APPLICATION EXAMPLE

AC500 PACKML LIBRARY

1. Contents

1 Disclaimer

A. For customers domiciled outside Germany /

Für Kunden mit Sitz außerhalb Deutschlands

„Warranty, Liability:

The user shall be solely responsible for the use of this products described within this file. ABB shall be under no warranty whatsoever. ABB's liability in connection with application of the products or examples provided or the files included within this products, irrespective of the legal ground, shall be excluded. The exclusion of liability shall not apply in the case of intention or gross negligence. The present declaration shall be governed by and construed in accordance with the laws of Switzerland under exclusion of its conflict of laws rules and of the Vienna Convention on the International Sale of Goods (CISG)."

„Gewährleistung und Haftung:

Der Nutzer ist allein für die Verwendung des in diesem Dokument beschriebenen Produkte und beschriebenen Anwendungsbeispiele verantwortlich.

ABB unterliegt keiner Gewährleistung. Die Haftung von ABB im Zusammenhang mit diesem Anwendungsbeispiel oder den in dieser Datei enthaltenen Dateien - gleich aus welchem Rechtsgrund - ist ausgeschlossen. Dieser Ausschluss gilt nicht im Falle von Vorsatz oder grober Fahrlässigkeit. Diese Erklärung unterliegt Schweizer Recht unter Ausschluss der Verweisungsnormen und des UN-Kaufrechts (CISG)."

B. Nur für Kunden mit Sitz in Deutschland

„Gewährleistung und Haftung:

Die in diesem Dokument beschriebenen Anwendungsbeispiele oder enthaltenen Dateien beschreiben eine mögliche Anwendung der AC500 bzw. zeigen eine mögliche Einsatzart. Sie stellen nur Beispiele für Programmierungen dar, sind aber keine fertigen Lösungen. Eine Gewähr kann nicht übernommen werden.

Der Nutzer ist für die ordnungsgemäße, insbesondere vollständige und fehlerfreie Programmierung der Steuerungen selbst verantwortlich. Im Falle der teilweisen oder ganzen Übernahme der Programmierbeispiele können gegen ABB keine Ansprüche geltend gemacht werden.

Die Haftung von ABB, gleich aus welchem Rechtsgrund, im Zusammenhang mit den Anwendungsbeispielen oder den in dieser Datei enthaltenen Beschreibung wird ausgeschlossen. Der Haftungsausschluss gilt jedoch nicht in Fällen des Vorsatzes, der groben Fahrlässigkeit, bei Ansprüchen nach dem Produkthaftungsgesetz, im Falle der Verletzung des Lebens, des Körpers oder der Gesundheit oder bei schuldhafter Verletzung einer wesentlichen Vertragspflicht. Im Falle der Verletzung einer wesentlichen Vertragspflicht ist die Haftung jedoch auf den vertragstypischen, vorhersehbaren Schaden begrenzt, soweit nicht zugleich ein anderer der in Satz 2 dieses Unterabsatzes erwähnten Fälle gegeben ist. Eine Änderung der Beweislast zum Nachteil des Nutzers ist hiermit nicht verbunden.

Es gilt materielles deutsches Recht unter Ausschluss des UN-Kaufrechts."

2 Introduction

2.1 Scope of the document

This documentation is a guide to walk the user through the function blocks available, display their features and explain their role in standardizing the machines to PackML.

2.2 Compatibility

The application example explained in this document have been used with the below engineering system versions. They should also work with other versions, nevertheless some small adaptations may be necessary, for future versions.

- AC500 V2 PLC
- Automation Builder 1.2.0 or newer

2.3 Overview

This a guide to walk the user through the function blocks available, display their features and explain their role in standardizing the machines to PackML, followed by the description of the application example.

3 Overview of PackML Library

To begin it is necessary to get a quick inside on the PackML standardization.

3.1 What is PackML?

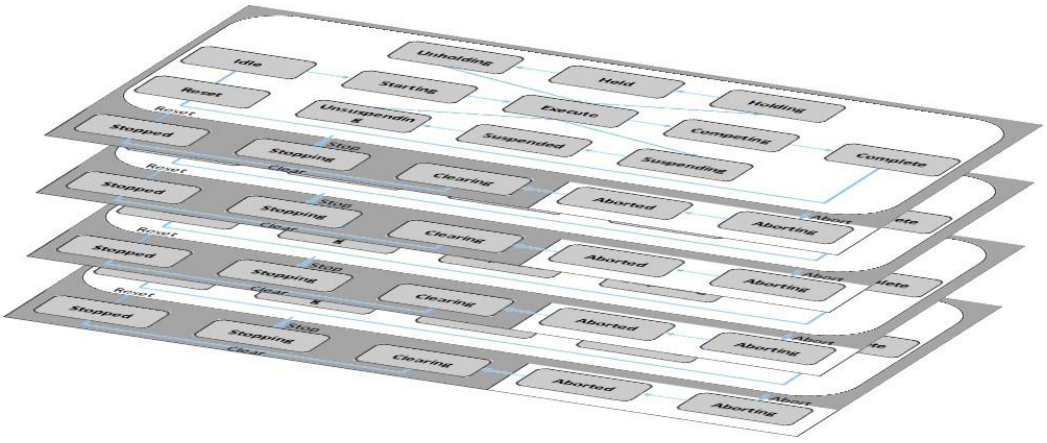
PackML, which stands for Packaging Machine Language, defines a common approach, or machine language, for automated machines. The primary goals are to encourage a common "look and feel" across a plant floor and to enable and encourage industry innovation. PackML was adopted as part of the ISA88 industry standard in August 2008 and has been implemented by users and machine builders on a wide variety of control platforms. Those implementing PackML are realizing cost benefits of higher reliability, better supply chain integration, reduced engineering and training costs, and shorter project cycles.

PackTags – PackML define the communication between different machines or a supervisory control system by the use of standardized variables. They are divided into Control Tags, Status Tags and Admin Tags. In this way communication with a line control system or M2M communication is possible, regardless of the field bus type used or from which manufacturer the components are produced.

A list with all PackTags can be seen in the attachments.

Unit, Equipment Module, Control Module – Based on ISA88 an assembly or a production machine is divided into these modules, forming a hierarchical order. The machine, named Unit, controls several of its Equipment Modules (EM) which further on control their Control Modules (CM). An Equipment Module can be an Infeed section or an labeler etc., a Control Module can be a drive or a sensor, etc.

TR88 also has predefined 3 mode setups, the production, maintenance and manual mode. In general, each mode consists of user definable setup of states. In total he can choose of a maximum number of 17 states per mode.



The PackML Library is a toolbox, intended for standardizing projects. It is supposed to simplify maintenance and support, minimize implementation work and lower costs. It also allows for the next user to have a program that is systematic and friendly to use. It contains all basic function blocks to build up a new project (e.g. a production line) with the PackML standard, or adapt your already existing project to the PackML standard.

3.2 Solution Content

The user Guide is equipped with a Library that includes all the essential function blocks needed to adapt PackML to your project, as well as a template for the AC500 including a template for webserver and CP600 based HMI.

3.2.1 Library

The library includes a variety of function blocks. They are supposed to be used as tools to adapt your project to the PackML standard. To explain them they are divided into three groups: Event Handling, Mode & State Manager and the Operating Times blocks.

3.2.2 AC500 Template

To see the functions of the library in action a template with a unit, two equipment modules and two control modules is included. Within this guide the steps for creating the template are explained together with the connections to the library.

3.2.3 Webserver Template

The CoDeSys visualization is used for monitoring and operating a created control program. The Web visualization is a target specific application. CoDeSys can process the objects in a way that when downloading them to a controller they are in the XML-Format. There the Webserver controls the data also in the XML-Format and provides a continuously updated visualization. With this anyone can connect via an Internet platform to the visualization.

3.2.4 CP600 Template

In general, the whole PackML visualization functionalities are done by HMI function blocks. These blocks are all not integrated in the appropriate function POUs. They are separated for more flexibility. Depending on the utilized CPU it can be possible that the HMI memory is very limited. With the standalone HMI blocks the user can decide if visualizations are needed and if so which ones.

Using the Panel Builder software, a template for the CP600 was created so the user also has this option of an HMI visualization.

The template pages are created with widgets from the Widget Gallery. All pages have blue navigation bar on the top and if available grey command buttons on the right.

On the bottom of each page there is a display which shows the date and time, the current mode and the current state. It also shows in red the number of active events (if there are any), in yellow the number of total events (only if all events are not triggered), and “no events” if there are currently no alarms or warnings triggered.

3.3 First Steps

1. Install the latest version of the ABB Automation Builder.
2. Establish Ethernet communication.
3. When adapting PackML to already existing Code, decide whether to change the Code and divide it into Modules or use it all as the Unit Code.
4. When building a new process with the PackML standard, refer to the “creating new Project” chapter.

3.4 Preconditions for the Use of the PackML Library

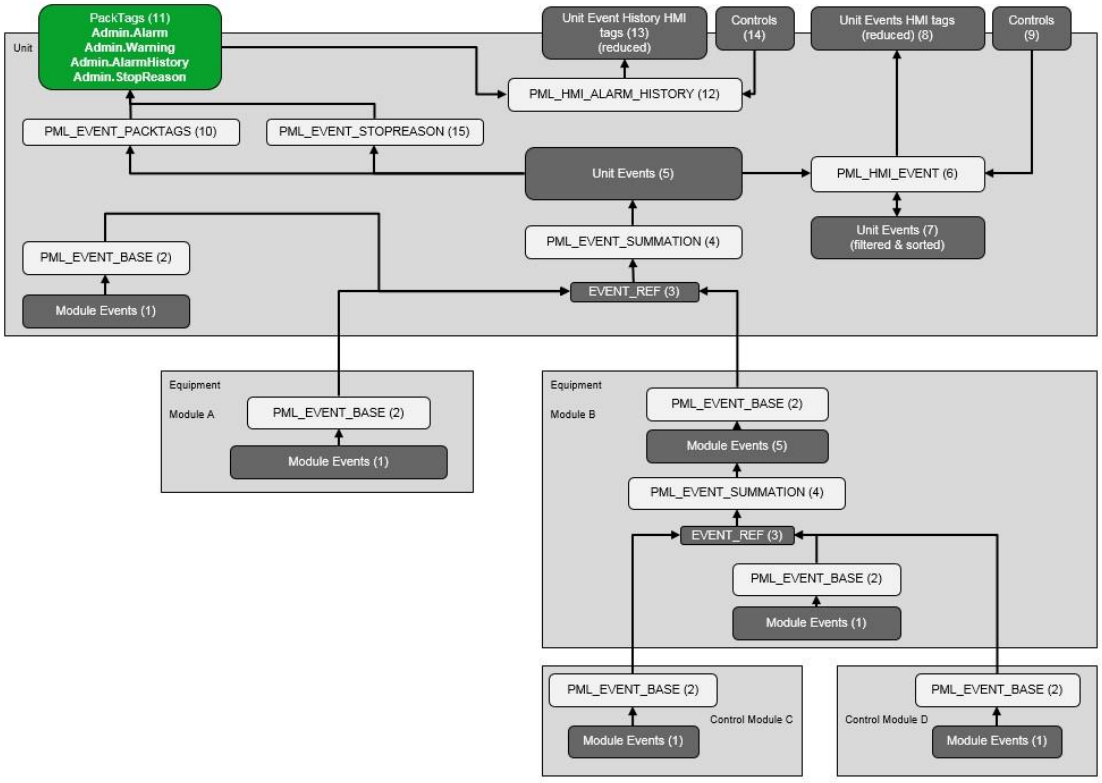
	The Function Blocks of the PackML Library are only working in the RUN mode of the PLC. Usage of these libraries in the simulation mode may not provide any valid or usable diagnostic information.

4 Event Handling of PackML Library

This chapter is relevant if alarms or warnings can occur during the course of production, maintenance or any process.

4.1 Usage of Event Function Blocks

The following figure shows how the main data flow can be seen for a layout example of a modularized project.



1. For each module a maximum number of configured or storable events can be declared in its program code. The size of the array is only limited by the resources of the PLC. An example declaration is shown below:

```
VAR
    EventsList: ARRAY [0..9] OF PML_EVENT_CFG_TYPE;
END_VAR
```

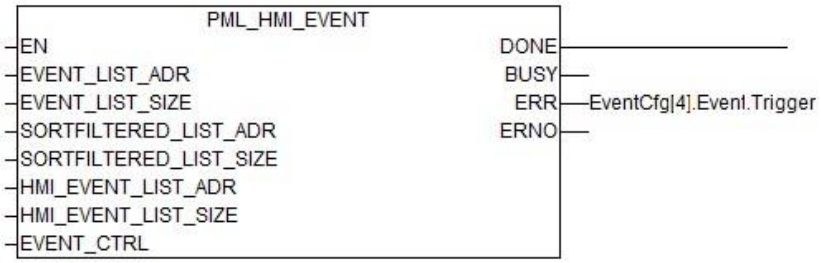
The content of the data type PML_EVENT_CFG_TYPE is shown below:

```
TYPE PML_EVENT_CFG_TYPE :
STRUCT
    Event           : PML_EVENT_TYPE;
    EventType       : BOOL;
    PathArray       : ARRAY [0..9] OF INT;
    PathString      : STRING(80);
    NumOccurred     : INT := 0;
    AlmDateTime     : DATE_AND_TIME;
    AckDateTime     : DATE_AND_TIME;
    Subroutine      : STRING;
END_STRUCT END_TYPE
```

Whereas the content of data type PML_EVENT_TYPE is:

```
TYPE PML_EVENT_TYPE :
STRUCT
    Trigger         : BOOL          := FALSE;
    ID              : DINT          := 0;
    Value           : DINT          := 0;
    Message         : STRING[30]    := '';
    Category        : DINT          := 0;
    AlmDateTime     : ARRAY [0..6] OF DINT;
    AckDateTime     : ARRAY [0..6] OF DINT;
END_STRUCT END_TYPE
```

To activate an event there is a variable named “Trigger” of type BOOL, which is set to TRUE as soon as an event condition is true. The next figure shows an implementation example:



If the function block sets its Error output to TRUE, the fifth configured event is triggered.

More elements of the event can be assigned before or in that moment the trigger is set, or already during declaration. Such as:

- ID — DINT, Globally unique number for this event. Mandatory when events are summated from different tasks where they are defined in.
- Value — DINT, Additional information such as error number from a function block.
- Message — STRING, general message of this event
- Category — DINT, Severity numbers (0..9) can be defined to cause fault reactions
- EventType — BOOL, Defines if this event is an Alarm (TRUE) or Warning (FALSE)
- Subroutine — STRING, Information about the subroutine e.g. a function block.

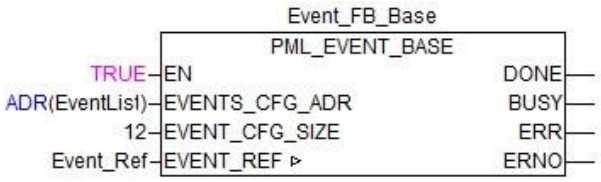
The following figure shows an example how events can be predefined by declaration:

```
Event_FB_Base_1     : PML_EVENT_BASE;
EventCfg            : ARRAY [0..1] OF PML_EVENT_CFG_TYPE := (Event :=(Id := 11, Value:=11, Category:=1,  Message:='Battery low'),  EventType:=FALSE,  Subroutine:= 'Time_FB_Manager'),
                                                            (Event :=(Id := 12, Value:=11, Category:=2,  Message:='Time Rollover'), EventType:=FALSE,  Subroutine:= 'Time_FB_Manager');
```

On the lowest level of each path, the PathString and the PathArray need to be defined as well.

```
Event_AR_List   : ARRAY [0..1] OF PML_EVENT_CFG_TYPE := (Event  :=(Id := 111, Value:=11,  Category:=1,  Message:='EM1_Alarm'),   PathString  := 'EM1',  PathArray := 1,0,0,0,0,0,0,0,0,0,
                                                         (Event  :=(Id := 112, Value:=11,  Category:=0,  Message:='EM1_Warning'), PathString  := 'EM1',  PathArray := 1,0,0,0,0,0,0,0,0,0,
```

5. Each event list or event configuration list must be connected with a function Block of PML_EVENT_BASE. It will observe the state of the Trigger variables in the connected list. If a change in the state of the trigger is detected the event will get a timestamp and will be copied to a variable of type PML_EVENT_REF_TYPE.



6. The variable of type PML_EVENT_REF_TYPE can be seen as a mailbox which receives events from all connected function blocks of type PML_EVENT_BASE. There is no limitation about the number of function blocks of type PML_EVENT_BASE which can be connected to the same variable of type PML_EVENT_REF_TYPE.
7. The function block PML_EVENT_SUMMATION will read all events from the variable of type PML_EVENT_REF and summates them in a new list which represents all events of this module.

   When an event passes the summation FB, module specific information will be added to the event variable. “PathString” (STRING) in order to provide the user of the machine with information about the source of a displayed event.

   It also has outputs which give general information about the current event summation from a module. E.g. that there is an active event or not or which categories the triggered events have. This can be used for fault reactions or HMI visualization.
8. This event list is declared (in the same way as all the other event lists) as an ARRAY [0..n] of type PML_EVENT_CFG_TYPE. In general, all event lists can be summated again from another module by the use of a function block of type PML_EVENT_BASE. The size is only limited by PLC resources. Have in mind that there is no need to have a larger list than the

maximum of possible events which can be summated. Smaller lists are also possible whereas in case an incoming event cannot be stored in the list, its category will still be taken into account to determine the “EVENT_CATEGORY” output at the summation function blocks.

The Trigger variables from event lists which are filled by a summation function block may not be set from application’s event condition.

The sequence of events within these lists is determined by the chronological order when the events arrive during operation and not by the time they were triggered.

9. When there is no further summation the list represents all triggered events of this machine since the last reset of the event system. The function block of type PML_HMI_EVENT can be used to filter and sort these events in a separate list (7). It also copies a selection of alarms to an event list which can be used as HMI tag (8). There is also a data type for a variable which should be used as HMI tag to have control the event display from the HMI.
10. This event list is declared as an ARRAY [0..n] OF TYPE PML_EVENT_CFG_TYPE and contains filtered events in a sorted order. It must be the same size as the source list (5).
11. This event list is declared as an ARRAY [0..n] OF TYPE PML_EVENT_CFG_TYPE and contains events which are displayed on HMI. The size of this list can be configured and depends on how many events should be displayed on HMI at the same time.
12. Variable of type PML_HMI_EVENT_CTRL_TYPE which contains necessary controls for the HMI. It should be used as an HMI tag and is connected to the function block PML_HMI_EVENT.
13. The function block PML_EVENT_PACKTAGS is used to extract the PACKML relevant event information from the main event list and copy them to the PackTags.
14. The event PackTags will represent the events as defined within PackML.
15. The function block PMI_HMI_ALARM_HISTORY reduces the list of PackML alarm history list to an extract which can be shown on an HMI. An instance of an additional control data type is connected to this block and used as
16. HMI tag in order to provide control elements for the user
17. This event list is declared as an ARRAY [0..n] OF TYPE PML_EVENT_TYPE and can be used as HMI tag
18. Variable of type PML_HMI_EVENT_CTRL_TYPE which contains necessary controls for the HMI. It should be used as an HMI tag and is connected to the function block PMI_HMI_ALARM_HISTORY.
19. The function block PML_EVENT_STOPREASON is used to select events from the event list depending on their category or their AlarmDateTime and copy them to the PackTags.

Additional information to (2):

As mentioned before the base FB is used to send the configured events one by one through the mailbox, to eventually end up at the SUMMATION function block. The normal operation of the base function block is to check all events whether their Trigger has been set to TRUE since the last cycle. This indicates that the event is ready to be send to the mailbox. In that case another condition has to be checked: Is the referring mailbox empty (EVENT_REF.Num_1 and EVENT_REF.Num_2 are both 0). There are two spots where the event writes its ID in, so all interruptions between tasks won’t result in an event coping itself into the mailbox when another event has already started the process. The two variables are only set back to 0 when the summation function block has copied the event into its list.

On a Reset the base function block sets all Triggers to False so they need to rewrite themselves into the mailbox if they still occur after the reset.

Attention:

All inputs named …_ADR needs to be connected to a variable that is an array of the correct event Datatype. The variable to the …_SIZE input needs to be the exact size of the array.

Events from master list to HMI:

The HMI Event function block (PML_HMI_EVENT) uses the master list, created by the summation function block, to sort and filter the list according to the user's expectations. Therewith he is able to filter according status, event type, value, category and alarm time and sort according to status, event type, category and Alarm Time. One by one the events that fulfill the requests are copied to another list.

The events continue through the HMI function block to a third list, which will be displayed on the HMI. It only copies the number of events of the size of the HMI event list.

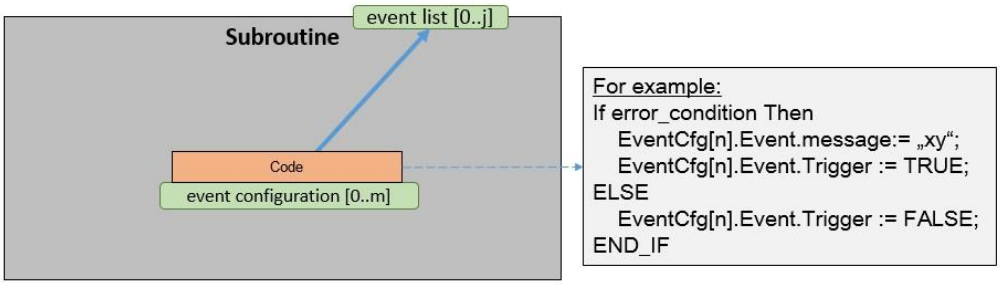
Events from master list to PackTags:

When declaring the event it was divided into the PackTags and the user defined part. In this step only the PackTags defined part gets copied to a separate alarm and warning list, depending on event type. When the command update list is executed, the warnings and alarms are copied to the appropriate list. If the command clear list is executed, then the alarm list is copied to the alarm history list.

The separate HMI alarm history FB is used to display the history alarm list on the HMI according to the same principle as displaying the events to the HMI.

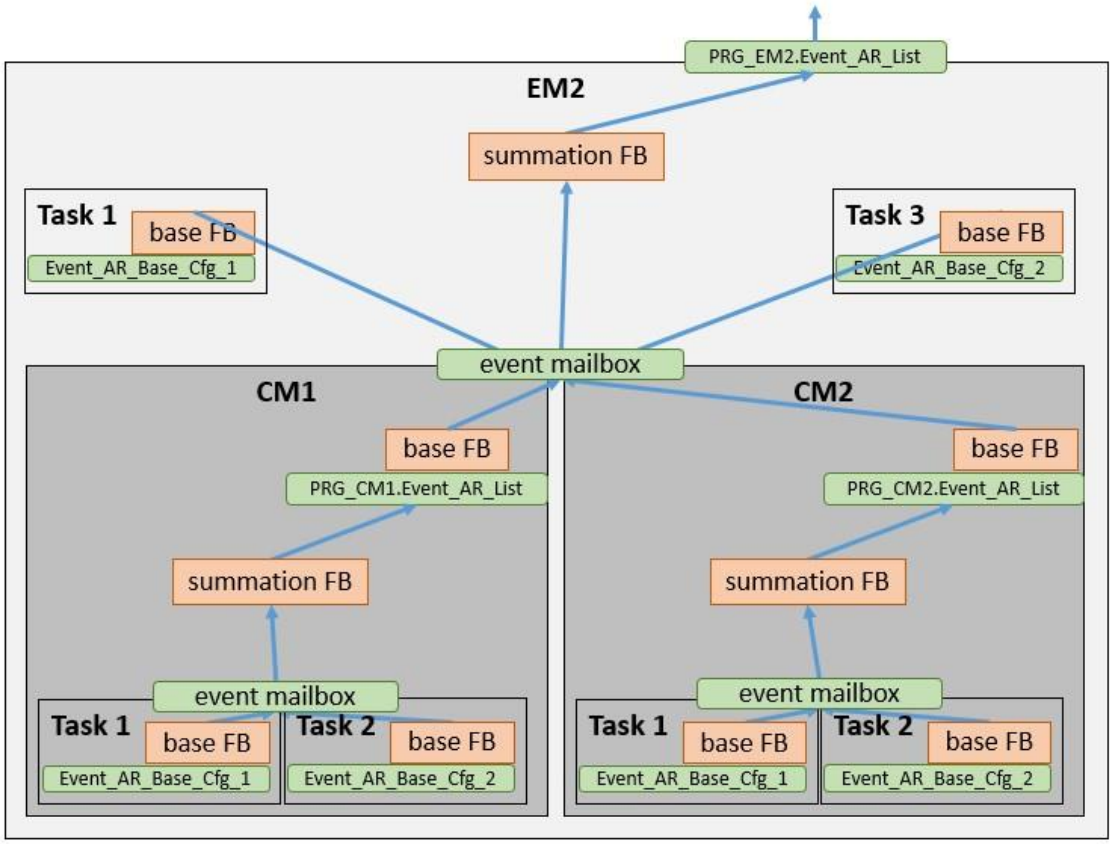
4.1.1 Event handling in subroutines

In subroutines with only one task, where multitasking is not an issue, base and summation function blocks are not needed. This is where the "subroutine" variable of the datatype PML_EVENT_CFG_TYPE comes into play. Instead of the summation function block adding a prefix and path to the event, the event writes the current subroutine. It is defined in the configuration. It then goes directly into an event list.

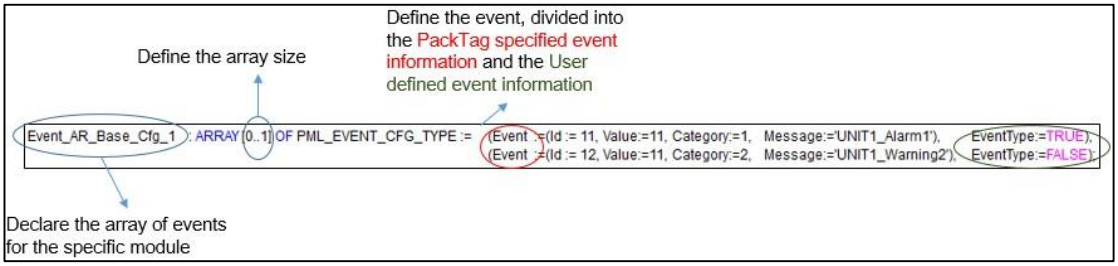


4.1.2 Template Example

This template project includes a unit with two Equipment Modules and one of the Equipment Module contains two Control Modules. Neither the whole application is running in one task (as implemented in the template project) nor different programs are running in multiple tasks (as shown in the scheme below), there is no influence on the functionality of the function blocks.



The events defaults are configured, as shown below, in the module code / program where they occur. The list that is created by the summation FB is sent to the upper layer mailbox. This mailbox gets used in the interface of the module and the next module.

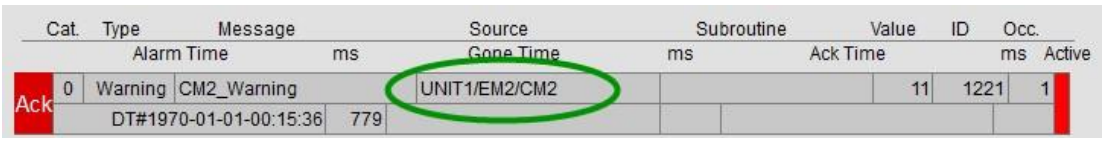


All events will end up in a Unit master list. The template project includes the HMI function block to display the events. Additionally, the HMI function block offers the possibility to filter and sort the event list to the user's inputs. However, it can display any list it has connected to the EVENT_LIST_ADR input. The interface for the visualization event lists is the HMI_EVENT_LIST_ADR input.

For the MODULE_CFG input of the summation function block, the module, where the list is stored, needs to be declared, including ID and Name (see figure below). The purpose for this is to track the events path.

```
Event_DT_Summation_Cfg      : PML_EVENT_MODULE_CFG_TYPE      :=      (ID := 1,     NAME := 'UNIT1');
```

The following extract of the event list visualization shows a tracking example of a CM event.



4.2 Covered PackTags

			Tag Name	Datatype
Alarm[#]			UnitName.Admin.Alarm[#]	Alarm Structure
	Trigger		UnitName.Admin.Alarm[#].Trigger	Bool
	ID		UnitName.Admin.Alarm[#].ID	Int (32bit)
	Value		UnitName.Admin.Alarm[#].Value	Int (32bit)
	Message		UnitName.Admin.Alarm[#].Message	String
	Category		UnitName.Admin.Alarm[#].Category (Event Grouping)	Int (32bit)
	AlmDate-Time		UnitName.Admin.Alarm[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Alarm[#].AlmDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.Alarm[#].AlmDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.Alarm[#].AlmDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Alarm[#].AlmDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Alarm[#].AlmDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Alarm[#].AlmDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Alarm[#].AlmDateTime[6]	Int (32bit)
	AckDate-Time		UnitName.Admin.Alarm[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Alarm[#].AckDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.Alarm[#].AckDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.Alarm[#].AckDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Alarm[#].AckDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Alarm[#].AckDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Alarm[#].AckDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Alarm[#].AckDateTime[6]	Int (32bit)
Alarm-Extent			UnitName.Admin.AlarmExtent	Int(32bit)
Alarm-History[#]			UnitName.Admin.AlarmHistory[#]	Alarm Structure
	Trigger		UnitName.Admin.AlarmHistory[#].Trigger	Bool
	ID		UnitName.Admin.AlarmHistory[#].ID	Int (32bit)
	Value		UnitName.Admin.AlarmHistory[#].Value	Int (32bit)
	Message		UnitName.Admin.AlarmHistory[#].Message	String

	Category		UnitName.Admin.AlarmHistory[#].Category (Event Grouping)	Int (32bit)
	AlmDa-teTime		UnitName.Admin.AlarmHistory[#]	Date-Time Array
		[0] (year)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[0]	Int (32bit)
		[1] (month)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[1]	Int (32bit)
		[2] (day)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[3]	Int (32bit)
		[4] (min)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.AlarmHistory[#].AlmDate-Time[6]	Int (32bit)
	AckDate-Time		UnitName.Admin.AlarmHistory[#]	Date-Time Array
		[0] (year)	UnitName.Admin.AlarmHistory[#].AckDate-Time[0]	Int (32bit)
		[1] (month)	UnitName.Admin.AlarmHistory[#].AckDate-Time[1]	Int (32bit)
		[2] (day)	UnitName.Admin.AlarmHistory[#].AckDate-Time[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.AlarmHistory[#].AckDate-Time[3]	Int (32bit)
		[4] (min)	UnitName.Admin.AlarmHistory[#].AckDate-Time[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.AlarmHistory[#].AckDate-Time[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.AlarmHistory[#].AckDate-Time[6]	Int (32bit)
AlarmHis-toryExtent			UnitName.Admin.AlarmHistoryExtent	Int (32bit)
Stop-Reason			UnitName.Admin.StopReason	Alarm Structure
	Trigger		UnitName.Admin.StopReason.Trigger	Bool
	ID		UnitName.Admin.StopReason.ID	Int (32bit)
	Value		UnitName.Admin.StopReason.Value	Int (32bit)
	Message		UnitName.Admin.StopReason.Message	String
	Category		UnitName.Admin.StopReason.Category (Event Grouping)	Int (32bit)

	AlmDate-Time		UnitName.Admin.StopReason[#]	Date-Time Array
		[0] (year)	UnitName.Admin.StopReason[#].AlmDate-Time[0]	Int (32bit)
		[1] (month)	UnitName.Admin.StopReason[#].AlmDate-Time[1]	Int (32bit)
		[2] (day)	UnitName.Admin.StopReason[#].AlmDate-Time[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.StopReason[#].AlmDate-Time[3]	Int (32bit)
		[4] (min)	UnitName.Admin.StopReason[#].AlmDate-Time[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.StopReason[#].AlmDate-Time[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.StopReason[#].AlmDate-Time[6]	Int (32bit)
	AckDate-Time		UnitName.Admin.StopReason[#]	Date-Time Array
		[0] (year)	UnitName.Admin.StopReason[#].AckDate-Time[0]	Int (32bit)
		[1] (month)	UnitName.Admin.StopReason[#].AckDate-Time[1]	Int (32bit)
		[2] (day)	UnitName.Admin.StopReason[#].AckDate-Time[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.StopReason[#].AckDate-Time[3]	Int (32bit)
		[4] (min)	UnitName.Admin.StopReason[#].AckDate-Time[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.StopReason[#].AckDate-Time[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.StopReason[#].AckDate-Time[6]	Int (32bit)
StopRea-sonExtent			UnitName.Admin.StopReasonExtent	Int (32bit)
Warning[#]			UnitName.Admin.Warning[#]	Alarm Structure
	Trigger		UnitName.Admin.Warning[#].Trigger	Bool
	ID		UnitName.Admin.Warning[#].ID	Int (32bit)
	Value		UnitName.Admin.Warning[#].Value	Int (32bit)
	Message		UnitName.Admin.Warning[#].Message	String
	Category		UnitName.Admin.Warning[#].Category (Event Grouping)	Int (32bit)
	AlmDate-Time		UnitName.Admin.Warning[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Warning[#].AlmDateTime[0]	Int (32bit)

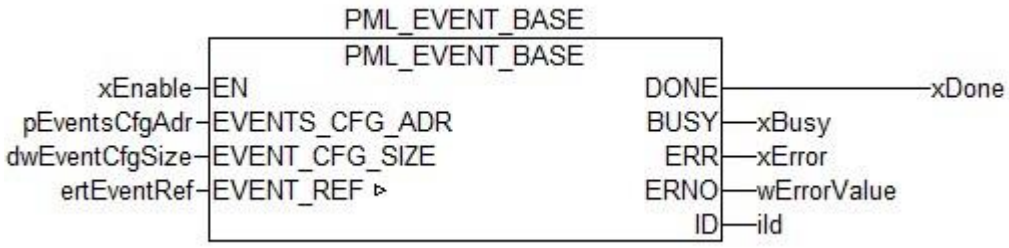
		[2] (day)	UnitName.Admin.Warning[#].AlmDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Warning[#].AlmDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Warning[#].AlmDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Warning[#].AlmDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Warning[#].AlmDateTime[6]	Int (32bit)
	AckDate-Time		UnitName.Admin.Warning[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Warning[#].AckDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.Warning[#].AckDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.Warning[#].AckDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Warning[#].AckDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Warning[#].AckDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Warning[#].AckDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Warning[#].AckDateTime[6]	Int (32bit)
Warning-Extent			UnitName.Admin.WarningExtent	Int (32bit)

4.2.1 Event Function Blocks

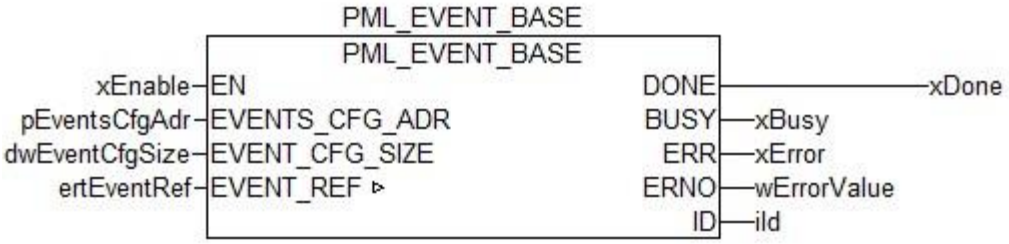
All event function blocks are used to transfer events to different positions in the module hierarchy.

4.2.1.1 PML_EVENT_BASE

The event base function block is used to configure events and accordingly registers them in the mailbox. It continually includes measures that prevent errors when multitasking. When it has completed the coping process, it releases the event to be read by the module. Once it has gotten the command that the master has read the event and the mailbox is emptied, it also sets all triggers to False if there is a Reset command.



4.2.1.1.1 Input Description



The inputs marked with a triangle ▷ are of the class VAR_IN_OUT (input and output variable). These inputs must be connected to a variable.

EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

EVENT_CFG_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This input can be connected to the address of the configured events or an event list.

EVENT_CFG_SIZE

Data type: DWORD

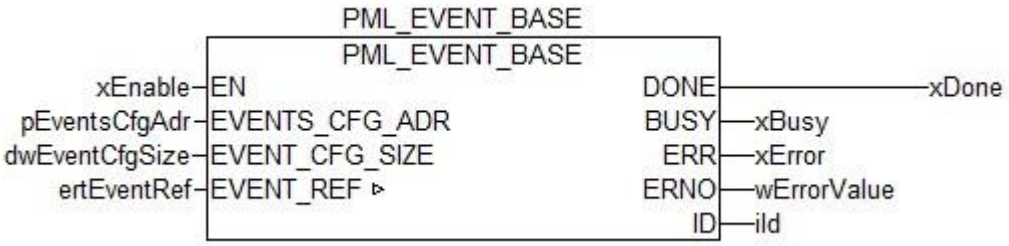
Size of the connected event array.

EVENT_REF

Data type: PML_EVENT_REF_TYPE

This input is from the datatype Event Ref, the mailbox of a module that takes care of event exchange between two or more modules. The base function block sends events to upper modules.

4.2.1.1.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

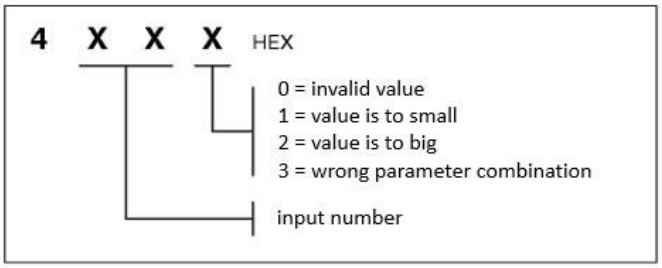
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



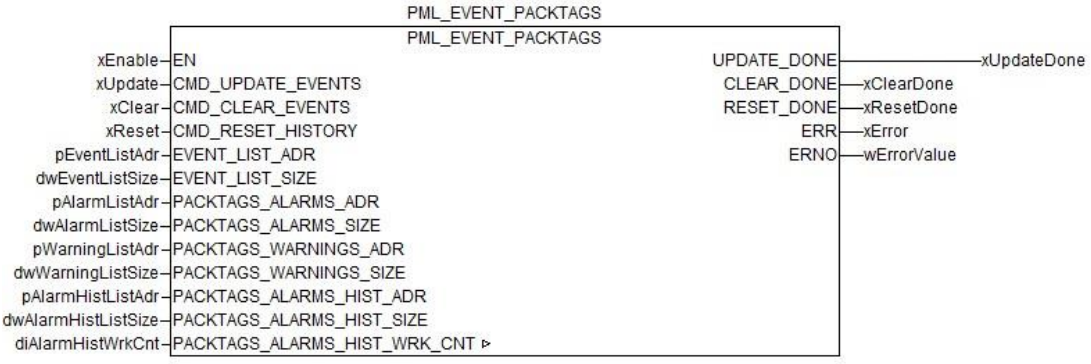
ID

Data type: INT

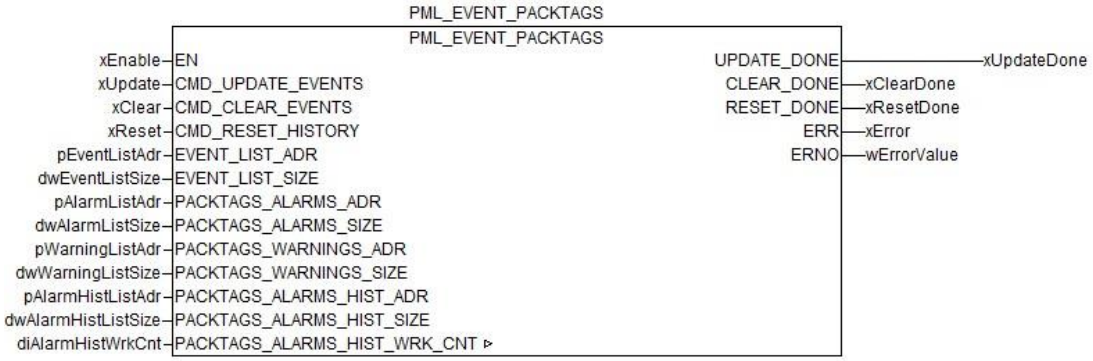
Internally given ID to clearly indicate a Base function block and its referring events. The ID is unique for each Base function block connected to one mailbox of EVENT_REF_TYPE.

4.2.1.2 PML_EVENT_PACKTAGS

The event PackTags function block is used to copy the part of the event information that is PackTags certified to a separate warning and alarm list, depending on their event type.



4.2.1.2.1 Input Description



	The inputs marked with a triangle ▷ are of the class VAR_IN_OUT (input and output variable). These inputs must be connected to a variable.

EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

CMD_UPDATE_EVENTS

Data type: BOOL

If this input is set to TRUE, the function block extracts the PackML specified event information for each event from the master list and copies them to separate alarm and warning lists.

CMD_CLEAR_EVENTS

Data type: BOOL

This input is set to TRUE when all alarms are to be copied from the alarm list to the alarm history list and the alarm and warning list are to be cleared.

CMD_RESET_HISTORY

Data type: BOOL

This input is set to TRUE when the history alarm list is to be cleared and the internal working counter for adding alarms to history is to be reset.

EVENT_LIST_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This input can be connected to the address of the master event list.

EVENT_LIST_SIZE

Data type: DWORD

Size of master list that is connected at the corresponding ..ADR input.

PACKTAGS_ALARMS_ADR

Data type: POINTER TO PACKML_EVENT_TYPE

This input can be connected to the address of the alarm list where the PackTag certified part of the event information will be copied to if the event type is TRUE (=Alarm).

PACKTAGS_ALARMS_SIZE

Data type: DWORD

Size of alarm list that is connected at the corresponding ..ADR input.

PACKTAGS_WARNINGS_ADR

Data type: POINTER TO PACKML_EVENT_TYPE

This input can be connected to the address of the warning list where the PackTag certified part of the event information will be copied to if the event type is FALSE (=Warning).

PACKTAGS_WARNINGS_SIZE

Data type: DWORD

Size of alarm list that is connected at the corresponding ..ADR input.

PACKTAGS_ALARMS_HIST_ADR

Data type: Pointer to PACKML_EVENT_TYPE

This input can be connected to the address of the alarm history list where the PackTag certified part of the event information will be copied to if the clear command input is TRUE.

PACKTAGS_ALARMS_HIST_SIZE

Data type: DWORD

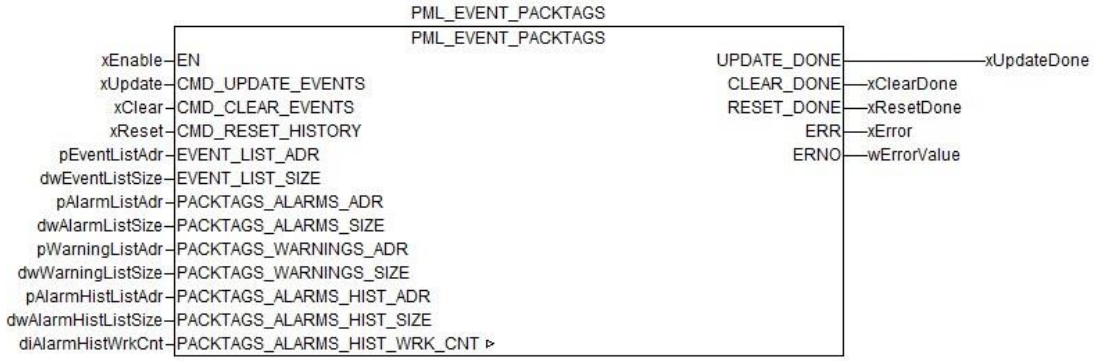
Size of alarm history list that is connected at the corresponding ..ADR input.

PACKTAGS_ALARM_HIST_WRK_CNT

Data type: DINT

Working counter for alarm history list, which indicates how many events are stored in the alarm history list.

4.2.1.2.2 Output Description



UPDATE_DONE

Data type: BOOL

Output indicates the processing state of the block. As soon as the update routine has been finished the output will be set to TRUE.

CLEAR_DONE

Data type: BOOL

Output indicates the processing state of the block. As soon as the clear routine has been finished the output will be set to TRUE.

RESET_DONE

Data type: BOOL

Output indicates the processing state of the block. As soon as the reset routine has been finished the output will be set to TRUE.

ERR (error)

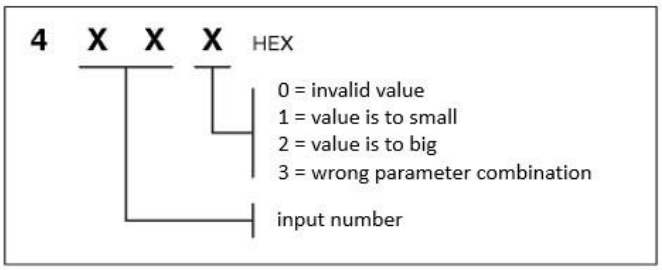
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

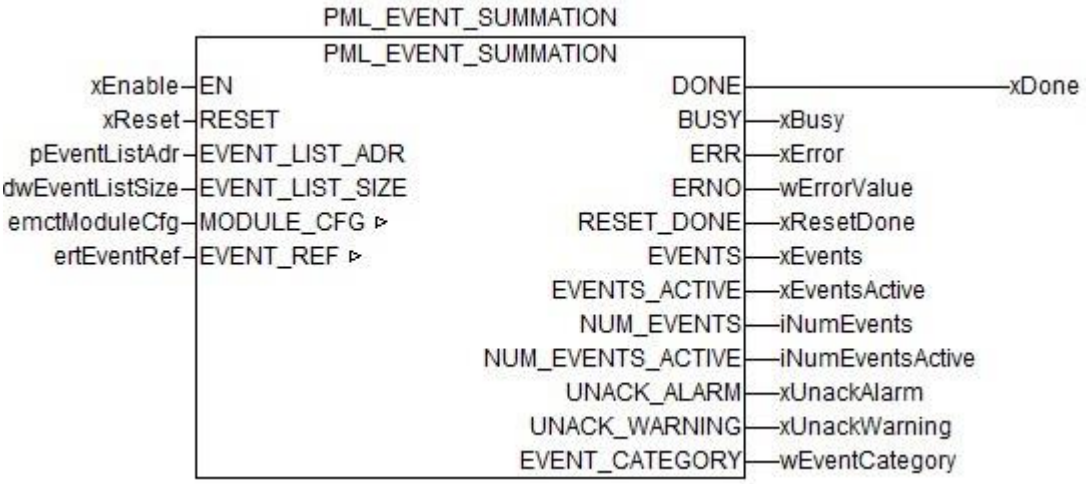
Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



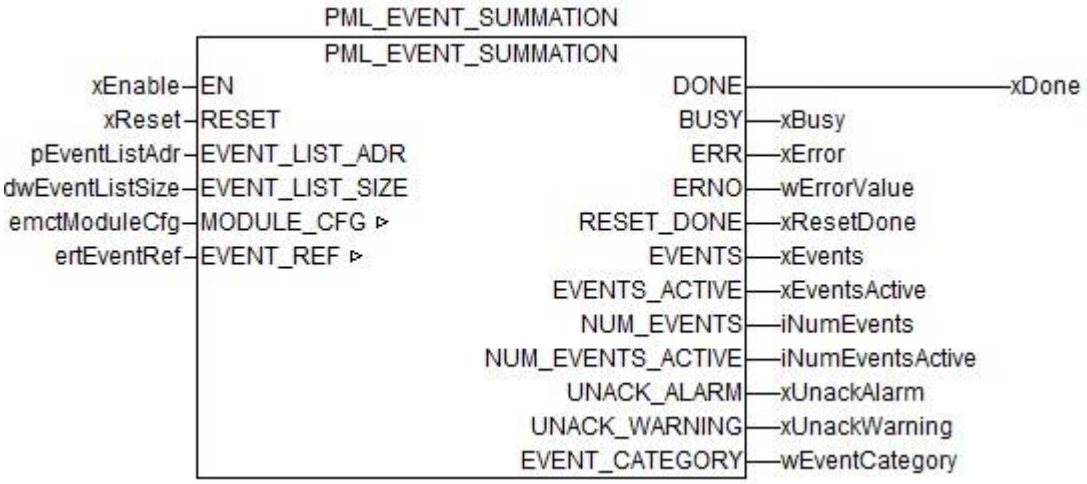
4.2.1.3 PML_EVENT_SUMMATION

This function block collects all events from lower level control modules and summates them in one common list.

Additionally, it offers the user information about the events of the hierarchal layer.



4.2.1.3.1 Input Description



The inputs marked with a triangle ▷ are of the class VAR_IN_OUT (input and output variable). These inputs must be connected to a variable.

EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

RESET

Data type: BOOL

This input is set to TRUE when all events are to be reset from the master event list. Also connected event base function blocks will reset the trigger of their configured events.

EVENT_LIST_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This input can be connected to the address of the event list where all events from lower modules are summed.

EVENT_LIST_SIZE

Data type: DWORD

Size of event list that is connected at the corresponding ..ADR input.

MODULE_CFG

Data type: PML_EVENT_MODULE_CFG_TYPE

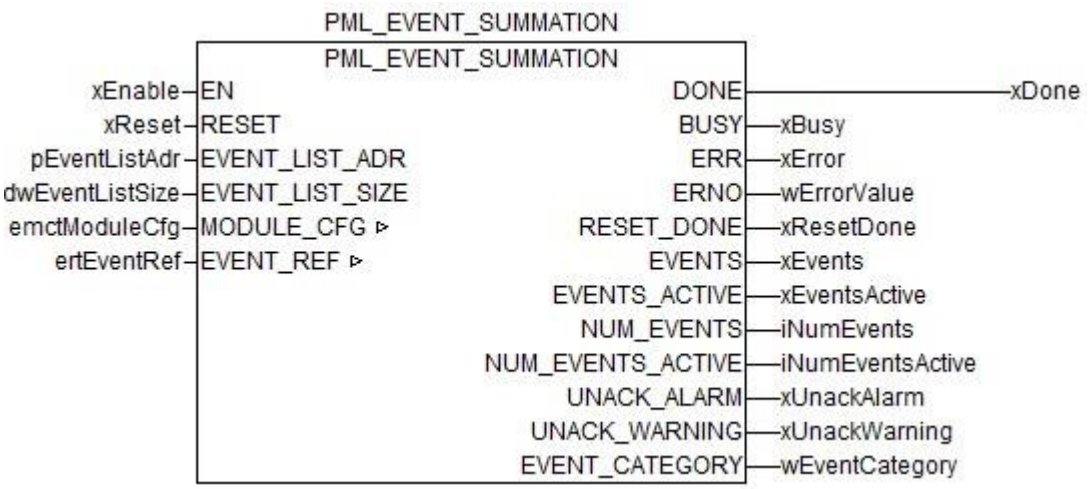
This input contains information about the event summation (module). It is important for the event tracking (Path) to know in which module an event was triggered.

EVENT_REF

Data type: PML_EVENT_REF_TYPE

This input is from the datatype Event Ref, the mailbox of a module that takes care of event exchange between two or more modules. The summation function block receives events from lower modules.

4.2.1.3.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

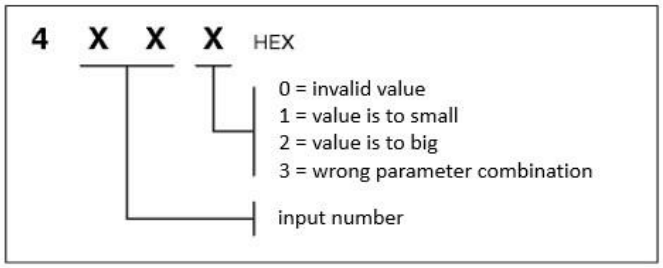
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



The function block indicates a full Event_List if ERR is TRUE and ERNO has the value 9999.

RESET_DONE

Data type: BOOL

This output is set to TRUE when the reset routine is finished.

EVENTS

Data type: BOOL

This output is set to TRUE when there is an Event in the master list, active or not.

EVENTS_ACTIVE

Data type: BOOL

This output is set to TRUE when there is an active Event in the master list.

NUM_EVENTS

Data type: INT

This output gives out the total number of events in the master list.

NUM_EVENTS_ACTIVE

Data type: INT

This output gives out the total number of active events in the master list.

UNACK_ALARM

Data type: BOOL

This output is set to TRUE if there is at least one unacknowledged alarm.

UNACK_WARNING

Data type: BOOL

This output is set to TRUE if there is at least one unacknowledged warning.

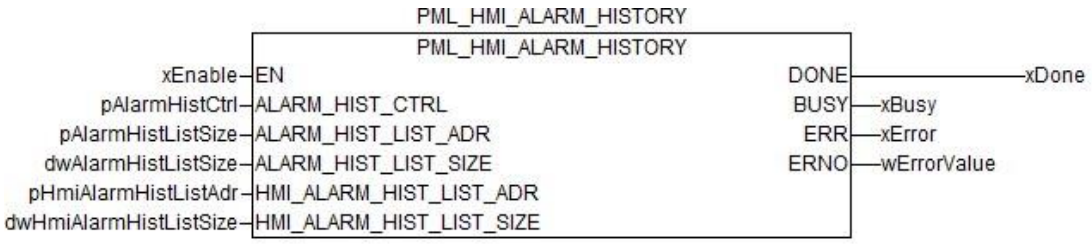
EVENT_CATEGORY

Data type: WORD

Bit 0 to 14 are set to 1 if there is one event with a corresponding category, bit 15 indicates that there is on event of category 15 or higher.

4.2.1.4 PML_HMI_ALARM_HISTORY

The alarm history function block displays the alarm history list on the HMI. (This function block is a modified version of the PML_HMI_EVENT).



4.2.1.4.1 Input Description

EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

ALARM_HIST_CTRL

Data type: POINTER TO PML_HMI_EVENT_CTRL_TYPE

This input can be connected to the instance address of the control buttons.

ALARM_HIST_LIST_ADR

Data type: POINTER TO PACKML_EVENT_TYPE

This input can be connected to the address of the alarm history list where the PackTag certified part of the event information is stored.

ALARM_HIST_LIST_SIZE

Data type: DWORD

Size of alarm history list that is connected at the corresponding ..ADR input.

HMI_ALARM_HIST_LIST_ADR

Data type: POINTER TO PACKML_EVENT_TYPE

This input can be connected to the address of the hmi alarm history list which is shown on the HMI.

HMI_ALARM_HIST_LIST_SIZE

Data type: DWORD

Size of hmi alarm history list that is connected at the corresponding ..ADR input.

4.2.1.4.2 Output Description

DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

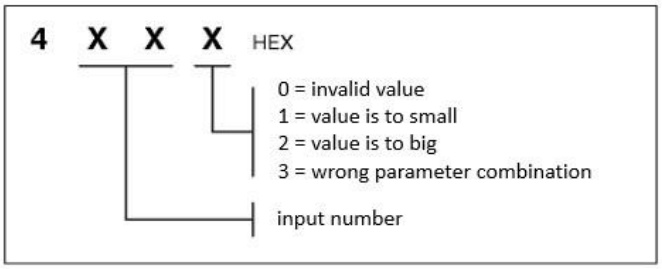
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

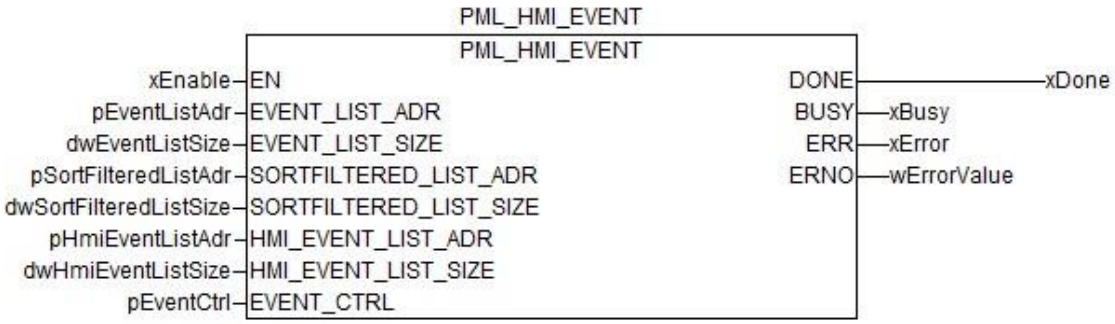
Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



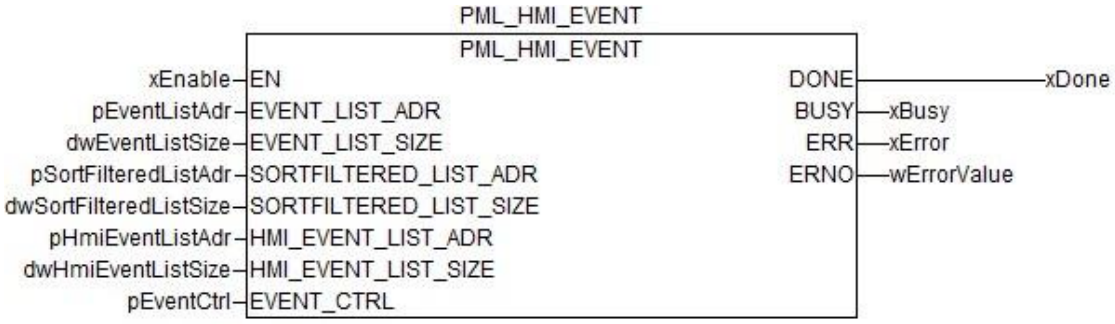
4.2.1.5 PML_HMI_EVENT

The event HMI function block takes the connected event list and copies it to a shorted one to be displayed on the HMI.

It also can be used for sorting and filtering of the shown list.



4.2.1.5.1 Input Description



EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

EVENT_LIST_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This input can be connected to the address of the master event list.

EVENT_LIST_SIZE

Data type: DWORD

Size of master list that is connected at the corresponding ..ADR input.

SORTFILTERED_LIST_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This Input can be connected to the address of the list where sorted and filtered events are to be added.

SORTFILTERED_LIST_SIZE

Data type: DWORD

Size of sorted and filtered master list that is connected at the corresponding ..ADR input.

HMI_EVENT_LIST_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This input can be connected to the address of the hmi event list which is shown on the HMI.

HMI_EVENT_LIST_SIZE

Data type: DWORD

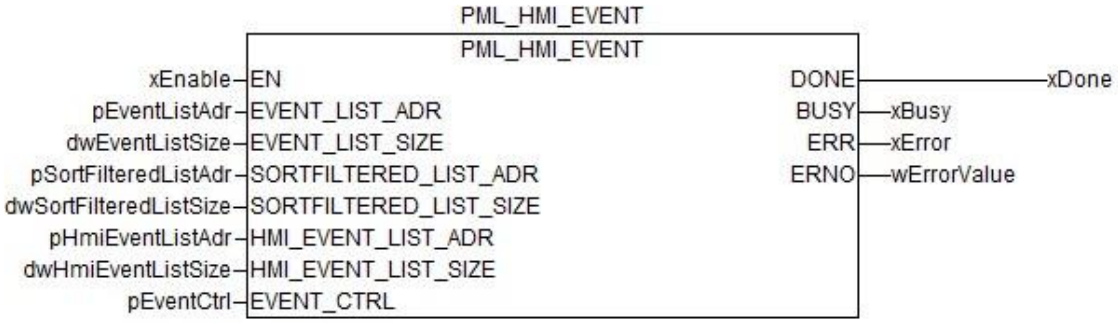
Size of hmi event list that is connected at the corresponding ..ADR input.

EVENT_CTRL

Data type: POINTER TO PML_HMI_EVENT_CTRL_TYPE

This input can be connected to the instance address of the control buttons.

4.2.1.5.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

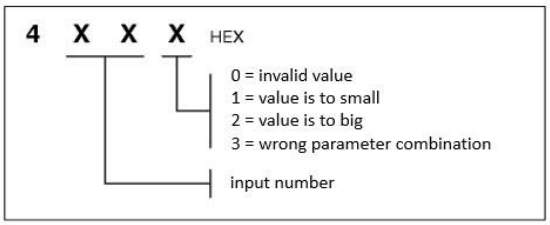
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

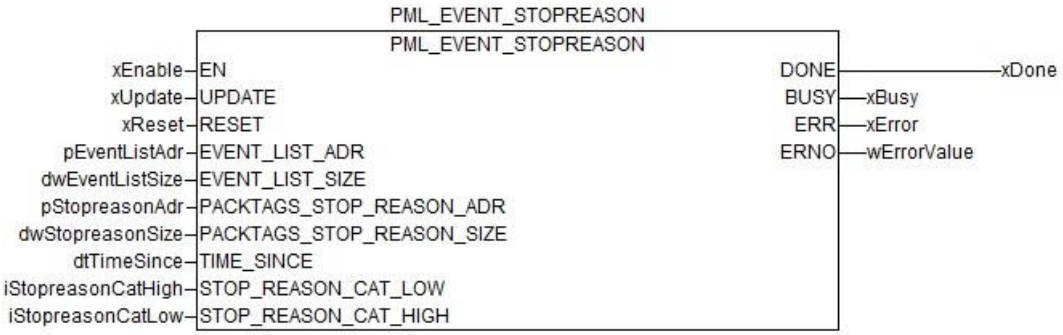
Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.

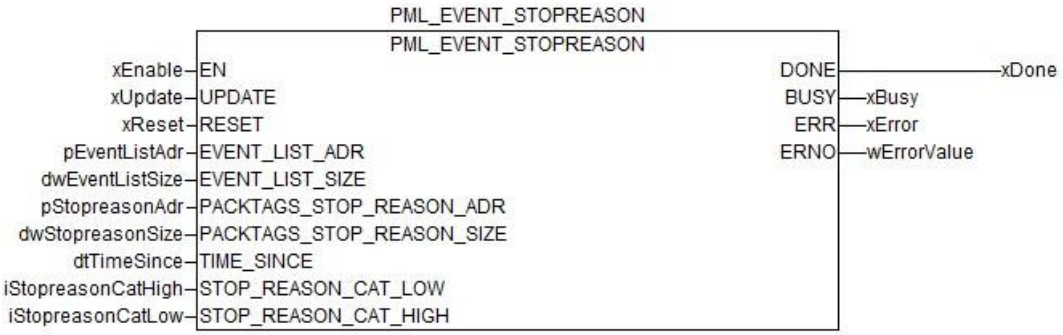


4.2.1.6 PML_EVENT_STOPREASON

The stop reason function block copies certain events from the connected event list to another stop reason event list depending on its inputs.



4.2.1.6.1 Input Description



EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

UPDATE

Data type: BOOL

This input is set to TRUE when certain events are to be copied from the master list to the stop reason list. The event selection depends on the time input and the category inputs.

RESET

Data type: BOOL

This input is set to TRUE when the stop reason list is to be cleared.

EVENT_LIST_ADR

Data type: POINTER TO PML_EVENT_CFG_TYPE

This input can be connected to the address of the master event list.

EVENT_LIST_SIZE

Data type: DWORD

Size of master list that is connected at the corresponding ..ADR input.

PACKTAGS_STOP_REASON_ADR

Data type: POINTER TO PML_EVENT_TYPE

This Input can be connected to the address of the list where the stop reason events are to be added.

PACKTAGS_STOP_REASON_SIZE

Data type: DWORD

Size of stop reason list that is connected at the corresponding ..ADR input.

TIME_SINCE

Data type: DATE_AND_TIME

Events that have a higher alarm date time are copied to the stop reason list.

STOP_REASON_CAT_LOW

Data type: INT

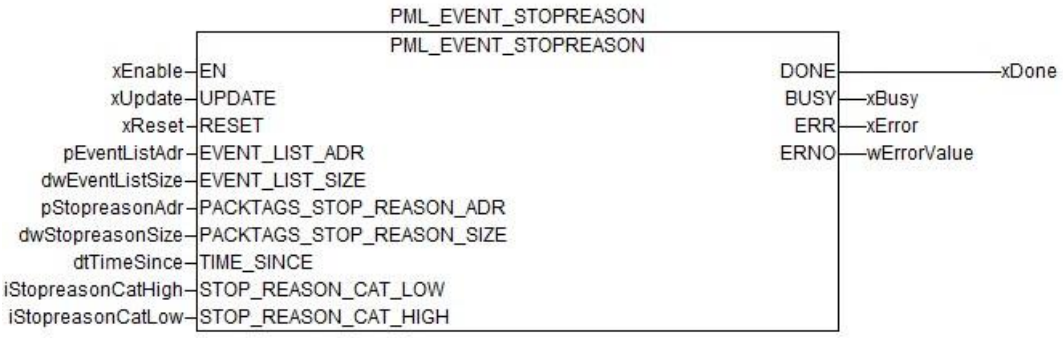
Events which category is higher or equal as this input are copied to the stop reason list.

STOP_REASON_CAT_HIGH

Data type: INT

Events which category is lower or equal as this input are copied to the stop reason list.

4.2.1.6.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

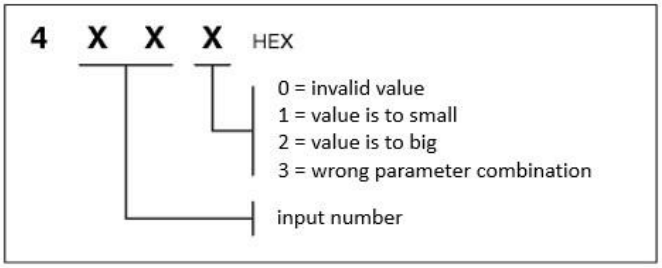
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



4.3 EVENT Datatypes

- PML_EVENT_CFG_TYPE (STRUCT)
- PML_EVENT_MODULE_CFG_TYPE (STRUCT)
- PML_EVENT_REF_TYPE (STRUCT)
- PML_EVENT_TYPE (STRUCT)
- PML_HMI_EVENT_CTRL_TYPE (STRUCT)

4.3.1 PML_EVENT_CFG_TYPE

The Event configuration data type describes the structure of an event. It is divided into two parts, the part that is PackML specified (“Event”, part of datatype PACKML_EVENT_TYPE) and the added, user definable event information.

```
STRUCT
    Event               : PML_EVENT_TYPE;
    EventType           : BOOL;
    PathArray           : ARRAY [0..9] OF INT;
    PathString          : STRING(PML_PATHSTRING_LEN_CONST);
    NumOccurred         : INT := 0;
    AlmDateTime         : DATE_AND_TIME;
    GoneDateTime        : DATE_AND_TIME;
    GoneDateTime_ms     : BYTE;
    AckDateTime         : DATE_AND_TIME;
    Acknowledged        : BOOL;
    Subroutine          : STRING(PML_SUBROUTINE_LEN_CONST);


END_STRUCT
```

In case of flexibility and to prevent of unused memory space the string length of Sub-routine and PathString are free definable. Make sure they are initialized in the global variables of your project.

Event

Data type: PACKML_EVENT_TYPE

This structure stores the PackML defined event information.

EventType

Data type: BOOL

This variable stores whether the event is a Warning or an Alarm (False = Warning; True = Alarm).

PathArray

Data type: ARRAY [0..9] OF INT

This array stores the spot the event is saved in the list.

PathString

Data type: STRING (n)

This variable stores the path the event has taken to get to the master list.

Due to limited memory space, the string length can be set in the global variables. (PML_PATHSTRING_LEN_CONST)

NumOccurred

Data type: INT

This variable stores the number of times the Trigger of the event has been set to TRUE.

AlmDateTime

Data type: DATE_AND_TIME

This variable stores the time the event Trigger was set to TRUE.

GoneDateTime

Data type: DATE_AND_TIME

This variable stores the time the event Trigger was set to FALSE.

GoneDateTime_ms

Data type: BYTE

This variable stores the time in milliseconds the event Trigger was set to FALSE.

AckDateTime

Data type: DATE_AND_TIME

This variable stores the time the event was acknowledged.

Acknowledged

Data type: BOOL

Bit shows if the event was acknowledged by using an HMI.

Subroutine

Data type: STRING (n)

Can Contain additional information about the source of an event

Due to limited memory space, the string length can be set in the global variables. (PML_SUBROUTINE_LEN_CONST)

4.3.2 PML_EVENT_MODULE_CFG_TYPE

This datatype is for identifying which modules are included in the project. It is used to identify the address and path array of the events.

```
STRUCT
    ID          : INT;
    NAME        : STRING(80);
```

ID

Data type: INT

This variable stores the ID of the module.

NAME

Data type: STRING (80)

This variable stores the name of the module.

4.3.3 PML_EVENT_REF_TYPE

```
(* Send Event *)
    ReadFlag            : BOOL;
    EventFbCnt          : INT;
    EventTotalCnt       : INT;
    Event               : PML_EVENT_CFG_TYPE;
```

ReadFlag

Data type: BOOL

This variable stores whether the mailbox has read the event that is entered.

EventFbCnt

Data type: INT

This variable stores the number of connected function blocks of type PML_EVENT_BASE

EventTotalCnt

Data type: INT

This variable stores the number of all events from all connected functions blocks of type PML_EVENT_BASE

Event

Data type: PML_EVENT_CFG_TYPE

This variable stores the event in the mailbox.

4.3.4 PML_EVENT_TYPE

```
STRUCT
    Trigger      : BOOL;
    ID           : DINT;
    Value        : DINT;
    Message      : STRING(PML_MESSAGE_LEN_CONST);
    Category     : DINT;
    AlmDateTime  : ARRAY [0..6] OF DINT;
    AckDateTime  : ARRAY [0..6] OF DINT;
END_STRUCT
```

Trigger

Data type: BOOL

Indicates if Event (Alarm/Warning) is True or False

ID

Data type: DINT

Shows the configured Event (Alarm/Warning) Id

Value

Data type: DINT

Shows the configured Event (Alarm/Warning) Value

Message

Data type: STRING (n)

Shows the configured Event (Alarm/Warning) Message

Due to limited memory space, the string length can be set in the global variables. (PML_MESSAGE_LEN_CONST)

Category

Data type: DINT

Shows the configured Event (Alarm/Warning) Category

AlmDateTime

Data type: ARRAY [0..6] OF DINT

Shows the time, referring to PackML specification, the Event (Alarm/Warning) was triggered.

ARRAY [0] – Year
ARRAY [1] – Month
ARRAY [2] – Day
ARRAY [3] – Hour
ARRAY [4] – Minute
ARRAY [5] – Second
ARRAY [6] - uSecond

AckDateTime

Data type: ARRAY [0..6] OF DINT

Shows the time, referring to PackML specification, the Event (Alarm/Warning) was acknowledged.

ARRAY [0] – Year
ARRAY [1] – Month
ARRAY [2] – Day
ARRAY [3] – Hour
ARRAY [4] – Minute
ARRAY [5] – Second
ARRAY [6] - uSecond

4.3.5 PML_HMI_EVENT_CTRL_TYPE

```
STRUCT
    (* HMI controls *)
    BTN_UP1                     : BOOL;
    BTN_UP2                     : BOOL;
    BTN_UP3                     : BOOL;
    BTN_DOWN1                   : BOOL;
    BTN_DOWN2                   : BOOL;
    BTN_DOWN3                   : BOOL;
    BTN_REFRESH                 : BOOL;
    BTN_ACK_ALL                 : BOOL;
    FILTER_STATUS               : BOOL;
    FILTER_STATUS_CFG           : BOOL;
    FILTER_EVENT_TYPE           : BOOL;
    FILTER_EVENT_TYPE_CFG       : BOOL;
    FILTER_VALUE                : BOOL;
    FILTER_VALUE_CFG_LOW        : INT;
    FILTER_VALUE_CFG_HIGH       : INT;
    FILTER_CATEGORY             : BOOL;
    FILTER_CATEGORY_CFG_LOW     : INT;
    FILTER_CATEGORY_CFG_HIGH    : INT;
    FILTER_TIME_COME            : BOOL;
    FILTER_TIME_COME_CFG_LOW    : DATE_AND_TIME;
    FILTER_TIME_COME_CFG_HIGH   : DATE_AND_TIME;
    FILTER_ACKNOWLEDGE          : BOOL;
    FILTER_ACKNOWLEDGE_CFG      : BOOL;
    SORT_CAT                    : BOOL;
    SORT_CAT_CFG                : BOOL;
    SORT_STATUS                 : BOOL;
    SORT_STATUS_CFG             : BOOL;
    SORT_EVENT_TYPE             : BOOL;
    SORT_EVENT_TYPE_CFG         : BOOL;
    SORT_TIME_COME              : BOOL;
    SORT_TIME_COME_CFG          : BOOL;
    SORT_ACKNOWLEDGE            : BOOL;
    SORT_ACKNOWLEDGE_CFG        : BOOL;
END_STRUCT
```

Btn_Up1

Data type: BOOL

This variable moves the connected list to the top when TRUE.

Btn_Up2

Data type: BOOL

This variable moves the connected list one page size up when TRUE.

Btn_Up3

Data type: BOOL

This variable moves the connected list one event up when TRUE.

Btn_Down1

Data type: BOOL

This variable moves the connected list to the bottom when TRUE.

Btn_Down2

Data type: BOOL

This variable moves the connected list one page size down when TRUE.

Btn_Down3

Data type: BOOL

This variable moves the connected list one event down when TRUE.

Btn_Refresh

Data type: BOOL

This variable refreshes the connected list.

Btn_Ack_All

Data type: BOOL

This variable acknowledges all unacknowledged events when TRUE.

Filter_Status

Data type: BOOL

This input is set to TRUE when the function block should filter with the FILTER_STATUS_CFG condition

Filter_Status_Cfg

Data type: BOOL

True = active, False = not active

Filter_Event_Type

Data type: BOOL

This input is set to TRUE when the function block should filter with the FILTER_EVENT_TYPE_CFG condition.

Filter_Event_Type_Cfg

Data type: BOOL

True = Alarm, False = Warning

Filter_Value

Data type: BOOL

The input is set to TRUE when the function block should filter with the FILTER_VALUE_CFG_LOW and FILTER_VALUE_CFG_HIGH conditions.

Filter_Value_Cfg_Low

Data type: INT

Lower limit of Value filtering range.

Filter_Value_Cfg_High

Data type: INT

Upper limit of Value filtering range.

Filter_Category

Data type: BOOL

This input is set to TRUE when the function block should filter with the FILTER_CATEGORY_CFG_LOW and FILTER_CATEGORY_CFG_HIGH conditions.

Filter_Category_Cfg_Low

Data type: INT

Lower limit of Category filtering range.

Filter_Category_Cfg_High

Data type: INT

Upper limit of Category filtering range.

Filter_Time_Come

Data type: BOOL

This input is set to TRUE when the function block should filter with the FILTER_TIME_COME_CFG_LOW and FILTER_TIME_COME_CFG_HIGH conditions.

Filter_Time_Come_Cfg_Low

Data type: DATE_AND_TIME

Lower limit of time come filtering range.

Filter_Time_Come_Cfg_High

Data type: DATE_AND_TIME

Upper limit of time come filtering range.

Sort_Cat

Data type: BOOL

This input is set to TRUE when the function block should sort with the SORT_CAT_CFG condition.

Sort_Cat_Cfg

Data type: BOOL

TRUE = highest event first, FALSE = lowest event first

Sort_Status

Data type: BOOL

This input is set to TRUE when the function block should sort with the SORT_STATUS_CFG condition.

Sort_Status_Cfg

Data type: BOOL

TRUE = active first, FALSE = inactive first

Sort_Event_Type

Data type: BOOL

This input is set to TRUE when the function block should sort with the SORT_EVENT_TYPE_CFG condition.

Sort_Event_Type_Cfg

Data type: BOOL

TRUE = alarm first, FALSE = warning first

Sort_Time_Come

Data type: BOOL

This input is set to TRUE when the function block should sort with the SORT_TIME_COME_CFG condition.

Sort_Time_Come_Cfg

Data type: BOOL

TRUE = newest events first, FALSE = oldest events first

Sort_Acknowledge

Data type: BOOL

This input is set to TRUE when the function block should sort with the SORT_ACKNOWLEDGE_CFG condition.

Sort_Acknowledge_Cfg

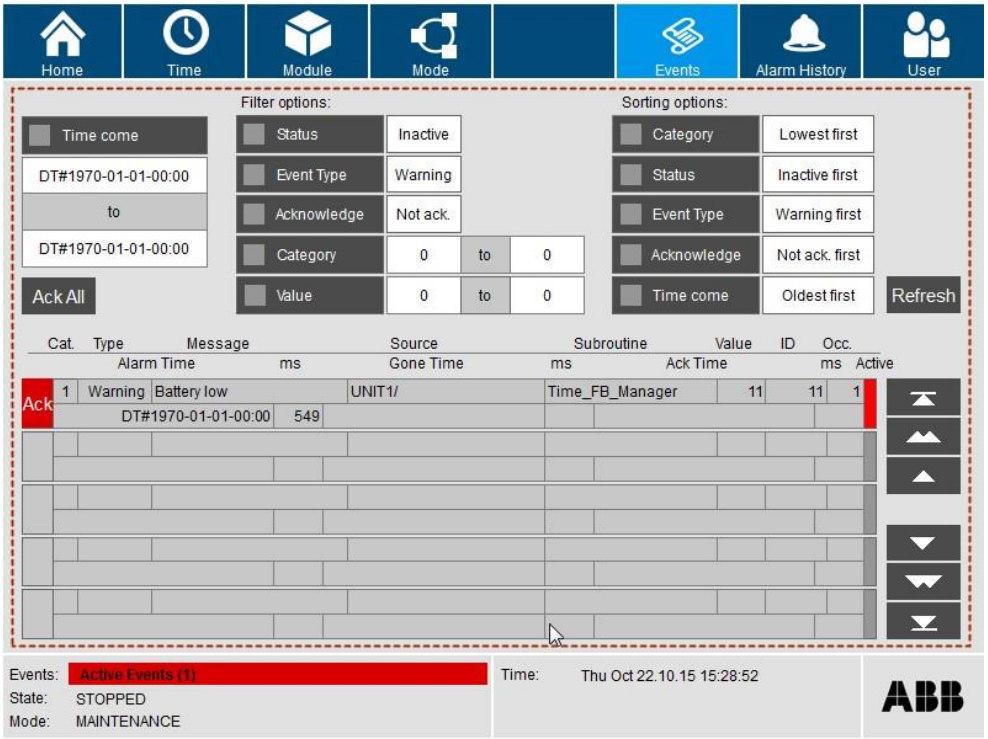
Data type: BOOL

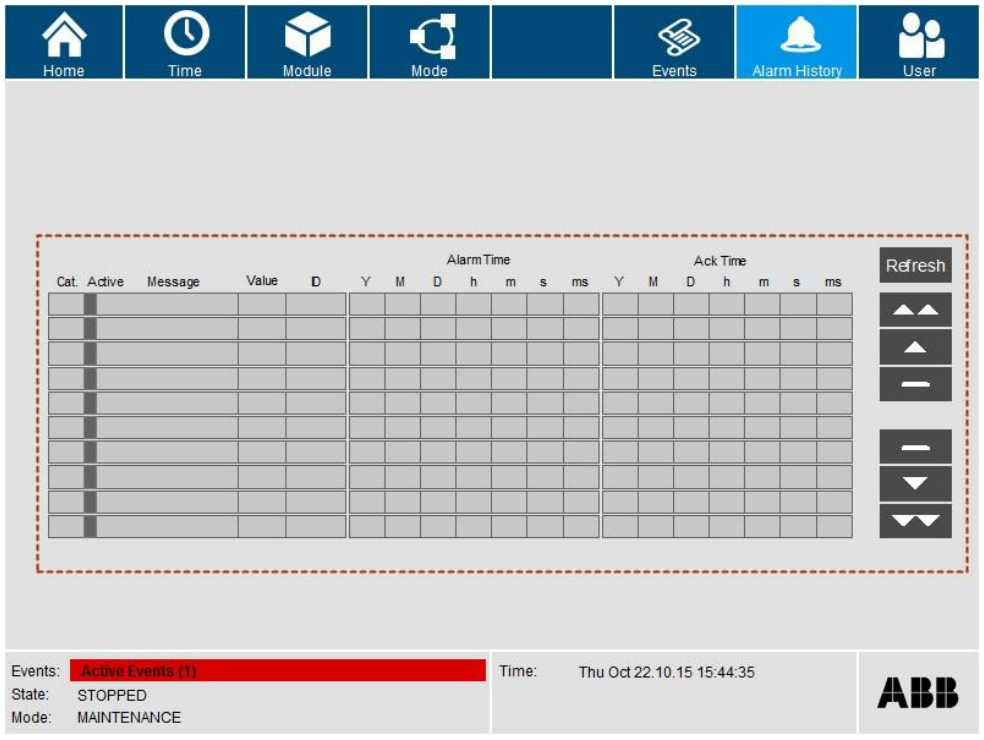
TRUE = acknowledged first, FALSE = not acknowledged first

4.4 Visualization

4.4.1 Webserver Template

means it is a predefined visualization template in CoDeSys and part of the Library



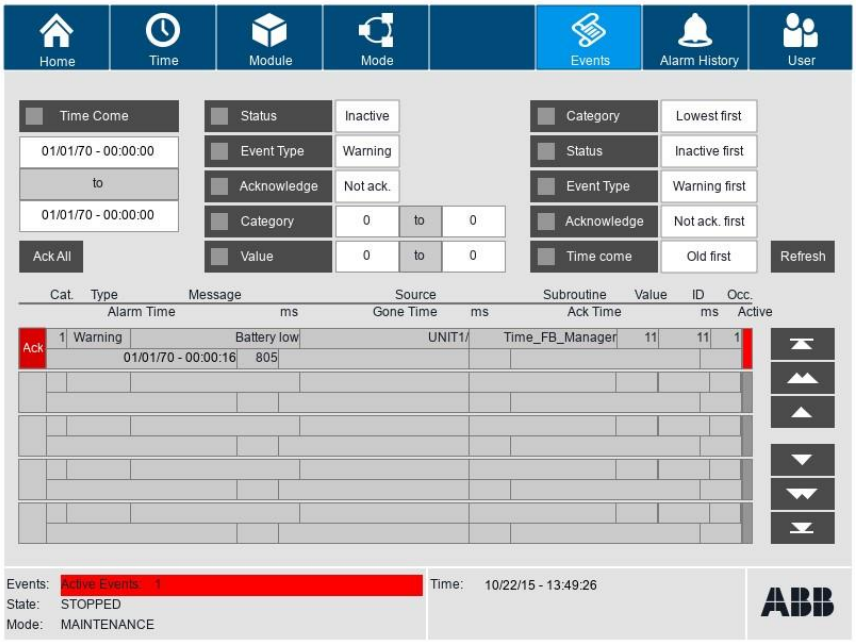


The CoDeSys webserver visualizations are implemented using place holders. Instead of reconfiguring each element on the used HMI page connect the variable instance to the referring visualization place holder.

4.4.2 CP600 Template

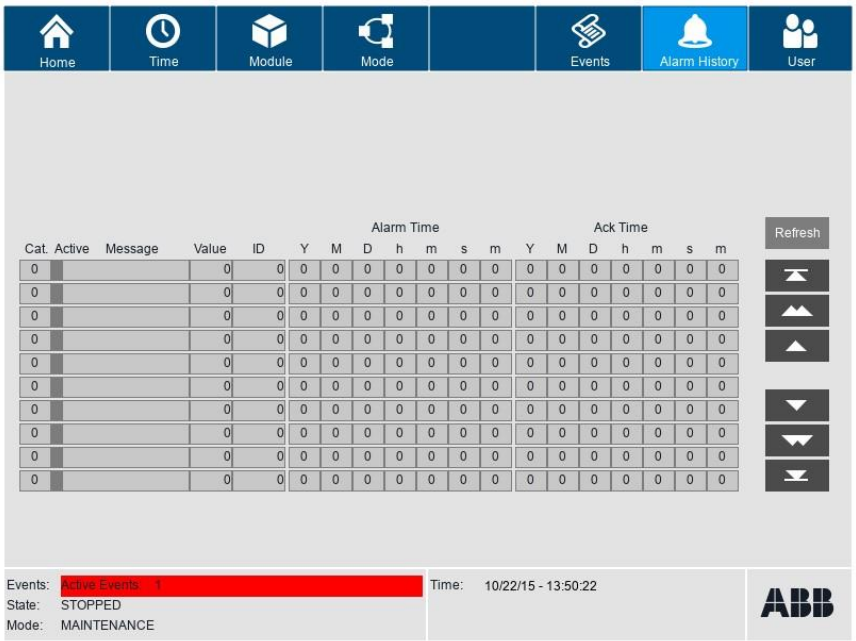
Events:

The event list elements on the template page in the CP600 has to be connected to the HMI_EVENT_LIST_ADR variables.



On the top of the events page the user can select filtering and sorting options. On the bottom a limited number of the chosen events are displayed with the option to scroll through the entire list.

Alarm History:



This page is used to display and scroll through the alarm history list.

4.4.3 Change display size

1. Change the array size and adjust all size inputs on the FBs in CoDeSys.
2. Reimport Tags (Symbol file) from CoDeSys to the CP600.
3. Copy one event structure from the list and paste it onto the bottom.
4. Connect the fields to the correct variables.

5 Operating Time of PackML Library

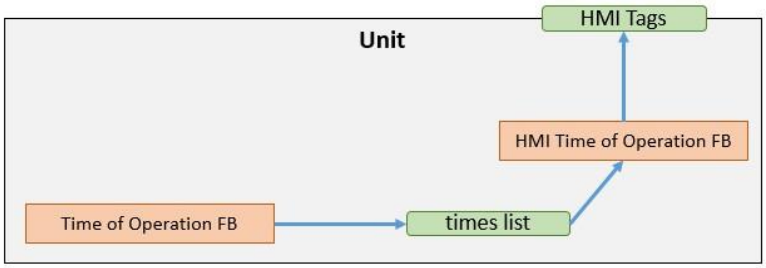
5.1 Covered PackTags

			Tag Name	Datatype
Admin				
ModeCurrent-Time[#]			UnitName.Admin.ModeCurrent-Time[#]	Int (32bit)
ModeCumulative-Time[#]			UnitName.Admin.Mode-CumulativeTime[#]	Int (32bit)
StateCurrent-Time[#,#]	(Mode, State)		UnitName.Admin.StateCurrent-Time[#,#] (Mode, State)	Int (32bit)
StateCumulative-Time[#,#]	(Mode, State)		UnitName.Admin.State-CumulativeTime[#,#] (Mode, State)	Int (32bit)
AccTimeSinceReset	AccTime-SinceReset		UnitName.Admin.AccTimeSince-Reset	Int(32bit)
PLCDateTime			UnitName.Admin.PLCDateTime	Date-Time Array
		[0] (year)	UnitName.Admin.PLCDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.PLCDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.PLCDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.PLCDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.PLCDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.PLCDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.PLCDateTime[6]	Int (32bit)

5.2 Usage of Time of Operation Function Blocks

These function blocks are essential for the PackTags. They are used to calculate and display current and cumulative times for all states in each mode.

5.2.1 Template Example

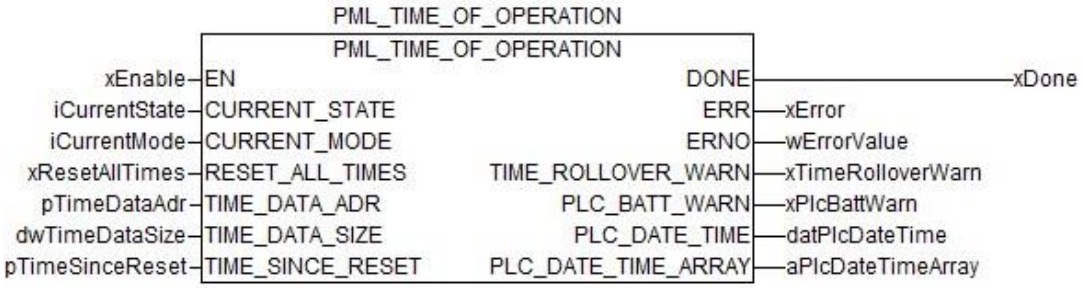


In the template Project the function blocks are implemented on the Unit layer. It is the hierarchal level which represents the mode state model outwards the machine, so it is the only place to detect the operating times for each mode and each state.

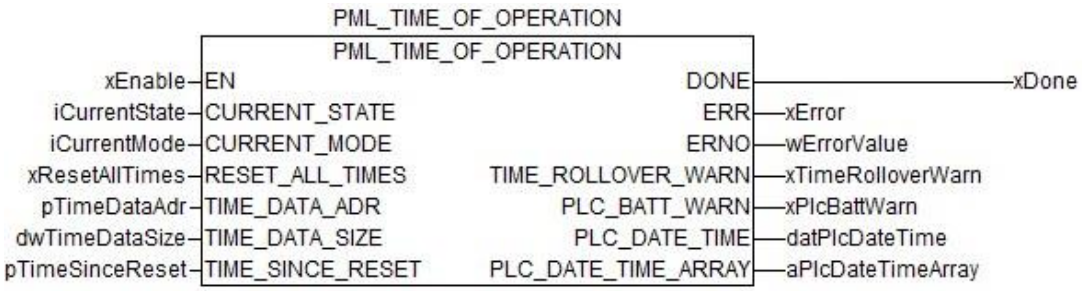
5.3 Time of Operation Function Blocks

5.3.1 PML_TIME_OF_OPERATION

The Time of Operation function block is used to calculate current and cumulative times for all modes and all states, following the PackML standard.



5.3.1.1 Input Description



EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

CURRENT_ STATE

Data type: INT

The input can be connected to the number associated with the currently active state. See attachments for PackML specified state numbers.

CURRENT_MODE

Data type: INT

The input can be connected to the number associated with the currently active mode. See attachments for PackML specified mode numbers.

RESET_ALL_TIMES

Data type: BOOL

The input is set to TRUE when all times of all modes and states are to do a reset.

TIME_DATA_ADR

Data type: POINTER TO PML_MODESTATE_TIME_TYPE

This input can be connected to the address of the full time array for all modes and states. Make sure array is persistent in order to keep data during power off times.

TIME_DATA_SIZE

Data type: DWORD

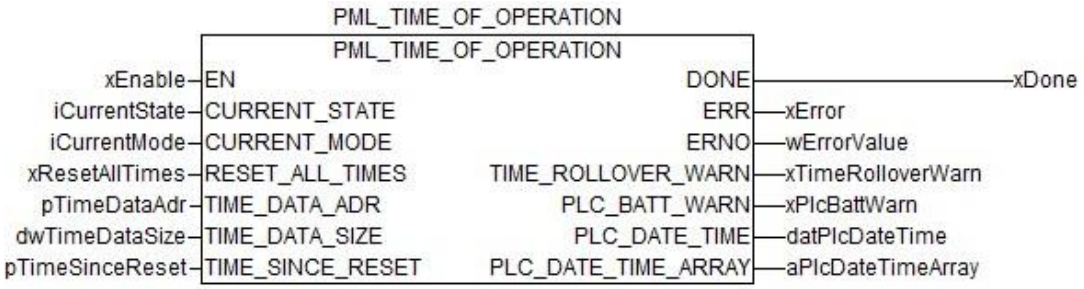
Size of modes and states time array that is connected at the corresponding ..ADR input.

TIME_SINCE_RESET

Data type: POINTER TO DINT

This input can be connected to the address of machine total time variable. Make sure array is persistent in order to keep data during power off times.

5.3.1.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERR (error)

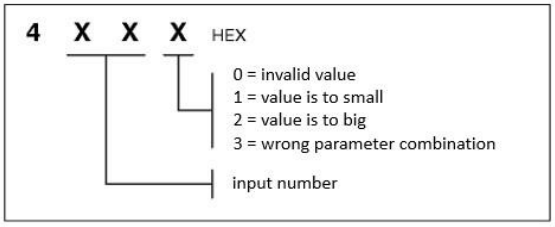
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



TIME_ROLLOVER_WARN

Data type: BOOL

This Output is set to TRUE when the time goes beyond the permissible range (2144891648 seconds).

PLC_BATT_WARN

Data type: BOOL

This output gives is set to TRUE when there is a PLC Battery Warning.

PLC_DATE_TIME

Data type: DATA_AND_TIME

This output gives out the current PLC time.

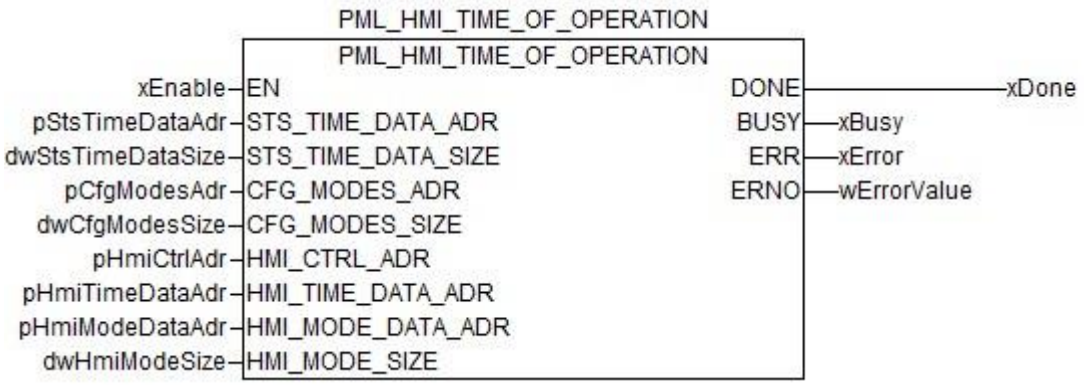
PLC_DATA_TIME_ARRAY

Data type: ARRAY [0…6] OF DINT

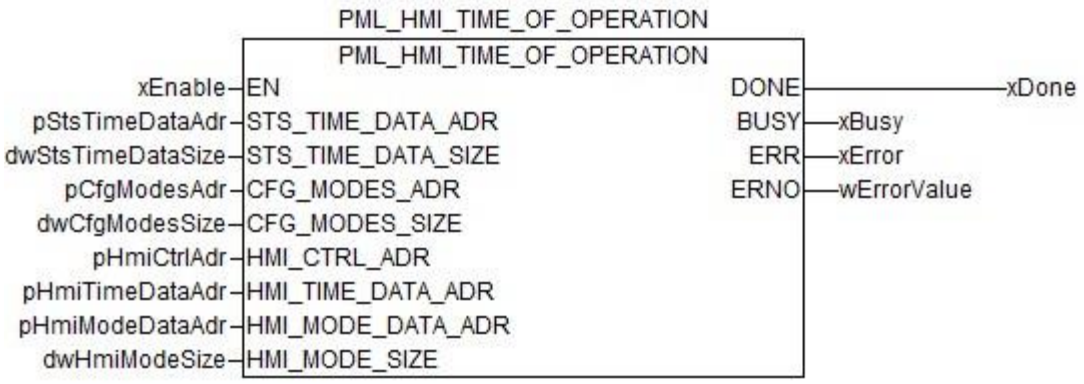
This output gives out the current PLC time, separated in arrays because of PackML specification.

5.3.2 PML_HMI_TIME_OF_OPERATION

The Time of Operation function block for the HMI is used to display the mode and state times on the HMI, including the corresponding controls. It also allows to reset all times via the HMI.



5.3.2.1 Input Description



EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

STS_TIME_DATA_ADR

Data type: POINTER TO PML_MODESTATE_TIME_TYPE

This input can be connected to the address of the modes and states full time array.

STS_TIME_DATA_SIZE

Data type: DWORD

Size of modes and states time array that is connected at the corresponding ..ADR input.

CFG_MODES_ADR

Data type: POINTER TO PML_MODE_CFG_TYPE

This input can be connected to the address of the mode configuration. It is needed, so only the times are displayed from the Mode that is active.

CFG_MODES_SIZE

Data type: DWORD

Size of mode configuration list that is connected at the corresponding ..ADR input.

HMI_CTRL_ADR

Data type: POINTER TO PML_HMI_MSTIME_CTRL_TYPE

This input can be connected to the address of the instance for the control buttons of time management.

HMI_TIME_DATA_ADR

Data type: POINTER TO PML_MODESTATE_TIME_TYPE

This input can be connected to the address of the list where the configured time array from the connected list are copied to and then displayed on the HMI.

HMI_MODE_DATA_ADR

Data type: POINTER TO PML_MODE_CFG_TYPE

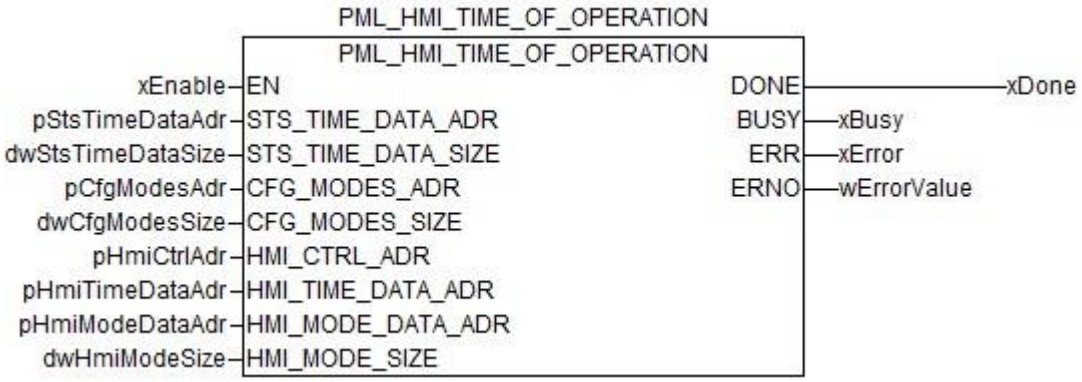
This input can be connected to the address of the hmi list which shows the mode configuration on the HMI.

HMI_MODE_SIZE

Data type: DWORD

Size of hmi mode configuration list that is connected at the corresponding ..ADR input.

5.3.2.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

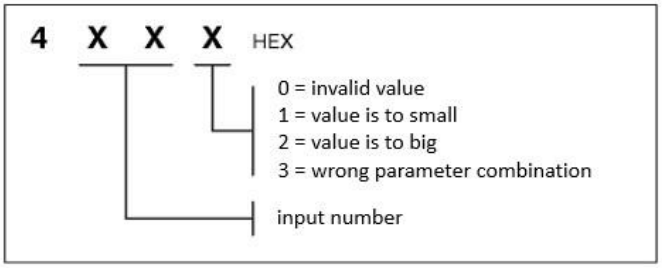
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



5.4 Time of Operation Datatypes

- PML_HMI_MSTIME_CTRL_TYPE
- PML_MODESTATE_TIME_TYPE
- PML_STATE_TIME_TYPE

5.4.1 PML_HMI_MSTIME_CTRL_TYPE

```
STRUCT
    (* HMI controls *)
    Btn_Update              : BOOL;
    Btn_Up1                 : BOOL;
    Btn_Up2                 : BOOL;
    Btn_Down1               : BOOL;
    Btn  Down2              : BOOL;
    TimeSinceReset          : DINT;
    TimeRolloverWarning     : BOOL;
    Btn_Reset_All_Times     : BOOL;
END_STRUCT
```

Btn_Update

Data type: BOOL

This variable updates the connected time array.

Btn_Up1

Data type: BOOL

This variable moves the connected list to the top when TRUE.

Btn_Up2

Data type: BOOL

This variable moves the connected list one page size up when TRUE.

Btn_Down1

Data type: BOOL

This variable moves the connected list to the bottom when TRUE.

Btn_Down2

Data type: BOOL

This variable moves the connected list one page size down when TRUE.

TimeSinceReset

Data type: DINT

This variable stores the time since the last reset.

TimeRolloverWarning

Data type: BOOL

This variable stores whether there is a rollover warning or not.

Btn_Reset_All_Times

Data type: BOOL

This variable resets all mode and state times

5.4.2 PML_MODESTATE_TIME_TYPE

```
STRUCT
    ModeCurrentTime       : DINT;
    ModeCumulativeTime    : DINT;
    StateCurrentTime      : PACKML_STATE_TIME_TYPE;
    StateCumulativeTime   : PACKML_STATE_TIME_TYPE;
```

ModeCurrentTime

Data type: DINT

This variable stores the total time elapsed at the current mode.

ModeCumulativeTime

Data type: DINT

This variable stores the total time elapsed by all modes.

StateCurrentTime

Data type: PACKML_STATE_TIME_TYPE

This structure stores the total time elapsed at the current state.

StateCumulativeTime

Data type: PACKML_STATE_TIME_TYPE

This structure stores the total time elapsed by all states.

5.4.3 PML_STATE_TIME_TYPE

```
STRUCT
    State   : ARRAY[0..17] OF DINT       := 18(0);
END_STRUCT
```

State

Data type: ARRAY [0..17] OF DINT

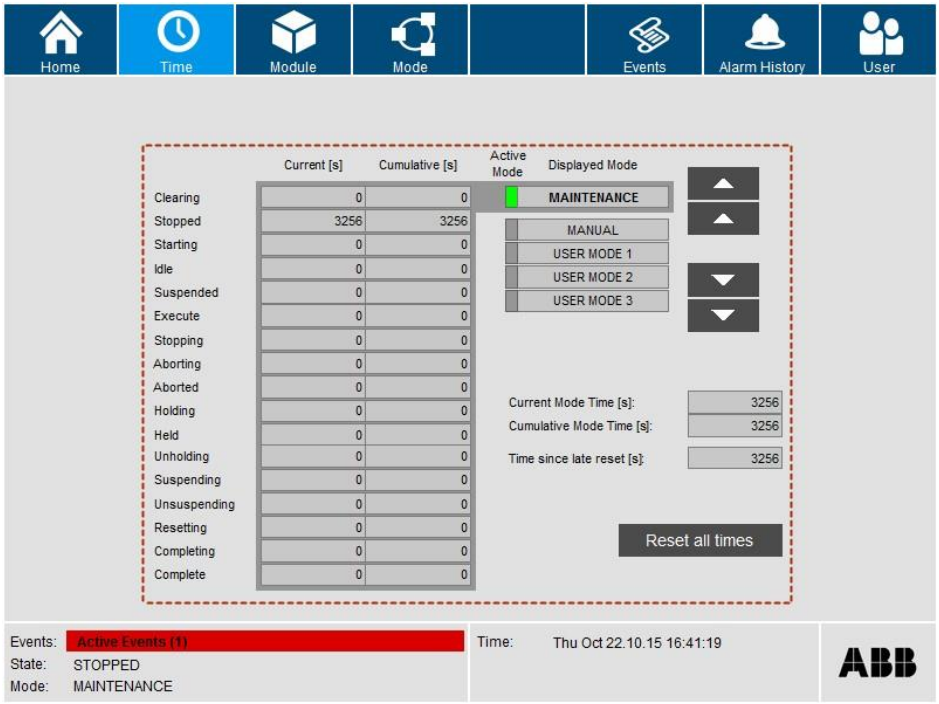
This array stores the time elapsed in each PackML state.

5.5 Visualization

The page for the times is used to display the current and cumulative times of each state of the selected mode (mode in the first line). The currently active mode will be marked. Additionally the user is able to reset all times and see the time elapsed since the last reset.

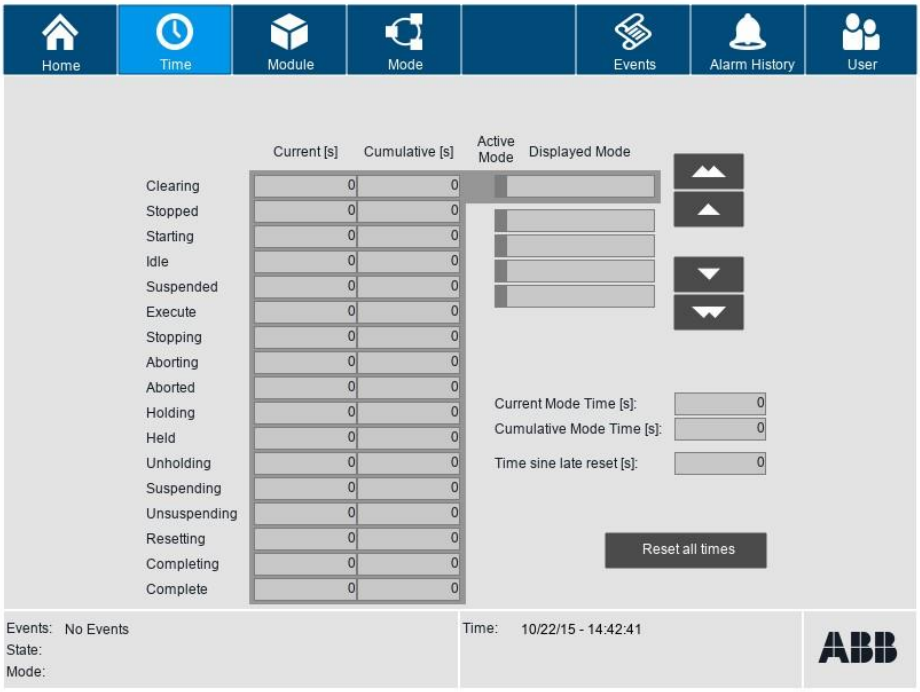
5.5.1 Webserver Template

means it is a predefined visualization template in CoDeSys and part of the Library



The CoDeSys webserver visualizations are implemented using place holders. Instead of reconfiguring each element on the used HMI page connect the variable instance to the referring visualization place holder.

5.5.2 CP600 Template



5.5.3 Change display size

1. Change the array size and adjust all size inputs on the FBs in CoDeSys.
2. Reimport Tags from CoDeSys to the CP600.
3. Copy one list structure from the list and paste it onto the bottom.
4. Connect the fields to the correct variables.

6 Modes & States of PackML Library

6.1 Covered PackTags

Command		
UnitMode	UnitName.Command.UnitMode	Int (32bit)
UnitModeChangeRequest	UnitName.Command.UnitModeChangeRequest	Bool
CntrlCmd	UnitName.Command.CntrlCmd	Int (32bit)
CmdChangeRequest	UnitName.Command.CmdChangeRequest	Bool
Status		
UnitModeCurrent	UnitName.Status.UnitModeCurrent	Int (32bit)
UnitModeRequested	UnitName.Status.UnitModeRequested	Bool
UnitModeChangeInProcess	UnitName.Status.UnitModeChangeInProcess	Bool
StateCurrent	UnitName.Status.StateCurrent	Int (32bit)
StateRequested	UnitName.Status.StateRequested	Int (32bit)
StateChangeInProcess	UnitName.Status.StateChangeInProcess	Bool
Admin		
StatesDisabled	UnitName.Admin.StatesDisabled	Int(32bit)

6.2 3.2 Usage of the mode and state machine Function Blocks

In case of flexibility and to receive the modular structure that that guide promises, the machine functionalities are divided into modes and states. A mode describes an entire procedure that occurs within for example a production line. The states in the mode are the different situation that can happen. The three predefined PackML modes (Production, Maintenance, Manual) can be used additional to the user defined modes (e.g. Clean-out). In general the mode configuration (which states are active and which states allow a mode change) are user definable.

On a unit level the machine has to at least appear as having one state machine, as it is required for PackML. This is for a common outwards machine representation when another unit or a line controller sends a command for a specific state, it needs to be able to react. Lower machine modules are not prohibited from adapting the mode-state-machine.

To start off, all Modes that can occur need to be defined, in the appropriate module code. This can include the 3 predefined Modes (Production, Maintenance and Manual) from OMAC and also the user defined ones if needed.

User_defined_name: Array of [#...#] of PML_MODE_CFG_TYPE :=

Example

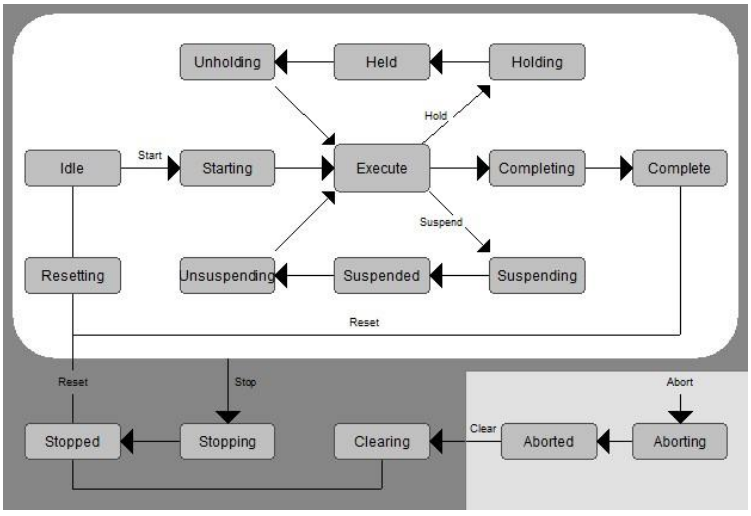
(Name: 'PRODUCTION',
ID: 1,
Disabled States: 2#00000000000000000110000000100000,
Mode Transitions: 2#00000000000000000000001000010100,
Active: FALSE,
Initial State: 2);

(Name: 'Maintenance',
ID: 1,
Disabled States: 2#00000000000000000110000000100000,
Mode Transitions: 2#00000000000000000000001000010100,
Active: TRUE,
Initial State: 2)

The 'Disabled States' and 'Mode Transitions' are two DWORDs. 'Disabled States' describe the states that are not available in the mode and 'Mode Transitions' describe the states where a mode change is allowed.

Caution: When creating the two 17-bit rows it is crucial to set the correct bits. It starts on the right. The first bit is always 0 because the first PackML defined state starts with number 1. The next 17 bits have to be set according to the attached state table.

The following picture shows the state model for one mode, which has all states active. Depending on the above mentioned bit settings each of these states can be disabled.



The mode-state-manager is used to manage these mode state models, including mode and state transitioning. When it gets a mode or state change command it goes through all conditions required to allow the mode or state change and then returns as an output whether the change was successful.

State changes can be initialized by either a user command via HMI or by the application code. Leaving an acting state (state ending with ..ing) happens via a State Complete command. Normally this bit is set by the application code.

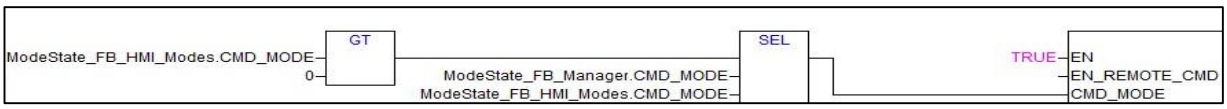
6.2.1 Template Example

In the template example ten modes are configured (seven user defined modes additional to the three PackML predefined ones: Production, Maintenance and Manual).

6.3 State machine Function Blocks

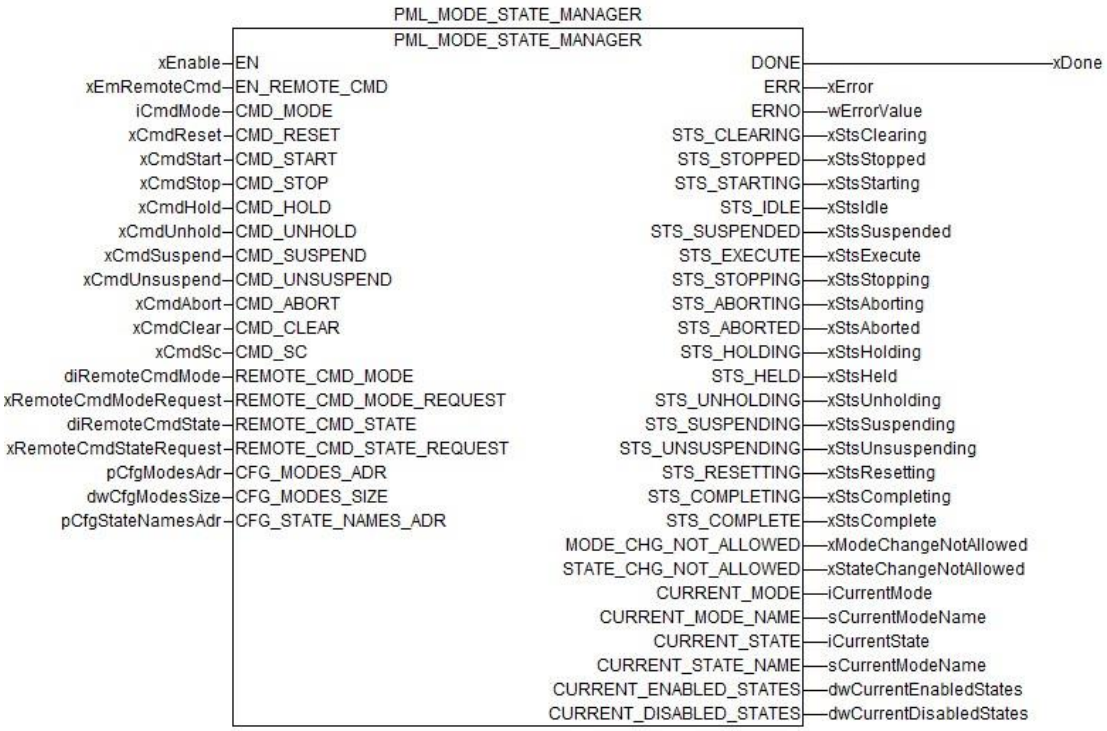
The two function blocks (PML_MODE_STATE_MANAGER and PML_HMI_MODE_SELECT) handle everything regarding modes and states in the Unit, letting the submodules know how to react and displaying the appropriate information on the HMI. On the one hand side the mode state function block has to process the commands coming from an HMI or the application code and on the other hand side the remote commands. In case of repair or maintenance on the hardware, the function block has to make sure that no commands from a line controller or another Unit, which could change the actual state, are executed.

When using the Mode-State-Machine in a module it is important to determine where the commands are coming from. The GT FB will be TRUE when the first input is bigger than 0. If on the first input the SEL FB receives a TRUE the 3rd input will become the output (commands are coming from the HMI/Web Server), otherwise the second input becomes the output (current mode). Therefore, if the HMI sends a mode change, the Mode-State-Manager will receive it, otherwise it will continue working with its current mode.

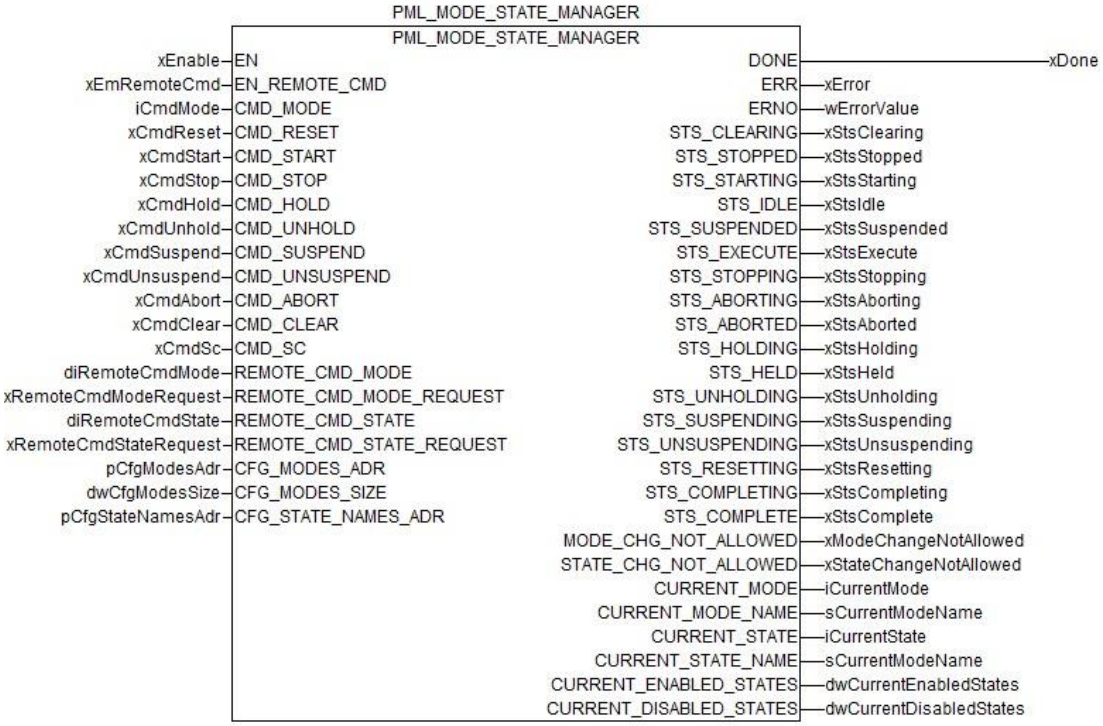


6.3.1 PML_MODE_STATE_MANAGER

The Mode State Manager function block is used to handle inputs from different sources, disable specific states, update mode information and control Mode and State transition.



6.3.1.1 Input Description



EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

EN_REMOTE_CMD

Data type: BOOL

This input is set to TRUE when a possible mode change command from a remote source is allowed.

CMD_MODE

Data type: INT

This input can be connected to the mode the machine shall switch into. The target mode has to be connected depending on its configured ID.

CMD_RESET

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Reset State.

CMD_START

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Start State.

CMD_STOP

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Stop State.

CMD_HOLD

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Hold State.

CMD_UNHOLD

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Unhold State.

CMD_SUSPEND

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Suspend State.

CMD_UNSUSPEND

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Unsuspend State.

CMD_ABORT

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Abort State.

CMD_CLEAR

Data type: BOOL

This input is set to TRUE when the machine shall switch into the Clear State.

CMD_SC

Data type: BOOL

This input is set to TRUE when the machine gets the State Complete command from the application code. There further conditions need to be fulfilled before.

REMOTE_CMD_MODE

Data type: DINT

This input can be connected to the mode that the machine shall switch into coming from a remote source.

REMOTE_CMD_MODE_REQUEST

Data type: BOOL

This input is set to TRUE when a remote source wants to carry out a mode change.

REMOTE_CMD_STATE

Data type: DINT

This input can be connected to the state the machine shall switch into coming from a remote source. See the attached list for the mapping of a state command on a specified number.

REMOTE_CMD_STATE_REQUEST

Data type: BOOL

This input is set to TRUE when a remote source wants to carry out a state change.

CFG_MODES_ADR

Data type: POINTER TO PML_MODE_CFG_TYPE

This input can be connected to the address of the mode configuration.

CFG_MODES_SIZE

Data type: DWORD

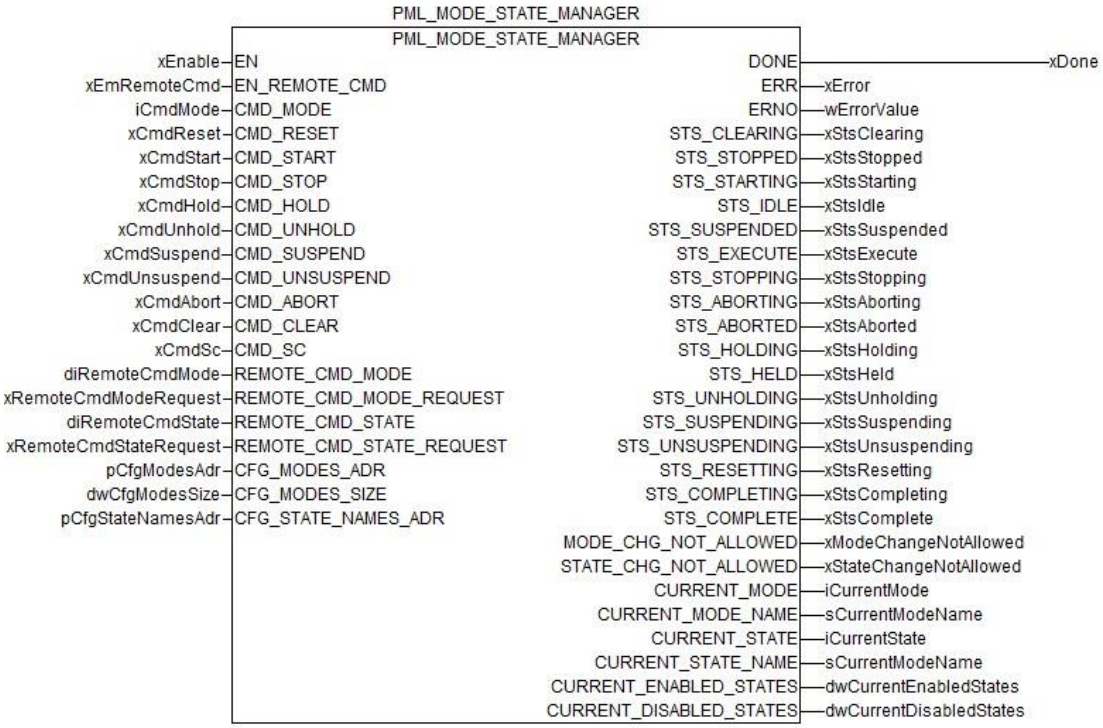
Size of mode configuration list that is connected at the corresponding ..ADR input.

CFG_STATE_NAMES_ADR

Data type: POINTER TO PML_STATE_NAMES_TYPE

This input points to the address of the PackML defined state names.

6.3.1.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERR (error)

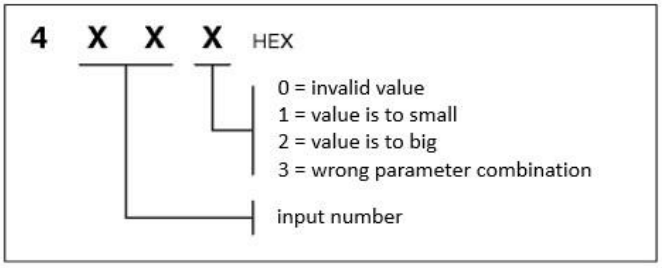
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



STS_CLEARING

Data type: BOOL

This output is set to TRUE when CLEARING the current machine state is.

STS_STOPPED

Data type: BOOL

This output is set to TRUE when STOPPED the current machine state is.

STS_STARTING

Data type: BOOL

This output is set to TRUE when STARTING the current machine state is.

STS_IDLE

Data type: BOOL

This output is set to TRUE when IDLE the current machine state is.

STS_SUSPENDED

Data type: BOOL

This output is set to TRUE when SUSPENDED the current machine state is.

STS_EXECUTE

Data type: BOOL

This output is set to TRUE when EXECUTE the current machine state is.

STS_STOPPING

Data type: BOOL

This output is set to TRUE when STOPPING the current machine state is.

STS_ABORTING

Data type: BOOL

This output is set to TRUE when ABORTING the current machine state is.

STS_ABORTED

Data type: BOOL

This output is set to TRUE when ABORTED the current machine state is.

STS_HOLDING

Data type: BOOL

This output is set to TRUE when HOLDING the current machine state is.

STS_HELD

Data type: BOOL

This output is set to TRUE when HELD the current machine state is.

STS_UNHOLDING

Data type: BOOL

This output is set to TRUE when UNHOLDING the current machine state is.

STS_SUSPENDING

Data type: BOOL

This output is set to TRUE when SUSPENDING the current machine state is.

STS_UNSUSPENDING

Data type: BOOL

This output is set to TRUE when UNSUSPENDING the current machine state is.

STS_RESETTING

Data type: BOOL

This output is set to TRUE when RESETTING the current machine state is.

STS_COMPLETING

Data type: BOOL

This output is set to TRUE when COMPLETING the current machine state is.

STS_COMPLETE

Data type: BOOL

This output is set to TRUE when COMPLETE the current machine state is.

MODE_CHG_NOT_ALLOWED

Data type: BOOL

This output is set to TRUE when a mode change is currently not allowed.

STATE_CHG_NOT_ALLOWED

Data type: BOOL

This output is set to TRUE when a state change is currently not allowed.

CURRENT_MODE

Data type: INT

This output gives out the number associated with the currently active mode.

CURRENT_MODE_NAME

Data type: String [20]

This output gives out the name of the currently active mode.

CURRENT_STATE

Data type: INT

This output gives out the number associated with the currently active state.

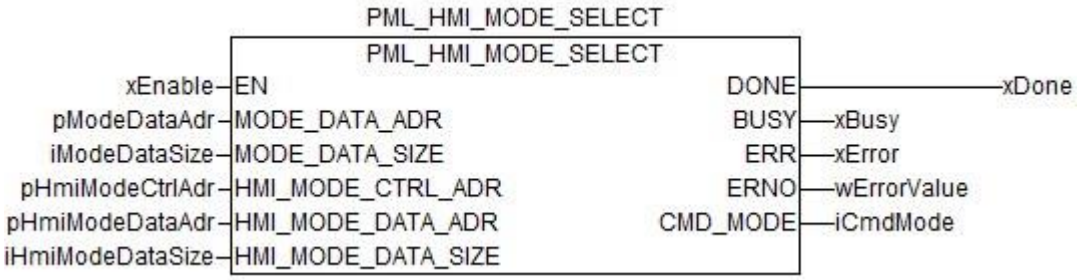
CURRENT_STATE_NAME

Data type: String [20]

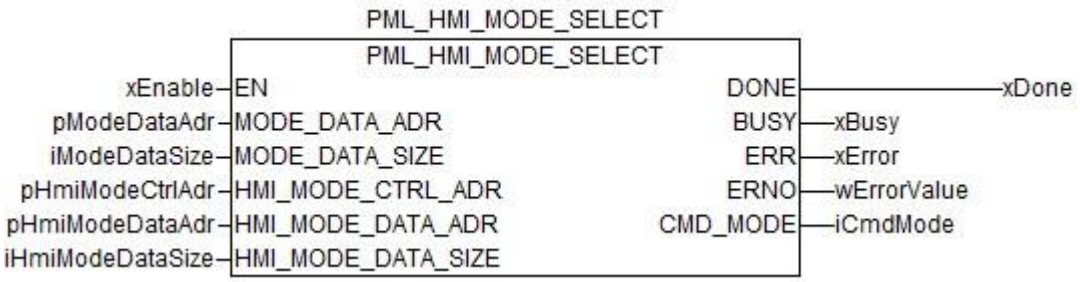
This output gives out the name of the currently active state.

6.3.2 PML_HMI_MODE_SELECT

The Mode Select function block for the HMI is used to display the different modes available, including the corresponding controls. It allows you to activate a specific mode via the HMI.



6.3.2.1 Input Description



EN (enable)

Data type: BOOL

In order to enable the Function Block processing, input EN has to be continuously set to TRUE. The block is not processed if input. While EN = FALSE input is set to TRUE, the inputs are continuously checked for validity and plausibility. If this is not the case, processing is aborted and corresponding error is displayed at output ERR/ERNO.

MODE_DATA_ADR

Data type: POINTER TO PML_MODE_CFG_TYPE

This input can be connected to the address of the list which shows the mode configuration on the HMI.

MODE_DATA_SIZE

Data type: DWORD

Size of mode configuration list that is connected at the corresponding ..ADR input.

HMI_MODE_CTRL_ADR

Data type: POINTER TO PML_HMI_MODE_CTRL_TYPE

This input can be connected to the address of the instance for the control buttons of mode selection.

HMI_MODE_DATA_ADR

Data type: POINTER TO PML_MODE_CFG_TYPE

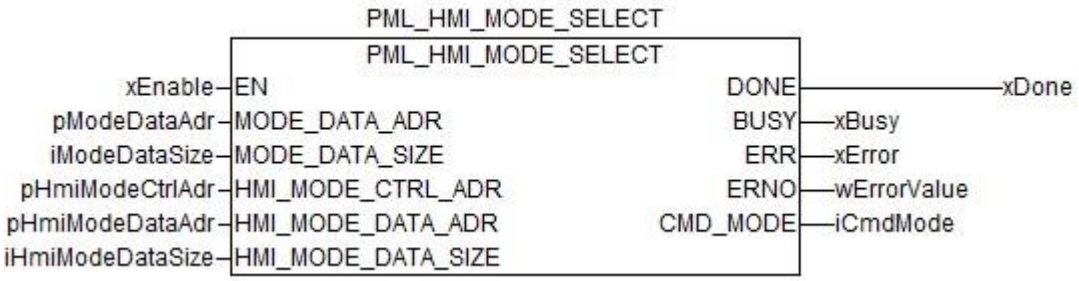
This input can be connected to the address of the hmi list which shows the mode configuration on the HMI.

HMI_MODE_SIZE

Data type: DWORD

Size of hmi mode configuration list that is connected at the corresponding ..ADR input.

6.3.2.2 Output Description



DONE (done)

Data type: BOOL

Output DONE indicates the processing state of the block. After completion or abortion of processing (due to an error), DONE is set to TRUE for one cycle. This output always has to be considered together with output ERR. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

BUSY (busy)

Data type: BOOL

This output is set to TRUE when EN is TRUE.

ERR (error)

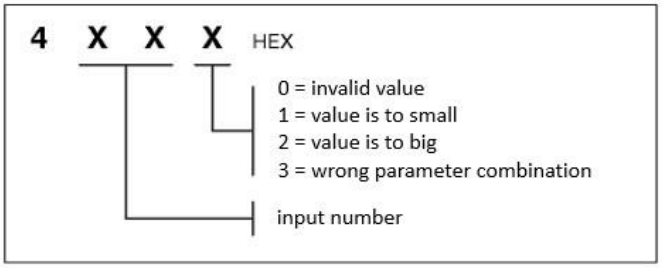
Data type: BOOL

Output ERR indicates whether an error occurred during data reception. If ERR is TRUE, an error occurred. In this case, the error number can be read at output ERNO.

ERNO (error number)

Data type: WORD

Output ERNO provides an error identifier if an error occurs during execution of the function block. ERNO always has to be considered together with the output ERR. The value output at ERNO is only valid if ERR is TRUE. The error messages encoding at output ERNO is explained below.



CMD_MODE

Data type: INT

This output gives out the target ID of the users HMI selection for mode change command.

6.4 Modes & States Datatype

- PML_HMI_MODE_CTRL_TYPE
- PML_MODE_CFG_TYPE
- PML_STATE_ENUM
- PML_STATE_NAMES_TYPES

6.4.1 PML_HMI_MODE_CTRL_TYPE

```
STRUCT
    (* Display controls *)
    Btn_Up1          : BOOL;
    Btn_Up2          : BOOL;
    Btn_Down1        : BOOL;
    Btn_Down2        : BOOL;
    Btn_Activate     : BOOL;
```

Btn_Up1

Data type: BOOL

This variable moves the connected list one mode up when TRUE.

Btn_Up2

Data type: BOOL

This variable moves the connected list one page size up when TRUE.

Btn_Down1

Data type: BOOL

This variable moves the connected list one mode down when TRUE.

Btn_Down2

Data type: BOOL

This variable moves the connected list one page size down when TRUE.

Btn_Activate

Data type: BOOL

This variable activates the highlighted mode.

6.4.2 PML_MODE_CFG_TYPE

```
STRUCT
    Name            : STRING(80);
    Id              : INT;
    DisableStates   : DWORD;
    ModeTransitions : DWORD;
    Active          : BOOL;
    InitState       : INT;
```

Name

Data type: STRING (80)

This variable stores the name of the mode.

Id

Data type: INT

This variable stores the ID of the mode.

DisableStates

Data type: DWORD

This variable stores the disabled states of the mode.

ModeTransitions

Data type: DWORD

This variable stores the states where a mode transition is allowed.

Active

Data type: BOOL

This variable stores whether the mode is active or not. It should also be used to define the first mode after starting up the PLC.

InitState

Data type: INT

This variable stores the first state after starting up the PLC when this mode is the first mode active.

6.4.3 PML_STATE_ENUM

```
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);
```

6.4.4 PML_STATE_NAMES_TYPE

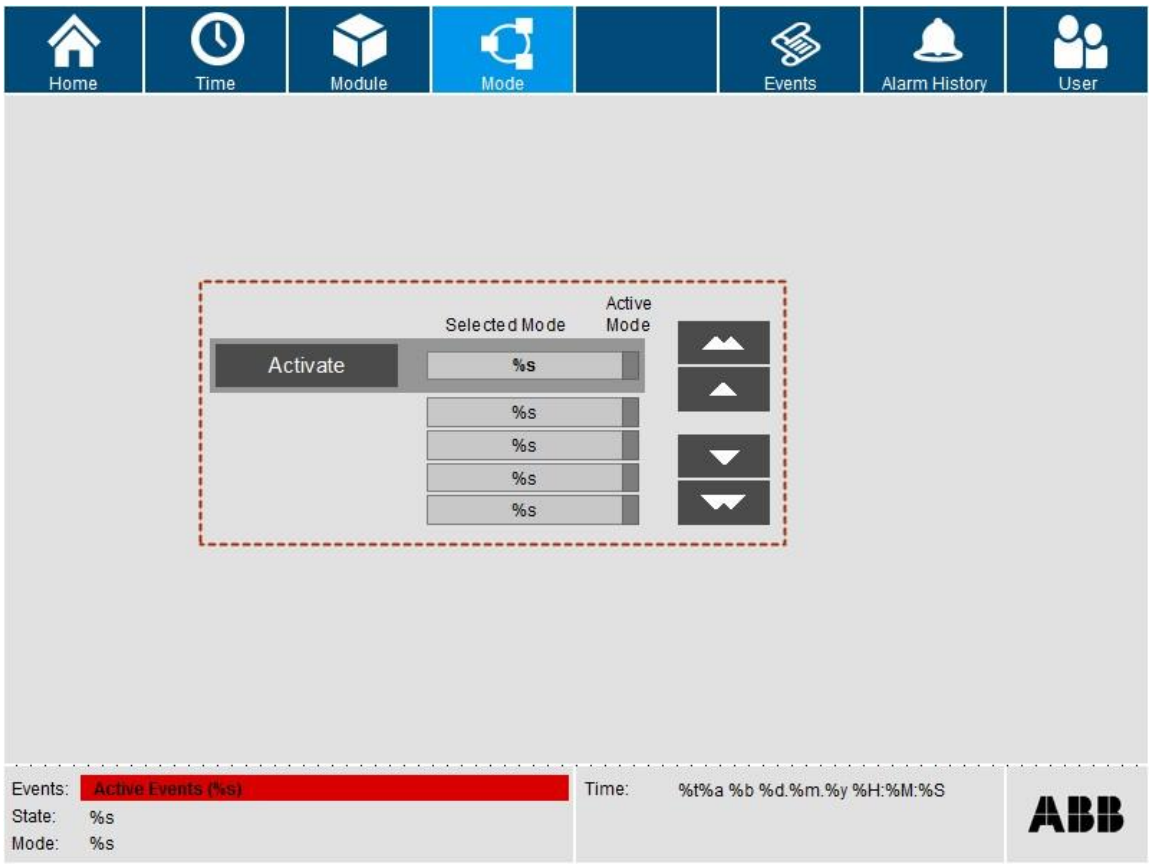
```
ARRAY[0..17] OF STRING :=
    'UNDEFINED',        (* 0 *)
    'CLEARING',         (* 1 *)
    'STOPPED',          (* 2 *)
    'STARTING',         (* 3 *)
    'IDLE',             (* 4 *)
    'SUSPENDED',        (* 5 *)
    'EXECUTE',          (* 6 *)
    'STOPPING',         (* 7 *)
    'ABORTING',         (* 8 *)
    'ABORTED',          (* 9 *)
    'HOLDING',          (* 10 *)
    'HELD',             (* 11 *)
    'UNHOLDING',        (* 12 *)
    'SUSPENDING',       (* 13 *)
    'UNSUSPENDING',     (* 14 *)
    'RESETTING',        (* 15 *)
    'COMPLETING',       (* 16 *)
    'COMPLETE';         (* 17 *)
```

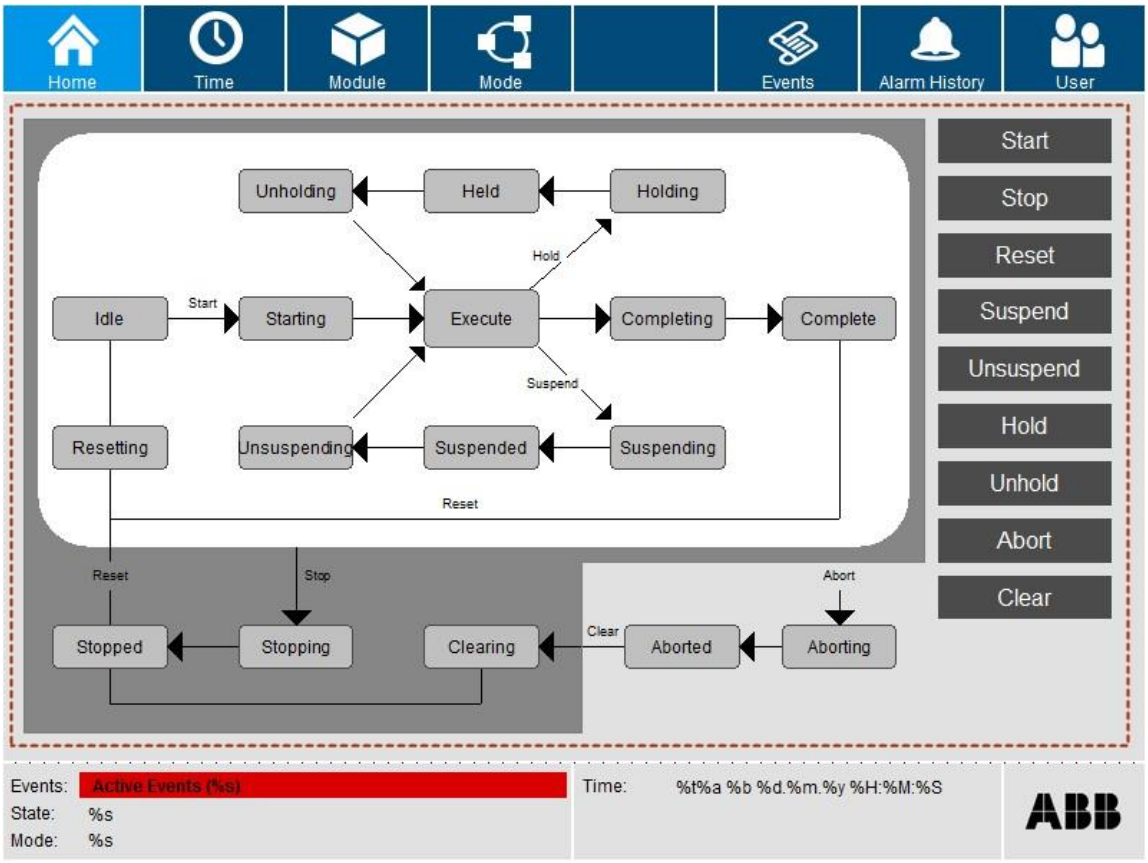
6.5 Visualization

The mode state visualization pages are used to display the currently active state machine. Additionally there is the possibility to initialize a mode or state change.

6.5.1 Webserver Template

means it is a predefined visualization template in CoDeSys and part of the Library





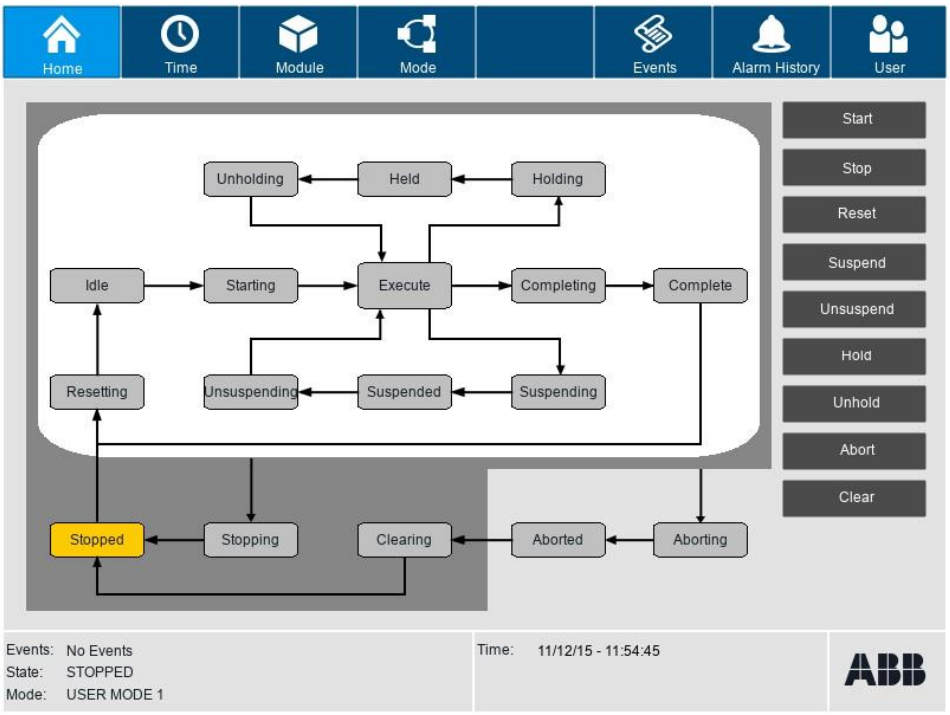
The CoDeSys webserver visualizations are implemented using place holders. Instead of reconfiguring each element on the used HMI page connect the variable instance to the referring visualization place holder.

6.5.2 CP600 Template

1. Mode Select

This page displays the configured modes. The top one in the list will be selected if the activate button is pressed.

2. Mode States



This page displays the different state machines according to the active mode. States get deactivated according to the mode. State changes can be initialized by using the buttons on the right.

6.5.3 Change display size

1. Change the array size, add another Mode to the default information and adjust all size inputs on the FBs in CoDeSys.
2. Reimport Tags from CoDeSys to the CP600.
3. Copy one list structure from the list and paste it onto the bottom.
4. Connect the fields to the correct variables.

7 PackML User guide – An example for modular programming

7.1 Benefits

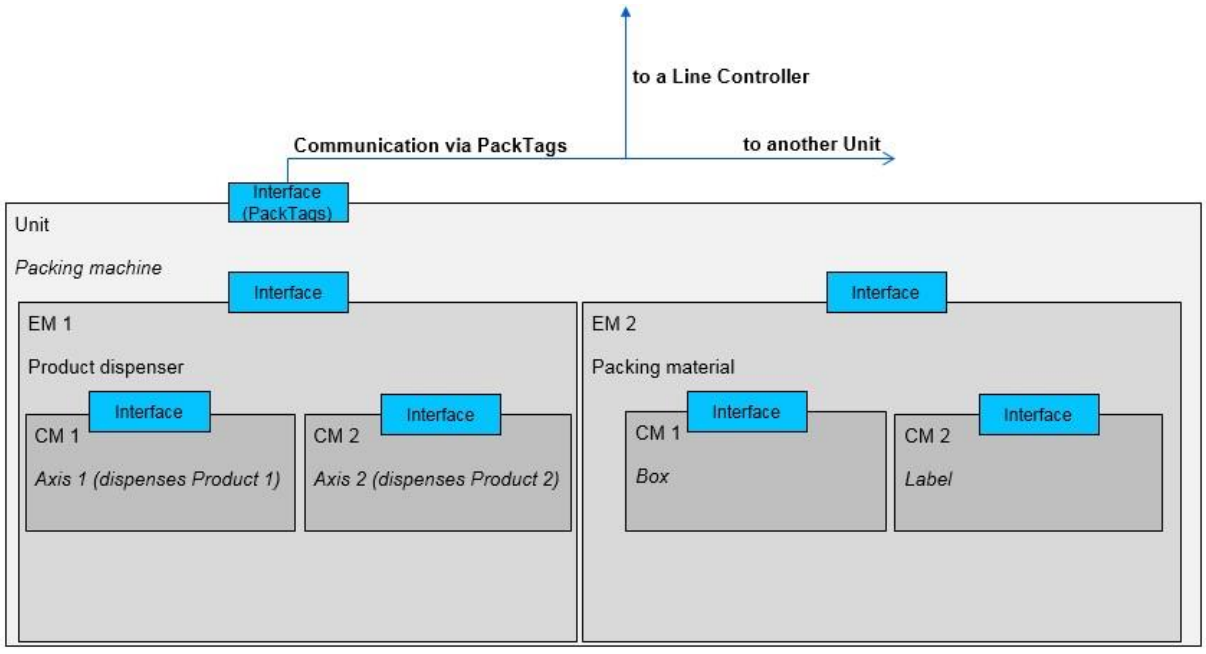
Modular programming allows an organized project structure. Due to all modules being independent of one other you are able to reuse them without having to change variables. You are also able to quickly perform a trouble shooting routine on single modules. Another benefit is a shorter time to market.

7.2 Modules

In general a module describes one or multiple logical code segments, of the whole application code, belonging together. They are helpful for the clearness of large projects because of planning, programing and testing of each module on its own. Using interfaces, they communicate data and information to their environment, e.g. other modules.

To be able to stand independently it is important that the code doesn't depend on information from a higher module.

They exchange data with other modules using interfaces. On Unit level there is an additional special interface, the PackML defined PackTags and also tags to and from the HMI. The PackML command tags to send requests to the Mode State Manager and the PackML status tags to send the proper reply to the command. This is for a common communication method to a line controller or another Unit.



All communication is done through these interfaces. It lets the modules write their information on the interface for the upper module to extract the information. The interfaces from submodules can be chosen more freely. If the project includes event handling the interface will have the event mailbox.

7.2.1 The usage of datatypes

All interfaces are made up of multiple datatypes. The HMI communicates with modules by having a datatype for each module which includes their functionalities and the datatypes from the lower modules.

7.3 Module-to-module Interface

For Module-to-Module communication there are datatypes used that unlike for the HMI only get used once. If one module controls another one, it doesn't know what happens inside the lower module. Therefore it sends a command to the next module after the current module has fulfilled all its requirements.

7.3.1 Commands

There is a specific datatype for the commands that a module can send to the next module. It lets the higher module tell the lower module to for example switch states. The lower module then can follow through with its code and can react to the command in many different ways. Whether or not the command was followed through with, the higher module will only know if it gets an appropriate Status in return.

7.3.2 Status

Similar to the commands there is a specific datatype for the module to module status updates. Statuses are information that lets the higher module know that for example something is complete. Just like for the commands it can only be passed to the one higher module. If the information needs to be passed on further, it has to be noted in the higher modules code.

7.3.3 Event List

As explained in Chapter 2.2 in the Library Description, there is one summated list for each module. These lists are stored in the module so that they can be send to a higher module using its interface (mailbox). The Unit list will then include its events, as well as all the lower ones.

7.3.4 Example given by template project

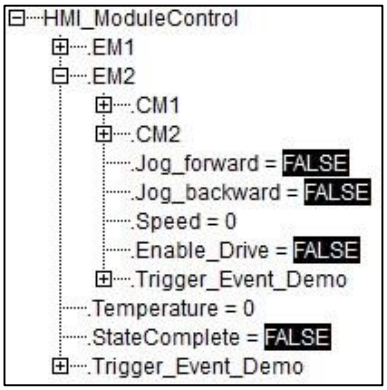
```
STRUCT
    Cmd_Reset                : BOOL;
    Cmd_Abort                : BOOL;
    Cmd_Stop                 : BOOL;
    Cmd_Start                : BOOL;
    Cmd_Clear                : BOOL;
    Cmd_Hold                 : BOOL;
    Cmd_UnHold               : BOOL;
    Cmd_Suspend              : BOOL;
    Cmd_Unsuspend            : BOOL;
    Sts_Clearing_SC          : BOOL;
    Sts_Starting_SC          : BOOL;
    Sts_Execute_SC           : BOOL;
    Sts_Stopping_SC          : BOOL;
    Sts_Aborting_SC          : BOOL;
    Sts_Holding_SC           : BOOL;
    Sts_UnHolding_SC         : BOOL;
    Sts_Suspending_SC        : BOOL;
    Sts_UnSuspending_SC      : BOOL;
    Sts_Resetting_SC         : BOOL;
    Sts_Completing_SC        : BOOL;
```

For Module-to-Module communication there are used two datatypes. One is for the commands and one is for the statuses. They are all BOOLs. The commands are used to trigger a certain section of the code. For example, if the program code aborting is called, it has the condition that the command aborting is set to TRUE. If the other conditions are also fulfilled and the code is executed, it sends a status bit back to the interface that sets the Sts_Aborting_SC to TRUE. This information is always available for only one level. What the next module does with this information is not important.

7.4 HMI Interfaces

All HMI tags are in the Interface of the Unit. The differences between these interfaces, from the Module-to-Module interfaces is the used datatypes. There are only two functionalities needed, either sending commands from the Unit down to the submodules or sending status reports back up to the Unit. To simplify the information, exchange each datatype includes, the commands/statuses from its own module but also the commands/statuses datatypes from all lower submodules.

This allows the user to add a functionality to a lower submodule without having to change every other datatype from the above modules.



7.4.1 Commands (HMI-to-module)

The commands coming from the HMI start on Unit level and are distributed downwards to the submodules according to the hierarchy. Commands need permission from each module before it can be sent to down to the next submodule. The information exchange can be interrupted in every submodule.

Possible reasons for denying permission:

– The current state – operation not allowed

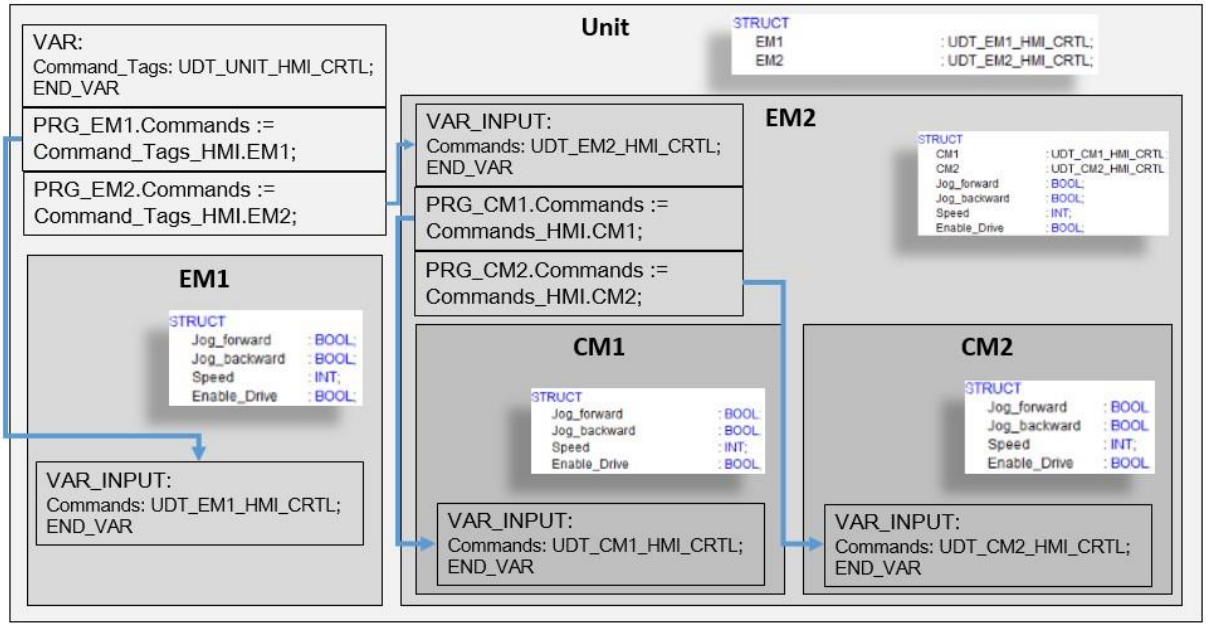
– The current mode – e.g. in maintenance mode no remote commands are allowed - Current position

7.4.2 Status (Module-to-HMI)

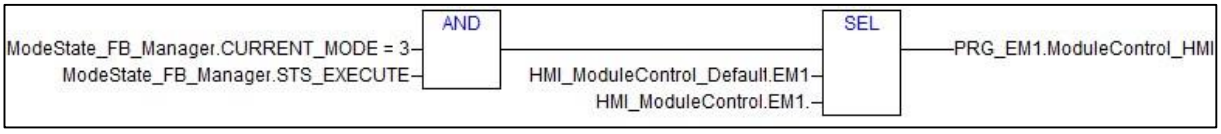
For sending a status report it is the same procedure working the other way round. The Unit application code receive information from its submodules, waits till the needed data is complete and then writes it on the HMI Tags.

7.4.3 Example given by template project

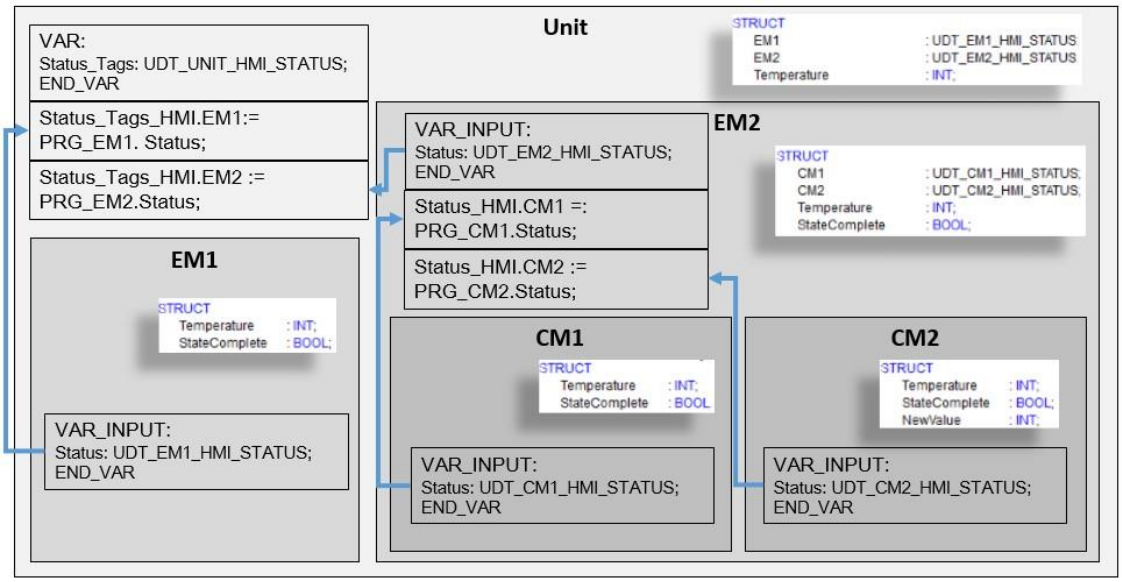
As displayed below commands are passed down via assignments.



This example shows that the Module Control command for the EM1 only gets copied to the program code when the Mode State Manager is in the mode Production and in the state Execute.



For statuses the information is passed upwards as shown in the next picture.



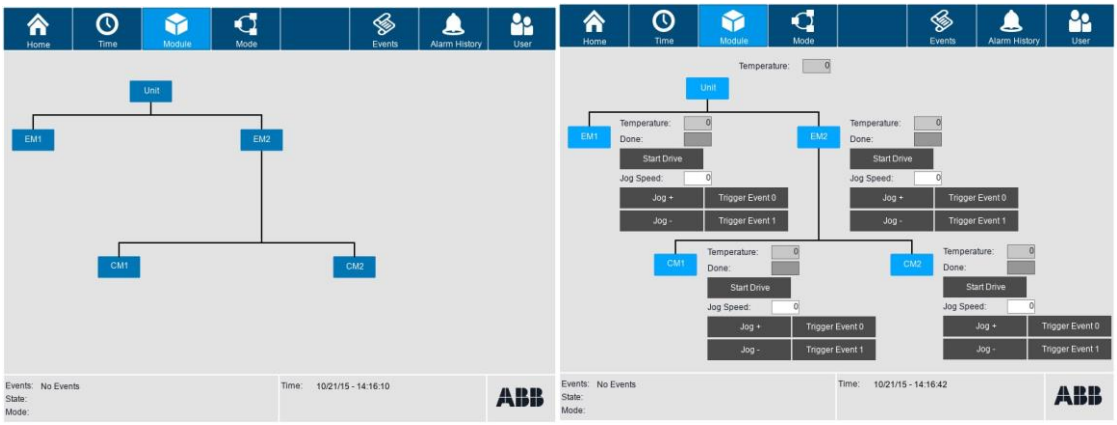
8 PackML User Guide – Tips & Tricks

8.1 Things to consider when using CP600 templates

The template pages are created with widgets from the Widget Gallery. All pages have navigation bar on the top of the page in blue and commands, if available, on the right in grey.

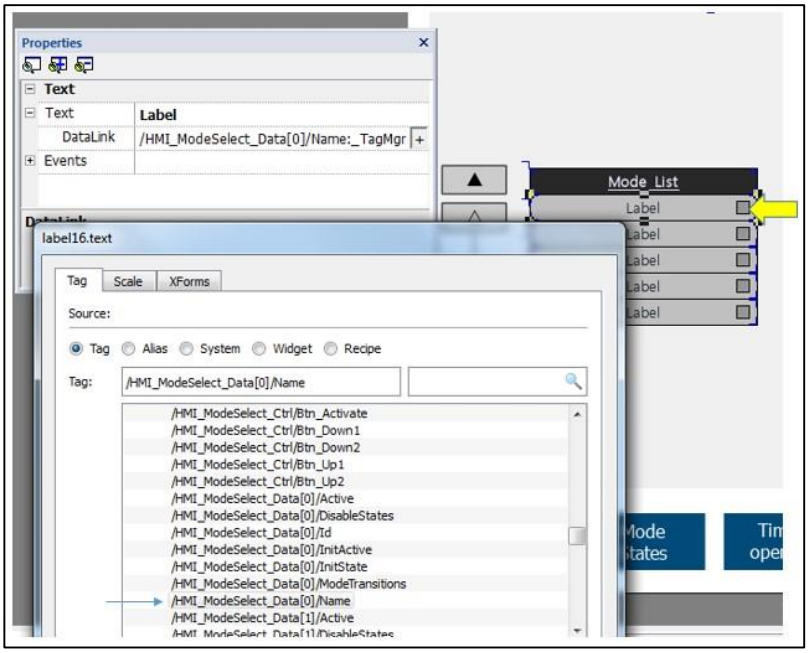
On the bottom of each page there is a display which shows the date and time and the current mode and state. It also shows in red the number of active events (if there are any), in yellow the number of total events (only if all events are not triggered), and “no events” if there are currently no alarms or warnings (triggered or not).

The Module Control page gets a second navigation bar. By clicking on the blue buttons in the module / machine structure, individual module commands and module statuses are shown.



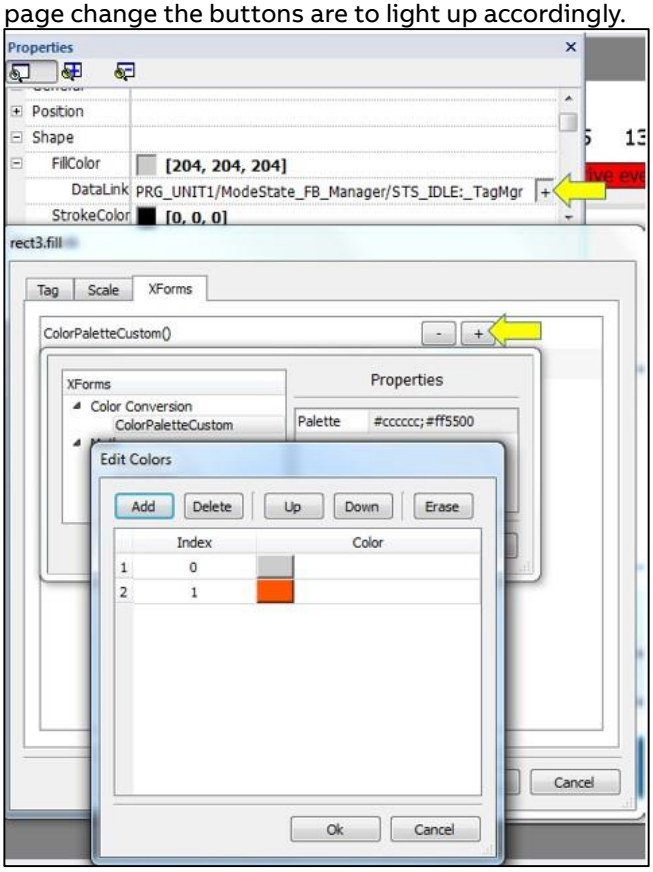
8.1.1 Things to consider when creating Template Pages

1. Each widget, which needs an action happing on command (a variable changing or a button being pressed) has to have the appropriate variable declared into its properties.

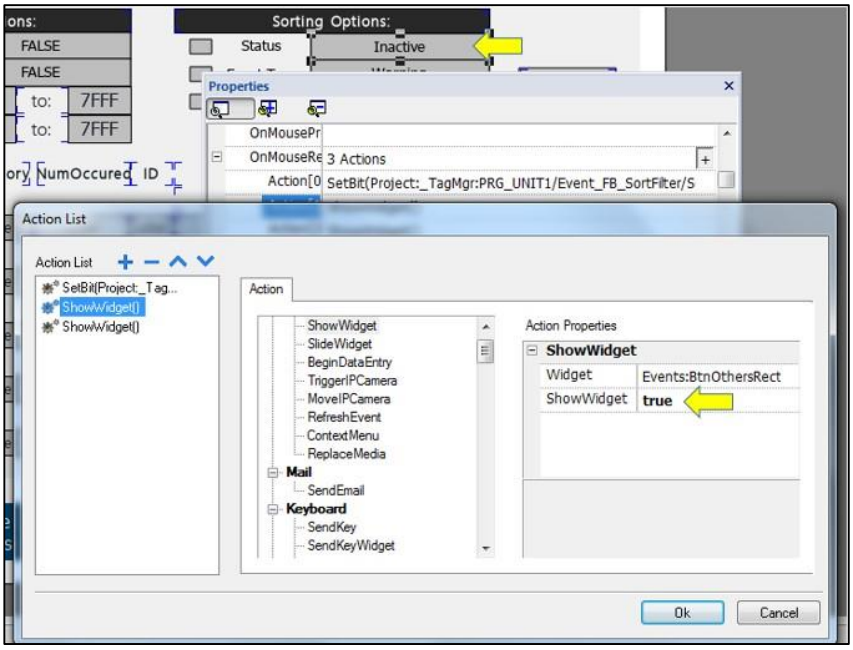


2. Each widget can also change color on a bit change. This bit can come from an imported tag or from a separate protocol. In the image color is orange on True (1) and grey on False (0).

   In the example project the self-created tags are used for the navigation bar when on a page change the buttons are to light up accordingly.

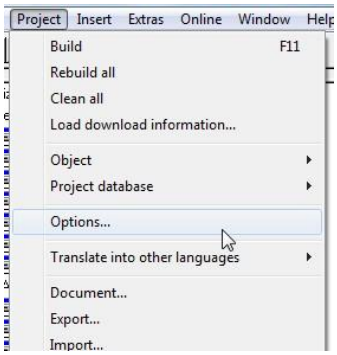


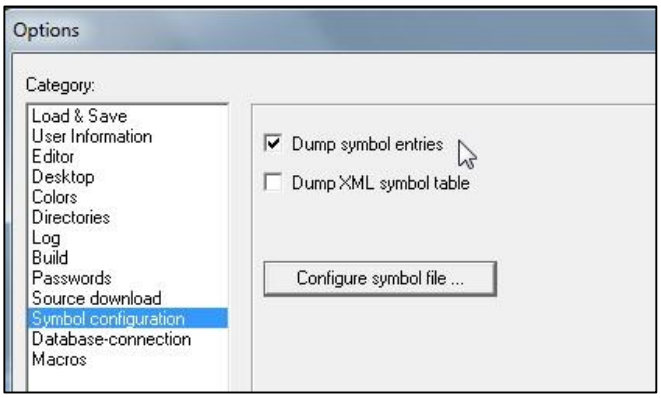
3. When wanting to switch between two texts, depending on for example the sorting option (Inactive or active) the user needs to use “show widget” to set showing one to True and the other one to False depending on the bit of the variable.



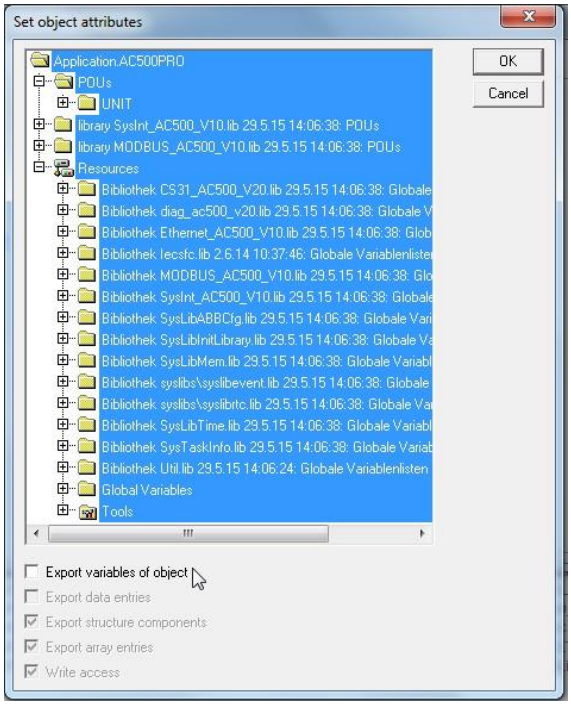
8.1.2 Importing Tags from CoDeSys

1. In CoDeSys under options, make sure the “Dump symbol entries” is selected.

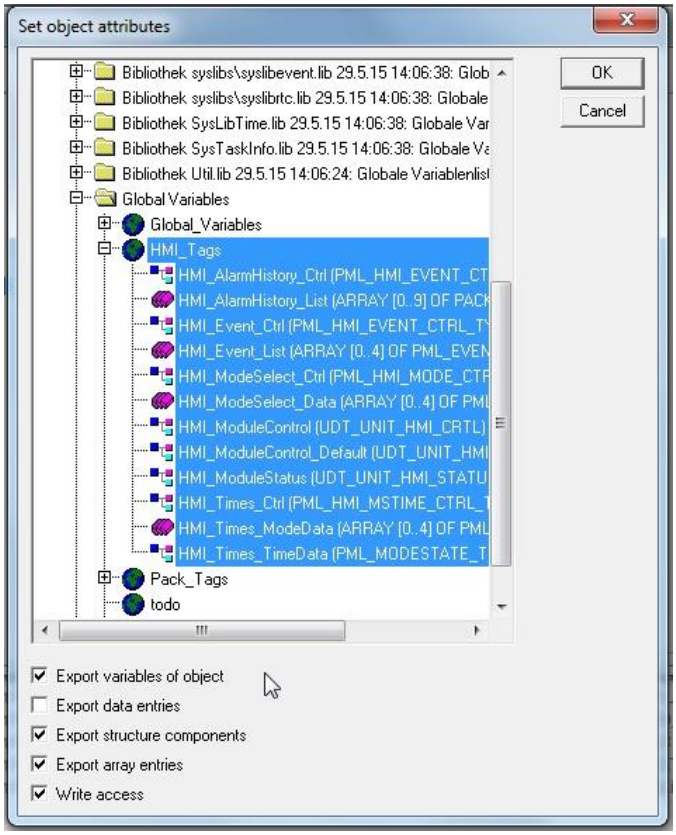




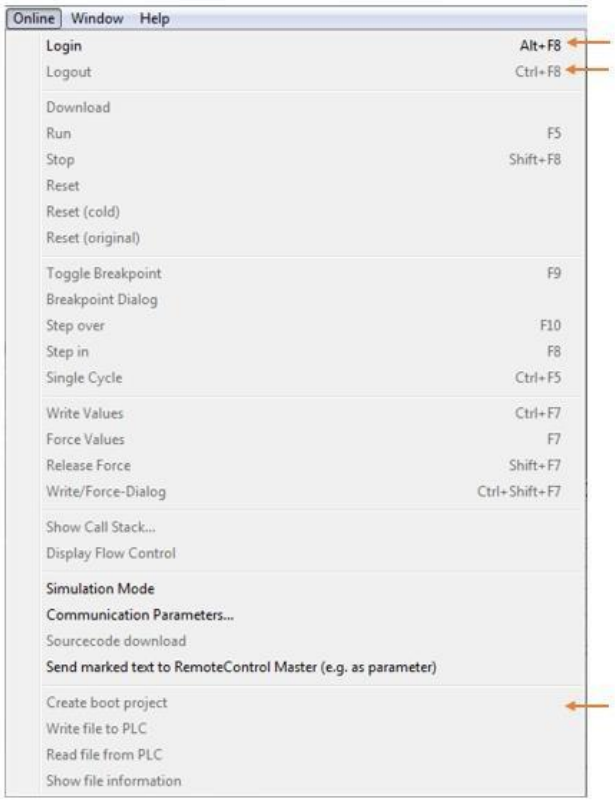
2. First, click on Configure symbol file and select all folders and unselect the “export variables of object” field.



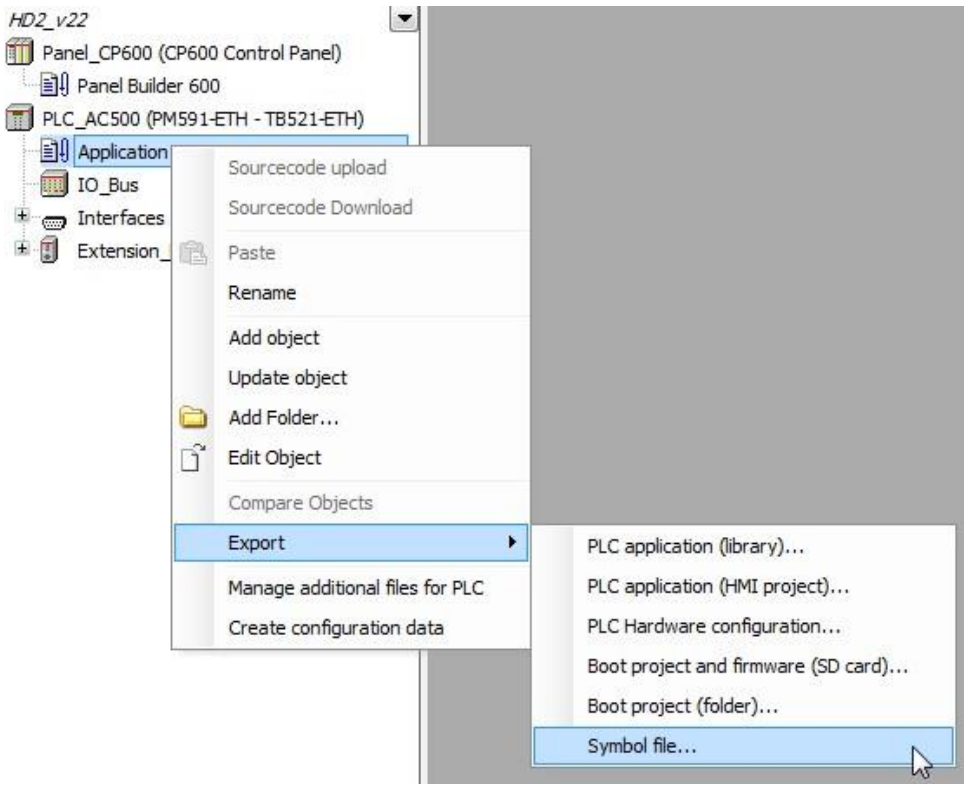
3. Press ok twice and re-enter the Configure Symbol file. Now you can select single variable files and select the “export variables of object” field to the ones that should be imported into the Panel Builder.



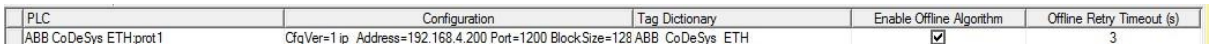
4. After selecting all wanted variables, clean all, rebuild all, log in and create a boot project.



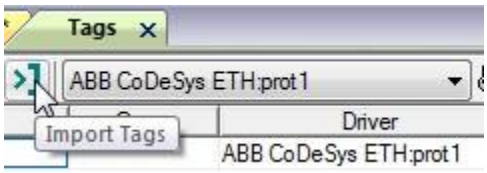
5. Go into the Automation Builder, right click on Application, under export select Symbol file. Choose a name for your file and save it.



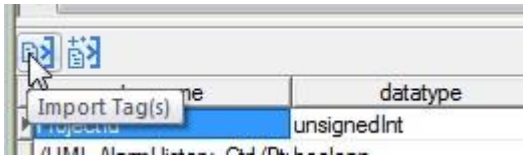
6. Open the Panel Builder and create under Protocols one for the Tags that are to be imported. Make sure to include the correct IP address under Configuration.



7. In the Tags, click on “Import Tags” and select the correct controller, already selected in the protocol. Select the appropriate file.



8. At the bottom all imported tags are shown. The user must select all and press import tags at the bottom.



9 PackML User Guide – Glossary

BOOL

Variables of the type BOOL can have the values TRUE and FALSE. For this, 8 bit of memory space are reserved.

Integer Data Types

BYTE, DINT, DWORD, INT, WORD

The different numerical types are responsible for a different numerical range. Due to this, it is possible that information are lost when converting greater data types to smaller data types.

For integer data types the following range limits are valid:

Type	BYTE	INT	DINT	WORD	DWORD
Lower limit	0	-32768	-2147483648	0	0
Upper limit	255	32767	2147483647	65535	4294967295
Memory space	8 bits	16 bits	32 bits	16 bits	32 bits

Functions

Functions are subroutines which have multiple input parameters and return exactly one result element. The returned result can be of an elementary or a derived data type. Due to this, a function may also return an array, a structure, an array of structures and so on. For the same input parameters, functions always return the same result (they do not have an internal memory).

Therefore, the following rules can be derived:

Within functions, global variables can neither be read nor written.

Within functions, absolute operands can neither be read nor written.

Within functions, function Function Blocks must not be called.

Function Blocks

Function Blocks are subroutines which can have as many inputs, outputs and internal variables as required. They are called from a program or from another Function Block. As they can be used several times (with different data records), Function Blocks (code and interface) can be considered as type. When assigning an individual data record

(declaration) to the Function Block, a Function Block instance is generated. In contrast to functions, Function Blocks can contain statically local data which are saved from one call to the next. Therefore e.g. counters can be realized which may not forget their counter value. I.e. Function Blocks can have an internal memory.

Functions and Function Blocks differ in two essential points:

A Function Block has multiple output parameters, a function only one. The output parameters of functions and Function Blocks differ syntactically.

In contrast to a function, a Function Block can have an internal memory.

Function Blocks with historical values (memory)

For Function Blocks with historical values it has to be observed that instance names may not be defined several times if different data sets should be called.

Function Blocks without historical values (memory)

For Function Blocks without historical values only one instance has to be defined for the Function Block type. This instance can be used for several calls of the Function Block (also with different I/O values).

10 Appendix

10.1 Attachments

State IDs, predefined by the OMAC

0	Undefined
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”

Mode IDs, predefined by the OMAC, definition of user defined modes is possible

0	Invalid
1	Production
2	Maintenance
3	Manual
04 - 31	User Definable

State change commands, predefined by the OMAC

0	Undefined
1	Reset
2	Start
3	Stop
4	Hold
5	Unhold
6	Suspend
7	Unsuspend
8	Abort
9	Clear

10.2 Command Tags (Complete Listing)

Command			TAGNAME	DATA TYPE
UnitMode			UnitName.Command.UnitMode	Int (32bit)
UnitModeChangeRequest			UnitName.Command.UnitModeChangeRequest	Bool
MachSpeed			UnitName.Command.MachSpeed	Real
MaterialInterlock			UnitName.Command.MaterialInterlock	Bool Structure
CntrlCmd			UnitName.Command.CntrlCmd	Int (32bit)
CmdChangeRequest			UnitName.Command.CmdChangeRequest	Bool
RemoteInterface[#]			UnitName.Command.RemoteInterface[#]	Interface
	Number		UnitName.Command.RemoteInterface[#].Number	Int (32bit)
	ControlCmdNumber		UnitName.Command.RemoteInterface[#].ControlCmdNumber	Int (32bit)
	CmdValue		UnitName.Command.RemoteInterface[#].CmdValue	Int (32bit)
	Parameter[#]		UnitName.Command.RemoteInterface[#].Parameter[#]	Descriptor Structure
		ID	UnitName.Command.RemoteInterface[#].Parameter[#].ID	Int (32bit)
		Name	UnitName.Command.RemoteInterface[#].Parameter[#].Name	String
		Unit	UnitName.Command.RemoteInterface[#].Parameter[#].Unit	String
		Value	UnitName.Command.RemoteInterface[#].Parameter[#].Value	Real
Parameter[#]			UnitName.Command.Parameter[#]	Descriptor Structure
	ID		UnitName.Command.Parameter[#].ID	Int (32bit)
	Name		UnitName.Command.Parameter[#].Name	String
	Unit		UnitName.Command.Parameter[#].Unit	String

	Value		UnitName.Command.Parameter[#].Value	User Defined
Product[#]			UnitName.Command.Product[#]	Product Structure
	ProductID		UnitName.Command.Product[#].ProductID	Int (32bit)
	ProcessVariables[#]		UnitName.Command.Product[#].ProcessVariables[#]	Descriptor Structure
		ID	UnitName.Command.Product[#].ProcessVariables[#].ID	Int (32bit)
		Name	UnitName.Command.Product[#].ProcessVariables[#].Name	String
		Unit	UnitName.Command.Product[#].ProcessVariables[#].Unit	String
		Value	UnitName.Command.Product[#].ProcessVariables[#].Value	Real
	Ingredients[#]		UnitName.Command.Product[#].Ingredients[#]	Ingredient
		IngredientID	UnitName.Command.Product[#].Ingredients[#].IngredientID	Int (32bit)
		Parameter[#]	UnitName.Command.Product[#].Ingredients[#].Parameter[#]	Descriptor Structure
			UnitName.Command.Product[#].Ingredients[#].Parameter[#].ID	Int (32bit)
			UnitName.Command.Product[#].Ingredients[#].Parameter[#].Name	String
			UnitName.Command.Product[#].Ingredients[#].Parameter[#].Unit	String
			UnitName.Command.Product[#].Ingredients[#].Parameter[#].Value	Real

10.3 Status Tags (Complete Listing)

Status			TAGNAME	DATATYPE
UnitModeCurrent			UnitName.Status.UnitModeCurrent	Int (32bit)
UnitModeRequested			UnitName.Status.UnitModeRequested	Bool
UnitModeChangeInProcess			UnitName.Status.UnitModeChangeInProcess	Bool
StateCurrent			UnitName.Status.StateCurrent	Int (32bit)
StateRequested			UnitName.Status.StateRequested	Int (32bit)
StateChangeInProcess			UnitName.Status.StateChangeInProcess	Bool
MachSpeed			UnitName.Status.MachSpeed	Real
CurMachSpeed			UnitName.Status.CurMachSpeed	Real
MaterialInterlock[#]			UnitName.Status.MaterialInterlock	Bool Array [32]
EquipmentInterlock			UnitName.Status.EquipmentInterlock	Bool Structure [2]
	Blocked		UnitName.Status.EquipmentInterlock.Blocked	Bool
	Starved		UnitName.Status.EquipmentInterlock.Starved	Bool
RemoteInterface[#]			UnitName.Status.RemoteInterface[#]	Interface
	Number		UnitName.Status.RemoteInterface[#].Number	Int (32bit)
	ControlCmdNumber		UnitName.Status.RemoteInterface[#].ControlCmdNumber	Int (32bit)
	CmdValue		UnitName.Status.RemoteInterface[#].CmdValue	Int (32bit)
	Parameter[#]		UnitName.Status.RemoteInterface[#].Parameter[#]	Descriptor Structure
		ID	UnitName.Status.RemoteInterface[#].Parameter[#].ID	Int (32bit)
		Name	UnitName.Status.RemoteInterface[#].Parameter[#].Name.	String
		Unit	UnitName.Status.RemoteInterface[#].Parameter[#].Unit	String

		Value	UnitName.Status.RemoteInterface[#].Parameter[#].Value	Real
Parameter[#]			UnitName.Status.Parameter[#]	Descriptor Structure
	ID		UnitName.Status.Parameter[#].ID	Int (32bit)
	Name		UnitName.Status.Parameter[#].Name	String
	Unit		UnitName.Status.Parameter[#].Unit	String
	Value		UnitName.Status.Parameter[#].Value	User Defined
Product[#]			UnitName.Status.Product[#]	Product Structure
	ProductID		UnitName.Status.Product[#].ProductID	Int (32bit)
	ProcessVariables[#]		UnitName.Status.Product[#].ProcessVariables[#]	Descriptor Structure
		ID	UnitName.Status.Product[#].ProcessVariables[#].ID	Int (32bit)
		Name	UnitName.Status.Product[#].ProcessVariables[#].Name	String
		Unit	UnitName.Status.Product[#].ProcessVariables[#].Unit	String
		Value	UnitName.Status.Product[#].ProcessVariables[#].Value	Real
	Ingredients[#]		UnitName.Status.Product[#].Ingredients[#]	Ingredient
		IngredientID	UnitName.Status.Product[#].Ingredients[#].IngredientID	Int (32bit)
		Parameter[#]	UnitName.Status.Product[#].Ingredients[#].Parameter[#]	Descriptor Structure
			UnitName.Status.Product[#].Ingredients[#].Parameter[#].ID	Int (32bit)
			UnitName.Status.Product[#].Ingredients[#].Parameter[#].Name	String
			UnitName.Status.Product[#].Ingredients[#].Parameter[#].Unit	String
			UnitName.Status.Product[#].Ingredients[#].Parameter[#].Value	Real

10.4 Admin Tags (Complete Listing)

Admin			TAGNAME	DATATYPE
Parameter[#]			UnitName.Admin.Parameter[#]	Descriptor Structure
	ID		UnitName.Admin.Parameter[#].ID	Int (32bit)
	Name		UnitName.Admin.Parameter[#].Name	String
	Unit		UnitName.Admin.Parameter[#].Unit	String
	Value		UnitName.Admin.Parameter[#].Value	Real
Alarm[#]			UnitName.Admin.Alarm[#]	Alarm Structure
	Trigger		UnitName.Admin.Alarm[#].Trigger	Bool
	ID		UnitName.Admin.Alarm[#].ID	Int (32bit)
	Value		UnitName.Admin.Alarm[#].Value	Int (32bit)
	Message		UnitName.Admin.Alarm[#].Message	String
	Category		UnitName.Admin.Alarm[#].Category (Event Grouping)	Int (32bit)
	AlmDateTime		UnitName.Admin.Alarm[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Alarm[#].AlmDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.Alarm[#].AlmDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.Alarm[#].AlmDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Alarm[#].AlmDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Alarm[#].AlmDateTime[4]	Int (32bit)

		[5] (sec)	UnitName.Admin.Alarm[#].AlmDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Alarm[#].AlmDateTime[6]	Int (32bit)
	AckDateTime		UnitName.Admin.Alarm[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Alarm[#].AckDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.Alarm[#].AckDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.Alarm[#].AckDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Alarm[#].AckDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Alarm[#].AckDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Alarm[#].AckDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Alarm[#].AckDateTime[6]	Int (32bit)
AlarmExtent			UnitName.Admin.AlarmExtent	Int(32bit)
AlarmHistory[#]			UnitName.Admin.AlarmHistory[#]	Alarm Structure
	Trigger		UnitName.Admin.AlarmHistory[#].Trigger	Bool
	ID		UnitName.Admin.AlarmHistory[#].ID	Int (32bit)
	Value		UnitName.Admin.AlarmHistory[#].Value	Int (32bit)
	Message		UnitName.Admin.AlarmHistory[#].Message	String
	Category		UnitName.Admin.AlarmHistory[#].Category (Event Grouping)	Int (32bit)
	AlmDateTime		UnitName.Admin.AlarmHistory[#]	Date-Time Array
		[0] (year)	UnitName.Admin.AlarmHistory[#].AlmDateTime[0]	Int (32bit)

		[1] (month)	UnitName.Admin.AlarmHistory[#].AlmDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.AlarmHistory[#].AlmDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.AlarmHistory[#].AlmDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.AlarmHistory[#].AlmDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.AlarmHistory[#].AlmDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.AlarmHistory[#].AlmDateTime[6]	Int (32bit)
	AckDateTime		UnitName.Admin.AlarmHistory[#]	Date-Time Array
		[0] (year)	UnitName.Admin.AlarmHistory[#].AckDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.AlarmHistory[#].AckDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.AlarmHistory[#].AckDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.AlarmHistory[#].AckDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.AlarmHistory[#].AckDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.AlarmHistory[#].AckDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.AlarmHistory[#].AckDateTime[6]	Int (32bit)
AlarmHistoryExtent			UnitName.Admin.AlarmHistoryExtent	Int (32bit)
StopReason			UnitName.Admin.StopReason	Alarm Structure
	Trigger		UnitName.Admin.StopReason.Trigger	Bool
	ID		UnitName.Admin.StopReason.ID	Int (32bit)
	Value		UnitName.Admin.StopReason.Value	Int (32bit)
	Message		UnitName.Admin.StopReason.Message	String
	Category		UnitName.Admin.StopReason.Category (Event Grouping)	Int (32bit)

	AlmDateTime		UnitName.Admin.StopReason[#]	Date-Time Array
		[0] (year)	UnitName.Admin.StopReason[#].AlmDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.StopReason[#].AlmDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.StopReason[#].AlmDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.StopReason[#].AlmDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.StopReason[#].AlmDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.StopReason[#].AlmDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.StopReason[#].AlmDateTime[6]	Int (32bit)
	AckDateTime		UnitName.Admin.StopReason[#]	Date-Time Array
		[0] (year)	UnitName.Admin.StopReason[#].AckDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.StopReason[#].AckDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.StopReason[#].AckDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.StopReason[#].AckDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.StopReason[#].AckDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.StopReason[#].AckDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.StopReason[#].AckDateTime[6]	Int (32bit)
StopReasonExtent			UnitName.Admin.StopReasonExtent	Int (32bit)
Warning[#]			UnitName.Admin.Warning[#]	Alarm Structure
	Trigger		UnitName.Admin.Warning[#].Trigger	Bool
	ID		UnitName.Admin.Warning[#].ID	Int (32bit)
	Value		UnitName.Admin.Warning[#].Value	Int (32bit)
	Message		UnitName.Admin.Warning[#].Message	String

	Category		UnitName.Admin.Warning[#].Category (Event Grouping)	Int (32bit)
	AlmDateTime		UnitName.Admin.Warning[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Warning[#].AlmDateTime[0]	Int (32bit)
		[2] (day)	UnitName.Admin.Warning[#].AlmDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Warning[#].AlmDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Warning[#].AlmDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Warning[#].AlmDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Warning[#].AlmDateTime[6]	Int (32bit)
	AckDateTime		UnitName.Admin.Warning[#]	Date-Time Array
		[0] (year)	UnitName.Admin.Warning[#].AckDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.Warning[#].AckDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.Warning[#].AckDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.Warning[#].AckDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.Warning[#].AckDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.Warning[#].AckDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.Warning[#].AckDateTime[6]	Int (32bit)
WarningExtent			UnitName.Admin.WarningExtent	Int (32bit)
ModeCurrentTime[#]			UnitName.Admin.ModeCurrentTime[#]	Int (32bit)
ModeCumulativeTime[#]			UnitName.Admin.ModeCumulativeTime[#]	Int (32bit)
StateCurrentTime[#,#]	(Mode,State)		UnitName.Admin.StateCurrentTime[#,#] (Mode,State)	Int (32bit)
StateCumulativeTime[#,#]	(Mode,State)		UnitName.Admin.StateCumulativeTime[#,#] (Mode,State)	Int (32bit)

ProdConsumedCount[#]			UnitName.Admin.ProdConsumedCount[#]	Count Structure
	ID		UnitName.Admin.ProdConsumedCount[#].ID	Int(32bit)
	Name		UnitName.Admin.ProdConsumedCount[#].Name	String
	Unit		UnitName.Admin.ProdConsumedCount[#].Unit	String
	Count		UnitName.Admin.ProdConsumedCount[#].Count	Int(32bit)
	AccCount		UnitName.Admin.ProdConsumedCount[#].AccCount	Int(32bit)
ProdProcessedCount[#]			UnitName.Admin.ProdProcessedCount[#]	Count Structure
	ID		UnitName.Admin.ProdProcessedCount[#].ID	Int(32bit)
	Name		UnitName.Admin.ProdProcessedCount[#].Name	String
	Unit		UnitName.Admin.ProdProcessedCount[#].Unit	String
	Count		UnitName.Admin.ProdProcessedCount[#].Count	Int(32bit)
	AccCount		UnitName.Admin.ProdProcessedCount[#].AccCount	Int(32bit)
ProdDefectiveCount[#]			UnitName.Admin.ProdDefectiveCount[#]	Count Structure
	ID		UnitName.Admin.ProdDefectiveCount[#].ID	Int(32bit)
	Name		UnitName.Admin.ProdDefectiveCount[#].Name	String
	Unit		UnitName.Admin.ProdDefectiveCount[#].Unit	String
	Count		UnitName.Admin.ProdDefectiveCount[#].Count	Int(32bit)
	AccCount		UnitName.Admin.ProdDefectiveCount[#].AccCount	Int(32bit)
AccTimeSinceReset	AccTimeSinceReset		UnitName.Admin.AccTimeSinceReset	Int(32bit)
MachDesignSpeed			UnitName.Admin.MachDesignSpeed	Real
StatesDisabled			UnitName.Admin.StatesDisabled	Int(32bit)
PLCDateTime			UnitName.Admin.PLCDateTime	Date-Time Array

		[0] (year)	UnitName.Admin.PLCDateTime[0]	Int (32bit)
		[1] (month)	UnitName.Admin.PLCDateTime[1]	Int (32bit)
		[2] (day)	UnitName.Admin.PLCDateTime[2]	Int (32bit)
		[3] (hour)	UnitName.Admin.PLCDateTime[3]	Int (32bit)
		[4] (min)	UnitName.Admin.PLCDateTime[4]	Int (32bit)
		[5] (sec)	UnitName.Admin.PLCDateTime[5]	Int (32bit)
		[6] (usec)	UnitName.Admin.PLCDateTime[6]	Int (32bit)

ABB Automation Products GmbH
Eppelheimer Straße 82
69123 Heidelberg, Germany
Phone: +49 62 21 701 1444
Fax: +49 62 21 701 1382
E-Mail: plc.support@de.abb.com
www.abb.com/plc