PLC basic concept



Customer: Vistaprint

Project: Promo

Projectnr..: 10116.00

Version: 0.9

Storage:

# Table of contents

[1 Table of contents 2](#_Toc371519620)

[2 Introduction 4](#_Toc371519621)

[2.1 Scope 4](#_Toc371519622)

[2.2 Reference Documentation 4](#_Toc371519623)

[2.3 Version 4](#_Toc371519624)

[2.4 Abbreviations, definitions, glossary 4](#_Toc371519625)

[3 Basics 5](#_Toc371519626)

[4 Naming conventions 5](#_Toc371519627)

[4.1 Programming Conventions for IEC61131-3 in TwinCAT 3 5](#_Toc371519628)

[4.1.1 Libraries 5](#_Toc371519629)

[4.1.2 Language 5](#_Toc371519630)

[4.1.3 Names 5](#_Toc371519631)

[4.1.4 Library Identifier 7](#_Toc371519632)

[4.1.5 Error codes 7](#_Toc371519633)

[4.2 Programming conventions overrule the TwinCat notification 8](#_Toc371519634)

[4.2.1 Type Prefix 8](#_Toc371519635)

[4.2.2 Constants 8](#_Toc371519636)

[4.2.3 Unit Suffix 8](#_Toc371519637)

[4.2.4 IO Suffix 8](#_Toc371519638)

[4.3 Global variables 8](#_Toc371519639)

[4.3.1 Examples 8](#_Toc371519640)

[4.3.2 Header 9](#_Toc371519641)

[5 Designed structure of PLC-software 9](#_Toc371519642)

[5.1 Controller-tree 9](#_Toc371519643)

[5.2 Controller 10](#_Toc371519644)

[5.2.1 Common Interface (CIf) 11](#_Toc371519645)

[5.2.2 Specific Interface (SIf) 12](#_Toc371519646)

[5.2.3 Modes 12](#_Toc371519647)

[5.2.4 Commands / State 12](#_Toc371519648)

[5.2.5 IO handling 13](#_Toc371519649)

[5.2.6 Parameter handling 14](#_Toc371519650)

[5.2.7 State machine 14](#_Toc371519651)

[5.3 Example Level1 Cylinders (Class CYL) 16](#_Toc371519652)

[5.4 Alarm handling 17](#_Toc371519653)

[5.5 Software info 19](#_Toc371519654)

[5.6 Logger on PLC 19](#_Toc371519655)

[6 PLC Software Release Management 20](#_Toc371519656)

[6.1 History 20](#_Toc371519657)

[6.2 Test before release 20](#_Toc371519658)

[6.3 Dependencies 20](#_Toc371519659)

# Introduction

## Scope

This document is solely for the benefit of Vistaprint and all the persons that are involved at platform development.

## Reference Documentation

|  |  |  |
| --- | --- | --- |
| Dokument | Version | Datum |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Version

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Description | Author | State | Version | Date |
| Start | AVME/kvo | d | 0.1 | 18.04.2013 |
| Changes after review with R.Engelputzeder, P.Cipriano | AVME/kvo | d | 0.2 | 23.04.2013 |
| Different changes done | AVME/kvo | d | 0.3 | 02.05.2013 |
| Add common cylinder controller description | AVME/kvo | d | 0.4 | 15.05.2013 |
| Cmd/State -> with constants, initial value defined | AVME/kvo | d | 0.5 | 25.05.2013 |
| Programming converntions overrule from Array and structure canceld | AVME/pre | D | 0.7 | 10.07.2013 |
| Changes after first part of commissioning | AVME/mru | D | 0.8 | 10.10.2013 |
| Add history, test before release and dependencies, rework alarm reaction, substates and IO handling | AVME/mru | D | 0.9 | 06.11.2013 |
|  |  |  |  |  |
|  |  |  |  |  |

State: **d** = design, **r** = released

## Abbreviations, definitions, glossary

|  |  |
| --- | --- |
| Designation | Name |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Basics

Vistaprint will develop the next generation of machines and facilities with the industrial automation system of Beckhoff. The PLC software has to be design and programmed, so far as possible, with the basic principles of object oriented. This will facilitate the later integration, if needed, of similar modules in the same and other PLC-projects. In this document the basic „framework“ for this is defined and is the basic for all project with this platform. In this way it is possible to exchange developers and software modules between different projects.

# Naming conventions

Proposed in Specification of GEMINI Line/Printer

Common naming conventions will be as far as possible be taken from the Beckhoff conventions listed on webpage: <http://twincat3.infosys.beckhoff.com/content/1033/tc3_plc_intro/html/programmingconventionstc3.htm?id=3113>

## Programming Conventions for IEC61131-3 in TwinCAT 3

### Libraries

Beckhoff PLC libraries start with the prefix **Tc**. e. g.: **Tc3\_Utilities**The namespace of a Beckhoff PLC library is like the library's name.

The libraries have the filename extension\*.**compiled-library** and are embedded in the library repository.

### Language

The PLC programs are complete in english (program code, variable names, comments, ...)

#### Upper- and Lower Case

The first letter of a name is always written in upper case. If a name contains several words, the first letter of every word is written in upper case. Don't use separators (e. g. \_ ) between the words.

The PLC Compiler is not case-sensitive. Nether less it is useful to care about upper- and lower cases in the code because of the readability.

#### Valid Signs

The names contain solely the following letters, numbers and special characters: 0...9, A...Z, a...z,

### Names

The names have a consistent prefix thereby you can easily recognize the object type. The names should be descriptive so you can easily understand the purpose of the object.  
If names consist of several words, they should be directly linked together and the first letter of each word should be in upper case (e. g.GetData) for better readibility.

#### Objects

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Prefix** | **Description** | **Sample** |
| FUNCTION\_BLOCK | FB\_ | function block | FB\_GetData |
| ACTION |  | action (of a function block or a program) | Move |
| METHOD |  | methode (of a function block, a program or an interface) | Reset |
| PROGRAM |  | program | ModuleControl |
| FUNCTION |  | function | Convert |
| STRUCT | ST\_ | structure | ST\_BufferEntry |
| ENUM | E\_ | type of enumeration | E\_SignalStates |
| TYPE | T\_ | aliastype | T\_Nibble |
| UNION | U\_ | union | U\_Size |
| INTERFACE | I\_ | interface | I\_CylinderControl |

In the object names write the prefix generally in upper cases. Separator between prefix and objectname is the underline character '\_'.

Note: If interfaces of TcCOM objects are directly implemented, they only have the prefix 'I'. e. g..: ITcUnknown

#### Instances

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Prefix** | **Description** | **Declaration Sample** | **Call Sample** |
| SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, BYTE, WORD, DWORD, LWORD | n | integer, bit based number | n ErrorId :UDINT; nSize :UINT; nMask : WORD; | nErrorId := 16#745; nSize := SIZEOF(); nMask := nMask & 16#0FF0; |
| BOOL | b | boolean (bit) | bSwitch |  |
| REAL, LREAL | f | float | fValue |  |
| STRING | s | string | sNetId |  |
| WSTRING | ws | wide string (unicode) | wsName |  |
| TIME, LTIME | t | time | tDelay |  |
| DATE | d | date | dMonday |  |
| DATE\_AND\_TIME | dt | date and time | dtNewYear |  |
| ARRAY[...] OF ... | a | array | aMessages |  |
| POINTER TO ... | p | pointer | pData | pData := ADR(aBuffer); |
| REFERENCE TO ... | ref | reference | refSize | refSize REF= iSize; |
| *FUNCTION\_BLOCK* | fb | instance of function block | fbGetData | fbGetData(); |
| *STRUCT* | st | instance of structure | stBufferEntry | stBufferEntry.iCounter := 5; |
| *ENUM* | e | instance of enum | eSignalState :E\_SignalStates; | eSignalState := eSignal\_Stop; |
| *TYPE* | entsprechend dem internen Typen | instance of alias type | nNibble :T\_Nibble; sNetId :T\_AmsNetId; | nNibble := 16#1; sNetId := '1.2.3.4.5.6'; |
| *INTERFACE* | ip | interface pointer | ipCylinder :I\_Cylinder; | ipCylinder := fbCylinderBase; |
| *UNION* | u | instance of union | uConfig :U\_NC\_ConfigParam; | uConfig.sParam := '56.23.87'; |

In names of variables and instances the prefix is always written in lower cases.

Instances of function blocks which are defined in the IEC 61131-norm and are of great importance (F\_TRIG, TON,), have no other prefix.  
e. g.: fbAdsTimer :TON;

**Global variables** have the same naming rules, no second prefix is added.

**Properties** have the same naming rules, no second prefix is added.

**Constants** (lokal and global) have the prefix 'c'.  
e. g.: cMaxVelo :LREAL;

**Values of enumerations** are named like enumeration instances. You should insert an abbreviation of the enumeration type.  
e. g.: eSignal\_Red;

**HRESULTs** are an exception and are called 'hr'.  
e. g.: hr :HRESULT;

Note: Don't use the type DWORD for pointer, it is not allowed because of the 64bit-compatibility. Instead of this use per default **POINTER TO BYTE**. Alternatively you can use POINTER TO LREAL, if the pointer points to an ARRAY[...] OF LREAL.

Note: In case of convolution of different types declare only the external prefix. Exception: 'pp' for POINTER TO POINTER TO ... and 'pip' for POINTER TO INTERFACE can be used.

### Library Identifier

Alle declarations, global variables and constants in a library contain a short library identifier.  
e. g.: FB\_CM\_PowerSpectrum  
e. g.(global constant): cCM\_MinLog10Arg  
e. g.(global variable): iCM\_FFTLen

### Error codes

Error codes can be specified as nErrorId (Typ UDINT) or as hr (Typ HRESULT).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Declaration** | **Error range** | **no Error** | **Message** | **Testfunctions** |
| nErrorId :UDINT; | > 0 | 0 |  | IF nErrorId = 0 THEN ... END\_IF  IF nErrorId <> 0 THEN ... END\_IF |
| hr :HRESULT; | < 0 | >= 0 | > 0 | IF SUCCEEDED(hr) THEN ... END\_IF  IF FAILED(hr) THEN ... END\_IF |

## Programming conventions overrule the TwinCat notification

### Type Prefix

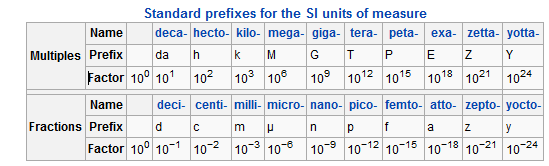
|  |  |  |
| --- | --- | --- |
| Type | IEC | Präfix IEC |
| 1 Bit | BOOL | bol |
| 8 Bit signed | SINT | sin |
| 8 Bit unsigned | USINT | usi |
| 16 Bit signed | INT | int |
| 16 Bit unsigned | UINT | uin |
| 32 Bit signed | DINT | din |
| 32 Bit unsigned | UDINT | udi |
| Floating Point 32 Bit | REAL | rea |
| String (zero term.) | String | str |
| DateTime | DATE\_TIME | dt |

### Constants

Constants have a “c\_” prefix before the datatyp.

### Unit Suffix

The plc unit of a variable has to be added at the end. Normally the si base units has to be used and their prefixes.



### IO Suffix

For io input variables a suffix for normally open (\_NO) or normally closed (\_NC) should be added.

## Global variables

Global variables will be written like local. Only a prefix “g\_” shows the different. The usage of global variables should be minimalized so far as possible.

### Examples

g\_udiVarInfo (global variable)

bolEnableHeating (local bool)

pstConfig^.stFeeder.uinType (dereferenced pointer)

stButtonColor (structure)

c\_SignalDelay (constants)

### Header

In every program, function block or function it has to be inserted a header. Important is to write a short function description and a version history with changes made. (\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

¦

¦ Vistaprint Schweiz GmbH

¦ CH - 8401 Winterthur

¦

¦ www.vistaprint.ch - info@vistaprint.ch

¦\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Function desription:

Basic controller implementation. This Controller should be the base father controller for all in the

feature implemented controller classes.

History:

Version Date Author Comment

---------------------------------------------------------------------------------------------------

0.00.1 25.04.2013 AVME/KVO Start history

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*)

# Designed structure of PLC-software

## Controller-tree

For the different machine / facilities, options and enhancements a modularized SW-design is a must. There are different ways to get such a software design. To design we use a diagram style after Jackson. But not the program design is made in this diagram only the hierarchically modulation of the different sw-packages. A sw-package in this drawing is called controller. Every controller is implemented as a state machine which is controlled by a standard commando interface. In the level 0 also called element level the controllers are placed with direct io-connection (per example a cylinder or a motor). The parent (the controllers above) could be placed in level 1, 2, 3 or 4 this depends from the design respective from the functionality. Not every hierarchy level has to be used. Basically there could be more or fewer levels but in automatic software until now with this levels a good [degree](http://dict.leo.org/ende/index_de.html#/search=degree&searchLoc=0&resultOrder=basic&multiwordShowSingle=on) [of](http://dict.leo.org/ende/index_de.html#/search=of&searchLoc=0&resultOrder=basic&multiwordShowSingle=on) [abstraction](http://dict.leo.org/ende/index_de.html#/search=abstraction&searchLoc=0&resultOrder=basic&multiwordShowSingle=on) is possible.

The modulation of a new machine or line software normally is made in compliance with the followed rules:

Level -1 (Driver):

* Device with bus protocols, sequences and other complex component
* No application logic
* Usable for all automation software which is using this device

p.e.: TCP-communication to a device, servo drive siemens, servo drive SEW, USB device

Level 0 (Element):

* Functional actor-/sensor group (mechanical and electrical)
* Application logic to control driver
* Manual commands for single movement or other action needed

p.e.: cylinder (2 end switch, 2 pneumatic valve), motor (overload feedback, contact feedback, main contact), servo axis (control driver, home sequence, parameter settings, jog, positioning, error handling)

Level 1 (Element group):

* Sequence controlling of elements
* Abstract simple sequences
* Semiautomatic or manual commands for tests
* Reduces number of interfaces to control from module controller (divide and conquer)

p.e.: pick and place sequences

Level 2 (Module):

* Functional part of machine or line
* Semiautomatic local functionality

p.e.: loading station, UV-device, inlet transport system

Level 3 (Module group):

* Functional part of machine or line
* Semiautomatic local functionality

p.e.: transport system, in line machine 1

Level 4 (Main)

* Master controller (top of hierarchy)
* Handles start/stop in automatic mode

Example of a controller tree drawing:



## Controller

Every controller has a fix interface structure which is common. So it is possible to access this interface from everywhere to send commands or to get the actual state of a controller.

Controllers are implemented as FUB’s. A FUB represent a class of a controller typ. The FUB instance called in a program represents the final implementation of a controller. The definition of a base controller (FB\_CtrlBase) allows defining the principal structure and implementation of a controller. A new controller is always inherited from the base controller or one of his Inheritances. So basic changes in command handling per example will only be changed in the base controller. To get the possibility to override and exceed the functionality of the base controller most of the functions are in methods resolved.

Basic concept:

The FB\_Ctrl\_Base implements the Interface I\_CtrlBase. In the followed overview the most important methods and their calls are designed. The blue ones are protected. So no public access is possible, but overload after inherits is possible. The green methods are for public calls on this way it’s possible to read out the FUB instance or common interface for public accessible arrays. The state methods have the fix prefix sta\_.



### Common Interface (CIf)

Because this struct is implemented and defined in the base controller (FB\_CtrlBase) every controller inherits this common structure.

T\_Ctrl\_CIf :

Job: T\_Ctrl\_Job;

Info: T\_Ctrl\_Info;

Config: T\_Ctrl\_Config;

T\_Ctrl\_Job :

/// mode if mode = AUTO -> ACmd will be used, otherwise the MCmd channel is open

eMode: E\_Mode;

/// automatic command from parent ctrl

intACmd: INT;

/// manual command from UI p.e.

intMCmd: INT;

//active cmd

strCmd: STRING;

//active state

intState: INT;

strState: STRING;

//if used - show substates for debugging

strSubState: STRING;

/// event channel to signal special events to parent parallel to status

intEvt: INT;

/// reset all alarms

bolAlmReset: BOOL;

/// activate io-forcing

bolIoForce: BOOL;

/// overwrite suspend all interlocks (for debugging/testing)

bolIlkOverwrite: BOOL;

/// activ alarm class of parent ctrl

eParent\_ActivAlmClass: E\_AlmClass;

/// activ alarm class of me ctrl

eMe\_ActivAlmClass: E\_AlmClass;

/// list of all implemented commands on this controller class

aCmdList: ARRAY[0..cMaxCmdPerCtrl] OF T\_Ctrl\_CmdListElement;

Controller instance information T\_Ctrl\_Info :

// name of controller instance

strName: STRING(80);

// class of controller

strType: STRING(80);

// unique controller instance identification number

intCtrlId: INT;

// unique identification number of parent controller instance

intParentCtrlId: INT;

// all inputs valid -> init done

bolInitDone: BOOL;

// Childs

p\_afbChild: ARRAY[0..Base\_Const.cMaxNrOfChilds] OF POINTER TO BYTE; // pointer array of childs FB

p\_aCIf: ARRAY[0.. Base\_Const.cMaxNrOfChilds] OF POINTER TO T\_Ctrl\_CIf; // common interface array of childs

Software Configuration T\_Ctrl\_SwCfg:

bolExist: BOOL; // hardware/controller exists otherwise go to state SINGLE\_ANSWER

bolEnable: BOOL; // enable statemachine

bolSimulation: BOOL; // set controller in simulation(statemachine active, io local simulated)

### Specific Interface (SIf)

Specific interface for cylinder T\_Ctrl\_SIf\_CYL:

// configuration

Cfg: T\_Ctrl\_SIf\_CYL\_Cfg;

// parameter

Par: T\_Ctrl\_SIf\_CYL\_Par;

// current values

Cur: T\_Ctrl\_SIf\_CYL\_Cur;

// io input

In: T\_Ctrl\_SIf\_CYL\_In;

// io output

Out: T\_Ctrl\_SIf\_CYL\_Out;

//alarm number used in this controller

Alm: ARRAY[1..6] OF T\_Ctrl\_Alm;

// interlock of output/movements

Ilk: T\_Ctrl\_SIf\_CYL\_Ilk;

### Modes

We differentiate two different modes. For the automatic sequences the commands are written over the Job.ACmd channel from parent controller to sub controller. The manual commands are for testing and services, so the selected controller follows in the manual mode no more his parent but the UI button per example.

Automatic (Production) mode -> command parent -> child over ACmd channel

Manual mode -> command UI -> UI over MCmd channel

For safety reason it’s important to distinguish mode and operation mode. For the operation mode it is enough to distinguish automatic and service. This operation mode description should be done in manual and safety documents. In service is it necessary to control per example a servo motor with open guard doors. This is only allowed with death man switch or safety speed supervision equipment.

### Commands / State

The command and state channels should be defined over enumeration. So they are shown in debugging and watch windows with the string defined in the enumeration. To be sure which commands are available respective implemented for a controller a command array extends the common interface.

Every controller has these states because they are already defined in base controller:

PON -> Power on, first state after power on, for initializing

OFF -> off, fast stop, switch off

ERROR -> Failure of devices, sensors etc.

ERRORQuit -> Reset failures and decide which is the next state

SINGLE\_ANSWER -> Simulation of command/state interface, no code is executed / needed

### IO handling

With local input and output bridges in the controller it could be guaranteed the most possible flexibility. So the controller instances could run in different task classes. In the io-bridges are also solved the io-forcing from UI and the local io simulation with disconnection of the hardware io’s.

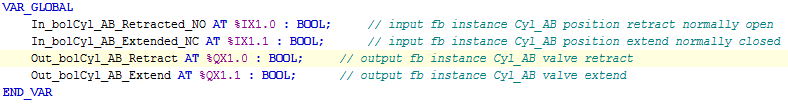
Problems with io-handling:

* If mapped io is in struct defined there are a lot of io’s without a connection to the hardware (p.e. cylinder 2 inputs and 2 outputs in default, usage for monostable without sensors only 1 output)
* Sensor signal could have a normally closed or normally open logic, this should be shown in the name (Suffix \_NC or \_NO) and implemented the different behaviour in one standard FUB
* …

One possible solution for this problem is to add global defined io variables as input or output to the FUB. So only the really used io variables will be defined, perhaps directly with explicit mapping information, and with a comparison at the handover the logic of the inputs (NC/NO) could be easy corrected. The disadvantage of this option is the separate define of every variable and the manually add to the handover of the instance of every controller.

Example:

Global io definition with mapping:



Call fb instance:

fbCyl\_AB.In\_bolRetracted := In\_bolCyl\_AB\_Retract\_NO;

fbCyl\_AB.In\_bolExtended := (NOT In\_bolCyl\_AB\_Extend\_NC);

fbCyl\_AB();

Out\_bolCyl\_AB\_Retract := fbCyl\_AB.Out\_bolRetract;

Out\_bolCyl\_AB\_Extend := fbCyl\_AB.Out\_bolExtend;

Handling in FUB:

Method InBridge:

// digitale and analoge inputs read from hardware or forcing

SIf.In.bolExtended.map := THIS^.In\_bolExtend;

SIf.In.bolRetracted.map := THIS^.In\_bolRetract;

IF (CIf.Config.bolSimulation = FALSE) THEN

getInBol(CIf.Job.bolIoForce,ADR(SIf.In.bolExtended));

getInBol(CIf.Job.bolIoForce,ADR(SIf.In.bolRetracted));

ELSE

//simulation of global IO

END\_IF

Global function getInBol:

p\_Input^.state := p\_Input^.map; //usage?

IF (p\_Input^.EnForce OR bol\_GlobalEnableForce) THEN

p\_Input^.intern := p\_Input^.force;

ELSE

p\_Input^.intern := p\_Input^.map;

END\_IF;

getInBol := p\_Input^.intern;

Alternative to getInBol is the Global function getInBolDebounced available.

The input is accepted only after the debounce time (udiDebounceTime\_ms). If the signal changes during this time, the timer will be restarted.

Method OutBridge:

// digitale and analoge inputs read from hardware or forcing

IF (CIf.SwCfg.bolSimulation = FALSE) THEN

setOutBol(CIf.Job.bolIoForce ,ADR(SIf.Out.bolSpeed\_1));

setOutBol(CIf.Job.bolIoForce ,ADR(SIf.Out.bolSpeed\_2));

THIS^.Out\_bolSpeed\_1 := SIf.Out.bolSpeed\_1.map;

THIS^.Out\_bolSpeed\_2 := SIf.Out.bolSpeed\_2.map;

END\_IF

Global function setOutBol:

// Copy real values

p\_Output^.state := p\_Output^.intern;

IF (p\_Output^.EnForce AND bol\_GlobalEnableForce) THEN

p\_Output^.map := p\_Output^.force;

ELSE

p\_Output^.map := p\_Output^.intern;

END\_IF;

setOutBol := p\_Output^.map;

### Parameter handling

In the FB\_Init FUB the parameters will be initialized hardcoded in PLC code. So it’s sure that an initial value is every time set. Generally all parameters and configurations of the controllers will be written after start up by the UI (C#).

Parameters to be written from the UI at initialisation:

For every controller:

Controller Software Configuration SwCfg: T\_Ctrl\_SwCfg

Controller configuration Cfg: T\_Ctrl\_SIf\_CYL\_Cfg

Controller parameter Par: T\_Ctrl\_SIf\_CYL\_Par

p.e. for the stopper controller 1 (STP\_1):

* In STP\_1 -> CIf.SwCfg
* In STP\_1 -> SIf.Cfg
* In STP\_1 -> SIf.Par

### State machine

In the base controller the default state machine is defined. Every State is defined as a method. So in the inherited controllers is it possible to overwrite only one state and the codes overview are easier.

Example implementation of the state method:

//--------------------------------------------------------------------------------------

// Entry action

IF (me.StateMachine.eState<>me.StateMachine.eStateLastCycle) THEN

me.StateMachine.eStateLastCycle := me.StateMachine.eState;

SIf.Out.bolExtend.intern := TRUE;

SIf.Out.bolRetract.intern := FALSE;

fbSimulationTimer.IN := FALSE;

fbSimulationTimer.PT := UDINT\_TO\_TIME(SIf.Par.udiSimExtendTime\_ms);

fbSimulationTimer(); // set time/reset timer

fbSimulationTimer.IN := TRUE; // start timer

END\_IF

//--------------------------------------------------------------------------------------

// Cyclic action

IF (me.StateMachine.eState=me.StateMachine.eStateLastCycle) THEN

// Simulation

IF (CIf.Config.bolSimulation) THEN

fbSimulationTimer(); // simulation time

IF (fbSimulationTimer.Q) THEN

SIf.In.bolExtended.intern := TRUE;

SIf.In.bolRetracted.intern := FALSE;

END\_IF

END\_IF

// State Change Condition

IF (me.StateMachine.udiTimeInState\_ms>SIf.Par.udiExtendTime\_ms) THEN

IF (SIf.In.bolRetracted.intern) THEN

// sensor retract always on cylinder

ELSE

// timeout extend cylinder

END\_IF

me.StateMachine.eStateNext := E\_State.ERROR;

ELSIF (SIf.In.bolExtended.intern) AND (SIf.In.bolRetracted.intern=FALSE) THEN

me.StateMachine.eStateNext := E\_State.EXTEND;

ELSIF (me.eCmd=E\_Cmd.RETRACT) THEN

me.StateMachine.eStateNext := E\_State.RETRACTBusy;

END\_IF

END\_IF

//--------------------------------------------------------------------------------------

// Exit action

IF (me.StateMachine.eState<>me.StateMachine.eStateNext) THEN

SIf.Cur.udiMoveTimeExtend := me.StateMachine.udiTimeInState\_ms;

END\_IF

Substates

Every controller could have substates. Typically, these substates are used in states like Run.

Example declaration of the substates:

|  |  |
| --- | --- |
| *Variable* | *Datatype* |
| cSUB\_INITIALIZED | INT:=301 |
| cSUB\_VER\_Up | INT:=302 |
| cSUB\_VER\_Down | INT:=303 |
| cSUB\_WAIT\_Pick | INT:=304 |
| cSUB\_HOR\_MoveToPlace | INT:=305 |
| cSUB\_HOR\_MoveToPick | INT:=306 |
| cSUB\_Finished | INT:=307 |

They has to be initialized in the entry action of the calling Method.

Example implementation of the substate initialisation:

//--------------------------------------------------------------------------------------

// Entry action

IF (me.StateMachine.intState<>me.StateMachine.intStateLastCycle) THEN

me.StateMachine.intStateLastCycle := me.StateMachine.intState;

me.SubStateMachine.intState := cSTA\_SUB\_START\_CYCLE; // init sub state machine

me.SubStateMachine.intStateNext := cSTA\_SUB\_START\_CYCLE; // init sub state machine

END\_IF

//--------------------------------------------------------------------------------------

States and substates have to be write on in the method getCmdState, like the commands do.

## Example Level1 Cylinders (Class CYL)

Different pneumatic cylinders use this class. So this is a real common class.

**Functionality:**

* + Position switch one or two valves
  + Supervision of non, on or two sensor feedbacks

**Commands:**

* OFF (no air supply -> no errors from position switches)
* STOP (no movement, bistabil both do = 0, monostabil do nothing)
* RETRACT (cylinder move to retracted position)
* EXTEND (cylinder move to extended position)

**Config:**

* Type (1-2 outputs, 0-2 inputs)

**Parameter:**

* Timeout move to retract position [s]
* Timeout move to extend position [s]
* Simulation time until retracted [s] (for simulation / typ without sensor for this position)
* Simulation time until extended [s] (for simulation / typ without sensor for this position)

**Interlock:**

* Retract (-> STOP)
* Extend (-> STOP)

**Errors:**

* Timeout sensor position retracted not activated category: STOP
* Timeout sensor position extended not activated category: STOP
* Timeout sensor retracted still activated category: STOP
* Timeout sensor extended still activated category: STOP
* sensor retracted still deactivated category: STOP
* sensor extended still deactivated category: STOP
* Interlock retract active category: WARNING
* Interlock extend active category: WARNING

**IO:** DI: Position switch retracted

Position switch extended

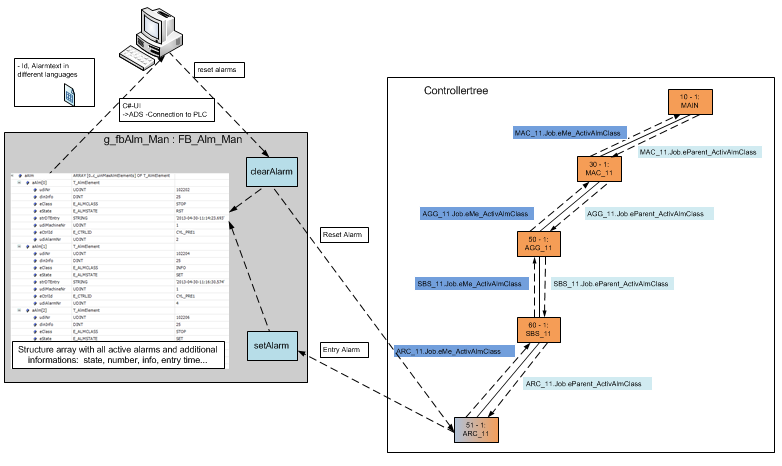
DO: Pneumatic valve retract

Pneumatic valve extend

Supervision of interlock where? And only warning or failure? What is an Interlock?

## Alarm handling

In the controller class the reaction of an alarm is defined. Over a method call (g\_fbAlm\_Man.setAlarm) the controller entries the alarm in the, from UI accessible, alarm active array. The method set the concerning alarm reaction class in the Job.eMe\_ActivAlmClass, if priority is higher than the already registered alarm class. Over this channel the parent controller could receive the highest alarm class and can decide if this alarm is also important for his parent or the other child controller. The other way is to write the highest active alarm class over the Job.eParent\_ActivAlmClass channel to the child controller. With this mechanism the controller react on the alarm class without regarding on modes or commands. The rules for the spreading of the reaction in the controller tree could be implemented in the concerning parent controllers.



In the FB\_Alm\_Man is the active alarm array defined:

aAlm(Array of T**\_**AlmElement):

The array size is limited with a constant of 100 elements (c\_uinMaxAlmElements: UINT := 99;). Every new activated alarm will be registered at the next free place in array. When the operator press the reset button all resettable alarms will be removed from the array and in the controller interface the reset bit will be set. This is resolved over a command channel (eAlmCmd) to [distinguish](http://dict.leo.org/?lp=ende&lang=de&searchLoc=0&cmpType=relaxed&relink=on&sectHdr=on&spellToler=std&search=bestimmt#/search=distinguish&searchLoc=0&resultOrder=basic&multiwordShowSingle=on) the service jobs from UI to alarm handler. The clear command calls the concerning method in the cyclic part of the alarm function block.

aAlmt [0.. c\_uinMaxAlmElements].

udiNr: UDINT; // unique alarm number := 100000\*udiMachineNr + eCtrlId \*100 + udiAlarmNr

dinInfo: DINT; // additional information - better string???

eClass: E\_AlmClass; // corresponding alarm class

eState: E\_AlmState; // clear, resetable

strDTEntry: STRING; // entry time

udiMachineNr: UDINT; // from machine (standard = 1)

eCtrlId: E\_CtrlId; // from controller

udiAlarmNr: UDINT; // local alarm number

E\_AlmState:

// no alarm active

NONE := 0,

// alarm clear entry -> set bit1

CLR := 1,

// entry resettable alarm -> set bit2

RST := 2,

// entry non resettable alarm -> set bit2 + bit3

SET := 3

E\_ALMClass:

// no alarm active

NONE := 0,

// info

INFO,

// warning

WARNING,

// tact stop

TSTOP,

// stop

STOP,

// off

OFF,

// Emergency off

EOFF

With this different type all possible cases for resetting or perhaps in further time to implement an alarm history on PLC can implemented. Normally an alarm with a stop reaction should not be cleared from alarm array (.CLR) without interaction of the UI with a reset. This is to prevent machine stops without a message to user. So only the information and warning alarms should use the clear in the controller! If the alarm cause in PLC is no more active the state has to be changed from non-resettable (.SET) to resettable (.RST)!

Examples:

Warning -> set alarm4:

IF (In\_bolEmpty) THEN

IF (SIf.Alm[4]. bol=FALSE) THEN

// set warning 4: conveyor is empty

g\_fbAlm\_Man.setAlarm(CIf.Info.eCtrlId,4,0, ADR(SIf.Alm), E\_AlmState.SET,CIf.Job.eMe\_ActivAlmClass);

SIf.Alm[4]. bol:= TRUE;

END\_IF:

ELSE

IF (SIf.Alm[4]. bol TRUE) THEN

// clear warning 4: conveyor is empty

g\_fbAlm\_Man.setAlarm(CIf.Info.eCtrlId,4,0,ADR(SIf.Alm), E\_AlmState.CLR,CIf.Job.eMe\_ActivAlmClass);

SIf.Alm[4]. bol:= FALSE;

END\_IF:

END\_IF

Alarm set in entry of error state:

// resettable error 5 with additional info 123: conveyor failure frequency inverter error number -> info:

g\_fbAlm\_Man.setAlarm(CIf.Info.eCtrlId,5,123, ADR(SIf.Alm), E\_AlmState.RST,CIf.Job.eMe\_ActivAlmClass);

Alarm sequence:

Any controller set an alarm over the method g\_fbAlm\_Man.setAlarm. The in the parameter of the called method given alarm event is entry in the Job.eMe\_ActivAlmClass if the priority is higher as the already set alarm class. This entry is read by the parent controller and there is the reaction of this alarm class implemented. So the active highest alarm class is spreading over the whole machine. The concerning controller level could decide if this alarm class goes further up or not and the reaction for the other child controllers. So this solution is flexible and application specific at the same time.

Alarm reaction

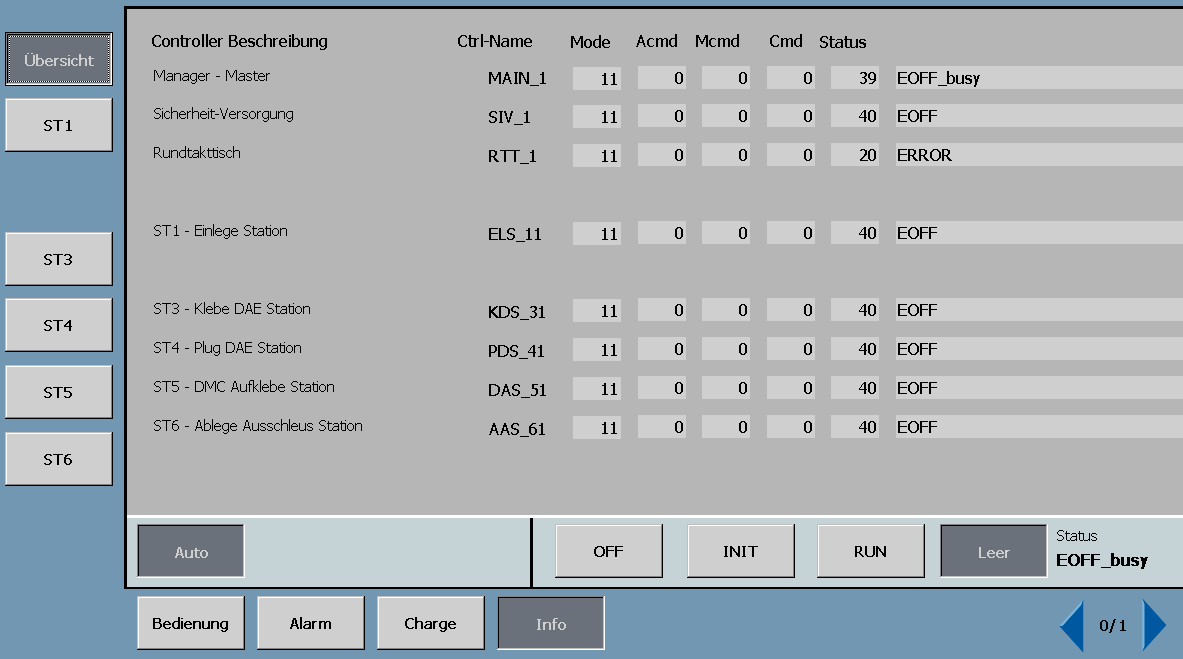
In some controllers are no EOFF-State needed. In this case, the controller has to go to the state cSTA\_OFFBusy instead of cSTA\_EOFFBusy. This behaviour should be solved in the method Always.

## Software info

For every defined controller:

* Show the common interface
* Possibility to set commands or change modes for administrator
* …

UI examples of software info:



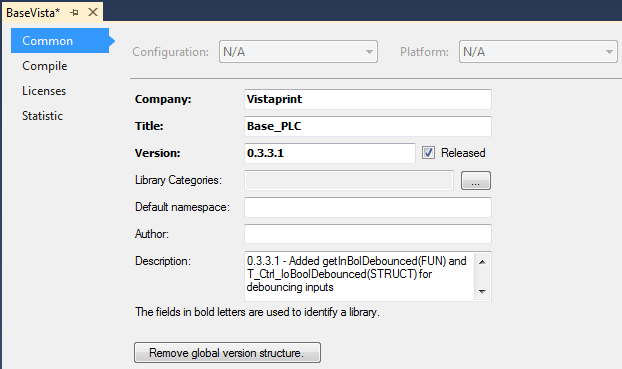
## Logger on PLC

For debugging and support it’s useful to have a state and command logger. Because of the time table, at the moment, such a logger will not be realized!

# PLC Software Release Management

## Version History

The version number and the associated comment have to be defined in the properties page of BaseVista Project. An example is shown in the following picture:



At the commit into SVN a version description with a version number in the message box must be written. This is necessary for a later finding of a certain version.

## Test before release

A new version of the library has to be tested in the BaseVistaSample Project. This is necessary because generate a library doesn’t check for every compiler error or warning. Only with the test in the sample project can guaranteed an error-free compilation. The BaseVistaSample Project acts also as test environment for new functions of the library.

## Dependencies

BASE\_PLC

BasePPT

BaseHts

BaseElements

BaseVistaSample Project

The BASE\_PLC is used in the BaseVistaSample Project, BaseElements, BaseHts, BasePPT and every specific project when needed.

BaseElements, BaseHts and BasePPT are just used, if in a solution needed. They haven’t any dependencies to each other.

In case of a new BASE\_PLC, all other library modules have to be updated with the new BASE\_PLC.

Todo…version in PLC string?? Other ideas