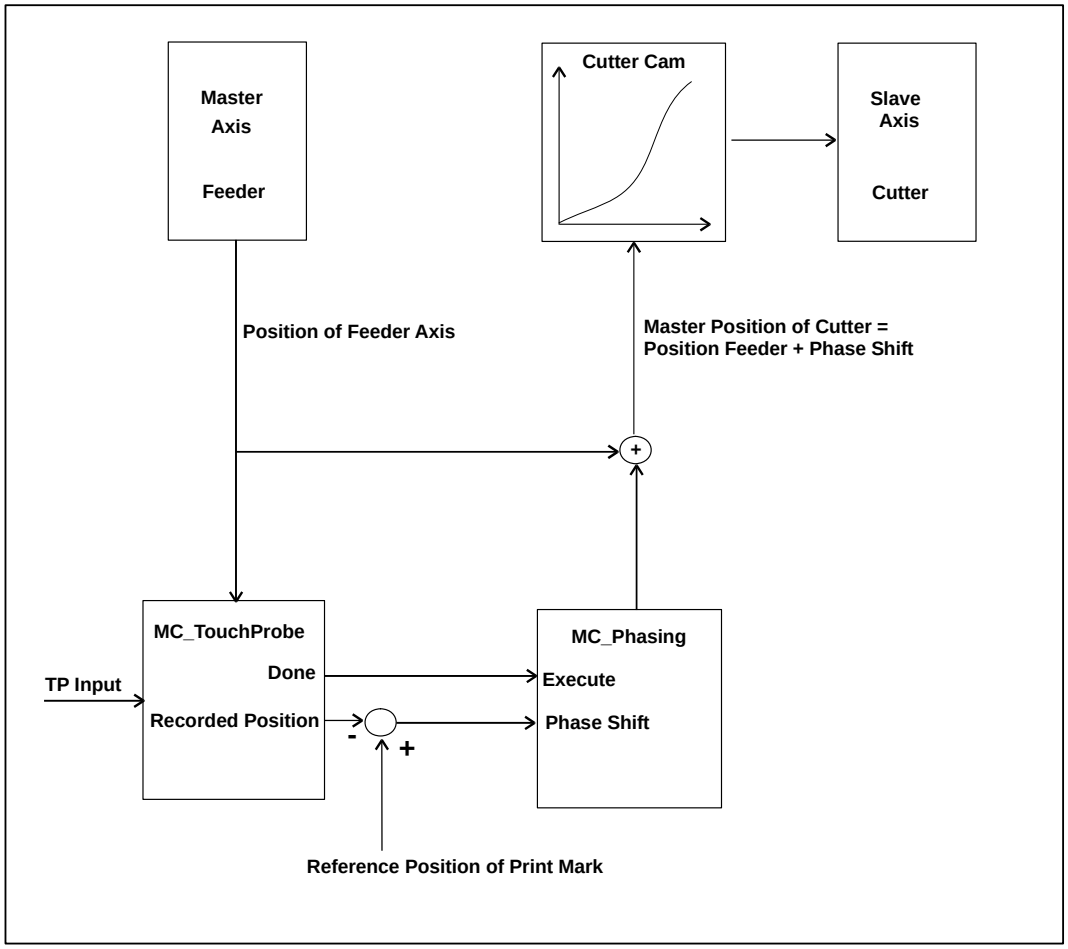


Figure 25 - Principle of Operation

The Function Block MC_TouchProbe latches the Position of the Feeder Axis at the time it sees the Print Mark. The difference to a Reference Position is given to a MC_Phasing, which creates a phase shift between the Feeder Axis and the master position of the Cutter Axis. The Cutter is advancing or delaying in regard to the Feeder.

Possible print mark layout:

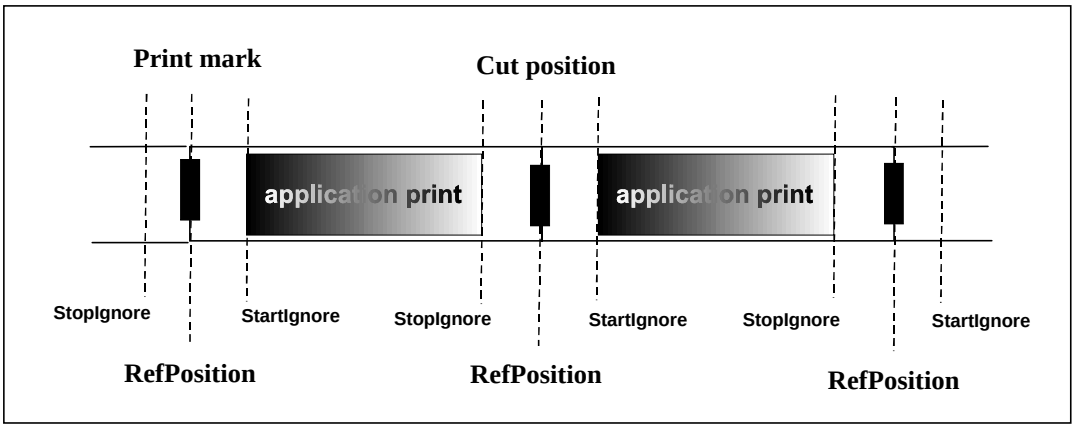


Figure 26 - Printmark Layout

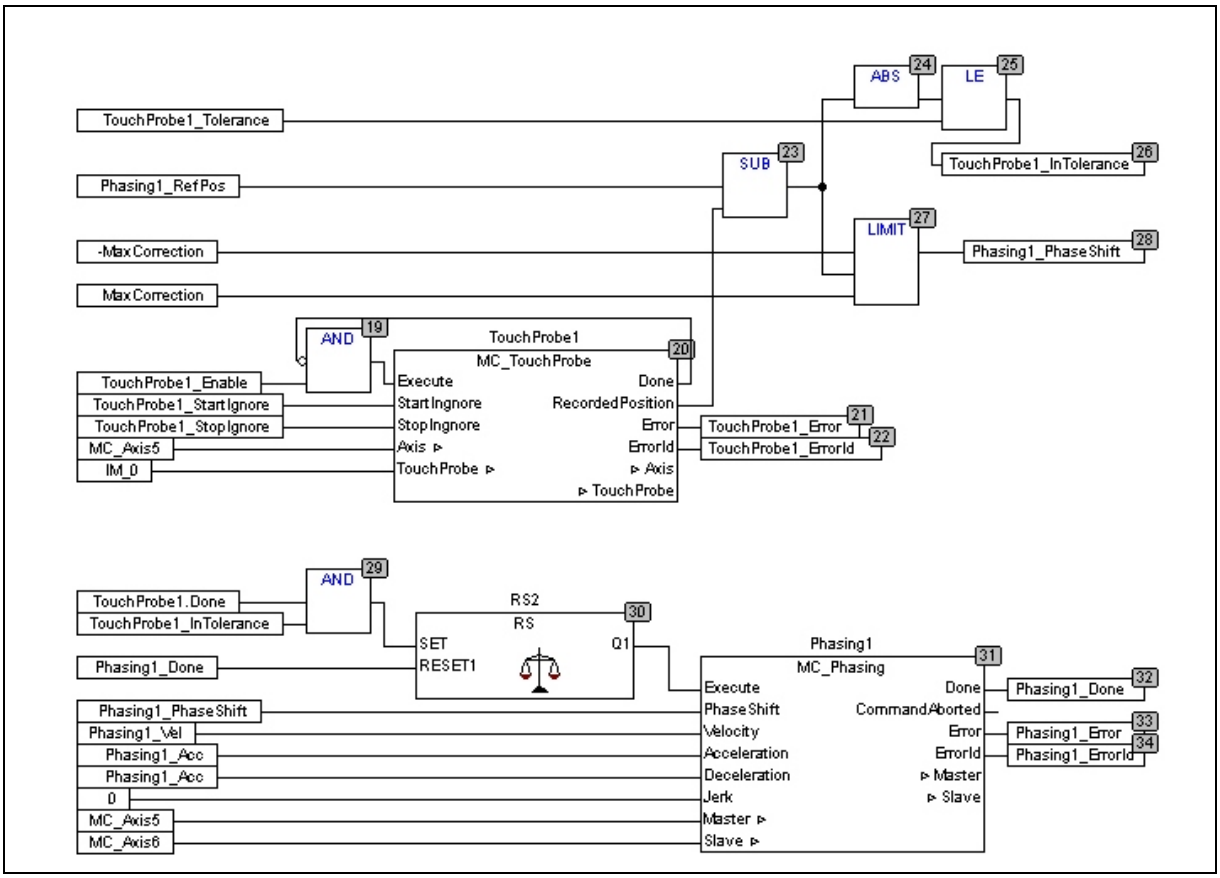


Figure 27 - Program in Function Block Diagram

2.12.4. Example 2 of registration

This example uses the MC_CamIn Function Block. The move has a defined cycle length. Registration compensation, when required, takes place within this cycle with the insertion of an offset value calculated by the software.
Looking at a packaging process where a labeled product coming off a web of non-rigid material must be cut with a knife to 50 cm lengths so that the label is always in the center of the product, you would want to compensate for any variation in product length during each cycle. This is illustrated below:

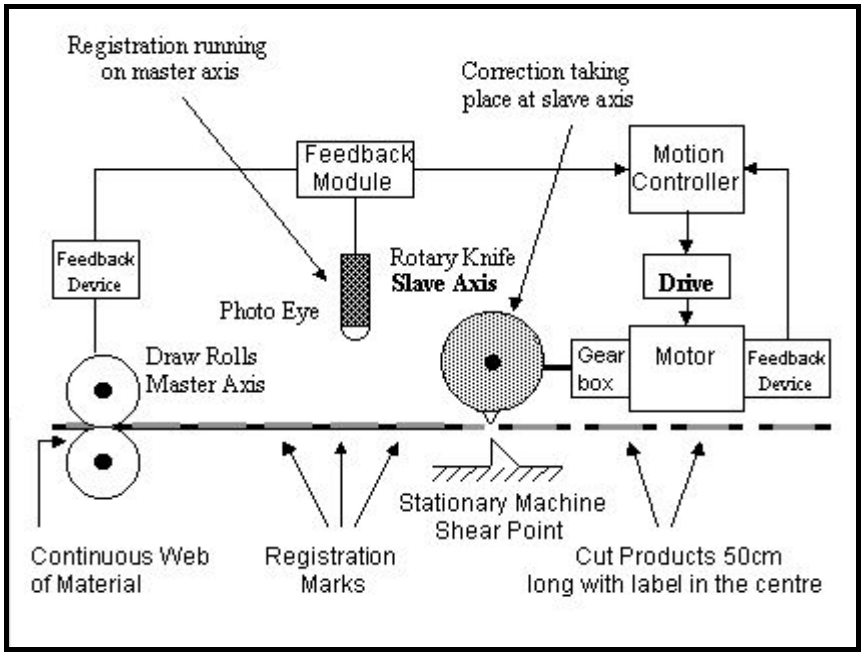
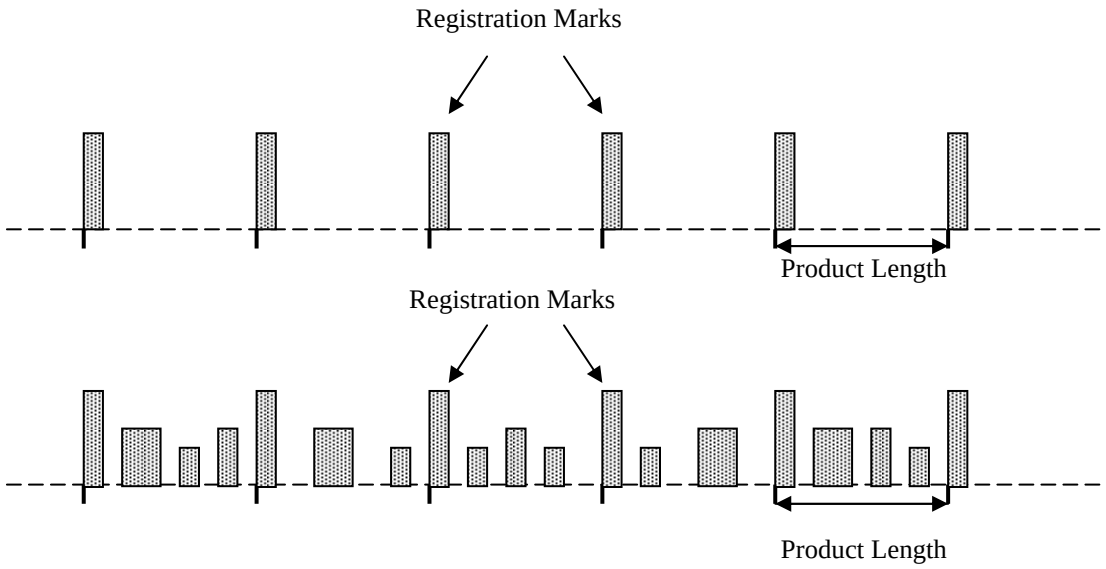


Figure 28 - Second example of registration

Without compensation, any error would accumulate and the label would no longer be centered. As an example, the product is being cut at a rate of 500 per minute. If the product becomes stretched so that the actual length is 501 mm, in 1 minute the label on the product would be off by 500mm (50cm) and in 2 minutes by 1m, etc.
By using a photo eye to detect registration marks on the product, any error in product length will be detected. The rotary knife will adjust its position to compensate for any error in product length so that the product is always cut at the correct position. Because the stretching of the material is gradual, the compensation will be minimal. If there is no stretching of the product, no compensation will occur.
The photo eye is watching for registration marks and sending a signal when it sees one. The Window, with 'FirstPosition' and 'LastPosition', within the MC_TouchProbe helps to find a correct mark. Additional checking could be done via an additional "good mark detector", which could decide if a mark is recognized as good.
When a mark is detected, that information is sent to MC_TouchProbe. The resulted 'RecordedPosition' output is the basis for the calculation of an offset for the master axis. This offset value is sent to the master/slave profile.
Two ways in which registration can be explained are shown below. In the first diagram, every mark is recognized. This can be realized with the WindowOnly input 'Reset'. This will mean every mark is recognized as good. This is acceptable when there is no chance that the photo eye will trigger off any other mark on the product. This is not the case in the 2nd diagram below which shows more than the registration marks. It is then possible to skip unwanted marks by using registration.



In order to apply this to use the PLCopen Function Blocks we have to consider the breakdown of the machine functionality – i.e. what the machines program will look like. The machine program can be broken down into the following actions:

1.	AxisRef	-	set up the master and slave axis to be used in the application
2.	Close the servo loop	-	initialize the drives
3.	Home routine	-	set the master and slave to home positions
4.	Cam profile	-	initialize all the data for the cam profile using table select and use cam-in before motion begins
5.	Move velocity	-	Start the master axis running at a constant velocity
6.	Registration	-	Make sure registration is present when the motion begins.

Certain things need to be done to ensure smooth operation with registration. In order to do this, it would be necessary to add the following parameters to the program, possibly on a separate Function Block.

1. Reading registration position change – It may be necessary to examine the amount of position change incurred by registration.
2. Reading consecutive bad marks – It may be necessary to read the number of consecutive bad marks since the last good mark for reference.
3. Status of registration – This should include the likes of fast input distance, position, number of good and bad marks and the total number of fast inputs that have occurred.
4. Filtering for slave and master – Filtering may be applied to either the slave or master to make the registration changes smoother when executed or to prevent registration changes when in contact with the material.

Below the basic layout of what the code could look like.

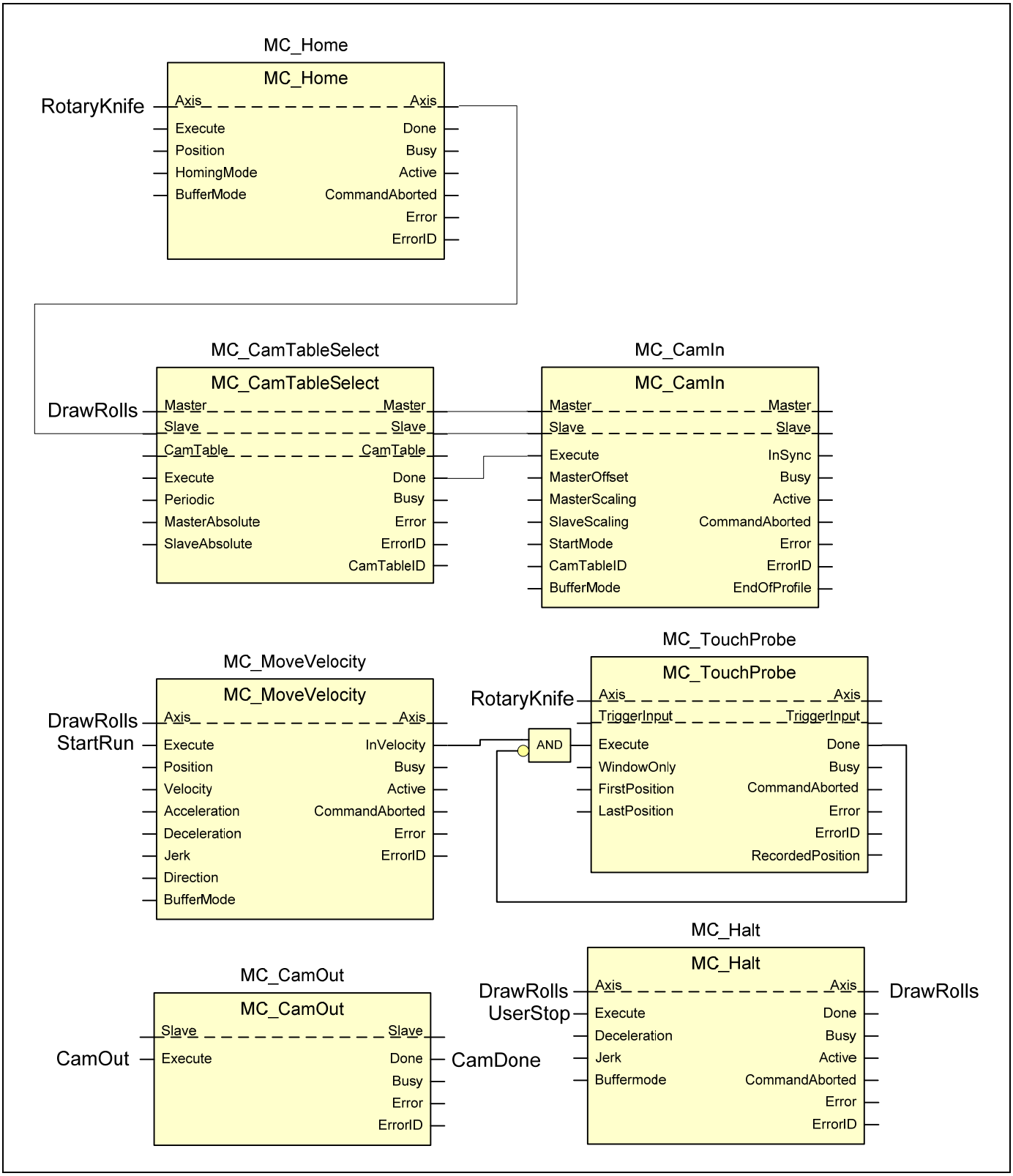


Figure 29 - Second example of registration

2.13. Capping application

Example of MC_TorqueControl in an application for placing a cap on a bottle. The cap includes a tamper ring and liner tightening. This means that there are several stages:

1. getting the cap at the right angle on the top of the bottle
2. turn till the tamper ring goes over the extension in the neck of the bottle
3. tighten the cap in the liner with a certain torque over a certain time, so it is closed correctly

With this, the following phases can be identified:

1. Start with MC_MoveVelocity with velocity value 'HiVelocity'
2. When acceleration phase is over, and after time delay 'T1', start to check the actual torque by using MC_ReadActualTorque.
3. When reaching angle 'NoCapStuck', reduce speed via MC_MoveVelocity to value 'MidVelocity', in order to lower the amount of axis´ kinetic energy
4. When 'TravlTorq' exceeds 'Torq', it signals that the tamper ring is reached. Start MC_TorqueControl with torque 'CapOnTorq' and 'LowVelocity' as inputs.
5. First the Velocity is very low activating the tamper ring. When the tamper ring is on place, the Velocity goes up to 'LowVelocity'.
6. When 'LowVelocity' and position 'CapOnDeg' achieved it signals that you run fine beyond the tamper ring, then MC_TorqueControl is used with 'LinerTorq' and 'CreepVelocity' as inputs. The 'CreepVelocity' limits the torque in the beginning.
7. When the cap liner meets the neck, the Velocity goes to zero and the torque increases.
8. When 'MoveTorq' is on for the time 'TorqTime' the liner has been compressed correctly!

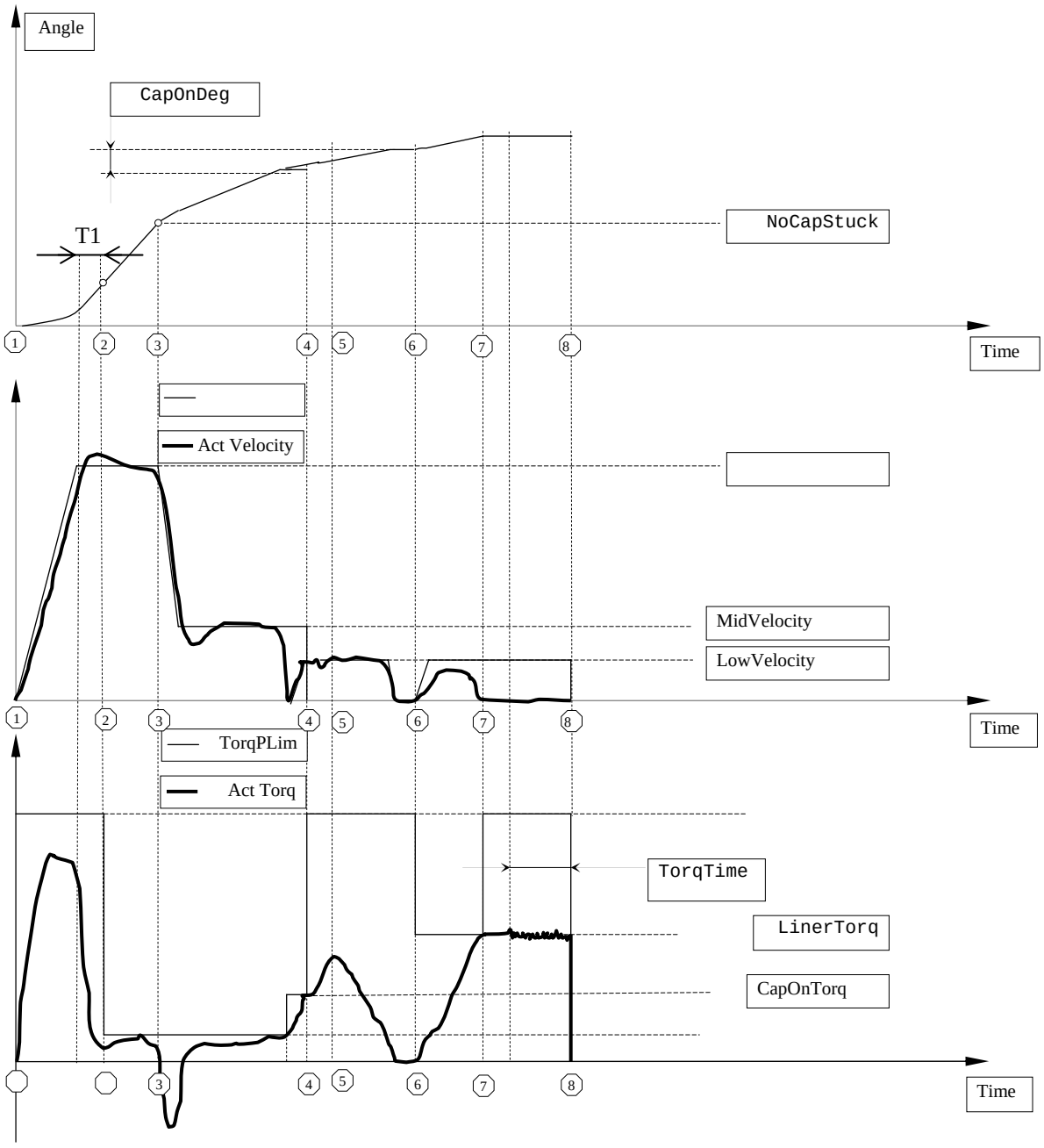


Figure 30 - Timing example for capping application

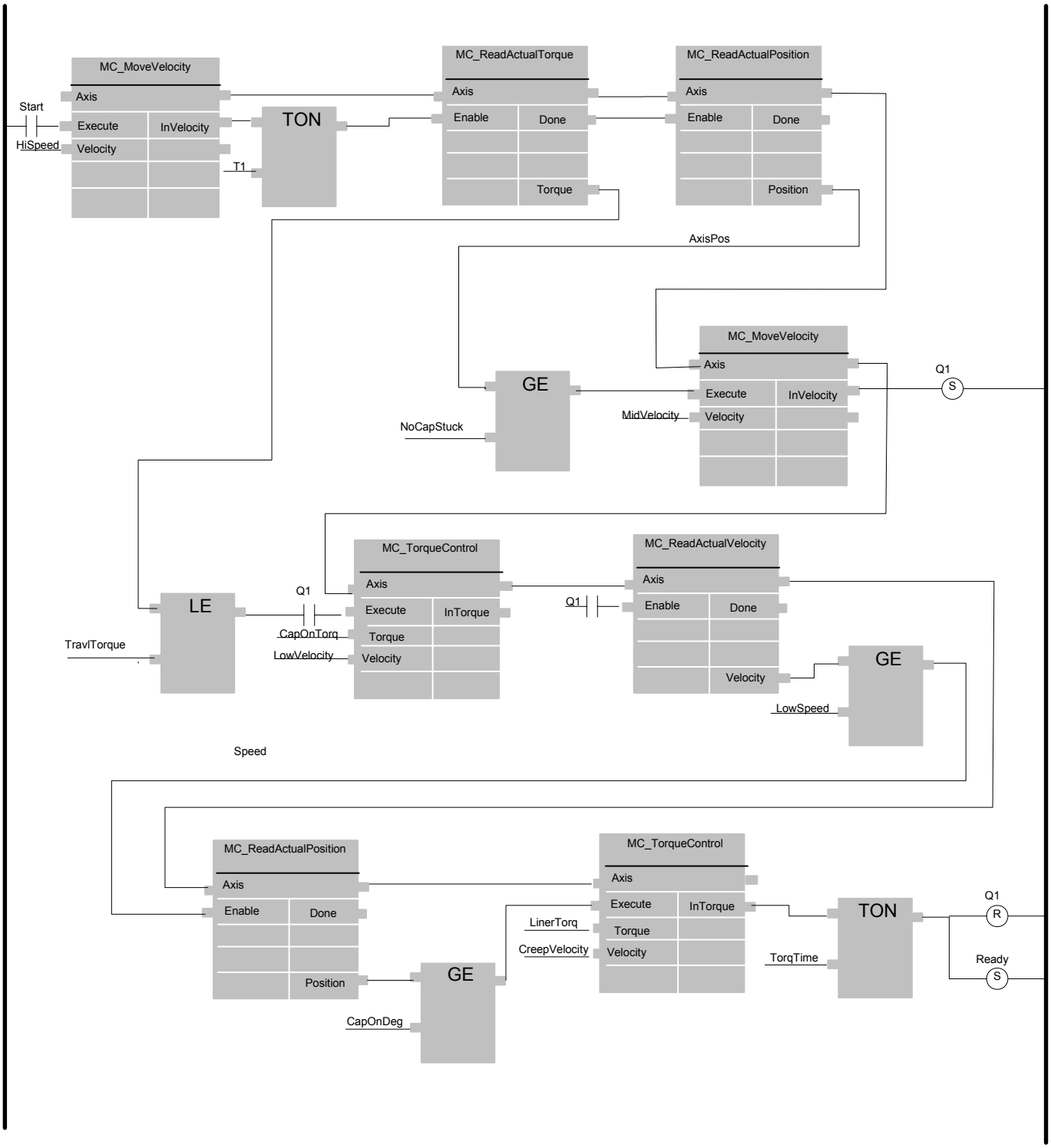
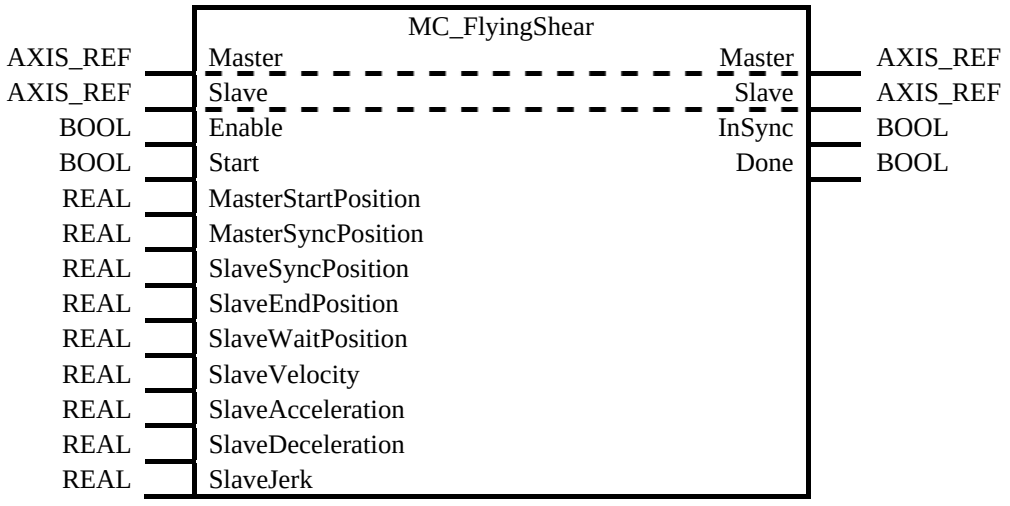


Figure 31 - Program example in LD for capping application

2.14. MC_FlyingShear

In many applications one needs to cut the material to a certain length while this is moving. This Function Block solves this aspect.

FB-Name	MC_FlyingShear		
This Function Block performs a defined synchronized motion between a continuously running master axis and a slave axis.			
VAR_IN_OUT			
	Master	AXIS_REF	
	Slave	AXIS_REF	
VAR_INPUT			
	Enable	BOOL	Enables the Function Block
	Start	BOOL	Starts a synchronizing sequence
	MasterStartPosition	REAL	Master position that determines the phase relation between master and slave axis [u]
	MasterSyncPosition	REAL	Master position where synchronized motion starts [u]
	SlaveSyncPosition	REAL	Corresponding slave position [u]
	SlaveEndPosition	REAL	Slave position where synchronized motion ends [u]
	SlaveWaitPosition	REAL	Slave position where slave axis waits [u]
	SlaveVelocity	REAL	Value of the maximum slave velocity (always positive) (not necessarily reached) [u/s].
	SlaveAcceleration	REAL	Value of the acceleration (always positive) (increasing energy of the motor) [u/s^2]
	SlaveDeceleration	REAL	Value of the deceleration (always positive) (decreasing energy of the motor) [u/s^2]
	SlaveJerk	REAL	Value of the Jerk [u/s^3]. (always positive)
VAR_OUTPUT			
	InSync	BOOL	Synchronized motion in progress
	Done	BOOL	Slave has reached waiting position
Notes: -			



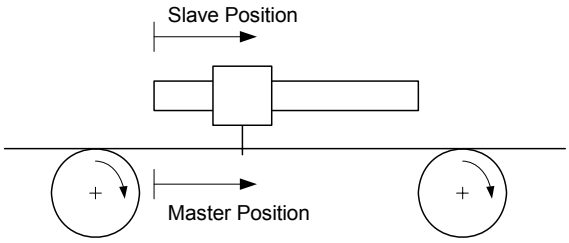


Figure 32 - Flying Shear

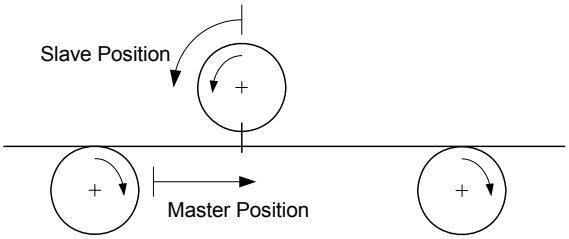


Figure 33 - Rotating Cutter

The figures above give a sketch that shows the function to be solved with this Function Block for the flying shear and rotating cutter respectively. The primary task is to cut a moving continuous flow of material at specified positions into discrete pieces (i.e. products). Many other applications are also possible with the same Function Block, like printing, filling, etc.

Both functions are quite similar, with the difference that for the Flying Shear the slave axis features a finite range of motion and for the rotating cutter the slave axis continues to move in one direction.

The motion can be split into two parts:

1. Synchronization between master and slave axis and synchronous motion of both axes (where the actual task is performed)
2. Slave axis moves to a defined waiting position and waits for next action request

For the first part, the master and slave axes have to operate at synchronous speeds while maintaining a specified phase relationship (e.g. to assure an accurate cutting point). For the standard MC_GearIn Function Block no phase relationship can be given, and thus MC_GearInPos is used. With this Function Block, the sequence of actions can be easily generated by a combination of MC_GearInPos and MC_MoveAbsolute. Figure 34 - shows a timing diagram of this process, where the solid line shows the Flying Shear, and the dashed line the Rotating Cutter.

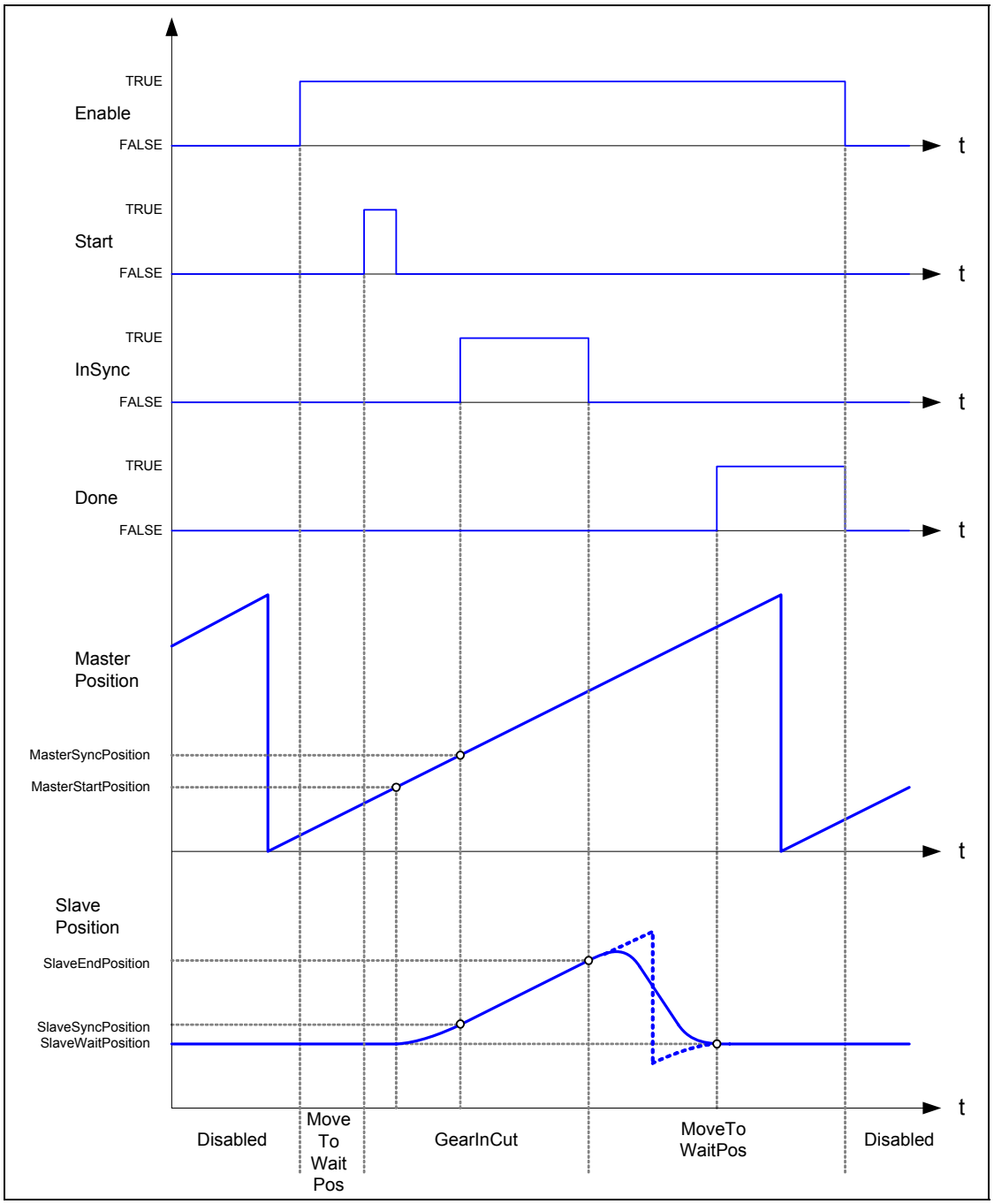


Figure 34 - Timing diagram for a single cut
(Note: solid line = Flying Shear; dashed line = Rotating Cutter)

The following Function Block (specified as an SFC) performs such a sequence. The difference between Flying Shear and Rotating Cutter is only within the parameters of the MC_MoveAbsolute Function Block that specifies the motion towards the waiting position of the slave.

A gear ratio of 1:1 between master and slave axis is assumed. This means that both axes have the same position scaling. If this assumption does not apply in a user's case, the user can adapt this ratio to his needs.

Variable Declaration:

```
FUNCTION_BLOCK MC_FlyingShear
    VAR_INPUT
        Enable :              BOOL;
        Start :               BOOL;
        MasterStartPosition : REAL;
        MasterSyncPosition :  REAL;
        SlaveSyncPosition :   REAL;
        SlaveEndPosition :    REAL;
        SlaveWaitPosition :   REAL;
        SlaveVelocity :       REAL;
        SlaveAcceleration :   REAL;
        SlaveDeceleration :   REAL;
        SlaveJerk :           REAL;
    END_VAR

    VAR_IN_OUT
        Master :              AXIS_REF;
        Slave :               AXIS_REF;
    END_VAR

    VAR_OUTPUT
        InSync :              BOOL;
        Done :                BOOL;
    END_VAR

    VAR
        NoEnable :            BOOL;
        MoveAbsSlave :        MC_MoveAbsolute;
        GearInPosSlave :      MC_GearInPos;
        ReadSlavePosition :   MC_ReadActualPosition;
    END_VAR

END_FUNCTION_BLOCK
```

SFC Diagram:

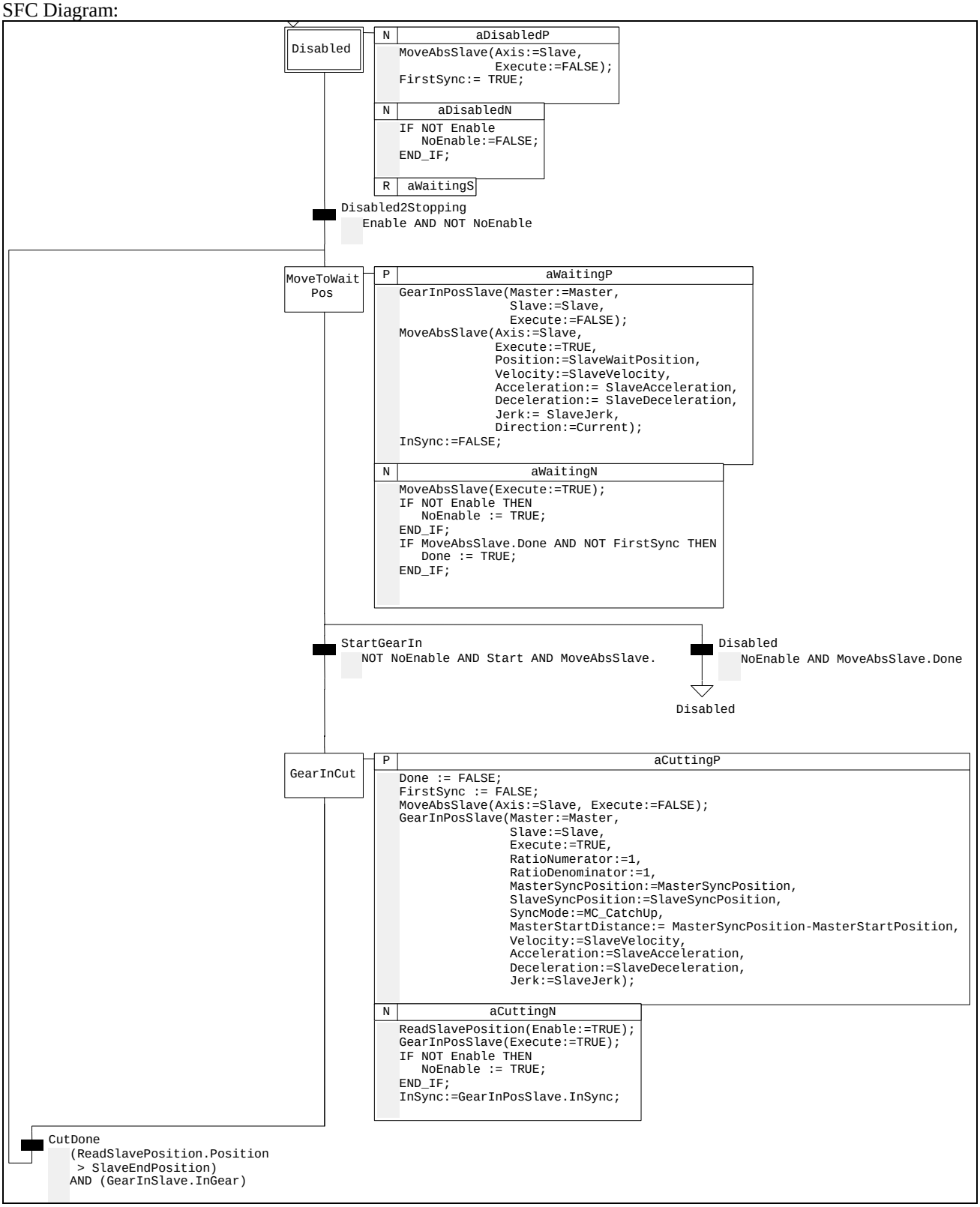


Figure 35 - SFC for Flying Shear

(Note: Italic "*Direction*" parameter is necessary only for Rotating Cutter)

According to the above mentioned two phases of the operating sequence two working states are introduced. 'In MoveToWaitPosition' the slave axis is moved to 'SlaveWaitPosition' and stops there.

As soon as the 'Start' input is activated, a transition to the 'GearInCut' state is performed, where the synchronized motion is

initiated by means of ‘GearInPosSlave’. In this state, the position of the slave axis is monitored, and as soon as it exceeds ‘SlaveEndPosition’, the ‘MoveToWaitPosition’ state is activated again.

There are two control signals that influence the state sequence:

1. The ‘Enable’ signal enables the block. As soon as it is activated, the slave axis moves to ‘SlaveWaitPosition’. If the ‘Enable’ signal is deactivated, the sequence is stopped as soon as the slave axis has reached ‘SlaveWaitPosition’ (in case of an active sequence at the moment when the ‘Enable’ signal is deactivated). This assures a defined and save state where the action of the Function Block is disabled. An immediate stop of the slave axis would cause serious harm in most applications.
2. The rising edge of the ‘Start’ signal initiates a single sequence of action. The parameters for this single action can be influenced until the moment of this rising edge, this means that each cut can be parameterized separately, e.g. by the result of a MC_TouchProbe Block. Thus registration applications can also be solved by means of this Function Block.

2.15. Synchronized Motion with SFC

This example uses Sequential Function Charts to program 4 motors (and drives) in total:

- 2 for horizontal movement (MotorHorLeft and MotorHorRight)
- 2 for rotating movements (MotorRotLeft and MotorRotRight))

At the axis of the rotating motors are a grated cylinder, here represented via 4 tubes, but normally with a smaller discrepancy (space between the grates especially if they are merged together in the same space). The axis of these motors are aligned for this. The rotating motors are synchronized, and then moved to each other in such a way that the 'gratings' (on the cylinders) fit. This synchronization starts wit a low speed, which is increased over time. Also, both synchronized motors are moved horizontally to the left and the right as a 'set' (via the other 2 motors in also a synchronized mode).

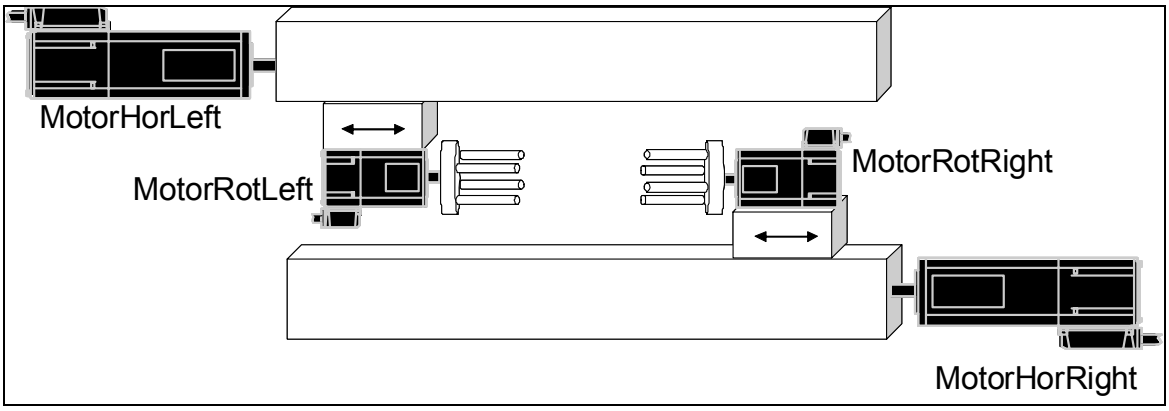


Figure 36 - Layout of the example

Short description of the application program:

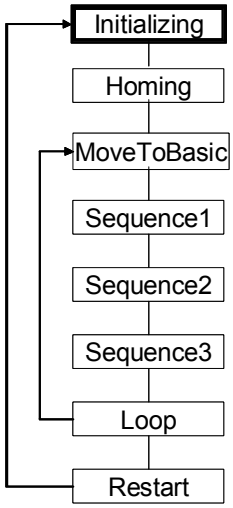


Figure 37 - Overview of the main program

1. MC_CamIn:
   Master: MotorRotRight
   Slave: MotorRotLeft
   MC_MoveSuperImp. MotorRotLeft

2. MC_MoveRelative MotorHorLeftt

3. MC_CamIn:
   Master: MotorHorLeft

Slave: MotorHorRight

4. MC_MoveAbsolute - MotorHorLeft

Main Program in SFC:

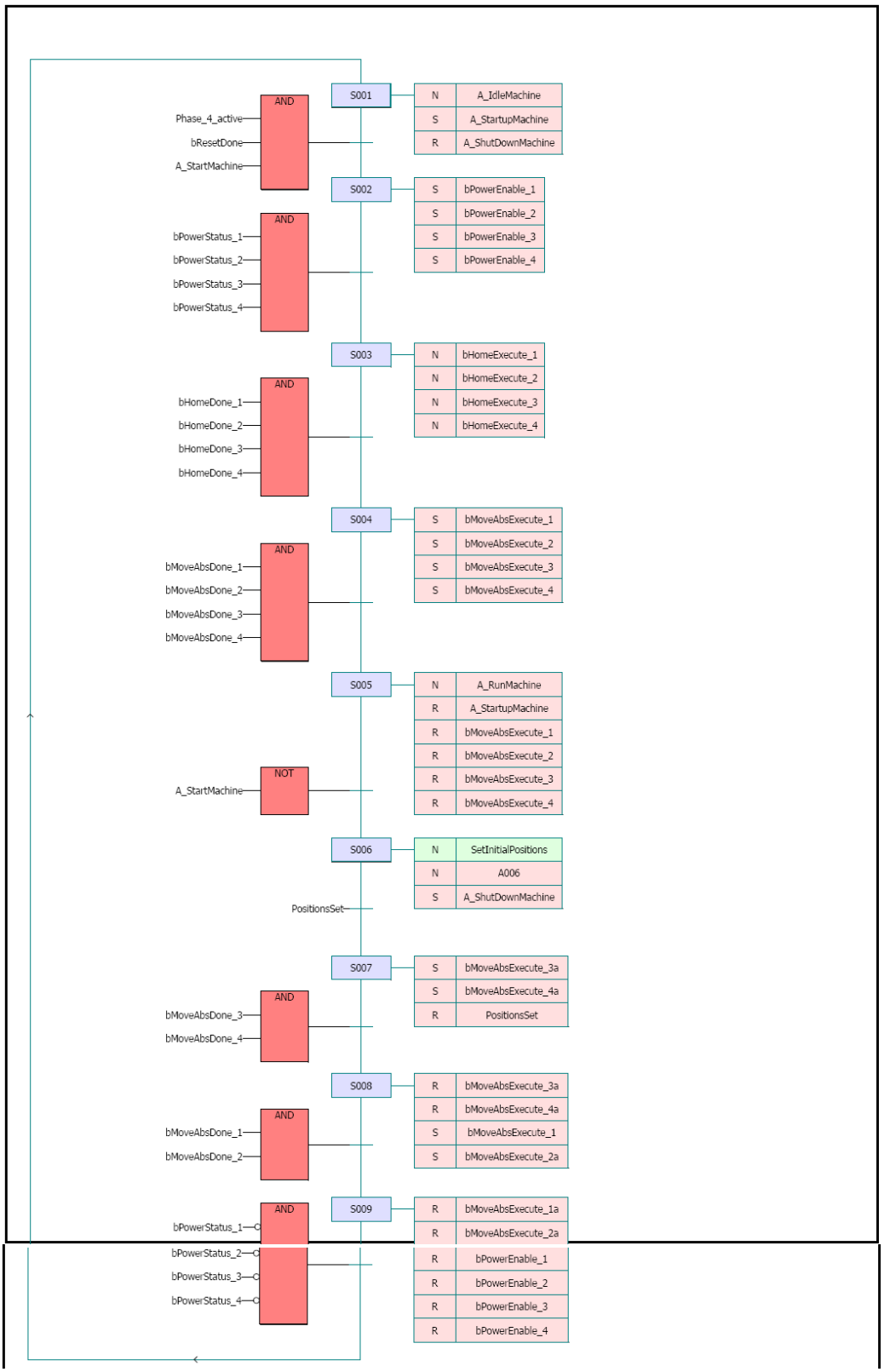


Figure 38 - Main SFC Program

Main SFC Program

Step 1 – Init
Actions:

N	A_IdleMachine
S	A_StartupMachine
R	A_ShutDownMachine

Transition Condition:
Phase_4_Active AND bResetDone AND A_StartMachine

Step 2 – Switch Power Stage ON
Actions:

S	bPowerEnable1
S	bPowerEnable2
S	bPowerEnable3
S	bPowerEnable4

Transition Condition:
bPowerStatus1 AND bPowerStatus2 AND bPowerStatus3 AND bPowerStatus4

Step 3: Homing procedures
Actions:

N	bHomeExecute_1
N	bHomeExecute_2
N	bHomeExecute_3
N	bHomeExecute_4

Transition Condition:
(bHomeDone1 AND bHomeDone2 AND bHomeDone3 AND bHomeDone4)

STEP 4: Move to Initial Positions
Actions:

S	bMoveAbsExecute_1
S	bMoveAbsExecute_2
S	bMoveAbsExecute_3
S	bMoveAbsExecute_4

Transition Condition:
(bMoveAbsDone_1 AND bMoveAbsDone_2 AND bMoveAbsDone_3 AND bMoveAbsDone_4)

STEP 5 - RunMachineProgram
Actions:

N	A_RunMachine
R	A_StartupMachine
R	bMoveAbsExecute_1
R	bMoveAbsExecute_2
R	bMoveAbsExecute_3
R	bMoveAbsExecute_4

TransitionCondition: NOT A_StartMachine

STEP 6: Start shutting down machine program
Actions:

N	SetInitialPositions
N	A006
S	A_ShutDownMachine

Transition Condition: PositionSet

STEP 7 – Move Servo 3 and 4
Actions:

S	bMoveAbsExecute_3a
S	bMoveAbsExecute_4a
R	PositionSet

Transition Condition: (bMoveAbsDone_3 AND bMoveAbsDone_4)

STEP 8: Move horizontal (Axis 1 and 2)
Actions:

R	bMoveAbsExecute_3a
R	bMoveAbsExecute_4a
S	bMoveAbsExecute_1
S	bMoveAbsExecute_2a

Transition Condition: (bMoveAbsDone_1 AND bMoveAbsDone_2)

Step 9: Turn Off Machine
Actions:

R	bMoveAbsExecute_1a
R	bMoveAbsExecute_2a
R	bPowerEnable_1
R	bPowerEnable_2
R	bPowerEnable_3
R	bPowerEnable_4

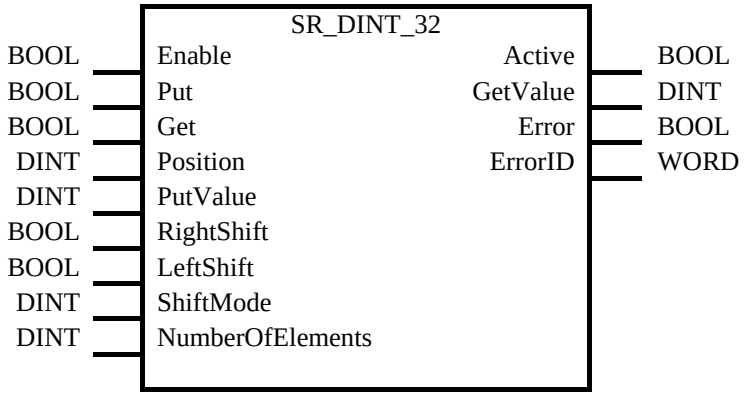
Transition Condition:
(bPowerStatus_1 AND bPowerStatus_2 AND bPowerStatus_3 AND bPowerStatus_4)

Return to Step 1

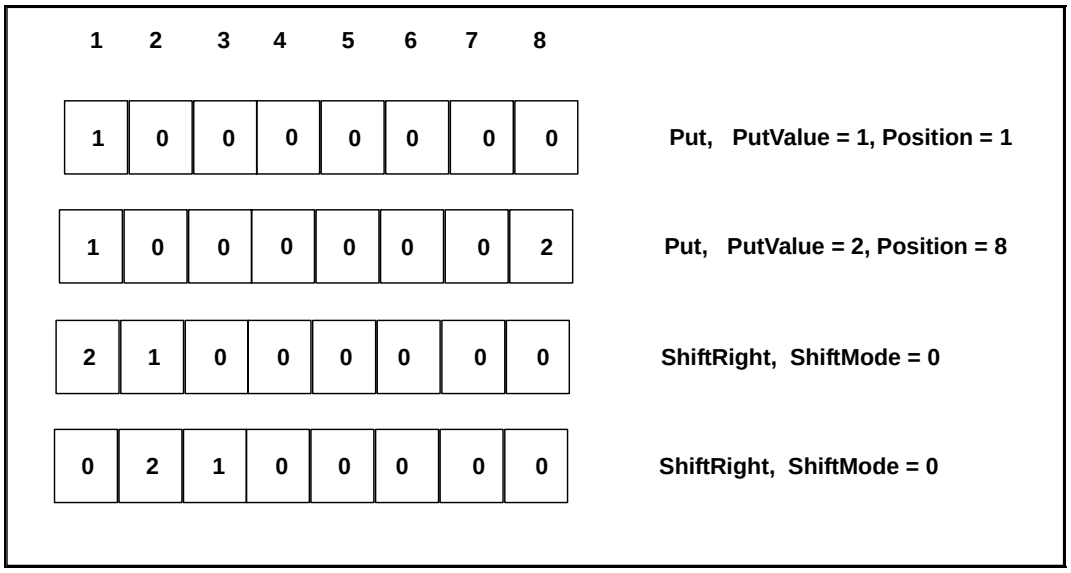
2.16. Shift Register as User Derived Function Block

In applications with machines with several stations, where information is transported with the product through the machine, a shift register can be used.
To avoid pointers in Shift Registers or FIFO's, one way is to define for each data type needed it's own FB.
If there is multiple Data to shift, a user FB can be made out of this Basic FB's. Almost all data types can be realized in this way.
The FB hereunder is just one of the whole possible set.

FB-Name	SR_DINT_32	
This FB is a Shift Register for DINT Values with a number of 2-32 elements.		
VAR_INPUT		
Enable	BOOL	Initializes the Shift Register on pos. edge. Sets the active elements to 0
Put	BOOL	Writes value into Shift Register on positive edge
Get	BOOL	Reads value from Shift Register on positive edge
Position	DINT	Number of Element to read or write
PutValue	DINT	Value for Put
RightShift	BOOL	Shifts all values one position to the right
LeftShift	BOOL	Shifts all values one position to the left
ShiftMode	DINT	0 = Shift with last element into first element (rotate) 1 = Shift with 0 into first element 2 = Shift with first element stays the same (filling)
NumberOfElements	DINT	Number of active Elements (2-32)
VAR_OUTPUT		
Active	BOOL	FB is enabled
GetValue	DINT	Value for Get
Error	BOOL	Signals that error has occurred within Function block
ErrorID	WORD	Error number
Notes:		



Examples:



Program in ST Code

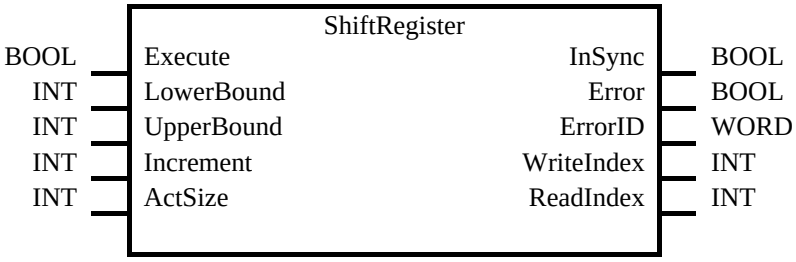
```
FUNCTION_BLOCK DINT_SR32

(* Function: The FB is a ShiftRegister Type Buffer for DINT Values with the maximum size of 32 Elements. *)

    (* Variable declaration*)

    VAR_INPUT
        Enable :           BOOL; (* Initializes the Shift Register on pos. edge *)
                                 (* Sets the active elements to Zero *)
        Put :              BOOL; (* Writes value into Shift Register on positive edge *)
        Get :              BOOL; (* Reads value from Shift Register on positive edge *)
        Position :         DINT; (* Position (number) of Element for Put and Get *)
        PutValue :         DINT; (* Value for Put  *)
        RightShift :       BOOL; (* Shifts all values one position to the right *)
        LeftShift :        BOOL; (* Shifts all values one position to the left *)
        ShiftMode :        DINT; (* 0 = Shifts last element into first element (rotate) *)
                                 (* 1 = Shifts a Zero into first element *)
                                 (* 2 = Shift and first element stays the same (filling) *)
        NumberOfElements : DINT; (* Number of active Elements (2-32) *)
    END_VAR

    VAR_OUTPUT
        Active :           BOOL; (* Goes high with Enable *)
        GetValue :         DINT; (* Value for Get *)
        Error :            BOOL; (* Set on Error in FB *)
        ErrorID :          DINT; (* -1 = PositionOutOfRange *)
                                 (* -2 = NumberOfElementsOutOfRange *)
    END_VAR

    VAR CONSTANT
        MaxElements :                DINT := 32; (* defines the maximum size *)
        PositionOutOfRange :         DINT := -1; (* ErrorID *)
        NumberOfElementsOutOfRange : DINT := -2; (* ErrorID *)
    END_VAR

    VAR
        SRARRAY :          ARRAY[1..MaxElements] OF DINT;
        EnableFlag :       BOOL;
        GetFlag :          BOOL;
        PutFlag :          BOOL;
        RightShiftFlag :   BOOL;
        LeftShiftFlag :    BOOL;
        Index :            DINT;
        StartPos :         DINT;
        LastValue :        DINT;
    END_VAR

    (* Code Sequence*)
```

```
    IF Enable AND NOT EnableFlag THEN (* Inits FB at pos. edge of Enable *)
        Active := TRUE;
        EnableFlag := TRUE;

        IF NumberOfElements < 2 OR NumberOfElements > MaxElements THEN
            ErrorID:= NumberOfElementsOutOfRange;
            Error:= TRUE;
        ELSE
            FOR Index:= 1 TO NumberOfElements DO (* delete active buffer *)
                SRARRAY[Index] := 0;
            END_FOR

            ErrorID := 0;
            Error := FALSE;
            PutFlag := FALSE;
            GetFlag := FALSE;
            RightShiftFlag := FALSE;
            LeftShiftFlag := FALSE;
            StartPos := 1;
        END_IF
    ELSIF NOT Enable AND EnableFlag THEN (* Disables FB at neg. edge of Enable *)
        EnableFlag := FALSE;
        Active := FALSE;
    END_IF

    IF Enable AND NOT Error THEN
        IF Put AND NOT PutFlag THEN (* pos.edge at Put *)
            PutFlag := TRUE;

            IF Position < 1 OR Position > NumberOfElements THEN
                ErrorID:= PositionOutOfRange;
                Error := TRUE;
            ELSE
                Index := (StartPos -1 + Position);

                IF Index >  NumberOfElements THEN
                    Index := Index - NumberOfElements;
                END_IF

                SRARRAY[Index]:= PutValue;
            END_IF

        ELSIF NOT Put AND PutFlag THEN
            PutFlag := FALSE;
        END_IF

        IF Get AND NOT GetFlag THEN (* pos. edge at Get *)
            GetFlag := TRUE;

            IF Position < 1 OR Position > NumberOfElements THEN
                ErrorID := PositionOutOfRange;
                Error := TRUE;
            ELSE
                Index := (StartPos -1 + Position);
                IF Index > NumberOfElements THEN
                    Index := Index - NumberOfElements;
                END_IF

                GetValue := SRARRAY[Index];
            END_IF

        ELSIF NOT Get AND GetFlag THEN
            GetFlag := FALSE;
        END_IF

        IF RightShift AND NOT RightShiftFlag THEN (* pos. edge at RightShift *)
            RightShiftFlag := TRUE;
            LastValue := SRARRAY[StartPos]; (* is used in Mode 2 *)
            StartPos := StartPos -1;        (* that is the shift right operation *)

            IF StartPos < 1 THEN
                StartPos := NumberOfElements;
            END_IF
            IF ShiftMode = 1 THEN           (* 0 --> first element *)
                SRARRAY[StartPos] := 0;
            ELSIF ShiftMode = 2 THEN        (* LastValue --> first element *)
                SRARRAY[StartPos] := LastValue;
            END_IF

        ELSIF NOT RightShift AND RightShiftFlag THEN
```

```
            RightShiftFlag := FALSE;
        END_IF

        IF LeftShift AND NOT LeftShiftFlag THEN (* pos.edge at LeftShift *)
            LeftShiftFlag := TRUE;
            Index := StartPos -1; (* Index for last element *)

            IF Index < 1 THEN
                Index := NumberOfElements;
            END_IF
            LastValue := SRARRAY[Index]; (* is needed in Mode2 *)
            Index := StartPos;           (* is needed in Mode1 + 2 *)
            StartPos := StartPos +1;     (* that is the shift left operation *)

            IF StartPos > NumberOfElements THEN
                StartPos := 1;
            END_IF
            IF ShiftMode = 1 THEN (* 0 --> last element *)
                SRARRAY[Index] := 0;
            ELSIF ShiftMode = 2 THEN (* LastValue --> first element *)
                SRARRAY[Index] := LastValue;
            END_IF

        ELSIF NOT LeftShift AND LeftShiftFlag THEN
            LeftShiftFlag := FALSE;
        END_IF
    END_IF

END_FUNCTION_BLOCK
```

2.17. ShiftRegister Logic

FB-Name	**ShiftRegister**	
This Function Block handles the management of a shift register in conjunction with a user supplied array that contains the data to be shifted. It avoids the usage of non-IEC-standard language elements such as pointers.		
VAR_INPUT		
Execute	BOOL	Increments shift register position
LowerBound	INT	Lower index bound of data array
UpperBound	INT	Upper index bound of data array
Increment	INT	By how many data fields should shift take place
ActSize	INT	Actual size of shift register (for partial usage of the data array)
VAR_OUTPUT		
Done	BOOL	Shift Operation was successful
Error	BOOL	Signals that error has occurred within Function Block
ErrorID	WORD	Error identification
WriteIndex	INT	Array index where data should be written
ReadIndex	INT	Array index where data should be read
Notes:		



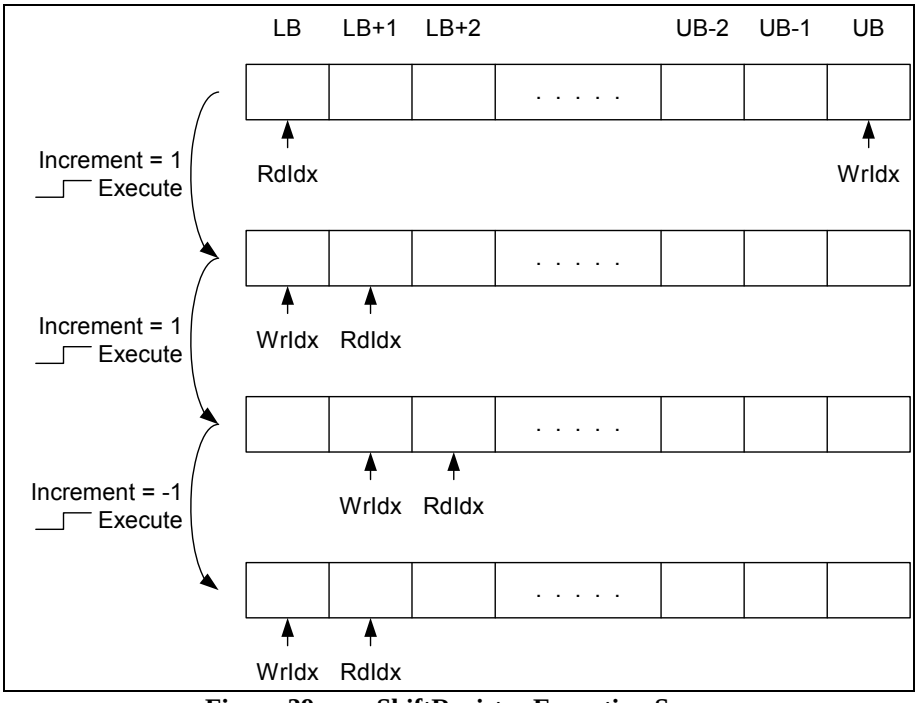


Figure 39 - ShiftRegister Execution Sequence

Sample code:

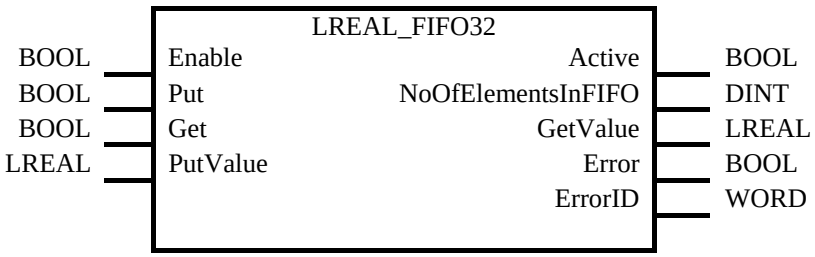
```
TYPE
    tMySampleData : STRUCT (* for example: user defined data type for shift register content *)
        rData : REAL;
```

```
        bData : BOOL;
    END_STRUCT;
    tMyData : ARRAY[1..10] OF tMySampleData; (* user declares data storage array *)
END_TYPE

VAR
    aMyData :    tMyData;
    myShiftReg : MC_ShiftRegister;
    readData :   tMySampleData; (* data that is read from the shift reg. *)
    writeData :  tMySampleData; (* data that is written to the shift reg. *)
    newProduct : BOOL;          (* indicates that a new product is to be worked on *)
    init :       BOOL;          (* indicates the first usage of the shift reg. *)
END_VAR

IF init THEN (* initialization *)
    myShiftReg(Execute := FALSE,
               LowerBound := 1,
               UpperBound := 10,
               Increment := 1,
               ActSize := 10);
END_IF
IF newProduct THEN
    ...
    aMyData[myShifReg.WriteIndex] := writeData; (* write data to shift reg. *)
    readData := aMyData[myShifReg.ReadIndex];   (* read data from shift reg. *)
    myShiftReg(Execute := TRUE);                (* forward shift register *)
    myShiftReg(Execute := FALSE);
...
END_IF
```

2.18. FIFO Function Block

Applications:
Touch Probe capturing with multiple parts between TP-Sensor and process (Cutting, Sealing, Filling).
Every TP performs a Put to the FIFO and every process action (Cut) performs a Get.

FB-Name	**LREAL_FIFO32**	
The FB 'MC_LREAL_FIFO32' is a 'first in first out' type ring buffer of the for LREAL values with a depth of max. 32 elements.		
VAR_INPUT		
Enable	BOOL	Initializes the FIFO buffer on pos. edge
Put	BOOL	Writes value into FIFO on positive edge
Get	BOOL	Reads value from FIFO on positive edge
PutValue	LREAL	Value for Put
VAR_OUTPUT		
Active	BOOL	FB is enabled
NoOfElementsInFIFO	DINT	Shows the number of elements in Buffer
GetValue	LREAL	Value for Get
Error	BOOL	Signals that error has occurred within Function Block
ErrorID	INT	Error number
Notes :		



Principal of a FIFO with a depth of 8 elements:

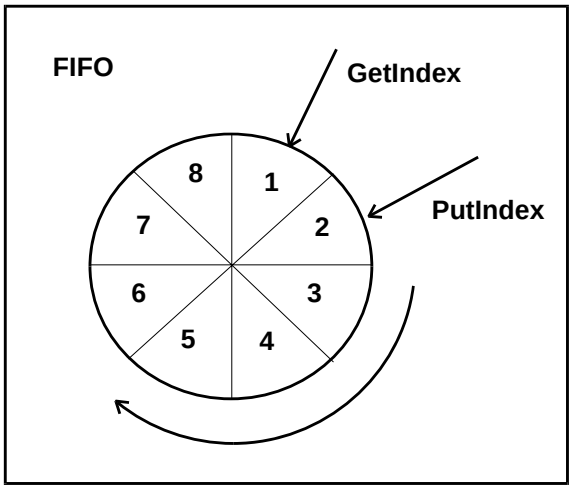


Figure 40 - Principle of a FIFO

A FIFO is a ring buffer with a pointer to the 'GetValue' and 'PutValue'. As long as 'NoOfElementsInFIFO' stays below 32 there can be arbitrarily performed puts and gets.
At the beginning both pointers point to element one and the 'NoOfElementsInFIFO' is zero.
With writing into the FIFO the 'PutValue' is written into the first element and the 'Put' pointer is incremented by one. The 'Get'

pointer stays at one and the 'NoOfElementsInFIFO' gets incremented by one.

Performing a 'Get' will then read the value out of the FIFO into the 'GetValue' and incremented the 'Get' pointer. The 'NoOfElementsInFIFO' gets decremented by one.

Program in ST:

```
FUNCTION_BLOCK LREAL_FIFO32

    (* Function: The FB is a first in first out (FIFO) type Buffer for
                 LREAL Values with the size of 32 Elements. *)

    (* Variable declaration*)
    VAR_INPUT
        Enable :            BOOL;  (* Inits the FIFO buffer on pos. edge *)
        Put :               BOOL;  (* Writes value  into FIFO on positive edge *)
        Get :               BOOL;  (* Reads value  into FIFO on positive edge *)
        PutValue :          LREAL; (* Value for Put  *)
    END_VAR

    VAR_OUTPUT
        Active:             BOOL;  (* Goes high with Enable *)
        NoOfElementsInFIFO: DINT;  (* Shows the number of elements in Buffer *)
        GetValue:           LREAL; (* Value for Get  *)
        Error:              BOOL;  (* is set at Error in FB *)
        ErrorID:            DINT;  (* -1 = FIFO empty, -2 = FIFO full *)
    END_VAR

    VAR CONSTANT
        MaxElements:        DINT:= 32;
        FIFO_Empty:         DINT:= -1;
        FIFO_Full:          DINT:= -2;
    END_VAR

    VAR (* local variables *)
        FIFOARRAY:          ARRAY[1..MaxElements] OF LREAL;
        EnableFlag:         BOOL;
        GetFlag:            BOOL;
        PutFlag:            BOOL;
        GetIndex:           DINT;
        PutIndex:           DINT;
    END_VAR

    (* Code Sequence*)

    IF Enable AND NOT EnableFlag THEN (* Initialisation at pos.edge of Enable *)
        EnableFlag := TRUE;
        Active := TRUE;
        ErrorID := 0;
        Error := FALSE;
        GetIndex := 1;
        PutIndex := 1;
        NoOfElementsInFIFO := 0;
        PutFlag := FALSE;
        GetFlag := FALSE;

    ELSIF NOT Enable AND EnableFlag THEN (* Disable FB at neg.edge of Enable *)
        EnableFlag := FALSE;
        Active := FALSE;
    END_IF

    IF Enable AND NOT Error THEN
        IF Put AND NOT PutFlag THEN (* pos.edge at Put *)
            PutFlag := TRUE;

            IF NoOfElementsInFIFO >= MaxElements THEN
                ErrorID := FIFO_Full;
                Error := TRUE;
            ELSE
                NoOfElementsInFIFO := NoOfElementsInFIFO + 1;
                FIFOARRAY[PutIndex] := PutValue;
                PutIndex := PutIndex + 1;
                IF PutIndex > MaxElements THEN
                    PutIndex := 1;
                END_IF
            END_IF
```

```
        ELSIF NOT Put AND PutFlag THEN (* neg.edge at Put *)
            PutFlag := FALSE;
        END_IF

        IF Get AND NOT GetFlag THEN (* pos.Edge of Get *)
            GetFlag := TRUE;

            IF NoOfElementsInFIFO < 1 THEN
                ErrorID := FIFO_Empty;
                Error := TRUE;
            ELSE
                NoOfElementsInFIFO := NoOfElementsInFIFO - 1;
                GetValue := FIFOARRAY[GetIndex];
                GetIndex := GetIndex + 1;

                IF GetIndex > MaxElements THEN
                    GetIndex := 1;
                END_IF
            END_IF

        ELSIF NOT Get AND GetFlag THEN (* neg.edge at Get *)
            GetFlag := FALSE;
        END_IF
    END_IF

END_FUNCTION_BLOCK
```

3. PLCopen Solutions for OMAC PackAL

The OMAC Packaging Workgroup has defined a PackAL specification as an Application Layer for Packaging. Within this specification, a number of Function Blocks have been defined at a higher level, like technology functions, specifically aimed at packaging machines. For more details on this set, refer to www.omac.org . If referred to PackAL in this specification, we refer to Version 1.01 of March 29, 2005.
One can represent the different levels in a diagram. The top layer is here represented by the OMAC PackML state diagram. However at this level other state diagrams can be implemented also.

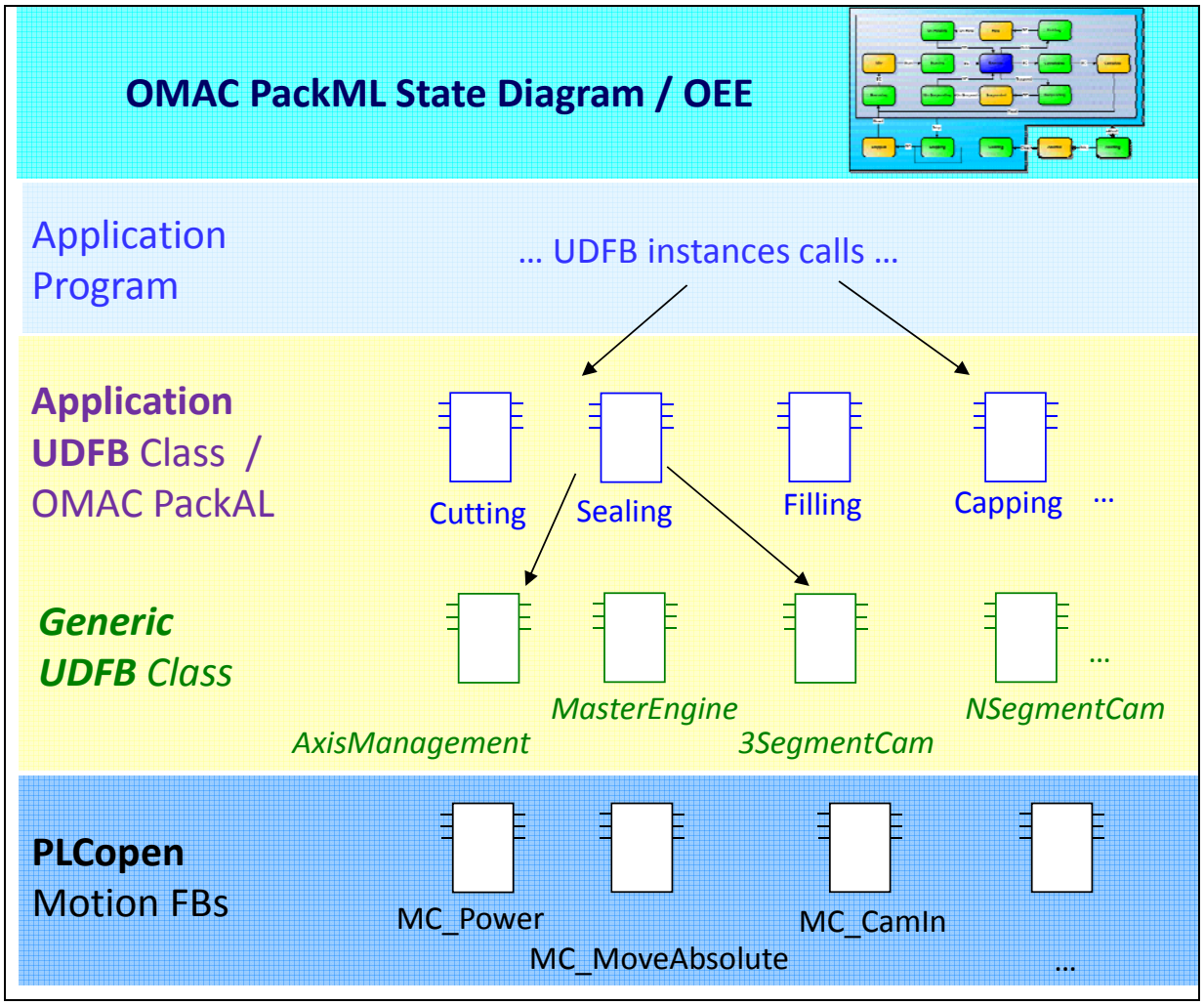


Figure 41 - Overview different levels of mapping OMAC

This specification shows how some of these functionalities can be created from the set of Function Blocks as defined by PLCopen for Motion Control as well as by the IEC 61131-3 standard concerning the basic functionalities, here represented at the lowest level. With these basic functionalities one can create generic User Derived Function Blocks, UDFBs, like MasterEngine, for which one encapsulates the lower level functionalities. The next higher level contains functionalities closer to the application level, like cutting and sealing. Based on these functionalities it becomes quite straightforward to implement the application program itself. As such, the coupling to a state diagram like PackML helps to structure the software development process, while providing a harmonized look-and-feel of the machines application.

PackAL has defined 3 sets of functionalities: Package Process Functions, Machine Communications, and Machine Behavior Organization. In this specification we only deal with the 'Packaging Process Functions'. They consist of:

- Wind / Unwind Axis (constant surface velocity)
- Wind / Unwind Axis (constant torque, ct mode)
- Dancer Control

- Registration
- Registration Correction
- Indexing
- Batch Counter
- Digital PLS (Digital CAM Switch)
- Set Override
- Jog Axis
- Flying Sync
- Gear_In with Dynamic Gear Factor
- Motion Command Stop with oriented halt

Some of these functionalities can be directly coupled to PLCopen Motion Control Function Blocks. However, some other functions need additional programming, or alternatives are possible especially concerning different inputs or feedback. For this reason we elaborated on the following functionalities:

- Wind / Unwind Axis with constant surface velocity and with constant torque, ct mode
- Dancer Control

'Enable' versus 'Execute'

The name 'Enable' refers to a level-sensitive input signal, whereas the name 'Execute' refers to an edge-trigger input signal.
The PackAL specification uses 'Enable' for all defined functions, creating a more continuous approach in the activation of the Function Blocks. The classical PLCopen approach is a more discrete version, where the inputs are maintained for a longer time till a new set of inputs are ready and the Function Block is re-triggered.
However, from Version 2.0 of Part 1, in par. 2.4.6 the input 'ContinuousUpdate' is defined. It is an extended input to all applicable Function Blocks.
If it is TRUE when the Function Block is triggered (rising 'Execute'), it will - as long as it stays TRUE – make the Function Block use the current values of the input variables and apply it to the ongoing movement. This does not influence the general behavior of the Function Block nor does it impact the state diagram. In other words it only influences the ongoing movement and its impact ends as soon as the Function Block is no longer 'Busy' or the input 'ContinuousUpdate' is set to FALSE. (Remark: it can be that certain inputs like 'BufferMode' are not really intended to change every cycle. However, this has to be dealt with in the application, and is not forbidden in the specification.)
If 'ContinuousUpdate' is FALSE with the rising edge of the 'Execute' input, a change in the input parameters is ignored during the whole movement and the original behavior of previous versions is applicable.
The 'ContinuousUpdate' is not a retriggering of the 'Execute' input of the Function Block. A retriggering of a Function Block which was previously aborted, stopped, or completed, would regain control on the axis and also modify its state diagram. Opposite to this, the 'ContinuousUpdate' only effects an ongoing movement.
Also, a 'ContinuousUpdate' of relative inputs (e.g. 'Distance' in MC_MoveRelative) always refers to the initial condition (at rising edge of 'Execute'). Example:

- MC_MoveRelative is started at 'Position' 0 with 'Distance' 100, 'Velocity' 10 and 'ContinuousUpdate' set TRUE. 'Execute' is Set and so the movement is started to position 100
- While the movement is executed (let the drive be at position 50), the input 'Distance' is changed to 130, 'Velocity' 20.
- The axis will accelerate (to the new 'Velocity' 20) and stop at 'Position' 130 and set the output 'Done' and does not accept any new values.

An alternative possibility for FBs that do not support the input 'ContinuousUpdate' is to create this input to match the 'Execute' behavior. Basically there are two ways of converting the continuous approach of the 'Enable' / 'InCommand' to the discrete 'Execute' / 'Done' approach:

1. 2Cycle approach: Set the 'Execute' in the first cycle, and 'Reset' in the next cycle automatically. This can be done with a simple exclusive OR (XOR) function which also checks the errors. The drawback is that it needs two cycles to set new values. However, the program to be used is very small.

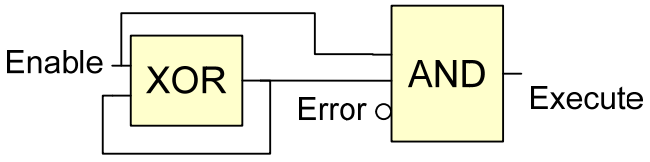


Figure 42 - Program example for 2Cycle approach

2. 1Cycle approach: Call the same instance of the relevant Function Block twice in the same cycle; one with ‘Execute’ SET and the other with RESET. The advantage is that every cycle the values are updated. The drawback is however that the program gets less self-explanatory, and the programmer has to program more in order to deal with the possible errors.

3.1. Wind / Unwind – General introduction

The OMAC Packaging Workgroup PackAL specification has defined two 'Wind' / 'Unwind' functionalities. However there are more solutions. This section deals with understanding of the winding / unwinding function. Following chapters will go more in detail of the mapping to the PackAL functions.

Winding with torque feedback can be done in several ways. A simple solution does not have any additional measuring inputs. Without measuring, the motor torque is used. This limits the torque range and accuracy due to the diameter transformation and variable losses

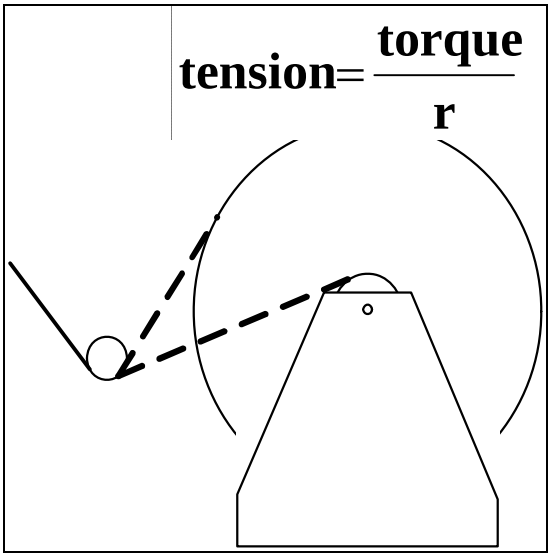
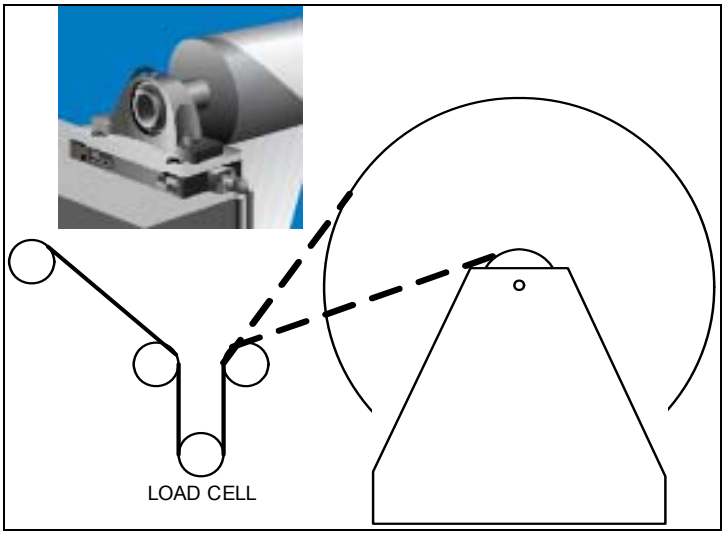
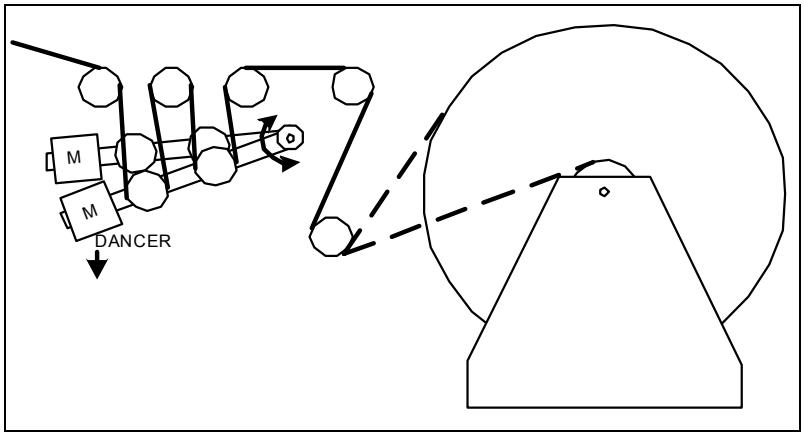


Figure 43 - Overview Winding / Unwinding

Another option could use the feedback from a load cell. A load cell is fast and accurate but expensive.



Yet another solution could be with a dancer control.



The dancer control could be simpler than the one described in the previous example.

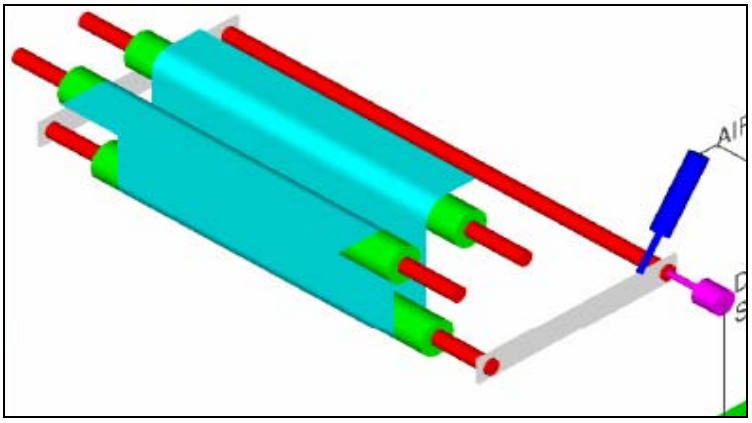


Figure 44 - Examples Dancer Control mechanics

3.2. OMAC PackAL Dancer Control

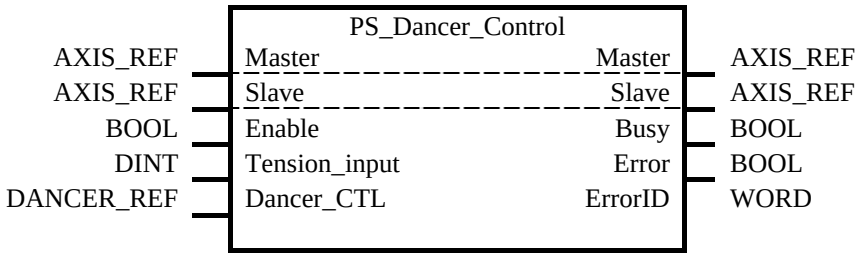
The OMAC Packaging Workgroup has defined a PackAL specification. In this specification a 'Dancer Control' functionality has been defined. This section maps this functionality on the existing PLCopen and IEC 61131-3 FBs.

Description from the OMAC packaging Workgroup:

FB-Name		**PS_Dancer_Control**		
This Function Block commands a controlled motion of an axis as slave of a dancer-coupled master axis. The Master axis is a physical or virtual axis. This FB provides the coupling between a slave axis (typically the infeed to the dancer) and a master axis (the outfeed of the dancer) via a dancer-PID controlled variable gearing factor.				
VAR_IN_OUT				
	B	Master	AXIS_REF	
	B	Slave	AXIS_REF	
VAR_INPUT				
	B	Enable	BOOL	Start at high level, stop at low level
	B	Tension_input	REAL	Input signal for tension of the Web, usually represented by the relative actual dancer position sensor
	B	Dancer_CTL	DANCER_REF	Structure with Dancer Controller Parameters
VAR_OUTPUT				
	B	Busy	BOOL	Executing status
	B	Error	BOOL	Signals that an error has occurred within Function Block
	E	ErrorID	WORD	Error identification

Notes:

- The FB's purpose is to generate and re-adjust a constant surface velocity (peripheral speed), relative to the master axis, for the rotary calibrated and controlled slave axis, depending on a dancer position. Via a position signal (dancer position signal), the tension between master (web) and slave (e.g. spool) is represented. For general use, the raw dancer position is often aligned to a "balance position" by an offset (and optional multiplier) in a first dancer scaling algorithm. The PID calculates the control value depending on the difference to a scaled command dancer position. This PID control value output then tunes the gear ratio between the master (web) and the slave (e.g. spool). A multiplier limits the distortion of the gear, offset adjusts to gear setpoint. Scaling algorithm, dancer balance position, PID factors, gear factor, delta and offset to the gear are application specific.
- Other possible implementations for Dancer Control, especially those with simpler two-switch control compared to PID control, may be represented by additional Function Blocks defined in the future.
- Tension_input is scanned in every cycle of the Function Block execution, other inputs in first cycle only.



Elements within the array structure of Structure **Dancer_CTL**: DANCER_REF:

B/E	Parameter	Type	Description
B	Tension_ctl	DINT	Target value for web tension, typically the target position for the dancer in balanced condition
B	GearRatio	REAL	Ratio of gear factor g(t) between master (web) and Slave (spool) to balance the dancer. Default is 1.0.
B	deltaGear	REAL	Delta scaling multiplier of PID output (-1.0 … +1.0) to GearRatio – must be smaller than 1 but never 0. deltagear would be e.g. 0.8 or 0.9 to limit the gear distortion
B	Gearoffset	REAL	Scaling offset to fit GearRatio distortion by satisfying equation g(t) = deltaGear* PIDout + Gearoffset

B	fKp	REAL	PID Control Proportional gain (P)
B	fTn	REAL	PID Control Integral gain Tn (I) [s]
B	fTv	REAL	PID Control Derivative gain Tv (D-T1) [s]
B	fTd	REAL	PID Control Derivative damping time Td (D-T1) [s]
B	Accel_limit	REAL	Corresponds indirectly, i.e. relative to a maximum master velocity, to a maximum permitted acceleration ($p_a = a_{SlaveMax} / v_{MasterMax}$). The limit parameter p_a corresponds to the reciprocal value of the run-up time $t_H = 1 / p_a$

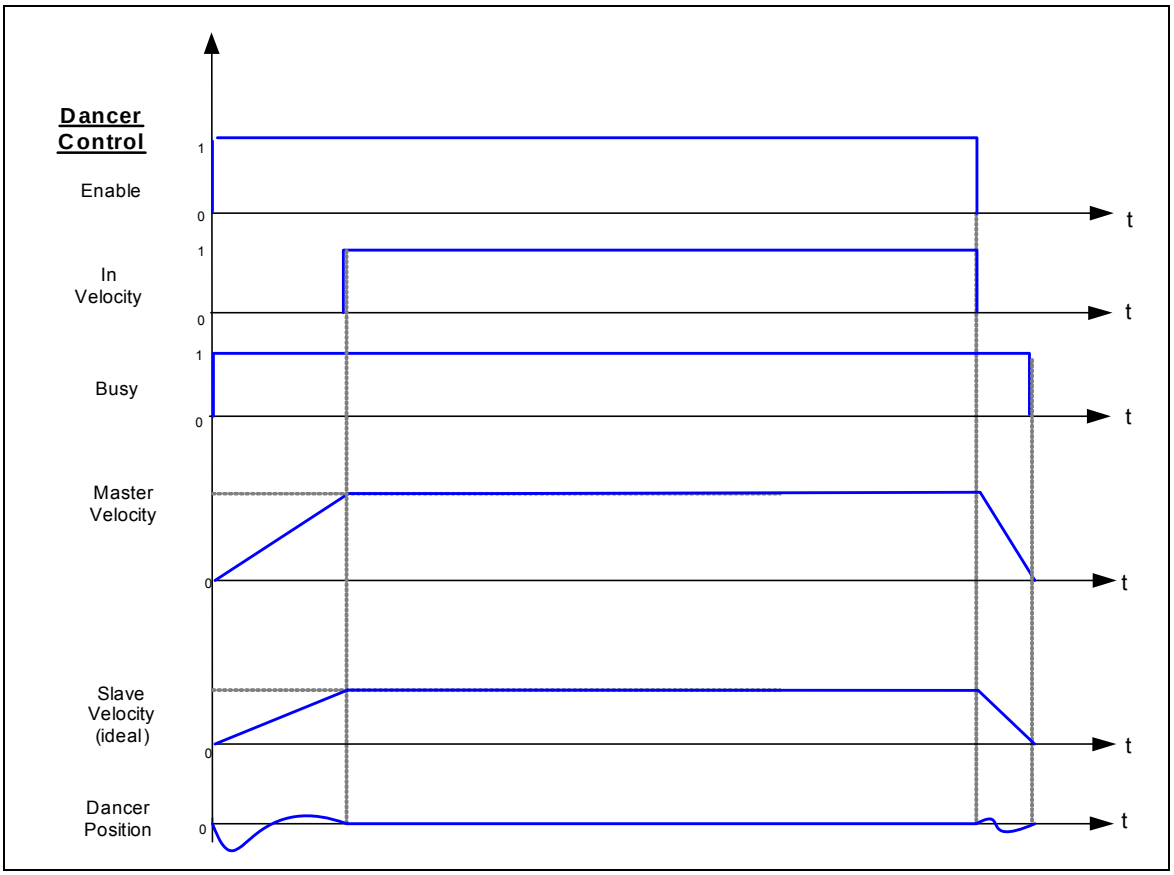


Figure 45 - PS_Dancer_Control timing diagram

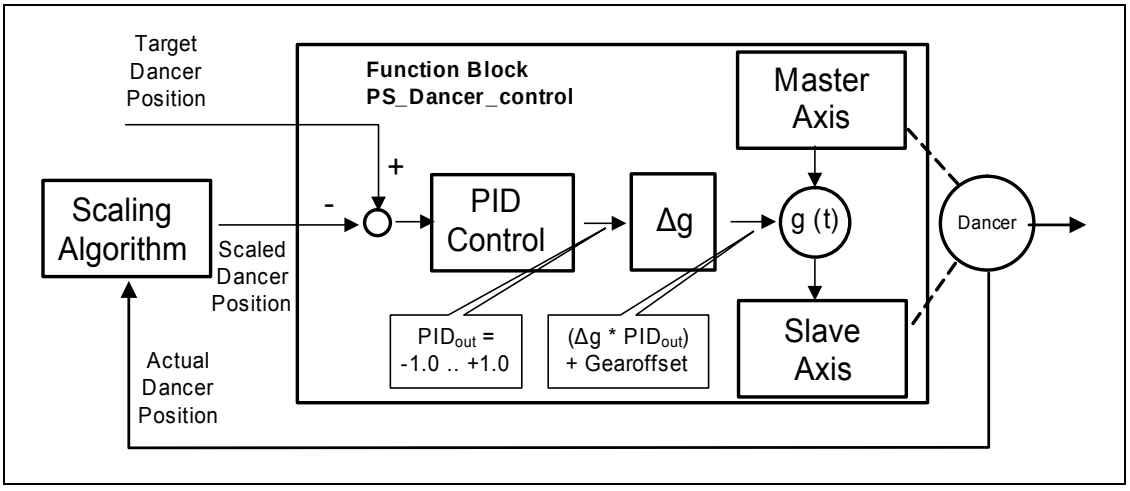


Figure 46 - PS_Dancer_Control Structure

This description can be met in the following way (with a 1Cycle solution):

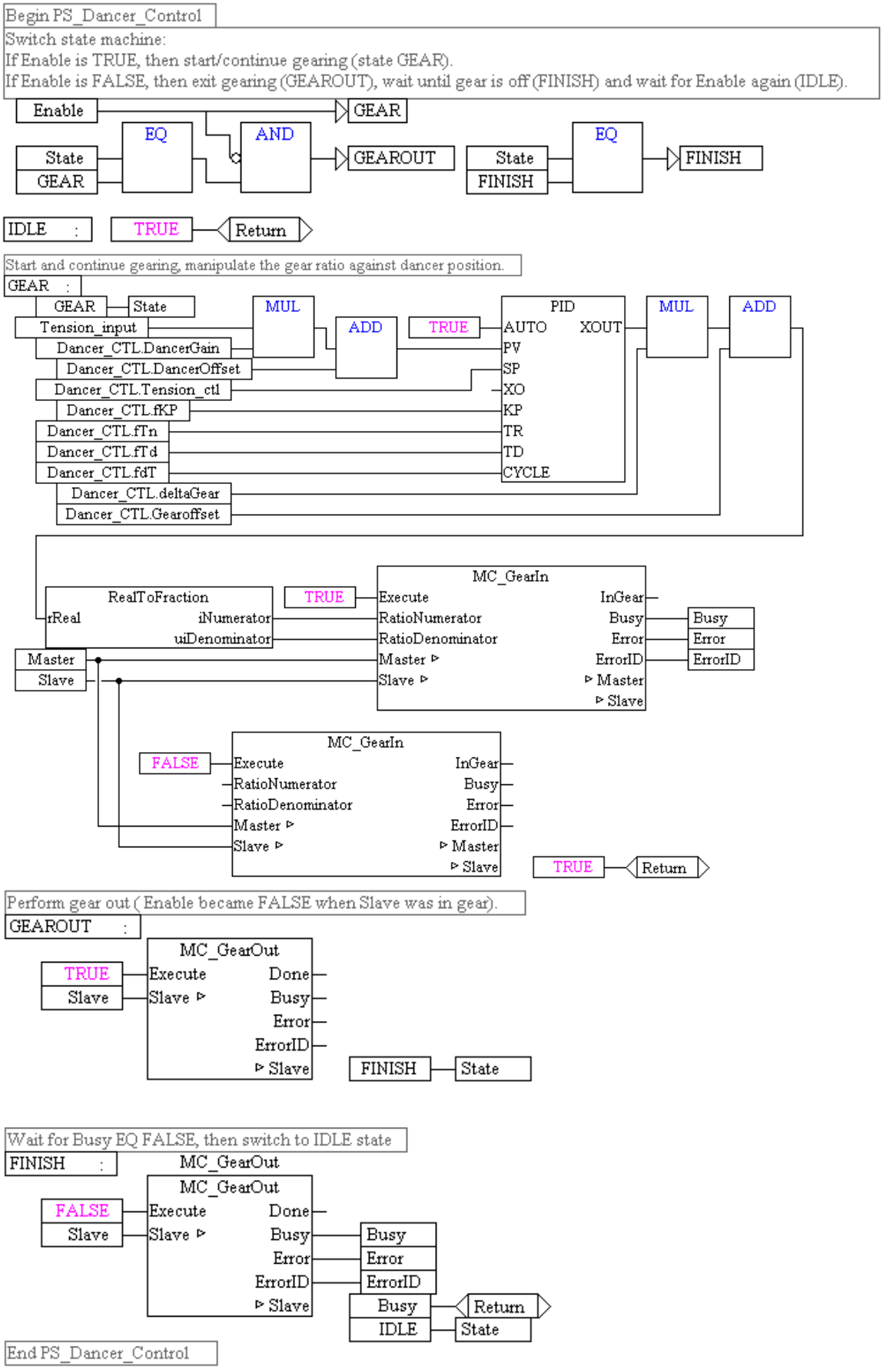


Figure 47 - Programming example in FBD

```
TYPE PS_Dancer_Control_State : (IDLE, GEAR, GEAROUT, FINISH); END_TYPE
```

```
TYPE
    DANCER_REF : STRUCT
        TensionCtl :      DINT;   (* Dancer target position *)
        GearRatio :       REAL;   (* manipulated gear ratio betwen master and slave *)
        DeltaGear :       REAL;   (* scaling multiplier of PID output *)
        GearOffset :      REAL;   (* scaling offset for GearRatio *)
        fKP :             REAL;   (* PID proportional constant *)
        fTn :             REAL;   (* PID integral constant *)
        fTv :             REAL;   (* PID derivative time constant *)
        fTd :             REAL;   (* PID derivative damping time *)
        fdT :             TIME;   (* PID cycle time *)
        AccelLimit :      REAL;   (* maximum slave acceleration relative to master velocity *)
                                  (* Configuration for Dancer Scaling Algorithm: *)
        DancerGain :      REAL;   (* scaling multiplier of dancer actual value *)
        DancerOffset :    REAL;   (* scaling offset of dancer actual value *)
    END_STRUCT
END_TYPE

(* enum *)
TYPE
    PS_Dancer_Control_State :
        (IDLE,
         GEAR,
         GEAROUT,
         FINISH);
END_TYPE
```

Alternatively, the Structured Text, ST, program could look like this (same data types and enums as above are used):

```
FUNCTION_BLOCK PS_Dancer_Control
    VAR_IN_OUT
        Master :        AXIS_REF;
        Slave :         AXIS_REF;
    END_VAR

    VAR_INPUT
        Enable :        BOOL;
        Tension_input : REAL;
        Dancer_CTL :    DANCER_REF;
    END_VAR

    VAR_OUTPUT
       Busy :           BOOL;
       Error :          BOOL;
       ErrorID :        WORD;
    END_VAR

    VAR
       GearIn :         MC_GearIn;
       GearOut :        MC_GearOut;
       RealToFraction : REAL_TO_FRACTION;
       PidDancer :      PID;
       PidOut :         REAL;
       State :          PS_Dancer_Control_State;
    END_VAR

    (* Begin PS_Dancer_Control *)
    (* Switch state machine: If Enable is TRUE, then start/continue gearing (state GEAR).
       If Enable is FALSE, then exit gearing (state GEAROUT), then wait until gear is off (state FINISH)
       and wait for Enable again (state IDLE). *)

    IF ENABLE THEN
        State := GEAR;
    ELSIF State = GEAR THEN
        State := GEAROUT;
    END_IF

    (* Execute state machine: *)
    CASE State OF
    (*-------------------------------------------------*)
    IDLE:
        GearIn(Execute := FALSE,
               Master := Master,
               Slave := Slave); (* prepare Execute for rising egde *)
```

```
        GearOut(Execute := FALSE,
                Slave := Slave); (* prepare Execute for rising egde *)

    (*-------------------------------------------------*)
    GEAR:
        (*Start and continue gearing, manipulate the gear ratio against dancer position.*)
        PidDancer(AUTO := TRUE,
                  PV := Tension_input * Dancer_CTL.DancerGain + Dancer_CTL.DancerOffset,
                  SP := Dancer_CTL.TensionCtl,
                  (* XO := , not used *)
                  KP := Dancer_CTL.fKP,
                  TR := Dancer_CTL.fTn,
                  TD := Dancer_CTL.fTd,
                  CYCLE := Dancer_CTL.fdT);

        PidOut := PidDancer.XOUT;

        RealToFraction(rReal := Dancer_CTL.DeltaGear * PidOut + Dancer_CTL.GearOffset);

        GearIn(Execute := TRUE,
               RatioNumerator := RealToFraction.iNumerator,
               RatioDenominator := RealToFraction.uiDenominator,
               Master := Master,
               Slave := Slave);

        Busy := GearIn.Busy;
        Error := GearIn.Error;
        ErrorID := GearIn.ErrorID;

    (*-------------------------------------------------*)
    GEAROUT:
        (* Perform gear out ( Enable became FALSE when Slave was in gear) *)
        GearOut(Execute := TRUE,
                Slave := Slave,

        Busy := GearOut.Busy;
        Error := GearOut.Error;
        ErrorID := GearOut.ErrorID;
        State := FINISH;

    (*-------------------------------------------------*)
    FINISH:
        (* Wait for GearOut.Busy = FALSE, then switch to IDLE state. *)
        GearOut(Execute := FALSE,
                Slave:= Slave,
        Busy := GearOut.Busy;
        Error := GearOut.Error;
        ErrorID := GearOut.ErrorID;

        IF GearOut.Busy = FALSE THEN
            State := IDLE;
        END_IF

        (*-------------------------------------------------*)
    END_CASE;

END_FUNCTION_BLOCK
```

Notes:

- A FB is needed for the conversion of *Real to Fraction* (Numerator/Denominator) as input for MC_GearIn.
- Two parameters are needed to configure the *Dancer Scaling Algorithm* (see last 2 parameters in DANCER_REF)

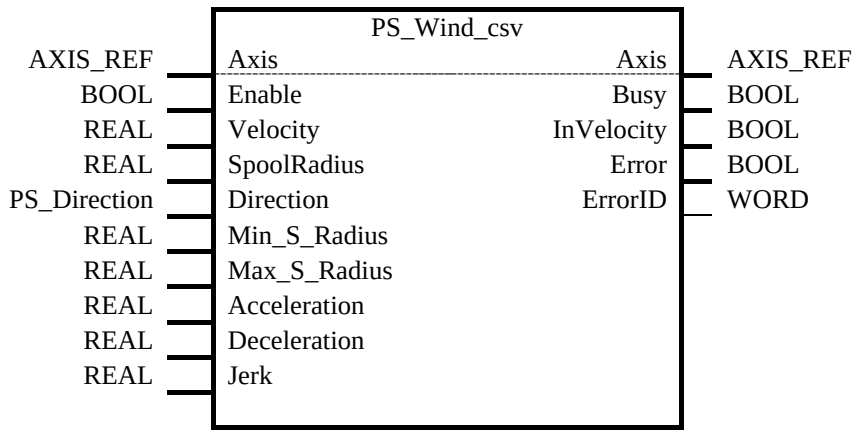
3.3. PackAL Wind / Unwind Axis (constant surface velocity, csv mode)

Definition in PackAL:

FB-Name		**PS_Wind_csv**		
This Function Block commands a controlled motion at a specified circumflex velocity for a wind / unwind axis. The circumflex is calculated from the radius of a wind/unwind spool, measured with a sensor.				
VAR_IN_OUT				
	B	Axis	AXIS_REF	
VAR_INPUT				
	B	Enable	BOOL	Start the motion at high level, stop at low level
	B	Velocity	REAL	Value of the maximum Surface velocity [u/s]
	B	SpoolRadius	REAL	Value of the radius of the spool [u]
	B	Min_S_Radius	REAL	Value of the Minimal Spool Radius, initial value [u]
	B	Max_S_Radius	REAL	Value of the Maximal Spool Radius, initial value [u]
	E	Direction	PS_Direction	Enum type (pos, neg)
	E	Acceleration	REAL	Value of the acceleration (increasing energy of the motor) [u/s^2]
	E	Deceleration	REAL	Value of the deceleration (decreasing energy of the motor) [u/s^2]
	E	Jerk	REAL	Value of the Jerk [u/s^3]
VAR_OUTPUT				
	B	Busy	BOOL	Executing status
	B	InVelocity	BOOL	command circumflex velocity phase active
	B	Error	BOOL	Signals that error has occurred within Function Block
	B	ErrorID	WORD	Error Number

Note: This FB contains the mathematical calculation for coupling between a rotary slave axis and a translatory master axis. Its purpose is to generate and re-adjust a constant surface velocity (peripheral speed), relative to the master axis, for the rotary calibrated and controlled slave axis, depending on its spool diameter. Via a signal proportional to the spool radius, the spool radius of this slave axis is automatically evaluated and used for the calculation. This radius must never have the value 0.0 mm, since otherwise a calculation is no longer possible.

- The FB decelerates the axis to stop while winding above the ‘Max_S_Radius’
- The FB decelerates the axis to stop while unwinding below the ‘Min_S_Radius’
- ‘SpoolRadius’ needs to satisfy ‘Min_S_Radius’ < ‘SpoolRadius’ < ‘Max_S_Radius’ to execute the action, otherwise ‘Error’ = True, ‘ErrorID’ set
- Error ID is implementation specific



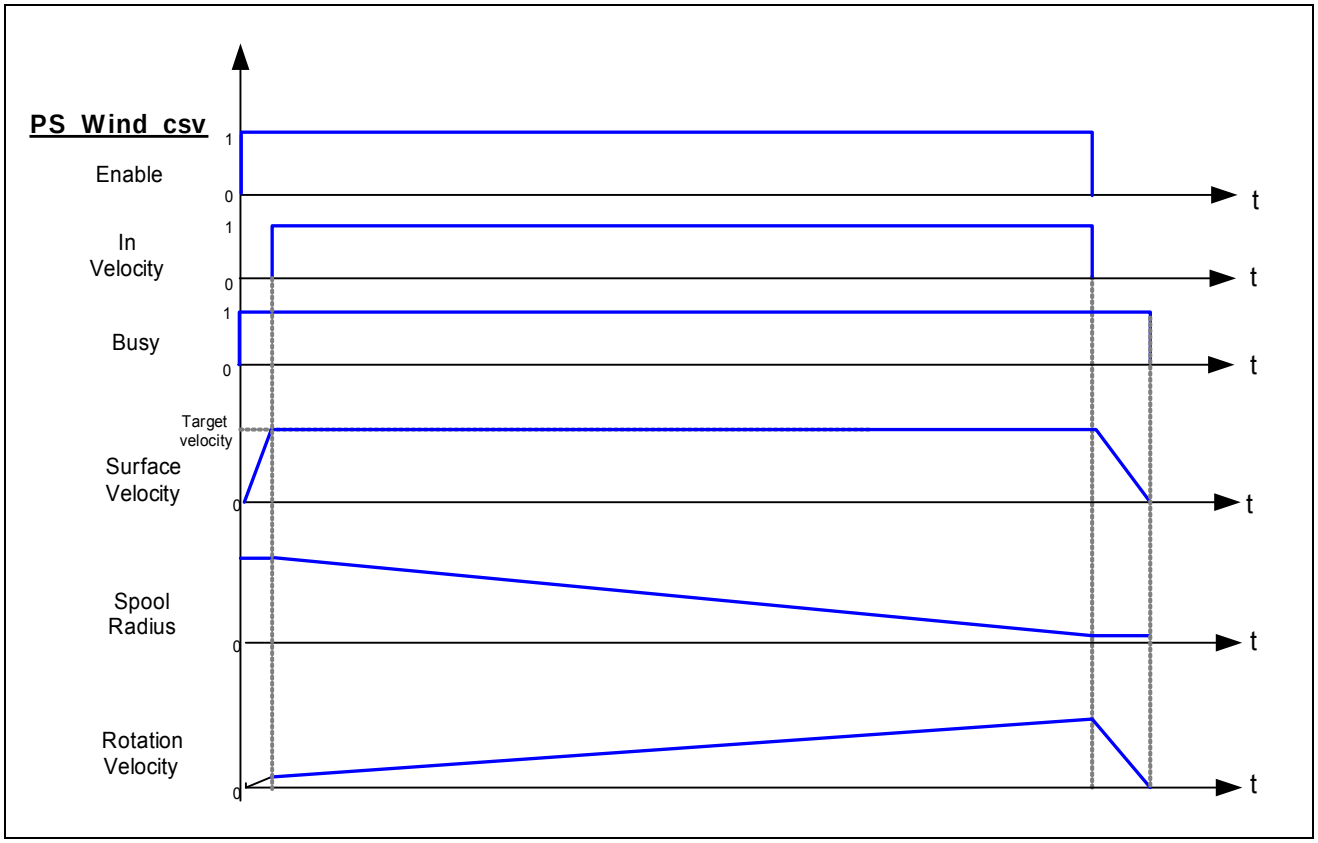


Figure 48 - Timing Diagram PS_wind_csv

End of definition PackAL

A simple implementation could be as follows for a 2Cycle approach (note: the conversion as shown in the previous chapter is not shown here completely).

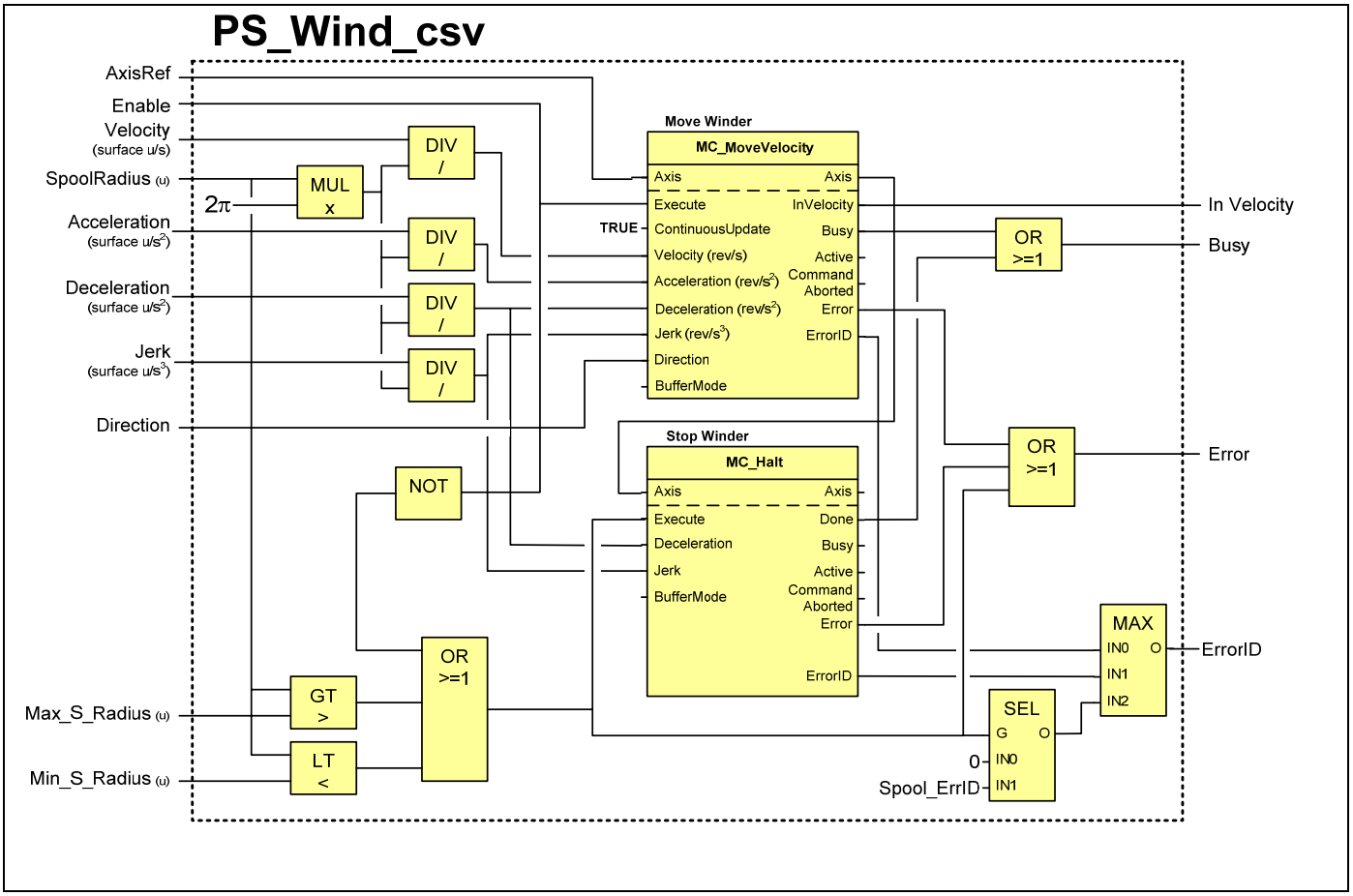


Figure 49 - Programming example Winding part 1

NOTES:
- As the spool radius is changing the units of MC_MoveVelocity are defined in rev/sec.
- Other possibility is injecting the 'SpoolRadius' into the axis scaling parameters, but this is more vendor specific operation.

Explanation:
The 'SpoolRadius' times 2 pi defines the relationship towards the 'Velocity', the 'Acceleration', 'Deceleration', and 'Jerk' (all surface related). These are issued as inputs for the MC_MoveVelocity Function Block.
There is a check included if the 'SpoolRadius' is between the 'Max_S_Radius' and 'Min_S_Radius'. If outside this range, it stops the winder while setting the 'Error' output. The combination of the two 'ErrorIDs' only works if there is only one 'Error' at a time, i.c. one 'ErrorID' valid.

PackAL Wind / Unwind Axis (constant tension, ct mode)

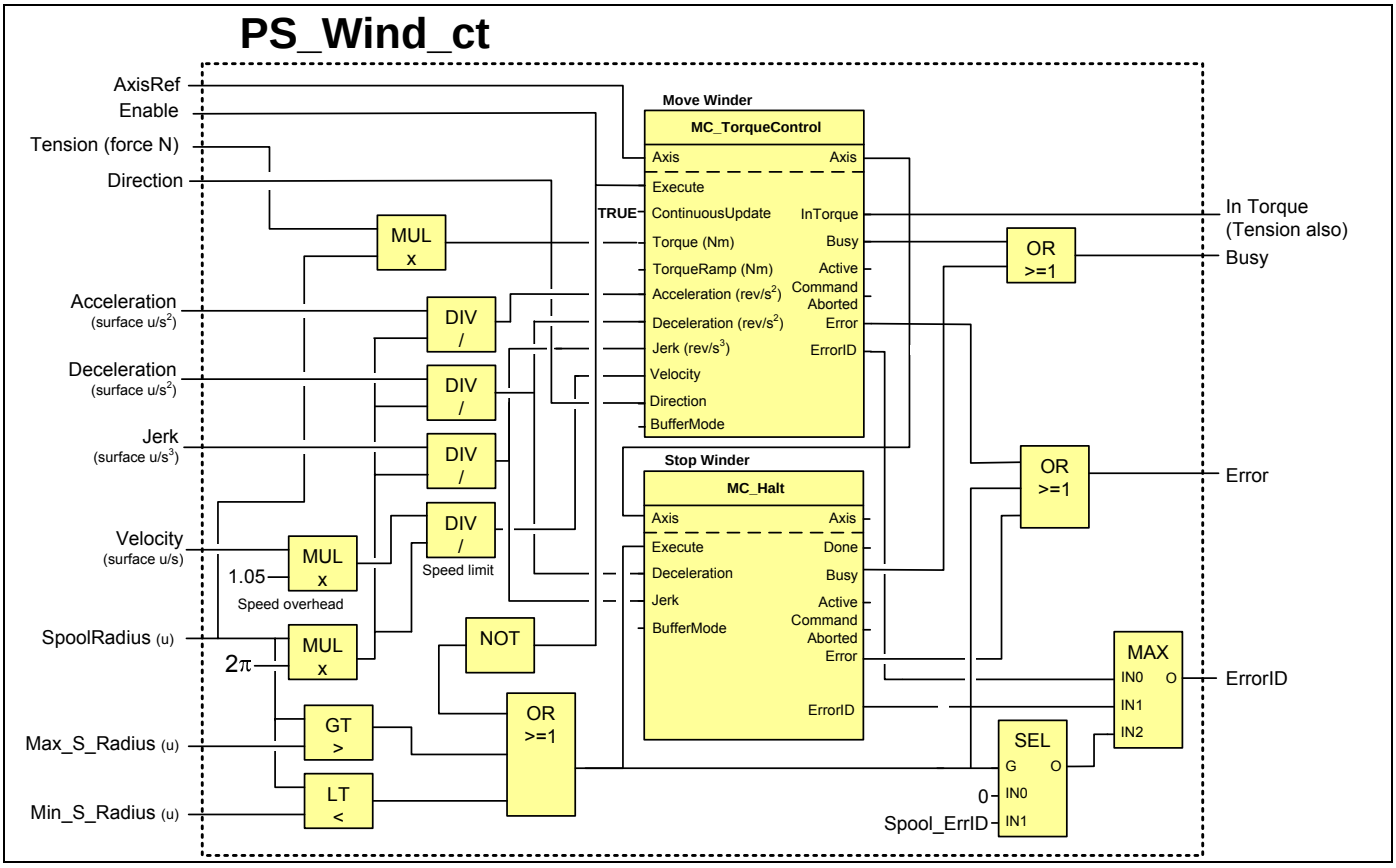


Figure 50 - Programming example Winding part 2

NOTES :

- The example assumes Tension (force) reference for the Function Block
- As the spool radius is changing the units of MC_MoveVelocity are defined in rev/sec.
- Other possibility is injecting the 'SpoolRadius' into the axis scaling parameters, but this is more vendor specific.
- Overspeed factor is needed in Speed limit calculation from 'SpoolRadius'