TwinCAT PLC PDF Auto-Gen

Generated on: PLC_SortingSystem_PLC
Total files: 11

Table of Contents

1 00_Internal PackAL\DUTs

1.1 E_PMLState.TcDUT

Declaration

```
(* states according to PackTags v3.0 *)
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_PMLState :
(
UNDEFINED := 0,
CLEARING := 1,
STOPPED := 2,
STARTING := 3,
IDLE := 4,
SUSPENDED := 5,
EXECUTE := 6,
STOPPING := 7,
ABORTING := 8,
ABORTED := 9,
HOLDING := 10,
HELD := 11,
UNHOLDING := 12,
SUSPENDING := 13,
UNSUSPENDING := 14,
RESETTING := 15,
COMPLETING := 16,
COMPLETE := 17
);
END_TYPE
```

1.2 E_PMLUnitMode.TcDUT

Declaration

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_PMLUnitMode :
(
UNDEFINED := 0,
AUTOMATIC := 1,
MAINTENANCE := 2,
MANUAL := 3,
SEMIAUTOMATIC := 4,
DRYRUN := 5,
USERMODE1 := 6,
USERMODE2 := 7,
IDLE := 8,
ESTOP := 9
);
END_TYPE
```

2 00_Internal PackAL\POUs

2.1 PS_PackML_StateMachine_Auto.TcPOU

Declaration

```
FUNCTION_BLOCK PS_PackML_StateMachine_Auto
VAR_INPUT
Start : BOOL;
Hold : BOOL;
UnHold : BOOL;
Suspend : BOOL;
UnSuspend : BOOL;
Abort : BOOL;
Stop : BOOL;
Complete : BOOL;
Clear : BOOL;
Reset : BOOL;
StateComplete : BOOL;
END_VAR
VAR_OUTPUT
Status : WORD;

ST_Starting : BOOL;
ST_Completing : BOOL;
ST_Resetting : BOOL;
ST_Holding : BOOL;
ST_UnHolding : BOOL;
ST_Suspending : BOOL;
ST_UnSuspending : BOOL;
ST_Clearing : BOOL;
ST_Stopping : BOOL;
ST_Aborting : BOOL;

(* State Complete*)
ST_Execute : BOOL;
ST_Complete : BOOL;
ST_Idle : BOOL;
ST_Held : BOOL;
ST_Suspended : BOOL;
ST_Stopped : BOOL;
ST_Aborted : BOOL;

(* additional *)
Error : BOOL;
ErrorID : DWORD;
ePMLState : E_PMLState := E_PMLState.IDLE;
END_VAR
VAR
ePMLStatePrev : E_PMLState;
bStateChange : BOOL;
StateCompletePrev : BOOL;
END_VAR
```

Implementation

```
(* check for state change *)
IF ePMLStatePrev <> ePMLState THEN
ePMLStatePrev := ePMLState;
bStateChange := TRUE;
ELSE
bStateChange := FALSE;
END_IF


(* change to new state if requested *)
CASE ePMLState OF
E_PMLState.UNDEFINED:
(* undefined state *)
IF bStateChange THEN
```

```
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Stopping := TRUE;
END_IF


E_PMLState.IDLE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := TRUE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Idle := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Idle := FALSE;
ST_Stopping := TRUE;
ELSIF Start THEN
ePMLState := E_PMLState.STARTING;
ST_Idle := FALSE;
ST_Starting := TRUE;
END_IF
```

```
E_PMLState.STARTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := TRUE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Starting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Starting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_Starting := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.EXECUTE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := TRUE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
```

```
ST_Execute := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Execute := FALSE;
ST_Stopping := TRUE;
ELSIF Suspend THEN
ePMLState := E_PMLState.SUSPENDING;
ST_Execute := FALSE;
ST_Suspending := TRUE;
ELSIF Hold THEN
ePMLState := E_PMLState.HOLDING;
ST_Execute := FALSE;
ST_Holding := TRUE;
ELSIF Complete THEN
ePMLState := E_PMLState.COMPLETING;
ST_Execute := FALSE;
ST_Completing := TRUE;
END_IF


E_PMLState.COMPLETING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := TRUE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Completing := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Completing := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.COMPLETE;
StateCompletePrev := StateComplete;
ST_Completing := FALSE;
ST_Complete := TRUE;
END_IF


E_PMLState.COMPLETE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
```

```
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := TRUE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Complete := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Complete := FALSE;
ST_Stopping := TRUE;
ELSIF Reset THEN
ePMLState := E_PMLState.RESETTING;
ST_Complete := FALSE;
ST_Resetting := TRUE;
END_IF


E_PMLState.RESETTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := TRUE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Resetting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Resetting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete THEN
ePMLState := E_PMLState.IDLE;
StateCompletePrev := StateComplete;
ST_Resetting := FALSE;
```

```
ST_Idle := TRUE;
END_IF


E_PMLState.HOLDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := TRUE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Holding := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Holding := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.HELD;
StateCompletePrev := StateComplete;
ST_Holding := FALSE;
ST_Held := TRUE;
END_IF


E_PMLState.HELD:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := TRUE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF
```

```
IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Held := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Held := FALSE;
ST_Stopping := TRUE;
ELSIF UnHold THEN
ePMLState := E_PMLState.UNHOLDING;
ST_Held := FALSE;
ST_UnHolding := TRUE;
END_IF


E_PMLState.UNHOLDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := TRUE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_UnHolding := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_UnHolding := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_UnHolding := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.SUSPENDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := TRUE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
```

```
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Suspending := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Suspending := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.SUSPENDED;
StateCompletePrev := StateComplete;
ST_Suspending := FALSE;
ST_Suspended := TRUE;
END_IF


E_PMLState.SUSPENDED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := TRUE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Suspended := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Suspended := FALSE;
ST_Stopping := TRUE;
ELSIF UnSuspend THEN
ePMLState := E_PMLState.UNSUSPENDING;
ST_Suspended := FALSE;
ST_UnSuspending := TRUE;
END_IF


E_PMLState.UNSUSPENDING:
```

```
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := TRUE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_UnSuspending := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_UnSuspending := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_UnSuspending := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.STOPPING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := TRUE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopping := FALSE;
ST_Aborting := TRUE;
```

```
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
StateCompletePrev := StateComplete;
ST_Stopping := FALSE;
ST_Stopped := TRUE;
END_IF


E_PMLState.STOPPED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := TRUE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
ELSIF Reset THEN
ePMLState := E_PMLState.RESETTING;
ST_Stopped := FALSE;
ST_Resetting := TRUE;
END_IF


E_PMLState.ABORTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := TRUE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF
```

```
IF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.ABORTED;
StateCompletePrev := StateComplete;
ST_Aborting := FALSE;
ST_Aborted := TRUE;
END_IF


E_PMLState.ABORTED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := TRUE;
END_IF

IF Clear THEN
ePMLState := E_PMLState.CLEARING;
ST_Aborted := FALSE;
ST_Clearing := TRUE;
END_IF


E_PMLState.CLEARING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := TRUE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
```

```
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
StateCompletePrev := StateComplete;
ST_Clearing := FALSE;
ST_Stopped := TRUE;
END_IF

END_CASE

(* auto clear with StateComplete = FALSE *)
StateCompletePrev := StateCompletePrev AND StateComplete;
```

2.2 PS_PackML_StateMachine_Maintenance.TcPOU

Declaration

```
FUNCTION_BLOCK PS_PackML_StateMachine_Maintenance
VAR_INPUT
Start : BOOL;
Hold : BOOL;
UnHold : BOOL;
Suspend : BOOL;
UnSuspend : BOOL;
Abort : BOOL;
Stop : BOOL;
Complete : BOOL;
Clear : BOOL;
Reset : BOOL;
StateComplete : BOOL;
END_VAR
VAR_OUTPUT
Status : WORD;

ST_Starting : BOOL;
ST_Completing : BOOL;
ST_Resetting : BOOL;
ST_Holding : BOOL;
ST_UnHolding : BOOL;
ST_Suspending : BOOL;
ST_UnSuspending : BOOL;
ST_Clearing : BOOL;
ST_Stopping : BOOL;
ST_Aborting : BOOL;

(* State Complete*)
ST_Execute : BOOL;
ST_Complete : BOOL;
ST_Idle : BOOL;
ST_Held : BOOL;
ST_Suspended : BOOL;
ST_Stopped : BOOL;
ST_Aborted : BOOL;

(* additional *)
Error : BOOL;
ErrorID : DWORD;
ePMLState : E_PMLState := E_PMLState.IDLE;
END_VAR
VAR
ePMLStatePrev : E_PMLState;
bStateChange : BOOL;
StateCompletePrev : BOOL;
END_VAR
```

Implementation

```
(* check for state change *)
IF ePMLStatePrev <> ePMLState THEN
ePMLStatePrev := ePMLState;
bStateChange := TRUE;
```

```
ELSE
bStateChange := FALSE;
END_IF


(* change to new state if requested *)
CASE ePMLState OF
E_PMLState.UNDEFINED:
(* undefined state *)
IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Stopping := TRUE;
END_IF


E_PMLState.IDLE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := TRUE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Idle := FALSE;
ST_Aborting := TRUE;
```

```
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Idle := FALSE;
ST_Stopping := TRUE;
ELSIF Start THEN
ePMLState := E_PMLState.STARTING;
ST_Idle := FALSE;
ST_Starting := TRUE;
END_IF


E_PMLState.STARTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := TRUE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Starting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Starting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_Starting := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.EXECUTE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := TRUE;
```

```
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Execute := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Execute := FALSE;
ST_Stopping := TRUE;
ELSIF Hold THEN
ePMLState := E_PMLState.HOLDING;
ST_Execute := FALSE;
ST_Holding := TRUE;
END_IF


E_PMLState.RESETTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := TRUE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Resetting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Resetting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.IDLE;
StateCompletePrev := StateComplete;
ST_Resetting := FALSE;
ST_Idle := TRUE;
END_IF


E_PMLState.HOLDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
```

```
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := TRUE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Holding := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Holding := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.HELD;
StateCompletePrev := StateComplete;
ST_Holding := FALSE;
ST_Held := TRUE;
END_IF


E_PMLState.HELD:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := TRUE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Held := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Held := FALSE;
ST_Stopping := TRUE;
ELSIF UnHold THEN
```

```
ePMLState := E_PMLState.UNHOLDING;
ST_Held := FALSE;
ST_UnHolding := TRUE;
END_IF


E_PMLState.UNHOLDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := TRUE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_UnHolding := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_UnHolding := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_UnHolding := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.STOPPING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := TRUE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
```

```
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopping := FALSE;
ST_Aborting := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
StateCompletePrev := StateComplete;
ST_Stopping := FALSE;
ST_Stopped := TRUE;
END_IF


E_PMLState.STOPPED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := TRUE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
ELSIF Reset THEN
ePMLState := E_PMLState.RESETTING;
ST_Stopped := FALSE;
ST_Resetting := TRUE;
END_IF


E_PMLState.ABORTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := TRUE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
```

```
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.ABORTED;
StateCompletePrev := StateComplete;
ST_Aborting := FALSE;
ST_Aborted := TRUE;
END_IF


E_PMLState.ABORTED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := TRUE;
END_IF

IF Clear THEN
ePMLState := E_PMLState.CLEARING;
ST_Aborted := FALSE;
ST_Clearing := TRUE;
END_IF


E_PMLState.CLEARING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := TRUE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
```

```
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
ST_Clearing := FALSE;
ST_Stopped := TRUE;
END_IF

END_CASE

(* auto clear with StateComplete = FALSE *)
StateCompletePrev := StateCompletePrev AND StateComplete;
```

2.3 PS_PackML_StateMachine_Manual.TcPOU

Declaration

```
FUNCTION_BLOCK PS_PackML_StateMachine_Manual
VAR_INPUT
Start : BOOL;
Hold : BOOL;
UnHold : BOOL;
Suspend : BOOL;
UnSuspend : BOOL;
Abort : BOOL;
Stop : BOOL;
Complete : BOOL;
Clear : BOOL;
Reset : BOOL;
StateComplete : BOOL;
END_VAR
VAR_OUTPUT
Status : WORD;

ST_Starting : BOOL;
ST_Completing : BOOL;
ST_Resetting : BOOL;
ST_Holding : BOOL;
ST_UnHolding : BOOL;
ST_Suspending : BOOL;
ST_UnSuspending : BOOL;
ST_Clearing : BOOL;
ST_Stopping : BOOL;
ST_Aborting : BOOL;

(* State Complete*)
ST_Execute : BOOL;
ST_Complete : BOOL;
ST_Idle : BOOL;
ST_Held : BOOL;
ST_Suspended : BOOL;
ST_Stopped : BOOL;
ST_Aborted : BOOL;

(* additional *)
Error : BOOL;
ErrorID : DWORD;
ePMLState : E_PMLState := E_PMLState.IDLE;
END_VAR
VAR
ePMLStatePrev : E_PMLState;
bStateChange : BOOL;
StateCompletePrev : BOOL;
END_VAR
```

Implementation

```
(* check for state change *)
IF ePMLStatePrev <> ePMLState THEN
ePMLStatePrev := ePMLState;
bStateChange := TRUE;
ELSE
bStateChange := FALSE;
END_IF


(* change to new state if requested *)
CASE ePMLState OF
E_PMLState.UNDEFINED:
(* undefined state *)
IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Stopping := TRUE;
END_IF


E_PMLState.IDLE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := TRUE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
```

```
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Idle := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Idle := FALSE;
ST_Stopping := TRUE;
ELSIF Start THEN
ePMLState := E_PMLState.STARTING;
ST_Idle := FALSE;
ST_Starting := TRUE;
END_IF


E_PMLState.STARTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := TRUE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Starting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Starting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_Starting := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.EXECUTE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
```

```
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := TRUE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Execute := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Execute := FALSE;
ST_Stopping := TRUE;
END_IF


E_PMLState.RESETTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := TRUE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Resetting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Resetting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.IDLE;
StateCompletePrev := StateComplete;
ST_Resetting := FALSE;
ST_Idle := TRUE;
END_IF


E_PMLState.STOPPING:
(* transient state *)

IF bStateChange THEN
```

```
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := TRUE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopping := FALSE;
ST_Aborting := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
StateCompletePrev := StateComplete;
ST_Stopping := FALSE;
ST_Stopped := TRUE;
END_IF


E_PMLState.STOPPED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := TRUE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
ELSIF Reset THEN
ePMLState := E_PMLState.RESETTING;
ST_Stopped := FALSE;
ST_Resetting := TRUE;
END_IF
```

```
E_PMLState.ABORTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := TRUE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.ABORTED;
StateCompletePrev := StateComplete;
ST_Aborting := FALSE;
ST_Aborted := TRUE;
END_IF


E_PMLState.ABORTED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := TRUE;
END_IF

IF Clear THEN
ePMLState := E_PMLState.CLEARING;
ST_Aborted := FALSE;
ST_Clearing := TRUE;
END_IF


E_PMLState.CLEARING:
(* transient state *)

IF bStateChange THEN
```

```
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := TRUE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
ST_Clearing := FALSE;
ST_Stopped := TRUE;
END_IF

END_CASE

(* auto clear with StateComplete = FALSE *)
StateCompletePrev := StateCompletePrev AND StateComplete;
```

2.4 PS_PackML_StateMachine_SemiAuto.TcPOU

Declaration

```
FUNCTION_BLOCK PS_PackML_StateMachine_SemiAuto
VAR_INPUT
Start : BOOL;
Hold : BOOL;
UnHold : BOOL;
Suspend : BOOL;
UnSuspend : BOOL;
Abort : BOOL;
Stop : BOOL;
Complete : BOOL;
Clear : BOOL;
Reset : BOOL;
StateComplete : BOOL;
END_VAR
VAR_OUTPUT
Status : WORD;

ST_Starting : BOOL;
ST_Completing : BOOL;
ST_Resetting : BOOL;
ST_Holding : BOOL;
ST_UnHolding : BOOL;
ST_Suspending : BOOL;
ST_UnSuspending : BOOL;
ST_Clearing : BOOL;
ST_Stopping : BOOL;
ST_Aborting : BOOL;

(* State Complete*)
ST_Execute : BOOL;
```

```
ST_Complete : BOOL;
ST_Idle : BOOL;
ST_Held : BOOL;
ST_Suspended : BOOL;
ST_Stopped : BOOL;
ST_Aborted : BOOL;

(* additional *)
Error : BOOL;
ErrorID : DWORD;
ePMLState : E_PMLState := E_PMLState.IDLE;
END_VAR
VAR
ePMLStatePrev : E_PMLState;
bStateChange : BOOL;
StateCompletePrev : BOOL;
END_VAR
```

Implementation

```
IF ePMLStatePrev <> ePMLState THEN
ePMLStatePrev := ePMLState;
bStateChange := TRUE;
ELSE
bStateChange := FALSE;
END_IF


(* change to new state if requested *)
CASE ePMLState OF
E_PMLState.UNDEFINED:
(* undefined state *)
IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Stopping := TRUE;
END_IF


E_PMLState.IDLE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
```

```
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := TRUE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Idle := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Idle := FALSE;
ST_Stopping := TRUE;
ELSIF Start THEN
ePMLState := E_PMLState.STARTING;
ST_Idle := FALSE;
ST_Starting := TRUE;
END_IF


E_PMLState.STARTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := TRUE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Starting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Starting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
```

```
StateCompletePrev := StateComplete;
ST_Starting := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.EXECUTE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := TRUE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Execute := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Execute := FALSE;
ST_Stopping := TRUE;
ELSIF Suspend THEN
ePMLState := E_PMLState.SUSPENDING;
ST_Execute := FALSE;
ST_Suspending := TRUE;
ELSIF Hold THEN
ePMLState := E_PMLState.HOLDING;
ST_Execute := FALSE;
ST_Holding := TRUE;
ELSIF Complete THEN
ePMLState := E_PMLState.COMPLETING;
ST_Execute := FALSE;
ST_Completing := TRUE;
END_IF


E_PMLState.COMPLETING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := TRUE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;
```

```
(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Completing := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Completing := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.COMPLETE;
StateCompletePrev := StateComplete;
ST_Completing := FALSE;
ST_Complete := TRUE;
END_IF


E_PMLState.COMPLETE:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := TRUE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Complete := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Complete := FALSE;
ST_Stopping := TRUE;
ELSIF Reset THEN
ePMLState := E_PMLState.RESETTING;
ST_Complete := FALSE;
ST_Resetting := TRUE;
END_IF


E_PMLState.RESETTING:
(* transient state *)

IF bStateChange THEN
```

```
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := TRUE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Resetting := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Resetting := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.IDLE;
StateCompletePrev := StateComplete;
ST_Resetting := FALSE;
ST_Idle := TRUE;
END_IF


E_PMLState.HOLDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := TRUE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Holding := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Holding := FALSE;
```

```
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.HELD;
StateCompletePrev := StateComplete;
ST_Holding := FALSE;
ST_Held := TRUE;
END_IF


E_PMLState.HELD:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := TRUE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Held := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Held := FALSE;
ST_Stopping := TRUE;
ELSIF UnHold THEN
ePMLState := E_PMLState.UNHOLDING;
ST_Held := FALSE;
ST_UnHolding := TRUE;
END_IF


E_PMLState.UNHOLDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := TRUE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
```

```
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_UnHolding := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_UnHolding := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_UnHolding := FALSE;
ST_Execute := TRUE;
END_IF


E_PMLState.SUSPENDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := TRUE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Suspending := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Suspending := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.SUSPENDED;
StateCompletePrev := StateComplete;
ST_Suspending := FALSE;
ST_Suspended := TRUE;
END_IF


E_PMLState.SUSPENDED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
```

```
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := TRUE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Suspended := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_Suspended := FALSE;
ST_Stopping := TRUE;
ELSIF UnSuspend THEN
ePMLState := E_PMLState.UNSUSPENDING;
ST_Suspended := FALSE;
ST_UnSuspending := TRUE;
END_IF


E_PMLState.UNSUSPENDING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := TRUE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_UnSuspending := FALSE;
ST_Aborting := TRUE;
ELSIF Stop THEN
ePMLState := E_PMLState.STOPPING;
ST_UnSuspending := FALSE;
ST_Stopping := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.EXECUTE;
StateCompletePrev := StateComplete;
ST_UnSuspending := FALSE;
```

```
ST_Execute := TRUE;
END_IF


E_PMLState.STOPPING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := TRUE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopping := FALSE;
ST_Aborting := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
StateCompletePrev := StateComplete;
ST_Stopping := FALSE;
ST_Stopped := TRUE;
END_IF


E_PMLState.STOPPED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := TRUE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
```

```
ST_Aborting := TRUE;
ELSIF Reset THEN
ePMLState := E_PMLState.RESETTING;
ST_Stopped := FALSE;
ST_Resetting := TRUE;
END_IF


E_PMLState.ABORTING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := TRUE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.ABORTED;
StateCompletePrev := StateComplete;
ST_Aborting := FALSE;
ST_Aborted := TRUE;
END_IF


E_PMLState.ABORTED:
(* final state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := FALSE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := TRUE;
END_IF

IF Clear THEN
ePMLState := E_PMLState.CLEARING;
ST_Aborted := FALSE;
```

```
ST_Clearing := TRUE;
END_IF


E_PMLState.CLEARING:
(* transient state *)

IF bStateChange THEN
(* transient state *)
ST_Starting := FALSE;
ST_Completing := FALSE;
ST_Resetting := FALSE;
ST_Holding := FALSE;
ST_UnHolding := FALSE;
ST_Suspending := FALSE;
ST_UnSuspending := FALSE;
ST_Clearing := TRUE;
ST_Stopping := FALSE;
ST_Aborting := FALSE;

(* final state *)
ST_Execute := FALSE;
ST_Complete := FALSE;
ST_Idle := FALSE;
ST_Held := FALSE;
ST_Suspended := FALSE;
ST_Stopped := FALSE;
ST_Aborted := FALSE;
END_IF

IF Abort THEN
ePMLState := E_PMLState.ABORTING;
ST_Stopped := FALSE;
ST_Aborting := TRUE;
ELSIF StateComplete AND NOT StateCompletePrev THEN
ePMLState := E_PMLState.STOPPED;
StateCompletePrev := StateComplete;
ST_Clearing := FALSE;
ST_Stopped := TRUE;
END_IF

END_CASE

(* auto clear with StateComplete = FALSE *)
StateCompletePrev := StateCompletePrev AND StateComplete;
```

2.5 PS_UnitModeManager.TcPOU

Declaration

```
FUNCTION_BLOCK PS_UnitModeManager
VAR_INPUT
Execute : BOOL;
eModeCommand : E_PMLUnitMode;
ePMLState : E_PMLState;
END_VAR
VAR_OUTPUT
eModeStatus : E_PMLUnitMode;
Done : BOOL;
Error : BOOL;
ErrorID : UDINT;
END_VAR
VAR
fbTrigger : R_TRIG;
END_VAR
```

Implementation

```
fbTrigger(CLK := Execute);
```

```
IF fbTrigger.Q THEN (* rising edge Execute *)
Done := FALSE;
Error := FALSE;
ErrorID := 0;

CASE eModeStatus OF
E_PMLUnitMode.UNDEFINED:
eModeStatus := eModeCommand;
Done := TRUE;

E_PMLUnitMode.AUTOMATIC:
IF (ePMLState = E_PMLState.STOPPED) OR (ePMLState = E_PMLState.ABORTED) OR (ePMLState = E_PMLState.IDLE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSIF ((ePMLState = E_PMLState.SUSPENDED) OR (ePMLState = E_PMLState.HELD) OR (ePMLState = E_PMLState.COMPLETE))
AND (eModeCommand = E_PMLUnitMode.SEMIAUTOMATIC) THEN

eModeStatus := eModeCommand;
Done := TRUE;
ELSIF (ePMLState = E_PMLState.HELD) AND (eModeCommand = E_PMLUnitMode.MAINTENANCE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSE
Error := TRUE;
ErrorID := 1;
END_IF

E_PMLUnitMode.MAINTENANCE:
IF (ePMLState = E_PMLState.STOPPED) OR (ePMLState = E_PMLState.ABORTED) OR (ePMLState = E_PMLState.IDLE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSIF (ePMLState = E_PMLState.HELD) AND ((eModeCommand = E_PMLUnitMode.AUTOMATIC) OR (eModeCommand =
E_PMLUnitMode.SEMIAUTOMATIC)) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSE
Error := TRUE;
ErrorID := 1;
END_IF

E_PMLUnitMode.MANUAL:
IF (ePMLState = E_PMLState.STOPPED) OR (ePMLState = E_PMLState.ABORTED) OR (ePMLState = E_PMLState.IDLE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSE
Error := TRUE;
ErrorID := 1;
END_IF

E_PMLUnitMode.SEMIAUTOMATIC:
IF (ePMLState = E_PMLState.STOPPED) OR (ePMLState = E_PMLState.ABORTED) OR (ePMLState = E_PMLState.IDLE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSIF ((ePMLState = E_PMLState.SUSPENDED) OR (ePMLState = E_PMLState.HELD) OR (ePMLState = E_PMLState.COMPLETE))
AND (eModeCommand = E_PMLUnitMode.AUTOMATIC) THEN

eModeStatus := eModeCommand;
Done := TRUE;
ELSIF (ePMLState = E_PMLState.HELD) AND (eModeCommand = E_PMLUnitMode.MAINTENANCE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSE
Error := TRUE;
ErrorID := 1;
END_IF

E_PMLUnitMode.DRYRUN:
eModeStatus := eModeCommand;
Done := TRUE;

E_PMLUnitMode.USERMODE1:
```

```
eModeStatus := eModeCommand;
Done := TRUE;

E_PMLUnitMode.USERMODE2:
eModeStatus := eModeCommand;
Done := TRUE;

E_PMLUnitMode.IDLE:
IF (ePMLState = E_PMLState.STOPPED) OR (ePMLState = E_PMLState.ABORTED) OR (ePMLState = E_PMLState.IDLE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSE
Error := TRUE;
ErrorID := 1;
END_IF

E_PMLUnitMode.ESTOP:
IF (ePMLState = E_PMLState.STOPPED) OR (ePMLState = E_PMLState.ABORTED) OR (ePMLState = E_PMLState.IDLE) THEN
eModeStatus := eModeCommand;
Done := TRUE;
ELSE
Error := TRUE;
ErrorID := 1;
END_IF
END_CASE
END_IF

(* reset with bExecute = FALSE *)
IF NOT Execute THEN
Done := FALSE;
Error := FALSE;
ErrorID := 0;
END_IF
```

3 01_Submodules\Axis

3.1 FB_Axis.TcPOU

Declaration

```
FUNCTION_BLOCK FB_Axis
VAR_OUTPUT
// =============== State variables ========================
bError : BOOL; // Error signal
nErrorID : UDINT; // Error ID
bMoves : BOOL; // Status of moving
END_VAR
VAR
// =============== Done signals of Halt and Reset =========
bResetDone : BOOL; // Status of function block MC_Reset
bHaltDone : BOOL; // Status of function block MC_Halt

// =============== State variables ========================
bStatusEnable : BOOL; // Enable feedback

// =============== Velocity, override =====================
fVelocity : LREAL; // Target velocity
fOverride : LREAL := 100.0; // Override 100%

// =============== Trigger function blocks ================
fbTriggerHalt : R_TRIG; // Trigger to recognize rising edge of halt input
fbTriggerReset : R_TRIG; // Trigger to recognize rising edge of reset input

// =============== Axis reference =========================
AxisRef : AXIS_REF; // Axis reference (process data plc/nc)

// =============== Motion function blocks =================
fbMcPower : MC_Power; // FB for enable and override
fbMcReset : MC_Reset; // FB to reset axis
fbMcStop : MC_Stop; // FB to stop axis
fbMcHalt : MC_Halt; // FB to halt axis
fbMcMove : MC_MoveVelocity; // FB to move axis (velocity movement)
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method Enable Declaration

```
METHOD Enable
VAR_INPUT
bEnable : BOOL;
END_VAR
```

Method Enable Implementation

```
// =========================================================

fbMCPower( Enable := bEnable,
Enable_Positive := bEnable,
Enable_Negative := bEnable,
Override := fOverride,
Axis := AxisRef,
Status => bStatusEnable);

IF fbMcPower.Error THEN
bError := TRUE;
nErrorID := fbMcPower.ErrorID;
END_IF
```

```
AxisRef.ReadStatus();

bMoves := AxisRef.Status.Moving;

// =========================================================
```

Method Halt Declaration

```
METHOD Halt
VAR_INPUT
bDriveHalt : BOOL; // Request stop
END_VAR
```

Method Halt Implementation

```
// =========================================================

IF bDriveHalt THEN
// Move Execute FALSE
fbMCMove( Execute := FALSE,
Axis := AxisRef);
ELSE
// Reset done signal
bHaltDone := FALSE;
END_IF

// Trigger
fbTriggerHalt(CLK := bDriveHalt);

IF fbTriggerHalt.Q THEN
fbMcHalt( Execute := TRUE,
Axis := AxisRef);
ELSE
fbMcHalt(Axis := AxisRef);
END_IF

// Done / Error / Command aborted
IF fbMcHalt.Done OR fbMcHalt.Error OR fbMcHalt.CommandAborted THEN
bError := fbMcHalt.Error;
nErrorID := fbMcHalt.ErrorID;
bHaltDone := fbMcHalt.Done;

fbMCMove( Execute := FALSE,
Axis := AxisRef);

fbMcHalt( Execute := FALSE,
Axis := AxisRef);
END_IF

// =========================================================
```

Method MoveBw Declaration

```
METHOD MoveBw
VAR_INPUT
END_VAR
```

Method MoveBw Implementation

```
// =========================================================

// Not error
IF NOT fbMcMove.Error THEN
fbMCMove( Execute := TRUE,
Velocity := fVelocity,
Direction := MC_Negative_Direction,
Axis := AxisRef );
END_IF
```

```
// Error / Command aborted
IF fbMCMove.Error OR fbMCMove.CommandAborted THEN
IF fbMcMove.Error THEN
bError := TRUE;
nErrorID := fbMCMove.ErrorID;
END_IF

fbMCMove( Execute := FALSE,
Axis := AxisRef);
END_IF

// =========================================================
```

Method MoveFw Declaration

```
METHOD MoveFw
VAR_INPUT
END_VAR
```

Method MoveFw Implementation

```
// =========================================================

// Not error
IF NOT fbMcMove.Error THEN
fbMCMove( Execute := TRUE,
Velocity := fVelocity,
Direction := MC_Positive_Direction,
Axis := AxisRef );
END_IF

// Error / Command aborted
IF fbMCMove.Error OR fbMCMove.CommandAborted THEN
IF fbMcMove.Error THEN
bError := TRUE;
nErrorID := fbMCMove.ErrorID;
END_IF

fbMCMove( Execute := FALSE,
Axis := AxisRef);
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
bDriveReset : BOOL; // Request reset
END_VAR
```

Method Reset Implementation

```
// =========================================================

IF NOT bDriveReset THEN
bResetDone := FALSE;
END_IF

// Trigger
fbTriggerReset(CLK := bDriveReset);

IF fbTriggerReset.Q THEN
bHaltDone := FALSE;

fbMcReset(Execute := TRUE, Axis := AxisRef);
ELSE
```

```
fbMcReset(Axis := AxisRef);
END_IF

// Done / Error
IF fbMcReset.Done OR fbMcReset.Error THEN
bError := fbMcReset.Error;
nErrorID := fbMcReset.ErrorID;
bResetDone := fbMcReset.Done;

fbMcReset( Execute := FALSE,
Axis := AxisRef);
END_IF

// =========================================================
```

Method Stop Declaration

```
METHOD Stop
VAR_INPUT
bDriveStop : BOOL; // Request stop
END_VAR
```

Method Stop Implementation

```
// =========================================================

IF bDriveStop THEN
// Move Execute FALSE
fbMCMove( Execute := FALSE,
Axis := AxisRef);
ELSE
// Reset done signal
bHaltDone := FALSE;
END_IF

// Trigger
fbTriggerHalt(CLK := bDriveStop);

IF fbTriggerHalt.Q THEN
fbMcStop( Execute := TRUE,
Axis := AxisRef);
ELSE
fbMcStop(Axis := AxisRef);
END_IF

// Done / Error / Command aborted
IF fbMcStop.Done OR fbMcStop.Error OR fbMcStop.CommandAborted THEN
bError := fbMcStop.Error;
nErrorID := fbMcStop.ErrorID;
bHaltDone := fbMcStop.Done;

fbMCMove( Execute := FALSE,
Axis := AxisRef);

fbMcStop( Execute := FALSE,
Axis := AxisRef);
END_IF

// =========================================================
```

Property bState_Enable Declaration

```
PROPERTY bState_Enable : BOOL
```

Property bState_Enable Get Declaration

```
VAR
END_VAR
```

Property bState_Enable Get Implementation

```
bState_Enable := bStatusEnable;
```

Property bState_HaltDone Declaration

```
PROPERTY bState_HaltDone : BOOL
```

Property bState_HaltDone Get Declaration

```
VAR
END_VAR
```

Property bState_HaltDone Get Implementation

```
bState_HaltDone := bHaltDone;
```

Property bState_ResetDone Declaration

```
PROPERTY bState_ResetDone : BOOL
```

Property bState_ResetDone Get Declaration

```
VAR
END_VAR
```

Property bState_ResetDone Get Implementation

```
bState_ResetDone := bResetDone;
```

Property fTargetVelocity Declaration

```
PROPERTY fTargetVelocity : LREAL
```

Property fTargetVelocity Get Declaration

```
VAR
END_VAR
```

Property fTargetVelocity Get Implementation

```
fTargetVelocity := fVelocity;
```

Property fTargetVelocity Set Declaration

```
VAR
END_VAR
```

Property fTargetVelocity Set Implementation

```
fVelocity := fTargetVelocity;
```

4 01_Submodules\Cylinder\ITFs

4.1 I_Cylinder.TcIO

Declaration

```
INTERFACE I_Cylinder
```

Method MoveToBase Declaration

```
METHOD MoveToBase
VAR_INPUT
END_VAR
```

Method MoveToWork Declaration

```
METHOD MoveToWork
VAR_INPUT
END_VAR
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
END_VAR
```

Property bState_AtBasePos Declaration

```
PROPERTY bState_AtBasePos : BOOL
```

Property bState_AtWorkPos Declaration

```
PROPERTY bState_AtWorkPos : BOOL
```

Property bState_MoveToWork Declaration

```
PROPERTY bState_MoveToWork : BOOL
```

5 01_Submodules\Cylinder\POUs

5.1 FB_Cylinder.TcPOU

Declaration

```
(* FB_Cylinder - number of control signals:
one direction is controllable
- type of feedback signal:
feedback in base and work position *)

FUNCTION_BLOCK FB_Cylinder IMPLEMENTS I_Cylinder
VAR_INPUT
bAtBasePos AT %I* : BOOL; // Hardware input signal: cylinder is at base position
bAtWorkPos AT %I* : BOOL; // Hardware input signal: cylinder is at work position
END_VAR
VAR_OUTPUT
bMoveToWork AT %Q* : BOOL; // Hardware output signal to move cylinder to work position
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method MoveToBase Declaration

```
METHOD MoveToBase
VAR_INPUT
END_VAR
```

Method MoveToBase Implementation

```
// =========================================================

StateMachine( bBasePosReq := TRUE,
bWorkPosReq := FALSE );

// =========================================================
```

Method MoveToWork Declaration

```
METHOD MoveToWork
VAR_INPUT
END_VAR
```

Method MoveToWork Implementation

```
// =========================================================

StateMachine( bBasePosReq := FALSE,
bWorkPosReq := TRUE );

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
END_VAR
```

Method Reset Implementation

```
// =========================================================
bMoveToWork := FALSE;

// =========================================================
```

Method StateMachine Declaration

```
METHOD PROTECTED StateMachine
VAR_INPUT
bBasePosReq : BOOL;
bWorkPosReq : BOOL;
END_VAR
```

Method StateMachine Implementation

```
// =========================================================
// Base or work position requested

IF bWorkPosReq THEN
bMoveToWork := TRUE;
END_IF

IF bBasePosReq OR NOT bWorkPosReq THEN
bMoveToWork := FALSE;
END_IF

// =========================================================
```

Property bState_AtBasePos Declaration

```
PROPERTY PUBLIC bState_AtBasePos : BOOL
```

Property bState_AtBasePos Get Declaration

```
VAR
END_VAR
```

Property bState_AtBasePos Get Implementation

```
bState_AtBasePos := bAtBasePos;
```

Property bState_AtWorkPos Declaration

```
PROPERTY PUBLIC bState_AtWorkPos : BOOL
```

Property bState_AtWorkPos Get Declaration

```
VAR
END_VAR
```

Property bState_AtWorkPos Get Implementation

```
bState_AtWorkPos := bAtWorkPos;
```

Property bState_MoveToWork Declaration

```
PROPERTY PUBLIC bState_MoveToWork : BOOL
```

Property bState_MoveToWork Get Declaration

```
VAR
END_VAR
```

Property bState_MoveToWork Get Implementation

```
bState_MoveToWork := bMoveToWork;
```

5.2 FB_CylinderDiag.TcPOU

Declaration

```
(* FB_CylinderDiag - number of control signals:
one direction is controllable
- type of feedback signal:
feedback in base and work position
- with position diagnosis *)

FUNCTION_BLOCK FB_CylinderDiag EXTENDS FB_Cylinder
VAR_INPUT
tTimeOut : TIME; // Time for watchdog that monitores if cylinder reaches base/work position
END_VAR
VAR_OUTPUT
bError : BOOL; // Error signal (diagnosed from position watchdog)
sErrorMsg : STRING; // Error message
END_VAR
VAR
fbTriggerError : R_TRIG; // Trigger to recognize rising edge of error
bErrorMove : BOOL; // Move error
fbTimerWatchDog : TON; // Watchdog timer for monitoring if cylinder reaches base/work position
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method StateMachine Declaration

```
METHOD PROTECTED StateMachine
VAR_INPUT
bBasePosReq : BOOL;
bWorkPosReq : BOOL;
END_VAR
```

Method StateMachine Implementation

```
// =========================================================

IF NOT bError THEN
// Calling method StateMachine of base class FB_Cylinder via 'SUPER^.'
SUPER^.StateMachine(bBasePosReq := bBasePosReq,
bWorkPosReq := bWorkPosReq );

// Diagnosis
Diag();
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
END_VAR
```

Method Reset Implementation

```
// =========================================================
// Calling method Reset of base class FB_Cylinder via 'SUPER^.'
SUPER^.Reset();

// Reset error
bError := FALSE;
sErrorMsg := '';

// =========================================================
```

Method Diag Declaration

```
METHOD PROTECTED Diag
VAR_INPUT
END_VAR
```

Method Diag Implementation

```
// =========================================================
// Timer starts when cylinder is not in base and work position

IF NOT bAtBasePos AND NOT bAtWorkPos THEN
fbTimerWatchDog(IN := TRUE,
PT := tTimeOut);
ELSE
fbTimerWatchDog(IN := FALSE,
PT := tTimeOut);
END_IF

// =========================================================
// Error if cylinder does not reach base or work position

fbTriggerError(CLK := fbTimerWatchDog.Q);

bError := fbTimerWatchDog.Q;

IF fbTriggerError.Q THEN
bErrorMove := bMoveToWork;
END_IF

IF bError THEN
IF bErrorMove = bMoveToWork THEN
IF bMoveToWork THEN
sErrorMsg := 'Work position not reached';
ELSE
sErrorMsg := 'Base position not reached';
END_IF
ELSE
bError := FALSE;
sErrorMsg := '';
END_IF
ELSE
sErrorMsg := '';
END_IF

// =========================================================
```

5.3 FB_CylinderTemp.TcPOU

Declaration

```
(* FB_CylinderTemp - number of control signals:
one direction is controllable
- type of feedback signal:
feedback in base and work position
- with temperature signal *)

FUNCTION_BLOCK FB_CylinderTemp EXTENDS FB_Cylinder
VAR_INPUT
fTempCurrent AT %I* : LREAL; // Hardware input signal: cylinder temperature
```

```
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

5.4 FB_CylinderTempDiag.TcPOU

Declaration

```
(* FB_CylinderTempDiag - number of control signals:
one direction is controllable
- type of feedback signal:
feedback in base and work position
- with temperature diagnosis *)

FUNCTION_BLOCK FB_CylinderTempDiag EXTENDS FB_CylinderTemp
VAR_INPUT
fTempMax AT %I* : LREAL; // Global input signal: maximal allowed cylinder temperature for temperature monitoring
fTempMin AT %I* : LREAL; // Global input signal: minimal allowed cylinder temperature for temperature monitoring
END_VAR
VAR_OUTPUT
bError : BOOL; // Error signal (diagnosed from temperature monitoring)
sErrorMsg : STRING; // Error message
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method StateMachine Declaration

```
METHOD PROTECTED StateMachine
VAR_INPUT
bBasePosReq : BOOL;
bWorkPosReq : BOOL;
END_VAR
```

Method StateMachine Implementation

```
// =========================================================

IF NOT bError THEN
// Calling method StateMachine of base class FB_Cylinder via 'SUPER^.'
SUPER^.StateMachine(bBasePosReq := bBasePosReq,
bWorkPosReq := bWorkPosReq );

// Diagnosis
Diag();
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
END_VAR
```

Method Reset Implementation

```
// =========================================================
// Calling method Reset of base class FB_Cylinder via 'SUPER^.'
SUPER^.Reset();

// Reset error
bError := FALSE;
sErrorMsg := '';

// =========================================================
```

Method Diag Declaration

```
METHOD PROTECTED Diag
VAR_INPUT
END_VAR
```

Method Diag Implementation

```
// =========================================================
// Error if temperatur of cylinder is not in accepted interval

bError := NOT (fTempCurrent >= fTempMin AND fTempCurrent <= fTempMax);

IF bError THEN
IF fTempCurrent > fTempMax THEN
sErrorMsg := 'Temperature above maximum';
ELSE
sErrorMsg := 'Temperature below minimum';
END_IF
ELSE
sErrorMsg := '';
END_IF

// =========================================================
```

5.5 FB_CylinderTempRecord.TcPOU

Declaration

```
(* FB_CylinderTempRecord - number of control signals:
one direction is controllable
- type of feedback signal:
feedback in base and work position
- with temperature recording for data analysis *)

FUNCTION_BLOCK FB_CylinderTempRecord EXTENDS FB_CylinderTemp
VAR_INPUT
bRecordStart : BOOL; // Signal to start the temperature recording
tIntervalTime : TIME; // Time of intervals to record the temperature
END_VAR
VAR_OUTPUT
bRecordDone : BOOL; // True if the temperature recording is done
aTemps : ARRAY[1..100] OF LREAL; // Array with recorded temperatures
END_VAR
VAR
fbTimerInterval : TON; // Timer to control the recording interval
nIndex : INT; // Index to handle temperature array
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method Record Declaration

```
METHOD Record
VAR_INPUT
END_VAR
```

Method Record Implementation

```
// =========================================================
// Recording array filled

bRecordDone := (nIndex = 100);

// =========================================================
// Recording temperatures

IF bRecordStart AND NOT bRecordDone AND NOT fbTimerInterval.Q THEN
fbTimerInterval(IN := TRUE,
PT := tIntervalTime );
END_IF

IF fbTimerInterval.Q THEN
fbTimerInterval(IN := FALSE);

nIndex := nIndex + 1;
aTemps[nIndex] := fTempCurrent;
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
END_VAR
```

Method Reset Implementation

```
// =========================================================

// Calling method Reset of base class FB_Cylinder via 'SUPER^.'
SUPER^.Reset();

// Reset temperature array
FOR nIndex := 1 TO 100 BY 1 DO
aTemps[nIndex] := 0;
END_FOR

// Reset index
nIndex := 0;

// =========================================================
```

6 01_Submodules\Signal handling\ITFs

6.1 I_SignalHandling.TcIO

Declaration

```
INTERFACE I_SignalHandling
```

Method Enable Declaration

```
METHOD PUBLIC Enable
VAR_INPUT
bEnable : BOOL;
END_VAR
```

Method SetOutput Declaration

```
METHOD SetOutput
VAR_INPUT
END_VAR
```

7 01_Submodules\Signal handling\POUs

7.1 FB_SignalHandling.TcPOU

Declaration

```
FUNCTION_BLOCK FB_SignalHandling IMPLEMENTS I_SignalHandling
VAR_INPUT
bInput AT %I* : BOOL; // Hardware input signal, e.g. sensor signal, enable for fuse
END_VAR
VAR_OUTPUT
bOut AT %Q* : BOOL; // Processed output signal, e.g. delayed, inverted
END_VAR
VAR
bEnableLocal : BOOL; // Enable processing of input signal
END_VAR
```

Implementation

```
// ==========================================================
;
// ==========================================================
```

Method Enable Declaration

```
METHOD Enable
VAR_INPUT
bEnable : BOOL;
END_VAR
```

Method Enable Implementation

```
// ==========================================================
// Store input in local variable

bEnableLocal := bEnable;

// ==========================================================
```

Method SetOutput Declaration

```
METHOD SetOutput
VAR_INPUT
END_VAR
```

Method SetOutput Implementation

```
// ==========================================================

bOut := bInput AND bEnableLocal;

// ==========================================================
```

7.2 FB_SignalHandlingDelay.TcPOU

Declaration

```
FUNCTION_BLOCK FB_SignalHandlingDelay EXTENDS FB_SignalHandling
VAR_INPUT
tDelay : TIME; // Time to delay input signal (time between hardware and processed software signal)
END_VAR
VAR
bStartProcessing : BOOL; // To start the processing/delay of input signal
fbTimerDelay : TON; // Timer to control the processing/delay of input signal
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method SetOutput Declaration

```
METHOD SetOutput
VAR_INPUT
END_VAR
```

Method SetOutput Implementation

```
// =========================================================

IF bEnableLocal AND bInput THEN
bStartProcessing := TRUE;
END_IF

IF bStartProcessing THEN
fbTimerDelay( IN := TRUE,
PT := tDelay);
ELSE
fbTimerDelay( IN := FALSE);
END_IF

IF fbTimerDelay.Q THEN
bStartProcessing := FALSE;

fbTimerDelay(IN := FALSE);

bOut := TRUE;
ELSE
bOut := FALSE;
END_IF

// =========================================================
```

7.3 FB_SignalHandlingIntern.TcPOU

Declaration

```
FUNCTION_BLOCK FB_SignalHandlingIntern IMPLEMENTS I_SignalHandling
VAR_INPUT
bInput : BOOL; // Input signal, e.g. sensor signal, enable for fuse
END_VAR
VAR_OUTPUT
bOut : BOOL; // Processed output signal, e.g. delayed, inverted
END_VAR
VAR
bEnableLocal : BOOL; // Enable processing of input signal
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method Enable Declaration

```
METHOD Enable
VAR_INPUT
bEnable : BOOL;
END_VAR
```

Method Enable Implementation

```
// =========================================================
// Store input in local variable

bEnableLocal := bEnable;

// =========================================================
```

Method SetOutput Declaration

```
METHOD SetOutput
VAR_INPUT
END_VAR
```

Method SetOutput Implementation

```
// =========================================================

bOut := bInput AND bEnableLocal;

// =========================================================
```

8 02_Subsystems

8.1 E_StateSeparatingAuto.TcDUT

Declaration

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_StateSeparatingAuto :
(
Init := 0,
Start,
CloseClamp,
OpenBarrier,
CloseBarrier,
OpenClamp
);
END_TYPE
```

8.2 E_StateSortingAutoAxis.TcDUT

Declaration

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_StateSortingAutoAxis :
(
WaitForCylinderAtWorkPos := 0,
MoveAxis,
StopAxis
);
END_TYPE
```

8.3 E_StateSortingAutoCylinder.TcDUT

Declaration

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_StateSortingAutoCylinder :
(
DetectBox := 0,
CylToWork,
CylToBase
);
END_TYPE
```

8.4 FB_SeparatingModule.TcPOU

Declaration

```
// Module to separate via sensor, barrier and clamp cylinder and axis
FUNCTION_BLOCK FB_SeparatingModule EXTENDS FB_Subsystem_Root
VAR_INPUT
// =============== Control the state and mode of machine ========================
bExecute : BOOL; // To control the process execution of separating module (true in state EXECUTE)
bSemiStart : BOOL; // To trigger one process exeuction in semi automatic mode

// =============== Information about number of boxes being sorted ===============
nBoxesOnTheWay : INT; // Actual number of boxes being processed

// =============== Cylinders buttons ============================================
bButtonClampToWorkIn : BOOL; // Signal of button to move clamp cylinder to work position
bButtonClampToBaseIn : BOOL; // Signal of button to move clamp cylinder to base position
bButtonBarrierToWorkIn : BOOL; // Signal of button to move barrier cylinder to work position
bButtonBarrierToBaseIn : BOOL; // Signal of button to move barrier cylinder to base position
END_VAR
VAR_OUTPUT
// =============== Process information ==========================================
bSeparated : BOOL; // Gets true if one separating process is done

// =============== For movement simulation on visualization =====================
bClampToWork : BOOL; // Simulation of clamp cylinder movement to work position
bBarrierToWork : BOOL; // Simulation of barrier cylinder movement to work position

// =============== Error variables ==============================================
bClampError : BOOL; // Error signal of clamp cylinder
sClampErrorMsg : STRING; // Error message of clamp cylinder
bBarrierError : BOOL; // Error signal of barrier cylinder
sBarrierErrorMsg : STRING; // Error message of barrier cylinder

// =============== Temperature recordings =======================================
aClampTemps : ARRAY[1..100] OF LREAL; // Array with recorded temperatures of clamp cylinder
aBarrierTemps : ARRAY[1..100] OF LREAL; // Array with recorded temperatures of barrier cylinder

// =============== Button return signals ========================================
bButtonClampToWorkOut : BOOL; // Processed signal for button to move clamp cylinder to work position
bButtonClampToBaseOut : BOOL; // Processed signal for button to move clamp cylinder to base position
bButtonBarrierToWorkOut : BOOL; // Processed signal for button to move barrier cylinder to work position
bButtonBarrierToBaseOut : BOOL; // Processed signal for button to move barrier cylinder to base position
END_VAR

VAR
// =============== Function block instance of sensor ============================
fbSensor : FB_SignalHandling; // Sensor needed for separating process

// =============== Function block instances for clamp cylinder ==================
fbClamp : FB_Cylinder; // Without diagnosis and temperature mode
fbClampDiag : FB_CylinderDiag; // With diagnosis of states
fbClampTemp : FB_CylinderTemp; // With temperature mode
fbClampTempDiag : FB_CylinderTempDiag; // With diagnosis of temperature
fbClampTempRecord : FB_CylinderTempRecord; // With record of temperatures

// =============== Interface instance for clamp cylinder ========================
ipClamp : I_Cylinder; // Interface for flexible access to clamp cylinder FBs

// =============== Function block instances for barrier cylinder ================
fbBarrier : FB_Cylinder; // Without diagnosis and temperature mode
fbBarrierDiag : FB_CylinderDiag; // With diagnosis of states
fbBarrierTemp : FB_CylinderTemp; // With temperature mode
fbBarrierTempDiag : FB_CylinderTempDiag; // With diagnosis of temperature
fbBarrierTempRecord : FB_CylinderTempRecord; // With record of temperatures

// =============== Interface instance for barrier cylinder ======================
ipBarrier : I_Cylinder; // Interface for flexible access to barrier cylinder FBs

// =============== Cylinder parameters for clamp and barrier ====================
tTimeOutClamp : TIME; // For clamp cylinder with diagnosis: time in which the cylinder should reach base/work position
tTimeOutBarrier : TIME; // For barrier cylinder with diagnosis: time in which the cylinder should reach base/work position
tRecordIntervalClamp : TIME; // Time of intervals to record the temperature of clamp cylinder
tRecordIntervalBarrier : TIME; // Time of intervals to record the temperature of barrier cylinder
tBarrierAtWork : TIME; // Time value for separating process: barrier cylinder stays at work position

// =============== Manual cylinder control ======================================
fbButtonClampToWork : FB_SignalHandlingIntern; // To move clamp cylinder manually to work position
fbButtonClampToBase : FB_SignalHandlingIntern; // To move clamp cylinder manually to base position
fbButtonBarrierToWork : FB_SignalHandlingIntern; // To move barrier cylinder manually to work position
fbButtonBarrierToBase : FB_SignalHandlingIntern; // To move barrier cylinder manually to base position

// =============== Common variables ==============================================
fbTimerBarrierAtWork : TON; // Timer for opening barrier in (semi) automatic mode
eStateAuto : E_StateSeparatingAuto; // State variable for automatic mode
eStateSemiAuto : E_StateSeparatingAuto; // State variable for semi automatic mode
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method Automatic Declaration

```
METHOD Automatic
VAR_INPUT
END_VAR
```

Method Automatic Implementation

```
// =========================================================
// Noticing work position of barrier cylinder

fbTimerBarrierAtWork(PT := tBarrierAtWork);

// =========================================================
// Control of main axis

IF NOT bHaltRequest THEN
fbAxis.MoveFw();

ELSIF NOT bHaltDone THEN
Stop(bHalt := TRUE);

ELSE
Stop(bHalt := FALSE);
END_IF

// =========================================================
// Process of separating boxes

CASE eStateAuto OF

// =========================================================
E_StateSeparatingAuto.Init:

// Calling method of FB via interface instance
ipBarrier.MoveToBase();

// Accessing property of FB via interface instance
IF ipBarrier.bState_AtBasePos AND bExecute THEN
eStateAuto := E_StateSeparatingAuto.Start;
END_IF

// =========================================================
E_StateSeparatingAuto.Start:

IF fbSensor.bOut AND fbAxis.bMoves THEN
eStateAuto := E_StateSeparatingAuto.CloseClamp;

// Accessing property of FB via interface instance
ELSIF NOT ipClamp.bState_AtWorkPos THEN
// Calling method of FB via interface instance
ipClamp.MoveToWork();
END_IF

// =========================================================
E_StateSeparatingAuto.CloseClamp:

// Calling method of FB via interface instance
ipClamp.MoveToBase();

// Accessing property of FB via interface instance
IF ipClamp.bState_AtBasePos THEN
bSeparated := TRUE;
eStateAuto := E_StateSeparatingAuto.OpenBarrier;
END_IF
```

```
// =========================================================
E_StateSeparatingAuto.OpenBarrier:

// Calling method of FB via interface instance
ipBarrier.MoveToWork();

// Accessing property of FB via interface instance
IF ipBarrier.bState_AtWorkPos THEN
fbTimerBarrierAtWork(IN := TRUE);
END_IF

IF fbTimerBarrierAtWork.Q THEN
fbTimerBarrierAtWork(IN := FALSE);
eStateAuto := E_StateSeparatingAuto.CloseBarrier;
END_IF

// =========================================================
E_StateSeparatingAuto.CloseBarrier:

// Calling method of FB via interface instance
ipBarrier.MoveToBase();

// Accessing property of FB via interface instance
IF ipBarrier.bState_AtBasePos THEN
bSeparated := FALSE;
eStateAuto := E_StateSeparatingAuto.OpenClamp;
END_IF

// =========================================================
E_StateSeparatingAuto.OpenClamp:
// Calling method of FB via interface instance
ipClamp.MoveToWork();

// Accessing property of FB via interface instance
IF ipClamp.bState_AtWorkPos THEN
eStateAuto := E_StateSeparatingAuto.Init;
END_IF
END_CASE

// =========================================================
```

Method CylinderOptions Declaration

```
METHOD CylinderOptions
VAR_INPUT
bClampDiag : BOOL; // If true the clamp cylinder has diagnosis functionality
bClampTemp : BOOL; // If true the clamp cylinder has temperature functionality
bClampRecord : BOOL; // If true the clamp cylinder has recording functionality
bBarrierDiag : BOOL; // If true the barrier cylinder has diagnosis functionality
bBarrierTemp : BOOL; // If true the barrier cylinder has temperature functionality
bBarrierRecord : BOOL; // If true the barrier cylinder has recording functionality
END_VAR
```

Method CylinderOptions Implementation

```
// =========================================================
// Selecting clamp cylinder

// Checking variables to enable/disable diagnosis and temperature mode
IF bClampDiag THEN
IF bClampTemp THEN
// ============== FB with diagnosis and temperature mode ==============
bClampError := fbClampTempDiag.bError; // Assigning output variable of chosen FB to local output variable
sClampErrorMsg := fbClampTempDiag.sErrorMsg;
ipClamp := fbClampTempDiag; // Assigning chosen FB instance to interface instance

ELSE
// ============== FB with diagnosis and without temperature mode ==============
fbClampDiag.tTimeOut := tTimeOutClamp; // Setting special data for selected FB
bClampError := fbClampDiag.bError; // Assigning output variable of chosen FB to local output variable
```

```
sClampErrorMsg := fbClampDiag.sErrorMsg;
ipClamp := fbClampDiag; // Assigning chosen FB instance to interface instance
END_IF
ELSE
bClampError := FALSE;
sClampErrorMsg := '';

IF bClampTemp THEN
IF bClampRecord THEN
// ============== FB without diagnosis and with temperature recording ==============
fbClampTempRecord.tIntervalTime := tRecordIntervalClamp; // Time of record interval
fbClampTempRecord.bRecordStart := bComprAirEnabledLocal; // Recording if compressed air is set

fbClampTempRecord.Record(); // Calling method of function block FB_CylinderTempRecord

aClampTemps := fbClampTempRecord.aTemps; // Saving output variable
ipClamp := fbClampTempRecord; // Assigning chosen FB instance to interface instance
ELSE
// ============== FB without diagnosis and with temperature mode, but without recording ==============
ipClamp := fbClampTemp; // Assigning chosen FB instance to interface instance
END_IF
ELSE
// ============== FB without diagnosis and without temperature mode ==============
ipClamp := fbClamp; // Assigning chosen FB instance to interface instance
END_IF
END_IF

// =========================================================
// Selecting barrier cylinder

// Checking variables to enable/disable diagnosis and temperature mode
IF bBarrierDiag THEN
IF bBarrierTemp THEN
// ============== FB with diagnosis and temperature mode ==============
bBarrierError := fbBarrierTempDiag.bError; // Assigning output variable of chosen FB to local output variable
sBarrierErrorMsg := fbBarrierTempDiag.sErrorMsg;
ipBarrier := fbBarrierTempDiag; // Assigning chosen FB instance to interface instance

ELSE
// ============== FB with diagnosis and without temperature mode ==============
fbBarrierDiag.tTimeOut := tTimeOutBarrier; // Setting special data for selected FB
bBarrierError := fbBarrierDiag.bError; // Assigning output variable of chosen FB to local output variable
sBarrierErrorMsg := fbBarrierDiag.sErrorMsg;
ipBarrier := fbBarrierDiag; // Assigning chosen FB instance to interface instance
END_IF
ELSE
bBarrierError := FALSE;
sBarrierErrorMsg := '';

IF bBarrierTemp THEN
IF bBarrierRecord THEN
// ============== FB without diagnosis and with temperature recording ==============
fbBarrierTempRecord.tIntervalTime := tRecordIntervalBarrier; // Time of record interval
fbBarrierTempRecord.bRecordStart := bComprAirEnabledLocal; // Recording if compressed air is set

fbBarrierTempRecord.Record(); // Calling method of function block FB_CylinderTempRecord

aBarrierTemps := fbBarrierTempRecord.aTemps; // Saving output variable
ipBarrier := fbBarrierTempRecord; // Assigning chosen FB instance to interface instance
ELSE
// ============== FB without diagnosis and with temperature mode, but without recording ==============
ipBarrier := fbBarrierTemp; // Assigning chosen FB instance to interface instance
END_IF
ELSE
// ============== FB without diagnosis and without temperature mode ==============
ipBarrier := fbBarrier; // Assigning chosen FB instance to interface instance
END_IF
END_IF

// =========================================================
```

Method Enable Declaration

```
METHOD Enable
VAR_INPUT
bComprAirEnabled : BOOL; // Enable of compressed air
bAxisEnable : BOOL; // Enable of axis
bSensorEnable : BOOL; // Enable of sensor
bManualAxisEnable : BOOL; // Enable of manual axis control
END_VAR
```

Method Enable Implementation

```
// =========================================================
// Calling method Enable of base class FB_Module via 'SUPER^.'

SUPER^.Enable( bComprAirEnabled := bComprAirEnabled,
bAxisEnable := bAxisEnable,
bSensorEnable := bSensorEnable,
bManualAxisEnable := bManualAxisEnable);

// =========================================================
// Sensor control

fbSensor.Enable(bEnable := bSensorEnable);

// =========================================================
// Cylinder control (accessing property of FB via interfaces intances)

// Clamp cylinder
fbButtonClampToWork.Enable(bEnable := bManualCylinderEnable AND NOT ipClamp.bState_AtWorkPos);
fbButtonClampToBase.Enable(bEnable := bManualCylinderEnable AND NOT ipClamp.bState_AtBasePos);

fbButtonClampToBase.bInput := NOT fbButtonClampToWork.bInput;

// Barrier cylinder
fbButtonBarrierToWork.Enable(bEnable := bManualCylinderEnable AND NOT ipBarrier.bState_AtWorkPos);
fbButtonBarrierToBase.Enable(bEnable := bManualCylinderEnable AND NOT ipBarrier.bState_AtBasePos);

fbButtonBarrierToBase.bInput := NOT fbButtonBarrierToWork.bInput;

// =========================================================
```

Method InputOutput Declaration

```
METHOD InputOutput
VAR_INPUT
END_VAR
```

Method InputOutput Implementation

```
// =========================================================
// Calling method SetOutput of base class FB_Module via 'SUPER^.'

SUPER^.InputOutput();

// =========================================================
// Sensor

fbSensor.SetOutput();

// =========================================================
// Cylinder buttons

fbButtonClampToWork.SetOutput();
fbButtonClampToBase.SetOutput();
fbButtonBarrierToWork.SetOutput();
fbButtonBarrierToBase.SetOutput();

// =========================================================
// Output variables for cylinders

// Accessing properties of selected FB via interface instances
bClampToWork := ipClamp.bState_MoveToWork;
bBarrierToWork := ipBarrier.bState_MoveToWork;

// =========================================================
// Buttons

fbButtonClampToWork.bInput := bButtonClampToWorkIn;
fbButtonClampToBase.bInput := bButtonClampToBaseIn;
fbButtonBarrierToWork.bInput := bButtonBarrierToWorkIn;
fbButtonBarrierToBase.bInput := bButtonBarrierToBaseIn;
bButtonClampToWorkOut := fbButtonClampToWork.bOut;
bButtonClampToBaseOut := fbButtonClampToBase.bOut;
bButtonBarrierToWorkOut := fbButtonBarrierToWork.bOut;
bButtonBarrierToBaseOut := fbButtonBarrierToBase.bOut;

// =========================================================
```

Method Maintenance Declaration

```
METHOD Maintenance
VAR_INPUT
END_VAR
```

Method Maintenance Implementation

```
// =========================================================
// Calling own method Manual of this class FB_SeparatingModule via 'THIS^.'

THIS^.Manual();

// =========================================================
```

Method Manual Declaration

```
METHOD Manual
VAR_INPUT
END_VAR
```

Method Manual Implementation

```
// =========================================================
// Calling method Manual of base class FB_Module via 'SUPER^.'

SUPER^.Manual();

// =========================================================
// Manual cylinder control (Calling methods of FBs via interface instances)

// Clamp cylinder to work position
IF fbButtonClampToWork.bOut THEN
ipClamp.MoveToWork();
// Clamp cylinder to base position
ELSIF fbButtonClampToBase.bOut THEN
ipClamp.MoveToBase();
END_IF

// Barrier cylinder to work position
IF fbButtonBarrierToWork.bOut THEN
ipBarrier.MoveToWork();
// Barrier cylinder to base position
ELSIF fbButtonBarrierToBase.bOut THEN
ipBarrier.MoveToBase();
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
bReset : BOOL; // True if machine is in state RESETTING
END_VAR
```

Method Reset Implementation

```
// =========================================================
// Calling method Reset of base class FB_Module via 'SUPER^.'

SUPER^.Reset(bReset := bReset);

// =========================================================
// Resetting cylinder and axis

IF fbTriggerResetStart.Q THEN
bSeparated := FALSE;

fbTimerBarrierAtWork(IN := FALSE);

// Calling methods of FBs via interface instances
ipClamp.Reset();
ipBarrier.Reset();
END_IF

// =========================================================
```

Method ResetState Declaration

```
METHOD PROTECTED ResetState
VAR_INPUT
END_VAR
```

Method ResetState Implementation

```
// =========================================================

eStateAuto := E_StateSeparatingAuto.Init;
eStateSemiAuto := E_StateSeparatingAuto.Init;

// =========================================================
```

Method Semiautomatic Declaration

```
METHOD Semiautomatic
VAR_INPUT
END_VAR
```

Method Semiautomatic Implementation

```
// =========================================================
// Noticing work position of barrier cylinder

fbTimerBarrierAtWork(PT := tBarrierAtWork);

// =========================================================
// Control of main axis

IF (nBoxesOnTheWay = 0) AND (eStateSemiAuto = E_StateSeparatingAuto.Init) THEN
Stop(bHalt := TRUE);
END_IF

IF bHaltRequest THEN
IF bHaltDone THEN
Stop(bHalt := FALSE);
ELSE
Stop(bHalt := TRUE);
END_IF
ELSE
fbAxis.MoveFw();
END_IF

// =========================================================
// Process of separating boxes

CASE eStateSemiAuto OF
// =========================================================
E_StateSeparatingAuto.Init:

// Calling method of FB via interface instance
ipBarrier.MoveToBase();

// Accessing property of FB via interface instance
IF ipBarrier.bState_AtBasePos AND bExecute AND bSemiStart THEN
eStateSemiAuto := E_StateSeparatingAuto.Start;
END_IF

// =========================================================
E_StateSeparatingAuto.Start:

IF fbSensor.bOut THEN
eStateSemiAuto := E_StateSeparatingAuto.CloseClamp;
ELSE
// Accessing property of FB via interface instance
IF NOT ipClamp.bState_AtWorkPos THEN
// Calling method of FB via interface instance
ipClamp.MoveToWork();
END_IF

fbAxis.MoveFw();
END_IF

// =========================================================
E_StateSeparatingAuto.CloseClamp:

// Calling method of FB via interface instance
ipClamp.MoveToBase();

// Accessing property of FB via interface instance
IF ipClamp.bState_AtBasePos THEN
bSeparated := TRUE;
eStateSemiAuto := E_StateSeparatingAuto.OpenBarrier;
END_IF

// =========================================================
E_StateSeparatingAuto.OpenBarrier:

// Calling method of FB via interface instance
ipBarrier.MoveToWork();

// Accessing property of FB via interface instance
IF ipBarrier.bState_AtWorkPos THEN
fbTimerBarrierAtWork( IN := TRUE );
END_IF

IF fbTimerBarrierAtWork.Q THEN
fbTimerBarrierAtWork( IN := FALSE );
eStateSemiAuto := E_StateSeparatingAuto.CloseBarrier;
END_IF

// =========================================================
E_StateSeparatingAuto.CloseBarrier:

// Calling method of FB via interface instance
```

```
ipBarrier.MoveToBase();

// Accessing property of FB via interface instance
IF ipBarrier.bState_AtBasePos THEN
bSeparated := FALSE;
eStateSemiAuto := E_StateSeparatingAuto.OpenClamp;
END_IF

// =========================================================
E_StateSeparatingAuto.OpenClamp:

// Calling method of FB via interface instance
ipClamp.MoveToWork();

// Accessing property of FB via interface instance
IF ipClamp.bState_AtWorkPos THEN
eStateSemiAuto := E_StateSeparatingAuto.Init;
END_IF
END_CASE

// =========================================================
```

Property bState_BarrierAtBasePos Declaration

```
PROPERTY bState_BarrierAtBasePos : BOOL
```

Property bState_BarrierAtBasePos Get Declaration

```
VAR
END_VAR
```

Property bState_BarrierAtBasePos Get Implementation

```
bState_BarrierAtBasePos := ipBarrier.bState_AtBasePos;
```

Property bState_BarrierAtWorkPos Declaration

```
PROPERTY bState_BarrierAtWorkPos : BOOL
```

Property bState_BarrierAtWorkPos Get Declaration

```
VAR
END_VAR
```

Property bState_BarrierAtWorkPos Get Implementation

```
bState_BarrierAtWorkPos := ipBarrier.bState_AtWorkPos;
```

Property bState_ClampAtBasePos Declaration

```
PROPERTY bState_ClampAtBasePos : BOOL
```

Property bState_ClampAtBasePos Get Declaration

```
VAR
END_VAR
```

Property bState_ClampAtBasePos Get Implementation

```
bState_ClampAtBasePos := ipClamp.bState_AtBasePos;
```

Property bState_ClampAtWorkPos Declaration

```
PROPERTY bState_ClampAtWorkPos : BOOL
```

Property bState_ClampAtWorkPos Get Declaration

```
VAR
END_VAR
```

Property bState_ClampAtWorkPos Get Implementation

```
bState_ClampAtWorkPos := ipClamp.bState_AtWorkPos;
```

Property tBarrierAtWorkPos Declaration

```
PROPERTY tBarrierAtWorkPos : TIME
```

Property tBarrierAtWorkPos Get Declaration

```
VAR
END_VAR
```

Property tBarrierAtWorkPos Get Implementation

```
tBarrierAtWorkPos := tBarrierAtWork;
```

Property tBarrierAtWorkPos Set Declaration

```
VAR
END_VAR
```

Property tBarrierAtWorkPos Set Implementation

```
tBarrierAtWork := tBarrierAtWorkPos;
```

Property tRecordIntervalOfBarrier Declaration

```
PROPERTY tRecordIntervalOfBarrier : TIME
```

Property tRecordIntervalOfBarrier Get Declaration

```
VAR
END_VAR
```

Property tRecordIntervalOfBarrier Get Implementation

```
tRecordIntervalOfBarrier := tRecordIntervalBarrier;
```

Property tRecordIntervalOfBarrier Set Declaration

```
VAR
END_VAR
```

Property tRecordIntervalOfBarrier Set Implementation

```
tRecordIntervalBarrier := tRecordIntervalOfBarrier;
```

Property tRecordIntervalOfClamp Declaration

```
PROPERTY tRecordIntervalOfClamp : TIME
```

Property tRecordIntervalOfClamp Get Declaration

```
VAR
END_VAR
```

Property tRecordIntervalOfClamp Get Implementation

```
tRecordIntervalOfClamp := tRecordIntervalClamp;
```

Property tRecordIntervalOfClamp Set Declaration

```
VAR
END_VAR
```

Property tRecordIntervalOfClamp Set Implementation

```
tRecordIntervalClamp := tRecordIntervalOfClamp;
```

Property tTimeOutOfBarrier Declaration

```
PROPERTY tTimeOutOfBarrier : TIME
```

Property tTimeOutOfBarrier Get Declaration

```
VAR
END_VAR
```

Property tTimeOutOfBarrier Get Implementation

```
tTimeOutOfBarrier := tTimeOutBarrier;
```

Property tTimeOutOfBarrier Set Declaration

```
VAR
END_VAR
```

Property tTimeOutOfBarrier Set Implementation

```
tTimeOutBarrier := tTimeOutOfBarrier;
```

Property tTimeOutOfClamp Declaration

```
PROPERTY tTimeOutOfClamp : TIME
```

Property tTimeOutOfClamp Get Declaration

```
VAR
END_VAR
```

Property tTimeOutOfClamp Get Implementation

```
tTimeOutOfClamp := tTimeOutClamp;
```

Property tTimeOutOfClamp Set Declaration

```
VAR
END_VAR
```

Property tTimeOutOfClamp Set Implementation

```
tTimeOutClamp := tTimeOutOfClamp;
```

8.5 FB_SortingModule.TcPOU

Declaration

```
// Module to select via sensor, cylinder and axis
FUNCTION_BLOCK FB_SortingModule EXTENDS FB_Subsystem_Root
VAR_INPUT
// =============== Cylinder buttons =============================================
bButtonCylToWorkIn : BOOL; // Signal of button to move cylinder to work position
bButtonCylToBaseIn : BOOL; // Signal of button to move cylinder to base position
END_VAR
VAR_OUTPUT
// =============== Process information ==========================================
bSelected : BOOL; // Gets true if one sorting process is done

// =============== For movement simulation on visualization =====================
bCylinderToWork : BOOL; // Simulation of cylinder movement to work position

// =============== Error variables ==============================================
bCylError : BOOL; // Error signal of cylinder
sCylErrorMsg : STRING; // Error message of cylinder

// =============== Temperature recording ========================================
aTemps : ARRAY[1..100] OF LREAL; // Array with recorded temperatures of cylinder

// =============== Button return signals ========================================
bButtonCylToWorkOut : BOOL; // Processed signal for button to move cylinder to work position
bButtonCylToBaseOut : BOOL; // Processed signal for button to move cylinder to base position
END_VAR
VAR
// =============== Function block instance of sensor ============================
fbSensorDelay : FB_SignalHandlingDelay; // Sensor needed for sorting process

// =============== Function block instances for cylinder ========================
fbCylinder : FB_Cylinder; // Without diagnosis and temperature mode
fbCylinderDiag : FB_CylinderDiag; // With diagnosis of states
fbCylinderTemp : FB_CylinderTemp; // With temperature mode
fbCylinderTempDiag : FB_CylinderTempDiag; // With diagnosis of temperature
fbCylinderTempRecord : FB_CylinderTempRecord; // With record of temperatures

// =============== Interface instance for cylinder ==============================
ipCylinder : I_Cylinder; // Interface for flexible access to cylinder FBs

// =============== Submodule parameters =========================================
tTimeOutCylinder : TIME; // For cylinder with diagnosis: time in which the cylinder should reach base/work
position
tRecordIntervalCylinder : TIME; // Time of intervals to record the temperature of cylinder
tSensorDelay : TIME; // Time to delay sensor signal (time between hardware and processed software signal)
tMoveAxis : TIME; // Time to move axis

// =============== Manual cylinder control ======================================
fbButtonCylToWork : FB_SignalHandlingIntern; // To move cylinder manually to work position
fbButtonCylToBase : FB_SignalHandlingIntern; // To move cylinder manually to base position

// =============== Common variables =============================================
fbTimerAxis : TON; // Timer to move axis in (semi) automatic mode
fbTriggerCylAtWork : R_TRIG; // Trigger to recognize rising edge of cylinder being at work position
eStateAutoAxis : E_StateSortingAutoAxis; // State variable for automatic mode - process to move axis
eStateAutoCylinder : E_StateSortingAutoCylinder; // State variable for automatic mode - process to move cylinder
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method Automatic Declaration

```
METHOD Automatic
VAR_INPUT
END_VAR
```

Method Automatic Implementation

```
// =========================================================
// Process of sorting boxes - axis movement

fbTriggerCylAtWork(CLK := ipCylinder.bState_AtWorkPos); // Accessing property of FB via interface instance

CASE eStateAutoAxis OF

// =========================================================
E_StateSortingAutoAxis.WaitForCylinderAtWorkPos:

IF fbTriggerCylAtWork.Q THEN
fbTimerAxis(IN := TRUE,
PT := tMoveAxis);

eStateAutoAxis := E_StateSortingAutoAxis.MoveAxis;
END_IF

// =========================================================
E_StateSortingAutoAxis.MoveAxis:

fbTimerAxis();

IF bHaltRequest THEN
eStateAutoAxis := E_StateSortingAutoAxis.StopAxis;
ELSIF fbTimerAxis.Q THEN
fbTimerAxis(IN := FALSE);

bSelected := TRUE;
eStateAutoAxis := E_StateSortingAutoAxis.StopAxis;
ELSE
fbAxis.MoveFw();
END_IF


// =========================================================
E_StateSortingAutoAxis.StopAxis:

bSelected := FALSE;

IF NOT bHaltDone THEN
Stop(bHalt := TRUE);
ELSE
Stop(bHalt := FALSE);

eStateAutoAxis := E_StateSortingAutoAxis.WaitForCylinderAtWorkPos;
END_IF
END_CASE

// =========================================================
// Process of sorting boxes - cylinder movement

CASE eStateAutoCylinder OF

// =========================================================
// Detecting a box
E_StateSortingAutoCylinder.DetectBox:

ipCylinder.MoveToBase(); // Calling method of FB via interface instance

IF fbSensorDelay.bOut THEN
eStateAutoCylinder := E_StateSortingAutoCylinder.CylToWork;
END_IF
```

```
// =========================================================
// Moving cylinder to work position
E_StateSortingAutoCylinder.CylToWork:

ipCylinder.MoveToWork(); // Calling method of FB via interface instance

IF ipCylinder.bState_AtWorkPos THEN // Accessing property of FB via interface instance
eStateAutoCylinder := E_StateSortingAutoCylinder.CylToBase;
END_IF

// =========================================================
// Moving cylinder back to base position
E_StateSortingAutoCylinder.CylToBase:

ipCylinder.MoveToBase(); // Calling method of FB via interface instance

IF ipCylinder.bState_AtBasePos THEN // Accessing property of FB via interface instance
eStateAutoCylinder := E_StateSortingAutoCylinder.DetectBox;
END_IF
END_CASE

// =========================================================
```

Method CylinderOptions Declaration

```
METHOD CylinderOptions
VAR_INPUT
bCylinderDiag : BOOL; // If true the cylinder has diagnosis functionality
bCylinderTemp : BOOL; // IF true the cylinder has temperature functionality
bCylinderRecord : BOOL; // If true the cylinder has recording functionality
END_VAR
```

Method CylinderOptions Implementation

```
// =========================================================
// Selecting cylinder

// Checking variables to enable/disable diagnosis and temperature mode
IF bCylinderDiag THEN
IF bCylinderTemp THEN
// ============== FB with diagnosis and temperature mode ==============
bCylError := fbCylinderTempDiag.bError; // Assigning output variable of chosen FB to local output variable
sCylErrorMsg := fbCylinderTempDiag.sErrorMsg;
ipCylinder := fbCylinderTempDiag; // Assigning chosen FB instance to interface instance

ELSE
// ============== FB with diagnosis and without temperature mode ==============
fbCylinderDiag.tTimeOut := tTimeOutCylinder; // Setting special data for selected FB
bCylError := fbCylinderDiag.bError; // Assigning output variable of chosen FB to local output variable
sCylErrorMsg := fbCylinderDiag.sErrorMsg;
ipCylinder := fbCylinderDiag; // Assigning chosen FB instance to interface instance
END_IF

ELSE
bCylError := FALSE;
sCylErrorMsg := '';

IF bCylinderTemp THEN
IF bCylinderRecord THEN
// ============== FB without diagnosis and with temperature recording ==============
fbCylinderTempRecord.tIntervalTime := tRecordIntervalCylinder; // Time of record interval
fbCylinderTempRecord.bRecordStart := bComprAirEnabledLocal; // Recording if compressed air is set

fbCylinderTempRecord.Record(); // Calling method of function block FB_CylinderTempRecord

aTemps := fbCylinderTempRecord.aTemps; // Saving output variable
ipCylinder := fbCylinderTempRecord; // Assigning chosen FB instance to interface instance
ELSE
// ============== FB without diagnosis and with temperature mode, but without recording ==============
ipCylinder := fbCylinderTemp; // Assigning chosen FB instance to interface instance
```

```
END_IF

ELSE
// ============== FB without diagnosis and without temperature mode ==============
ipCylinder := fbCylinder; // Assigning chosen FB instance to interface instance
END_IF
END_IF

// =========================================================
```

Method Enable Declaration

```
METHOD Enable
VAR_INPUT
bComprAirEnabled : BOOL; // Enable of compressed air
bAxisEnable : BOOL; // Enable of axis
bSensorEnable : BOOL; // Enable of sensor
bManualAxisEnable : BOOL; // Enable of manual axis control
END_VAR
```

Method Enable Implementation

```
// =========================================================
// Calling method Enable of base class FB_Module via 'SUPER^.'

SUPER^.Enable( bComprAirEnabled := bComprAirEnabled,
bAxisEnable := bAxisEnable,
bSensorEnable := bSensorEnable,
bManualAxisEnable := bManualAxisEnable);

// =========================================================
// Sensor control

fbSensorDelay.Enable(bEnable := bSensorEnable);

// =========================================================
// Cylinder control (accessing property of FB via interfaces intance)

fbButtonCylToWork.Enable(bEnable := bManualCylinderEnable AND NOT ipCylinder.bState_AtWorkPos);
fbButtonCylToBase.Enable(bEnable := bManualCylinderEnable AND NOT ipCylinder.bState_AtBasePos);

fbButtonCylToBase.bInput := NOT fbButtonCylToWork.bInput;

// =========================================================
```

Method InputOutput Declaration

```
METHOD InputOutput
VAR_INPUT
END_VAR
```

Method InputOutput Implementation

```
// =========================================================
// Calling method SetOutput of base class FB_Module via 'SUPER^.'

SUPER^.InputOutput();

// =========================================================
// Sensor

fbSensorDelay.SetOutput();

// =========================================================
// Cylinder buttons

fbButtonCylToWork.SetOutput();
fbButtonCylToBase.SetOutput();
```

```
// =========================================================
// Output variables for cylinder

bCylinderToWork := ipCylinder.bState_MoveToWork; // Accessing property of selected FB via interface instance

// =========================================================
// Buttons

fbButtonCylToWork.bInput := bButtonCylToWorkIn;
fbButtonCylToBase.bInput := bButtonCylToBaseIn;
bButtonCylToWorkOut := fbButtonCylToWork.bOut;
bButtonCylToBaseOut := fbButtonCylToBase.bOut;

// =========================================================
```

Method Maintenance Declaration

```
METHOD Maintenance
VAR_INPUT
END_VAR
```

Method Maintenance Implementation

```
// =========================================================
// Calling own method Manual of this class FB_SortingModule via 'THIS^.'

THIS^.Manual();

// =========================================================
```

Method Manual Declaration

```
METHOD Manual
VAR_INPUT
END_VAR
```

Method Manual Implementation

```
// =========================================================
// Calling method Manual of base class FB_Module via 'SUPER^.'

SUPER^.Manual();

// =========================================================
// Manual cylinder control (Calling methods of FB via interface instance)

// Cylinder to work position
IF fbButtonCylToWork.bOut THEN
ipCylinder.MoveToWork();
// Cylinder to base position
ELSIF fbButtonCylToBase.bOut THEN
ipCylinder.MoveToBase();
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD Reset
VAR_INPUT
bReset : BOOL; // True if machine is in state RESETTING
END_VAR
```

Method Reset Implementation

```
// =========================================================
// Calling action Reset of base class FB_Module via 'SUPER^.'

SUPER^.Reset(bReset := bReset);

// =========================================================
// Resetting cylinder and axis

IF fbTriggerResetStart.Q THEN
bSelected := FALSE;

fbTimerAxis(IN := FALSE);

// Calling method of FB via interface instance
ipCylinder.Reset();
END_IF

// =========================================================
```

Method ResetState Declaration

```
METHOD PROTECTED ResetState
VAR_INPUT
END_VAR
```

Method ResetState Implementation

```
// =========================================================

eStateAutoAxis := E_StateSortingAutoAxis.WaitForCylinderAtWorkPos;
eStateAutoCylinder := E_StateSortingAutoCylinder.DetectBox;

// =========================================================
```

Method Semiautomatic Declaration

```
METHOD Semiautomatic
VAR_INPUT
END_VAR
```

Method Semiautomatic Implementation

```
// =========================================================
// Calling own method Automatic of this class FB_SortingModule via 'THIS^.'

THIS^.Automatic();

// =========================================================
```

Method Starting Declaration

```
METHOD Starting
VAR_INPUT
bStarting : BOOL; // True if machine is in state STARTING
END_VAR
```

Method Starting Implementation

```
// =========================================================
// Setting time to delay sensor

fbSensorDelay.tDelay := tSensorDelay;

// =========================================================
// Calling method Starting of base class FB_Module via 'SUPER^.'
```

```
SUPER^.Starting(bStarting := bStarting);

// =========================================================
```

Property bState_CylinderAtBasePos Declaration

```
PROPERTY bState_CylinderAtBasePos : BOOL
```

Property bState_CylinderAtBasePos Get Declaration

```
VAR
END_VAR
```

Property bState_CylinderAtBasePos Get Implementation

```
bState_CylinderAtBasePos := ipCylinder.bState_AtBasePos;
```

Property bState_CylinderAtWorkPos Declaration

```
PROPERTY bState_CylinderAtWorkPos : BOOL
```

Property bState_CylinderAtWorkPos Get Declaration

```
VAR
END_VAR
```

Property bState_CylinderAtWorkPos Get Implementation

```
bState_CylinderAtWorkPos := ipCylinder.bState_AtWorkPos;
```

Property tAxisMovement Declaration

```
PROPERTY tAxisMovement : TIME
```

Property tAxisMovement Get Declaration

```
VAR
END_VAR
```

Property tAxisMovement Get Implementation

```
tAxisMovement := tMoveAxis;
```

Property tAxisMovement Set Declaration

```
VAR
END_VAR
```

Property tAxisMovement Set Implementation

```
tMoveAxis := tAxisMovement;
```

Property tDelayOfSensor Declaration

```
PROPERTY tDelayOfSensor : TIME
```

Property tDelayOfSensor Get Declaration

```
VAR
END_VAR
```

Property tDelayOfSensor Get Implementation

```
tDelayOfSensor := tSensorDelay;
```

Property tDelayOfSensor Set Declaration

```
VAR
END_VAR
```

Property tDelayOfSensor Set Implementation

```
tSensorDelay := tDelayOfSensor;
```

Property tRecordIntervalOfCylinder Declaration

```
PROPERTY tRecordIntervalOfCylinder : TIME
```

Property tRecordIntervalOfCylinder Get Declaration

```
VAR
END_VAR
```

Property tRecordIntervalOfCylinder Get Implementation

```
tRecordIntervalOfCylinder := tRecordIntervalCylinder;
```

Property tRecordIntervalOfCylinder Set Declaration

```
VAR
END_VAR
```

Property tRecordIntervalOfCylinder Set Implementation

```
tRecordIntervalCylinder := tRecordIntervalOfCylinder;
```

Property tTimeOutOfCylinder Declaration

```
PROPERTY tTimeOutOfCylinder : TIME
```

Property tTimeOutOfCylinder Get Declaration

```
VAR
END_VAR
```

Property tTimeOutOfCylinder Get Implementation

```
tTimeOutOfCylinder := tTimeOutCylinder;
```

Property tTimeOutOfCylinder Set Declaration

```
VAR
END_VAR
```

Property tTimeOutOfCylinder Set Implementation

```
tTimeOutCylinder := tTimeOutOfCylinder;
```

8.6 FB_Subsystem_Root.TcPOU

Declaration

```
// Base class of modules with axis and buttons for manual control
FUNCTION_BLOCK FB_Subsystem_Root
VAR_INPUT
// =============== Axis button variables =====================
bButtonAxisFwIn : BOOL; // Signal of button to move axis forwards
bButtonAxisBwIn : BOOL; // Signal of button to move axis backwards
END_VAR
VAR_OUTPUT
// =============== Error variables ===========================
bAxisError : BOOL; // Error signal of axis
nAxisErrorID : UDINT; // Error ID of axis error

// =============== Axis, button and module return signals ====
bAxisMoves : BOOL; // True if axis is moving
bButtonAxisFwOut : BOOL; // Processed signal for button to move axis forwards
bButtonAxisBwOut : BOOL; // Processed signal for button to move axis backwards
END_VAR
VAR
// =============== Done signals of Halt, Reset and Starting ==
bHaltDone : BOOL; // True if module is stopped
bResetDone : BOOL; // True if module is reset
bStartingDone : BOOL; // True if module is started

// =============== Enables ==================================
bComprAirEnabledLocal : BOOL; // Enable of compressed air
bAxisEnableLocal : BOOL; // Enable of axis
bSensorEnableLocal : BOOL; // Enable of sensor
bManualAxisEnableLocal : BOOL; // Enable of manual axis control
bManualCylinderEnable : BOOL; // Enable of manual cylinder control (not written with input of method Enable)

// =============== State variables ==========================
bHaltRequest : BOOL; // True if machine is in state ABORTING or STOPPING
bResetRequest : BOOL; // True if machine is in state RESETTING

// =============== Axis ====================================
fbAxis : FB_Axis; // Function block instance of FB_Axis
fAxisVelo : LREAL; // Velocity of axis

// =============== Manual axis control ======================
fbButtonAxisFw : FB_SignalHandlingIntern; // To move axis forwards
fbButtonAxisBw : FB_SignalHandlingIntern; // To move axis backwards

// =============== Common variables =========================
fbTriggerStartingStart : R_TRIG; // To diagnose rising edge of starting action starting
fbTriggerResetStart : R_TRIG; // To diagnose rising edge of starting reset
bMoved : BOOL; // History variable to save if axis has moved
END_VAR
```

Implementation

```
// =========================================================
;
// =========================================================
```

Method Enable Declaration

```
METHOD PUBLIC Enable
VAR_INPUT
bComprAirEnabled : BOOL; // Enable of compressed air
bAxisEnable : BOOL; // Enable of axis
bSensorEnable : BOOL; // Enable of sensor
bManualAxisEnable : BOOL; // Enable of manual axis control
END_VAR
```

Method Enable Implementation

```
// =========================================================
// Store inputs in local variables

bComprAirEnabledLocal := bComprAirEnabled;
bAxisEnableLocal := bAxisEnable;
bSensorEnableLocal := bSensorEnable;
bManualAxisEnableLocal := bManualAxisEnable;

// =========================================================
// Enable axis

fbAxis.Enable(bEnable := bAxisEnableLocal);

// =========================================================
// Enable cylinder

bManualCylinderEnable := bManualAxisEnableLocal AND bComprAirEnabledLocal;

// =========================================================
// Control of axis button

fbButtonAxisFw.Enable(bEnable := bManualAxisEnableLocal AND NOT fbButtonAxisBw.bOut);
fbButtonAxisBw.Enable(bEnable := bManualAxisEnableLocal AND NOT fbButtonAxisFw.bOut);

// =========================================================
```

Method Halt Declaration

```
METHOD PUBLIC Halt
VAR_INPUT
bHalt : BOOL; // True if machine is in state ABORTING or STOPPING
END_VAR
```

Method Halt Implementation

```
// =========================================================
// Store request

bHaltRequest := bHalt;

// =========================================================
// Halting axis

fbAxis.Halt(bDriveHalt := bHaltRequest);

// =========================================================
// Getting halt state

bHaltDone := fbAxis.bState_HaltDone;

// =========================================================
```

Method InputOutput Declaration

```
METHOD PUBLIC InputOutput
VAR_INPUT
END_VAR
```

Method InputOutput Implementation

```
// =========================================================
// Axis buttons and output variable for axis

fbButtonAxisFw.SetOutput();
fbButtonAxisBw.SetOutput();

bAxisMoves := fbAxis.bMoves;
bAxisError := fbAxis.bError;
nAxisErrorID := fbAxis.nErrorID;
```

```
fbButtonAxisFw.bInput := bButtonAxisFwIn;
fbButtonAxisBw.bInput := bButtonAxisBwIn;
bButtonAxisFwOut := fbButtonAxisFw.bOut;
bButtonAxisBwOut := fbButtonAxisBw.bOut;

// =========================================================
```

Method Manual Declaration

```
METHOD PUBLIC Manual
VAR_INPUT
END_VAR
```

Method Manual Implementation

```
// =========================================================
// Manual axis control

// Button to move axis backwards pressed
IF fbButtonAxisBw.bOut THEN
IF bHaltDone THEN
bHaltDone := FALSE;
END_IF

IF NOT bMoved THEN
bMoved := TRUE;
END_IF

// Move axis backwards
fbAxis.MoveBw();

// Button to move axis forwards pressed
ELSIF fbButtonAxisFw.bOut THEN
IF bHaltDone THEN
bHaltDone := FALSE;
END_IF

IF NOT bMoved THEN
bMoved := TRUE;
END_IF

// Move axis forwards
fbAxis.MoveFw();

// No button is pressed and halt is not done
ELSIF NOT bHaltDone THEN
// Request halt if axis has moved
IF bMoved THEN
Halt(bHalt := TRUE);
END_IF

// Halt is done but still requested
ELSIF bHaltRequest THEN
Halt(bHalt := FALSE);

bMoved := FALSE;
END_IF

// =========================================================
```

Method Reset Declaration

```
METHOD PUBLIC Reset
VAR_INPUT
bReset : BOOL; // True if machine is in state RESETTING
END_VAR
```

Method Reset Implementation

```
// =========================================================
// Store request

bResetRequest := bReset;

// =========================================================
// Resetting variables

bMoved := FALSE;
bHaltDone := FALSE;
bHaltRequest := FALSE;

// =========================================================
// Resetting axis

fbTriggerResetStart(CLK := bReset);

fbAxis.Reset(bDriveReset := bReset);

bResetDone := fbAxis.bState_ResetDone;

// =========================================================
```

Method ResetState Declaration

```
METHOD PROTECTED ResetState
VAR_INPUT
END_VAR
```

Method ResetState Implementation

```
// =========================================================
// Placeholder method for sub classes
;

// =========================================================
```

Method Starting Declaration

```
METHOD PUBLIC Starting
VAR_INPUT
bStarting : BOOL; // True if machine is in state STARTING
END_VAR
```

Method Starting Implementation

```
// =========================================================

fbTriggerStartingStart(CLK := bStarting);

IF fbTriggerStartingStart.Q THEN
bStartingDone := FALSE;
// Setting velocity of axis
fbAxis.fTargetVelocity := fAxisVelo;
// Resetting state variable
ResetState();
END_IF

IF fbAxis.bState_Enable THEN
bStartingDone := TRUE;
END_IF

// =========================================================
```

Method Stop Declaration

```
METHOD PUBLIC Stop
VAR_INPUT
bHalt : BOOL; // True if machine is in state ABORTING or STOPPING
END_VAR
```

Method Stop Implementation

```
// ========================================================
// Store request

bHaltRequest := bHalt;

// ========================================================
// Stopping axis

fbAxis.Stop(bDriveStop := bHaltRequest);

// ========================================================
// Getting halt state

bHaltDone := fbAxis.bState_HaltDone;

// ========================================================
```

Property bState_HaltDone Declaration

```
PROPERTY bState_HaltDone : BOOL
```

Property bState_HaltDone Get Declaration

```
VAR
END_VAR
```

Property bState_HaltDone Get Implementation

```
bState_HaltDone := bHaltDone;
```

Property bState_ResetDone Declaration

```
PROPERTY bState_ResetDone : BOOL
```

Property bState_ResetDone Get Declaration

```
VAR
END_VAR
```

Property bState_ResetDone Get Implementation

```
bState_ResetDone := bResetDone;
```

Property bState_StartingDone Declaration

```
PROPERTY bState_StartingDone : BOOL
```

Property bState_StartingDone Get Declaration

```
VAR
END_VAR
```

Property bState_StartingDone Get Implementation

```
bState_StartingDone := bStartingDone;
```

Property fAxisVelocity Declaration

```
PROPERTY fAxisVelocity : LREAL
```

Property fAxisVelocity Get Declaration

```
VAR
END_VAR
```

Property fAxisVelocity Get Implementation

```
fAxisVelocity := fAxisVelo;
```

Property fAxisVelocity Set Declaration

```
VAR
END_VAR
```

Property fAxisVelocity Set Implementation

```
fAxisVelo := fAxisVelocity;
```

9 03_Machine

9.1 FB_Machine.TcPOU

Declaration

```
FUNCTION_BLOCK FB_Machine
VAR_INPUT
// =============== Cylinder choice =========================================================
// Separating module - Clamp cylinder
bSeparatingClampDiag : BOOL := TRUE; // If true the clamp cylinder has diagnosis functionality
bSeparatingClampTemp : BOOL := TRUE; // If true the clamp cylinder has temperature functionality
bSeparatingClampRecord : BOOL := FALSE; // If true the clamp cylinder has recording functionality

// Separating module - Barrier cylinder
bSeparatingBarrierDiag : BOOL := FALSE; // If true the barrier cylinder has diagnosis functionality
bSeparatingBarrierTemp : BOOL := TRUE; // If true the barrier cylinder has temperature functionality
bSeparatingBarrierRecord : BOOL := TRUE; // If true the barrier cylinder has recording functionality

// Sorting module for metal boxes - Cylinder
bMetalSortingCylinderDiag : BOOL := TRUE; // If true the cylinder for metal boxes has diagnosis functionality
bMetalSortingCylinderTemp : BOOL := FALSE; // If true the cylinder for metal boxes has temperature functionality
bMetalSortingCylinderRecord : BOOL := FALSE; // If true the cylinder for metal boxes has recording functionality

// Sorting module for plastic boxes - Cylinder
bPlasticSortingCylinderDiag : BOOL := TRUE; // If true the cylinder for plastic boxes has diagnosis
functionality
bPlasticSortingCylinderTemp : BOOL := FALSE; // If true the cylinder for plastic boxes has temperature
functionality
bPlasticSortingCylinderRecord : BOOL := FALSE; // If true the cylinder for plastic boxes has recording
functionality
END_VAR
VAR
// =============== Error ===================================================================
bError : BOOL; // True if modules have errors

// =============== Axis variable ===========================================================
fAxisVelo : LREAL; // Axis velocity
fAxisVeloLastCycle : LREAL; // Axis velocity of last cycle

// =============== State machine variables =================================================
bStarted : BOOL; // True if state machine is in state EXECUTE
bStopped : BOOL; // True if state machine is in state STOPPED
bReset AT %Q* : BOOL; // True if state machine is in state RESETTING
bAborted : BOOL; // True if state machine is in state ABORTED
bIdle : BOOL; // True if state machine is in state IDLE
bAuto : BOOL; // True if state machine is in mode AUTOMATIC
bSemi : BOOL; // True if state machine is in mode SEMIAUTOMATIC
bManual : BOOL; // True if state machine is in mode MANUAL
bMaintenance : BOOL; // True if state machine is in mode MAINTENANCE
bStartButtonOff : BOOL; // To turn off start button

// =============== Machine elements ========================================================
fbSeparateModule : FB_SeparatingModule; // Separating module
fbMetalSorting : FB_SortingModule; // Module that sorts metal boxes
fbPlasticSorting : FB_SortingModule; // Module that sorts plastic boxes
fbComprAir : FB_SignalHandlingIntern; // Compressed air
fbPowerSupply : FB_SignalHandlingIntern; // Power supply
fbVisu : FB_Visu; // Visualization

// =============== Buttons =================================================================
fbButtonStart : FB_SignalHandlingIntern; // Button to start the machine
fbButtonStop : FB_SignalHandlingIntern; // Button to stop the machine
fbButtonReset : FB_SignalHandlingIntern; // Button to reset the machine
fbButtonAbort : FB_SignalHandlingIntern; // Button to abort the machine
fbButtonAuto : FB_SignalHandlingIntern; // Button to change machine mode to AUTOMATIC
fbButtonSemi : FB_SignalHandlingIntern; // Button to change machine mode to SEMI-AUTOMATIC
fbButtonManual : FB_SignalHandlingIntern; // Button to change machine mode to MANUAL
fbButtonMaintenance : FB_SignalHandlingIntern; // Button to change machine mode to MAINTENANCE
fbButtonPower : FB_SignalHandlingIntern; // Button to control power supply
fbButtonComprAir : FB_SignalHandlingIntern; // Button to control compressed air
```

```
// =============== State machine ============================================================
fbStateMachineAuto : PS_PackML_StateMachine_Auto; // State machine for mode AUTOMATIC
fbStateMachineSemiAuto : PS_PackML_StateMachine_SemiAuto; // State machine for mode SEMI-AUTOMATIC
fbStateMachineManual : PS_PackML_StateMachine_Manual; // State machine for mode MANUAL
fbStateMachineMaintenance : PS_PackML_StateMachine_Maintenance; // State machine for mode MAINTENANCE

ePMLUnitMode : E_PMLUnitMode := E_PMLUnitMode.IDLE; // Current mode
ePMLUnitModeRequested : E_PMLUnitMode; // Requested mode
fbUnitModeManager : PS_UnitModeManager; // Mode manager
ePMLState : E_PMLState := E_PMLState.IDLE; // Current state
ePMLStateRequested : E_PMLState; // Requested state

// =============== Common variables =========================================================
bInit : BOOL; // True if module parameters are initialized
bAxisEnable : BOOL; // True if it is allowed to move axis
bManualAxisEnable : BOOL; // True if it is allowed to move manually axis
fbTriggerStart : R_TRIG; // To diagnose rising edge of start button
fbTriggerSeparated : R_TRIG; // To diagnose rising edge of separated box
fbTriggerSelectedM : R_TRIG; // To diagnose rising edge of selected metal box
fbTriggerSelectedP : R_TRIG; // To diagnose rising edge of selected plastic box
nBoxesOnTheWay : INT; // Number of boxes being moved
aClampTemps : ARRAY[1..100] OF LREAL; // Temperature of clamp cylinder
aBarrierTemps : ARRAY[1..100] OF LREAL; // Temperature of barrier cylinder
aMetalTemps : ARRAY[1..100] OF LREAL; // Temperature of metal cylinder
aPlasticTemps : ARRAY[1..100] OF LREAL; // Temperature of plastic cylinder
bStartingIsSetToTrue : BOOL; // True if the starting methods were called with input TRUE
bResetIsSetToTrue : BOOL; // True if the resetting methods were called with input TRUE
bHaltIsSetToTrue : BOOL; // True if the halting methods were called with input TRUE
END_VAR
VAR CONSTANT
// =============== Axis velocities ==========================================================
cAxisVeloAutomatic : LREAL := 100.0; // Axis velocity in (semi) automatic mode
cAxisVeloNotAutomatic : LREAL := 50.0; // Axis velocity in non-automatic mode

// =============== Cylinder parameters for separating module (clamp and barrier cylinder) ===
cSeparatingTimeOutClamp : TIME := T#2.5S; // For clamp cylinder with diagnosis: time in which the cylinder should reach base/work position
cSeparatingRecordIntervalClamp : TIME := T#1S; // Time of intervals to record the temperature of clamp cylinder
cSeparatingTimeOutBarrier : TIME := T#3S; // For barrier cylinder with diagnosis: time in which the cylinder should reach base/work position
cSeparatingRecordIntervalBarrier : TIME := T#1S; // Time of intervals to record the temperature of barrier cylinder
cSeparatingBarrierAtWork : TIME := T#4.5S; // Time value for separating process: barrier cylinder stays at work position

// =============== Cylinder parameters for sorting module ===================================
cSortingTimeOutCylinder : TIME := T#3S; // For sorting cylinder with diagnosis: time in which the cylinder should reach base/work position
cSortingRecordInterval : TIME := T#1S; // Time of intervals to record the temperature of sorting cylinder
cSortingMoveAxis : TIME := T#2.5S; // Time to move sorting axis
cSortingSensorDelay : TIME := T#3S; // Time to delay signal of sorting sensor
END_VAR
```

Implementation

```
// =========================================================
// Getting errors of modules

bError := fbSeparateModule.bAxisError OR fbSeparateModule.bClampError OR fbSeparateModule.bBarrierError
OR fbMetalSorting.bAxisError OR fbMetalSorting.bCylError
OR fbPlasticSorting.bAxisError OR fbPlasticSorting.bCylError;

// =========================================================
// Setting velocities of axes

IF (ePMLUnitMode = E_PMLUnitMode.AUTOMATIC) OR (ePMLUnitMode = E_PMLUnitMode.SEMIAUTOMATIC) THEN
fAxisVelo := cAxisVeloAutomatic;
ELSE
fAxisVelo := cAxisVeloNotAutomatic;
```

```
END_IF

// =========================================================
// Noticing separated and selected boxes

fbTriggerSeparated(CLK := fbSeparateModule.bSeparated);
fbTriggerSelectedM(CLK := fbMetalSorting.bSelected);
fbTriggerSelectedP(CLK := fbPlasticSorting.bSelected);

// =========================================================
// Getting cylinder temperatures of modules

aBarrierTemps := fbSeparateModule.aBarrierTemps;
aClampTemps := fbSeparateModule.aClampTemps;
aMetalTemps := fbMetalSorting.aTemps;
aPlasticTemps := fbPlasticSorting.aTemps;

// =========================================================
// Cylinder options of modules - choose desired cylinders with more or less functionality

fbSeparateModule.CylinderOptions( bClampDiag := bSeparatingClampDiag,
bClampTemp := bSeparatingClampTemp,
bClampRecord := bSeparatingClampRecord,
bBarrierDiag := bSeparatingBarrierDiag,
bBarrierTemp := bSeparatingBarrierTemp,
bBarrierRecord := bSeparatingBarrierRecord);

fbMetalSorting.CylinderOptions( bCylinderDiag := bMetalSortingCylinderDiag,
bCylinderTemp := bMetalSortingCylinderTemp,
bCylinderRecord := bMetalSortingCylinderRecord);

fbPlasticSorting.CylinderOptions( bCylinderDiag := bPlasticSortingCylinderDiag,
bCylinderTemp := bPlasticSortingCylinderTemp,
bCylinderRecord := bPlasticSortingCylinderRecord);

// =========================================================
// Call of actions

// Init
General_Init();

// Mode and state requests
General_Requests();

// Enable for power supply, compressed air control and buttons
General_Enable();

// Input variables of separating and sorting modules
General_ModuleInputs();

// Output variables
General_SetOutput();

// State machine
General_StateMachine();

// Visualization
General_Visu();

// =========================================================
```

Method General_Enable Declaration

```
METHOD PRIVATE General_Enable
VAR_INPUT
END_VAR
```

Method General_Enable Implementation

```
// =========================================================
// Power supply

fbButtonPower.Enable(bEnable := TRUE);
fbPowerSupply.Enable(bEnable := TRUE);

fbPowerSupply.bInput := fbButtonPower.bOut;

// =========================================================
// Axis control

IF (ePMLState = E_PMLState.ABORTED) THEN
bAxisEnable := FALSE;
ELSIF fbPowerSupply.bOut THEN
bAxisEnable := TRUE;
END_IF

bManualAxisEnable := fbPowerSupply.bOut AND (ePMLState = E_PMLState.EXECUTE)
AND ((ePMLUnitMode = E_PMLUnitMode.MANUAL) OR (ePMLUnitMode = E_PMLUnitMode.MAINTENANCE));

// =========================================================
// Compressed air control

IF (ePMLUnitMode <> E_PMLUnitMode.MAINTENANCE) THEN
fbComprAir.bInput := fbPowerSupply.bOut;
ELSE
fbComprAir.bInput := fbButtonComprAir.bOut;
END_IF

fbComprAir.Enable(bEnable := TRUE);
fbButtonComprAir.Enable(bEnable := (ePMLUnitMode = E_PMLUnitMode.MAINTENANCE));

// =========================================================
// Enable of separate module

fbSeparateModule.Enable(bComprAirEnabled := fbComprAir.bOut,
bAxisEnable := bAxisEnable,
bSensorEnable := TRUE,
bManualAxisEnable := bManualAxisEnable);

// =========================================================
// Enable of sorting module for metal boxes

fbMetalSorting.Enable( bComprAirEnabled := fbComprAir.bOut,
bAxisEnable := bAxisEnable,
bSensorEnable := TRUE,
bManualAxisEnable := bManualAxisEnable);

// =========================================================
// Enable of sorting module for plastic boxes

fbPlasticSorting.Enable(bComprAirEnabled := fbComprAir.bOut,
bAxisEnable := bAxisEnable,
bSensorEnable := TRUE,
bManualAxisEnable := bManualAxisEnable);

// =========================================================
// States and modes

// Start button
fbButtonStart.Enable(bEnable := NOT ((ePMLState = E_PMLState.IDLE) AND bError));

// Stop button
fbButtonStop.Enable(bEnable := (ePMLState = E_PMLState.STOPPING) OR (ePMLState = E_PMLState.EXECUTE) OR
(ePMLState = E_PMLState.IDLE));

// Reset button
fbButtonReset.Enable(bEnable := (ePMLState = E_PMLState.STOPPED));

// Abort button
fbButtonAbort.bInput := NOT fbPowerSupply.bOut;
fbButtonAbort.Enable(bEnable := (ePMLState = E_PMLState.EXECUTE) OR (ePMLState = E_PMLState.STOPPING) OR
(ePMLState = E_PMLState.RESETTING));

// Automatic mode button
fbButtonAuto.Enable(bEnable := (ePMLState = E_PMLState.IDLE) AND (ePMLUnitMode <> E_PMLUnitMode.AUTOMATIC));

// Semi-automatic mode button
fbButtonSemi.Enable(bEnable := (ePMLState = E_PMLState.IDLE) AND (ePMLUnitMode <> E_PMLUnitMode.SEMIAUTOMATIC));

// Manual mode button
fbButtonManual.Enable(bEnable := (ePMLState = E_PMLState.IDLE) AND (ePMLUnitMode <> E_PMLUnitMode.MANUAL));

// Maintenance mode button
fbButtonMaintenance.Enable(bEnable := (ePMLState = E_PMLState.IDLE) AND (ePMLUnitMode <>
E_PMLUnitMode.MAINTENANCE));

// =========================================================
```

Method General_Init Declaration

```
METHOD PRIVATE General_Init
VAR_INPUT
END_VAR
```

Method General_Init Implementation

```
// =========================================================

IF NOT bInit THEN

// Time values of separating module
// Clamp cylinder
fbSeparateModule.tTimeOutOfClamp := cSeparatingTimeOutClamp;
fbSeparateModule.tRecordIntervalOfClamp := cSeparatingRecordIntervalClamp;

// Barrier cylinder
fbSeparateModule.tTimeOutOfBarrier := cSeparatingTimeOutBarrier;
fbSeparateModule.tRecordIntervalOfBarrier := cSeparatingRecordIntervalBarrier;

// Separating process
fbSeparateModule.tBarrierAtWorkPos := cSeparatingBarrierAtWork;

// Time values of sorting module (metal boxes)
fbMetalSorting.tTimeOutOfCylinder := cSortingTimeOutCylinder;
fbMetalSorting.tRecordIntervalOfCylinder := cSortingRecordInterval;
fbMetalSorting.tAxisMovement := cSortingMoveAxis;
fbMetalSorting.tDelayOfSensor := cSortingSensorDelay;

// Time values of sorting module (plastic boxes)
fbPlasticSorting.tTimeOutOfCylinder := cSortingTimeOutCylinder;
fbPlasticSorting.tRecordIntervalOfCylinder := cSortingRecordInterval;
fbPlasticSorting.tAxisMovement := cSortingMoveAxis;
fbPlasticSorting.tDelayOfSensor := cSortingSensorDelay;

// Init done
bInit := TRUE;
END_IF

// =========================================================
```

Method General_ModuleInputs Declaration

```
METHOD PRIVATE General_ModuleInputs
VAR_INPUT
END_VAR
```

Method General_ModuleInputs Implementation

```
// ========================================================
// Separating modul

// Sate and button signals
fbSeparateModule.bExecute := (ePMLState = E_PMLState.EXECUTE);
fbSeparateModule.bSemiStart := fbTriggerStart.Q;

// Axis velocity has changed
IF (fAxisVeloLastCycle <> fAxisVelo) THEN
fbSeparateModule.fAxisVelocity := fAxisVelo;
END_IF

// Information about number of boxes being sorted
fbSeparateModule.nBoxesOnTheWay := nBoxesOnTheWay;

// ========================================================
// Sorting modul for metal boxes

// Axis velocity has changed
IF (fAxisVeloLastCycle <> fAxisVelo) THEN
fbMetalSorting.fAxisVelocity := fAxisVelo;
END_IF

// ========================================================
// Sorting modul for plastic boxes

// Axis velocity has changed
IF (fAxisVeloLastCycle <> fAxisVelo) THEN
fbPlasticSorting.fAxisVelocity := fAxisVelo;
END_IF

// ========================================================
/// Remember axis velocity of this cycle for next cycle => call of property AxisVelocity is event triggered

fAxisVeloLastCycle := fAxisVelo;

// ========================================================
```

Method General_Requests Declaration

```
METHOD PRIVATE General_Requests
VAR_INPUT
END_VAR
```

Method General_Requests Implementation

```
// ========================================================
// Check for state requests

fbTriggerStart( CLK := fbButtonStart.bOut);

IF bStartButtonOff AND NOT fbButtonStart.bInput THEN
bStartButtonOff := FALSE;
END_IF

IF fbTriggerStart.Q THEN
bStartButtonOff := TRUE;

IF ePMLState = E_PMLState.ABORTED THEN
ePMLStateRequested := E_PMLState.CLEARING;

ELSIF ePMLState = E_PMLState.STOPPED THEN
ePMLStateRequested := E_PMLState.RESETTING;

ELSIF ePMLState = E_PMLState.IDLE AND fbPowerSupply.bOut THEN
ePMLStateRequested := E_PMLState.STARTING;
END_IF

ELSIF fbButtonStop.bOut THEN
ePMLStateRequested := E_PMLState.STOPPING;
```

```
ELSIF fbButtonReset.bOut THEN
ePMLStateRequested := E_PMLState.RESETTING;
END_IF

// ==========================================================
// Check for mode requests

IF fbButtonManual.bOut THEN
ePMLUnitModeRequested := E_PMLUnitMode.MANUAL;

ELSIF fbButtonAuto.bOut THEN
ePMLUnitModeRequested := E_PMLUnitMode.AUTOMATIC;

ELSIF fbButtonSemi.bOut THEN
ePMLUnitModeRequested := E_PMLUnitMode.SEMIAUTOMATIC;

ELSIF fbButtonMaintenance.bOut THEN
ePMLUnitModeRequested := E_PMLUnitMode.MAINTENANCE;
END_IF

IF fbButtonManual.bOut OR fbButtonAuto.bOut OR fbButtonSemi.bOut OR fbButtonMaintenance.bOut THEN
fbUnitModeManager( Execute := TRUE,
eModeCommand := ePMLUnitModeRequested,
ePMLState := ePMLState);
END_IF

IF fbUnitModeManager.Done THEN
ePMLUnitMode := fbUnitModeManager.eModeStatus;

fbUnitModeManager(Execute := FALSE);
END_IF

// ==========================================================
```

Method General_SetOutput Declaration

```
METHOD PRIVATE General_SetOutput
VAR_INPUT
END_VAR
```

Method General_SetOutput Implementation

```
// ==========================================================
// State variables

bStopped := (ePMLState = E_PMLState.STOPPED);
bStarted := (ePMLState = E_PMLState.EXECUTE);
bReset := (ePMLState = E_PMLState.RESETTING);
bAborted := (ePMLState = E_PMLState.ABORTED);
bIdle := (ePMLState = E_PMLState.IDLE AND fbPowerSupply.bOut);

// ==========================================================
// Mode variables

bAuto := (ePMLUnitMode = E_PMLUnitMode.AUTOMATIC);
bSemi := (ePMLUnitMode = E_PMLUnitMode.SEMIAUTOMATIC);
bManual := (ePMLUnitMode = E_PMLUnitMode.MANUAL);
bMaintenance := (ePMLUnitMode = E_PMLUnitMode.MAINTENANCE);

// ==========================================================
// Power supply and compressed air control

fbPowerSupply.SetOutput();
fbComprAir.SetOutput();

// ==========================================================
// Modules

fbSeparateModule.InputOutput();
```

```
fbMetalSorting.InputOutput();
fbPlasticSorting.InputOutput();

// =========================================================
// Buttons

// Power supply and compressed air control
fbButtonPower.SetOutput();
fbButtonComprAir.SetOutput();

// States
fbButtonStart.SetOutput();
fbButtonStop.SetOutput();
fbButtonReset.SetOutput();
fbButtonAbort.SetOutput();

// Modes
fbButtonAuto.SetOutput();
fbButtonSemi.SetOutput();
fbButtonManual.SetOutput();
fbButtonMaintenance.SetOutput();

// =========================================================
```

Method General_StateMachine Declaration

```
METHOD PRIVATE General_StateMachine
VAR_INPUT
END_VAR
```

Method General_StateMachine Implementation

```
// =========================================================
// Mode and state handling

CASE ePMLUnitMode OF

// =========================================================
E_PMLUnitMode.AUTOMATIC:

fbStateMachineAuto( Start := ePMLStateRequested = E_PMLState.STARTING,
Stop := fbButtonStop.bOut,
Reset := fbButtonReset.bOut OR ePMLStateRequested = E_PMLState.RESETTING,
Clear := ePMLStateRequested = E_PMLState.CLEARING,
Abort := fbButtonAbort.bOut OR (bError AND ePMLState = E_PMLState.EXECUTE));

ePMLState := fbStateMachineAuto.ePMLState;

Operating_Automatic();

// =========================================================
E_PMLUnitMode.MANUAL:

fbStateMachineManual( Start := ePMLStateRequested = E_PMLState.STARTING,
Stop := fbButtonStop.bOut,
Reset := fbButtonReset.bOut OR ePMLStateRequested = E_PMLState.RESETTING,
Clear := ePMLStateRequested = E_PMLState.CLEARING,
Abort := fbButtonAbort.bOut OR (bError AND ePMLState = E_PMLState.EXECUTE));

ePMLState := fbStateMachineManual.ePMLState;

Operating_Manual();

// =========================================================
E_PMLUnitMode.SEMIAUTOMATIC:

fbStateMachineSemiAuto( Start := ePMLStateRequested = E_PMLState.STARTING,
Stop := fbButtonStop.bOut,
Reset := fbButtonReset.bOut OR ePMLStateRequested = E_PMLState.RESETTING,
Clear := ePMLStateRequested = E_PMLState.CLEARING,
Abort := fbButtonAbort.bOut OR (bError AND ePMLState = E_PMLState.EXECUTE));

ePMLState := fbStateMachineSemiAuto.ePMLState;

Operating_Semiautomatic();

// =========================================================
E_PMLUnitMode.MAINTENANCE:

fbStateMachineMaintenance( Start := ePMLStateRequested = E_PMLState.STARTING,
Stop := fbButtonStop.bOut,
Reset := fbButtonReset.bOut OR ePMLStateRequested = E_PMLState.RESETTING,
Clear := ePMLStateRequested = E_PMLState.CLEARING,
Abort := fbButtonAbort.bOut OR (bError AND ePMLState = E_PMLState.EXECUTE));

ePMLState := fbStateMachineMaintenance.ePMLState;

Operating_Maintenance();
END_CASE

// =========================================================
```

Method General_Visu Declaration

```
METHOD PRIVATE General_Visu
VAR_INPUT
END_VAR
```

Method General_Visu Implementation

```
// =========================================================
// Calling visu function block

fbVisu( bError := bError,
fAxisVelo := fAxisVelo,
bComprAirEnabled := fbComprAir.bOut,
bPowerEnabled := fbPowerSupply.bOut,
bMainAxisMoves := fbSeparateModule.bAxisMoves,
bMetalAxisMoves := fbMetalSorting.bAxisMoves,
bPlasticAxisMoves := fbPlasticSorting.bAxisMoves,
bClampToWork := fbSeparateModule.bClampToWork,
bBarrierToWork := fbSeparateModule.bBarrierToWork,
bMetalToWork := fbMetalSorting.bCylinderToWork,
bPlasticToWork := fbPlasticSorting.bCylinderToWork,
bClampError := fbSeparateModule.bClampError,
sClampErrMsg := fbSeparateModule.sClampErrorMsg,
bMetalCylError := fbMetalSorting.bCylError,
sMetalErrMsg := fbMetalSorting.sCylErrorMsg,
bPlasticCylError := fbPlasticSorting.bCylError,
sPlasticErrMsg := fbPlasticSorting.sCylErrorMsg,
nMainAxisErrorId := fbSeparateModule.fbAxis.nErrorID,
nMetalAxisErrorId := fbMetalSorting.fbAxis.nErrorID,
nPlasticAxisErrorId := fbPlasticSorting.fbAxis.nErrorID,
bStarted := bStarted,
bStopped := bStopped,
bReset := bReset,
bAborted := bAborted,
bIdle := bIdle,
bAuto := bAuto,
bSemi := bSemi,
bManual := bManual,
bMaintenance := bMaintenance,
bStartButtonOff := bStartButtonOff,
bButtonPowerIn := fbButtonPower.bOut,
bButtonComprAirIn := fbButtonComprAir.bOut,
bButtonStartIn := fbButtonStart.bOut,
bButtonStopIn := fbButtonStop.bOut,
bButtonResetIn := fbButtonReset.bOut,
bButtonAbortIn := fbButtonAbort.bOut,
bButtonAutoIn := fbButtonAuto.bOut,
bButtonSemiIn := fbButtonSemi.bOut,
bButtonManuIn := fbButtonManual.bOut,
bButtonMaintenanceIn := fbButtonMaintenance.bOut,
bButtonMainFwIn := fbSeparateModule.bButtonAxisFwOut,
bButtonMainBwIn := fbSeparateModule.bButtonAxisBwOut,
bButtonMetalFwIn := fbMetalSorting.bButtonAxisFwOut,
bButtonMetalBwIn := fbMetalSorting.bButtonAxisBwOut,
bButtonPlasticFwIn := fbPlasticSorting.bButtonAxisFwOut,
bButtonPlasticBwIn := fbPlasticSorting.bButtonAxisBwOut,
bButtonClampToWorkIn := fbSeparateModule.bButtonClampToWorkOut,
bButtonBarrierToWorkIn := fbSeparateModule.bButtonBarrierToWorkOut,
bButtonMetalToWorkIn := fbMetalSorting.bButtonCylToWorkOut,
bButtonPlasticToWorkIn := fbPlasticSorting.bButtonCylToWorkOut,

bButtonPowerOut => fbButtonPower.bInput,
bButtonComprAirOut => fbButtonComprAir.bInput,
bButtonStartOut => fbButtonStart.bInput,
bButtonStopOut => fbButtonStop.bInput,
bButtonResetOut => fbButtonReset.bInput,
bButtonAutoOut => fbButtonAuto.bInput,
bButtonSemiOut => fbButtonSemi.bInput,
bButtonManuOut => fbButtonManual.bInput,
bButtonMaintenanceOut => fbButtonMaintenance.bInput,
bButtonMainFwOut => fbSeparateModule.bButtonAxisFwIn,
bButtonMainBwOut => fbSeparateModule.bButtonAxisBwIn,
bButtonMetalFwOut => fbMetalSorting.bButtonAxisFwIn,
bButtonMetalBwOut => fbMetalSorting.bButtonAxisBwIn,
bButtonPlasticFwOut => fbPlasticSorting.bButtonAxisFwIn,
bButtonPlasticBwOut => fbPlasticSorting.bButtonAxisBwIn,
bButtonClampToWorkOut => fbSeparateModule.bButtonClampToWorkIn,
bButtonBarrierToWorkOut => fbSeparateModule.bButtonBarrierToWorkIn,
bButtonMetalToWorkOut => fbMetalSorting.bButtonCylToWorkIn,
bButtonPlasticToWorkOut => fbPlasticSorting.bButtonCylToWorkIn);

// =========================================================
```

Method Operating_Automatic Declaration

```
METHOD PRIVATE Operating_Automatic
VAR_INPUT
END_VAR
```

Method Operating_Automatic Implementation

```
// =========================================================

CASE ePMLState OF

// =========================================================
// Turning axis reset off, passing it to axis instances
E_PMLState.IDLE:

IF bResetIsSetToTrue THEN
bResetIsSetToTrue := FALSE;

fbSeparateModule.Reset(bReset := FALSE);
fbMetalSorting.Reset(bReset := FALSE);
fbPlasticSorting.Reset(bReset := FALSE);

fbStateMachineAuto.StateComplete := FALSE;
END_IF

// =========================================================
// Resetting variables for executing automatic mode, checking status of axes
E_PMLState.STARTING:

IF NOT bStartingIsSetToTrue THEN
bStartingIsSetToTrue := TRUE;

fbSeparateModule.Starting(bStarting := TRUE);
```

```
fbMetalSorting.Starting(bStarting := TRUE);
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbSeparateModule.bState_StartingDone THEN
fbSeparateModule.Starting(bStarting := TRUE);
END_IF

IF NOT fbMetalSorting.bState_StartingDone THEN
fbMetalSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_StartingDone THEN
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF fbSeparateModule.bState_StartingDone AND fbMetalSorting.bState_StartingDone AND
fbPlasticSorting.bState_StartingDone THEN
fbStateMachineAuto.StateComplete := TRUE;
END_IF

// =========================================================
// Executing automatic mode
E_PMLState.EXECUTE:

IF bStartingIsSetToTrue THEN
bStartingIsSetToTrue := FALSE;

fbSeparateModule.Starting(bStarting := FALSE);
fbMetalSorting.Starting(bStarting := FALSE);
fbPlasticSorting.Starting(bStarting := FALSE);

fbStateMachineAuto.StateComplete := FALSE;
END_IF

fbSeparateModule.Automatic();
fbMetalSorting.Automatic();
fbPlasticSorting.Automatic();

IF fbTriggerSeparated.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay + 1;
END_IF

IF fbTriggerSelectedM.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

IF fbTriggerSelectedP.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

// =========================================================
// Stopping axes
E_PMLState.ABORTING:

IF NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF

IF NOT fbMetalSorting.bState_HaltDone THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF
```

```
IF NOT fbPlasticSorting.bState_HaltDone THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone AND fbPlasticSorting.bState_HaltDone THEN
fbStateMachineAuto.StateComplete := TRUE;
END_IF

// ========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.ABORTED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineAuto.StateComplete THEN
fbStateMachineAuto.StateComplete := FALSE;
END_IF

// ========================================================
E_PMLState.CLEARING:

fbStateMachineAuto.StateComplete := TRUE;

// ========================================================
// Selecting boxes on conveyor belt to destination, then stopping axes
E_PMLState.STOPPING:

IF fbTriggerSeparated.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay + 1;
END_IF

IF fbTriggerSelectedM.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

IF fbTriggerSelectedP.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

IF (nBoxesOnTheWay = 0) AND NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone OR (nBoxesOnTheWay <> 0) THEN
fbSeparateModule.Automatic();

IF (nBoxesOnTheWay = 0) THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF
END_IF

IF NOT fbMetalSorting.bState_HaltDone OR (nBoxesOnTheWay <> 0) THEN
fbMetalSorting.Automatic();

IF (nBoxesOnTheWay = 0) THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF
END_IF

IF NOT fbPlasticSorting.bState_HaltDone OR (nBoxesOnTheWay <> 0) THEN
fbPlasticSorting.Automatic();
```

```
IF (nBoxesOnTheWay = 0) THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone
AND fbPlasticSorting.bState_HaltDone AND (nBoxesOnTheWay = 0) THEN
fbStateMachineAuto.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.STOPPED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineAuto.StateComplete THEN
fbStateMachineAuto.StateComplete := FALSE;
END_IF

// =========================================================
// Resetting cylinders and axes
E_PMLState.RESETTING:

IF NOT bResetIsSetToTrue THEN
bResetIsSetToTrue := TRUE;

fbSeparateModule.Reset(bReset := TRUE);
fbMetalSorting.Reset(bReset := TRUE);
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbSeparateModule.bState_ResetDone THEN
fbSeparateModule.Reset(bReset := TRUE);
END_IF

IF NOT fbMetalSorting.bState_ResetDone THEN
fbMetalSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_ResetDone THEN
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF fbSeparateModule.bState_ResetDone AND fbMetalSorting.bState_ResetDone AND fbPlasticSorting.bState_ResetDone
AND fbSeparateModule.bState_ClampAtBasePos AND fbSeparateModule.bState_BarrierAtBasePos
AND fbMetalSorting.bState_CylinderAtBasePos AND fbPlasticSorting.bState_CylinderAtBasePos THEN
nBoxesOnTheWay := 0;
fbStateMachineAuto.StateComplete := TRUE;
END_IF
END_CASE

// =========================================================
```

Method Operating_Maintenance Declaration

```
METHOD PRIVATE Operating_Maintenance
VAR_INPUT
END_VAR
```

Method Operating_Maintenance Implementation

```
// =========================================================
CASE ePMLState OF

// =========================================================
// Turning axis reset off, passing it to axis instances
E_PMLState.IDLE:

IF bResetIsSetToTrue THEN
bResetIsSetToTrue := FALSE;

fbSeparateModule.Reset(bReset := FALSE);
fbMetalSorting.Reset(bReset := FALSE);
fbPlasticSorting.Reset(bReset := FALSE);

fbStateMachineMaintenance.StateComplete := FALSE;
END_IF

// =========================================================
// Checking status of axes
E_PMLState.STARTING:

IF NOT bStartingIsSetToTrue THEN
bStartingIsSetToTrue := TRUE;

fbSeparateModule.Starting(bStarting := TRUE);
fbMetalSorting.Starting(bStarting := TRUE);
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbSeparateModule.bState_StartingDone THEN
fbSeparateModule.Starting(bStarting := TRUE);
END_IF

IF NOT fbMetalSorting.bState_StartingDone THEN
fbMetalSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_StartingDone THEN
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF fbSeparateModule.bState_StartingDone AND fbMetalSorting.bState_StartingDone AND
fbPlasticSorting.bState_StartingDone THEN
fbStateMachineMaintenance.StateComplete := TRUE;
END_IF

// =========================================================
// Executing maintenance mode
E_PMLState.EXECUTE:

IF bStartingIsSetToTrue THEN
bStartingIsSetToTrue := FALSE;

fbSeparateModule.Starting(bStarting := FALSE);
fbMetalSorting.Starting(bStarting := FALSE);
fbPlasticSorting.Starting(bStarting := FALSE);

fbStateMachineMaintenance.StateComplete := FALSE;
END_IF

fbSeparateModule.Maintenance();
fbMetalSorting.Maintenance();
fbPlasticSorting.Maintenance();

// =========================================================
// Stopping axes
E_PMLState.ABORTING:

IF NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;
```

```
fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF

IF NOT fbMetalSorting.bState_HaltDone THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_HaltDone THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone AND fbPlasticSorting.bState_HaltDone THEN
fbStateMachineMaintenance.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.ABORTED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineMaintenance.StateComplete THEN
fbStateMachineMaintenance.StateComplete := FALSE;
END_IF

// =========================================================
E_PMLState.CLEARING:

fbStateMachineMaintenance.StateComplete := TRUE;

// =========================================================
// Stopping axes
E_PMLState.STOPPING:

IF NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF

IF NOT fbMetalSorting.bState_HaltDone THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_HaltDone THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone AND fbPlasticSorting.bState_HaltDone THEN
fbStateMachineMaintenance.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.STOPPED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineMaintenance.StateComplete THEN
fbStateMachineMaintenance.StateComplete := FALSE;
END_IF

// ==========================================================
// Resetting cylinders and axes
E_PMLState.RESETTING:

IF NOT bResetIsSetToTrue THEN
bResetIsSetToTrue := TRUE;

fbSeparateModule.Reset(bReset := TRUE);
fbMetalSorting.Reset(bReset := TRUE);
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbSeparateModule.bState_ResetDone THEN
fbSeparateModule.Reset(bReset := TRUE);
END_IF

IF NOT fbMetalSorting.bState_ResetDone THEN
fbMetalSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_ResetDone THEN
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF fbSeparateModule.bState_ResetDone AND fbMetalSorting.bState_ResetDone AND fbPlasticSorting.bState_ResetDone
AND fbSeparateModule.bState_ClampAtBasePos AND fbSeparateModule.bState_BarrierAtBasePos
AND fbMetalSorting.bState_CylinderAtBasePos AND fbPlasticSorting.bState_CylinderAtBasePos THEN
fbStateMachineMaintenance.StateComplete := TRUE;
END_IF
END_CASE

// ==========================================================
```

Method Operating_Manual Declaration

```
METHOD PRIVATE Operating_Manual
VAR_INPUT
END_VAR
```

Method Operating_Manual Implementation

```
// ==========================================================

CASE ePMLState OF

// ==========================================================
// Turning axis reset off, passing it to axis instances
E_PMLState.IDLE:

IF bResetIsSetToTrue THEN
bResetIsSetToTrue := FALSE;

fbSeparateModule.Reset(bReset := FALSE);
fbMetalSorting.Reset(bReset := FALSE);
fbPlasticSorting.Reset(bReset := FALSE);
```

```
fbStateMachineManual.StateComplete := FALSE;
END_IF

// ========================================================
// Checking status of axes
E_PMLState.STARTING:

IF NOT bStartingIsSetToTrue THEN
bStartingIsSetToTrue := TRUE;

fbSeparateModule.Starting(bStarting := TRUE);
fbMetalSorting.Starting(bStarting := TRUE);
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbSeparateModule.bState_StartingDone THEN
fbSeparateModule.Starting(bStarting := TRUE);
END_IF

IF NOT fbMetalSorting.bState_StartingDone THEN
fbMetalSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_StartingDone THEN
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF fbSeparateModule.bState_StartingDone AND fbMetalSorting.bState_StartingDone AND
fbPlasticSorting.bState_StartingDone THEN
fbStateMachineManual.StateComplete := TRUE;
END_IF

// ========================================================
// Executing manual mode
E_PMLState.EXECUTE:

IF bStartingIsSetToTrue THEN
bStartingIsSetToTrue := FALSE;

fbSeparateModule.Starting(bStarting := FALSE);
fbMetalSorting.Starting(bStarting := FALSE);
fbPlasticSorting.Starting(bStarting := FALSE);

fbStateMachineManual.StateComplete := FALSE;
END_IF

fbSeparateModule.Manual();
fbMetalSorting.Manual();
fbPlasticSorting.Manual();

// ========================================================
// Stopping axes
E_PMLState.ABORTING:

IF NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF

IF NOT fbMetalSorting.bState_HaltDone THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF
```

```
IF NOT fbPlasticSorting.bState_HaltDone THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone AND fbPlasticSorting.bState_HaltDone THEN
fbStateMachineManual.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.ABORTED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineManual.StateComplete THEN
fbStateMachineManual.StateComplete := FALSE;
END_IF

// =========================================================
E_PMLState.CLEARING:

fbStateMachineManual.StateComplete := TRUE;

// =========================================================
// Stopping axes
E_PMLState.STOPPING:

IF NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF

IF NOT fbMetalSorting.bState_HaltDone THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_HaltDone THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone AND fbPlasticSorting.bState_HaltDone THEN
fbStateMachineManual.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.STOPPED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineManual.StateComplete THEN
fbStateMachineManual.StateComplete := FALSE;
```

```
END_IF

// =========================================================
// Resetting cylinders and axes
E_PMLState.RESETTING:

IF NOT bResetIsSetToTrue THEN
bResetIsSetToTrue := TRUE;

fbSeparateModule.Reset(bReset := TRUE);
fbMetalSorting.Reset(bReset := TRUE);
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbSeparateModule.bState_ResetDone THEN
fbSeparateModule.Reset(bReset := TRUE);
END_IF

IF NOT fbMetalSorting.bState_ResetDone THEN
fbMetalSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_ResetDone THEN
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF fbSeparateModule.bState_ResetDone AND fbMetalSorting.bState_ResetDone AND fbPlasticSorting.bState_ResetDone
AND fbSeparateModule.bState_ClampAtBasePos AND fbSeparateModule.bState_BarrierAtBasePos
AND fbMetalSorting.bState_CylinderAtBasePos AND fbPlasticSorting.bState_CylinderAtBasePos THEN
fbStateMachineManual.StateComplete := TRUE;
END_IF
END_CASE

// =========================================================
```

Method Operating_Semiautomatic Declaration

```
METHOD PRIVATE Operating_Semiautomatic
VAR_INPUT
END_VAR
```

Method Operating_Semiautomatic Implementation

```
// =========================================================

CASE ePMLState OF

// =========================================================
// Turning axis reset off, passing it to axis instances
E_PMLState.IDLE:

IF bResetIsSetToTrue THEN
bResetIsSetToTrue := FALSE;

fbSeparateModule.Reset(bReset := FALSE);
fbMetalSorting.Reset(bReset := FALSE);
fbPlasticSorting.Reset(bReset := FALSE);

fbStateMachineSemiAuto.StateComplete := FALSE;
END_IF

// =========================================================
// Resetting variables for executing automatic mode, checking status of axes
E_PMLState.STARTING:

IF NOT bStartingIsSetToTrue THEN
bStartingIsSetToTrue := TRUE;

fbSeparateModule.Starting(bStarting := TRUE);
fbMetalSorting.Starting(bStarting := TRUE);
```

```
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbSeparateModule.bState_StartingDone THEN
fbSeparateModule.Starting(bStarting := TRUE);
END_IF

IF NOT fbMetalSorting.bState_StartingDone THEN
fbMetalSorting.Starting(bStarting := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_StartingDone THEN
fbPlasticSorting.Starting(bStarting := TRUE);
END_IF

IF fbSeparateModule.bState_StartingDone AND fbMetalSorting.bState_StartingDone AND
fbPlasticSorting.bState_StartingDone THEN
fbStateMachineSemiAuto.StateComplete := TRUE;
END_IF

// ==========================================================
// Executing semi-automatic mode
E_PMLState.EXECUTE:

IF bStartingIsSetToTrue THEN
bStartingIsSetToTrue := FALSE;

fbSeparateModule.Starting(bStarting := FALSE);
fbMetalSorting.Starting(bStarting := FALSE);
fbPlasticSorting.Starting(bStarting := FALSE);

fbStateMachineSemiAuto.StateComplete := FALSE;
END_IF

fbSeparateModule.Semiautomatic();
fbMetalSorting.Semiautomatic();
fbPlasticSorting.Semiautomatic();

IF fbTriggerSeparated.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay + 1;
END_IF

IF fbTriggerSelectedM.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

IF fbTriggerSelectedP.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

// ==========================================================
// Stopping axes
E_PMLState.ABORTING:

IF NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF

IF NOT fbMetalSorting.bState_HaltDone THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_HaltDone THEN
```

```
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone AND fbPlasticSorting.bState_HaltDone THEN
fbStateMachineSemiAuto.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.ABORTED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineSemiAuto.StateComplete THEN
fbStateMachineSemiAuto.StateComplete := FALSE;
END_IF

// =========================================================
E_PMLState.CLEARING:

fbStateMachineSemiAuto.StateComplete := TRUE;

// =========================================================
// Selecting boxes on conveyor belt to destination, then stopping axes
E_PMLState.STOPPING:

IF fbTriggerSeparated.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay + 1;
END_IF

IF fbTriggerSelectedM.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

IF fbTriggerSelectedP.Q THEN
nBoxesOnTheWay := nBoxesOnTheWay - 1;
END_IF

IF (nBoxesOnTheWay = 0) AND NOT bHaltIsSetToTrue THEN
bHaltIsSetToTrue := TRUE;

fbSeparateModule.Stop(bHalt := TRUE);
fbMetalSorting.Stop(bHalt := TRUE);
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF

IF NOT fbSeparateModule.bState_HaltDone OR (nBoxesOnTheWay <> 0) THEN
fbSeparateModule.Semiautomatic();

IF (nBoxesOnTheWay = 0) THEN
fbSeparateModule.Stop(bHalt := TRUE);
END_IF
END_IF

IF NOT fbMetalSorting.bState_HaltDone OR (nBoxesOnTheWay <> 0) THEN
fbMetalSorting.Semiautomatic();

IF (nBoxesOnTheWay = 0) THEN
fbMetalSorting.Stop(bHalt := TRUE);
END_IF
END_IF

IF NOT fbPlasticSorting.bState_HaltDone OR (nBoxesOnTheWay <> 0) THEN
fbPlasticSorting.Semiautomatic();
```

```
IF (nBoxesOnTheWay = 0) THEN
fbPlasticSorting.Stop(bHalt := TRUE);
END_IF
END_IF

IF fbSeparateModule.bState_HaltDone AND fbMetalSorting.bState_HaltDone
AND fbPlasticSorting.bState_HaltDone AND (nBoxesOnTheWay = 0) THEN
fbStateMachineSemiAuto.StateComplete := TRUE;
END_IF

// =========================================================
// Turning off axis stop, passing it to axis instances
E_PMLState.STOPPED:

IF bHaltIsSetToTrue THEN
bHaltIsSetToTrue := FALSE;

fbSeparateModule.Stop(bHalt := FALSE);
fbMetalSorting.Stop(bHalt := FALSE);
fbPlasticSorting.Stop(bHalt := FALSE);
END_IF

IF fbStateMachineSemiAuto.StateComplete THEN
fbStateMachineSemiAuto.StateComplete := FALSE;
END_IF

// =========================================================
// Resetting cylinders and axes
E_PMLState.RESETTING:

IF NOT bResetIsSetToTrue THEN
bResetIsSetToTrue := TRUE;

fbSeparateModule.Reset(bReset := TRUE);
fbMetalSorting.Reset(bReset := TRUE);
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbSeparateModule.bState_ResetDone THEN
fbSeparateModule.Reset(bReset := TRUE);
END_IF

IF NOT fbMetalSorting.bState_ResetDone THEN
fbMetalSorting.Reset(bReset := TRUE);
END_IF

IF NOT fbPlasticSorting.bState_ResetDone THEN
fbPlasticSorting.Reset(bReset := TRUE);
END_IF

IF fbSeparateModule.bState_ResetDone AND fbMetalSorting.bState_ResetDone AND fbPlasticSorting.bState_ResetDone
AND fbSeparateModule.bState_ClampAtBasePos AND fbSeparateModule.bState_BarrierAtBasePos
AND fbMetalSorting.bState_CylinderAtBasePos AND fbPlasticSorting.bState_CylinderAtBasePos THEN
nBoxesOnTheWay := 0;
fbStateMachineSemiAuto.StateComplete := TRUE;
END_IF
END_CASE

// =========================================================
```

10 04_Application

10.1 MAIN.TcPOU

Declaration

```
(* ============================================================================================================
===========================
OOP: Extended sample
===============================================================================================================
========================
Description:
This PLC sample contains an object-oriented program to control a sorting system.
The application can be controlled via an integrated visualization.

PLC_SortingSystem_PLC:
The functionality of the sorting system is implemented within this PLC project (PLC_SortingSystem_PLC).
It also contains the visualization to control the application.

PLC_SortingSystem_Simu:
To run the project without hardware, the simulation PLC can be used.
For example, It simulates the cylinder positions and temperatures, as well as the sensor signals (correspondent
to the box positions).
===============================================================================================================
========================
InfoSys:
https://infosys.beckhoff.de/content/1033/tc3_sampleprogram1/45035996374683915.html?id=6126826020926170206
https://infosys.beckhoff.de/content/1033/tc3_plc_intro/72057596565231755.html?id=3937754078542861062
===============================================================================================================
======================== *)
PROGRAM MAIN
VAR
fbMachine : FB_Machine;
END_VAR
```

Implementation

```
// =========================================================

fbMachine();

// =========================================================
```

11 05_Visu

11.1 FB_Visu.TcPOU

Declaration

```
FUNCTION_BLOCK FB_Visu
VAR_INPUT
// =============== Sensors to detect boxes ==================
bBarrierSensor AT %I* : BOOL; // True if barrier sensor detects a box
bMetalSensor AT %I* : BOOL; // True if metal sensor detects a metal box
bPlasticSensor AT %I* : BOOL; // True if plastic sensor detects a box

// =============== Axis ====================================
fAxisVelo : LREAL; // Axis velocity

// =============== State of machine elements ================
bComprAirEnabled : BOOL; // True if compressed air is enabled
bPowerEnabled : BOOL; // True if power supply is enabled
bMainAxisMoves : BOOL; // True if main axis is moving
bMetalAxisMoves : BOOL; // True if metal axis is moving
bPlasticAxisMoves : BOOL; // True if plastic axis is moving
bClampToWork : BOOL; // True if clamp cylinder moves to work position
bBarrierToWork : BOOL; // True if barrier cylinder moves to work position
bMetalToWork : BOOL; // True if metal cylinder moves to work position
bPlasticToWork : BOOL; // True if plastic cylinder moves to work position

// =============== Error ===================================
bError : BOOL; // True if modules have errors
bClampError : BOOL; // True if clamp cylinder has an error
sClampErrMsg : STRING; // Error message of clamp cylinder
bMetalCylError : BOOL; // True if metal cylinder has an error
sMetalErrMsg : STRING; // Error message of metal cylinder
bPlasticCylError : BOOL; // True if plastic cylinder has an error
sPlasticErrMsg : STRING; // Error message of plastic cylinder
nMainAxisErrorId : UDINT; // Error id of main axis
nMetalAxisErrorId : UDINT; // Error id of metal axis
nPlasticAxisErrorId : UDINT; // Error id of plastic axis

// =============== PS_PackML_StateMachine ===================
bStarted : BOOL; // True if state machine is in state EXECUTE
bStopped : BOOL; // True if state machine is in state STOPPED
bReset : BOOL; // True if state machine is in state RESETTING
bAborted : BOOL; // True if state machine is in state ABORTED
bIdle : BOOL; // True if state machine is in state IDLE
bAuto : BOOL; // True if state machine is in mode AUTOMATIC
bSemi : BOOL; // True if state machine is in mode SEMIAUTOMATIC
bManual : BOOL; // True if state machine is in mode MANUAL
bMaintenance : BOOL; // True if state machine is in mode MAINTENANCE
bStartButtonOff : BOOL; // To turn off start button

// =============== Buttons of visualization - inputs ========
// State buttons
bButtonPowerIn : BOOL; // Input signal of power button
bButtonComprAirIn : BOOL; // Input signal of compressed air button
bButtonStartIn : BOOL; // Input signal of start button
bButtonStopIn : BOOL; // Input signal of stop button
bButtonResetIn : BOOL; // Input signal of reset button
bButtonAbortIn : BOOL; // Input signal of abort button

// Mode buttons
bButtonAutoIn : BOOL; // Input signal of automatic button
bButtonSemiIn : BOOL; // Input signal of semi-automatic button
bButtonManuIn : BOOL; // Input signal of manual button
bButtonMaintenanceIn : BOOL; // Input signal of maintenance button

// Axis buttons
bButtonMainFwIn : BOOL; // Input signal of button to move main axis forwards
bButtonMainBwIn : BOOL; // Input signal of button to move main axis backwards
bButtonMetalFwIn : BOOL; // Input signal of button to move metal axis forwards
bButtonMetalBwIn : BOOL; // Input signal of button to move metal axis backwards
```

```
bButtonPlasticFwIn : BOOL; // Input signal of button to move plastic axis forwards
bButtonPlasticBwIn : BOOL; // Input signal of button to move plastic axis backwards

// Cylinder buttons
bButtonClampToWorkIn : BOOL; // Input signal of button to move clamp cylinder
bButtonBarrierToWorkIn : BOOL; // Input signal of button to move barrier cylinder
bButtonMetalToWorkIn : BOOL; // Input signal of button to move metal cylinder
bButtonPlasticToWorkIn : BOOL; // Input signal of button to move plastic cylinder
END_VAR
VAR_OUTPUT
fYClamp AT %Q* : LREAL; // Y-Coordinate of clamp cylinder
fYBarrier AT %Q* : LREAL; // Y-Coordinate of barrier cylinder
fYMetal AT %Q* : LREAL; // Y-Coordinate of metal cylinder
fYPlastic AT %Q* : LREAL; // Y-Coordinate of plastic cylinder
aBoxX AT %Q* : ARRAY[1..4] OF LREAL; // X-Coordinates of boxes
aBoxY AT %Q* : ARRAY[1..4] OF LREAL; // Y-Coordinates of boxes

// =============== Buttons of visualization - outputs =======
// State buttons
bButtonPowerOut : BOOL; // To turn on and off the machine
bButtonComprAirOut : BOOL; // To enable compressed air
bButtonStartOut : BOOL; // To change state into state START
bButtonStopOut : BOOL; // To change state into state STOP
bButtonResetOut : BOOL; // To change state into state RESET

// Mode buttons
bButtonAutoOut : BOOL; // To change mode into mode AUTOMATIC
bButtonSemiOut : BOOL; // To change mode into mode SEMI-AUTOMATIC
bButtonManuOut : BOOL; // To change mode into mode MANAUL
bButtonMaintenanceOut : BOOL; // To change mode into mode MAINTENANCE

// Axis buttons
bButtonMainFwOut : BOOL; // To move main axis forwards
bButtonMainBwOut : BOOL; // To move main axis backwards
bButtonMetalFwOut : BOOL; // To move metal axis forwards
bButtonMetalBwOut : BOOL; // To move metal axis backwards
bButtonPlasticFwOut : BOOL; // To move plastic axis forwards
bButtonPlasticBwOut : BOOL; // To move plastic axis backwards

// Cylinder buttons
bButtonClampToWorkOut : BOOL; // To move manually clamp cylinder to work position
bButtonBarrierToWorkOut : BOOL; // To move manually barrier cylinder to work position
bButtonMetalToWorkOut : BOOL; // To move manually metal cylinder to work position
bButtonPlasticToWorkOut : BOOL; // To move manually plastic cylinder to work position
END_VAR
VAR
// =============== Boxes movement ===========================
nLastBox : INT; // Box on last position of main belt
aBoxMetalMove : ARRAY[1..4] OF BOOL; // bBoxMetalMove[1] is true, if box 1 is moved by metal cylinder
aBoxMetalFree : ARRAY[1..4] OF BOOL; // bBoxMetalFree[1] is true, if any box blockades the movement of box 1 on
metal belt
aBoxPlasticMove : ARRAY[1..4] OF BOOL; // bBoxPlasticMove[1] is true, if box 1 is moved by plastic cylinder
aBoxPlasticFree : ARRAY[1..4] OF BOOL; // bBoxMetalFree[1] is true, if any box blockades the plastic belt
aBoxMainBelt : ARRAY[1..4] OF BOOL; // bBoxMainBelt[1] is true, if box 1 is on main belt and has no obstacles,
so box 1 is moved by main axis
aBoxMetalBelt : ARRAY[1..4] OF BOOL; // aBoxMetalBelt[1] is true, if box 1 is on metal belt and has no
obstacles, so box 1 is moved by metal axis
aBoxPlasticBelt : ARRAY[1..4] OF BOOL; // aBoxPlasticBelt[1] is true, if box 1 is on plastic belt and has no
obstacles, so box 1 is moved by plastic axis
fMainVelo : LREAL; // Velocity of boxes on main belt
fMetalVelo : LREAL; // Velocity of boxes on metal belt
fPlasticVelo : LREAL; // Velocity of boxes on plastic belt
bAccident : BOOL; // True if machine caused an accident
nState : INT;

// =============== Message =================================
nMessages : INT; // Number of messages
sMessageClamp : STRING; // Message of clamp cylinder
sMessageMetalCyl : STRING; // Message of metal cylinder
sMessagePlasticCyl : STRING; // Message of plastic cylinder
sMessageMainAxis : STRING; // Message of main axis
```

```
sMessageMetalAxis : STRING; // Message of metal axis
sMessagePlasticAxis : STRING; // Message of plastic axis

// =============== Axes movement ============================
nAngleMain : INT; // Angle of main axis element
nAngleMetal : INT; // Angle of metal axis element
nAnglePlastic : INT; // Angle of plastic axis element
fbTimerMainAxis : TON; // Timer for movement of main axis element
fbTimerMetalAxis : TON; // Timer for movement of metal axis element
fbTimerPlasticAxis : TON; // Timer for movement of plastic axis element
tMoveAxis : TIME := T#40MS; // Time for movement of elements

// =============== Color variables ==========================
nButtonColorStart : DWORD; // Color of start button
bButtonColorStart : BOOL; // To change color of start button
nButtonColorStop : DWORD; // Color of stop button
bButtonColorStop : BOOL; // To change color of stop button
nButtonColorReset : DWORD; // Color of reset button
bButtonColorReset : BOOL; // To change color of reset button
nButtonColorAuto : DWORD; // Color of automatic button
bButtonColorAuto : BOOL; // To change color of automatic button
nButtonColorSemi : DWORD; // Color of semi-automatic button
bButtonColorSemi : BOOL; // To change color of semi-automatic button
nButtonColorManu : DWORD; // Color of manual button
bButtonColorManu : BOOL; // To change color of manual button
nButtonColorMaint : DWORD; // Color of maintenance button
bButtonColorMaint : BOOL; // To change color of maintenance button
nButtonColorClamp : DWORD; // Color of button to move clamp cylinder
bButtonColorClamp : BOOL; // To change color of clamp button
nButtonColorBarrier : DWORD; // Color of button to move barrier cylinder
bButtonColorBarrier : BOOL; // To change color of barrier button
nButtonColorMetalCyl : DWORD; // Color of button to move metal cylinder
bButtonColorMetalCyl : BOOL; // To change color of metal cylinder button
nButtonColorPlasticCyl : DWORD; // Color of button to move plastic cylinder
bButtonColorPlasticCyl : BOOL; // To change color of plastic cylinder button
nButtonColorMainFw : DWORD; // Color of button to move main axis forwards
bButtonColorMainFw : BOOL; // To change color of button to move main axis fw.
nButtonColorMainBw : DWORD; // Color of button to move main axis backwards
bButtonColorMainBw : BOOL; // To change color of button to move main axis bw.
nButtonColorMetalFw : DWORD; // Color of button to move metal axis forwards
bButtonColorMetalFw : BOOL; // To change color of button to move metal axis fw.
nButtonColorMetalBw : DWORD; // Color of button to move metal axis backwards
bButtonColorMetalBw : BOOL; // To change color of button to move metal axis bw.
nButtonColorPlasticFw : DWORD; // Color of button to move plastic axis forwards
bButtonColorPlasticFw : BOOL; // To change color of button to move plastic axis fw.
nButtonColorPlasticBw : DWORD; // Color of button to move plastic axis backwards
bButtonColorPlasticBw : BOOL; // To change color of button to move plastic axis bw.
nButtonColorComprAir : DWORD; // Color of button to enable/disable compressed air
bButtonColorComprAir : BOOL; // To change color of button to control compressed air
END_VAR
VAR CONSTANT
// =============== Colors ===============================================
cYellow : DWORD := 16#00FFFF80; // Color code for yellow
cGreen : DWORD := 16#0080FF00; // Color code for green
cPink : DWORD := 16#00FF8080; // Color code for pink

// =============== Velocities ===========================================
cVelocityDenominator : LREAL := 250.0; // Velocity of visualization is a fraction of real axis velocity
cAxisMovementSemiPlusAuto : INT := 2; // Axis movement per cycle in mode Automatic and Semi Autoamtic
cAxisMovementManual : INT := 1; // Axis movement per cycle in mode Manual
cClampMovement : LREAL := 0.1; // Cylinder movement per cycle for clamp cylinder
cBarrierMovement : LREAL := 1.0; // Cylinder movement per cycle for barrier cylinder
cMetalPlasticCylMovement : LREAL := 1.0; // Cylinder movement per cycle for metal and plastic cylinder

// =============== Barrier, clamp, metal, plastic cylinder ==============
cYClampCylStartPos : LREAL := 0.0; // Y-start-position of clamp cylinder (where movement starts)
cYClampCylEndPos : LREAL := 3.0; // Y-end-position of clamp cylinder (where movement ends)
cYBarrierCylStartPos : LREAL := 0.0; // Y-start-position of barrier cylinder (where movement starts)
cYBarrierCylEndPos : LREAL := 63.0; // Y-end-position of barrier cylinder (where movement ends)
cYMetalPlasticCylStartPos : LREAL := 0.0; // Y-start-position of metal and plastic cylinder (where movement
starts)
```

```
cYMetalPlasticCylEndPos : LREAL := -73.0; // Y-end-position of metal and plastic cylinder (where movement ends)

// =============== Widths, distances ====================================
cBoxWidth : LREAL := 60.0; // Width of a box in the visu
cBarrierWidth : LREAL := 30.0; // Width of the barrier cylinder in the visu
cDiffBarrierToStartOfMetalCyl : LREAL := 191.0; // Difference on visu from barrier to start of metal cylinder
cDiffBarrierToEndOfMetalCyl : LREAL := 261.0; // Difference on visu from barrier to end of metal cylinder
cDiffBarrierToStartOfPlasticCyl : LREAL := 441.0; // Difference on visu from barrier to start of plastic cylinder
cYMetalPlasticCylOnMainBelt : LREAL := -9.0; // Y-Position of metal and plastic cylinder reaching the main belt
cYBoxesOnMetalPlasticBelt_min : LREAL := -65.0; // Minimal Y-Position of boxes when being on metal or plastic belt
cYBoxesOnMetalPlasticBelt_max : LREAL := -140.0; // Maximal Y-Position of boxes when being on metal or plastic belt

// =============== Positions of boxes - main belt =======================
cYforAllBoxesOnMainBelt : LREAL := -5.0; // Y-Position of boxes being on main belt
cYforAllBoxesNotOnMainBelt : LREAL := -60.0; // Y-Position of boxes not being on main belt

cYforAllBoxesStartPosMainBelt : LREAL := 0.0; // Y-Position of boxes when being on first position of main belt
cXforAllBoxesStartPosMainBelt : LREAL := -240.0; // X-Position of boxes when being on first position of main belt
cXforAllBoxesLastPossiblePosMainBelt : LREAL := 461.0; // Last possible X-position of boxes on main belt

cXforAllBoxesCorrectPushMetalCyl_min : LREAL := 180; // Minimal X-Position of boxes when being pushed by metal cylinder to metal belt
cXforAllBoxesCorrectPushMetalCyl_max : LREAL := 211; // Maximal X-Position of boxes when being pushed by metal cylinder to metal belt
cXforAllBoxesIncorrectPushMetalCyl_min : LREAL := 131; // Minimal X-Position of boxes when being pushed by metal cylinder, but not pushed to metal belt
cXforAllBoxesIncorrectPushMetalCyl_max : LREAL := 261; // Maximal X-Position of boxes when being pushed by metal cylinder, but not pushed to metal belt

cXforAllBoxesCorrectPushPlasticCyl_min : LREAL := 430; // Minimal X-Position of boxes when being pushed by plastic cylinder to plastic belt
cXforAllBoxesCorrectPushPlasticCyl_max : LREAL := 461; // Minimal X-Position of boxes when being pushed by plastic cylinder to plastic belt
cXforAllBoxesIncorrectPushPlasticCyl_min : LREAL := 381; // Minimal X-Position of boxes when being pushed by plastic cylinder, but not pushed to plastic belt

// Box 1
cXforBox1_MainBelt_Diff2to1_min : LREAL := 0.001; // Minimal distance from box 2 to box 1 so that box 1 can be moved on main belt
cXforBox1_MainBelt_Diff2to1_max : LREAL := 119.99999; // Maximal distance from box 2 to box 1 so that box 1 can be moved on main belt
cXforBox1_MainBelt_Diff3to1_min : LREAL := 60.001; // Minimal distance from box 3 to box 1 so that box 1 can be moved on main belt
cXforBox1_MainBelt_Diff3to1_max : LREAL := 179.99999; // Maximal distance from box 3 to box 1 so that box 1 can be moved on main belt
cXforBox1_MainBelt_Diff4to1_min : LREAL := 120.001; // Minimal distance from box 4 to box 1 so that box 1 can be moved on main belt
cXforBox1_MainBelt_Diff4to1_max : LREAL := 239.99999; // Maximal distance from box 4 to box 1 so that box 1 can be moved on main belt

// Box 2
cXforBox2_MainBelt_Diff1to2_min : LREAL := -0.00001; // Minimal distance from box 1 to box 2 so that box 2 can be moved on main belt
cXforBox2_MainBelt_Diff1to2_max : LREAL := -119.999; // Maximal distance from box 1 to box 2 so that box 2 can be moved on main belt
cXforBox2_MainBelt_Diff3to2_min : LREAL := 119.99999; // Minimal distance from box 3 to box 2 so that box 2 can be moved on main belt
cXforBox2_MainBelt_Diff3to2_max : LREAL := 0.001; // Maximal distance from box 3 to box 2 so that box 2 can be moved on main belt
cXforBox2_MainBelt_Diff4to2_min : LREAL := 179.99999; // Minimal distance from box 4 to box 2 so that box 2 can be moved on main belt
cXforBox2_MainBelt_Diff4to2_max : LREAL := 60.001; // Maximal distance from box 4 to box 2 so that box 2 can be moved on main belt

// Box 3
cXforBox3_MainBelt_Diff1to3_min : LREAL := -60.00001; // Minimal distance from box 1 to box 3 so that box 3 can be moved on main belt
```

```
cXforBox3_MainBelt_Diff1to3_max : LREAL := -179.999; // Maximal distance from box 1 to box 3 so that box 3 can be moved on main belt
cXforBox3_MainBelt_Diff2to3_min : LREAL := -0.00001; // Minimal distance from box 2 to box 3 so that box 3 can be moved on main belt
cXforBox3_MainBelt_Diff2to3_max : LREAL := -119.999; // Maximal distance from box 2 to box 3 so that box 3 can be moved on main belt
cXforBox3_MainBelt_Diff4to3_min : LREAL := 119.99999; // Minimal distance from box 4 to box 3 so that box 3 can be moved on main belt
cXforBox3_MainBelt_Diff4to3_max : LREAL := 0.001; // Maximal distance from box 4 to box 3 so that box 3 can be moved on main belt

// Box 4
cXforBox4_MainBelt_Diff1to4_min : LREAL := -120.00001; // Minimal distance from box 1 to box 4 so that box 4 can be moved on main belt
cXforBox4_MainBelt_Diff1to4_max : LREAL := -239.999; // Maximal distance from box 1 to box 4 so that box 4 can be moved on main belt
cXforBox4_MainBelt_Diff2to4_min : LREAL := -60.00001; // Minimal distance from box 2 to box 4 so that box 4 can be moved on main belt
cXforBox4_MainBelt_Diff2to4_max : LREAL := -179.999; // Maximal distance from box 2 to box 4 so that box 4 can be moved on main belt
cXforBox4_MainBelt_Diff3to4_min : LREAL := -0.00001; // Minimal distance from box 3 to box 4 so that box 4 can be moved on main belt
cXforBox4_MainBelt_Diff3to4_max : LREAL := -119.999; // Maximal distance from box 3 to box 4 so that box 4 can be moved on main belt

// =============== Positions of boxes - metal belt ======================
// Box 1
cXforBox1OnMetalBelt_min : LREAL := 240; // Minimal X-Position of box 1 when being on metal belt
cXforBox1OnMetalBelt_max : LREAL := 271; // Maximal X-Position of box 1 when being on metal belt
// Box 2
cXforBox2OnMetalBelt_min : LREAL := 300; // Minimal X-Position of box 2 when being on metal belt
cXforBox2OnMetalBelt_max : LREAL := 331; // Maximal X-Position of box 2 when being on metal belt
// Box 3
cXforBox3OnMetalBelt_min : LREAL := 360; // Minimal X-Position of box 3 when being on metal belt
cXforBox3OnMetalBelt_max : LREAL := 391; // Maximal X-Position of box 3 when being on metal belt
// Box 4
cXforBox4OnMetalBelt_min : LREAL := 420; // Minimal X-Position of box 4 when being on metal belt
cXforBox4OnMetalBelt_max : LREAL := 451; // Maximal X-Position of box 4 when being on metal belt

// =============== Positions of boxes - plastic belt ====================
// Box 1
cXforBox1OnPlasticBelt_min : LREAL := 470; // Minimal X-Position of box 1 when being on plastic belt
cXforBox1OnPlasticBelt_max : LREAL := 521; // Maximal X-Position of box 1 when being on plastic belt
// Box 2
cXforBox2OnPlasticBelt_min : LREAL := 530; // Minimal X-Position of box 2 when being on plastic belt
cXforBox2OnPlasticBelt_max : LREAL := 581; // Maximal X-Position of box 2 when being on plastic belt
// Box 3
cXforBox3OnPlasticBelt_min : LREAL := 590; // Minimal X-Position of box 3 when being on plastic belt
cXforBox3OnPlasticBelt_max : LREAL := 641; // Maximal X-Position of box 3 when being on plastic belt
// Box 4
cXforBox4OnPlasticBelt_min : LREAL := 650; // Minimal X-Position of box 4 when being on plastic belt
cXforBox4OnPlasticBelt_max : LREAL := 701; // Maximal X-Position of box 4 when being on plastic belt
END_VAR
```

Implementation

```
// =========================================================
// Resetting visu

IF bReset THEN
Reset();
END_IF

// =========================================================
// Because of the visualisation size, the boxes velocity is a fraction of the axis velocity

IF bAuto OR bSemi OR bButtonMainFwIn THEN
fMainVelo := (fAxisVelo / cVelocityDenominator);
ELSIF bButtonMainBwIn THEN
fMainVelo := -(fAxisVelo / cVelocityDenominator);
```

```
END_IF

IF bAuto OR bSemi OR bButtonMetalFwIn THEN
fMetalVelo := -(fAxisVelo / cVelocityDenominator);
ELSIF bButtonMetalBwIn THEN
fMetalVelo := (fAxisVelo / cVelocityDenominator);
END_IF

IF bAuto OR bSemi OR bButtonPlasticFwIn THEN
fPlasticVelo := -(fAxisVelo / cVelocityDenominator);
ELSIF bButtonPlasticBwIn THEN
fPlasticVelo := (fAxisVelo / cVelocityDenominator);
END_IF

// =========================================================
// In case of accident the power button is turned off

IF bAccident THEN
IF bButtonPowerOut THEN
bButtonPowerOut := FALSE;
END_IF

IF bStopped THEN
bAccident := FALSE;
END_IF
END_IF

// =========================================================
// Visuazation functions

// Rotation of axis elements
AxisRotation();

// Buttons, order of boxes, error messages
ButtonSignals();
ButtonColors();
LastBox();
ErrorMessages();

// Motion of cylinders
Barrier_Clamp_Cyl();
Metal_Plastic_Cyl();

// Motion of boxes
AxisMoveBoxes();
CylinderMoveBoxes();

// =========================================================
```

Method AxisMoveBoxes Declaration

```
METHOD PRIVATE AxisMoveBoxes
VAR_INPUT
END_VAR
```

Method AxisMoveBoxes Implementation

```
// =========================================================
// Checking traction standby on main belt
// bBoxMainBelt[1] is true, if box 1 is on main belt and has no obstacles, so box 1 is moved by main axis

aBoxMainBelt[1] := // box 1 on main belt
aBoxY[1] >= cYforAllBoxesOnMainBelt
// not previous to box 2
AND ( (aBoxX[2] - aBoxX[1] - fMainVelo) >= cXforBox1_MainBelt_Diff2to1_max
OR (aBoxX[2] - aBoxX[1] - fMainVelo) <= cXforBox1_MainBelt_Diff2to1_min
OR aBoxY[2] < cYforAllBoxesNotOnMainBelt)
// not previous to box 3
AND ( (aBoxX[3] - aBoxX[1] - fMainVelo) >= cXforBox1_MainBelt_Diff3to1_max
OR (aBoxX[3] - aBoxX[1]- fMainVelo) <= cXforBox1_MainBelt_Diff3to1_min
OR aBoxY[3] < cYforAllBoxesNotOnMainBelt)
// not previous to box 4
AND ( (aBoxX[4] - aBoxX[1] - fMainVelo) >= cXforBox1_MainBelt_Diff4to1_max
OR (aBoxX[4] - aBoxX[1]- fMainVelo) <= cXforBox1_MainBelt_Diff4to1_min
OR aBoxY[4] < cYforAllBoxesNotOnMainBelt);

aBoxMainBelt[2] := // box 2 on main belt
aBoxY[2] >= cYforAllBoxesOnMainBelt
// not previous to box 1
AND ( (aBoxX[1] - aBoxX[2] - fMainVelo) >= cXforBox2_MainBelt_Diff1to2_min
OR (aBoxX[1] - aBoxX[2] - fMainVelo) <= cXforBox2_MainBelt_Diff1to2_max
OR aBoxY[1] < cYforAllBoxesNotOnMainBelt)
// not previous to box 3
AND ( (aBoxX[3] - aBoxX[2] - fMainVelo) >= cXforBox2_MainBelt_Diff3to2_min
OR (aBoxX[3] - aBoxX[2] - fMainVelo) <= cXforBox2_MainBelt_Diff3to2_max
OR aBoxY[3] < cYforAllBoxesNotOnMainBelt)
// not previous to box 4
AND ( (aBoxX[4] - aBoxX[2] - fMainVelo) >= cXforBox2_MainBelt_Diff4to2_min
OR (aBoxX[4] - aBoxX[2] - fMainVelo) <= cXforBox2_MainBelt_Diff4to2_max
OR aBoxY[4] < cYforAllBoxesNotOnMainBelt);

aBoxMainBelt[3] := // box 3 on main belt
aBoxY[3] >= cYforAllBoxesOnMainBelt
// not previous to box 1
AND ( (aBoxX[1] - aBoxX[3] - fMainVelo) >= cXforBox3_MainBelt_Diff1to3_min
OR (aBoxX[1] - aBoxX[3] - fMainVelo) <= cXforBox3_MainBelt_Diff1to3_max
OR aBoxY[1] < cYforAllBoxesNotOnMainBelt)
// not previous to box 2
AND ( (aBoxX[2] - aBoxX[3] - fMainVelo) >= cXforBox3_MainBelt_Diff2to3_min
OR (aBoxX[2] - aBoxX[3] - fMainVelo) <= cXforBox3_MainBelt_Diff2to3_max
OR aBoxY[2] < cYforAllBoxesNotOnMainBelt)
// not previous to box 4
AND ( (aBoxX[4] - aBoxX[3] - fMainVelo) >= cXforBox3_MainBelt_Diff4to3_min
OR (aBoxX[4] - aBoxX[3] - fMainVelo) <= cXforBox3_MainBelt_Diff4to3_max
OR aBoxY[4] < cYforAllBoxesNotOnMainBelt);

aBoxMainBelt[4] := // box 4 on main belt
aBoxY[4] >= cYforAllBoxesOnMainBelt
// not previous to box 1
AND ( (aBoxX[1] - aBoxX[4] - fMainVelo) >= cXforBox4_MainBelt_Diff1to4_min
OR (aBoxX[1] - aBoxX[4] - fMainVelo) <= cXforBox4_MainBelt_Diff1to4_max
OR aBoxY[1] < cYforAllBoxesNotOnMainBelt)
// not previous to box 2
AND ( (aBoxX[2] - aBoxX[4] - fMainVelo) >= cXforBox4_MainBelt_Diff2to4_min
OR (aBoxX[2] - aBoxX[4] - fMainVelo) <= cXforBox4_MainBelt_Diff2to4_max
OR aBoxY[2] < cYforAllBoxesNotOnMainBelt)
// not previous to box 3
AND ( (aBoxX[3] - aBoxX[4] - fMainVelo) >= cXforBox4_MainBelt_Diff3to4_min
OR (aBoxX[3] - aBoxX[4] - fMainVelo) <= cXforBox4_MainBelt_Diff3to4_max
OR aBoxY[3] < cYforAllBoxesNotOnMainBelt);

// ===========================================================
// Motion of boxes on main belt - forwards

IF bMainAxisMoves AND (bAuto OR bSemi OR bButtonMainFwIn) THEN
FOR nState := 1 TO 4 BY 1 DO
IF aBoxMainBelt[nState] THEN
// not previous to moving metal cylinder
IF aBoxX[nState] >= (cBoxWidth*nState + cBarrierWidth) AND aBoxX[nState] < (cBoxWidth*nState +
cDiffBarrierToEndOfMetalCyl - 1) THEN
// Metal cylinder over main belt
IF fYMetal <= cYMetalPlasticCylOnMainBelt THEN
IF aBoxX[nState] < (cBoxWidth*nState + cDiffBarrierToStartOfMetalCyl - cBoxWidth + 0.25)
AND aBoxX[nState] > (cBoxWidth*nState + cDiffBarrierToStartOfMetalCyl - cBoxWidth - 0.15) THEN
aBoxX[nState] := cBoxWidth*nState + cDiffBarrierToStartOfMetalCyl - cBoxWidth;
ELSIF aBoxX[nState] <= (cBoxWidth*nState + cDiffBarrierToStartOfMetalCyl - cBoxWidth - 0.15) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF
// Metal cylinder not over main belt
ELSE
aBoxX[nState] := aBoxX[nState] + fMainVelo;
```

```
END_IF
END_IF

// not previous to moving plastic cylinder
IF aBoxX[nState] >= (cBoxWidth*nState + cDiffBarrierToEndOfMetalCyl - 1) THEN
// Plastic cylinder over main belt
IF fYPlastic <= cYMetalPlasticCylOnMainBelt THEN
IF aBoxX[nState] < (cBoxWidth*nState + cDiffBarrierToStartOfPlasticCyl - cBoxWidth + 0.25)
AND aBoxX[nState] > (cBoxWidth*nState + cDiffBarrierToStartOfPlasticCyl - cBoxWidth - 0.15) THEN
aBoxX[nState] := cBoxWidth*nState + cDiffBarrierToStartOfPlasticCyl - cBoxWidth;
ELSIF aBoxX[nState] <= (cBoxWidth*nState + cDiffBarrierToStartOfPlasticCyl - cBoxWidth - 0.15) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF
// Plastic cylinder not over main belt
ELSE
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF
END_IF

// barrier open and box 1 in the first place
IF fYBarrier >= (cBoxWidth + 1) AND aBoxX[nState] >= cBoxWidth*(nState - 1) AND aBoxX[nState] <
(cBoxWidth*nState + cBarrierWidth) THEN
IF aBoxX[nState] + fMainVelo < (cBoxWidth*nState + cBarrierWidth) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
ELSIF aBoxX[nState] < (cBoxWidth*nState + cBarrierWidth) THEN
aBoxX[nState] := (cBoxWidth*nState + cBarrierWidth);
END_IF
END_IF

// not previous to clamp
IF aBoxX[nState] + fMainVelo < -cBoxWidth*(3 - nState) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
ELSIF aBoxX[nState] < -cBoxWidth*(nState + 1) THEN
aBoxX[nState] := -cBoxWidth*(nState + 1);
END_IF

// clamp open, not after clamp
IF fYClamp >= 1 AND aBoxX[nState] >= -cBoxWidth*(nState + 1) THEN
IF aBoxX[nState] + fMainVelo < cBoxWidth*(nState - 1) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
ELSIF aBoxX[nState] < cBoxWidth*(nState - 1) THEN
aBoxX[nState] := cBoxWidth*(nState - 1);
END_IF
END_IF
END_IF
END_FOR
END_IF

// ============================================================
// Motion of boxes on main belt - backwards

IF bMainAxisMoves AND bButtonMainBwIn THEN
FOR nState := 1 TO 4 BY 1 DO
IF aBoxMainBelt[nState] THEN
// not previous to moving metal cylinder
IF aBoxX[nState] >= (cBoxWidth*nState + cBoxWidth + cBarrierWidth) AND aBoxX[nState] < (cBoxWidth*nState +
cDiffBarrierToEndOfMetalCyl + 1) THEN
IF NOT (aBoxX[nState] < (cBoxWidth*nState + cDiffBarrierToEndOfMetalCyl + 0.25)
AND aBoxX[nState] > (cBoxWidth*nState + cDiffBarrierToEndOfMetalCyl - 0.15)
AND fYMetal <= cYMetalPlasticCylOnMainBelt) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
ELSE
aBoxX[nState] := cBoxWidth*nState + cDiffBarrierToEndOfMetalCyl;
END_IF
END_IF

// not previous to moving plastic cylinder
IF aBoxX[nState] >= (cBoxWidth*nState + cDiffBarrierToEndOfMetalCyl + 1) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF
```

```
// not previous to moving barrier cylinder
IF aBoxX[nState] > cBoxWidth*(nState - 1) AND aBoxX[nState] < (cBoxWidth*nState + cBoxWidth + cBarrierWidth)
THEN
IF NOT (aBoxX[nState] < (cBoxWidth*nState + cBarrierWidth + 0.25)
AND aBoxX[nState] > (cBoxWidth*nState + cBarrierWidth - 0.15)
AND fYBarrier < (cBoxWidth + 1)) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
ELSE
aBoxX[nState] := cBoxWidth*nState + cBarrierWidth;
END_IF
END_IF

// barrier open and box 1 after the first place
IF fYBarrier >= (cBoxWidth + 1) AND aBoxX[nState] < cBoxWidth*(nState + 1) AND aBoxX[nState] > cBoxWidth*(nState
- 1) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF

// clamp open, not previous to clamp
IF fYClamp >= 1 AND aBoxX[nState] <= cBoxWidth*(nState-1) AND aBoxX[nState] > -cBoxWidth*(3-nState) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF

// not at the beginning of main belt
IF aBoxX[nState] > -cBoxWidth*(4-nState) AND aBoxX[nState] <= -cBoxWidth*(3-nState) THEN
aBoxX[nState] := aBoxX[nState] + fMainVelo;
END_IF
END_IF
END_FOR
END_IF

// =========================================================
// =========================================================
// Checking traction standby on metal belt

aBoxMetalBelt[1] := // box 1 on metal belt
aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max
// not previous to box 2
AND ((aBoxY[2] - aBoxY[1] - fMetalVelo) >= cBoxWidth OR (aBoxY[2] - aBoxY[1] - fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max))
// not previous to box 3
AND ((aBoxY[3] - aBoxY[1]- fMetalVelo) >= cBoxWidth OR (aBoxY[3] - aBoxY[1]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max))
// not previous to box 4
AND ((aBoxY[4] - aBoxY[1]- fMetalVelo) >= cBoxWidth OR (aBoxY[4] - aBoxY[1]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max));

aBoxMetalBelt[2] := aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max
AND ((aBoxY[1] - aBoxY[2] - fMetalVelo) >= cBoxWidth OR (aBoxY[1] - aBoxY[2] - fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max))
AND ((aBoxY[3] - aBoxY[2]- fMetalVelo) >= cBoxWidth OR (aBoxY[3] - aBoxY[2]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max))
AND ((aBoxY[4] - aBoxY[2]- fMetalVelo) >= cBoxWidth OR (aBoxY[4] - aBoxY[2]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max));

aBoxMetalBelt[3] := aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max
AND ((aBoxY[1] - aBoxY[3] - fMetalVelo) >= cBoxWidth OR (aBoxY[1] - aBoxY[3] - fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max))
AND ((aBoxY[2] - aBoxY[3]- fMetalVelo) >= cBoxWidth OR (aBoxY[2] - aBoxY[3]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max))
AND ((aBoxY[4] - aBoxY[3]- fMetalVelo) >= cBoxWidth OR (aBoxY[4] - aBoxY[3]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max));

aBoxMetalBelt[4] := aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max
AND ((aBoxY[1] - aBoxY[4] - fMetalVelo) >= cBoxWidth OR (aBoxY[1] - aBoxY[4] - fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max))
AND ((aBoxY[2] - aBoxY[4] - fMetalVelo) >= cBoxWidth OR (aBoxY[2] - aBoxY[4] - fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max))
AND ((aBoxY[3] - aBoxY[4]- fMetalVelo) >= cBoxWidth OR (aBoxY[3] - aBoxY[4]- fMetalVelo) <= -cBoxWidth
OR NOT (aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max));
```

```
// ========================================================
// Motion of boxes on metal belt and back to the start position number 4

FOR nState := 1 TO 4 BY 1 DO
IF bMetalAxisMoves THEN
IF aBoxMetalBelt[nState] THEN
// Check the state and direction of motion
IF bAuto OR bSemi OR (NOT bAuto AND bButtonMetalFwIn) THEN
// Move box to the end of metal belt and then back to position 4
IF aBoxY[nState] <= (cYBoxesOnMetalPlasticBelt_min + 0.1) AND aBoxY[nState] > cYBoxesOnMetalPlasticBelt_max THEN
aBoxY[nState] := aBoxY[nState] + fMetalVelo;
END_IF

ELSIF NOT bAuto AND bButtonMetalBwIn THEN
IF aBoxY[nState] + fMetalVelo <= (cYBoxesOnMetalPlasticBelt_min - 0.1) THEN
aBoxY[nState] := aBoxY[nState] + fMetalVelo;
END_IF
END_IF
END_IF
END_IF

IF aBoxY[nState] <= cYBoxesOnMetalPlasticBelt_max AND nLastBox = 0 THEN
aBoxY[nState] := cYforAllBoxesStartPosMainBelt;
aBoxX[nState] := cXforAllBoxesStartPosMainBelt + nState*cBoxWidth;
END_IF
END_FOR

// ========================================================
// ========================================================
// Checking traction standby on plastic belt

aBoxPlasticBelt[1] := // box 1 on plastic belt
aBoxX[1] > cXforBox1OnPlasticBelt_min AND aBoxX[1] <= cXforBox1OnPlasticBelt_max
// not previous to box 2
AND ((aBoxY[2] - aBoxY[1] - fPlasticVelo) >= cBoxWidth OR (aBoxY[2] - aBoxY[1] - fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[2] > cXforBox2OnPlasticBelt_min AND aBoxX[2] <= cXforBox2OnPlasticBelt_max))
// not previous to box 3
AND ((aBoxY[3] - aBoxY[1]- fPlasticVelo) >= cBoxWidth OR (aBoxY[3] - aBoxY[1]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[3] > cXforBox3OnPlasticBelt_min AND aBoxX[3] <= cXforBox3OnPlasticBelt_max))
// not previous to box 4
AND ((aBoxY[4] - aBoxY[1]- fPlasticVelo) >= cBoxWidth OR (aBoxY[4] - aBoxY[1]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[4] > cXforBox4OnPlasticBelt_min AND aBoxX[4] <= cXforBox4OnPlasticBelt_max));

aBoxPlasticBelt[2] := aBoxX[2] > cXforBox2OnPlasticBelt_min AND aBoxX[2] <= cXforBox2OnPlasticBelt_max
AND ((aBoxY[1] - aBoxY[2] - fPlasticVelo) >= cBoxWidth OR (aBoxY[1] - aBoxY[2] - fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[1] > cXforBox1OnPlasticBelt_min AND aBoxX[1] <= cXforBox1OnPlasticBelt_max))
AND ((aBoxY[3] - aBoxY[2]- fPlasticVelo) >= cBoxWidth OR (aBoxY[3] - aBoxY[2]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[3] > cXforBox3OnPlasticBelt_min AND aBoxX[3] <= cXforBox3OnPlasticBelt_max))
AND ((aBoxY[4] - aBoxY[2]- fPlasticVelo) >= cBoxWidth OR (aBoxY[4] - aBoxY[2]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[4] > cXforBox4OnPlasticBelt_min AND aBoxX[4] <= cXforBox4OnPlasticBelt_max));

aBoxPlasticBelt[3] := aBoxX[3] > cXforBox3OnPlasticBelt_min AND aBoxX[3] <= cXforBox3OnPlasticBelt_max
AND ((aBoxY[1] - aBoxY[3] - fPlasticVelo) >= cBoxWidth OR (aBoxY[1] - aBoxY[3] - fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[1] > cXforBox1OnPlasticBelt_min AND aBoxX[1] <= cXforBox1OnPlasticBelt_max))
AND ((aBoxY[2] - aBoxY[3]- fPlasticVelo) >= cBoxWidth OR (aBoxY[2] - aBoxY[3]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[2] > cXforBox2OnPlasticBelt_min AND aBoxX[2] <= cXforBox2OnPlasticBelt_max))
AND ((aBoxY[4] - aBoxY[3]- fPlasticVelo) >= cBoxWidth OR (aBoxY[4] - aBoxY[3]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[4] > cXforBox4OnPlasticBelt_min AND aBoxX[4] <= cXforBox4OnPlasticBelt_max));

aBoxPlasticBelt[4] := aBoxX[4] > cXforBox4OnPlasticBelt_min AND aBoxX[4] <= cXforBox4OnPlasticBelt_max
AND ((aBoxY[1] - aBoxY[4] - fPlasticVelo) >= cBoxWidth OR (aBoxY[1] - aBoxY[4] - fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[1] > cXforBox1OnPlasticBelt_min AND aBoxX[1] <= cXforBox1OnPlasticBelt_max))
AND ((aBoxY[2] - aBoxY[4] - fPlasticVelo) >= cBoxWidth OR (aBoxY[2] - aBoxY[4] - fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[2] > cXforBox2OnPlasticBelt_min AND aBoxX[2] <= cXforBox2OnPlasticBelt_max))
AND ((aBoxY[3] - aBoxY[4]- fPlasticVelo) >= cBoxWidth OR (aBoxY[3] - aBoxY[4]- fPlasticVelo) <= -cBoxWidth
OR NOT (aBoxX[3] > cXforBox3OnPlasticBelt_min AND aBoxX[3] <= cXforBox3OnPlasticBelt_max));

// ========================================================
// Motion of boxes on metal belt and back to the start position number 4

FOR nState := 1 TO 4 BY 1 DO
```

```
IF bPlasticAxisMoves THEN
IF aBoxPlasticBelt[nState] THEN
// Check the state and direction of motion
IF bAuto OR bSemi OR (NOT bAuto AND bButtonPlasticFwIn) THEN
// Move box to the end of metal belt and then back to position 4
IF aBoxY[nState] <= (cYBoxesOnMetalPlasticBelt_min + 0.1) AND aBoxY[nState] > cYBoxesOnMetalPlasticBelt_max THEN
aBoxY[nState] := aBoxY[nState] + fPlasticVelo;
END_IF

ELSIF NOT bAuto AND bButtonPlasticBwIn THEN
IF aBoxY[nState] + fPlasticVelo <= (cYBoxesOnMetalPlasticBelt_min - 0.1) THEN
aBoxY[nState] := aBoxY[nState] + fPlasticVelo;
END_IF
END_IF
END_IF
END_IF

IF aBoxY[nState] <= cYBoxesOnMetalPlasticBelt_max AND nLastBox = 0 THEN
aBoxY[nState] := cYforAllBoxesStartPosMainBelt;
aBoxX[nState] := cXforAllBoxesStartPosMainBelt + nState*cBoxWidth;
END_IF
END_FOR

// ========================================================
```

Method AxisRotation Declaration

```
METHOD PRIVATE AxisRotation
VAR_INPUT
END_VAR
```

Method AxisRotation Implementation

```
// ========================================================
// Rotation of main axis

fbTimerMainAxis(IN := bMainAxisMoves,
PT := tMoveAxis);

IF fbTimerMainAxis.Q THEN
fbTimerMainAxis(IN := FALSE);

IF bAuto OR bSemi THEN
nAngleMain := nAngleMain + cAxisMovementSemiPlusAuto;

ELSIF bButtonMainFwIn THEN
nAngleMain := nAngleMain + cAxisMovementManual;

ELSIF bButtonMainBwIn THEN
nAngleMain := nAngleMain - cAxisMovementManual;
END_IF
END_IF

// ========================================================
// Rotation of metal axis

fbTimerMetalAxis( IN := bMetalAxisMoves,
PT := tMoveAxis);

IF fbTimerMetalAxis.Q THEN
fbTimerMetalAxis( IN := FALSE );

IF bAuto OR bSemi THEN
nAngleMetal := nAngleMetal + cAxisMovementSemiPlusAuto;

ELSIF bButtonMetalFwIn THEN
nAngleMetal := nAngleMetal + cAxisMovementManual;

ELSIF bButtonMetalBwIn THEN
nAngleMetal := nAngleMetal - cAxisMovementManual;
```

```
END_IF
END_IF

// =========================================================
// Rotation of plastic axis

fbTimerPlasticAxis( IN := bPlasticAxisMoves,
PT := tMoveAxis);

IF fbTimerPlasticAxis.Q THEN
fbTimerPlasticAxis( IN := FALSE );

IF bAuto OR bSemi THEN
nAnglePlastic := nAnglePlastic + cAxisMovementSemiPlusAuto;

ELSIF bButtonPlasticFwIn THEN
nAnglePlastic := nAnglePlastic + cAxisMovementManual;

ELSIF bButtonPlasticBwIn THEN
nAnglePlastic := nAnglePlastic - cAxisMovementManual;
END_IF
END_IF

// =========================================================
```

Method Barrier_Clamp_Cyl Declaration

```
METHOD PRIVATE Barrier_Clamp_Cyl
VAR_INPUT
END_VAR
```

Method Barrier_Clamp_Cyl Implementation

```
// =========================================================
// Moving clamp cylinder

IF bClampToWork THEN
IF (fYClamp + cClampMovement) < cYClampCylEndPos THEN
fYClamp := fYClamp + cClampMovement;
ELSE
fYClamp := cYClampCylEndPos;
END_IF

ELSE
IF (fYClamp - cClampMovement) > cYClampCylStartPos THEN
fYClamp := fYClamp - cClampMovement;
ELSE
fYClamp := cYClampCylStartPos;
END_IF
END_IF

// =========================================================
// =========================================================
// Moving barrier cylinder

IF bBarrierToWork THEN
IF (fYBarrier + cBarrierMovement) < cYBarrierCylEndPos THEN
fYBarrier := fYBarrier + cBarrierMovement;
ELSE
fYBarrier := cYBarrierCylEndPos;
END_IF

ELSE
IF (fYBarrier - cBarrierMovement) > cYBarrierCylStartPos THEN
fYBarrier := fYBarrier - cBarrierMovement;
ELSE
fYBarrier := cYBarrierCylStartPos;
END_IF
END_IF
```

```
// =========================================================
// Checking position of boxes related to the barrier cylinder

IF fYBarrier <= cBoxWidth AND NOT bBarrierToWork THEN
FOR nState := 1 TO 4 BY 1 DO
IF aBoxX[nState] > (-cBoxWidth + 0.11 + cBoxWidth*nState) AND aBoxX[nState] < (cBoxWidth*nState + cBarrierWidth)
AND aBoxY[nState] = 0
AND NOT bStopped THEN
bAccident := TRUE;
END_IF
END_FOR
END_IF

// =========================================================
```

Method ButtonColors Declaration

```
METHOD PRIVATE ButtonColors
VAR_INPUT
END_VAR
```

Method ButtonColors Implementation

```
// =========================================================
// Start button

IF (bAccident OR (bPowerEnabled AND (NOT bStarted OR (bSemi AND bStarted))
AND NOT bButtonStopIn AND NOT bButtonAbortIn AND NOT bButtonResetIn
AND (bAuto OR bSemi OR bManual OR bMaintenance))) AND NOT (bError AND bIdle) THEN

IF bButtonStartOut THEN
nButtonColorStart := cGreen;
ELSE
nButtonColorStart := cYellow;
END_IF

bButtonColorStart := TRUE;
ELSE
bButtonColorStart := FALSE;
END_IF

// =========================================================
// Stop button

IF (bStarted OR bIdle) AND (bAuto OR bSemi OR bManual OR bMaintenance) THEN
IF bButtonStopOut THEN
nButtonColorStop := cGreen;
ELSE
nButtonColorStop := cYellow;
END_IF

bButtonColorStop := TRUE;
ELSE
bButtonColorStop := FALSE;
END_IF

// =========================================================
// Reset button

IF bStopped AND bPowerEnabled THEN
IF bButtonResetOut THEN
nButtonColorReset := cGreen;
ELSE
nButtonColorReset := cYellow;
END_IF

bButtonColorReset := TRUE;
ELSE
bButtonColorReset := FALSE;
END_IF
```

```
// =========================================================
// Mode buttons

IF bPowerEnabled THEN
IF NOT bAuto THEN
IF bIdle THEN
nButtonColorAuto := cYellow;
bButtonColorAuto := TRUE;
ELSE
bButtonColorAuto := FALSE;
END_IF
ELSE
nButtonColorAuto := cGreen;
bButtonColorAuto := TRUE;
END_IF

IF NOT bSemi THEN
IF bIdle THEN
nButtonColorSemi := cYellow;
bButtonColorSemi := TRUE;
ELSE
bButtonColorSemi := FALSE;
END_IF
ELSE
nButtonColorSemi := cGreen;
bButtonColorSemi := TRUE;
END_IF

IF NOT bManual THEN
IF bIdle THEN
nButtonColorManu := cYellow;
bButtonColorManu := TRUE;
ELSE
bButtonColorManu := FALSE;
END_IF
ELSE
nButtonColorManu := cGreen;
bButtonColorManu := TRUE;
END_IF

IF NOT bMaintenance THEN
IF bIdle THEN
nButtonColorMaint := cYellow;
bButtonColorMaint := TRUE;
ELSE
bButtonColorMaint := FALSE;
END_IF
ELSE
nButtonColorMaint := cGreen;
bButtonColorMaint := TRUE;
END_IF
END_IF

// =========================================================
// Buttons to move cylinders and axes

IF (bManual OR bMaintenance) AND bStarted THEN
// Cylinders
IF bButtonClampToWorkOut THEN
nButtonColorClamp := cPink;
ELSE
nButtonColorClamp := cYellow;
END_IF

IF bButtonBarrierToWorkOut THEN
nButtonColorBarrier := cPink;
ELSE
nButtonColorBarrier := cYellow;
END_IF
```

```
IF bButtonMetalToWorkOut THEN
nButtonColorMetalCyl := cPink;
ELSE
nButtonColorMetalCyl := cYellow;
END_IF

IF bButtonPlasticToWorkOut THEN
nButtonColorPlasticCyl := cPink;
ELSE
nButtonColorPlasticCyl := cYellow;
END_IF

bButtonColorClamp := TRUE;
bButtonColorBarrier := TRUE;
bButtonColorMetalCyl := TRUE;
bButtonColorPlasticCyl := TRUE;

// Axes
IF bButtonMainBwOut THEN
nButtonColorMainBw := cPink;
ELSE
nButtonColorMainBw := cYellow;
END_IF

IF bButtonMainFwOut THEN
nButtonColorMainFw := cPink;
ELSE
nButtonColorMainFw := cYellow;
END_IF

IF bButtonMetalBwOut THEN
nButtonColorMetalBw := cPink;
ELSE
nButtonColorMetalBw := cYellow;
END_IF

IF bButtonMetalFwOut THEN
nButtonColorMetalFw := cPink;
ELSE
nButtonColorMetalFw := cYellow;
END_IF

IF bButtonPlasticBwOut THEN
nButtonColorPlasticBw := cPink;
ELSE
nButtonColorPlasticBw := cYellow;
END_IF

IF bButtonPlasticFwOut THEN
nButtonColorPlasticFw := cPink;
ELSE
nButtonColorPlasticFw := cYellow;
END_IF

bButtonColorMainBw := NOT bButtonMainFwOut;
bButtonColorMainFw := NOT bButtonMainBwOut;
bButtonColorMetalBw := NOT bButtonMetalFwOut;
bButtonColorMetalFw := NOT bButtonMetalBwOut;
bButtonColorPlasticBw := NOT bButtonPlasticFwOut;
bButtonColorPlasticFw := NOT bButtonPlasticBwOut;
ELSE
bButtonColorClamp := FALSE;
bButtonColorBarrier := FALSE;
bButtonColorMetalCyl := FALSE;
bButtonColorPlasticCyl := FALSE;
bButtonColorMainBw := FALSE;
bButtonColorMainFw := FALSE;
bButtonColorMetalBw := FALSE;
bButtonColorMetalFw := FALSE;
bButtonColorPlasticBw := FALSE;
bButtonColorPlasticFw := FALSE;
```

```
END_IF

// ========================================================
// Button to control compressed air

IF bMaintenance AND bStarted THEN
IF bButtonComprAirIn THEN
nButtonColorComprAir := cGreen;
ELSE
nButtonColorComprAir := cYellow;
END_IF

bButtonColorComprAir := TRUE;
ELSE
bButtonColorComprAir := FALSE;
END_IF

// ========================================================
```

Method ButtonSignals Declaration

```
METHOD PRIVATE ButtonSignals
VAR_INPUT
END_VAR
```

Method ButtonSignals Implementation

```
// ========================================================
// When machine is in requested state, buttons are turned off again

IF bStartButtonOff THEN
bButtonStartOut := FALSE;
END_IF

IF bStopped THEN
bButtonStopOut := FALSE;
END_IF

IF bReset THEN
bButtonResetOut := FALSE;
END_IF

IF bManual THEN
bButtonManuOut := FALSE;
END_IF

IF bAuto THEN
bButtonAutoOut := FALSE;
END_IF

IF bSemi THEN
bButtonSemiOut := FALSE;
END_IF

IF bMaintenance THEN
bButtonMaintenanceOut := FALSE;
END_IF

// ========================================================
```

Method CylinderMoveBoxes Declaration

```
METHOD PRIVATE CylinderMoveBoxes
VAR_INPUT
END_VAR
```

Method CylinderMoveBoxes Implementation

```
// =========================================================
// Metal cylinder - checking the positions of boxes
// bBoxMetalFree[1] is true, if any box blockades the movement of box 1 on metal belt

aBoxMetalFree[1] := NOT (aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max AND
(aBoxY[1] - aBoxY[2]) <= (cBoxWidth + 1))
AND NOT (aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max AND (aBoxY[1] - aBoxY[3])
<= (cBoxWidth + 1))
AND NOT (aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max AND (aBoxY[1] - aBoxY[4])
<= (cBoxWidth + 1));

aBoxMetalFree[2] := NOT (aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max AND
(aBoxY[2] - aBoxY[1]) <= (cBoxWidth + 1))
AND NOT (aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max AND (aBoxY[2] - aBoxY[3])
<= (cBoxWidth + 1))
AND NOT (aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max AND (aBoxY[2] - aBoxY[4])
<= (cBoxWidth + 1));

aBoxMetalFree[3] := NOT (aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max AND
(aBoxY[3] - aBoxY[1]) <= (cBoxWidth + 1))
AND NOT (aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max AND (aBoxY[3] - aBoxY[2])
<= (cBoxWidth + 1))
AND NOT (aBoxX[4] > cXforBox4OnMetalBelt_min AND aBoxX[4] <= cXforBox4OnMetalBelt_max AND (aBoxY[3] - aBoxY[4])
<= (cBoxWidth + 1));

aBoxMetalFree[4] := NOT (aBoxX[1] > cXforBox1OnMetalBelt_min AND aBoxX[1] <= cXforBox1OnMetalBelt_max AND
(aBoxY[4] - aBoxY[1]) <= (cBoxWidth + 1))
AND NOT (aBoxX[2] > cXforBox2OnMetalBelt_min AND aBoxX[2] <= cXforBox2OnMetalBelt_max AND (aBoxY[4] - aBoxY[2])
<= (cBoxWidth + 1))
AND NOT (aBoxX[3] > cXforBox3OnMetalBelt_min AND aBoxX[3] <= cXforBox3OnMetalBelt_max AND (aBoxY[4] - aBoxY[3])
<= (cBoxWidth + 1));

// =========================================================
// Metal cylinder - boxes motion

FOR nState := 1 TO 4 BY 1 DO
IF bMetalToWork AND aBoxMetalMove[nState] AND aBoxMetalFree[nState] THEN
IF (fYMetal - cMetalPlasticCylMovement) >= (cYMetalPlasticCylEndPos + cMetalPlasticCylMovement) THEN
fYMetal := fYMetal - cMetalPlasticCylMovement;

IF (fYMetal - aBoxY[nState]) <= cYMetalPlasticCylOnMainBelt THEN
aBoxY[nState] := aBoxY[nState] - cMetalPlasticCylMovement;
END_IF

ELSIF fYMetal = (cYMetalPlasticCylEndPos + cMetalPlasticCylMovement) AND (fYMetal - aBoxY[nState]) <=
cYMetalPlasticCylOnMainBelt THEN
aBoxY[nState] := aBoxY[nState] - 2*cMetalPlasticCylMovement;

ELSE
aBoxMetalMove[nState] := FALSE;
END_IF
END_IF
END_FOR

// =========================================================
// =========================================================
// Plastic cylinder - checking the positions of boxes

aBoxPlasticFree[1] := NOT (aBoxX[2] > (cXforBox2OnPlasticBelt_min + 20) AND aBoxX[2] <=
cXforBox2OnPlasticBelt_max AND (aBoxY[1] - aBoxY[2]) <= (cBoxWidth + 1))
AND NOT (aBoxX[3] > (cXforBox3OnPlasticBelt_min + 20) AND aBoxX[3] <= cXforBox3OnPlasticBelt_max AND (aBoxY[1] -
aBoxY[3]) <= (cBoxWidth + 1))
AND NOT (aBoxX[4] > (cXforBox4OnPlasticBelt_min + 20) AND aBoxX[4] <= cXforBox4OnPlasticBelt_max AND (aBoxY[1] -
aBoxY[4]) <= (cBoxWidth + 1));

aBoxPlasticFree[2] := NOT (aBoxX[1] > (cXforBox1OnPlasticBelt_min + 20) AND aBoxX[1] <=
cXforBox1OnPlasticBelt_max AND (aBoxY[2] - aBoxY[1]) <= (cBoxWidth + 1))
AND NOT (aBoxX[3] > (cXforBox3OnPlasticBelt_min + 20) AND aBoxX[3] <= cXforBox3OnPlasticBelt_max AND (aBoxY[2] -
aBoxY[3]) <= (cBoxWidth + 1))
AND NOT (aBoxX[4] > (cXforBox4OnPlasticBelt_min + 20) AND aBoxX[4] <= cXforBox4OnPlasticBelt_max AND (aBoxY[2] -
aBoxY[4]) <= (cBoxWidth + 1));
```

```
aBoxPlasticFree[3] := NOT (aBoxX[1] > (cXforBox1OnPlasticBelt_min + 20) AND aBoxX[1] <=
cXforBox1OnPlasticBelt_max AND (aBoxY[3] - aBoxY[1]) <= (cBoxWidth + 1))
AND NOT (aBoxX[2] > (cXforBox2OnPlasticBelt_min + 20) AND aBoxX[2] <= cXforBox2OnPlasticBelt_max AND (aBoxY[3] -
aBoxY[2]) <= (cBoxWidth + 1))
AND NOT (aBoxX[4] > (cXforBox4OnPlasticBelt_min + 20) AND aBoxX[4] <= cXforBox4OnPlasticBelt_max AND (aBoxY[3] -
aBoxY[4]) <= (cBoxWidth + 1));

aBoxPlasticFree[4] := NOT (aBoxX[1] > (cXforBox1OnPlasticBelt_min + 20) AND aBoxX[1] <=
cXforBox1OnPlasticBelt_max AND (aBoxY[4] - aBoxY[1]) <= (cBoxWidth + 1))
AND NOT (aBoxX[2] > (cXforBox2OnPlasticBelt_min + 20) AND aBoxX[2] <= cXforBox2OnPlasticBelt_max AND (aBoxY[4] -
aBoxY[2]) <= (cBoxWidth + 1))
AND NOT (aBoxX[3] > (cXforBox3OnPlasticBelt_min + 20) AND aBoxX[3] <= cXforBox3OnPlasticBelt_max AND (aBoxY[4] -
aBoxY[3]) <= (cBoxWidth + 1));

// =========================================================
// Plastic cylinder - boxes motion

FOR nState := 1 TO 4 BY 1 DO
IF bPlasticToWork AND aBoxPlasticMove[nState] AND aBoxPlasticFree[nState] THEN
IF (fYPlastic - cMetalPlasticCylMovement) >= (cYMetalPlasticCylEndPos + cMetalPlasticCylMovement) THEN
fYPlastic := fYPlastic - cMetalPlasticCylMovement;

IF (fYPlastic - aBoxY[nState]) <= cYMetalPlasticCylOnMainBelt THEN
aBoxY[nState] := aBoxY[nState] - cMetalPlasticCylMovement;
END_IF

ELSIF fYPlastic = (cYMetalPlasticCylEndPos + cMetalPlasticCylMovement) AND (fYPlastic - aBoxY[nState]) <=
cYMetalPlasticCylOnMainBelt THEN
aBoxY[nState] := aBoxY[nState] - 2*cMetalPlasticCylMovement;

ELSE
aBoxPlasticMove[nState] := FALSE;
END_IF
END_IF
END_FOR

// =========================================================
```

Method ErrorMessages Declaration

```
METHOD PRIVATE ErrorMessages
VAR_INPUT
END_VAR
```

Method ErrorMessages Implementation

```
// =========================================================
// Error messages

nMessages := 0;

// Main axis
IF nMainAxisErrorId = 0 THEN
sMessageMainAxis := '';
ELSE
nMessages := nMessages + 1;
sMessageMainAxis := CONCAT('Error ID of main axis: ', UDINT_TO_STRING(nMainAxisErrorId));
END_IF

// Metal axis
IF nMetalAxisErrorId = 0 THEN
sMessageMetalAxis := '';
ELSE
nMessages := nMessages + 1;
sMessageMetalAxis := CONCAT('Error ID of metal axis: ', UDINT_TO_STRING(nMetalAxisErrorId));
END_IF

// Plastic axis
IF nPlasticAxisErrorId = 0 THEN
```

```
sMessagePlasticAxis := '';
ELSE
nMessages := nMessages + 1;
sMessagePlasticAxis := CONCAT('Error ID of plastic axis: ', UDINT_TO_STRING(nPlasticAxisErrorId));
END_IF

// Clamp cylinder
IF bClampError THEN
nMessages := nMessages + 1;
sMessageClamp := CONCAT('Error of clamp cylinder: ', sClampErrMsg);
ELSE
sMessageClamp := '';
END_IF

// Metal cylinder
IF bMetalCylError THEN
nMessages := nMessages + 1;
sMessageMetalCyl := CONCAT('Error of metal cylinder: ', sMetalErrMsg);
ELSE
sMessageMetalCyl := '';
END_IF

// Plastic cylinder
IF bPlasticCylError THEN
nMessages := nMessages + 1;
sMessagePlasticCyl := CONCAT('Error of plastic cylinder: ', sPlasticErrMsg);
ELSE
sMessagePlasticCyl := '';
END_IF

// =========================================================
```

Method LastBox Declaration

```
METHOD PRIVATE LastBox
VAR_INPUT
END_VAR
```

Method LastBox Implementation

```
// =========================================================
// Checking the last position on main belt

IF (aBoxX[1] >= (cXforAllBoxesStartPosMainBelt + 1*cBoxWidth - 0.05))
AND (aBoxX[1] < (cXforAllBoxesStartPosMainBelt + 2*cBoxWidth - 0.05)) AND (aBoxY[1] = 0) THEN
nLastBox := 1;

ELSIF (aBoxX[2] >= (cXforAllBoxesStartPosMainBelt + 2*cBoxWidth - 0.05))
AND (aBoxX[2] < (cXforAllBoxesStartPosMainBelt + 3*cBoxWidth - 0.05)) AND (aBoxY[2] = 0) THEN
nLastBox := 2;

ELSIF (aBoxX[3] >= (cXforAllBoxesStartPosMainBelt + 3*cBoxWidth - 0.05))
AND (aBoxX[3] < (cXforAllBoxesStartPosMainBelt + 4*cBoxWidth - 0.05)) AND (aBoxY[3] = 0) THEN
nLastBox := 3;

ELSIF (aBoxX[4] >= cXforAllBoxesStartPosMainBelt + 4*cBoxWidth - 0.05)
AND (aBoxX[4] < (cXforAllBoxesStartPosMainBelt + 5*cBoxWidth - 0.05)) AND (aBoxY[4] = 0) THEN
nLastBox := 4;

ELSE
nLastBox := 0;
END_IF

// =========================================================
```

Method Metal_Plastic_Cyl Declaration

```
METHOD PRIVATE Metal_Plastic_Cyl
VAR_INPUT
```

```
END_VAR
```

Method Metal_Plastic_Cyl Implementation

```
// =========================================================
// Moving metal cylinder

IF bMetalToWork THEN
IF NOT (aBoxMetalMove[1] OR aBoxMetalMove[2] OR aBoxMetalMove[3] OR aBoxMetalMove[4]) THEN
IF ((fYMetal - cMetalPlasticCylMovement) >= cYMetalPlasticCylEndPos) THEN
fYMetal := fYMetal - cMetalPlasticCylMovement;
END_IF

ELSE
FOR nState := 1 TO 4 BY 1 DO
IF aBoxMetalMove[nState] THEN
IF ((fYMetal - cMetalPlasticCylMovement) >= (aBoxY[nState] - cYMetalPlasticCylOnMainBelt - 0.1))
AND ((fYMetal - cMetalPlasticCylMovement) >= cYMetalPlasticCylEndPos) THEN
fYMetal := fYMetal - cMetalPlasticCylMovement;
END_IF
END_IF
END_FOR
END_IF

ELSE
IF ((fYMetal + cMetalPlasticCylMovement) < cYMetalPlasticCylStartPos) THEN
fYMetal := fYMetal + cMetalPlasticCylMovement;
ELSE
fYMetal := cYMetalPlasticCylStartPos;
END_IF
END_IF

// =========================================================
// Checking position of boxes related to the metal cylinder

IF (fYMetal < (cYMetalPlasticCylOnMainBelt + 0.1)) AND (fYMetal > (cYMetalPlasticCylOnMainBelt - 0.1)) AND
bMetalToWork THEN
FOR nState := 1 TO 4 BY 1 DO
// box on main belt
IF (aBoxY[nState] >= cYforAllBoxesOnMainBelt) THEN
// box in right position related to the metal cylinder
IF (aBoxX[nState] > (cXforAllBoxesCorrectPushMetalCyl_min + cBoxWidth*nState))
AND (aBoxX[nState] <= (cXforAllBoxesCorrectPushMetalCyl_max + cBoxWidth*nState)) THEN
aBoxMetalMove[nState] := TRUE;
END_IF

// box in wrong position related to the metal cylinder >> accident
IF ((aBoxX[nState] > (cXforAllBoxesIncorrectPushMetalCyl_min + cBoxWidth*nState)
AND aBoxX[nState] <= (cXforAllBoxesCorrectPushMetalCyl_min + cBoxWidth*nState))
OR (aBoxX[nState] > (cXforAllBoxesCorrectPushMetalCyl_max + cBoxWidth*nState)
AND aBoxX[nState] < (cXforAllBoxesIncorrectPushMetalCyl_max + cBoxWidth*nState)))
AND NOT bStopped THEN
bAccident := TRUE;
END_IF
END_IF
END_FOR
END_IF

// =========================================================
// =========================================================
// Moving plastic cylinder

IF bPlasticToWork THEN
IF NOT (aBoxPlasticMove[1] OR aBoxPlasticMove[2] OR aBoxPlasticMove[3] OR aBoxPlasticMove[4]) THEN
IF ((fYPlastic - cMetalPlasticCylMovement) >= cYMetalPlasticCylEndPos) THEN
fYPlastic := fYPlastic - cMetalPlasticCylMovement;
END_IF

ELSE
FOR nState := 1 TO 4 BY 1 DO
```

```
IF aBoxPlasticMove[nState] THEN
IF ((fYPlastic - cMetalPlasticCylMovement) >= (aBoxY[nState] - cYMetalPlasticCylOnMainBelt - 0.1))
AND ((fYPlastic - cMetalPlasticCylMovement) >= cYMetalPlasticCylEndPos) THEN
fYPlastic := fYPlastic - cMetalPlasticCylMovement;
END_IF
END_IF
END_FOR
END_IF

ELSE
IF ((fYPlastic + cMetalPlasticCylMovement) < cYMetalPlasticCylStartPos) THEN
fYPlastic := fYPlastic + cMetalPlasticCylMovement;
ELSE
fYPlastic := cYMetalPlasticCylStartPos;
END_IF
END_IF

// ========================================================
// Checking position of boxes related to the plastic cylinder

IF ((aBoxX[1] > (cXforAllBoxesLastPossiblePosMainBelt + 1*cBoxWidth))
OR (aBoxX[2] > (cXforAllBoxesLastPossiblePosMainBelt + 2*cBoxWidth))
OR (aBoxX[3] > (cXforAllBoxesLastPossiblePosMainBelt + 3*cBoxWidth))
OR (aBoxX[4] > (cXforAllBoxesLastPossiblePosMainBelt + 4*cBoxWidth)))
AND NOT bStopped THEN
bAccident := TRUE;
END_IF

IF (fYPlastic < (cYMetalPlasticCylOnMainBelt + 0.1)) AND (fYPlastic > (cYMetalPlasticCylOnMainBelt - 0.1)) AND
bPlasticToWork THEN
FOR nState := 1 TO 4 BY 1 DO
// box on main belt
IF aBoxY[nState] >= cYforAllBoxesOnMainBelt THEN
// box in right position related to the plastic cylinder
IF (aBoxX[nState] > (cXforAllBoxesCorrectPushPlasticCyl_min + cBoxWidth*nState))
AND (aBoxX[nState] <= (cXforAllBoxesCorrectPushPlasticCyl_max + cBoxWidth*nState)) THEN
aBoxPlasticMove[nState] := TRUE;
END_IF

// box in wrong position related to the plastic cylinder >> accident
IF (aBoxX[nState] > (cXforAllBoxesIncorrectPushPlasticCyl_min + cBoxWidth*nState))
AND (aBoxX[nState] <= (cXforAllBoxesCorrectPushPlasticCyl_min + cBoxWidth*nState))
AND NOT bStopped THEN
bAccident := TRUE;
END_IF
END_IF
END_FOR
END_IF

// ========================================================
```

Method Reset Declaration

```
METHOD PRIVATE Reset
VAR_INPUT
END_VAR
```

Method Reset Implementation

```
// ========================================================
// Resetting visu

// Cylinder buttons
bButtonClampToWorkOut := FALSE;
bButtonBarrierToWorkOut := FALSE;
bButtonMetalToWorkOut := FALSE;
bButtonPlasticToWorkOut := FALSE;

// Axis buttons
bButtonMainBwOut := FALSE;
```

```
bButtonMainFwOut := FALSE;
bButtonMetalBwOut := FALSE;
bButtonMetalFwOut := FALSE;
bButtonPlasticBwOut := FALSE;
bButtonPlasticFwOut := FALSE;

// State and mode buttons
bButtonStartOut := FALSE;
bButtonStopOut := FALSE;
bButtonResetOut := FALSE;
bButtonComprAirOut := FALSE;
bButtonManuOut := FALSE;
bButtonAutoOut := FALSE;
bButtonSemiOut := FALSE;
bButtonMaintenanceOut := FALSE;

// Position of boxes and cylinders
FOR nState := 1 TO 4 BY 1 DO
aBoxX[nState] := 0;
aBoxY[nState] := 0;

aBoxMetalMove[nState] := FALSE;
aBoxPlasticMove[nState] := FALSE;
aBoxMainBelt[nState] := FALSE;
aBoxMetalBelt[nState] := FALSE;
aBoxPlasticBelt[nState] := FALSE;
END_FOR

bAccident := FALSE;

// ========================================================
```