PLCopen®
for efficiency in automation

PLCopen - Promotional Committee 2
–
Training

Application Examples for Motion Control

Porting “Function blocks for motion control” into OOP

Version 1.0 – Official Release

Date: November 2, 2022

Total number of pages: 46

Application Example for Motion Control

The following paper is a document created within the PLCopen Promotional Committee 2 – Training.

It summarizes the results of the PLCopen Promotional Committee meetings, containing contributions of its members as well as external sources:

Name	*Company*
John Dixon	ABB
Dominik Franz	ABB
Klaus Bernzen	Beckhoff
Wolfgang Doll	Codesys
Daniel Wall	Eaton
Anders Lekve Brandseth	Framo
Rene Simon	Hochschule Harz
Yves de la Broise	IntervalZero
Yo Takahashi	Mitsubishi Electric
Saele Beltrani	SACMI
Filippo Venturi	SACMI
Juliane Fischer	Technische Universität München
Georg Rempfler	Wyon
Eelco van der Wal	PLCopen

Change Status List:

Version number	Date	Change comment
V 0.1	Sept 8, 2021	First version by Juliane Fischer, based on the video with Yves de la Broise.
V 0.1a	Sept 10, 2021	Discussed in the webmeeting & commented
V 0.2	Sept 23	As result of the webmeeting Sept. 22. First doc version with template
V 0.3	Oct. 6, 2021	As a result of the webmeeting on Oct. 6
V 0.4	Dec. 17, 2021	Adding draft of the Axis interface
V 0.5	Feb. 11, 2022	Adding the OO example for warehousing
V 0.51	April 20, 2022	Intro reworked and discussed during the webmeeting
V 0.52	May 4, 2022	Open issue's introduction resolved.
V 0.53	May 19, 2022	After webmeeting, adding multi-axes definitions and inclusion of 3rd example.
V 0.54	June 15, 2022	As result from the input from Yves, adding of Part 4 and the webmeeting
V 0.55	June 29, 2022	With input from Juliane Fischer and webmeeting
V 0.99	June 30, 2022	Published as Release for Comments till August 31, 2022
V 1.0	Nov. 2, 2022	Published as Version 1.0 – Official Release

Contents

List of figures

1 Introduction to this document

With the published specification "Function blocks for motion control (formerly Part 1 and Part 2)", the PLCopen Task Force Motion Control provided a set of standardized Function Blocks to ease modularization and reuse of motion control software. This document presents an object-oriented implementation of the motion control specification, which can be combined with the set of procedural standard Function Blocks (FBs). The general design of the proposed object-oriented (OO) implementation is a single *Axis Class* implementing different functions as *Methods* instead of formerly used multiple FBs. A benefit of the proposed software design is the compatibility with procedural motion control FBs: The standard Motion Control libraries can call the *Axis Class* internally to combine both approaches in one application. Thus, the user of the OO implementation needs not to be familiar with the detailed OO principles or language elements for using it.

As common in object-oriented programming (OOP), an *interface* is used to define the motion standard since it describes how a *class* is presented to the outside (sometimes including the behavior). More precisely, an *interface* is the definition of the functionalities that a *class* may implement. The *class* is the actual implementation of the defined functionalities, including vendor-specific aspects. Correspondingly, this document standardizes a motion *interface*. For using this standard, an *axis class* needs to be implemented, which follows ("implements") this standardized motion *interface*. In short: the *interface* defines the functionalities, but not how they are implemented (their content), which is done vendor-specific in an axis class.

1.1. Goals of this working group

In this document we use three application examples:

(1) A labeling example where a label is put on a product on a belt
(2) A warehousing example, where a pallet is moved out of a warehouse shelve
(3) A combination of multiple axes FBs: CamIn and GearIn.

Via these examples it is shown how the standardized FBs from the PLCopen motion control specification (https://www.plcopen.org/technical-activities/motion-control) can be ported to OOP by using a standardized interface *itfAxis* as introduced below. To apply the standard in a vendor-specific implementation, the programmer develops a *class*, which implements the interfaces *itfAxis* and, thus, has all the functionalities standardized in *itfAxis* without implementation. Then the actual, vendor-specific implementation of these functionalities is programmed.

The advantage of the proposed interface *itfAxis* is that one can decide how to program: on the one hand, the standard FBs can be used, and they can internally call the *itfAxis* methods. On the other hand, it is possible to program in OOP by using the defined methods to start a new command, get the status of an axis, and update or abort a command.

The details on the proposed *interface* and the contained *methods* as well as several user-defined data types are introduced below.

This document focuses on the motion control part of the axes only. In real projects, the axis class will have many other properties and methods for communication, hardware configuration, and additional aspects. For simplicity, these are not explained in this document.

2 Elements of the OOP Motion Control Library

The starting point for porting the motion FBs to OOP is the definition of the interface *itfAxis* as standardization for the *axis class* as a representation of the PLCopen motion control specification. Initially, several *ENUMS* are defined, which are used inside the interface *itfAxis* (cf. Figure 1: top).

2.1. References

In this document we will not see the AXIS_REF, CAM_REF, MC_INPUT_REF, MC_OUTPUT_ REF, etc., because they are vendor specific and not used as inputs for any Method.

We expect vendors to add and ID property to the interfaces in a vendor specific way.

2.2. Commands

To represent a previously made command, an *itfCommand* interface and its various extensions for motion control are defined. The itfCommand contains "Getter Functions" to query the status of the command. To be compliant with the IEC 61131-3 standard, the "Getter Functions" are implemented as *methods*. (Note: the actual values can be implemented as properties, which can be more compact compared to "Getter Functions" (methods). Thus, they are simpler to use. Only a Get-Method of these properties is required for reading the current values of the linked variables.) An *Abort method* is defined for canceling a command that is running. In preparation for future control strategies (for example, event-driven programming as defined by the IEC 61499), the method *Wait* is defined. In case of event-driven programming, synchronous calls would be possible, and this command would enable waiting on a command to finish or time-out. It is not included in the current motion control specification but represents an extension that could be used in event-driven architectures for synchronous calls.

The *itfAxisCommand* interface extends *itfCommand* by adding an Update method that can be called when the inputs of a move change. This mimics the functionality of the ContinuousUpdate input of classic FBs. It is used to update the call parameters (position, acceleration, etc.) of a command like *MoveAbsolute.*

Overall, the defined commands (like *itfCommand*, *itfAxisCommand*, etc.) are generic and can be used with every API, Application Programming Interface (absolute move, relative move, velocity move, halt or stop). This allows to use schedulers, error handlers and the like.

2.3. Interface Axis definition

The *itfAxis* interface itself is organized in different sub-folders to group the functionalities according to their categories (cf. Figure 1, middle). These categories correspond to the FBs from the motion control specification. The first folder is the folder *ActualValues*, which contains the ActualPosition, ActualTorque and ActualVelocity "Getter Function" to query the actual status of the axis.

In the second folder *Control*, nine methods are contained for the axis control such as *Power*, *Reset* and *SetPosition.* Using a Stop / Gear / Cam command, the axis is moved to a specific state: either Stop or Synchronized_Motion. Since the setting of the Execute to FALSE does not make sense in OOP, a *Release Method* is added to return the axis to the *StandStill* state.

The third and fourth folders contain methods for *Motion* – for example, the method *MoveAbsolute*, for an absolute movement of an axis. The method comprises input parameters but no outputs. Its status is returned with a return variable of the Class-type *itfAxisCommand*, which contains variables related to motion and a reference number of the command. The user can adapt this interface to include all additional, required information. When a method like *MoveAbsolute* is called, the command status is returned. Simple programs can ignore the return value and use the status of the axis instead to check when the triggered movement is completed (axis returns to the status *StandStill*). The return value enables to follow up on the command by using the methods of the *itfAxisCommand* interface.

Finally, the folder *Status* contains 13 properties (see Chapter 5.2.2 Status) to observe the state of the axis. The motion control specification defines four FBs for this, namely, *MC_ReadStatus*, *MC_ReadAxisInfo*, *MC_ReadAxisError*, and *MC_ReadMotionState*. The return values of these standard FBs are turned into properties such as the property *Status*. In OOP it is not practical to have all axis states (e.g., standstill, error, stop, etc.) as individual Boolean properties. For reasons of simpler manipulation, a property of an ENUM-type is defined instead. All possible states of the axis are merged in the ENUM *AXIS_STATUS* (cf. Fig. 1 and Chapter 5.1 ENUMs). The property *Status* returns a variable of the ENUM *AXIS_STATUS* type. The property *MotionStatus* is defined in a similar way.

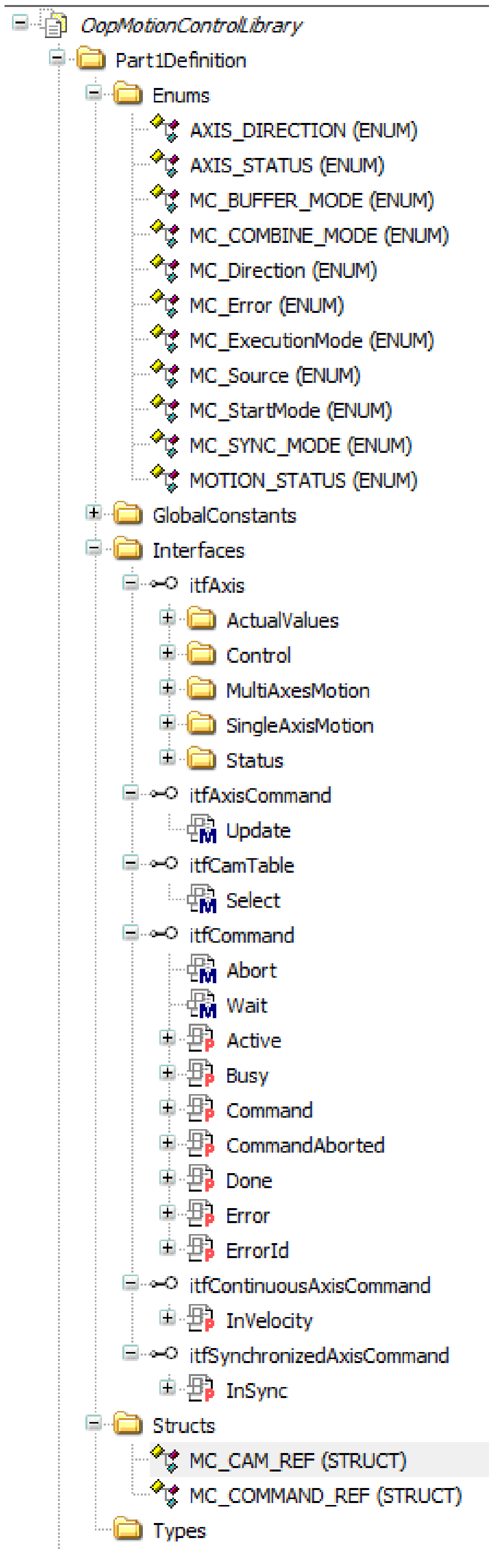


Figure 1: Project tree showing *ENUMS*, interfaces and *STRUCTS* of the OOP Motion Control Library

3 Command interface definitions

3.1. Base itfCommand

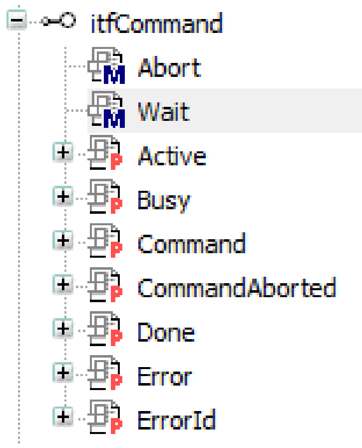


Figure 2: Interface implemented by Classes returned by an a-synchronous method

This is the interface for any vendor implemented Command class. This class contains the status of a running command and is not limited to the Motion Control.

3.1.1. Properties

Name	Access	Type
Done	Read	BOOL
Busy	Read	BOOL
Active	Read	BOOL
CommandAborted	Read	BOOL
Error	Read	BOOL
ErrorId	Read	MC_ERROR

For a description of the properties, see PLCopen Motion Control Part 1.

The properties *InVelocity* and *InSync* are extensions of this base type, see Chapter 3.3 and 3.4 hereunder.

3.1.2. Methods

The interface does not show it, but the Command classes will implement an internal Get method that will update the properties.

METHOD Abort : **MC_ERROR**

VAR_INPUT
END_VAR

END_METHOD

The Method Wait hereunder is a placeholder for IEC-61499:

METHOD Wait : **MC_ERROR**

VAR_INPUT

Timeout : **TIME;**

AbortOnTimeout : **BOOL;**

END_VAR

END_METHOD

3.2. itfAxisCommand : Extends itfCommand

Extension of *itfCommand* for Axis motion methods.

3.2.1. Added Methods

METHOD Update : **MC_ERROR**

VAR_INPUT

Position : **REAL**;
Velocity : **REAL**;
EndVelocity : **REAL**;
Acceleration : **REAL**;
Deceleration : **REAL**;
Jerk : **REAL**;

END_VAR

END_METHOD

3.3. itfContinuousAxisCommand : Extends itfAxisCommand

For moves that set the axis in ContinuousMotion state.

3.3.1. Added Properties

Name	Access	Type
InVelocity	Read	BOOL

3.4. itfSynchronizedAxisCommand : Extends itfAxisCommand

For moves that set the axis in SynchronizedMotion state.

3.4.1. Added Properties

Name	Access	Type
InSync	Read	BOOL

4 itfCamTable interface definition

4.1. Methods

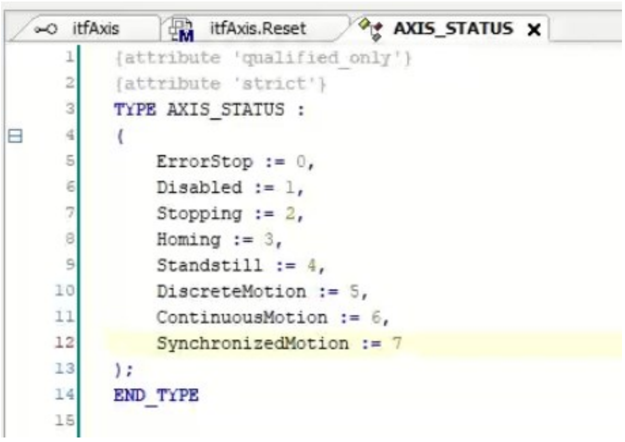
METHOD Select : **itfCommand**

VAR_INPUT
Periodic : **BOOL**;
MasterAbsolute: **BOOL**;
SlaveAbsolute: **BOOL**;
CamTable: **MC_CAM_REF**;
END_VAR

END_METHOD

5 itfAxis interface definition

5.1. ENUMs




```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE AXIS_STATUS :
(
    ErrorStop := 0,
    Disabled := 1,
    Stopping := 2,
    Homing := 3,
    Standstill := 4,
    DiscreteMotion := 5,
    ContinuousMotion := 6,
    SynchronizedMotion := 7
);
END_TYPE
```

Figure 3: ENUM AXIS_STATUS defining all the possible states of an axis

No.	MC_AXIS_STATUS
	mcErrorStop
	mcDisabled
	mcStandstill
	mcHoming
	mcStopping
	mcDiscreteMotion
	mcContinuousMotion
	mcSynchronizedMotion

No.	MC_MOTION_STATUS
	mcConstantVelocity
	mcAccelerating
	mcDecelerating

No.	MC_AXIS_DIRECTION
	mcDirectionPositive
	mcDirectionNegative

For a description of the status, see PLCopen Motion Control Part 1.

5.2. Properties

5.2.1. Actual values

Name	Access	Type
ActualPosition	Read	REAL
ActualTorque	Read	REAL
ActualVelocity	Read	REAL

For a description of the properties, see PLCopen Motion Control Part 1.

5.2.2. Status

Name	Access	Type
AxisWarning	Read	BOOL
CommunicationReady	Read	BOOL
Direction	Read	MC_AXIS_DIRECTION
ErrorId	Read	MC_ERROR
HomeAbsSwitch	Read	BOOL
IsHomed	Read	BOOL
LimitSwitchNegative	Read	BOOL
LimitSwitchPositive	Read	BOOL
MotionStatus	Read	MC_MOTION_STATUS
PowerOn	Read	BOOL
ReadyForPowerOn	Read	BOOL
Simulation	Read	BOOL
Status	Read	MC_AXIS_STATUS

For a description of the properties, see PLCopen Motion Control Part 1.

5.3. Methods

5.3.1. Control

METHOD Power : **itfCommand**

VAR_INPUT
Enable : **BOOL**;
EnablePositive : **BOOL**;
EnableNegative : **BOOL**;
END_VAR

END_METHOD

METHOD ReadBoolParameter : **MC_ERROR**

VAR_INPUT
ParameterNumber : **INT**;
END_VAR

VAR_OUTPUT
    Value : **BOOL**;
END_VAR

END_METHOD

METHOD ReadParameter : **MC_ERROR**

VAR_INPUT
    ParameterNumber : **INT**;
END_VAR

VAR_OUTPUT
    Value : **INT**;
END_VAR

END_METHOD

METHOD Release : **MC_ERROR**

VAR_INPUT
END_VAR

END_METHOD

METHOD Reset : **MC_ERROR**

VAR_INPUT
END_VAR

END_METHOD

METHOD SetOverride : **MC_ERROR**

VAR_INPUT
    VelFactor : **REAL**;
    AccFactor : **REAL**;
    JerkFactor : **REAL**;
END_VAR

END_METHOD

METHOD SetPosition : **itfCommand**

VAR_INPUT
    Position : **REAL**;
    Relative : **BOOL**;
    ExecutionMode : **MC_EXECUTION_MODE**;
END_VAR

END_METHOD

METHOD WriteBoolParameter: **MC_ERROR**

VAR_INPUT
ParameterNumber : **INT**;
Value : **BOOL**;
END_VAR

END_METHOD

METHOD WriteParameter: **MC_ERROR**

VAR_INPUT
ParameterNumber : **INT**;
Value : **REAL**;
END_VAR

END_METHOD

METHOD DigitalCamSwitch : **itfCommand**

VAR_INPUT
Switches : **MC_CAMSWITCH_REF**;
Outputs : **MC_OUTPUT_REF**;
TrackOptions : **MC_TRACK_REF**;
Enable : **BOOL**;
EnableMask : **DWORD**;
ValueSource : **MC_SOURCE**;
END_VAR

END_METHOD

METHOD TouchProbe : **itfCommand**

VAR_INPUT
TriggerInput : **MC_TRIGGER_REF**;
WindowOnly : **BOOL**;
FirstPosition : **REAL**;
LastPosition : **REAL**;
END_VAR

END_METHOD

METHOD AbortTrigger : **itfCommand**

VAR_INPUT
    TriggerInput : **MC_TRIGGER_REF**;
END_VAR

END_METHOD

5.3.2. Single Axis Motion

METHOD Home : **itfAxisCommand**

VAR_INPUT
    Position : **REAL**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD Stop : **itfAxisCommand**

VAR_INPUT

    Deceleration : **REAL**;
    Jerk : **REAL**;
END_VAR

END_METHOD

METHOD Halt : **itfAxisCommand**

VAR_INPUT
    Deceleration : **REAL**;
    Jerk : **REAL**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD MoveAbsolute : **itfAxisCommand**

VAR_INPUT
    Position : **REAL**;
    Velocity : **REAL**;
    Acceleration : **REAL**;
    Deceleration : **REAL**;
    Jerk : **REAL**;
    Direction : **MC_DIRECTION**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD MoveRelative : **itfAxisCommand**

VAR_INPUT
    Distance : **REAL**;
    Velocity : **REAL**;
    Acceleration : **REAL**;
    Deceleration : **REAL**;
    Jerk : **REAL**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD MoveAdditive : **itfAxisCommand**

VAR_INPUT
    Distance : **REAL**;
    Velocity : **REAL**;
    Acceleration : **REAL**;
    Deceleration : **REAL**;
    Jerk : **REAL**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD MoveSuperimposed : **itfAxisCommand**

VAR_INPUT
Distance : **REAL**;
VelocityDiff : **REAL**;
Acceleration : **REAL**;
Deceleration : **REAL**;
Jerk : **REAL**;
END_VAR

END_METHOD

METHOD HaltSuperimposed : **itfAxisCommand**

VAR_INPUT
Deceleration : **REAL**;
Jerk : **REAL**;
END_VAR

END_METHOD

METHOD MoveVelocity : **itfContinousAxisCommand**

VAR_INPUT
Velocity : **REAL**;
Acceleration : **REAL**;
Deceleration : **REAL**;
Jerk : **REAL**;
BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD MoveContinuousAbsolute : **itfContinousAxisCommand**

VAR_INPUT
Position : **REAL**;
EndVelocity : **REAL**;
Velocity : **REAL**;
Acceleration : **REAL**;
Deceleration : **REAL**;
Jerk : **REAL**;
Direction : **MC_DIRECTION**;
BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

```
METHOD MoveContinuousRelative : itfContinousAxisCommand
VAR_INPUT
    Distance : REAL;
    EndVelocity : REAL;
    Velocity : REAL;
    Acceleration : REAL;
    Deceleration : REAL;
    Jerk : REAL;
    Direction : MC_DIRECTION;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD
```

```
METHOD TorqueControl : itfAxisCommand
VAR_INPUT
    Torque : REAL;
    TorqueRamp : REAL;
    Velocity : REAL;
    Acceleration : REAL;
    Deceleration : REAL;
    Jerk : REAL;
    Direction : MC_DIRECTION;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD
```

```
METHOD PositionProfile : itfCommand
VAR_INPUT
    TimeScale : REAL;
    PositionScale : REAL;
    Offset : REAL;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD
```

METHOD VelocityProfile : **itfCommand**

VAR_INPUT
    TimeScale : **REAL**;
    VelocityScale : **REAL**;
    Offset : **REAL**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

METHOD AccelerationProfile : **itfCommand**

VAR_INPUT
    TimeScale : **REAL**;
    AccelerationScale : **REAL**;
    Offset : **REAL**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

5.3.3. Multi-Axes Motion

METHOD CamIn : **itfSynchronizedCommand**

VAR_INPUT
    Master : **itfAxis**;
    MasterOffset : **REAL**;
    SlaveOffset : **REAL**;
    MasterScaling : **REAL**;
    SlaveScaling : **REAL**;
    MasterStartDistance : **REAL**;
    MasterSyncPosition : **REAL**;
    StartMode : **MC_START_MODE**;
    MasterValueSource : **MC_SOURCE**;
    CamTable : **itfCamTable**;
    BufferMode : **MC_BUFFER_MODE**;
END_VAR

END_METHOD

```
METHOD GearIn : itfSynchronizedCommand
VAR_INPUT
    Master : itfAxis;
    RatioNumerator: REAL;
    RatioDenominator: REAL;
    MasterValueSource : MC_SOURCE;
    Acceleration: REAL;
    Deceleration: REAL;
    Jerk: REAL;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD

METHOD GearInPos : itfSynchronizedCommand
VAR_INPUT
    Master : itfAxis;
    RatioNumerator : REAL;
    RatioDenominator : REAL;
    MasterValueSource : MC_SOURCE;
    MasterSyncPosition : REAL;
    SlaveSyncPosition : REAL;
    SyncMode : MC_SYNC_MODE;
    MasterStartDistance : REAL;
    Velocity : REAL;
    Acceleration : REAL;
    Deceleration : REAL;
    Jerk : REAL;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD

METHOD PhasingAbsolute : itfSynchronizedCommand
VAR_INPUT
    Master : itfAxis;
    PhaseShift : REAL;
    Velocity : REAL;
    Acceleration : REAL;
    Deceleration : REAL;
    Jerk : REAL;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD
```

```
METHOD PhasingRelative : itfSynchronizedCommand
VAR_INPUT
    Master : itfAxis;
    PhaseShift : REAL;
    Velocity : REAL;
    Acceleration : REAL;
    Deceleration : REAL;
    Jerk : REAL;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD

METHOD CombineAxes : itfSynchronizedCommand
VAR_INPUT
    Master1 : itfAxis;
    Master2 : itfAxis;
    CombineMode : MC_COMBINE_MODE;
    GearRatioNumeratorM1 : INT;
    GearRatioDenominatorM1 : INT;
    GearRatioNumeratorM2 : INT;
    GearRatioDenominatorM2 : INT;
    MasterValueSourceM1 : MC_SOURCE;
    MasterValueSourceM2 : MC_SOURCE;
    BufferMode : MC_BUFFER_MODE;
END_VAR
END_METHOD
```

6 Warehousing example

6.1. 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.

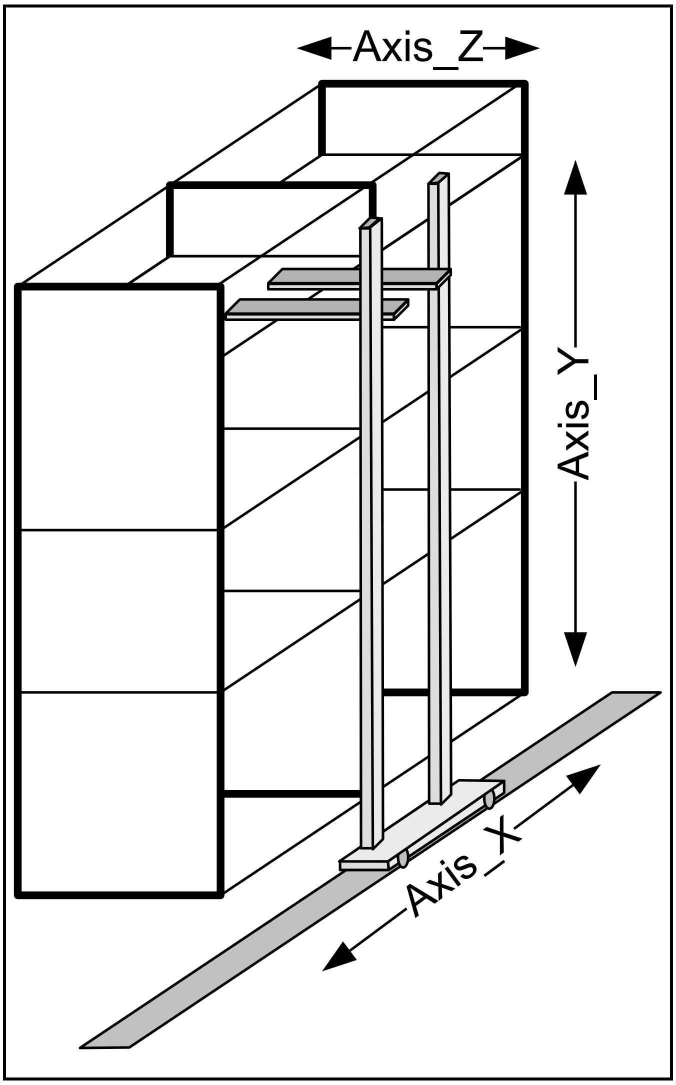


Figure 4: Overview of the warehousing example

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.

6.2. First programming example (using FBs from Part 1)

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

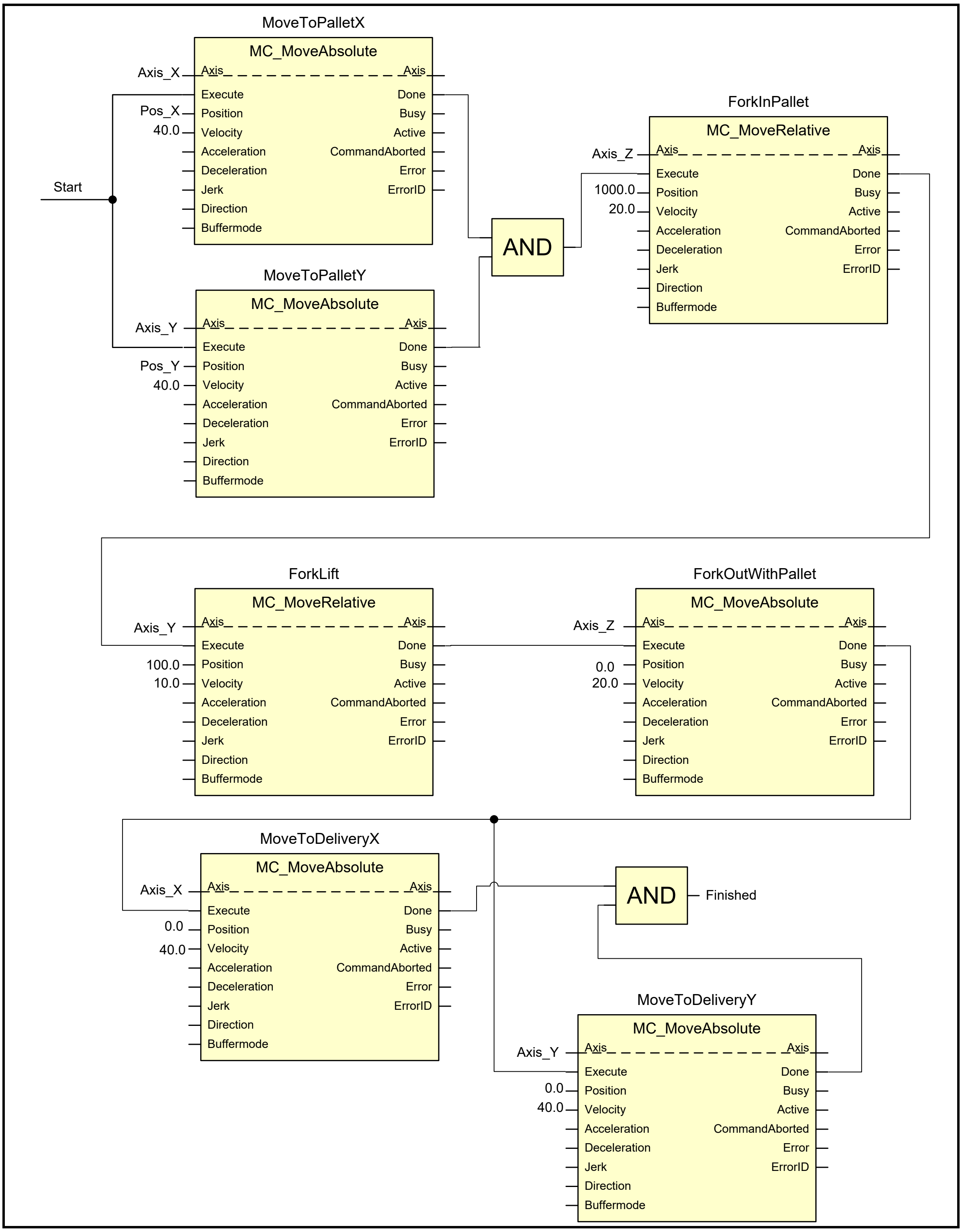


Figure 5: First Program for warehousing example
Note: not all the specified inputs are shown in FBs above.

6.3. Timing diagram

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

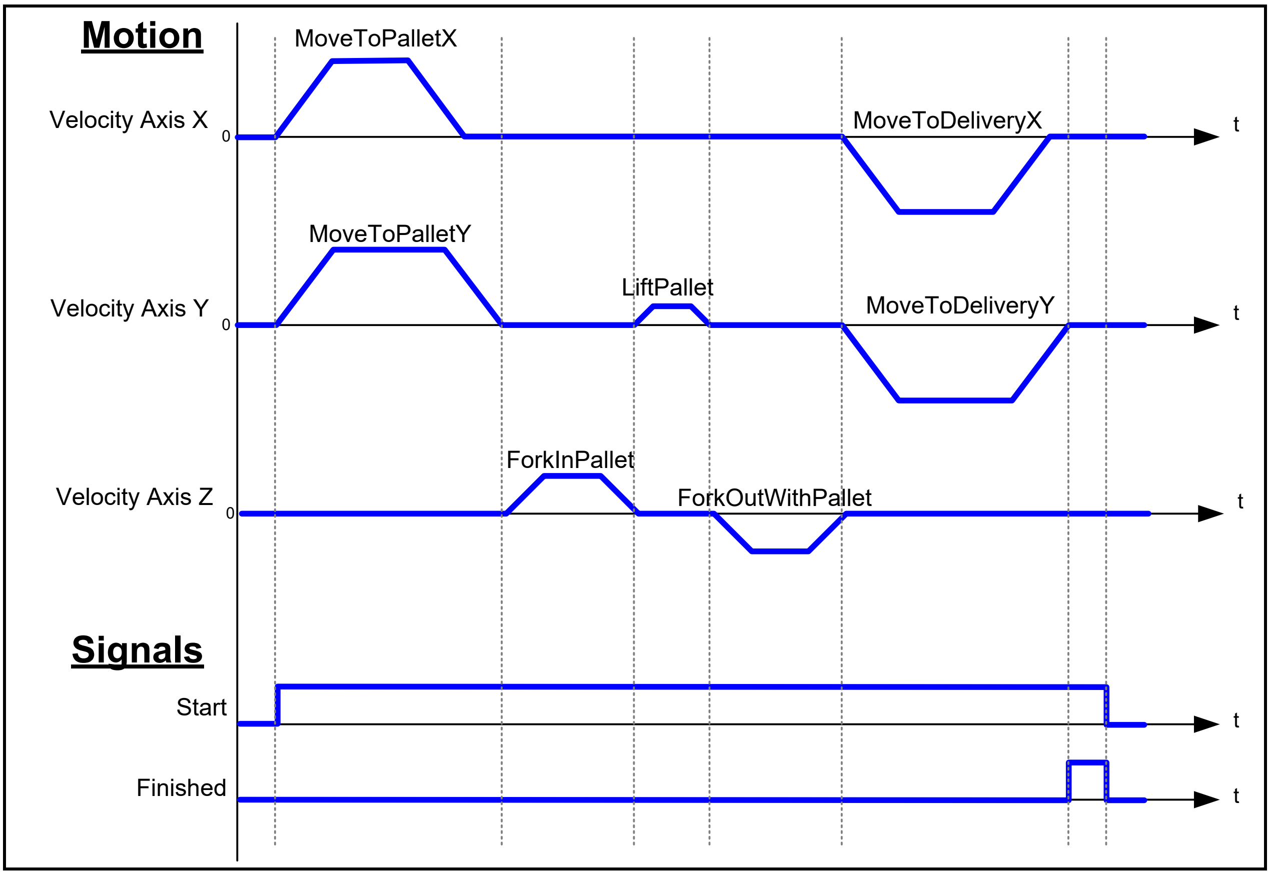


Figure 6: Timing diagram for warehousing example

6.4. Conversion to OOP

This example is now used to show the OO programming with the ST language. The programming is quite straightforward for the PLC programmer. In the background, the standard PLCopen Motion Control library part 1 (v2) is used. Usually, those FBs are called in a cyclic manner, which does not match so fine with the OOP idea. If feedback is not necessary (Done, Error...) it might be OK, if the FB is not called cyclic. With the method GetCommandStatus, there is a way to call the underlying FB (like MC_MoveAbsolute), and with the GetCommandStatus (by internally calling MC_MoveAbsolute and other instances) the status information (Done, Error, etc.) of the movement is returned. Based on this, the next command can be started. The implementation of the methods is, of course, vendor specific.

The program uses a state machine, and the different states reflect the different stages of the whole trajectory. With the use of the variables lastCommandX, lastCommandY and lastCommandZ, the program gets very transparent (see Figure 7: variable declaration part on top and implementation of state machine as CASE instruction at bottom). The resulting timing diagram is depicted in Figure 8: top, including a legend at the bottom.

```
PROGRAM WarehousingExample
VAR
        Axis_X : itfAxis;
        Axis_Y : itfAxis;
        Axis_Z : itfAxis;
        stateOOP : INT;
        lastCommandX : itfCommand;
        lastCommandY : itfCommand;
        lastCommandZ : itfCommand;
        stepOn : BOOL := FALSE;
        targetPosX : REAL := 400;
        targetPosY : REAL := 600;
END_VAR

CASE stateOOP OF
    0: ;
    10:
    // init
        Axis_X.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        Axis_Y.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        Axis_Z.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);

        IF stepOn THEN
            stateOOP := stateOOP + 10;
        END_IF
    20:
    // start movement in XY
        lastCommandX := Axis_X.MoveAbsolute(Position:=targetPosX, Velocity:=40, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
        lastCommandY := Axis_Y.MoveAbsolute(Position:=targetPosY, Velocity:=40, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
        stateOOP := stateOOP + 10;
    30:
    // wait till movement is finished to 'forkIn'
        IF lastCommandX.Done AND lastCommandY.Done THEN
            lastCommandZ := Axis_Z.MoveRelative(Distance:=100, Velocity:=20, Acceleration:=0, Deceleration:=0, Jerk:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            stateOOP := stateOOP + 10;
        END_IF
    40:
    // lift pallet if forked in
        IF lastCommandZ.Done THEN
            lastCommandY := Axis_Y.MoveRelative(Distance:=100, Velocity:=10, Acceleration:=0, Deceleration:=0, Jerk:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            stateOOP := stateOOP + 10;
        END_IF
    50:
    // fork out with pallet if forked in is done
        IF lastCommandY.Done THEN
            lastCommandZ := Axis_Z.MoveAbsolute(Position:=0, Velocity:=20, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            stateOOP := stateOOP + 10;
        END_IF
    60:
    // move to delivery if fork out is done
        IF lastCommandZ.Done THEN
            lastCommandX := Axis_X.MoveAbsolute(Position:=0, Velocity:=40, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            lastCommandY := Axis_Y.MoveAbsolute(Position:=0, Velocity:=40, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            stateOOP := stateOOP + 10;
        END_IF
    70:
    // wait till finished
        IF lastCommandX.Done AND lastCommandY.Done THEN
            stateOOP := stateOOP + 10;
        END_IF
    80:
    // ready
    ;
END_CASE
```

Figure 7: Program example in OOP

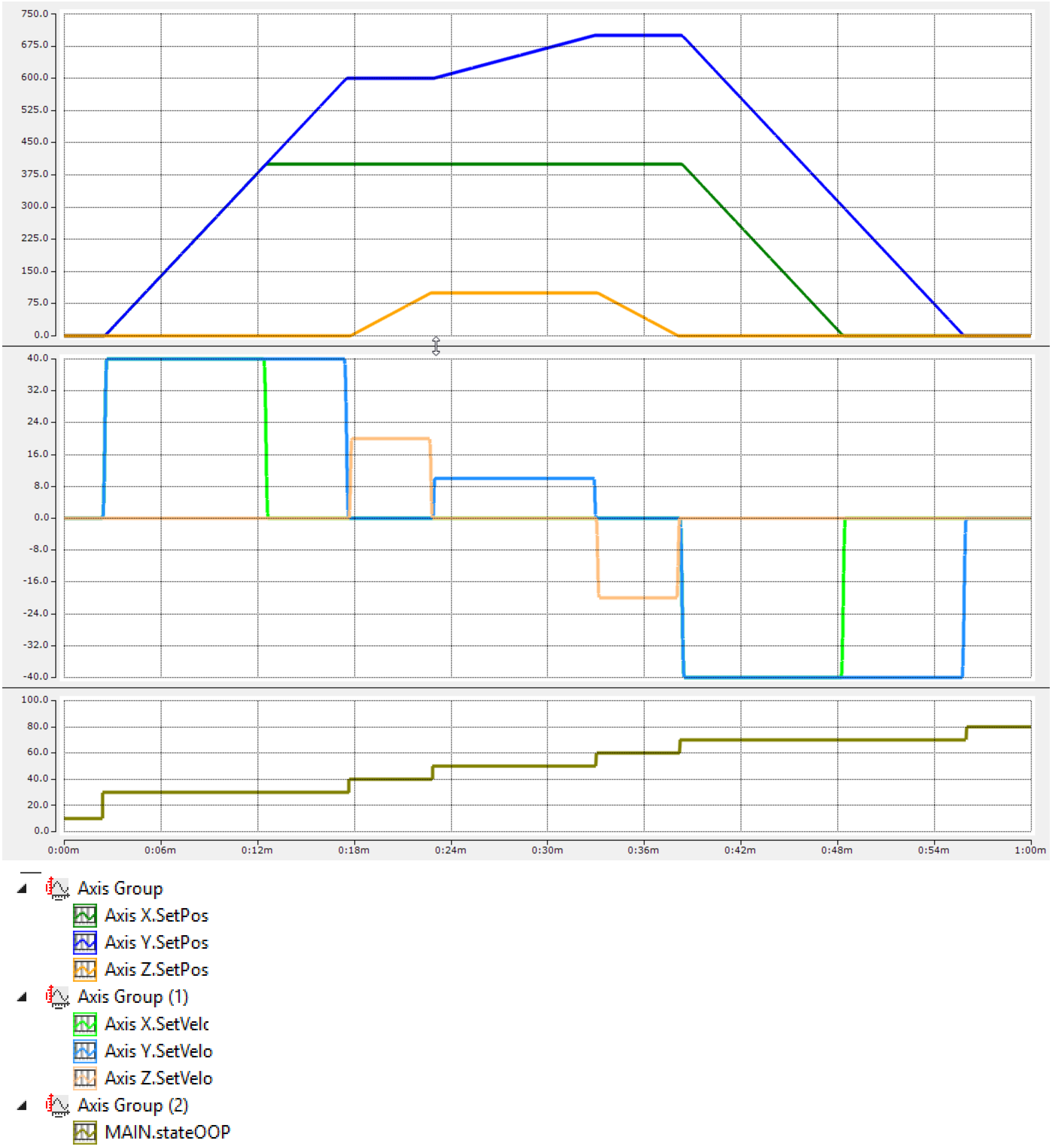


Figure 8: Timing Diagram of warehouse example implemented in OOP with legend

7 Labeling example

7.1. 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 (cf. Figure 9: top). 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.

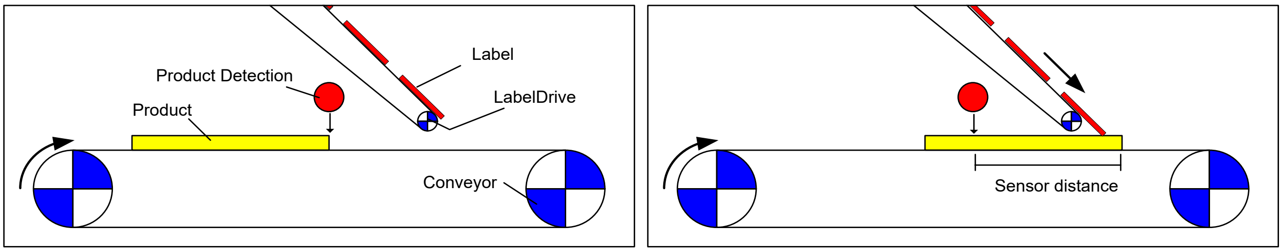


Figure 9: Labeling machine

7.2. Programming example

This example shows a way to solve this task in the programming language FBD in Figure 10:.

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.

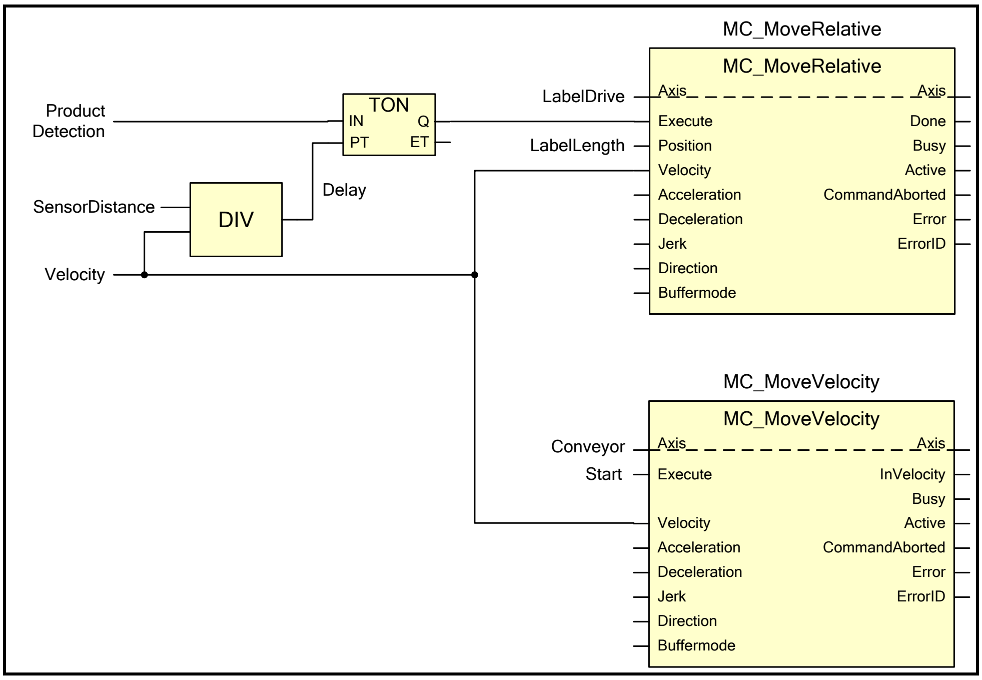


Figure 10: Program example for labeling machine

7.3. Conversion to OOP

Using the programming language ST and the introduced OOP elements for motion control, the labeling example is converted to OOP. Similar to the warehouse example, the standard PLCopen Motion Control Library part 1 (v2) is used in the background and the program uses a state machine to reflect the different states of the process. See Figure 11:Figure 7: variable declaration part and Figure 12: implementation part of the main program with a state machine. The resulting timing diagram is depicted in Figure 13: top, including a legend at the bottom.

```
PROGRAM LabelingExample
VAR
    // Hardware definitions
    LabelDrive : Axis;
    Conveyor : Axis;
    LabelLength : REAL := 100;
    SensorDistance : REAL := 10;
    Velocity : REAL := 5;
    ProductDetection : BOOL := FALSE;
    DelayTimer : TON;

    // States
    ConveyorState : INT := 0;
    LabelDriveState : INT := 0;

    // Control
    Start : BOOL := FALSE;

    // Commands
    ConveyorMove : itfCommand;
    LabelDriveMove : itfCommand;
END_VAR
```

Figure 11: Variable declaration part of the main program

```
// Run the timer. As it is a classic FB it should be called every cycle
DelayTimer(IN:=ProductDetection, PT:= INT_TO_TIME(REAL_TO_INT(SensorDistance * 1000 / Velocity)));

CASE ConveyorState OF
    0: // Disabled
        IF Start THEN
            ConveyorState := ConveyorState + 1;
        END_IF

    1: // Power On
        Conveyor.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        IF Conveyor.Status = AXIS_STATUS.Standstill THEN
            ConveyorState := ConveyorState + 1;
        END_IF

    2: // Wait for LabelDrive
        IF LabelDrive.Status = AXIS_STATUS.Standstill THEN
            ConveyorMove := Conveyor.MoveVelocity(Velocity:=Velocity, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=MC_Direction.mcPositiveDirection, BufferMode:=MC_BUFFER_MODE.mcAborting);
            ConveyorState := ConveyorState + 1;
        END_IF

    3: // Moving
        IF NOT Start THEN
            ConveyorMove := Conveyor.Halt(Deceleration:=0, Jerk:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            ConveyorState := ConveyorState + 1;
        END_IF

    4: // Stopping
        IF ConveyorMove.Done THEN
            ConveyorState := ConveyorState + 1;
        END_IF

    5: // Power off
        Conveyor.Power(Enable:=FALSE, EnablePositive:=FALSE, EnableNegative:=FALSE);
        IF Conveyor.Status = AXIS_STATUS.Disabled THEN
            ConveyorState := 0;
        END_IF
END_CASE

CASE LabelDriveState OF
    0: // Disabled
        IF Start THEN
            LabelDriveState := LabelDriveState + 1;
        END_IF

    1: // Power On
        LabelDrive.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        IF LabelDrive.Status = AXIS_STATUS.Standstill THEN
            LabelDriveState := LabelDriveState + 1;
        END_IF

    2: // Wait for Conveyor
        IF Conveyor.Status = AXIS_STATUS.Standstill THEN
            LabelDriveState := LabelDriveState + 1;
        END_IF

    3: // Wait for product
        IF NOT Start THEN
            LabelDriveState := 5;
        END_IF
        IF DelayTimer.Q THEN
            LabelDriveMove := LabelDrive.MoveRelative(Distance:=LabelLength, Velocity:=Velocity, Acceleration:=0, Deceleration:=0, Jerk:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            LabelDriveState := LabelDriveState + 1;
        END_IF

    4: // Wait for label to be transfered
        IF LabelDriveMove.Done AND NOT DelayTimer.Q THEN
            LabelDriveState := 3;
        END_IF

    5: // Power off
        Conveyor.Power(Enable:=FALSE, EnablePositive:=FALSE, EnableNegative:=FALSE);
        IF Conveyor.Status = AXIS_STATUS.Disabled THEN
            LabelDriveState := 0;
        END_IF

END_CASE
```

Figure 12: Implementation part of the labeling example converted in OOP and implemented with a state machine for each of the 2 axes

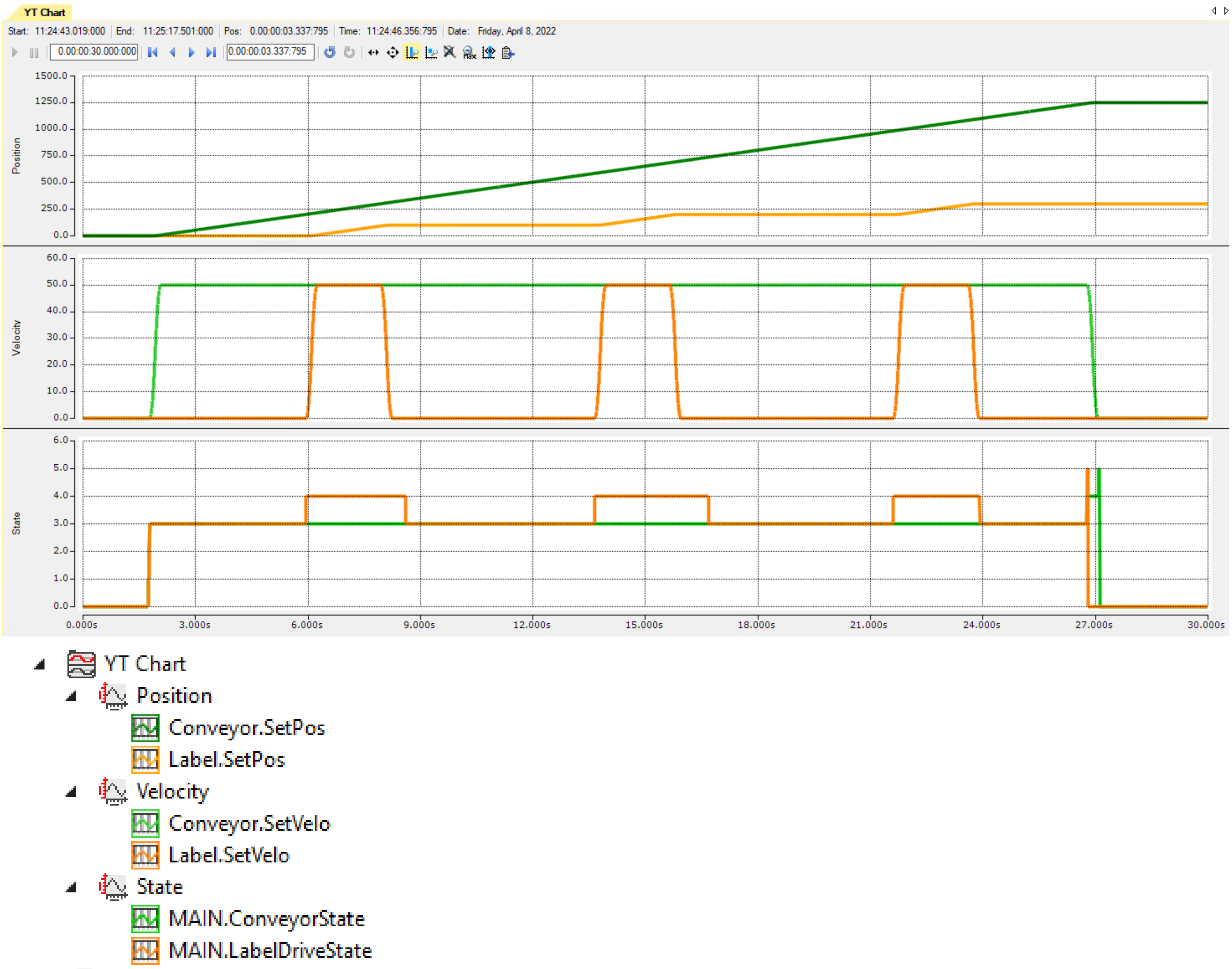


Figure 13: Timing diagram of labeling example with legend

8 Example with Cam and Gear

8.1. Application description

This is a simple demonstration of an application with a master axis moving at a fixed speed, a second axis as CAM slave and a third Gear slave to demonstrate the implementation of axes synchronization with OOP.

8.2. Classical Programming example

There are three drives, namely a MasterDrive, a CamDrive and a GearDrive. The first step is switching on the power. The CamDrive is linked to the Master Drive via the CamTableSelect and CamIn. The GearDrive is connected via the GearIn. Once both slave axes are ready (InSync and InGear with the master axis) as well as the master axis, the master axis starts moving with the Velocity, and both slave axes follow. A classical implementation of the example in FBD is depicted in Figure 14:.

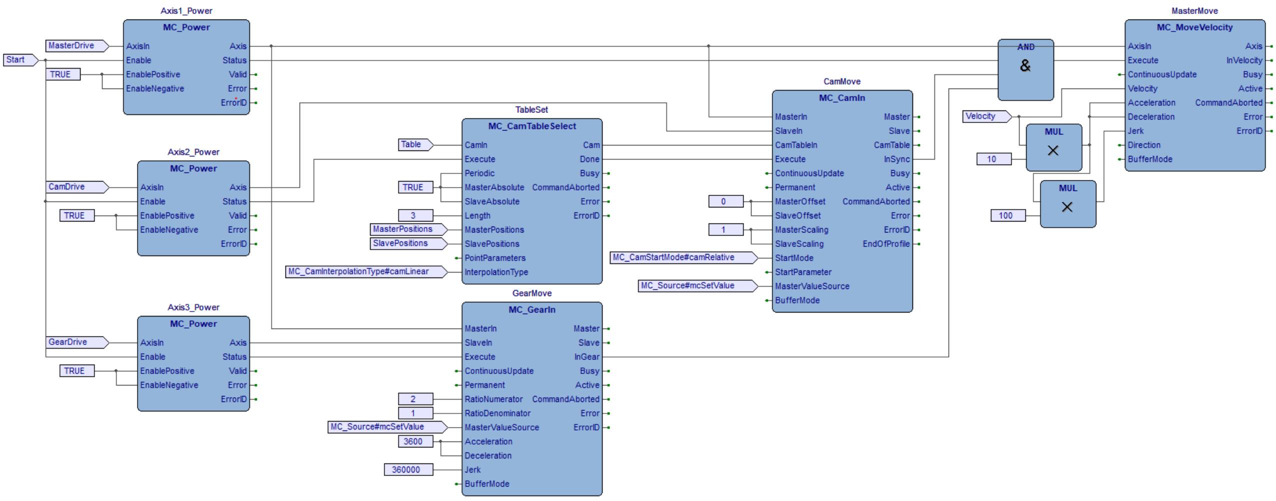


Figure 14: Classical FBD implementation of a synchronization example with three drives

8.3. Conversion to OOP

The synchronization example is converted to OOP using the interfaces, ENUMS and STRUCTs introduced in this document. The declaration part of the main program is depicted in Figure 15: It contains the hardware definitions, states of the drives and commands to enable the synchronization of the two slave axes with the master axis. For the synchronization, the defined interface *itfSynchronizedAxisCommand* is used for controlling the CamDrive (CamIn) and GearDrive (GearIn). The implementation parts of the different drives are depicted in Figure 16: MasterDrive at top, CamDrive in the middle and GearDrive at the bottom.

The example illustrates that it is relatively simple to implement synchronized drives with the commands provided by the newly defined interface.

```
PROGRAM CamGearExample
VAR
    // Hardware definitions
    MasterDrive : Axis;
    CamDrive : Axis;
    GearDrive : Axis;
    Table : CamTable;
    TableData : MC_CAM_REF;
    Velocity : REAL := 5;

    // States
    MasterState : INT := 0;
    CamState : INT := 0;
    GearState : INT := 0;

    // Control
    Start : BOOL := FALSE;

    // Commands
    MasterMove : itfCommand;
    TableSet : itfCommand;
    CamMove : itfSynchronizedAxisCommand;
    GearMove : itfSynchronizedAxisCommand;
END_VAR
```

Figure 15: Variable declaration part of the synchronization example's main program

```
CASE MasterState OF
    0: // Disabled
        IF Start THEN
            MasterState := MasterState + 1;
        END_IF

    1: // Power On
        MasterDrive.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        IF MasterDrive.Status = AXIS_STATUS.Standstill THEN
            MasterState := MasterState + 1;
        END_IF

    2: // Wait for other drives
        IF CamMove.InSync AND GearMove.InSync THEN
            MasterMove := MasterDrive.MoveVelocity(Velocity:=Velocity, Acceleration:=0, Deceleration:=0, Jerk:=0, Direction:=MC_Direction.mcPositiveDirection, BufferMode:=MC_BUFFER_MODE.mcAborting);
            MasterState := MasterState + 1;
        END_IF

    3: // Moving
        IF NOT Start THEN
            MasterMove:= MasterDrive.Halt(Deceleration:=0, Jerk:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            MasterState := MasterState + 1;
        END_IF

    4: // Stopping
        IF NOT MasterMove.Done THEN
            MasterState := MasterState + 1;
        END_IF

    5: // Power off
        MasterDrive.Power(Enable:=FALSE, EnablePositive:=FALSE, EnableNegative:=FALSE);
        IF MasterDrive.Status = AXIS_STATUS.Disabled THEN
            MasterState := 0;
        END_IF
END_CASE

CASE CamState OF
    0: // Disabled
        IF Start THEN
            CamState := CamState + 1;
        END_IF

    1: // Power On
        CamDrive.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        IF CamDrive.Status = AXIS_STATUS.Standstill AND MasterDrive.Status = AXIS_STATUS.Standstill THEN
            TableSet := Table.Select(CamTable:=TableData, Periodic:=TRUE, MasterAbsolute:=TRUE, SlaveAbsolute:=TRUE);
            CamState := CamState + 1;
        END_IF

    2: // Set CAM Table
        IF TableSet.Done THEN
            CamMove := CamDrive.CamIn(Master:=MasterDrive, MasterOffset:=0, SlaveOffset:=0, MasterScaling:=1, SlaveScaling:=1, MasterStartDistance:=0, MasterSyncPosition:=0,
                StartMode:=MC_StartMode.mcRelative, MasterValueSource:=MC_Source.mcSetValue,CamTable:=Table, BufferMode:=MC_BUFFER_MODE.mcAborting);
            CamState := CamState + 1;
        END_IF

    3: // Synchronized
        IF NOT Start THEN
            CamDrive.Release();
            CamState := CamState + 1;
        END_IF

    4: // Power off
        CamDrive.Power(Enable:=FALSE, EnablePositive:=FALSE, EnableNegative:=FALSE);
        IF CamDrive.Status = AXIS_STATUS.Disabled THEN
            CamState := 0;
        END_IF
END_CASE

CASE GearState OF
    0: // Disabled
        IF Start THEN
            GearState := GearState + 1;
        END_IF

    1: // Power On
        GearDrive.Power(Enable:=TRUE, EnablePositive:=TRUE, EnableNegative:=TRUE);
        IF GearDrive.Status = AXIS_STATUS.Standstill AND MasterDrive.Status = AXIS_STATUS.Standstill THEN
            GearMove := GearDrive.GearIn(Master:=MasterDrive, Ratio:=2, MasterValueSource:=MC_Source.mcSetValue,Acceleration:=0, Deceleration:=0, Jerk:=0, BufferMode:=MC_BUFFER_MODE.mcAborting);
            GearState := GearState + 1;
        END_IF

    2: // Synchronized
        IF NOT Start THEN
            GearDrive.Release();
            GearState := GearState + 1;
        END_IF

    3: // Power off
        GearDrive.Power(Enable:=FALSE, EnablePositive:=FALSE, EnableNegative:=FALSE);
        IF GearDrive.Status = AXIS_STATUS.Disabled THEN
            GearState := 0;
        END_IF
END_CASE
```

Figure 16: Implementation of the three drives (each with a state machine in ST)

Appendix 1: Porting "Functions blocks for motion control: Part 4- Coordinated Motion" into OOP

1 Goal

This Appendix presents the OOP version of the "Part 4 – Coordinated motion", Version 1.0, motion control standard. It follows the same architecture and principles as the Part 1 & 2 conversion with the itfGroup interface and command interfaces. Please read the porting document for Part 1 & 2 first.

1.1. Coordinate interface

The current group position is defined as an array. This is very good for two- or three-axes cartesian groups, but it is confusing and possibly insufficient for larger groups. For example, a five-axes system will have distances and rotations in the same array, which feels like mixing apples and oranges. A six-axes robot in PCS coordinate system needs a configuration information to uniquely identify its status.
Using an interface for position, velocity and acceleration allows for extensions and different representations depending on the group.

1.1.1. Coordinate system property

To follow proper coding principles, interfaces should be independent and self-contained, so the coordinate interface has to contain the coordinate system the position is represented in. Using a different coordinate system to a REAL ARRAY would change the location completely.
So the coordinate system property has been removed from the itfGroup motion commands and transferred to the itfGroupPosition and itfPath interfaces.

1.1.2. Coordinate transformation

It would be helpful for the itfGroupPosition to be able to transform itself into a different coordinate system. However, the transformation requires the knowledge of the number of axes and transformation matrixes. Saving this information to the itfPosition is not user-friendly as it would make the creation of an itfGroupPosition much more complex. And the GroupPosition instance would save information that belongs to the group instead of the position.
It is easier to add a TransformPosition method in the itfGroup interface to transform a position from one coordinate system to another.

1.2. Short overview of the Function Blocks of Part 4

Coordinated Function Blocks
MC_AddAxisToGroup
MC_RemoveAxisFromGroup
MC_UngroupAllAxes
MC_GroupReadConfiguration
MC_GroupEnable
MC_GroupDisable
MC_GroupHome
MC_SetKinTransform
MC_SetCartesianTransform
MC_SetCoordinateTransform
MC_ReadKinTransform
MC_ReadCartesianTransform
MC_ReadCoordinateTransform
MC_GroupSetPosition
MC_GroupReadActualPosition
MC_GroupReadActualVelocity

MC_GroupReadActualAcceleration
MC_GroupStop
MC_GroupHalt
MC_GroupInterrupt
MC_GroupContinue
MC_GroupReadStatus
MC_GroupReadError
MC_GroupReset
MC_MoveLinearAbsolute
MC_MoveLinearRelative
MC_MoveCircularAbsolute
MC_MoveCircularRelative
MC_MoveDirectAbsolute
MC_MoveDirectRelative
MC_PathSelect
MC_MovePath
MC_GroupSetOverride
Coordinated
MC_SyncAxisToGroup
MC_SyncGroupToAxis
MC_SetDynCoordTransform
MC_TrackConveyorbelt
MC_TrackRotaryTable

Table 5: Short overview of the Function Blocks

2 Command interface definitions

2.1. itfGroupCommand: Extends itfCommand

For group motion methods.

2.1.1. Added Methods

```
METHOD Update : MC_ERROR
VAR_INPUT
        Position : itfGroupPosition;
        Velocity : REAL;
        Acceleration : REAL;
        Deceleration : REAL;
        Jerk : REAL;
END_VAR
END_METHOD
```

2.2. itfSynchronizedGroupCommand : Extends itfGroupCommand

For the synchronized to axis motion.

2.2.1. Added Properties

Name	Access	Type	Description
InSync	Read	BOOL	

3 Coordinate interface definitions

3.1. *itfGroupPosition interface*

The interface represents the position of a group in a specific coordinate system. It can be extended to define the names of the axes more clearly or add configuration parameters for more complex groups such as robotic arms.

3.1.1. Properties

Name	Access	Type	Description
Base	Read/Write	REAL ARRAY	
CoordinateSystem	Read/Write	MC_COORDINATE_SYSTEM	
Type	Read	MC_GROUP_POSITION_TYPE	To differentiate the base class from derivates.

3.2. *itfGroupVelocity interface*

The interface represents the velocity of the axes of a group in a specific coordinate system. It can be extended to define more clearly the names of the axes for more complex groups such as robotic arms.

3.2.1. Properties

Name	Access	Type	Description
Base	Read/Write	REAL ARRAY	
CoordinateSystem	Read/Write	MC_COORDINATE_SYSTEM	
Type	Read	MC_GROUP_POSITION_TYPE	To differentiate the base class from derivates.

3.3. *itfGroupAcceleration interface*

The interface represents the acceleration of the axes of a group in a specific coordinate system. It can be extended to define more clearly the names of the axes for more complex groups such as robotic arms.

3.3.1. Properties

Name	Access	Type	Description
Base	Read/Write	REAL ARRAY	
CoordinateSystem	Read/Write	MC_COORDINATE_SYSTEM	
Type	Read	MC_GROUP_POSITION_TYPE	To differentiate the base class from derivates.

3.4. *itfPath interface definition*

3.4.1. Methods

```
METHOD Select : itfCommand
VAR_INPUT
        Data : MC_PATH_DATA_REF;
        Description: MC_PATH_REF;
END_VAR
END_METHOD
```

4 itfGroup interface

4.1. ENUMs

No.	MC_GROUP_STATUS
	mcErrorStop
	mcDisabled
	mcStandstill (1)
	mcHoming
	mcStopping
	mcMoving

(1) The name Standby is switched to Standstill in order to use the same name as in the MC_AXIS_STATUS ENUM

4.2. Properties

4.2.1. Actual values

Name	Access	Type
AcsAxes	Read	itfAxis ARRAY
McsAxes	Read	itfAxis ARRAY
PcsAxes	Read	itfAxis ARRAY
AcsPosition	Read	itfGroupPosition
McsPosition	Read	itfGroupPosition
PcsPosition	Read	itfGroupPosition
AcsVelocity	Read	itfGroupVelocity
McsVelocity	Read	itfGroupVelocity
PcsVelocity	Read	itfGroupVelocity
PathVelocity	Read	REAL
AcsAcceleration	Read	itfGroupAcceleration
McsAcceleration	Read	itfGroupAcceleration
PcsAcceleration	Read	itfGroupAcceleration
PathAcceleration	Read	REAL

4.2.2. Status

Name	Access	Type
ErrorId	Read	MC_ERROR
Status	Read	MC_GROUP_STATUS

4.2.3. Transform

Name	Access	Type
KinTransform	Read	MC_KIN_TRANSFORM
CartesianTransform	Read	MC_CARTESIAN_TRANSFORM
CoordinateTransform	Read	MC_COORDINATE_TRANSFORM

4.3. Methods

4.3.1. Transform

```
METHOD SetKinTransformation : MC_ERROR
VAR_INPUT
	KinTransform : MC_KIN_TRANSFORM;
	ExecutionMode : MC_EXECUTION_MODE;
END_VAR
END_METHOD

METHOD SetCartesianTransform : MC_ERROR
VAR_INPUT
	TransX : REAL;
	TransY : REAL;
	TransZ : REAL;
	RotAngle1 : REAL;
	RotAngle2 : REAL;
	RotAngle3 : REAL;
	ExecutionMode : MC_EXECUTION_MODE;
END_VAR
END_METHOD

METHOD SetCoordinateTransform : MC_ERROR
VAR_INPUT
	CoordTransform : MC_COORDINATE_TRANSFORM;
	ExecutionMode : MC_EXECUTION_MODE;
END_VAR
END_METHOD
METHOD TransformPosition : itfGroupPosition
VAR_INPUT
	Position : itfGroupPosition;
	TargetCoordSystem : MC_CoordinateSystem;
END_VAR
END_METHOD

METHOD TransformVelocity : itfGroupVelocity
VAR_INPUT
	Velocity : itfGroupVelocity;
	TargetCoordSystem : MC_CoordinateSystem;
END_VAR
END_METHOD

METHOD TransformAcceleration : itfGroupAcceleration
VAR_INPUT
	Acceleration : itfGroupAcceleration;
	TargetCoordSystem : MC_CoordinateSystem;
END_VAR
END_METHOD
```

4.3.2. Control

METHOD AddAxis : **MC_ERROR**
VAR_INPUT
    IdentInGroup: **MC_IDENT_REF**
    Axis: **itfAxis**
END_VAR
END_METHOD

METHOD RemoveAxis : **MC_ERROR**
VAR_INPUT
    IdentInGroup: **MC_IDENT_REF**
END_VAR
END_METHOD

METHOD UngroupAllAxes : **MC_ERROR**
END_METHOD

METHOD Enable : **MC_ERROR**
END_METHOD

METHOD Disable : **MC_ERROR**
END_METHOD

METHOD SetPosition : **itfCommand**
VAR_INPUT
    Position: **itfGroupPosition**;
    Relative : **BOOL**;
    BufferMode: **MC_BUFFER_MODE;**
END_METHOD

METHOD SetOverride : **MC_ERROR**
VAR_INPUT
    VelFactor : **REAL;**
    AccFactor : **REAL;**
    JerkFactor : **REAL;**
END_VAR_INPUT
END_METHOD

4.3.3. Motion

```
METHOD Home : itfGroupCommand
VAR_INPUT
	Position : itfGroupPosition;
	BufferMode: MC_BUFFER_MODE;
END_VAR
END_METHOD

METHOD Stop : itfGroupCommand
VAR_INPUT
	Deceleration : REAL;
	Jerk : REAL;
	BufferMode: MC_BUFFER_MODE;
END_VAR_INPUT

METHOD Halt : itfGroupCommand
VAR_INPUT
	Deceleration : REAL;
	Jerk : REAL;
	BufferMode: MC_BUFFER_MODE;
END_VAR_INPUT

METHOD Interrupt : itfCommand
VAR_INPUT
	Deceleration : REAL;
	Jerk : REAL;
END_VAR_INPUT

METHOD Continue : itfCommand
VAR_INPUT
END_VAR_INPUT

METHOD Reset : itfCommand
VAR_INPUT
END_VAR_INPUT

METHOD MoveLinearAbsolute : itfGroupCommand
VAR_INPUT
	Position : itfGroupPosition;
	Velocity : REAL;
	Acceleration : REAL;
	Deceleration : REAL;
	Jerk : REAL;
	BufferMode: MC_BUFFER_MODE;
	TransitionMode : MC_TRANSITION_MODE;
	TransitionParameter : REAL ARRAY;
END_VAR_INPUT
```

METHOD MoveLinearRelative : **itfGroupCommand**
VAR_INPUT
Distance : **itfGroupPosition;**
Velocity : **REAL;**
Acceleration : **REAL;**
Deceleration : **REAL;**
Jerk : **REAL;**
BufferMode: **MC_BUFFER_MODE;**
TransitionMode : **MC_TRANSITION_MODE**;
TransitionParameter : **REAL ARRAY**;
END_VAR_INPUT

METHOD MoveCircularAbsolute : **itfGroupCommand**
VAR_INPUT
CircMode : **MC_CIRC_MODE**;
AuxPoint : **itfGroupPosition;**
EndPoint : **itfGroupPosition;**
PathChoice : **MC_CIRC_PATHCHOICE**;
Velocity : **REAL;**
Acceleration : **REAL;**
Deceleration : **REAL;**
Jerk : **REAL;**
BufferMode: **MC_BUFFER_MODE;**
TransitionMode : **MC_TRANSITION_MODE**;
TransitionParameter : **REAL ARRAY**;
END_VAR_INPUT

METHOD MoveCircularRelative : **itfGroupCommand**
VAR_INPUT
CircMode : **MC_CIRC_MODE**;
AuxPoint : **itfGroupPosition;**
EndPoint : **itfGroupPosition;**
PathChoice : **MC_CIRC_PATHCHOICE**;
Velocity : **REAL;**
Acceleration : **REAL;**
Deceleration : **REAL;**
Jerk : **REAL;**
BufferMode: **MC_BUFFER_MODE;**
TransitionMode : **MC_TRANSITION_MODE**;
TransitionParameter : **REAL ARRAY**;
END_VAR_INPUT

METHOD MoveDirectAbsolute : **itfGroupCommand**
VAR_INPUT
Position : **itfGroupPosition;**
BufferMode: **MC_BUFFER_MODE;**
TransitionMode : **MC_TRANSITION_MODE**;
TransitionParameter : **REAL ARRAY**;
END_VAR_INPUT

METHOD MoveDirectRelative : **itfGroupCommand**
VAR_INPUT
Distance : **itfGroupPosition;**
BufferMode: **MC_BUFFER_MODE;**
TransitionMode : **MC_TRANSITION_MODE**;
TransitionParameter : **REAL ARRAY**;
END_VAR_INPUT

METHOD MovePath : **itfCommand**
VAR_INPUT
PathData : **itfPath**;
BufferMode: **MC_BUFFER_MODE;**
TransitionMode : **MC_TRANSITION_MODE**;
TransitionParameter : **REAL ARRAY**;
END_VAR_INPUT

METHOD SyncToAxis : **itfSynchronizedGroupCommand**
VAR_INPUT
Master : **itfAxis**;
PathData : **itfPath**;
Mode : **MC_PATH_MODE**;
TuCNumerator : **INT ARRAY**;
TuCDenominator : **INT ARRAY**;
Acceleration : **REAL;**
Deceleration : **REAL;**
Jerk : **REAL;**
BufferMode: **MC_BUFFER_MODE;**
END_VAR_INPUT

METHOD SetDynCoordTransform : **itfCommand**
VAR_INPUT
Master : **itfGroup**;
Coordtransform : **MC_COORD_REF**;
Mode : **MC_PATH_MODE**;
CoordSystem **: MC_COORDINATE_SYSTEM**;
BufferMode: **MC_BUFFER_MODE;**
END_VAR_INPUT

METHOD TrackConveyorBelt : **itfCommand**
VAR_INPUT
ConveyorBelt : **itfAxis**;
ConveyorBeltOrigin : **itfGroupPosition**;
InitialObjectPosition : **itfGroupPosition**;
BufferMode: **MC_BUFFER_MODE;**
END_VAR_INPUT

METHOD TrackRotaryTable : **itfCommand**
VAR_INPUT
RotaryTable : **itfAxis**;
RotaryTableOrigin : **itfGroupPosition**;
InitialObjectPosition : **itfGroupPosition**;
BufferMode: **MC_BUFFER_MODE;**
END_VAR_INPUT

4.4. itfAxis Extension

4.4.1. Added Methods

METHOD SyncToGroup : **itfSynchronizedAxisCommand**
VAR_INPUT
    Group : **itfGroup**;
    RatioNumerator : **INT**;
    RatioDenominator : **UINT**;
    Acceleration : **REAL;**
    Deceleration : **REAL;**
    Jerk : **REAL;**
    BufferMode: **MC_BUFFER_MODE;**
END_VAR_INPUT

