

BRDK Architecture

Programming guidelines

Requirements

Develeopment tool:	Automation Studio 4.12 and newer
Hardware:	All CPU's

TABLE OF CONTENTS

Requirements	2
TABLE OF CONTENTS	3
1. Motivation	4
2. Architecture	4
2.1. Hierarchy and service modules	4
2.2. Equipment modules	4
2.3. Use of control modules	5
2.4. Program structure	6
3. Naming conventions	8
3.1 Variable naming	9
3.2 Structure / type naming	9
3.3 Constants naming	9
3.4 Global structures and variables naming	9
3.5 Pointer variable naming	10
3.6 IO variable naming	10
3.7 Enumerator naming	10
3.8 Function block naming	10
4. Descriptions in .var and .typ files	11
5. Alarm system	11
6. General coding guidelines	14
7. Handling non-indexed code in indexed modules	15

1. Motivation

The purpose of this document, is to describe the architecture of Automation Studio projects in BRDK, as well as setting out fixed rules for naming conventions, structures, etc.

2. Architecture

All projects must be set up in the framework and architecture, described in this document. All machines are divided into equipment modules in a hierarchical state machine, following the PackML standard and based on the BrdkPackML library.

2.1. Hierarchy and service modules

BRDK use templates for equipment modules, with and without servo axis, indexed equipment modules, and a selection of services, which can be connected in a hierarchical structure. Individual modules can be named, according to the machine requirements.

Figure 1 shows a **Main** module, which is the parent of child modules **Infeed**, **Roundtable** and **Fillers**. Take note that **Fillers** is an indexed module, meaning that one task will handle multiple instances of the same equipment module.

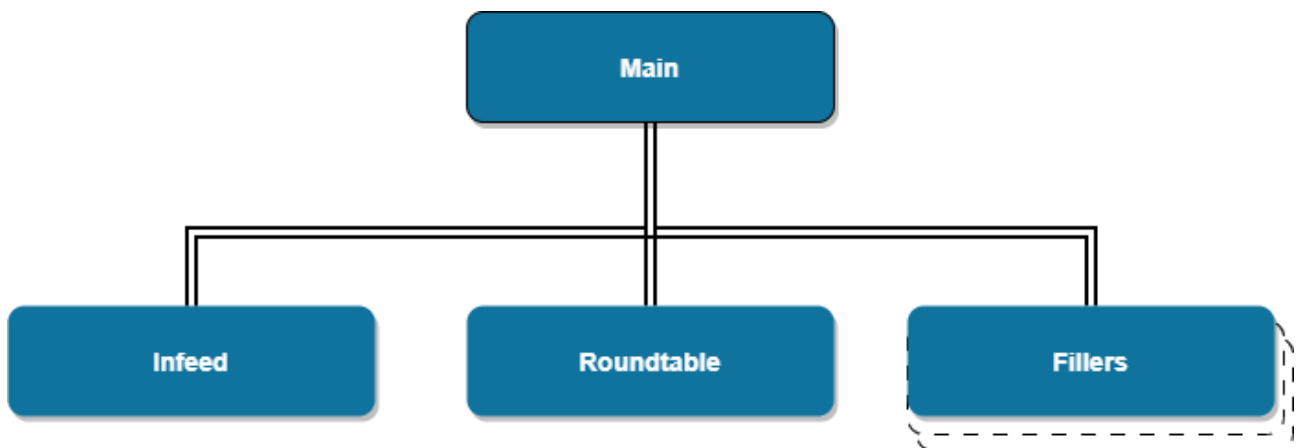


Figure 1 - Example module hierarchy

Services provide standard solutions for a wide range of features, that most projects will need:

- Alarm handling
- Filehandling
- Config and recipe handling
- Language selection
- Logger
- IO handling for simulation

The templates allow for advanced process related code to be added freely. Only restriction is to always keep the naming conventions, structure and PackML state model and hierarchy.

2.2. Equipment modules

All electromechanical parts of the machine must be divided into individual equipment modules, which must be interconnected in a hierarchy, as shown previously. How this should be done, requires an understanding of the machine, and it's processes.

There will always be a "**Main**" module in the top of the hierarchy. This module will be the one that handles normal start/stop functionality of the machine, and underlying modules in the parent/child relation will then follow the states of the **Main** module, according to the **PackML** standard.

An equipment module is implemented as a function block of the **EquipmentModule** type, declared in the **BrdkPackML** library.

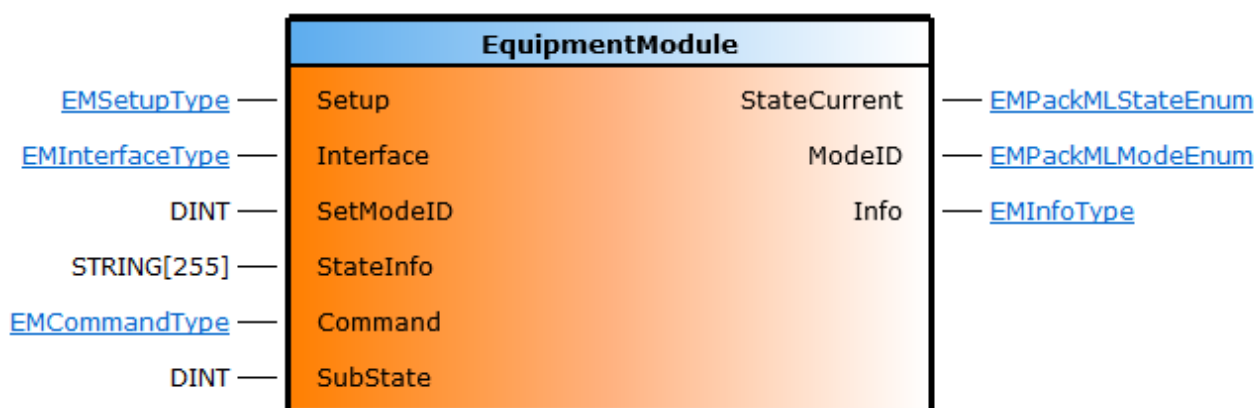


Figure 2- Interface of the EquipmentModule function block

An equipment module must always have it's own separate task. If the machine must support several instances of the same equipment module, it will be implemented as an array of **EquipmentModule** function blocks, which will then be called in a loop in the task. The way the template for indexed modules is implemented, it basically handles all instances of the module, so that the application programmer can program the process, in the exact same way as a single instance equipment module.

2.3. Use of control modules

Control modules can be used at the lowest level, to encapsulate hardware-near logic in a convenient way.

Control modules may be implemented as a function block, or as a standalone task, using the **ControlModule** function block.

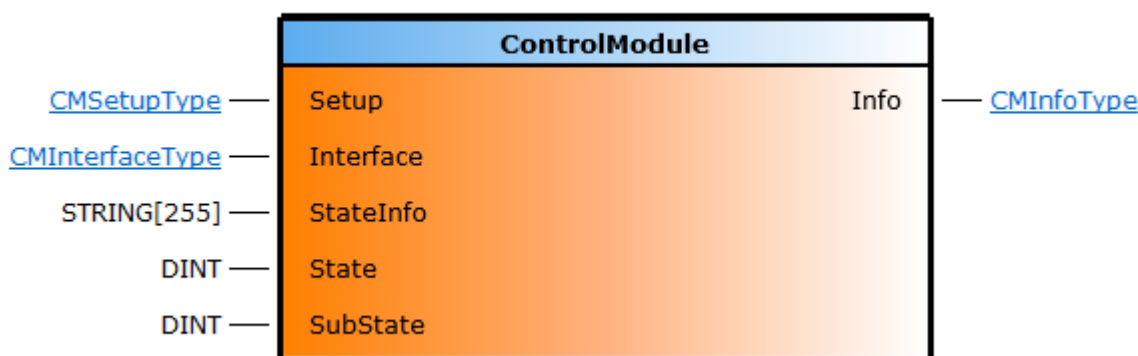


Figure 3 Interface of the ControlModule function block

If a control module is implemented as a function block, it should be done as a local function block in the task. Avoid implementing the control module in a user library. The reasoning for this is, that a user library is not download safe, meaning that any code change in the control module code, will require a reboot of the PLC, which is not desirable.

In either case, the interface of the control module should add a useful abstraction layer from the code or function blocks inside it. If the interface of the control module has largely the same interface as the function block(s) used internally in the control module, then this is usually a clear indicator that the use of a control module was not required.

If using a single servo axis, it will often not make sense to implement as a control module, since the **mpAxisBasic** function block already encapsulates all basic axis functionality. This usually just leads to an unnecessary layer, which does not add to the readability of the code.

2.4. Program structure

The project should have a structure as seen in Figure 4 - Project structure.

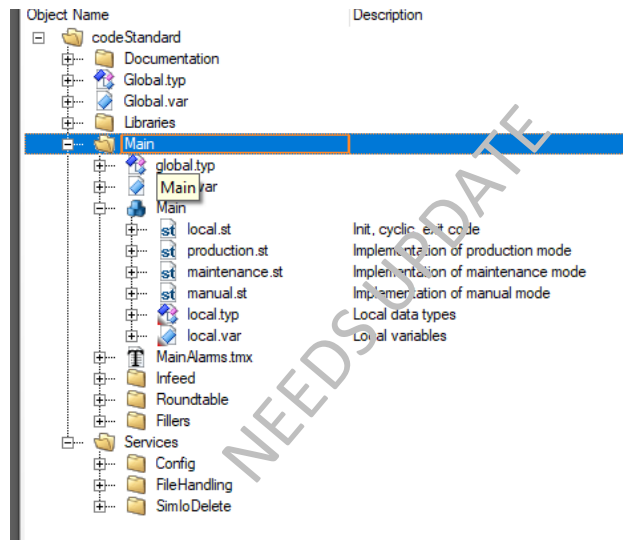


Figure 4 - Project structure

Each equipment module will have its own package, with contents as listed here:

global.typ : Type declaration file for global interface structures and config/recipe structures.

Global structures are used for interfacing between modules. Indexed modules will have an array of its global structures, so that each instance can be addressed individually.

Ideally everything would be local and contained within each module and all interfacing would be done with PV mapping. The implementation and practical use of PV mapping is useless and thereby benched for now.

Name	Type	& Reference	Replicable	Value	Des
glInfeedIfType			<input checked="" type="checkbox"/>		
cmd	glInfeedIfCmdType	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
status	glInfeedIfStatusTy...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		

gModuleNamelfType: The global interface type. All tasks global interface structures have these substructures:

gModuleNameStatusType: Contains all status variables from the module that are relevant for others. Only the module itself updates these variables.

gModuleNameCmdType: Contains all commands that the module can receive from other modules.

To avoid timing issues and race conditions between modules running in different cycletimes, commands must always be implemented in the following way: The external module will set the command TRUE. The module receiving the command will then reset the command to FALSE and start executing the command. In that way, only rising edge on the command input will trigger execution.

Level sensitive commands and status output are not timing critical. For example, the McAxisBasic functionblock has a Power input, which powers on the axis. This input is level sensitive, and not sensitive to timing.

gModuleNameParType: (optional) Contains any parameters that can be given to the module, from other modules. If the module needs parameters for executing the commands, these can be placed in this structure.

gModuleNameConfigType: Contains configuration parameters for the local module.

gModuleNameRecipeType: Same as above but for recipe parameters and registered with the recipe service.

global.var : Variable declaration file for global variables. If the module is indexed (an array of identical modules), the variables will be arrays.

Name	Type	Constant	Retain	Replicable	Value
glnfeedInterface	glnfeed_if_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	

gModuleNameInterface: Instance of *gModuleNameIfType*

gModuleNameConfig/Recipe: Instance of *gModuleNameConfigType/RecipeType*

These variable are retained, meaning that data is preserved through a warm restart of the target (which also includes power cycling the target)

The reason these are individual variables, and not just part of *gModuleNameIfType*, is that it is not possible to make just part of a structure retained – only a complete structure can be set up as retained.

Config and recipe are global to avoid extra mapping through global interface from one module to another.

These variables are automatically registered with the config/recipe service, which can store the values in .xml/.csv files on a file device (USB or internal flash, see brdkFile).

local.st: Init, Cyclic and Exit parts of the task for the module. All predefined PackML modes are called here, and references for dynamic variables are configured. Code that must be executed in all modes, should be located here. Function block calls are typically done here.

production.st: Implementation of the production mode of the equipment module. This is typically where the majority of the module's functionality is implemented.

maintenance.st: Implementation of the maintenance mode of the equipment module. If the module has a maintenance or cleaning routine, then it will be implemented here. The mode is optional, and if not needed, the mode and file can be deleted.

manual.st: Implementation of the manual mode. Manual functions of the module can be implemented here. This mode is also optional.

alarm.st: Implementation of the alarm handling using BrdkAlarmControl function block.

local.typ Definition of all local types.

Name	Type	& Reference	Replicable	Value	De
hmi_typ			<input checked="" type="checkbox"/>		
example	BOOL	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
local_typ			<input checked="" type="checkbox"/>		
em	Equi...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
hw	local...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
configName	STR...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
recipeName	STR...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
MpRecipeRegPar_Config	Mp...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
MpRecipeRegPar_Recipe	Mp...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
hmi	nmi...	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
local_hw_typ			<input checked="" type="checkbox"/>		
ai_example	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
config_typ			<input checked="" type="checkbox"/>		
example	REAL	<input type="checkbox"/>	<input checked="" type="checkbox"/>		
recipe_typ			<input checked="" type="checkbox"/>		
example	DINT	<input type="checkbox"/>	<input checked="" type="checkbox"/>		

Figure 5- local.typ file

localType: Contains all local variables, function blocks and sub-structures. **em** and **cm** function blocks are pre-declared here, as well as instances of the below types

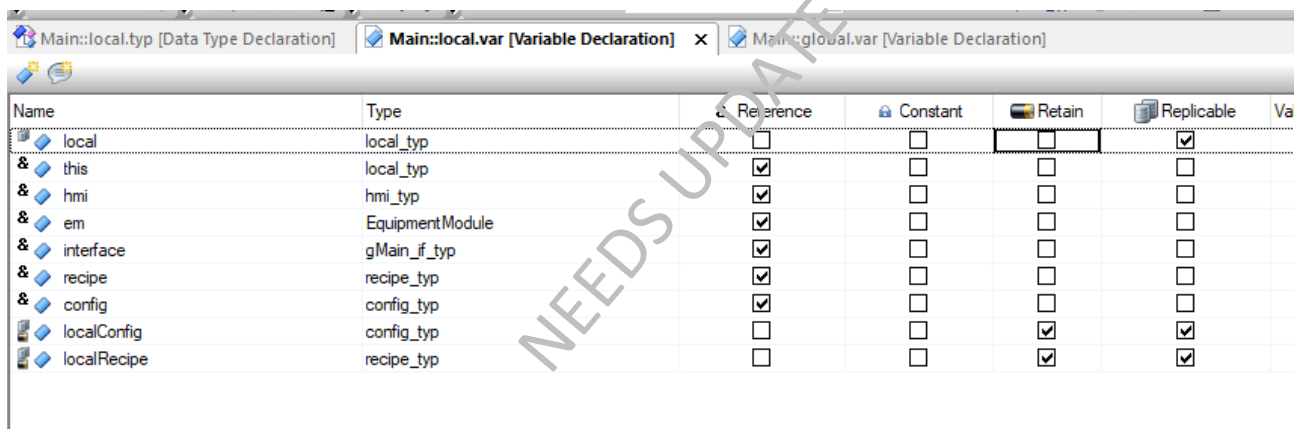
hmiType: Contains all variables that have bindings to the hmi. All variables that are connected to the HMI **must** be part of an hmiType structure. Only exceptions are the recipe, config, 2-way bindings, hw and global interface.status structures, which may also be connected directly the the hmi.

hwType: Contains all io mapped variables for the local module.

alarmType: Contains the MpAlarmXcore associated with the module and an array of BrdAlarmControl function blocks. Increase the size of the array if more alarms are needed.

local.var

Declaration of local variables.



Name	Type	Reference	Constant	Retain	Replicable	Va
local	local_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
& this	local_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
& hmi	hmi_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
& em	EquipmentModule	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
& interface	gMain_if_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
& recipe	recipe_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
& config	config_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
localConfig	config_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
localRecipe	recipe_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Figure 6 - local.var file

local: Instance of localType. If the module is indexed (an array of identical modules), then local will be an array of localType.

this: Dynamic reference to localType. If the module is indexed, then *this* will always access the memory location of the current index of the *local* structure. When programming, always use *this*. *local* should only ever be used in very specific cases, i.e if needing to scan through all the instances of the module in a FOR loop.

hmi: Dynamic reference to *local.hmi*. This gives easier access to the hmi structure of the module. When accessing hmi variables in the code, always use the *hmi* reference.

em: Dynamic reference to *local.em* function block.

interface: Dynamic reference to the global interface of the module (gModuleNameInterface). If the module is indexed, then **interface** will access the current index of the gModuleNameInterface array.

recipe: Dynamic reference *gModuleNameRecipe*. If the module is indexed, then *recipe* will always access the current index of *gModuleNameRecipe*.

config: Dynamic reference to *gModuleNameConfig*. If the module is indexed, then *config* will always access the current index of *gModuleNameConfig*.

ModuleNameAlarms.tmx

This file contains texts for all alarms related to the module. The texts can be stored in all relevant languages. Actual implementation of alarms is covered later in this document.

3. Naming conventions

Choosing meaningful names for variables and data types is a key factor in creating clear program code.

All names should be descriptive and easy to read.

3.1 Variable naming

To increase readability, variables start with small letters and capital letters are used when combining words (called camel-case). For example:

```
actPressure := actForce / pistonArea;
commandCount := commandCount + 1;
```

The only exception to this rule is counter variables. Individual letters can be used for these, for example:
i, j, and k

Variable names can include both letters and numbers, but they must start with a letter. Reserved keywords are not allowed.

3.2 Structure / type naming

User data types start with a small letter and capital letters are used when combining words. All data types ends with Type. For example:

recipe_typ	
name	STRING[80]
ingidien1	USINT
activateLight	BOOL

A substructure / sub type is named with first the name of the first structure followed by NewName and ends with Type. For example:

recipe_typ	
name	STRING[80]
ingidien1	USINT
activateLight	BOOL
time	recipe_time_typ
recipe_time_typ	
hour	USINT
min	USINT
sec	USINT

3.3 Constants naming

Constants are written in CAPITAL LETTERS. Underscore characters are used to more clearly separate words written together (SCREAMING_SNAKE_CASE). For example:

MAX_MEM	USINT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	50
CONSTANTS	USINT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	10

3.4 Global structures and variables naming

Structures and variables that are global starts with the prefix “g” followed by a capital letter.

An exception is the variables in a global structure, they are not started with a small g because the structure already is defined so. For example:

gRecipe_typ			<input checked="" type="checkbox"/>	
name	STRING[80]	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
ingredient	USINT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
activateLight	BOOL	<input type="checkbox"/>	<input checked="" type="checkbox"/>	

Structure.

gStatus	USINT
---------	-------

Variable.

3.5 Pointer variable naming

All variables that are used as a pointer, starts with the prefix “p” followed by a capital letter and a describing name of to what the pointer is for. For example:

 pDevice UDINT






3.6 IO variable naming

Variables assigned to hardware inputs or outputs start with a prefix that identifies the type of I/O point followed by VariableName.

Prefix	Type
di	Digital input
do	Digital output
ai	Analog input
ao	Analog output
at	Temperature input
si	Safety input

Table: Overview of prefixes and types

For example:

 di_sensorStop	BOOL
 do_lightTower	BOOL
 ai_pressureSensor	INT
 ao_valvePosition	INT
 at_tankTemp	INT

As an alternative, it is allowed to make substructures inside the hwType structure, for each type of IO. E.g. all digital input variables for this module, combined together in a hwDiType substructure. This also negates the need for each variable to have the “di” prefix, as the substructure name already holds this information.

3.7 Enumerator naming

All enumerators are named with enumerationName and ends on Enum

The constants in one must be written with CAPITAL LETTERS, and start with a reference to the enumerator.

For example:

Name	Type	& Reference	Replicable	Value	Description [1]
SHIFT_REG_STATUS_ENUM					Enumerations for shift register status
SHIFT_REG_STATUS_VALID				0	Shift register is valid
SHIFT_REG_STATUS_SHIFTING				1	Shifting is underway
SHIFT_REG_STATUS_SHIFT_DONE				2	Shifting is complete
SHIFT_REG_STATUS_UPDATING_JIGS				3	Jig positions are being assigned

Figure 7- Enumerations with assigned values

All enumeration values should have their constant values explicitly assigned, as shown in Figure 7.

3.8 Function block naming

An instance of a function block, must always be named so that the first part of the instance name, is the same as the function block type. For example, an instance of a **TON** function block could be called **TON_delay**, making it easy to see directly the datatype, when debugging. It should never be named something like **delayTimer**, as it is then not immediately possible to determine that this is an instance of the **TON** functionblock.

An instance of a functionblock could also be named ending in “_0”, like TON_0. If declaring an array of functionblocks, the number should be omitted, as the index of the array would then substitute the

number. Example: TON_[4].

4. Descriptions in .var and .typ files

All entries in .typ and .var files should have a description, as seen in Figure 8. This will provide the function, that doing a “mouse over” of the variables in the code editor, will show this description, adding to the readability of the code.

Name	Type	& Reference	Constant	Retain	Replicable	Val...	Description [1]
local	local_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>		Local structure instance
this	local_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		Reference to local structure instance
hmi	hmi_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		Reference to HMI structure
em	EquipmentModule	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		Reference to em function block
interface	gRoundtable_if_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		Reference to global interface of this module
recipe	recipe_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		Reference to recipe structure for this module
config	config_typ	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		Reference to config structure for this module
localConfig	config_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Local config instance. Multiple instances, if indexed module
localRecipe	recipe_typ	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Local recipe instance. Multiple instances, if indexed module

Figure 8- Descriptions in a .var file

If variables or structure members have a physical engineering unit, then that engineering unit should be provided in the description as [unit] as seen in Figure 9.

hmi_typ			<input checked="" type="checkbox"/>			
boilerTemp	REAL	<input type="checkbox"/>	<input checked="" type="checkbox"/>			Temperature of medium boiler tank [°C]
homingOffset	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>			Servo motor homing offset [mm]

Figure 9 - Engineering units in descriptions

5. Alarm system

The alarm system is implemented using the MpAlarmX system, in the same hierarchy as is used for the equipment modules in the framework.

Each module will have its own MpAlarmX configuration (.mpalarmxcore), in the Configuration View, as well as a Mpcomgroup configuration, which will define the hierarchy of the alarm system, as seen in Figure 10:

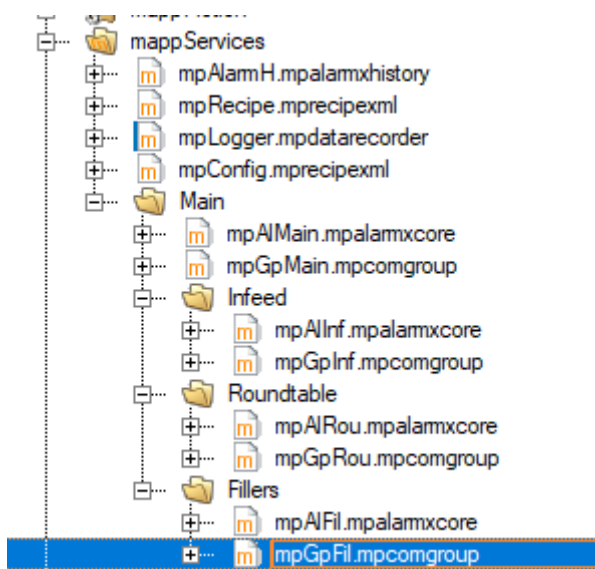


Figure 10- mpAlarmX and mpComGroup configurations for each module

The contents of the mpComGroup component is basically all the children objects of the current module:

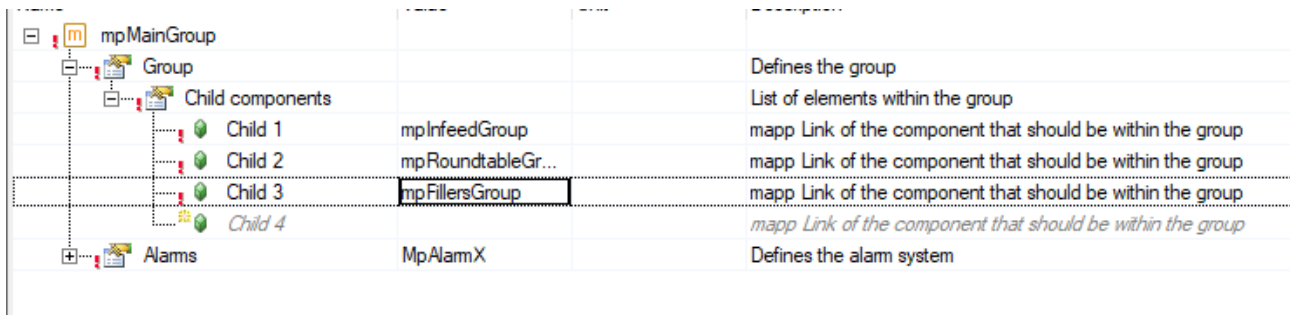


Figure 11 - mapp hierarchy structure for the main module

This means, that the **Main** mpComGroup can now collect alarms from its children modules, provided that the alarms are escalated from the children to their parent.

If mappAxis is used in the project, remember to include these in the module group as well. If needed, the axis alarms can be aggregated, so that only a generic axis error alarm from that axis will be shown on the higher level alarm system.

mappAlarmX config – Alarm list and mappings

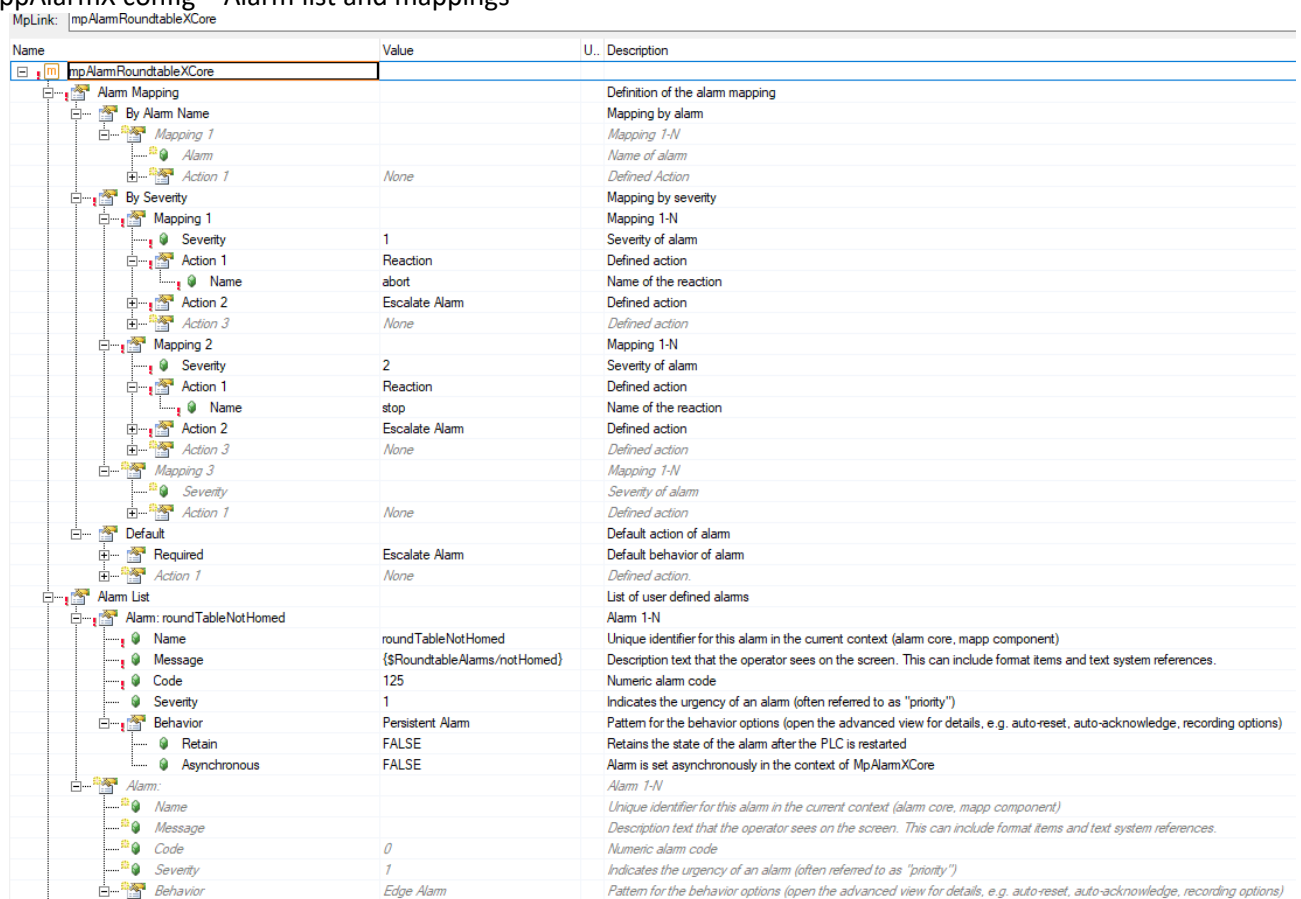


Figure 12 - mappAlarmX config example for a module

One mappAlarmX config object exists for each equipment module. In the “Alarm Mapping” section, the alarms must be mapped according to severity:

- **Severity 1:** Triggers an abort reaction. Take note that the alarm must also be escalated to the parent level, to display the alarm in the HMI.

- **Severity 2:** Triggers a stop reaction.

Default mapping is “Escalate alarm”, to ensure that alarms with all other severities are still escalated.

Actual alarms are then configured in the Alarm List section.

- **Alarm name** – Unique alarm name identifier
- **Message** – Actual alarm text to be shown in HMI. Note that in Figure 12, *\$Roundtable* is the namespace of the text, and *notHomed* is the TextID. These reference to the TMX file for the module alarms, mentioned earlier in this document.
- **Code** – Unique ID for the alarm, to be shown in the HMI. Reserve ranges for each equipment module, according to this list:
 - **Main module:** 1000-1099
 - **Next em** in hierarchy: 1100-1199
 - **For indexed modules** – There are no individual codes for each instance of an indexed module. Only a text snippet in the message will inform the operator which instance of the module that threw the alarm.
- **Severity** – This will determine the machine reaction to the alarm, according to the mappings set up previously.

The cyclic code for alarm handling for the Roundtable module is seen in Figure 13.

BrdkAlarmControl Function block is setup and will cyclically check for any roundtable alarms, with the reactions “abort” or “stop”, and will trigger the commands locally on the module.

The PackML hierarchy will then handle if and how the reactions should be escalated, or if just the local module itself should abort/stop.

The alarm itself is set or reset within the function block, based on the input ‘Condition’.

The input ‘Inhibit’ can be used to mute the alarm, e.g. to avoid safety alarms during boot when safety is not up and running yet.

```

ACTION alarm:

    // Lock alarm
    this.alarm.BrdkAlarmControl_[0].Enable := TRUE;
    this.alarm.BrdkAlarmControl_[0].MpLink := ADR(mpAlarmRoundtableXCore);
    this.alarm.BrdkAlarmControl_[0].Name := 'roundtableNotHomed';
    this.alarm.BrdkAlarmControl_[0].Condition := NOT interface.status.homingOk;
    this.alarm.BrdkAlarmControl_[0].Delay := t#0s;
    this.alarm.BrdkAlarmControl_[0].Inhibit := FALSE;
    this.alarm.BrdkAlarmControl_[0]();

    // Alarm handling function block for local module
    this.alarm.MpAlarmXCore_0(MpLink := ADR(mpAlarmRoundtableXCore), Enable := TRUE);

    // Check for local alarm reactions (check after MpAlarmXCore to have reactions updated)
    IF MpAlarmXCheckReaction(mpAlarmRoundtableXCore, 'abort') THEN
        em.Command.Abort := TRUE;
    END_IF

    IF MpAlarmXCheckReaction(mpAlarmRoundtableXCore, 'stop') THEN
        em.Command.Stop := TRUE;
    END_IF

END_ACTION

```

Figure 13- Cyclic alarm code for an equipment module

6. General coding guidelines

-EDGEPOS/EDGENEG – It is not allowed to use these functions to detect rising or falling edge conditions. The reason is that this will implicitly declare one bool variable, used to store the bool value from last scan. But when using indexed modules, this will of course not work. Always instead declare “old” variables for detecting rising/falling edge.

-Function block calling – Function blocks should always be called at the end of the scan. And they should be called cyclically, outside any state machines, except for very special use cases (i.e. a timer, that needs to be counting only while in a specific state).

Use always function block syntax with inputs set in separate lines, to the actual function block call:

```

this.TON_0.PT := config.delayTime;
this.TON_0();

```

Figure 14 - Function block parameter input and call

Avoid using single line syntax, where parameters are set internally in the function block call, as this is harder to read.

-Using function blocks with execute input

Functions blocks with execute inputs (or inputs working in an execute-style manner) must be handled in the following way:

To ensure that the function block always sees a rising edge condition on the execute input, the input must be reset as soon as the function block is done. Typically, as soon as the function block “Done” output goes high, the signal must be evaluated, and the input reset.

If there is a possibility that the machine can be aborted or stopped, while the function block is active, it is essential to ensure that the execute input is also reset in the corresponding PackML states (Aborting, Clearing, Stopping).

As a further safeguard, the execute input can also be set FALSE in the substate, preceding the one where the function block is to be used.

7. Handling non-indexed code in indexed modules

When implementing indexed equipment modules in an array, there is often the need to have some parameters, variables and code, which are not indexed. This has historically caused problems, because our implementation of indexed modules is done specifically to support the indexed modules themselves.

The problems are typically:

- Code will need to run in local.st, because all the PackML modes are running inside the FOR loop. This looks messy and makes the local.st file harder to read.
- Config and recipe parameters cannot be placed inside the local structures, since those are indexed. So often they will be placed in the config and recipe structures of the parent module, often Main. And will then need to be broadcast to the equipment modules through the global interface.status structure.
- Variables cannot be declared inside the local “this” structure, since this is an array. This has often led to just a bunch of local variables, outside of a structure being declared in the local.var file.

The solution

To combat these problems, the following standard for handling non-indexed variables have been implemented:

- Separate action file, .var and .typ files for non-indexed use is added to the local scope of the indexed equipment modules:

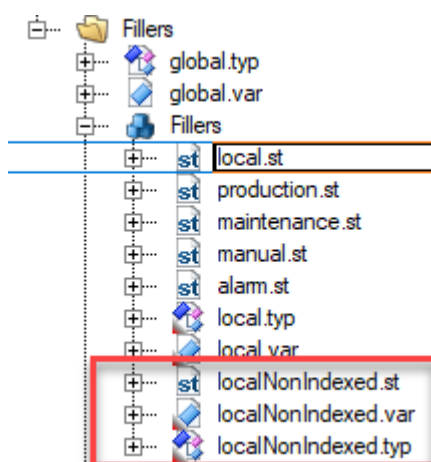


Figure 15- Non-indexed addition to indexed equipment module

- Local config and recipe structures for non-indexed variables. These structures will then be registered to the global recipe and config handling system and saved in the same way as the local config/recipe structures. Since they are declared in the local scope of the equipment module, they can be accessed directly in the code, eliminating the need for mapping through the global interface of another module.

Name	Type	& Reference	Replicable	Value	Description [1]
configNonIdx_typ			<input checked="" type="checkbox"/>		
firstPu	INT	<input type="checkbox"/>	<input checked="" type="checkbox"/>		index of the first print unit configured

Figure 16 - Local non-indexed .typ file

- Action file is called in the local.st cyclic code, before entering the FOR loop:

```

PROGRAM _CYCLIC

    localNonIndexed; //Run non indexed code

    FOR i := 0 TO NUM_Fillers DO

        this ACCESS ADR(local[i]);
        em ACCESS ADR(local[i].em);
        hmi ACCESS ADR(local[i].hmi);
        interface ACCESS ADR(gFillersInterface[i]);
        config ACCESS ADR(localConfig[i]);
        recipe ACCESS ADR(localRecipe[i]);

        CASE em.ModeID OF
            MODE_PRODUCTION: production;
            MODE_MAINTENANCE: maintenance;
            MODE_MANUAL: manual;
            ELSE em.Command.StateComplete := TRUE;
        END_CASE

        em();
        alarm;
    
```

Figure 17 - Calling the non-indexed action

