rexroth
A Bosch Company

ctrlX PackML Template Project

PLC template project for PackML state machine implementation

Edition 1

Table of contents

1. About

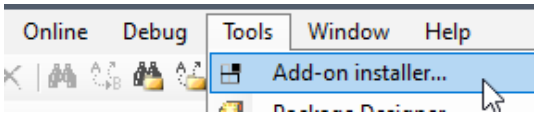
Edition	Date	Comment
1	2025-05-05	Template Version 3.6.0.0

2. Disclaimer

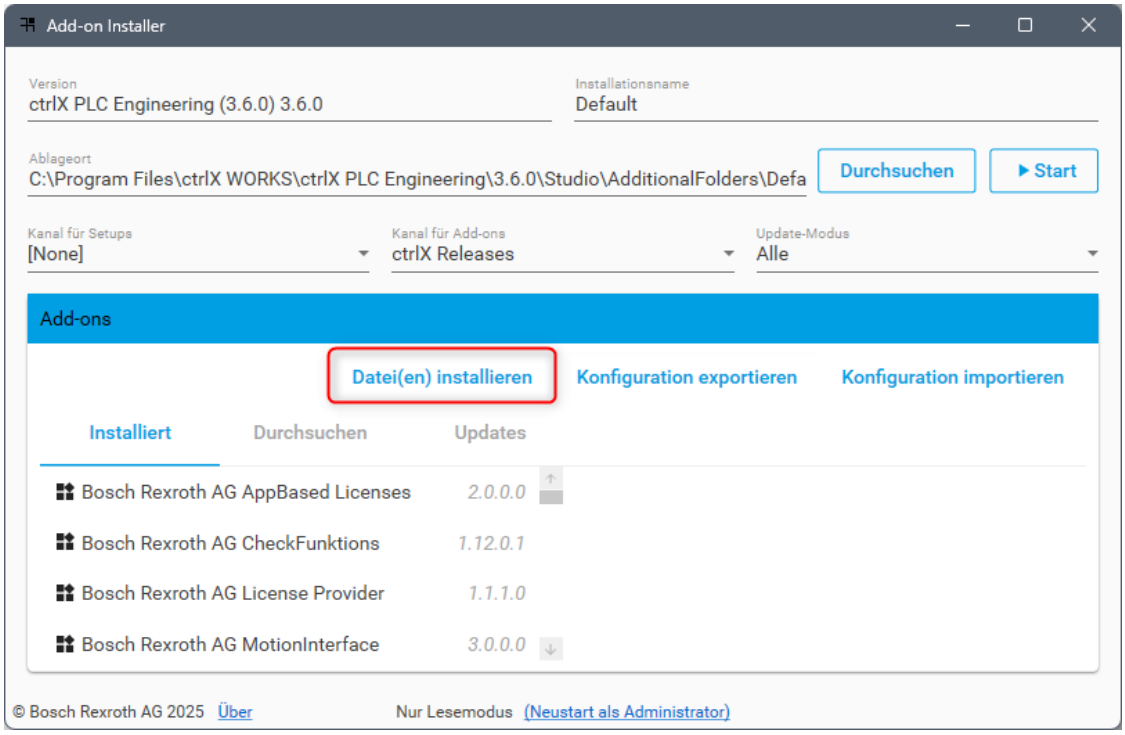
Use of this software is permitted only under the terms and conditions defined in article *Terms and Conditions for the Provision of Products of Bosch Rexroth AG Free of Charge*, included in the software installation package. The software is licensed under the MIT License.

3. Installation Guide

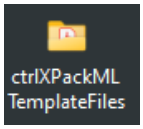
The PackML Template will be provided as a so called Add-On package. To install this Add-On, start the Add-On Installer in ctrlX PLC Engineering.



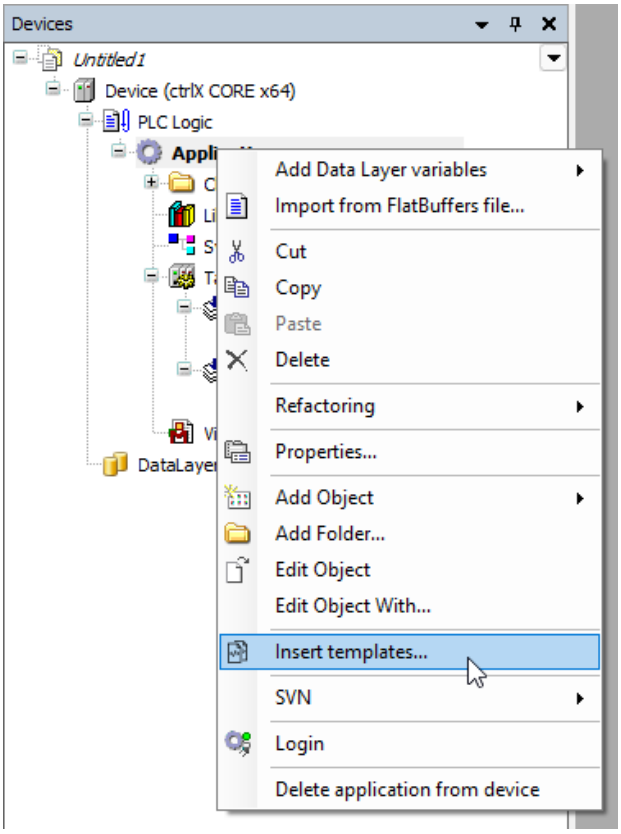
Install package file.



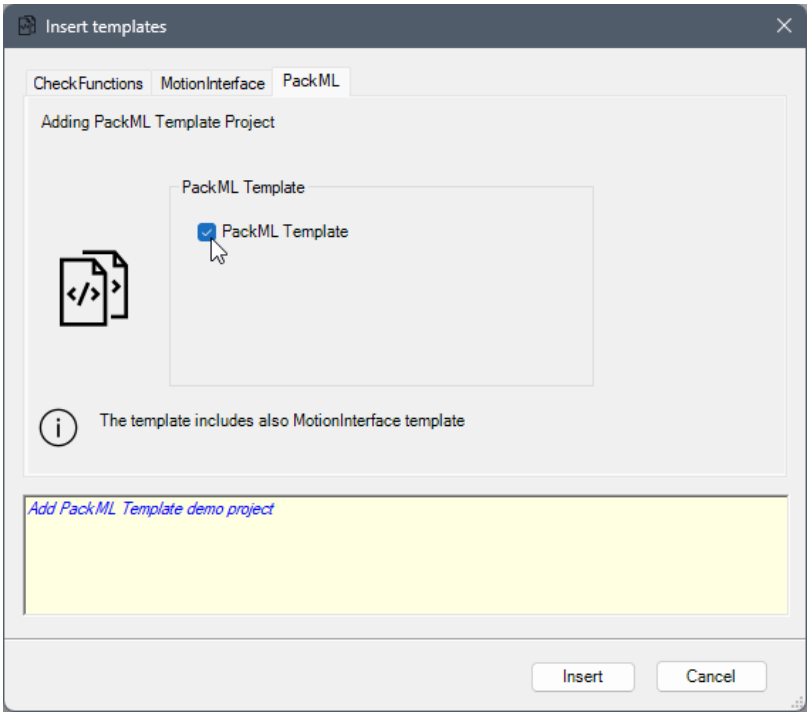
During installation a folder with additional files will be created on the desktop.



ctrlX PLC Engineering needs to be closed during installation. After restart of ctrlX PLC Engineering, create a new project and select “Insert template” from the Application node.



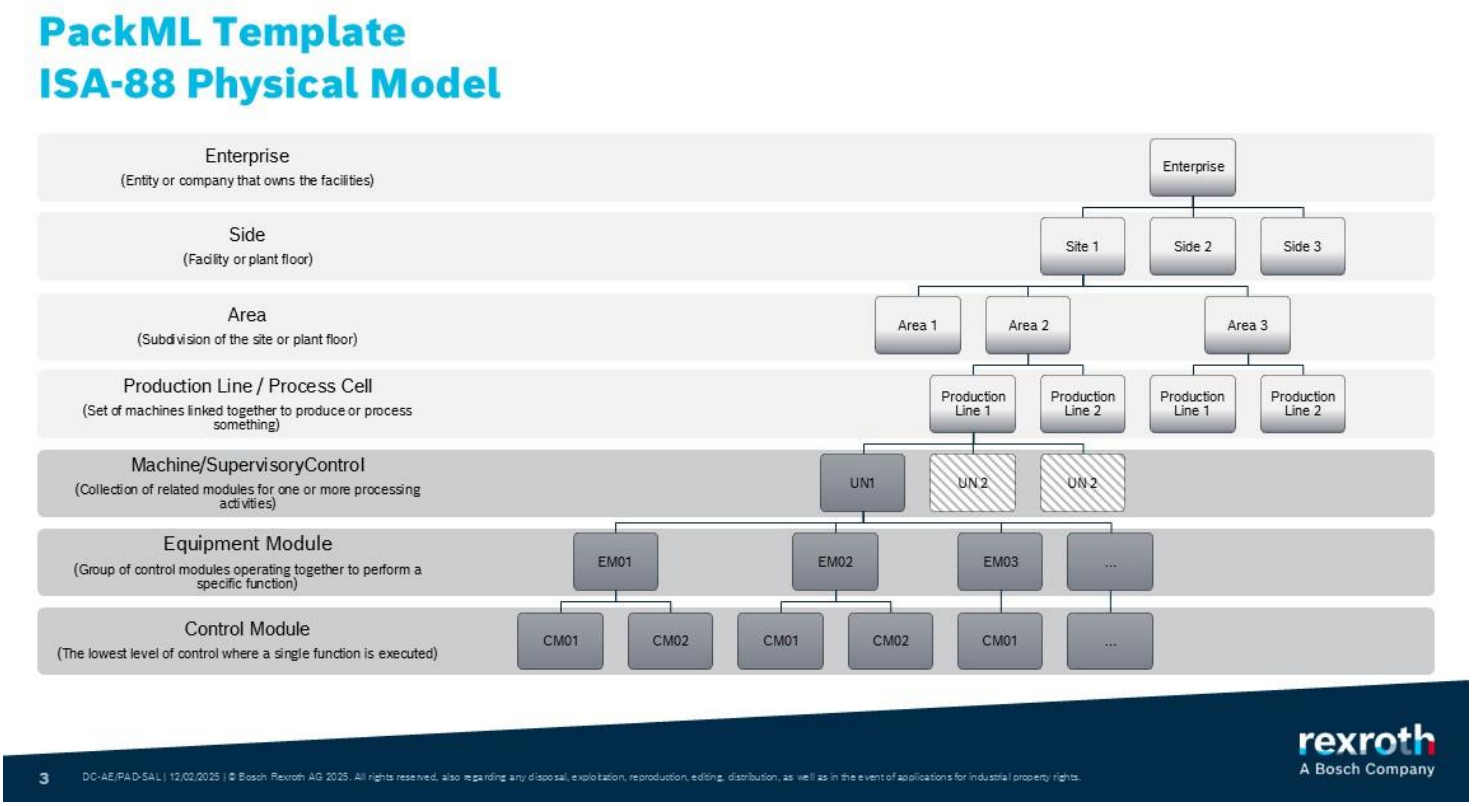
Select PackML Template and press Insert.



Now, the template code will be imported to your project.

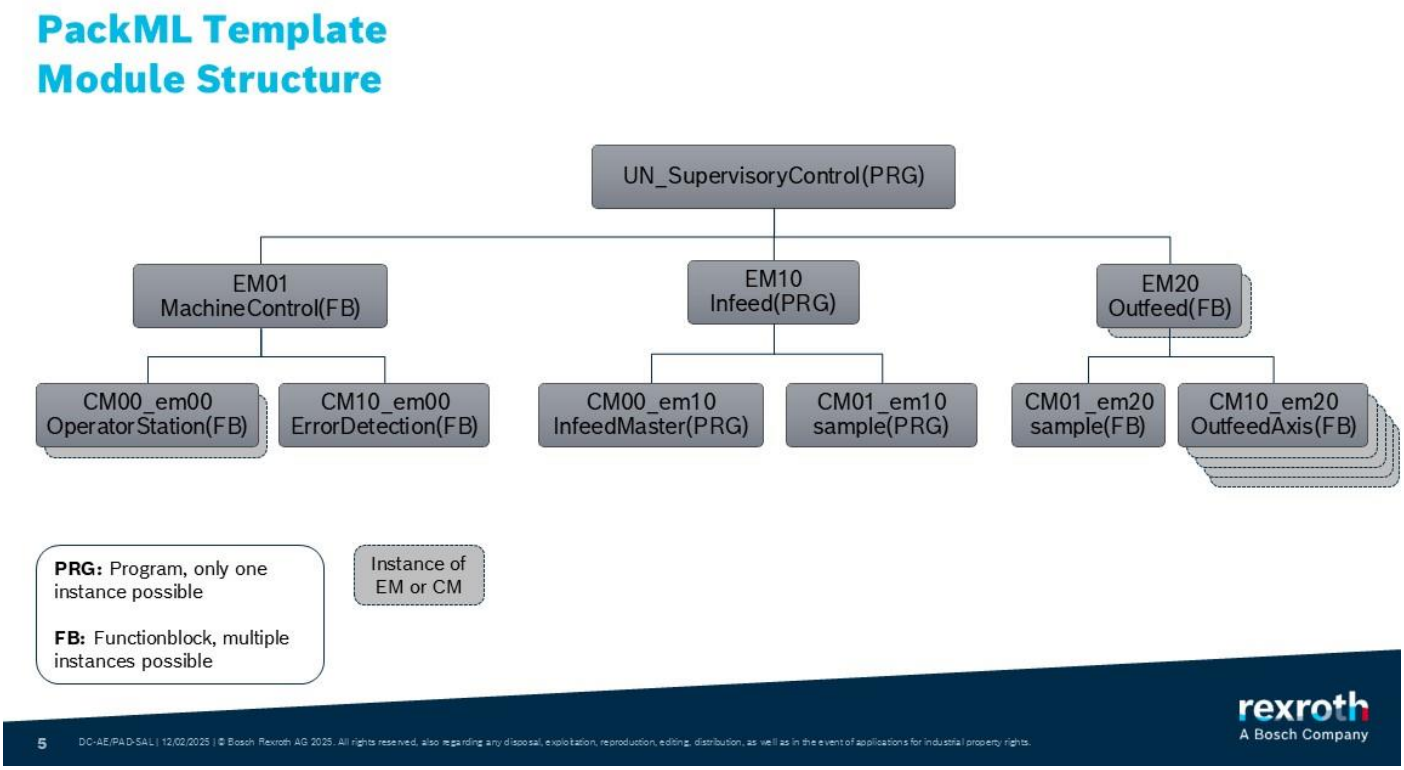
4. Introduction and Overview

ISA-88, the larger batch industry standard upon which PackML is based, defines a so-called physical model, which describes the hierarchy of physical assets of a commercial enterprise.



5. Project structure

This Template brings a small example how to use CXA_PackML_Toolkit.libray. Unit will be abbreviated with UN, Equipment Module will be abbreviated with EM and Command Module will be abbreviated with CM.



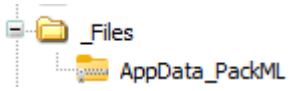
5.1. Prerequisites

This template project is ready to run on a virtual ctrlX CORE. You just need at least version 3.6 of the Motion and PLC App. Licenses are not required for a virtual control.

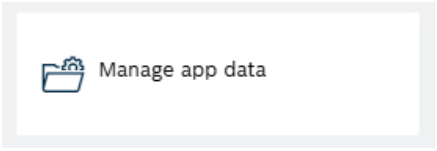
To run this template project as it is on a real ctrlX CORE hardware, you need the following licences:

- Base Licence "Motion Standard 10 Axes"
- Option "Motion Electronic Gear"
- ctrlX PLC Basic (02VRS+)
- ctrlX PLC Standard (02VRS+ / add-on)

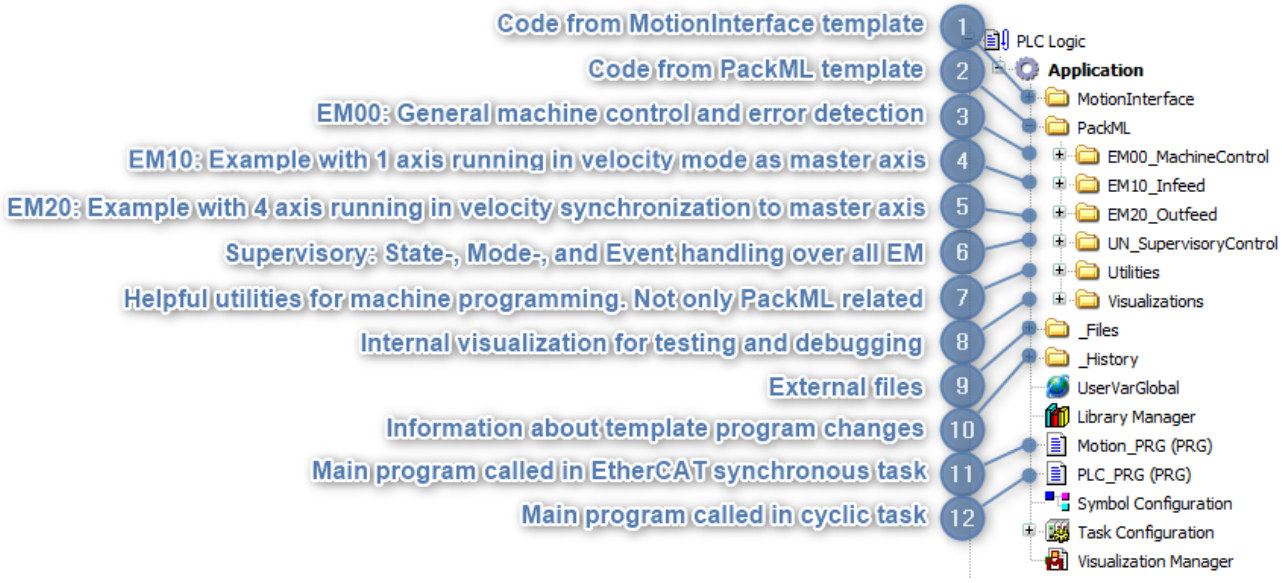
The axes configuration, which is used in this project, is included as an AppData Archive.



You can import this to your virtual or real control with the “manage app data” dialog.



5.2. General structure in PLC Code

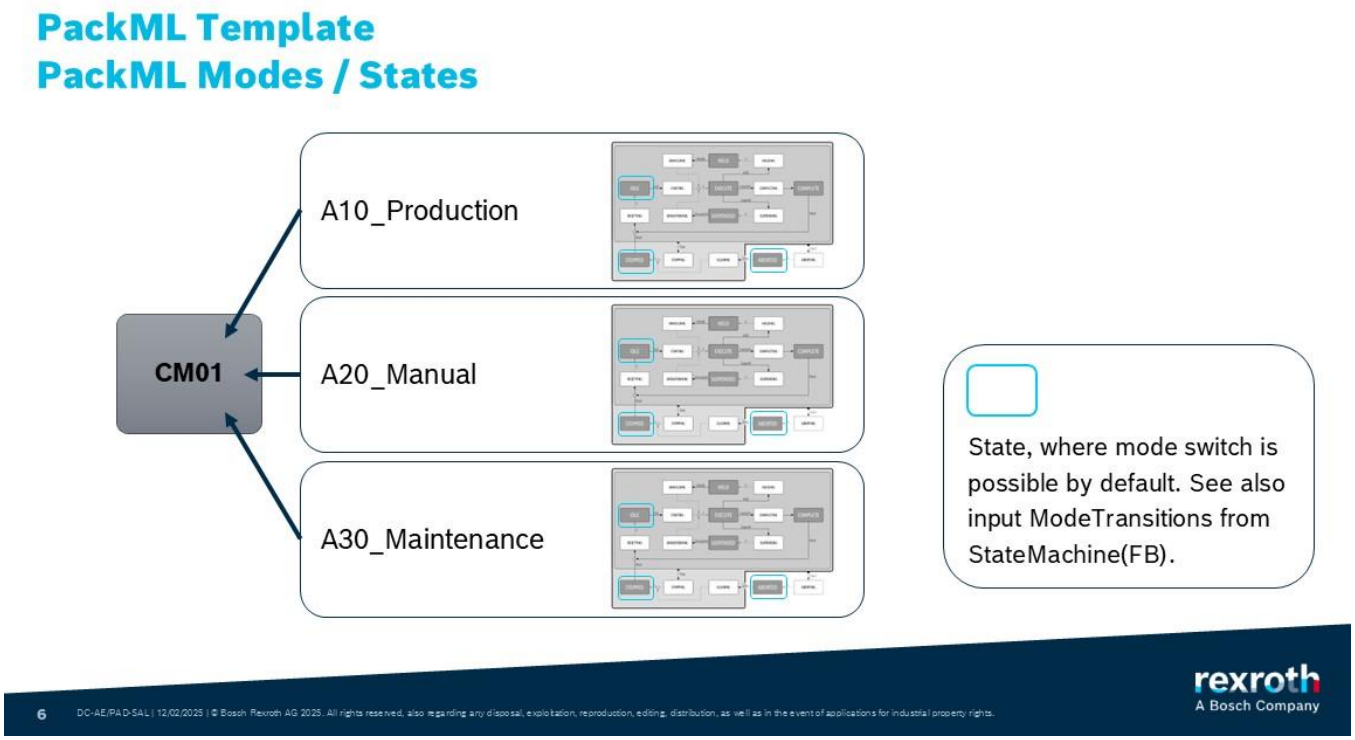


Project is based on the CXA_PackML.library. About detailed information, please refer to library documentation (ctrlX_AUTOMATION_PackML_Edition_xx). This documentation tries to explain the general usage of the PackML_Toolkit FBs.

5.3. PackML State Machine

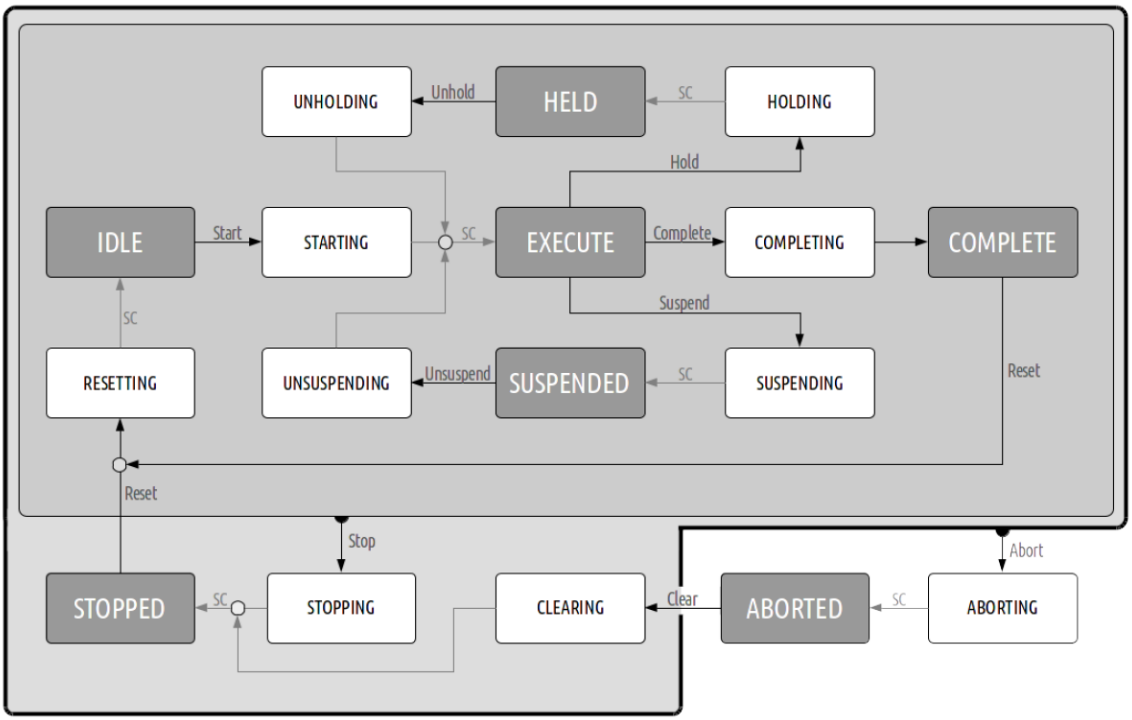
The PackML template project differs between modes and states. It implements 3 different modes: Producing, Maintenance and Manual, which can be extended up to 32 different modes.

Enumeration	Mode
0	Undefined
1	Production
2	Maintenance
3	Manual



Every mode implements the same state machine, but not all states have to be implemented in every mode. It is possible to configure rules in which state it is enabled to switch between modes.

For more information refer to the PackML implementation guide (https://www.omac.org/packml).



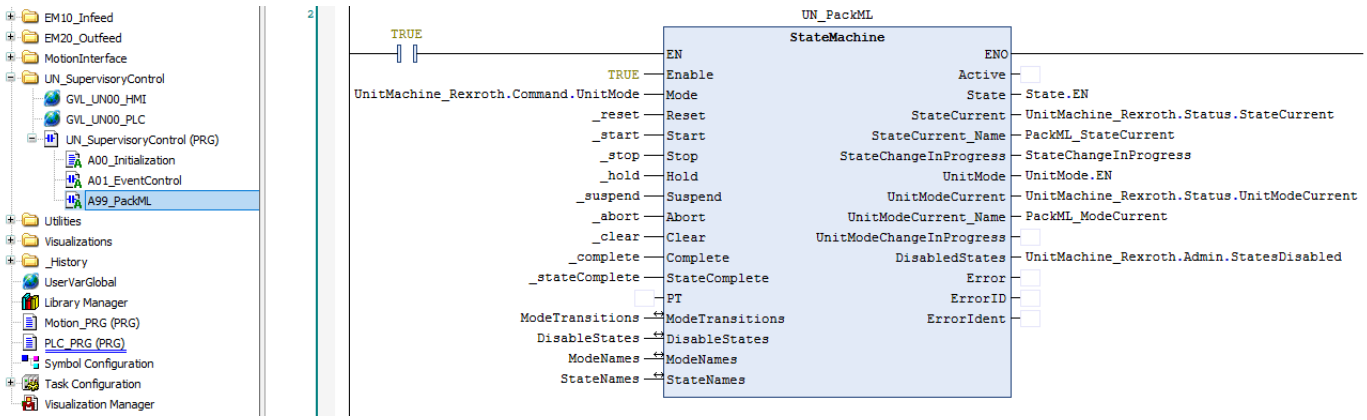
The state model consists of 17 distinct states and 11 transition commands, where a particular transition initiates action on the state machine only if the associated predecessor is active. For example, if the state machine is in IDLE state, then the Start command will force a transition of the state model from IDLE state to STARTING state. If a state other than IDLE is active, then the Start command has no effect.

Enumeration	Distinct State
1	CLEARING
2	STOPPED
3	STARTING
4	IDLE
5	SUSPENDED
6	EXECUTE
7	STOPPING
8	ABORTING
9	ABORTED
10	HOLDING
11	HELD
12	UNHOLDING
13	SUSPENDING
14	UNSUSPENDING
15	RESETTING
16	COMPLETING
17	COMPLETE

Transition Commands
Clear
Reset
Start
Hold
Unhold
Suspend
Unsuspend
Complete
Stop
Abort
StateComplete (SC)

State	Description
EXECUTE	Acting State - The unit/machine is in a stable acting state - unit/machine is producing.
STOPPED IDLE COMPLETE	Wait State – A stable state used to identify that a unit/machine has achieved a defined set of conditions. In such a state the unit/machine is holding or maintaining a status until a command causes a transition to an Acting state. The unit/machine is powered and stationary.
RESETTING STARTING SUSPENDING UNSUSPENDING COMPLETING HOLDING UNHOLDING ABORTING CLEARING STOPPING	Acting State – A state which represents some processing activity, for example ramping up speed. It implies the single or repeated execution of processing steps in a logical order, for a finite time or until a specific condition has been reached, for example within the Starting state the quality and validity of the received data is checked, before ramping up speed for execution.
HELD ABORTED	Wait state – A state which represents an error state on the Unit which will generate an alarm or warning. In this state the unit/machine is not producing, until the operator made a transition to the EXECUTING state. The state holds the unit/machine operations while material blockage are cleared, or safe correction of an equipment fault before the production may be resumed.
SUSPENDED	Wait State – In this state the unit/machine is not producing any products. It will either stop running or continue to cycle without producing until external process conditions return to normal, at which time the SUSPENDED state will transition to the UNSUSPENDING state, typically without any operator intervention.

The implementation of this state machine is realized in the StateMachine(FB), which is implemented once in the UN.



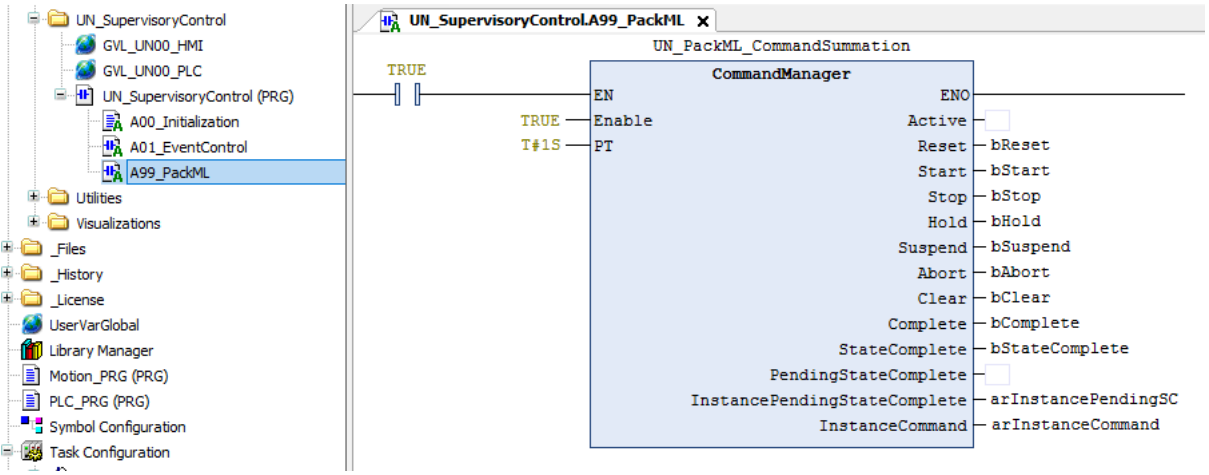
With four input structures, it is possible to adapt the behaviour in a simple way.

Input	Description
ModeTransitions	To allow a mode transition in a given state, the bit associated to this state must be true for both ModeTransitions[current mode] and ModeTransitions[target mode]. For example, if the machine is currently producing mode (UnitModeCurrent = 1), aborted state (StateCurrent = 9), then the machine may be transitioned to manual mode (UnitModeCurrent = 3) provided: ModeTransitions[1] = bxxxx xx1x xxxx xxxx, ModeTransitions[3] = bxxxx xx1x xxxx xxxx
DisableStates	Defines for each mode the states which are disabled. State status is defined bitwise, with a bit value of one indicating that the state is disabled. For example, if SUSPENDING, SUSPENDED, UNSUSPENDING, COMPLETING and COMPLETE is disabled in Maintenance mode: DisableStates[2] = b0000 0000 0000 00011 0110 0000 0010 0000
ModeNames	User-definable mode names.
StateNames	User-definable mode names.

5.4. CommandManager, CommandClient

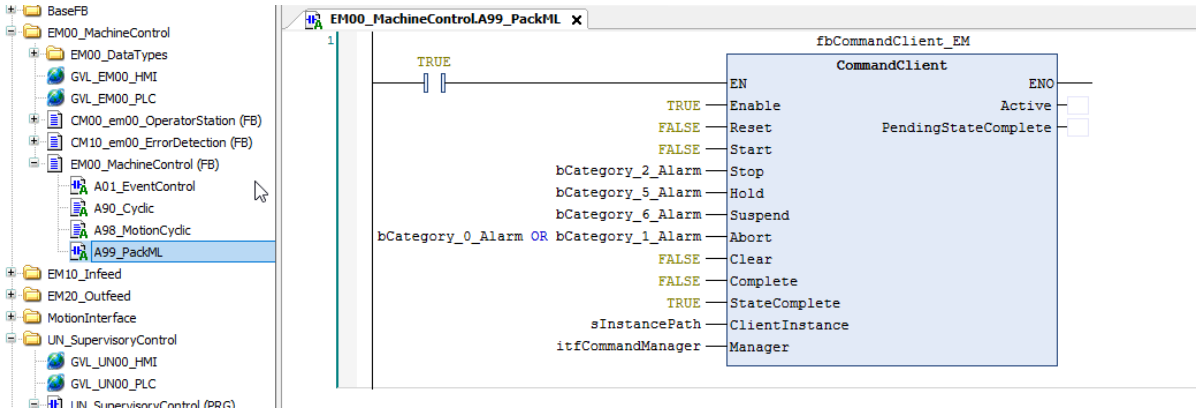
The purpose of the CommandManager(FB) is, to collect all transition commands (e.g. Start, Abort) from all underlying EM and CM.

Call of CommandManager(FB) in UN

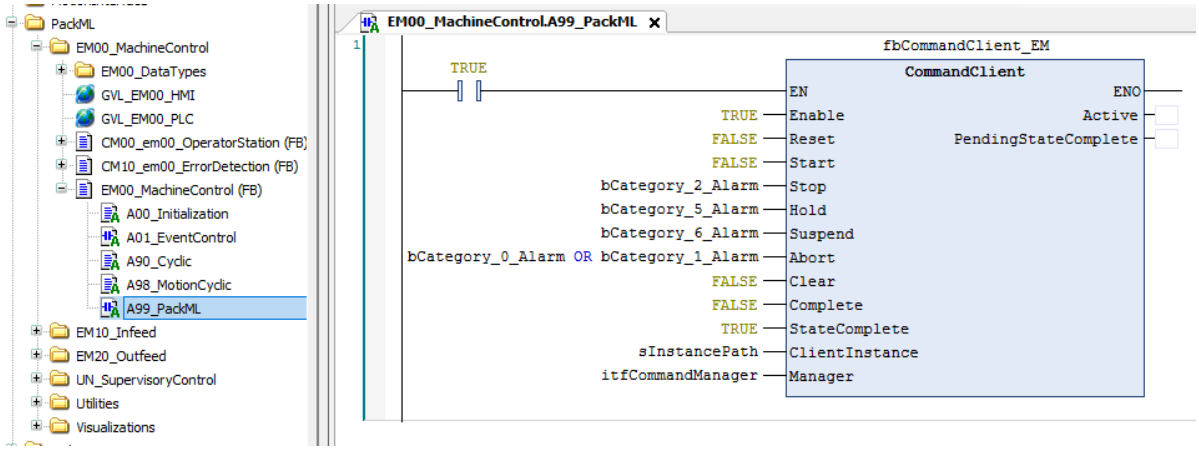


The CommandClient(FB) provides inputs to request a transition command in every EM or CM.

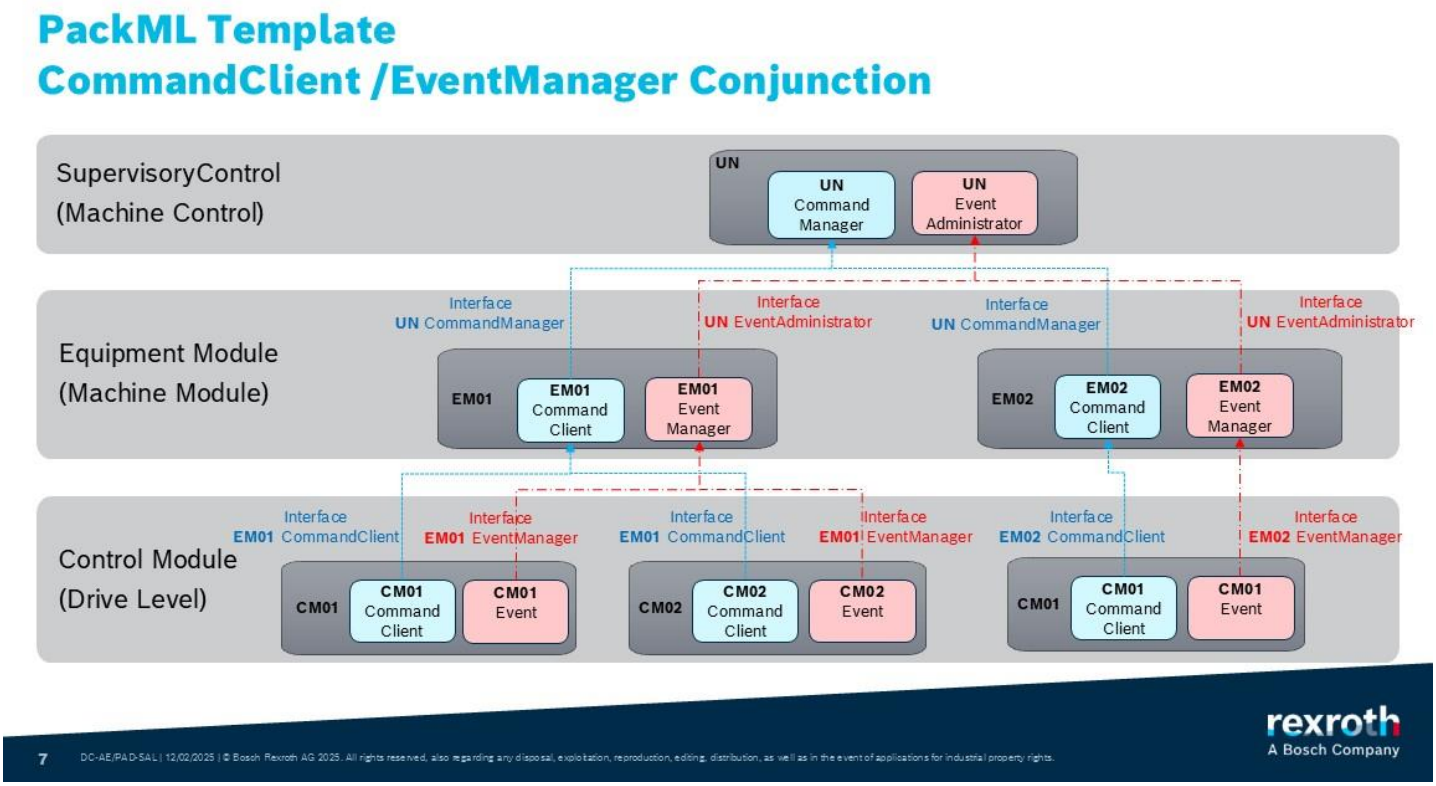
Call of CommandClient(FB) in EM



Call of CommandClient(FB) in CM



The CommandClient of the CM will register to the CommandClient of the EM and the CommandClient of the EM will register to the CommandManager of the UN. This is realized with the interface input “Manager”.

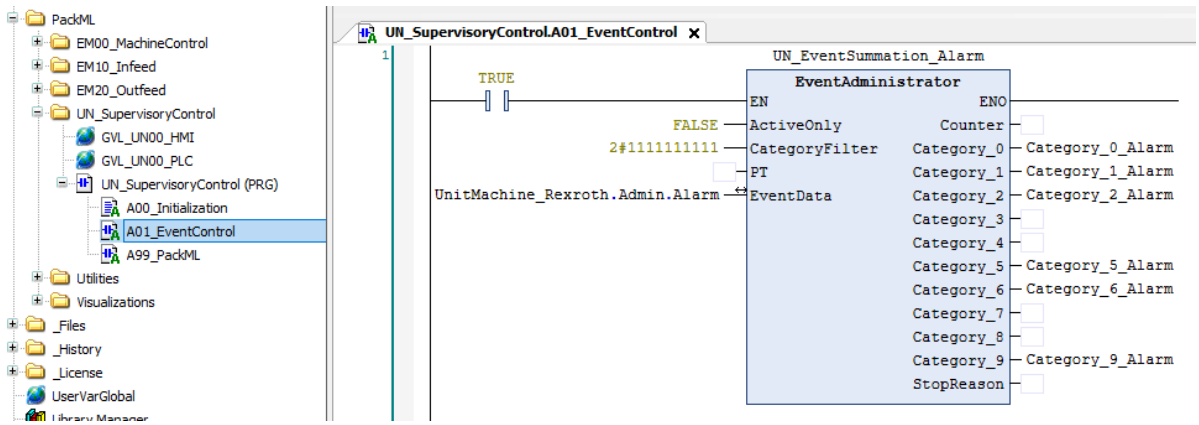


The advantage of this structure is shown in its flexibility. You can add/remove multiple CM without changing the CommandClient call in EM. You can also add/remove multiple EM without changing the CommandManager call in UN.

5.5. Event, EventManager, EventAdministrator

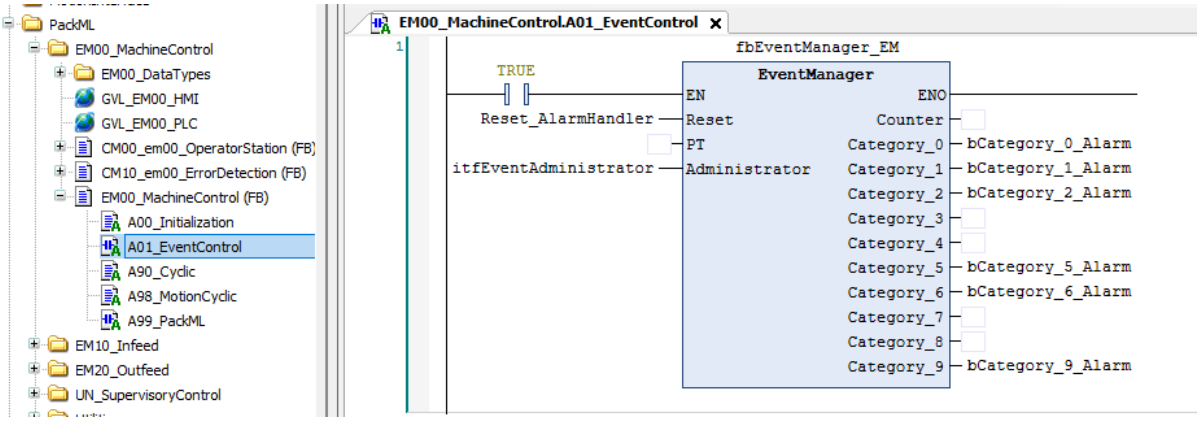
An Event could be an alarm, an error, a warning or just a message. The purpose of the EventAdministrator(FB) is to collect all Events over all EM.

Call of EventAdministrator(FB) in UN.



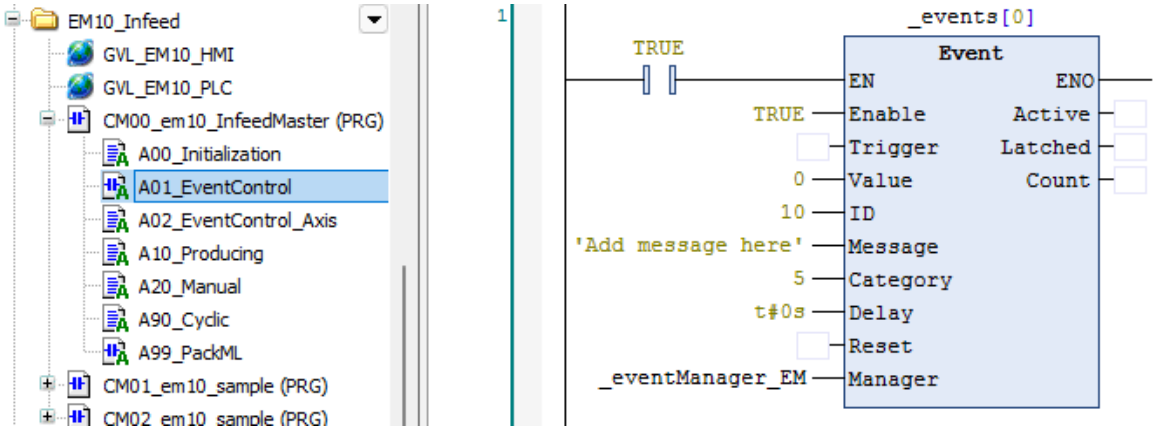
Inside an EM, the EventManager(FB) collects all Events over all CM.

Call of EventManager(FB) in EM.



Inside a CM the Event(FB) can trigger module specific events

Call of Event(FB) in CM



The Event(FB) of the CM will register to the EventManager(FB) of the EM and the EventManager(FB) of the EM will register to the EventAdministrator(FB) of the UN. This is realized with the interface inputs “Manager” and “Administrator”.

New implementations of Event(FB) will automatically register to EventAdministrator(FB). No code adjustments are necessary.

See also picture in 5.4 CommandManager, CommandClient.

5.6. Call Tree

Task	POU	Description
MainTask	PLC_PRG	Cyclic PLC Task with 200ms (default value virtual ctrlX CORE) cycle time. Contains PackML Statemachine, Eventhandling, MotionInterface, ImcInterface. Call of UN, EM, CM main POUs.
MotionTask	Motion_PRG	Event triggered Task by ctrlX scheduler. Synchronous to motion value generation. Call of motion synchronous actions in UN, EM and CM. For example, process controller, PLS and so on.

6. How To

6.1. Add new Equipment Module

First you need to decide whether you want to implement the EM as a PRG or FB.
PRG:

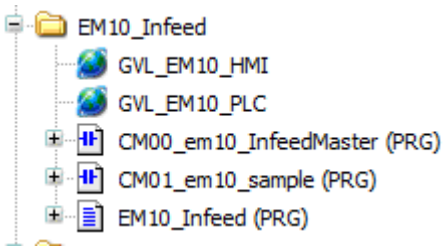
- No instances possible, just use once.
- No flexibility with inputs and outputs
- Simple implementation with global variables
- Less object oriented programming

FB:

- Multiple instances possible
- Use same code for several physical modules (axes, I/Os)
- More flexibility with inputs and outputs
- More object oriented programming

6.1.1. PRG

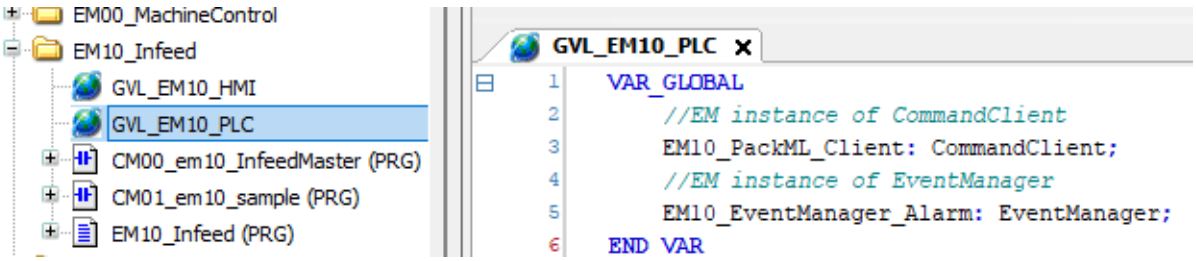
The implementation of a single instance EM as PRG is shown in EM10_Infeed



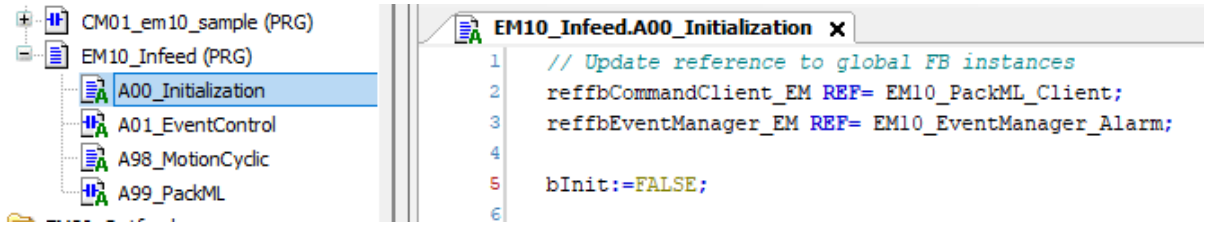
To add a new EM, follow these steps:

- copy the folder EM10
- adapt the names of the POUs

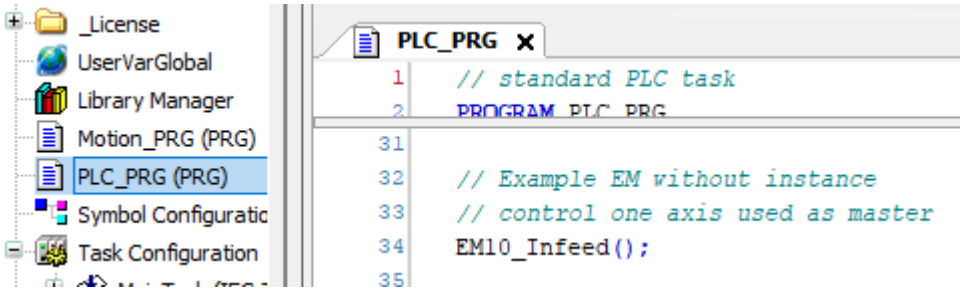
- adapt instance name in GVL_xxx_PLC:



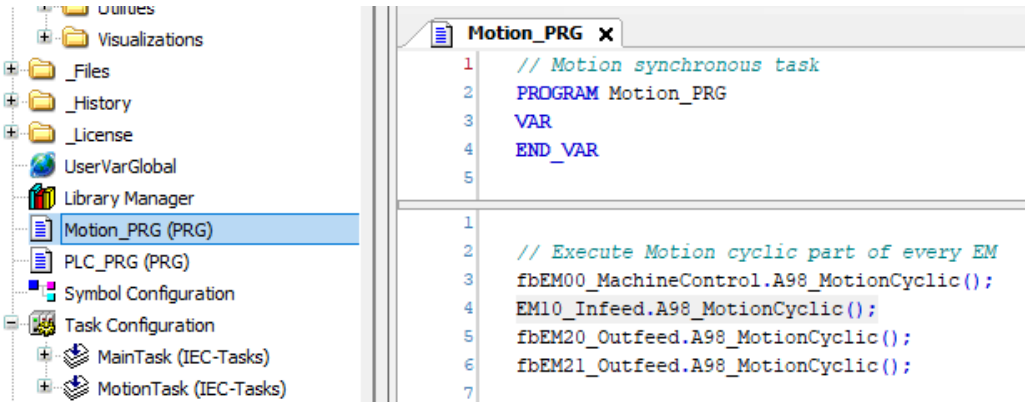
- adapt A00_Initialization



- add call of EM(PRG) in PLC_PRG

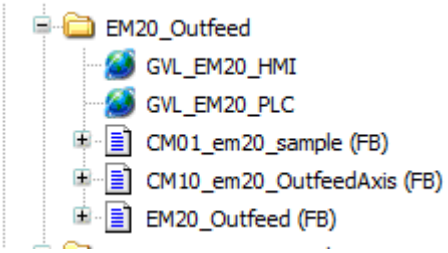


- add motion cyclic action to EM(PRG) and call to Motion_PRG, if necessary.



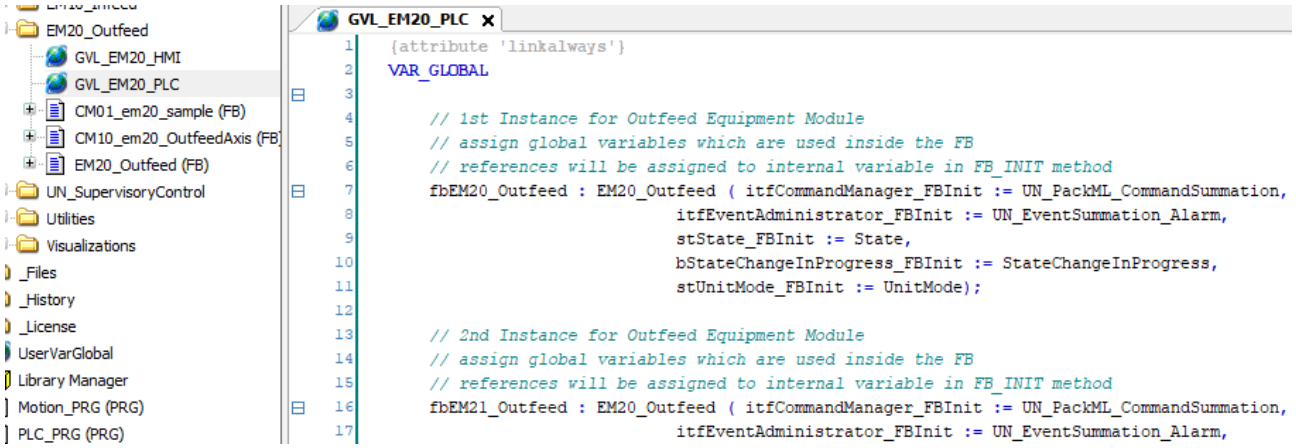
6.1.2. FB

The implementation of multiple instances EM as FB is shown in EM20_Outfeed

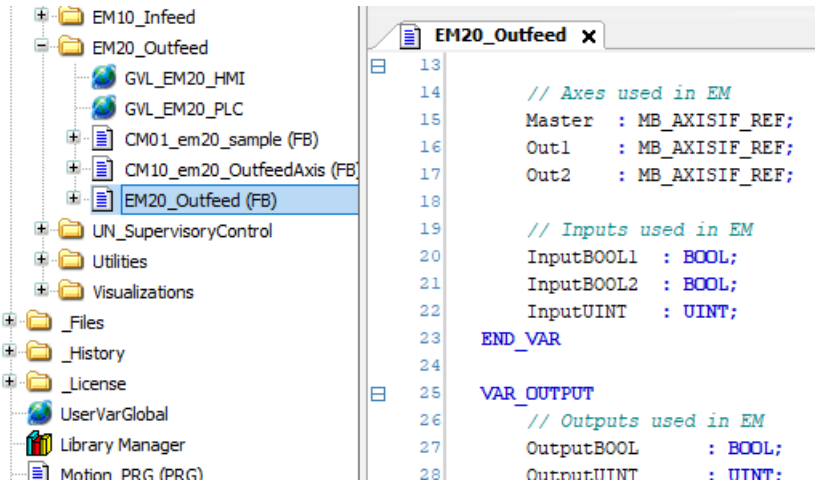


To add a new EM, follow these steps:

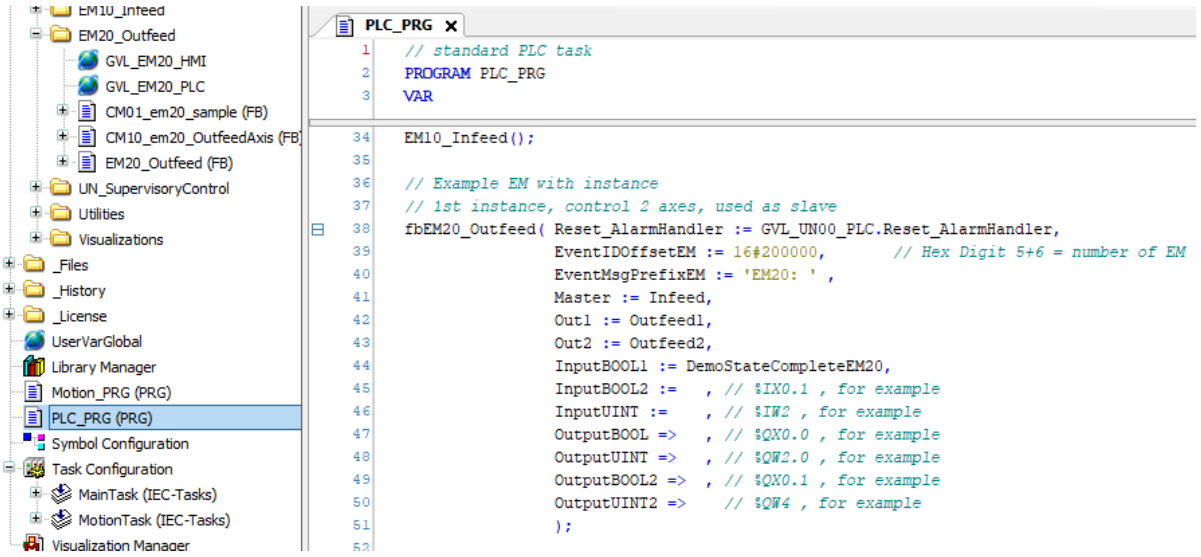
- copy the folder EM20
- adapt the names of the POUs
- add instance declaration to GVL_EMxx_PLC



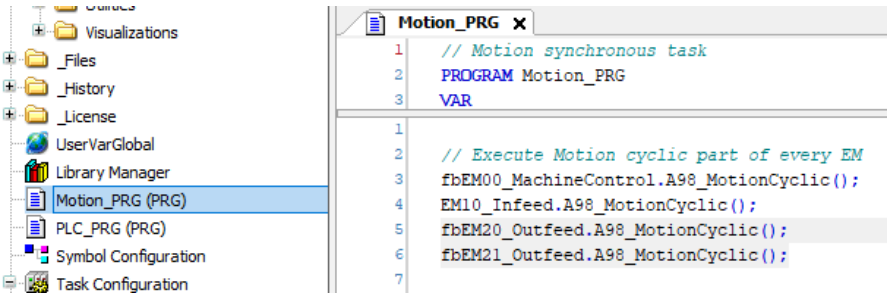
- adapt FB interface to application needs, axes, I/Os and so on.



- add call of EM(FB) in PLC_PRG



- add call of Motion cyclic action to Motion_PRG



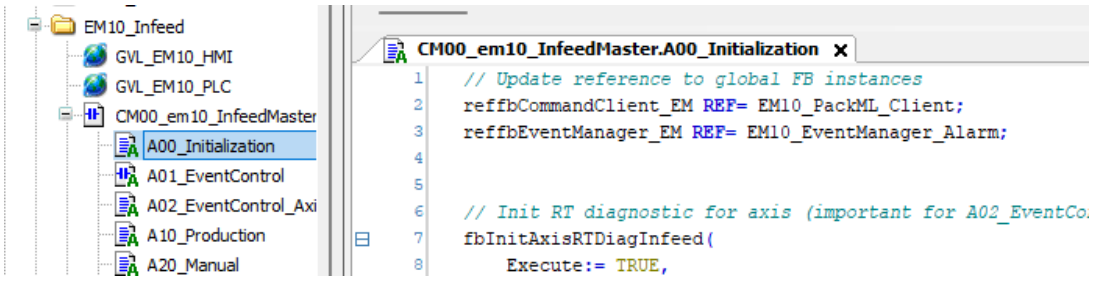
6.2. Add new Control Module

The differences between implementation of a CM as PRG or FB are similar to EM. Please refer to “6.1 Add new Equipment Module”.

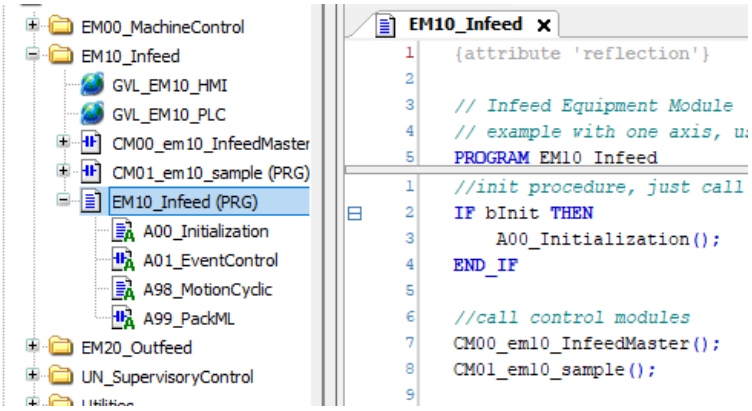
6.2.1. PRG

To add a new CM, follow these steps:

- copy CM(PRG) from EM10_Infeed. E.g. CM_00_em10 InfeedMaster(PRG)
- adapt name of PRG
- adapt A00_Initialization to corresponding EM



- call CM in EM

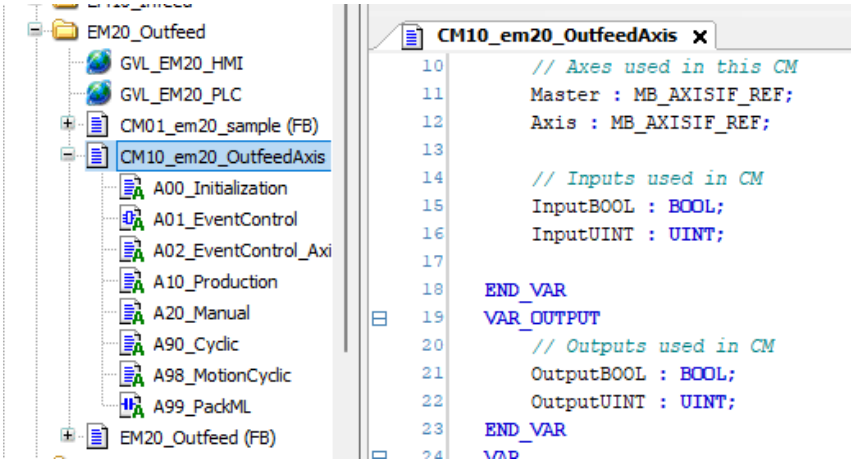


- Add motion cyclic action to CM(PRG) and add call to motion cyclic action of EM, if necessary.

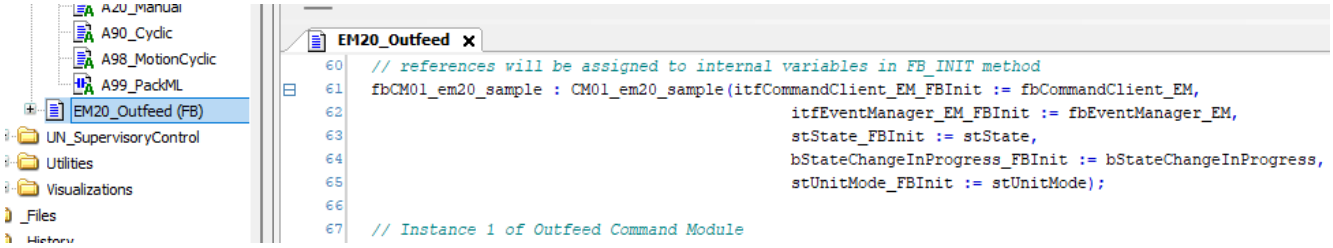
6.2.2. FB

To add a new CM, follow these steps:

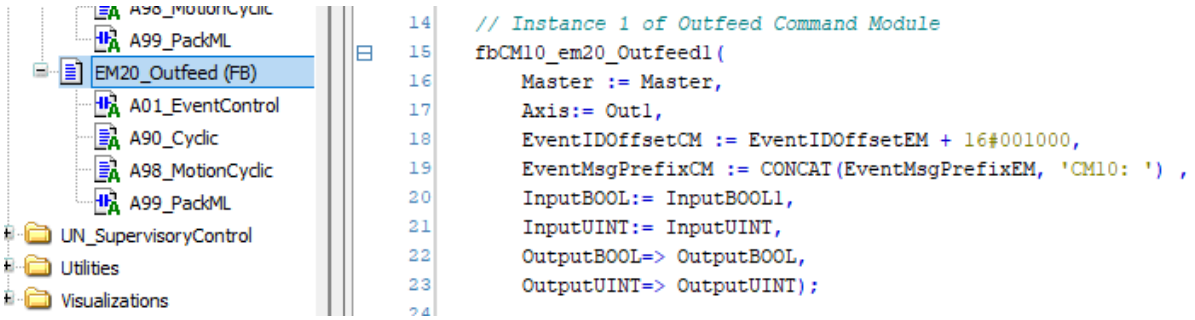
- copy CM(FB) from EM20_Outfeed
- adapt FB name
- adapt FB interface to application needs; axes, I/Os and so on



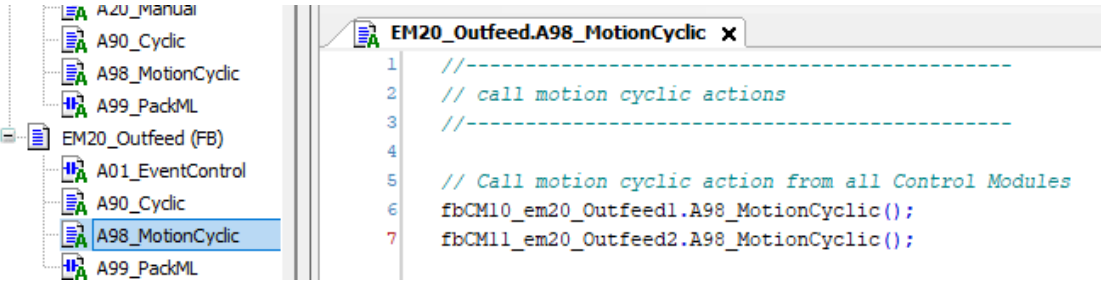
- add declaration of instance to EM



- add call to EM



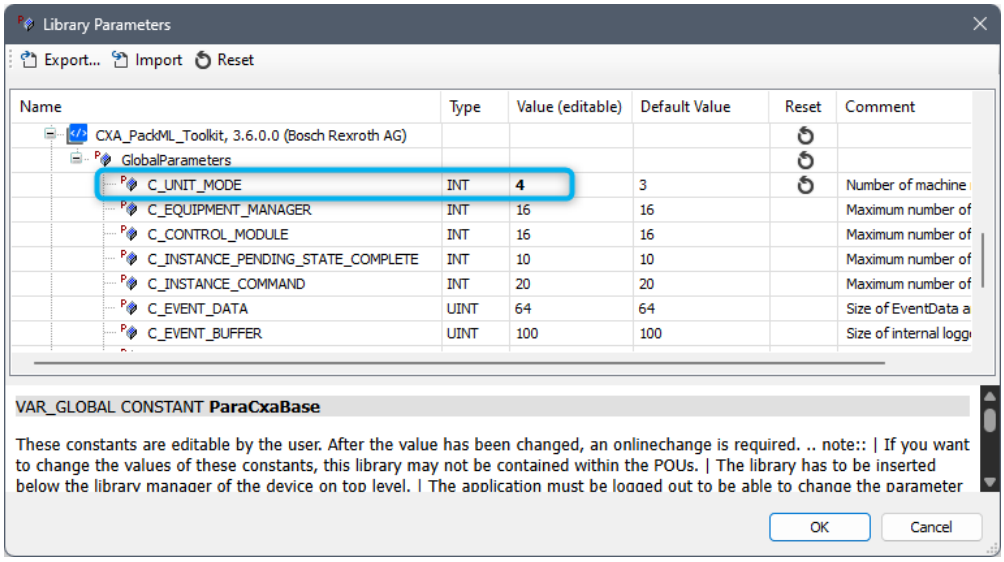
- add motion cyclic call to EM



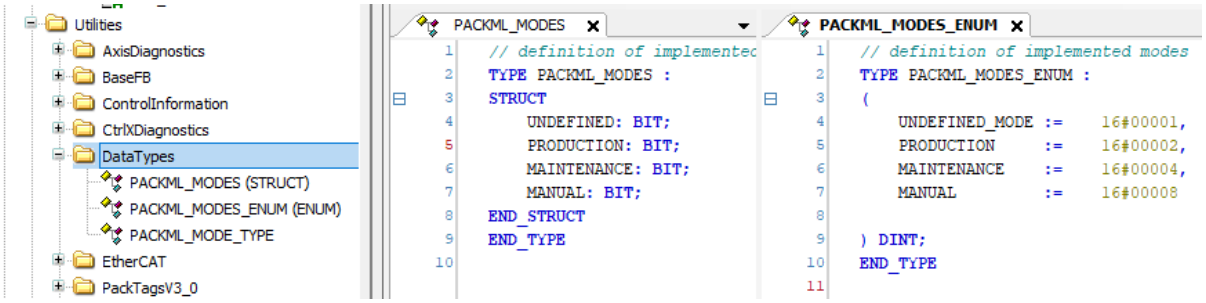
6.3. Add new mode

To add a new operation mode to your project, follow these steps:

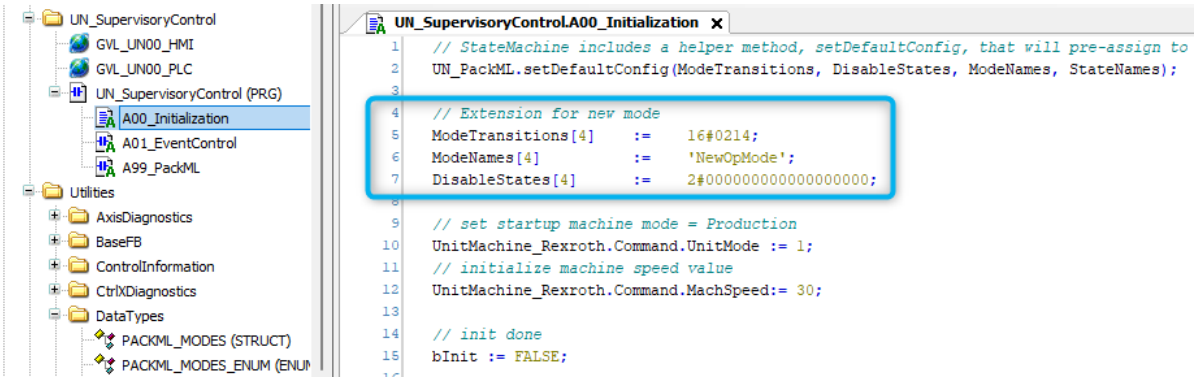
- Increase number of modes variable in library parameters



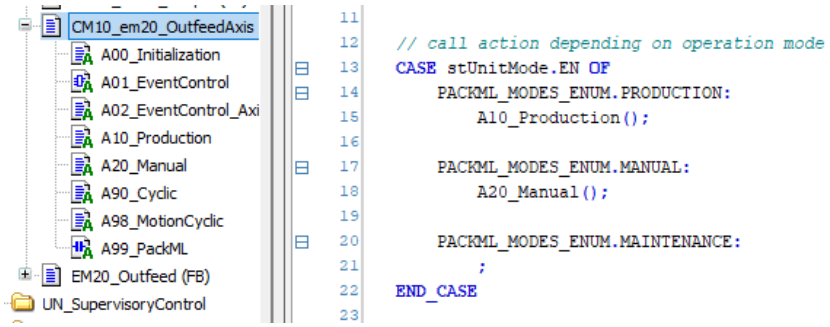
- Extend DataTypes PACKML_MODES and PACKML_MODES_ENUM



- Extend StateMachine configuration variables on your own.



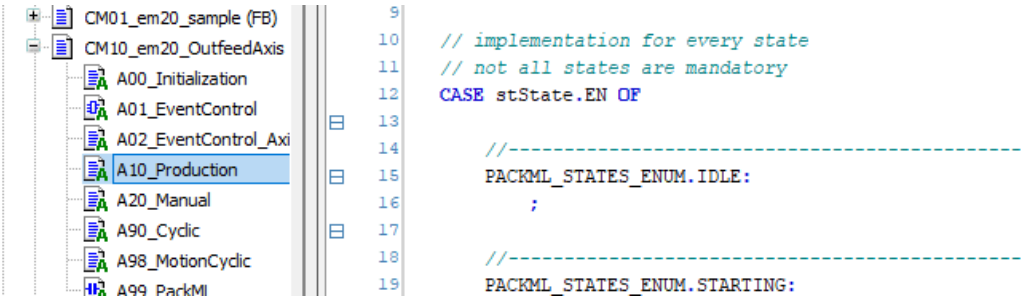
6.4. Implement mode in CM



Inside a CM(FB) the actual mode is represented by the variable ‘stUnitMode’. This variable is reference to the global variable ‘UnitMode’. A CM can be structured in action corresponding to its mode.

6.5. Implement state in CM

Inside a CM(FB) the actual state is represented by the variable ‘stState’. This variable is reference to the global variable ‘State’. Inside the mode specific action you can implement a CASE instruction to differ between several states.



6.5.1. StateChangeInProgress

Each state is structured in different phases. When a state becomes active the bit ‘StateChangeInProgress’ is active just for one cycle. You can use this to program an init phase of the state. Inside a CM(FB) the local variable ‘bStateChangeInProgress’ is initialized as a reference to the global variable ‘StateChangeInProgress’.

For example: Run synchronization command in STARTING only once.

```
//-----------------------------------------------
PACKML_STATES_ENUM.STARTING:

    //entry step
    IF bStateChangeInProgress THEN
        // synchronize to Master Axis
        arAxisCtrl_gb[Axis.AxisNo].SyncMode.SyncDynValues.Acc := 100;
        arAxisCtrl_gb[Axis.AxisNo].SyncMode.SyncDynValues.Dec := 100;
        arAxisCtrl_gb[Axis.AxisNo].SyncMode.SyncDynValues.VelPos := 10;
        arAxisCtrl_gb[Axis.AxisNo].SyncMode.SyncDynValues.VelNeg := 10;

        arAxisCtrl_gb[Axis.AxisNo].SyncMode.Master := Master;
        arAxisCtrl_gb[Axis.AxisNo].Admin._OpModeBits.MODE_SYNC_VEL := TRUE;
    END_IF
```

6.5.2. StateComplete/NotStateComplete

The transition of some states to the next state is not related to an explicit transition command. StateComplete indicates, that all activities in this state are done and the transition to the next state can be processed. Following states are based on StateComplete.

State	Successor State
RESETTING	IDLE
STARTING	EXECUTE
SUSPENDING	SUSPENDED
UNSUSPENDING	EXECUTE
COMPLETING	COMPLETE
HOLDING	HELD
UNHOLDING	EXECUTE
ABORTING	ABORTED
CLEARING	STOPPED
STOPPING	STOPPED

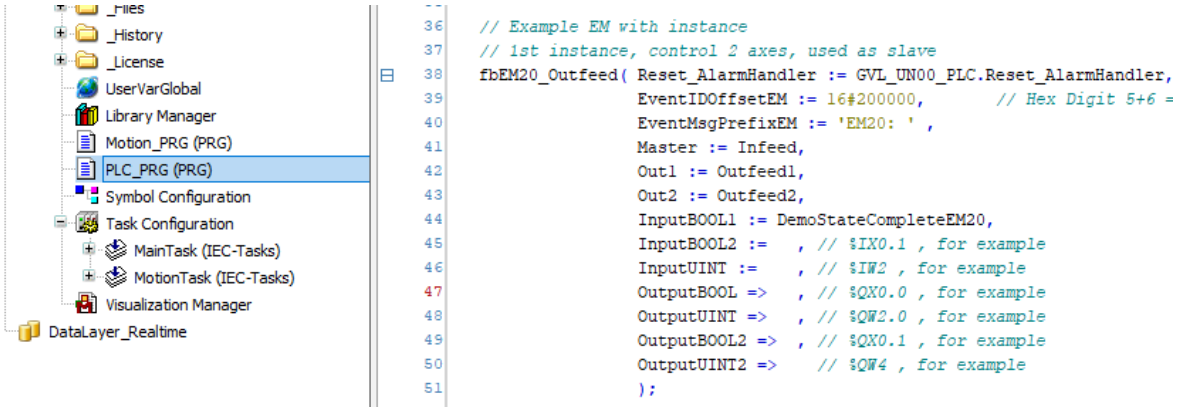
Inside a CM the StateComplete functionality is represented by the variable bNotStateComplete. As long as bNotStateComplete = TRUE, the actual state keeps active and the state machine will not switch to the next state. If you don't implement bNotStateComplete, the state machine will switch immediately to the next step.

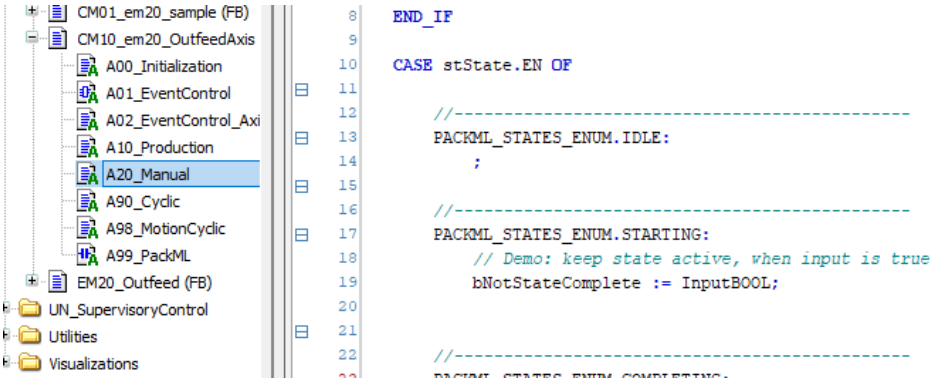
For example, switch from STARTING to EXECUTE only when axis is synchronized:

```
// starting activities are done
IF arAxisStatus_gb[Axis.AxisNo].Admin._OpModeAckBits.MODE_SYNC_VEL
    AND arAxisStatus_gb[Axis.AxisNo].Data.SyncMode.InSync THEN
    // leave state
    bNotStateComplete := FALSE;
ELSE
    // stay in this state
    bNotStateComplete := TRUE;
END_IF
```

The template project includes an example for demonstration purposes. UserVarGlobal includes 3 variables to switch bNotStateComplete in 3 different EM in STARTING state in mode Manual.

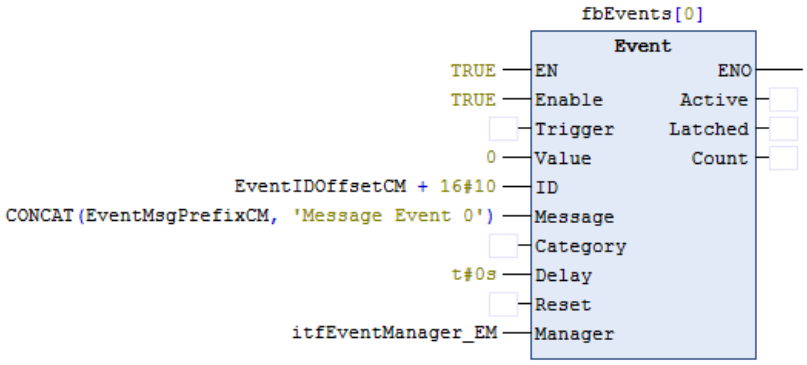
```
DemoStateCompleteEM10 : BOOL := FALSE;
DemoStateCompleteEM20 : BOOL := FALSE;
DemoStateCompleteEM21 : BOOL := FALSE;
```





6.6. Implement Event

Every Event is represented by an instance of Event(FB)



General usage:

- Enable: Activates evaluation of the event
- Trigger: Triggers Event. For example, safety door was opened.
- Value: Optional event value.
- ID: Event identification number which is unambiguously. In this template project the Event ID is structured in three hexadecimal parts.

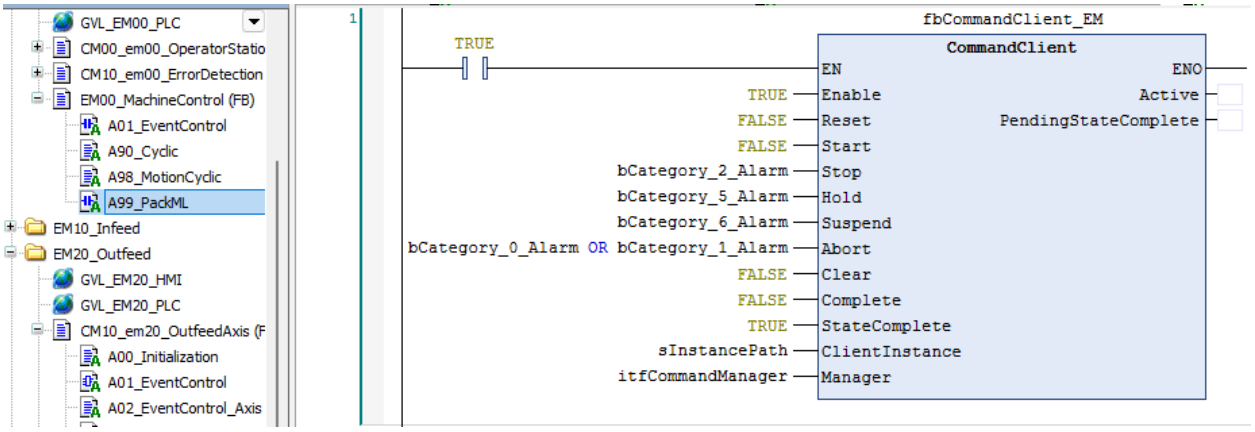
16#	XX	XX	XX
	ID from EM	ID from CM	Event ID

For example: ID 16#201001 shows, the event 01 was trigger from CM10 in EM20.

- Message: Text message about event. EM(FB) and CM(FB) work with Prefix to differ between different instances.
- Category: Category from 0 – 9 depending on the severity. In this template project several reactions are defined for several categories.

Category	Reaction
0, 1	Abort
2	Stop
5	Hold
6	Suspend
3,4,7,8,9	No reaction

This behaviour can be changed in every EM. For example:

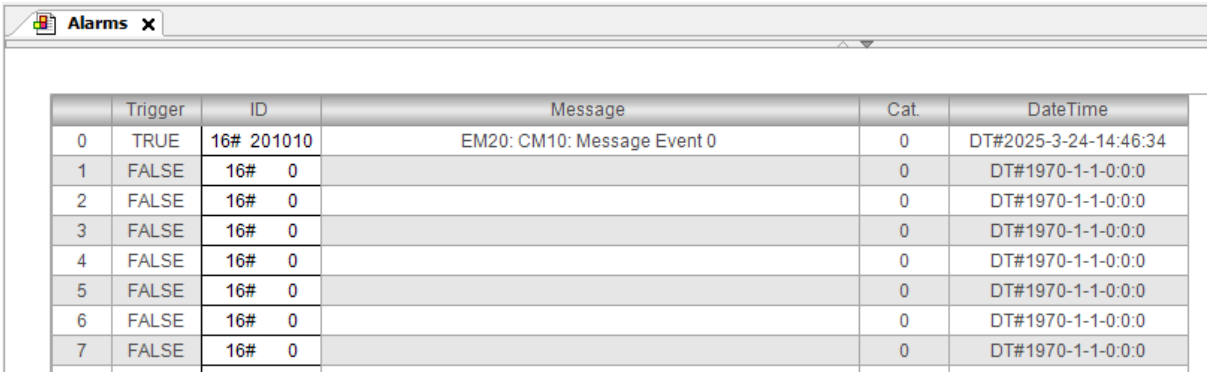


- Delay: Delay for event trigger
- Reset: Separate reset for this event. If Reset = TRUE, the event will reset automatically when Trigger = FALSE.
- Active: Indicates if event is still active (Trigger=TRUE).
- Latched: Indicates if event was or is still active but not reset.
- Count: Counts positive Edges on Trigger without reset.

For more details about the Event(FB) refer to the CXA_PackML_Toolkit.library documentation.

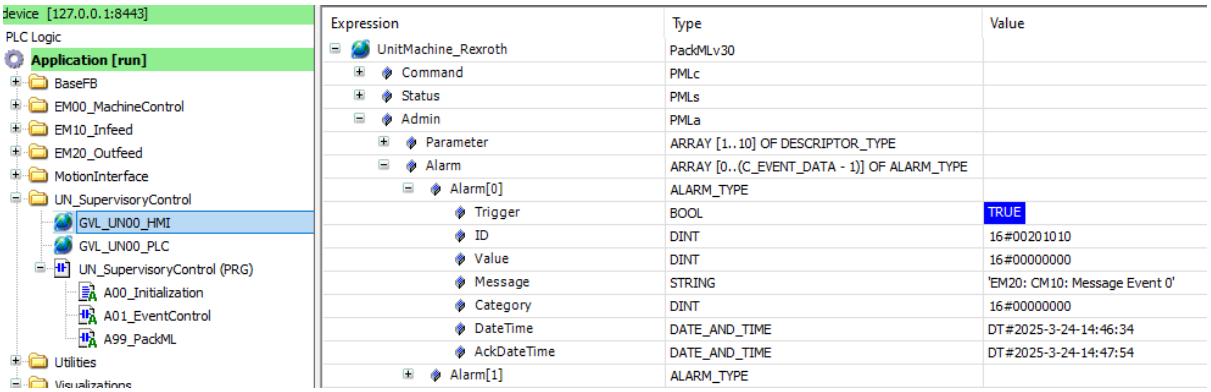
6.7. Read active and acknowledged events

The active alarms are shown in the internal visualization Alarm.

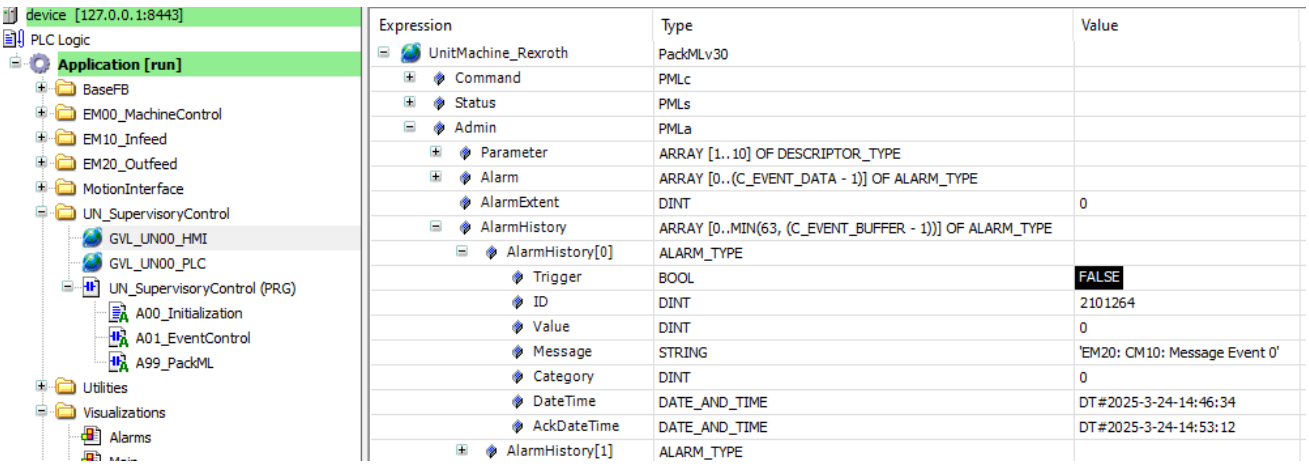


	Trigger	ID	Message	Cat.	DateTime
0	TRUE	16# 201010	EM20: CM10: Message Event 0	0	DT#2025-3-24-14:46:34
1	FALSE	16# 0		0	DT#1970-1-1-0:0:0
2	FALSE	16# 0		0	DT#1970-1-1-0:0:0
3	FALSE	16# 0		0	DT#1970-1-1-0:0:0
4	FALSE	16# 0		0	DT#1970-1-1-0:0:0
5	FALSE	16# 0		0	DT#1970-1-1-0:0:0
6	FALSE	16# 0		0	DT#1970-1-1-0:0:0
7	FALSE	16# 0		0	DT#1970-1-1-0:0:0

And in the global variable GVL_UN00_HMI. UnitMachine_Rexroth.Admin.Alarm



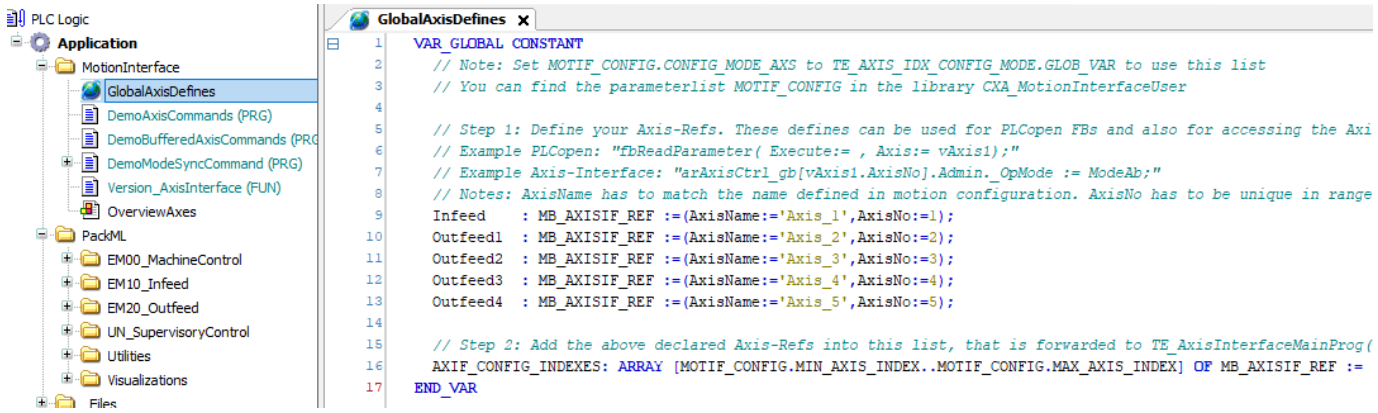
Reset and solved (Trigger = FALSE) alarms will be shifted to the variable UnitMachine_Rexroth.Admin.AlarmHistory



AckDateTime represents the time when the event was reset and solved.

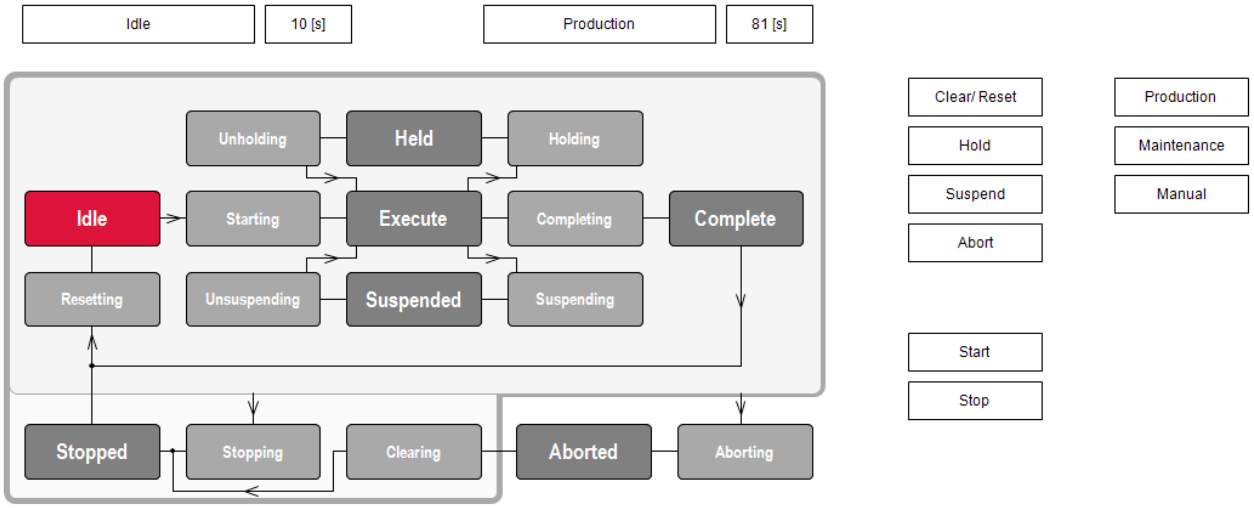
6.8. Add new axis

All axes, which are used in this example, are declared in GlobalAxesDefines. Just add a new MB_AXISIF_REF declaration to the variable list and add this to the AXIF_CONFIG_INDEXES array.

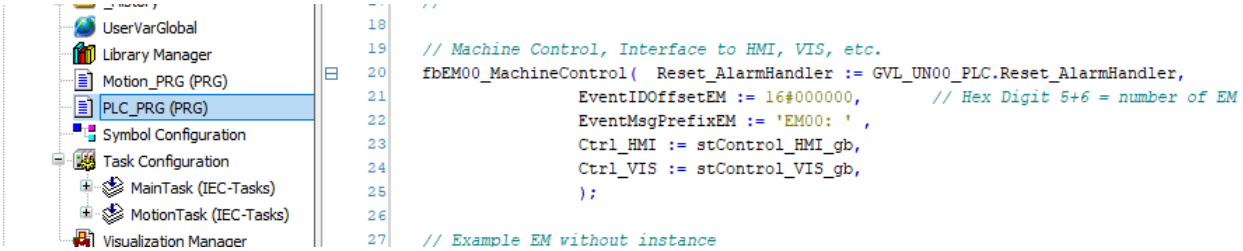


6.9. Switch states

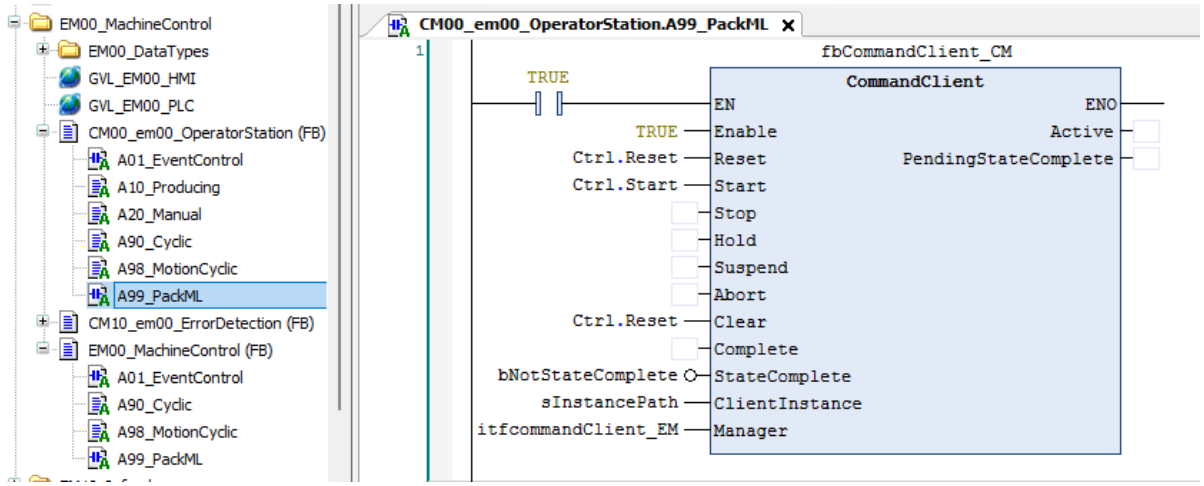
The Main visualization shows the actual mode and state. The buttons give you the possibility to execute transition commands.



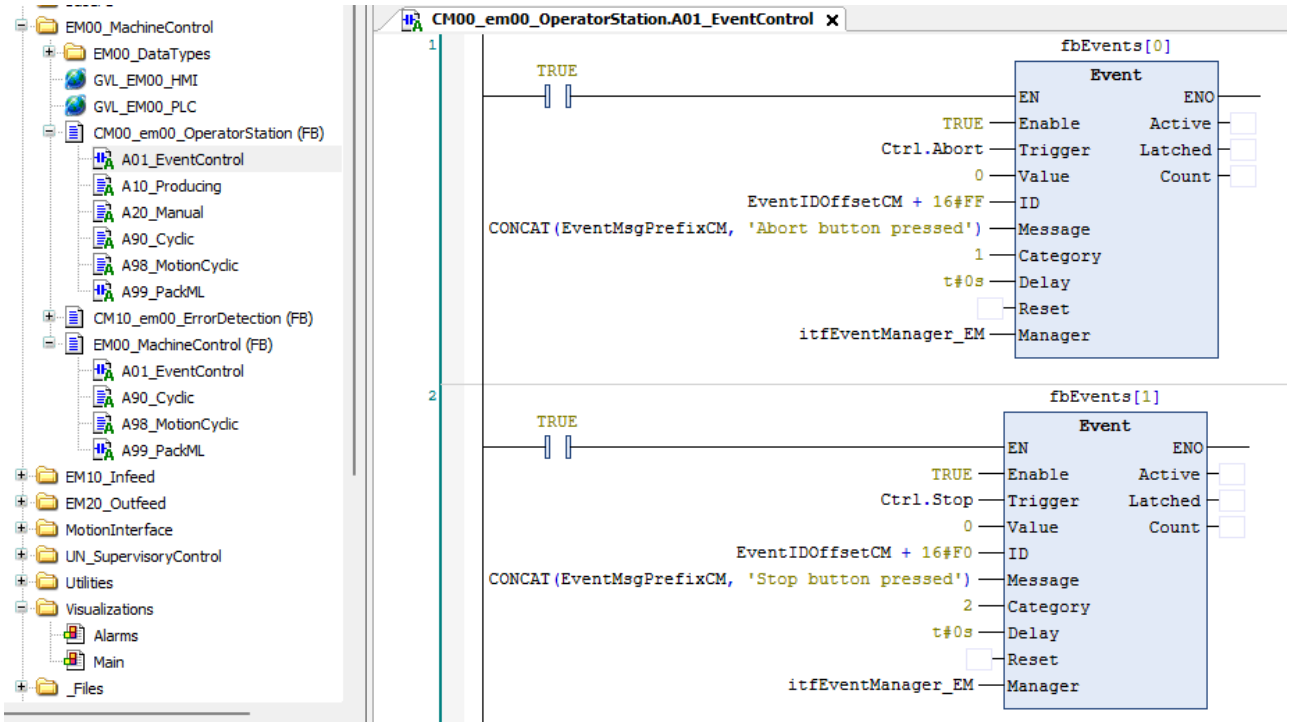
The buttons are linked to a global structure stControl_VIS_gb which is used as an input of EM00_MachineControl(FB). You can use the second structure stControl_HMI_gb to command the state machine from an external source (OPC UA, digital I/O).



This structure is linked to the CM00_em00_Operatorstation(FB) and used as an input of CommandClient(FB).



Some commands are realized as events. In this case the command will be shown in the Alarm array.



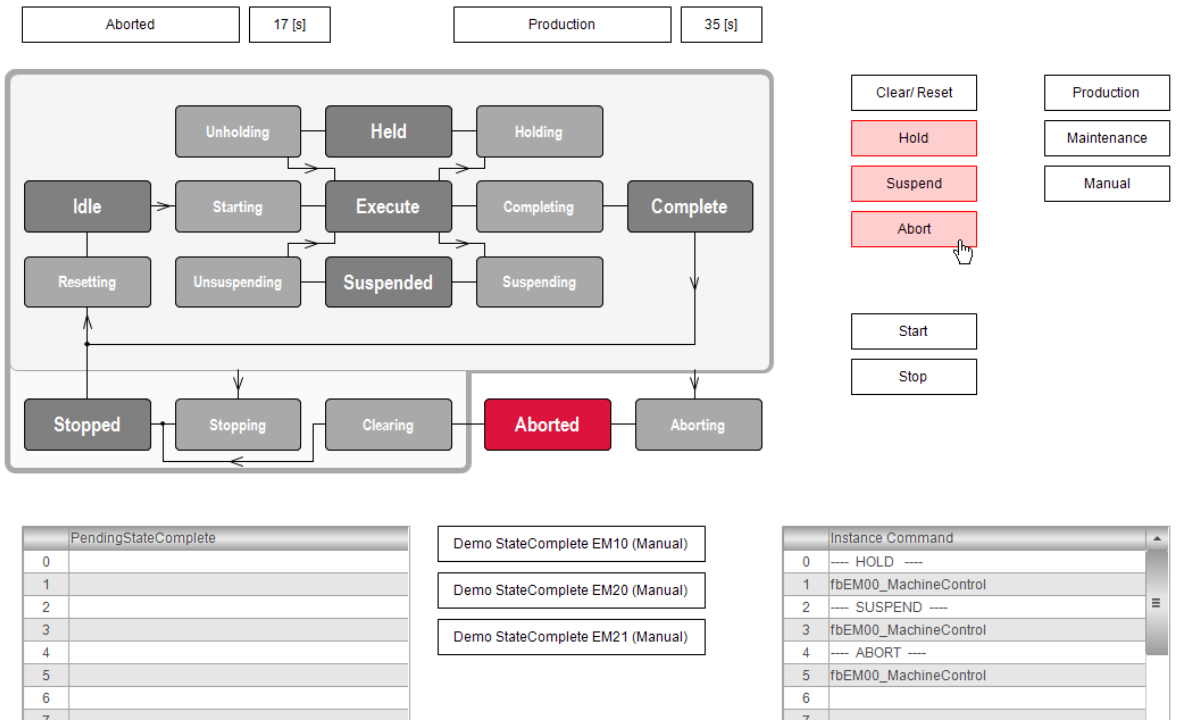
6.10. Debug with internal visualization

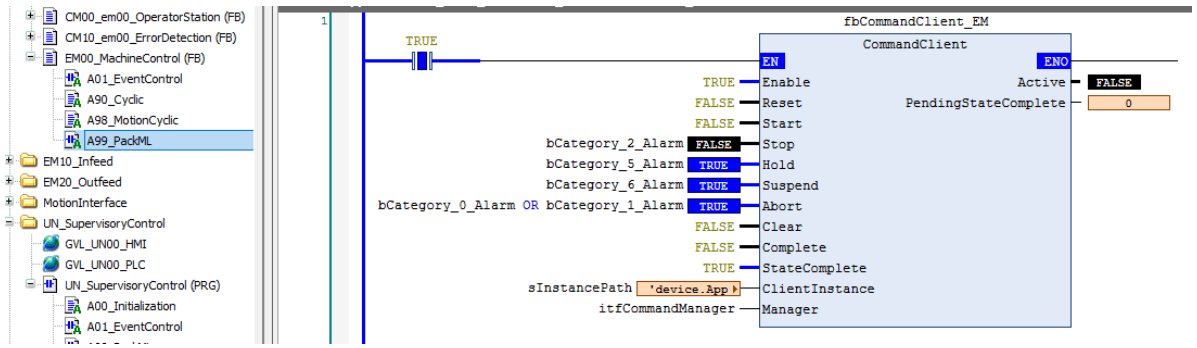
6.10.1. Main

The CommandManager(FB) has 2 outputs. InstancePendingStateComplete and InstanceCommand. These arrays of string give detailed information about the actual status of the state machine and are shown in the Main visualization.

InstanceCommand will show which instance request a transition command through a CommandClient(FB).

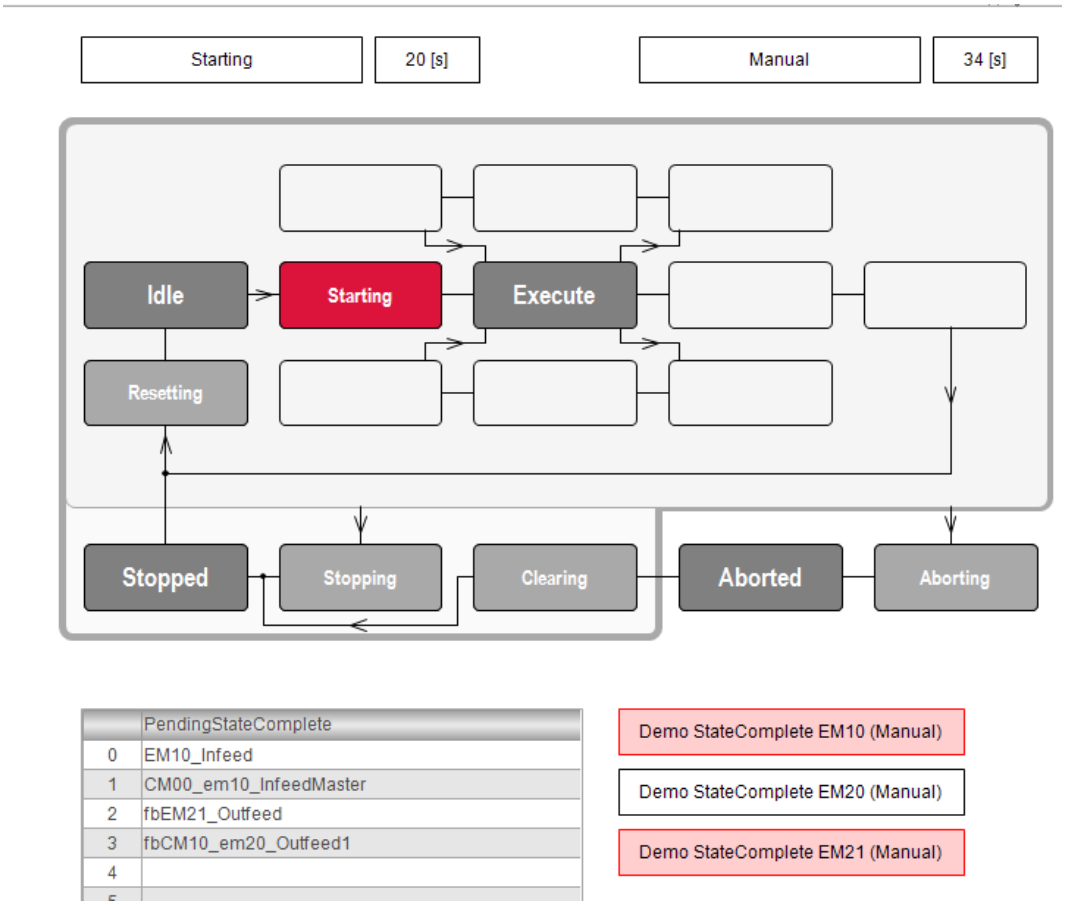
Example:





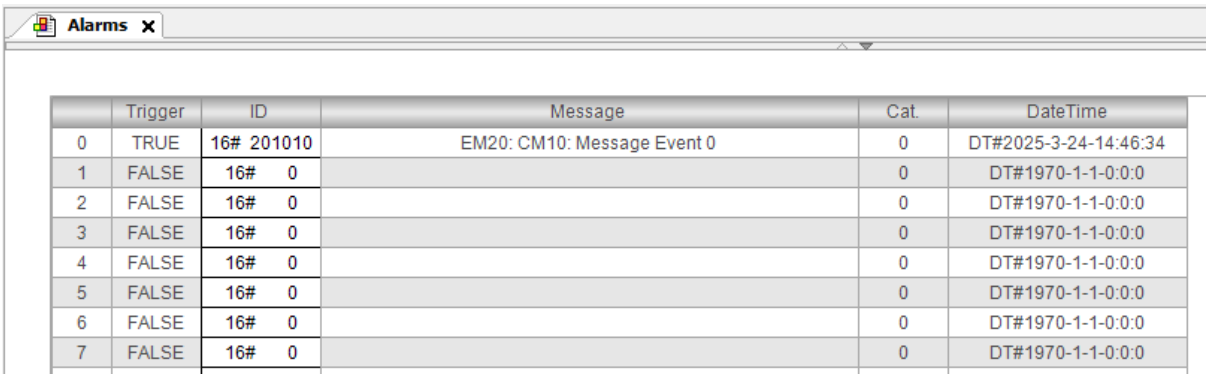
InstancePendingStateComplete will show the instance name of EM and CM which is not reporting StateComplete.

Example:



6.10.2. Alarm

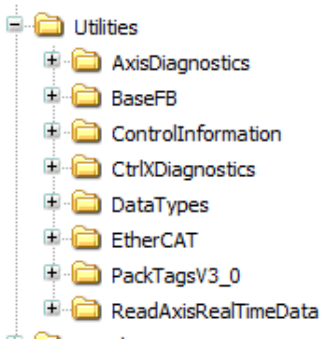
Active Alarms are shown in the visualization Alarm



	Trigger	ID	Message	Cat.	DateTime
0	TRUE	16# 201010	EM20: CM10: Message Event 0	0	DT#2025-3-24-14:46:34
1	FALSE	16# 0		0	DT#1970-1-1-0:0:0
2	FALSE	16# 0		0	DT#1970-1-1-0:0:0
3	FALSE	16# 0		0	DT#1970-1-1-0:0:0
4	FALSE	16# 0		0	DT#1970-1-1-0:0:0
5	FALSE	16# 0		0	DT#1970-1-1-0:0:0
6	FALSE	16# 0		0	DT#1970-1-1-0:0:0
7	FALSE	16# 0		0	DT#1970-1-1-0:0:0

7. Utilities

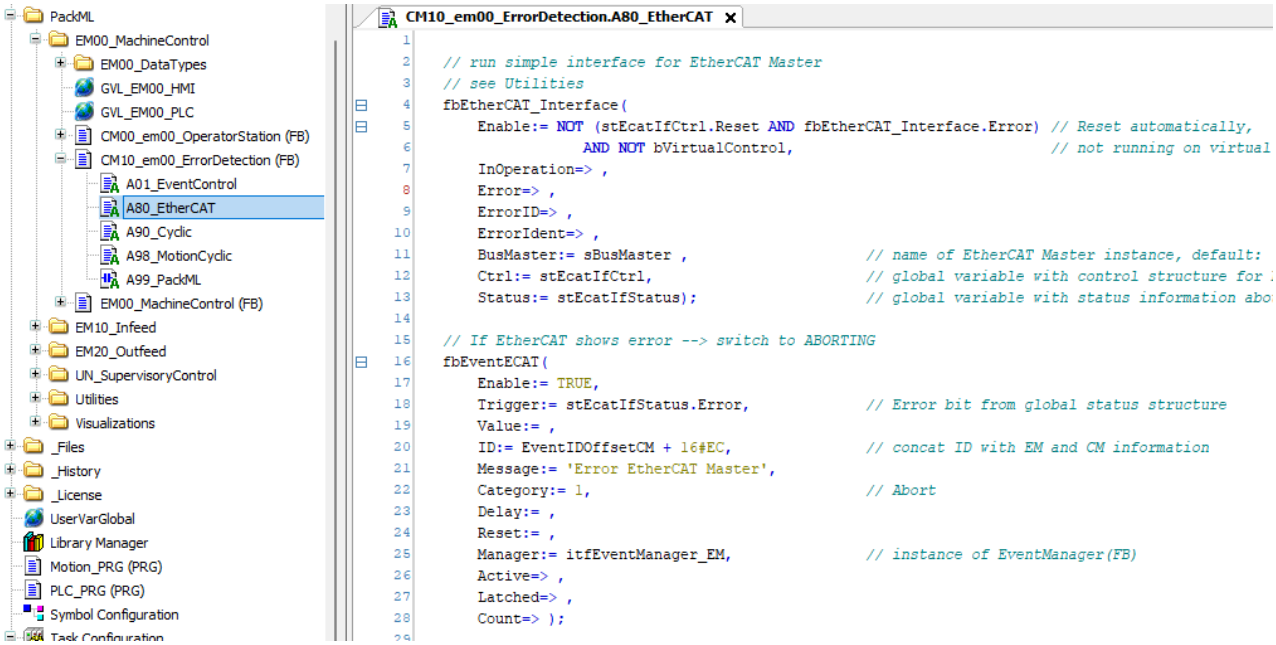
The utilities folder contains some helpful programming examples for different use cases. Everything is provided with open sources, so it is possible to understand the behaviour more in detail and to change the behaviour to the application needs.



7.1. EtherCAT

FB for a simple EtherCAT interface. Read EtherCAT Master diagnostics from IL_ECATMasterState (CXA_EthercatMaster.library) and command bus state with IL_ECATMasterSetBusState.

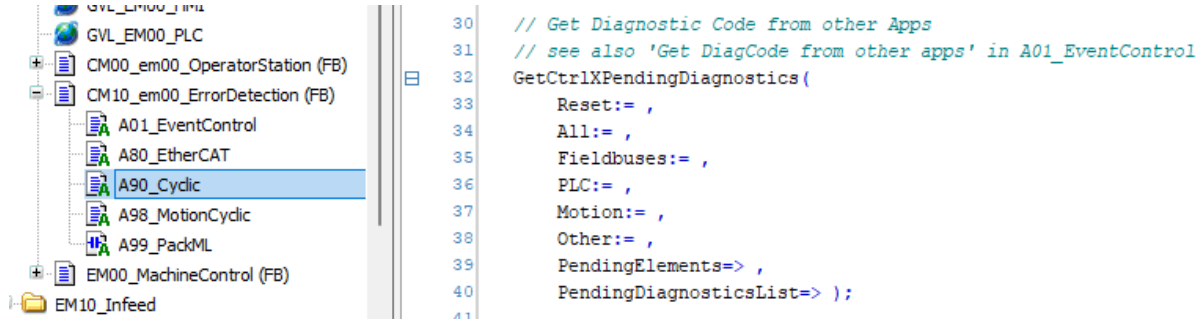
It is possible to detect errors or switch the bus state through the structures stEcatIfCtrl and stEcatIfStatus similar to ImcInterface. The implementation is done in CM10_em00_ErrorDetection.



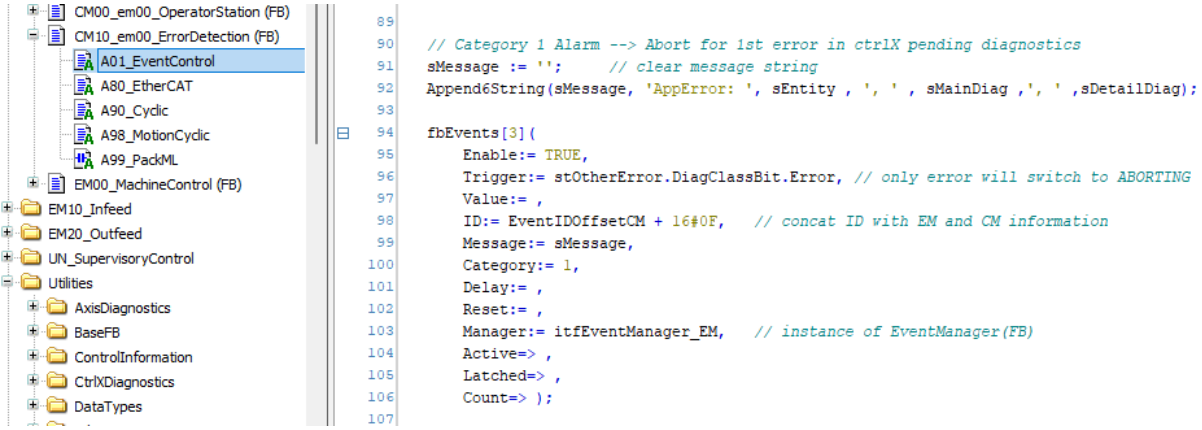
7.2. CtrlXDiagnostics

Program to read actual pending diagnostics from ctrlX CORE with a possibility to filter diagnostics from different apps.
The implementation is done in CM10_em00_ErrorDetection.

Run cyclic Update Diagnostics

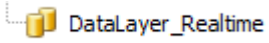


Example: Create Category 1 Event with ctrlX CORE Error from PLC or Fieldbus App.



7.3. AccessRealTimeData

This folder includes two examples to read RealTime Data from DataLayer without using the configuration under DataLayer_Realtime.



7.3.1. _ExampleReadRealTimeEtherCatData(PRG)

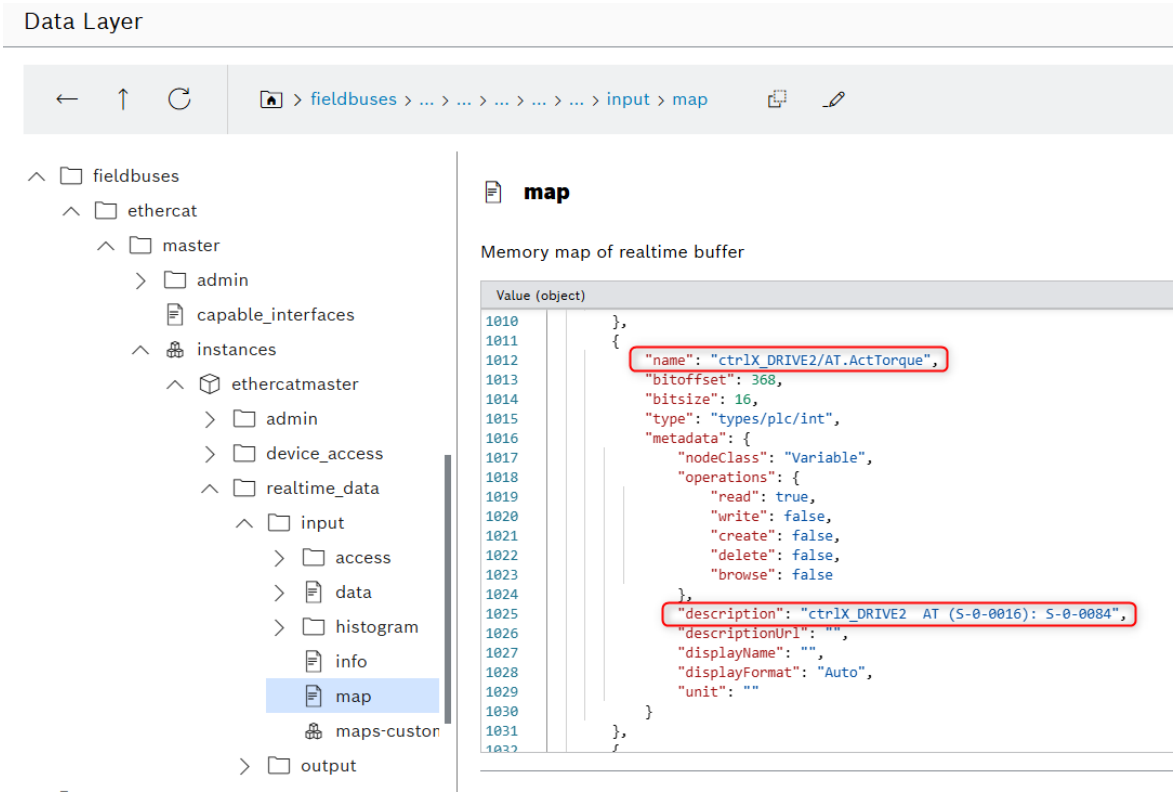
This example shows how to read and write data from EtherCAT devices in real time.

FB_GetEtherCatRealTimeDataMemoryMap reads the realtime memory allocation of all EtherCAT devices.

FUN_GetEtherCatRealTimeDataName will search in the description for the device and parameter name. It will return the correct name of the realtime data in memory map which is necessary to get a handle.

FUN_GetHandleRealTimeData is used to create a handle to the realtime data based on the name.

FUN_ReadRealTimeInput and FUN_WriteRealTimeOutput are used to access the data in realtime.



7.3.2. _ExampleReadRealTimeAxisDiag(PRG)

The Motion App has the possibility to configure user specific information as real time capable data in the data layer.

Available data is listed under the DataLayer node motion/axs/<AxisName>/state/realtime/available-data.

available-data

available data for realtime configuration

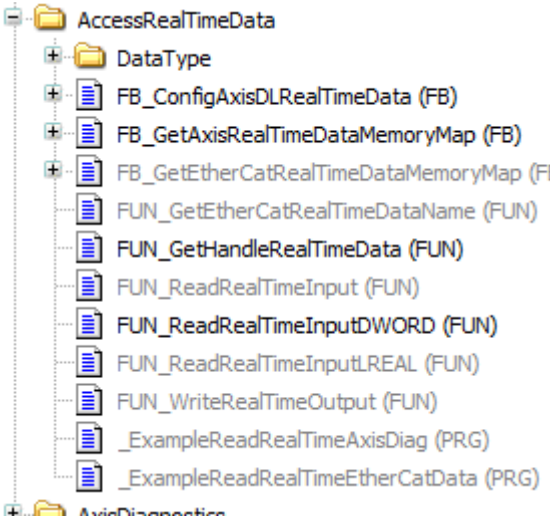
Value (arstring) - 117
state/values/ipo/pos
state/values/ipo/vel
state/values/ipo/acc
state/values/ipo/jrk
state/values/ipo-add/dist-from-start
state/values/ipo-add/dist-to-target
state/values/ipo-add/time-from-start

Information you want to read in realtime in the PLC needs to be configured to the DataLayer node motion/axs/<AxisName>/cfg/realtime/active-data.

active-data

configuration of realtime data

Value (arstring) - 2
state/diag-info/diag-main
state/diag-info/diag-detail



FB_ConfigAxisDLRealTimeData will do this configuration of required realtime data. With FB_GetAxisRealTimeDataMemoryMap and FUN_GetHandleRealTimeData you can get a handle which must be used with FUN_ReadRealTimeInputDWORD to read the data in realtime. An example implementation is shown in _ExampleReadRealTimeAxisDiag.

The implementation is used inside FB_ConfigAxisDLRealTimeData (see 7.4 AxisDiagnostics).

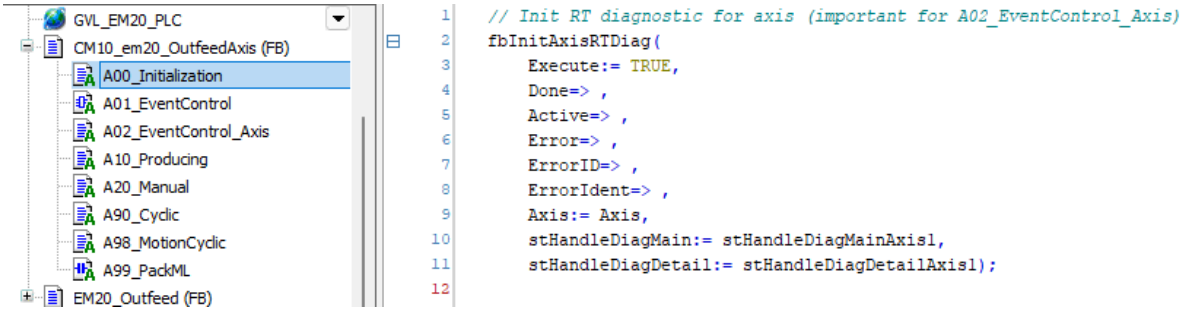
7.4. AxisDiagnostics

These FBs are used to simplify the diagnostic of an axis and create several events depending on the error/warning severity. When you want to supply the Event(FB) with diagnostic information as soon as the axis error happens, it is important to read the axis diagnostic information in reatltime.

7.4.1. FB_InitAxisRTDiag

This FB writes the configuration of realtime diagnostics to the Datalayer and delivers a handle for the realtime reading function.

It is implemented in the initialization routine.

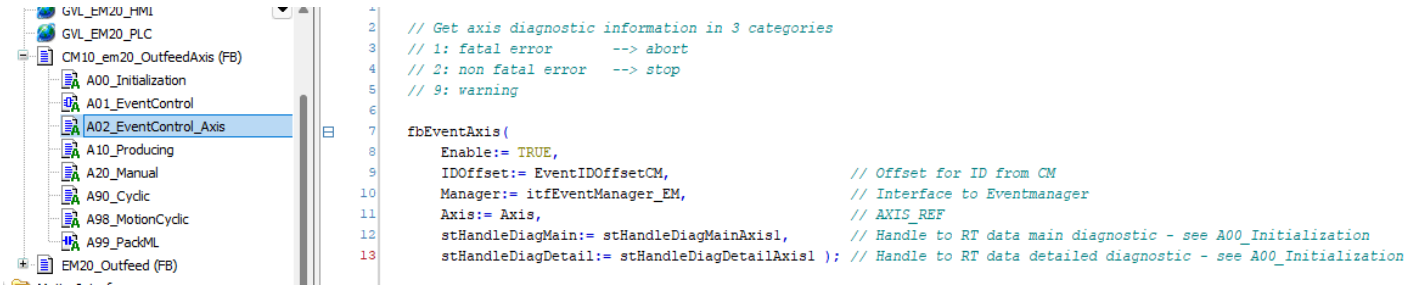


7.4.2. FB_EventAxis

This FB reads the realtime axis diagnostic and reports several events depending on the error/warning severity.

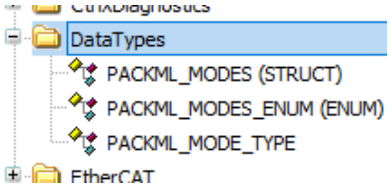
Fatal error will lead to category 1 event, nonfatal error will lead to category 2 event and warning will lead to category 9 event.

The implementation is done in A02_EventControl_Axis.



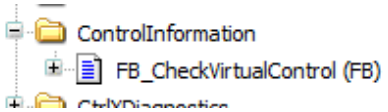
7.5. DataTypes

DataTypes for PackML modes definition. This needs to be extended, when implementing a new operation mode



7.6. ControlInformation

FB to check if PLC is running on a virtual device. This might be helpful when evaluating EtherCAT diagnostics.



7.7. PackTags

This folder includes predefined datatypes to implement PackTags according to PackML. The datatypes can be adapted to the application needs.