(left empty side before page 1)
Note: Most of Browser pdf presentations does not support the book mode.
Remove this page if you want to have a PDF file in book mode.

OFB – Object oriented Function Block Graphic
–
using LibreOffice draw
–
Basics and Handling

Dr. Hartmut Schorrig
www.vishia.org 2025-11-09

Table of Contents

(empty)

1 Open/Libre Office for Graphical programming

One of the ***advantages of textual programming*** is: You can visit your program code with any desired editor, such as Notepad++, or VIM on Linux or just a powerful *Integrated Development Environment*. For development of course, compiler tool suites are necessary. But to discuss content, behavior, look whats happen you need only standard tools. For long time maintenance it means it may be sufficient only to have the source code itself, if maintenance actions cannot be done only by parameterize (with given *Operation and Monitoring* tools). For updating the program, you need beside this sources only the compilation tools. Whereby often it's also possible to use newer versions of compilation tools which are compatible.

If you ***use graphical programming***, then the graphical sources can be viewed often only with the original tools which may be vendor specific, need licenses to use etc. Sometimes older source files cannot be opened with newer (currently in use) versions of the tools. It means only for view what is contained in your device, you need a specific tool. Additional often code changes are sophisticated in the tool chain, needs specific knowledge (about set options etc.). If the tool used some years ago is no more current in use, and the people are in pension, it is a problem.

This may be one reason that textual programming is preferred, though for the graphical programming it was rumored also for more as 30 years, it would be replace completely the textual programming.

That's why graphical programming is the playground for some big tool providers, whereas different approaches are given with the tools which are not compatible. Whereas textual programming is also familiar for common software, sometimes Open Source.

The ***second reason to favor textual programming is: The sources are immediately comparable with simple text diff tools.*** And the third reason is: Tools are interchangeable, the source is always understandable as text source.

Now, to favor the graphical programming, this paper offers the idea and shows approaches related with usable software for content evaluation to ***use a common graphical draw tool*** for the graphical programming, which is usable for everybody without effort, which is compatible also with some other tools and which is enough powerful to use. For that ***LibreOffice*** was tested to draw the diagrams, and a translator to evaluate the content was written (just in progress). This concept is presented here.

Some basic ideas are:

- Use Style Sheets to designate semantic information to graphical blocks,
- Evaluate it reading information from the odg file, it is a simple zip file containing XML information.
- Translate the content to other formats or just make immediately code generation.

A second approach of this work is: For graphical programming the familiar idea to use Function Block Diagrams (FBD) to present functional content are combined with important features of the UML class diagrams. All in all the Function Blocks (FBlocks) are seen as instances of classes, which is self evident often for code implementation (in C++) but also in C where Object Oriented classes can be implement with `struct` data and the appropriate operations for this data. It means the FBlock Diagrams are advanced with UML features of class diagrams.

And also, UML class diagrams (without the FBlock idea) can be drawn and translated also with this approach.

This graphical approach is near to the textual source code. Both are combined, the core sources are (should be) immediately textually.

The purported advantage of graphical programming, it would be save time and money because the coder engineers are not necessary, this is false. The important reason for graphical programming is: You have a presentation of functionality which can be discussed with your consumer, with anybody with physical knowledge. That's the benefit.

2 Harmonize FBlock Diagrams and UML-Class Diagrams

TODO longer explain FBlock etc. beginning with Ladder graphic.

The ***Unified Modeling Language*** (UML) was created in the beginning of the 1990th based on different existing modeling approaches, firstly by Grady Booch, Ivar Jacobson and James Rumbaugh wiki. Another contribution to UML comes from David Harel wiki who had development ***state machine technology*** firstly introduced with his own tool "*Statemate*" and then applied to the UML tool *Rhapsody* (original from I-Logix, now IBM).

The focus of UML was also code generation for particular devices, but also the approach of commonly describing of systems which can be applied to particular software, with focus of Object Orientation.

In opposite, the technology for ***Function Block Diagrams*** (FBD) inclusively code generation for particular usual firstly automation devices was created already in the 1960th with the IEC 61131 Norm for "*Programmable Logic Controllers*". It was also similar used for some other approaches such as LabVIEW wiki or simulation tools. Especially Simulink from Mathworks wiki is used here for some comparisons with the here shown technology. This tools has its basics in the 1980th but currently further developed and used.

Both approaches, the UML and the FBD tools are designated as "*model driven development*". But there are not related. The FBD tools does not use diagrams from the UML, and it is usual not seen as "Object Oriented" and the UML seems not accept a diagram kind which is firstly for a particular software or device and not for a commonly described system. One important difference is: FBD tools are always instance-oriented, each Function Block is an instance. Whereas UML is class- or system-oriented.

The code generation is usual familiar from the FBD tools. In UML, code generation generates only the frames of the classes respectively instances, it is not so frequently used.

The FBD tools focus only to the functional aspect of a device or a software. The operation system and managing to properly run the software (organization of threads, hardware access etc.) is usual done by specific settings (for example the "*hardware config*" part of configuration for automation devices with the Siemens TIA portal). The system itself is hard coded given and does not need an elaborately description presentation.

In opposite, the UML focuses to the whole system. For example the operation system itself is a "*component*", which is presented with interactions etc. in the component diagram. Also some hardware components.

In this manner the here presented combination of the UML Class and the FBlock diagram is only a part of a possible "UML 3.0". It does not replace all basics from UML, of course. It is only a contribution for this imagined UML 3.0.

How to name this combination of a FBlock and Class Diagram ... Let's use the abbreviation ***OFB***. The "*O*" stands for “ObjectOrientation” which is also near to the UML (*Unified Modeling Language*) . The diagram, graphical programming is named ***OFBgl*** with *“gl”* as *“graphic language”*. A textual representation of the same content should be named ***FBcL*** as *”Function Block connection Language”*. The focus to the UML is not presented in this abbreviation, but UML is familiar and recognizable.

What else: The ***event connection*** between FBlocks are also used here as important part. Events are familiar in UML for state machines. An Event connection is also used in FBlock Diagrams with the standard **IEC61499** for automation devices as a basically feature. Also in Simulink events are designated and used for "*triggered subsystems*" as well as for state machines. Events should be familiar in Object Orientation.

The presence of events in all diagrams simplifies state machine technology. State machines should also a part of OFB in a proper way (yet in development 2025-07).

empty

3 Approaches for the graphic, basic consideration

Table of Contents

This chapter shows how capabilities of ***Open-*** or ***LibreOffice*** are used to draw diagrams.

3.1 First explain using and elöemens of libreoffice

TODO denomination of LOffcDraw from where to get, page shape, connector, style, position in cm / mm

3.2 Question of sizes and grid snapping in diagram

Commercial tools for graphical programming have often not a proper answers to this question. Often sizes are able to scale in any kind, as the user want to have. Grid snapping is sometimes supported or not, and, sometimes sophisticated algorithm are implemented which avoids lines through blocks and make instead mad ways around all blocks. LibreOffice is here more friendly, it let the user decide about the connection path. This may be only a marginalia.

Let's think about font sizes and grid, requirements:

- In a usual document a proper font size is 9..11 pt, this document uses 9 pt but for A5 page format. A smaller font (7 pt, 6 pt) is not suitable for reading because of the recognizably of the words and their contexts, it is only for read the package leaflet of medical products.

- A diagram should have place in a document on a A4 or size-B page (~ 18 cm text width). It means the size of a proper view is **~18 * 10..12 cm**. Using a whole side in landscape orientation may have a size of 25 * 17 cm, but in landscape mode the document must be rotated only for this page, this is not suitable for reading a PDF document on the screen.

- A diagram has two tasks:

  a) Documentation

  b) Base for the software

For the approach b) the diagram may be well editable as a whole on a large screen, for example with resolution 2650 * 1200 pixel. To document this complex diagram it can be shown in landscape orientation in a document, or better: It should be reduced in size to fit on a normal page in portrait format. Details are then no longer legible, but important things and orientation should be shown in larger font. Then the overview can be explained and details can be shown as part from exact the same diagram in a higher resolution.

- A common and contradictory question for diagrams is: How comprehensive should it be. Should it contain only one block and some less aggregated ones? Or should it contain the whole truth of a module? The answer of this question depends on the available size for presentation. There should not be to less content.

The UML has the advantage that you can use more as one class diagrams to explain the

same class in different contexts. That is a very great advantage and it should be usable also for some Function Block presentations! (Not yet in professional tools). This helps to decide how many content a diagram should contain.

- The readability of a word which is isolated of a sentence, an identifier of a block or line or such one is given also with a smaller font size than 11 pt, especially if it is present in bold font or maybe also in a non proportional font (as for programming language source code). Because in proportional fonts often important small characters such as “il” are to small and bad visible

- For positioning a proper grid size and the **possibility of positioning with cursor keys (!)** is essential. LibreOffice has the property that the step size for the cursor key is anytime 1 mm, independent of other settings. It's possible use cursor keys for fine positioning (Alt-Cursor...) but this is too fine.

There is a specific property of LibreOffice: The step width by moving with cursor keys is normally 1 mm. You can do fine adjusting in combination with the Alt-key, but this is too fine. If also a grid fine spacing with snap points of 1 mm is selected (a 5 mm grid with 5 fine divisions), then the placing is very proper. All elements are placed in a 1 mm grid, the 1 mm is enough fine for details and enough raw to simple snap in the grid points.

From that, the idea comes to have a standard size of small elements of 2 mm. The mid point is also in 1 mm grid snapping raster. You can have a near distance of lines of 1 mm, well obviously.

To show enough content in a diagram you may use an A3 paper in landscape orientation. On a larger monitor (2560 or 3280 pixel width) it is editable in entire page mode. The diagram has a width of ~40 cm. 1 mm space is ~ 6 pixel on the screen.

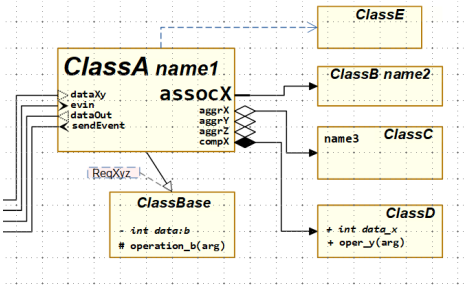


Figure 0: View A4-width as Part (280 DPI)

If you present the whole diagram in a document in portrait format, it is demagnified to ~ 17..18 cm, it means ~40%. As you see right side, the name of `ClassA` is readable, also the "`assocX`" with a font size of 10 pt Consolas bold in the original. Here it is presented with ~ 4 pt because of the demagnification. The others or not readable, but you can recognize the aggregations, compositions and associations. The symbols may be obviously though they have a size of only 0.8 mm height.

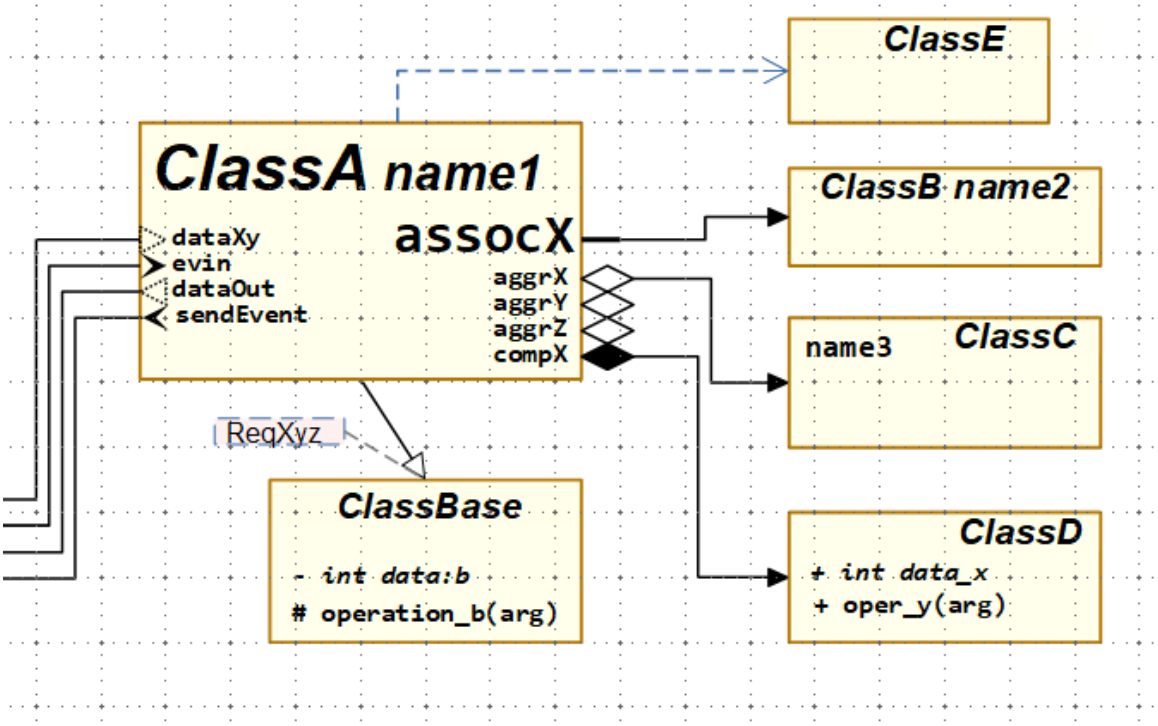


Figure 1: View in original size if this document is displayed with 2 pages on screen (112 DPI)

The same

content is presented here right side in original magnification. The font size of 6 pt for the most elements is just readable. It is Consolas bold. The type names of the classes are Arial 8 pt, the name of ClassA is Arial 14 pt. This is a 1:1 presentation, drawn in portrait A4 it is really 1/1 site width.

It means you can have an overview, but you don't see some details in the documentation. Parts of the same diagram can be shown in original size, then all is readable.

You should place different approaches of the same module in more as one diagram. This is definitely supported by UML, and should also be usable for function block presentations. In commercial tools such as Simulink it is not possible, but here it is.

As living example look on the following Class-Object-diagram:

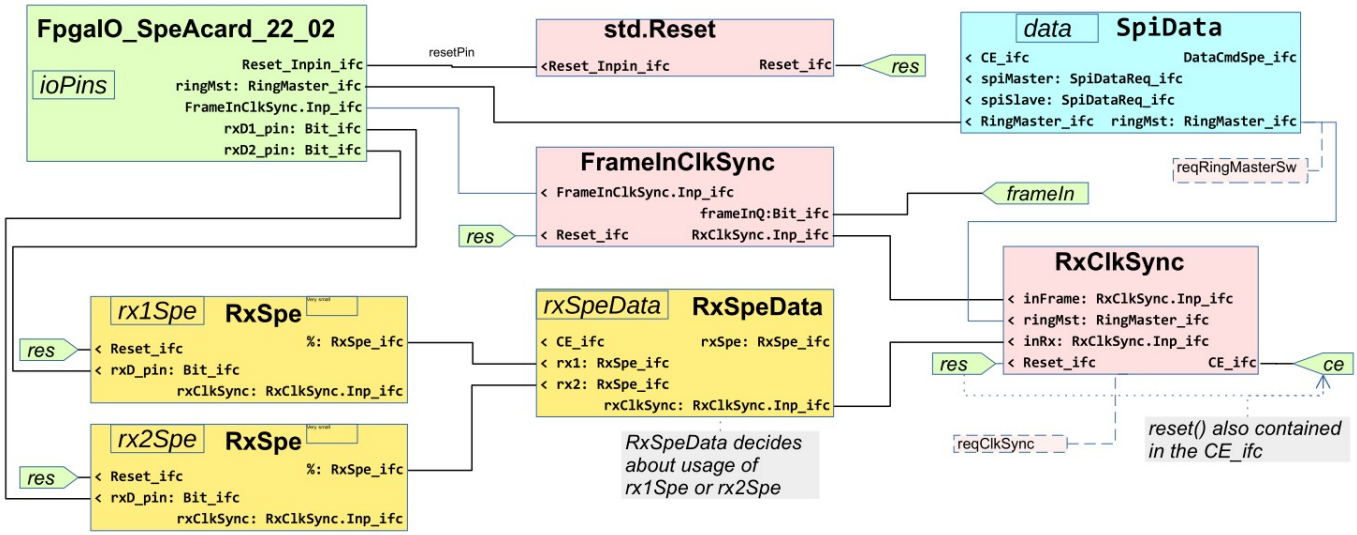


Figure 2: Example for a Module Diagram

This diagram should be well readable in normal view of a pdf viewer. The font and size of the

names is consolas 6 pt bold. The original draw area is the width of a A4 page. The pixel

solution is 1351 x 480, results from a Zoom of 200 % on a 1980 pixel width monitor.

The diagram shows a coherence of different blocks to build a synchronized *clock enable* (ce) in a FPGA. You see two receiver (Rx) modules, which are combined with a third module, with equal light-brown colors. Its a selection of the active input. The output of this third module has the same interface type `RxClkSync.Inp_ifc` as the module in the mid. Both are selected from the red right module. With less explanations the coherence should be understandable.

3.3 Using styles for graphic elements

The first used is a rectangle shape which presents a class or Function Block (FBlock). The rectangle should be marked with the style for indirect formatting `ofbClass` or also `ofbFBlock`. This formatting style results in a predefined appearance (color, line width, text font etc.). But not the appearance determines the kind of the shape, **the name of the style defines its semantic**.

With given indirect formatting style, you can modify the appearance with additional direct formatting, for example change the color of the shape. You can also define your own style. If this style starts with the identifier of the semantic defining style, followed by a “-” and then your own name, it works proper. This may be interesting for specific solutions, showing a special type of shapes only in appearance, which are all of the same kind.

For possible styles of FBlock shapes see *5.2 All Elements with their styles* on page *42*

From view of UML class diagrams:

A class or FBlock should have a name and a type designation. This can be written either as text in the FBlock (class) shape, as also in an extra shape `ofnClassObjName` for more free positioning. The text of the `ofbFBock` is positioned right top in the shape area. *Maybe press ctrl-M to remove other automatic formatting informations.*

The original UML class diagram has the following approach:

- A class is a rectangle box containing the type name of the class.
- Some data or operations may be named inside the class box, it does not need to be completely.
- All relations to other classes are shown with references to the other classes. This references are often non directed, but sometimes only in a specific direction marked with a little arrow on end. This relations are associations, aggregations, compositions, inheritance, dependencies.

The last point is not mapped to the languages which presents the software which is presented by the UML diagrams. Because: The fact that a class has an aggregation to any other class is a property of the class, and not a property of relations between the classes. It is exactly the same as for data. A data element has a type, and a reference has also a type, the type (or super/basic type) of the referenced class. The name and type of a reference is a property of the class, it is not a property of the relation between the classes.

For that reason the shown relations to other classes are assigned to the class itself. They are existing also if there is no connection. Then, of course in the implementation it's a null or nil pointer. Or it is just not shown here in this diagram, instead shown in another diagram, but nevertheless it is an element of the class. Look on the images on the page before. There are some not connected aggregations, which may have a meaning on explanation to the diagram.

The pin contains a text, which is the identifier for the pin and can also contain a type specification, a constant value or also a connection information. The text is written outside left or right from the small pin shape by using the LibreOffice property, that a text can exceed the bounds of the element's graphic. More as that, the left or right margin of the text is set to a value greater or equal the size of the element, and in this kind the text is written outside, left or right next to the element. If you want to have a little more distance, you can also insert spaces left or right of the text. The spaces are removed while evaluation of the text.

Why it is necessary in LibreOffice to set the “Left” value to the negative “Right” value, or also to a higher negative value, otherwise it does not work. It is not consequential. Second, In an older version of LibreOffice it was possible that the Distance value (here “Right”) can be greater than the size of the element, to insert a small space right of the shape. From Version ~6.4 this was no more possible, unfortunately. That should be small questions to the LibreOffice community.

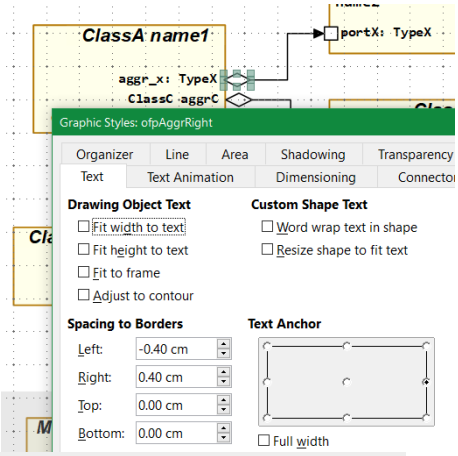


Figure 3: Style_ofpAggrRight_TextProp.png

The pin for connection to the class or FBlock is shown as this small shape or figure. However, it is not the shape itself that marks the shape as pin for code generation, the associated style sheet is the essential one. The look of the figure can be changed if desired, it is for human.

But **the style sheet marks the semantic of the figure, the kind of the element.** The settings in the style sheet, especially the size of the text, can be overridden by direct formatting. This is for larger fonts explained in the chapter before and shown in page . Also the settings in the style sheet can be changed for centralized approach. The name of the style sheet is the important one.

Style sheets are a proven concept for text writing. The direct formatting approach can be also used to a style sheet formatting approach, and both can be combined. A style sheet allows change a formatting style for all designated elements (paragraphs, parts of text etc.) to achieve a uniform presentation. It is an advantage that is often not enough known. That's for 3.3

Pinscommon explanations.

3.4 Pins

An input or output of a Function Block (FBlock) is named ***Pin of the FBlock*** in the UFBgl. Hence on the pins connections between the FBlocks are connected, using connectors in LibreOffice, see next chapter.

But some connections are connected also to the whole FBlock, for example as destination for an aggregation. But this builds also a pin in the internal data map.

The pins are either simple small figures with a fixed size, known from UML as the diamond (filled / non filled) for Composition and Aggregation, or adequate forms for events and data, or they are simple text fields. The pin appearance does not play any role for the interpretation and converting of the graphic, but may be proper for manual view. For interpretation the associated style (indirect formatting) is essential. The style determines the kind of the pin.

The first idea for UFBgl was, using a common pin style which is proper for appearance, and defining several styles for the connection kinds between pins (aggregation, composition, data or event flow etc). This idea comes, because the end point of connectors can define in a UML-conform and interesting way, not only with an arrow left or right. Then the connector style would determine the pin kind. But this idea is worse, because pins should be well defined also in non connected states, for example for association of event and data pins. They should show the capability of a FBlock. Hence it is better to have different styles which determine the kind of the pin. The connector style (see next chapter, and on page

Hence, the sometimes existing `ofRef…` or `ofc…` styles should not be used for content semantic, only for appearance. All connection styles (except a few special cases) are the same for functionality, only different in appearance.

For the pins the simplest variant is, have a text field with the associated style.

3.5 Connectors of LibreOffice for References between classe

The connectors as known from LibreOffice are the proper possibility to connect FBlocks or classes. The connection can be done between pins of the FBlock, or also from/to the FBlock itself.

You can use connectors with orthogonal lines, or straight or curve connectors as if you want.

LibreOffice assigns four connection points ("glue points") to each element by itself. This is sufficient for the pins. It is very simple to connect for example the end point of a diamond of an aggregation with the mid of a port as destination of the aggregation, or also with any other class if the whole class is referenced.

For the larger class block with maybe more connections on different positions you can add some more glue points.

Using connectors between elements in your graphic, the connection remains stable if you move some blocks. You may adjust the inflection points (more precise the mid points between inflection). Some commercial tools such as Simulink try to adjust connections between blocks by itself by sophisticated algorithm, which should avoid lines crossing blocks, and make instead mad ways around all blocks only to avoid crossing a free but reserved area for a name of a block. LibreOffice is here more friendly, it does nothing by itself, only move the connection as necessary, and let the user decide about the outfit of the connection path.

A connector as reference between blocks should have also a Style. If the connected elements are well dedicated by Style Sheets, you can use the `ofRef` style for all connectors. It produces a small arrow on the end, and a line width of 0.2 mm, no more.

But there is also a possibility using connectors as in UML. The connectors have especially the start arrow outfit as in UML necessary (diamond for aggregation). Then you can use for the connected elements the common style `ofPinLeft` or `ofPinRight` which does not specify the kind of the element. The connector specifies it. That is the originally approach of UML, also possible here (but not recommended). Both are supported by code generation.

3.6 Connect Points for more complex reference

LibreOffice seems to be have the disadvantage that additional inflection points on orthogonal connectors are not possible. Look on the example left side. The connection from `aggr2` to `port2` through ClassF is not nice.

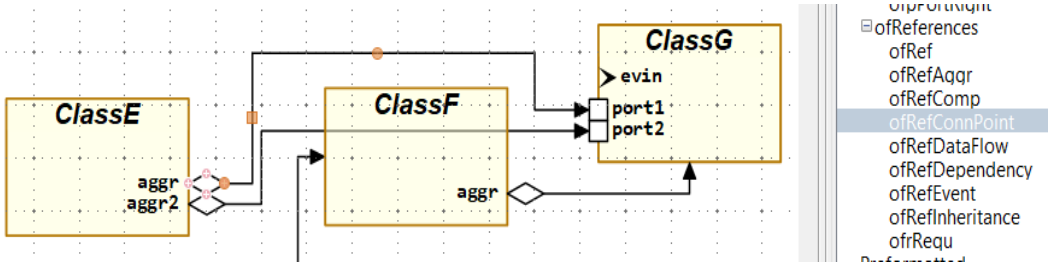


Figure 4: Figure 5:

The solution is shown also in this mage. From `aggr1` to `port1` two connection lines are concatenated. The first line is of type (style) `ofrConnPoint`, its without arrow on end. Both lines together appears as one line, with proper inflection points.

Another question is: Having aggregations or other references with one destination and more sources. In UML often there are drawn parallel. But it is more consequently to use a connection point as it is known from any electrical circuit scheme and also from Function Block Diagrams for data flow. The difference is only: Data flow and electrical schemes has one source and more destination. An aggregation has one destination and can have more sources. The reference line to the connection point is either a simple `ofRef` which has an arrow on its end, or it is the same as in the image above for concatenation of reference lines, with style or type `ofrConnPoint`.

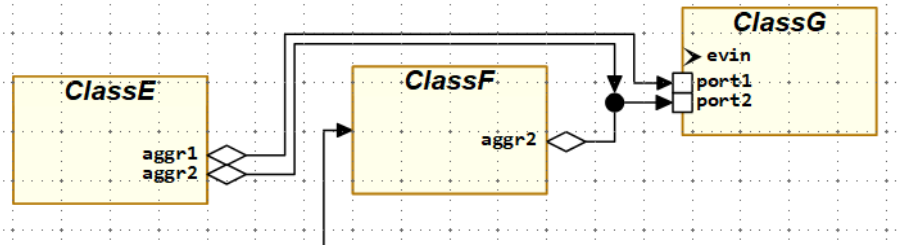


Figure 5: OFB/ConnPoint.png

3.7 Diagrams with cross reference Xref

The cross reference or usual nominated as Xref is an often used symbol to replace too much lines in one graphic, or also to make connections to several sheets of a graphic. The last one should not be in focus here, because on graphic sheet presents one aspect, spread one diagram over several sheets is not familiar for UML or also Function Block Diagrams.

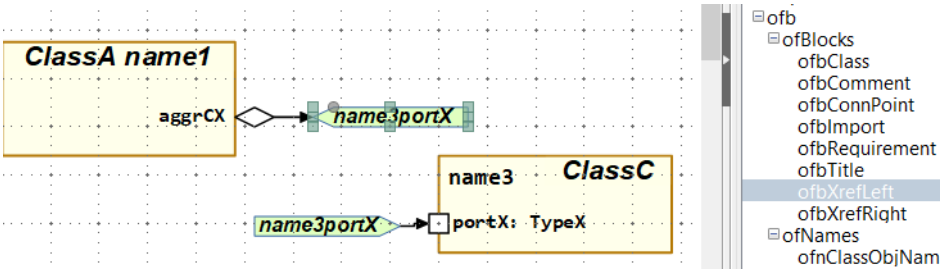


Figure 6: UMLdiagramXrefExample.png Cross Reference usage

You may use a Xref for signals and connections, which are well known from name, and which have basically connection meanings (such as “reset”) and may be connected to more as one block.

- The figure for the Xref can have any form, but should use the given form (copy it from template). The Style Sheet should be either **`ofbXrefLeft`** or **`ofbXrefRight`**, whereby the difference is only the text alignment to left or right.

- The name in the Xref symbol should be a mnemonic name for the functionality, valid for this diagram. Here it is a combination of the type of the port and part of name, maybe proper.

- The line from a block to the Xref should be the same type (here a simple **`ofRef`**) as without Xref.

- The line from the Xref to the block should have usual the same type, but this is not evaluated. Because the type of connection can be also composition or association here, the type for the association is used here, it is not specificated to the aggregation or composition with the filled or non filled diamond.

You can use Xref connections for all line types. The evaluation of the graphic builds a list for all Xref by name per sheet, and checks the connections.

3.8 Appearance of the GUI in LibreOffice draw

LibreOffice has the feature to customize the GUI. The standard offered icons are sometimes for another approach (drawing free graphics) and overloaded for the approach of Function Block diagrams.

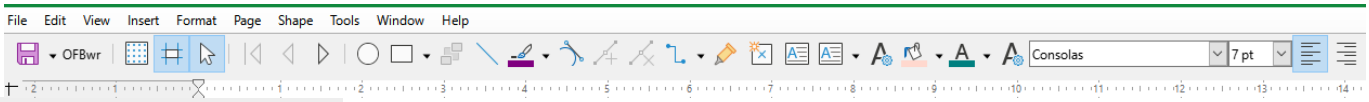


Figure 7: LOffc/IconsTop1.png

The image above shows an straightforward icon bar. It may be seen as important that the icon bar does not change its appearance during work. All fast accessible or state showing icons are given.

Functions which can be called also via the menu and are more rarely, should not waste space here. For example the disk to “save” is given here to see the red point in an unsafe state. Saving itself is faster done with <ctrl-S>, or just use the icon. Whereas *“save as”* can be found, if rarely necessary, in the *“File – save all”* menu.

LibreOffice allows to configure the icons on the tool bar by calling *“View - Toolbar - Customize”* from the menu, valid for all instances of the LibreOffice draw tool (also for the others of the suite)..

X

empty page

Left you see a button ***“OFBwr”***. This calls a macro which calls a batch file (Windows) or a shell script to force the translation of the before stored graphic. It means to translate after changes press *“Save”* (the disk) and then the immediately right side given *“OFBwr”*. The translation process needs only a few seconds, displayed on a command window, and can open after them a comparison (diff view tool) with the last generated sources to compare what is changed. Also a compilation and start of the executable can be done to see results, but this is controlled by the batch file / shell script and not in responsibility of the graphic tool.

All in all a fast work is possible.

4 Capabilities and concepts of OFB diagrams

Table of Contents

4.1 Modules and Files

In OFB, a ***Module*** represents a self-contained software unit. From an external perspective, it behaves as a black box with clearly defined **inputs and outputs** (its interface) and a specified functional behaviour. It should be independently testable. Internally, the implementation is described graphically.

A single LibreOffice Draw (.odg) file may contain:

- exactly one module,
- multiple modules, or
- only a part of a module, where the complete module is distributed across several graphic files.

During translation, each module is converted into one header file and one implementation file in the target language (e.g., C/C++). Conceptually, a module corresponds to the contents of a class in object-oriented design.

An .odg file typically contains multiple pages. A module has to be organised on consecutive pages.

If a module is distributed across multiple files, the translation order—defined by the Java command-line translator arguments has to be reflect this structure. The final page of one file has to be end with the partial module, and the following file have to continue it seamlessly.

Each page belonging to a module have to contain a shape with the style `ofbTitle` specifying the module name.

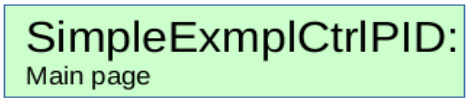


Figure 8: og/ofbTitle-1.png

Pages with identical module names are associated with the same module.

Modules can also be organised into reusable libraries.

4.1.1 Module as Library – Definition of FBtypes

An OFB module can be used exclusively to define some **FBtype** (Function Block Types) that serve as interfaces for other modules. In this case, the implementation of the dependent modules may exist entirely in the target language (e.g., handwritten C/C++ code). The FBtype definition has to be kept consistent with the corresponding target-language interface (header file), or the target-language files may be generated from the graphical model independently in advance. This approach supports separation of responsibilities and distribution of development effort and domain knowledge. The module interface, expressed as an **FBtype**, remains identical in all usage scenarios.

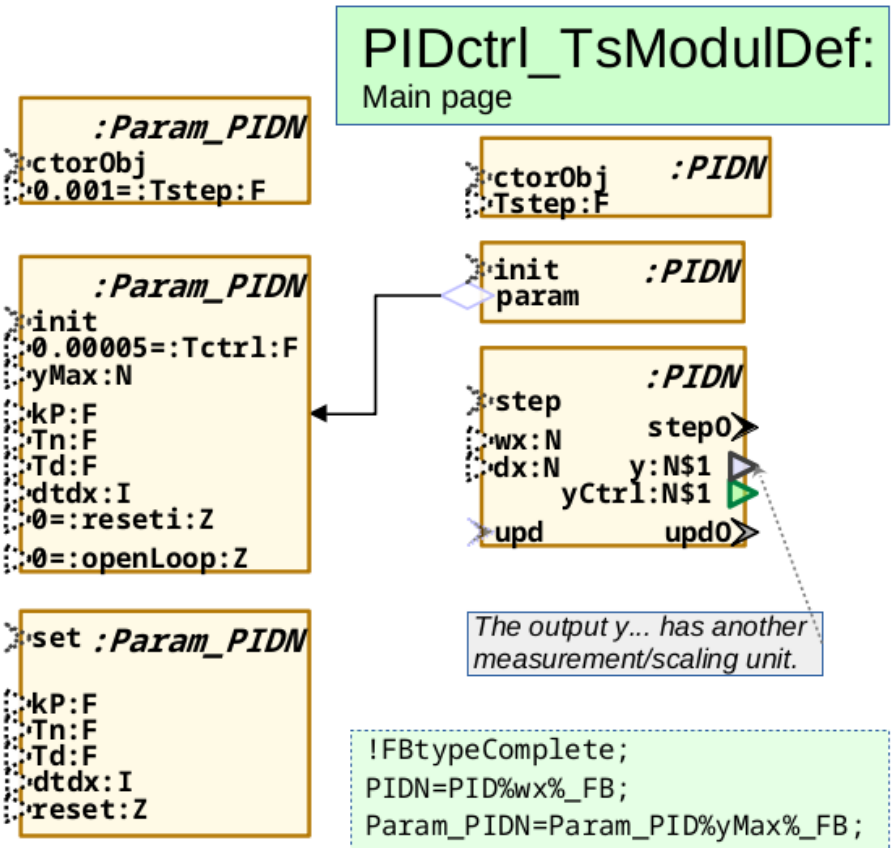


Figure 9: ExmplCtrl/LibCtrl_PID-FBtype.png

An example is the complete FBtype definition of a PID controller used in control engineering. A notable aspect is the definition uses non-fixed data types. The PID controller is available as implementation in C language with the same interface in `float`, `double`, `int16` and `int32` arithmetic. Here the used data type of the `yMax` pin defines the used name of the `Param_PIDN` FBtype, it is `Param_PIDF_FB` since the data type is `float` or `F` as one-character data type designation.

Each **Graphic Block** (**GBlock**) represents a single operation in the target language together with its associated data. **Input** pins correspond to function arguments, while **output** pins often are simple mapped to variables in a data `struct` (in C) or to public members of a C++ class. But also private encapsulation is supported.

From this graphical definition, the following prototype (forward declaration) is generated for the parameter-setting operation:

```
void set_Param_PIDF_FB(Param_PIDF_s* thiz,
float kP, float Tn, float Td, int32 dtdx, bool
reset);
```

Alternatively, an output pin may represent the return value of an operation or act as a “getter” for private data, or provide output arguments by reference.

Between the `PIDN` and the `Param_PIDN` an aggregation relation (as in UML) is given.

This module graphic is similar as a class Diagram in UML. The **FBtypes** are **classes in Object Oriented manner**, some properties of the classes are defined.

4.1.2 Module as Functionality - with Interface and Content

The following graphic illustrates a complete module. It represents a typical control application that uses the PID controller defined earlier as a library FBtype. Details regarding implementation are deliberately omitted at this point.

In particular, the handling of event inputs/outputs and event flow is described in *4.4.3 Event Determination from Data Flow (OFB Approach)* page *23* which even use this example and explains data flow, event synchronisation, and step timing, including the mechanisms required to ensure data consistency.

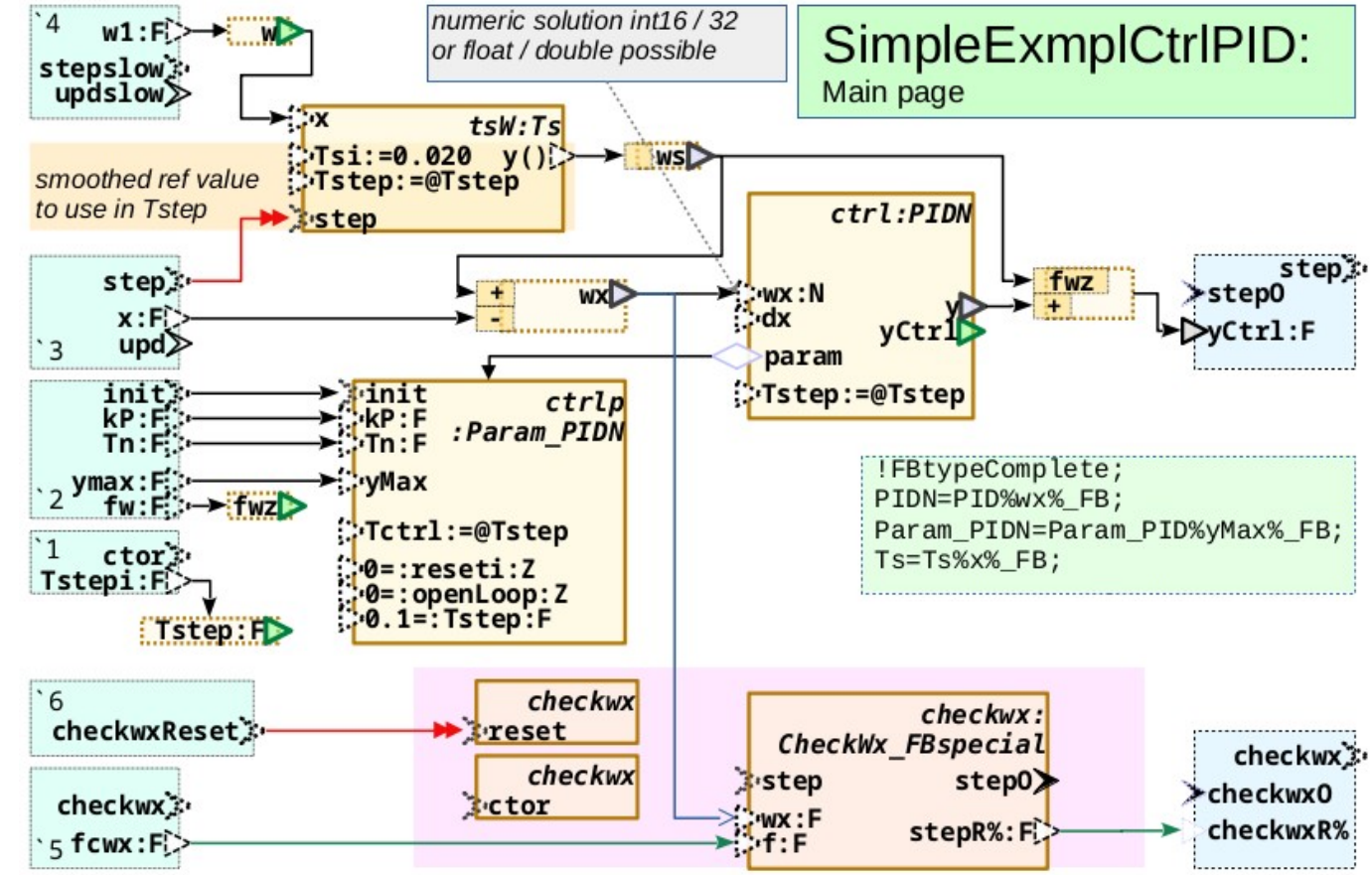


Figure 10: ExmplCtrl/SimpleExmplCtrlPID.png

Key structural aspects of a module:

- Each module contains a title box styled `ofbTitle`, as described previously.

- A module defines explicit inputs and outputs. These are arranged in cyan and light-blue boxes styled `ofbMdlInp` and `ofbMdlOut` (typically placed on the left and right side of the diagram).

Note on styles: Styles are used to define visual appearance (font, colour, etc.) Here the styles are used especially to encode semantic meaning. In OFB diagrams. Styles act as identifiers that allow the translator to interpret graphical elements.

- Module input/output pins are represented by triangular shapes (rectangles are also possible). The geometry is not relevant; however, the assigned style have to be one of the `ofp…` styles (e.g., `ofpDinRight`, `ofpDoutLeft`, `ofpEvinRight`, `ofpEvoutLeft` etc.)

- These pins define both the external interface of the module (black-box view / FBtype) as well as the internal connections to functional elements.

- A module contains **FBlock** and **FBexpr** elements representing functional operations and expressions. An FBlock is adequate an instance of a class (of a FBtype), or a composite relation (UML) from the module to this FBlock.

Each FBlock is identified by its name. At least one graphical occurrence have to denominate

the FBtype identifier. In this example the instance `ctrl` is of the FBtype alias `PIDN`. Once defined the same FBlock name may appear multiple times in the module to show it in different contexts (e.g., `checkwx`).

Since FBtype names can be long (due to namespaces or naming conventions), short aliases can be used. These mappings are defined in the green `ofbAlias` block, which associates abbreviated names with the fully qualified identifiers stored in the project repository. The alias mechanism also supports type-generic FBtypes.

For example, the placeholder `%wx%` in `PID%wx%_FB` is replaced by a one-character data-type identifier inferred from connected signals. If the propagated type is float, the resulting FBtype becomes `PIDF_FB`.

The mapping to the actual implementation data type identifier is defined separately in a target configuration file passed to the translator. This allows adaptation to platform-specific representations. In the example, the concrete C type is: `PIDf_Ctrl_emC`

4.1.3 Graphic Blocks, Pins, Texts and Connections

The ***semantic*** of the graphic (the meaning, which is presented) is not determined by the graphic form, instead it is determined by the used **style**. The style (even known as *"indirect formatting"*) determines the appearance, line thickness, colour, font etc. This is for a proper view. The intrinsic meaning of the style name is the semantic.

- All Function Blocks (FBlock, also FBtype) has the style `ofbFBlock`.
- An input data pin has the style `ofpDinLeft` or `ofpDinRight`, whereas `ofpDin` is the relevant part for semantic.

The diagram consists intrinsically of *Graphic Blocks* (***GBlocks***) styled with `ofb…` styles. GBlocks represent functional elements and should not overlap in grapic. A minimum spacing of 1 mm is necessary to ensure clear structural association.

Each one FBlock is built from possible more as one GBlock as functional representation of the GBlocks.

Pins are graphical elements associated with styles `ofp…` (or `ofPin`). Pins belong logically to a ***GBlock***. and are functional related to the correspond FBlock. The association between the graphical pins (GPins) and its GBlock is determined by position. At least one side of the pin have to lie within the GBlock boundary. Pins may slightly extend beyond the block to make connection points visible.

Additionally the **text content** to GBlocks and GPins has its semantic relevance. There are simple rules. The name is left, and the data type is on the right side of the colon. The FBlock with `ctrl:PIDN` has the instance name `ctrl` which is used in target language, whereas the type is the `PIDN` shown in the library FBtype definition at the page before, with its alias factual context.

There are more textual deterministic possible. The `:=` or even `=:` is a data assignment. `Tstep:=@Tstep` means, that the pin `Tstep` of this FBlock is connected with the `Tstep` variable of the module, even without a drawn connection. The connection can be defined by text.

`Tsi:=0.020` on the `tsW` FBlock means, the value `0.020f` is assigned to the pin respectively the argument `Tsi` The `f` is automatically completed on code generation for this `float` input. The FBtype definition (not shown here) determines the data type `:F` or float for this pin.

The designation `y()` in this FBlock means, the pin `y` is implemented by a getter operation, not only by a simple `struct` variable in C language.

Connections between GBlocks, typically between their pins, are drawn with LibreOffice ***Connectors***. Connectors remain attached to their endpoints and follow graphical repositioning, ensuring structural consistency.

Connections can be even designated with specific styles, for example `ofcAggr` for an aggregation (UML), completed with the non filled diamond.

4.2 Additional FBlock capabilities

4.2.1 Multiple View of the same FBlock in Graphic

OFB adopts an important principle known from UML class diagrams: A class (or functional element) may appear multiple times across one or more diagrams, each showing a different perspective. An FBlock in diagram is therefore a view, not the definition itself.

Traditional Function Block Diagrams, in contrast, typically treat each graphical block as a separate instance, often without an explicit identifier. UML relies on a central repository that stores all model data, while diagrams are merely graphical projections of that repository. OFB follows the same conceptual approach. Its internal repository is constructed from the total set of graphical sources read by the translator.

Key rules:

- FBlocks and FBtypes have unique names within a module.
- A GBlock may display only the instance name (without type) if at least one is dedicated also with its FBtype name.
- All GBlocks with the same name refer to the same logical FBlock. Their pins and properties are merged internally.
- An FBlock is module-scoped but may appear on multiple pages.
- An FBtype is project-wide and may be referenced across several modules.

Conceptually, each FBlock instance can contribute to its FBtype definition All occurrences of FBlocks with its given FBtype and FBtypes (GBlocks without instance name) contribute to the overall class definition. They represent structural properties or connectivity shown in different graphical views.. An FBtype definition is considered complete only when explicitly marked by `!FBtypeComplete;` in an `ofbAlias` block or when a defining module provides the full specification. This prevents accidental redefinition.

Unlike UML tools, the OFB repository is yet not interactively browsable within LibreOffice Draw. However, it exists internally after parsing and can be emitted as a report. This multi-view capability enables complex interconnections to be presented in a structured way:

- Different diagrams may focus on different signal groups or functional aspects.
- The same connection may appear in several views for clarity.
- Inputs, outputs, and processing paths can be separated across pages while still referring to the same uniquely identified elements.

Although distributing connectivity across several diagrams may initially seem unfamiliar, search and analysis tools allow developers to locate all occurrences of a given FBlock instance. The approach emphasises readability and separation of concerns.

4.2.2 Aggregations and Associations Between FBlocks

OFB supports UML like structural relations:

- Composition – a block owns another block. This is given with the FBlock which is places as member of a module.
- Aggregation – one block references another one.
- Association – looser structural reference (rarely used)
- Dependency – one block needs another one.

At runtime, these relations are implemented as typed references, for aggregations passed during the `init` operation. This provides a clean Object Oriented mapping in C using `struct` and operations, or adequate C++ constructs.

In traditional Function Block graphics sometimes addresses to memory are forwarded as Integer value data flow between FBlocks implemented in C language. This simple trick is not necessary in the OFB graphic.

4.2.3 Vectorisation and Sliced FBlocks

Individual signals can be combined into a *"cable harness."* In other FBlock tools, this is sometimes called as *"bus"* and sometimes as *"multiplexer"* which is in fact incorrect. Such a harness is shown in the right image with the signals `a`, `b`, `c` (here in another order, as test case). The output signal `yf` is feed by the input signal `xb`, dedicated as `c` in the harness.

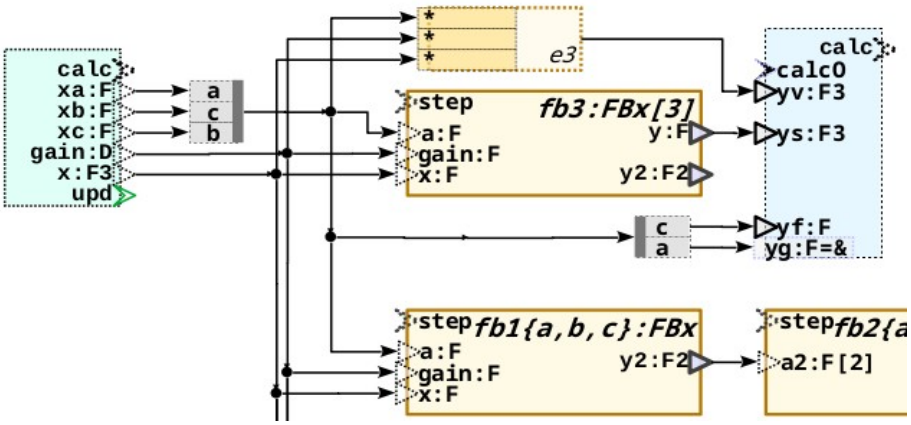


Figure 11: OFB/ArraySlideDemux_Vector_FBlock.png

Such a harness can be used immediately for vector operations, with the signal order in the harness top to down. The FBlock `f3` multiplies:

```
thiz->yv[0] = (xa * gain * x[0]);
thiz->yv[1] = (xb * gain * x[1]);
thiz->yv[2] = (xc * gain * x[2]);
```

FBlocks can be both vectorised as here shown for `fb3`, as well as the signal order in the harness determines the vector.

```
step_FBx(&thiz->fb3[0], xa, gain, x[0]);
step_FBx(&thiz->fb3[1], xb, gain, x[1]);
step_FBx(&thiz->fb3[2], xc, gain, x[2]);
```

As they can also drawn as sliced FBlocks with different names with the same FBtype, shown here for `fb1a`, `fb1b` and `fb1c`. Vector connections as here for `x` are distributed to the correct sliced FBlock in order. But the sliced parts of names follows the signal identification in the harness:

```
step_FBx(&thiz->fb1a, xa,(float)(gain), x[0]);
step_FBx(&thiz->fb1a2,xb,(float)(gain), x[2]);
step_FBx(&thiz->fb1b, xc,(float)(gain), x[1]);
```

4.2.4 Built Variants in Modules

It is an important challenge to deal with variants of the software. One approach is, having only one source which is held in a repository. It should be able to built different executables from this source, which contains optional functionalities. In C language this is done e.g. by conditional compilation (`#ifdef`).

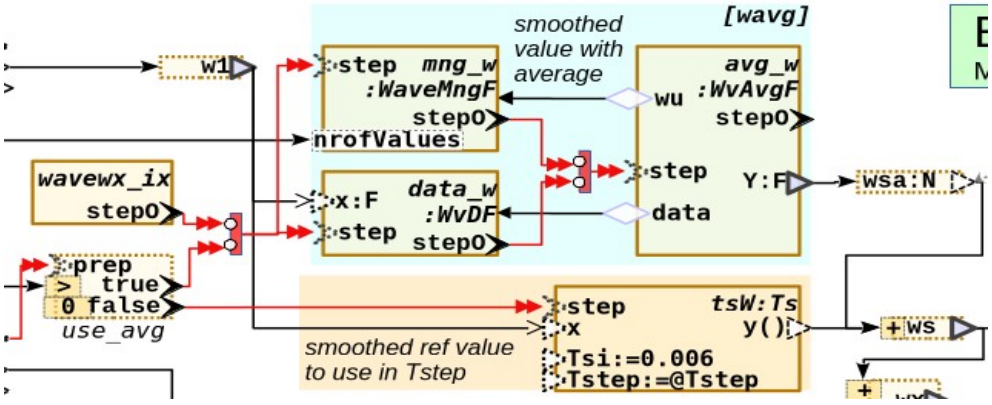


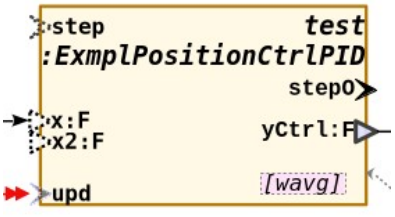
Figure 12: ExmplCtrl/VariantSelPIDctrlAvg.png

The source is the LibreOffice Draw graphic with OFB. The picture above shows a part of the comprehensive example of the last page. Here the average calculation in the `ofbFBarea` box is marked with the option identifier `[avg]` to select this part of the module for target code generation.

On use of the module a specific pin of style `ofpOption` can be used to mark this option. This is illustrated in the picture right side on bottom in the FBlock which represents the module.

In this example a conditional execution is intended. Alternatively an connection can be marked even with the identification `[avg]` to select it.

Figure 13: ofpOtion-avg-on-PICctrl.png



This image shows the selection of the optional content of the used module.

4.3 Graphical Diagram translation to target code

The OFB toolchain converts a graphical model into executable artifacts (target code) through a multi level translation process.

The *“daily user”* process to start translation can be immediately integrated in the LibreOffice Draw tool. With a button to press, see right side. In LibreOffice this button invokes a macro, which calls a batch process to execute translation. The arguments and the scripts to translate should be prepared only one time for the project. The translation uses Java.

It is even possible to start this translation batch process as command in a batch script.

The translation process can read more then one graphical inputs (odg files) or inputs from other supported sources.

The translation separates following steps:

- read the graphical representation,
- semantic interpretation (*FBcl*),
- target-specific code generation.

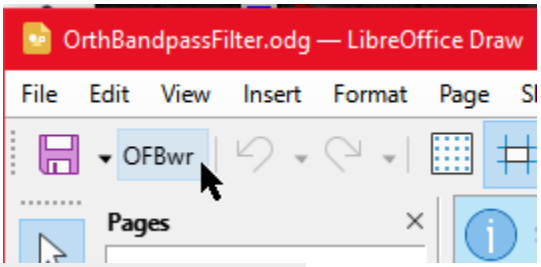


Figure 14: LOffc/IconsTop2.png

In this one-pass translation, the prepared semantic data are only existing temporary, reported in log files. It is planned to offer a translation control environment (an IDE, Integrated Development Environment). This uses the data from different sources and present it in a *“Repository Tree”,* which is familiar in UML tools. Then it is possible to view the semantic data as a whole, work in different graphic files, translate the only one changed graphic source file, start target code generation as necessary for different platforms, debug, work circular.

4.3.1 Read the Graphic Structure

The source diagram is read directly from the internal XML representation (`content.xml`) of a `LibreOffice.odg` file. This data are first mapped to internal Java structures representing the drawing model, e.g.:

- `>>OdgPage` – page-level container
- `>>OdgShape` – generic graphical object
- `>>OdgGBlock` – semantic representation of a graphic functional block (GBlock)
- >>`OdgGpin`. – representation of a pin

These classes mirror the graphical layout while already attaching OFB specific semantics derived from styles, not from the geometry. The referenced classes are documented in the internal Javadoc and are relevant only for developers extending the translator.

The read graphic data can be written out in a textual form (`*.fbg` file) which contains all graphic information. This files are comparable as simple text, with possible textual diffing and version tracking.

4.3.2 Save as Functional Model (FBcl) - IEC61499 compatible

In the next step, the graphic-oriented data are transformed into a functional representation, e.g.:

- `>>FBlock_FBcl`
- `>>FBtype_FBcl`
- `>>Pin_FBcl`
- >>`PinType_FBcl`

This abstraction layer removes purely graphical aspects and produces a structural model describing function blocks, connections, and execution relations. The resulting representation is saved and can read using a textual format called **FBcl** (***Function Block Connection Language***). The syntax of FBcl uses the IEC 61499 standard (*.fbd) with small OFB specific extensions encoded in comment sections.

Rationale for IEC 61499 usage:

- Provides a standardised, tool-independent structure
- Enables textual diffing and version tracking
- Allows import/export with other IEC 61499 compatible tools
- Acts as a stable intermediate representation between diagram and code

After all modules from one or more sources (.odg, .fbd) are read, the model is complete and ready for code generation.

From this data in a later version a *"Repository Tree"* should be shown and can be managed, e.g. to exclude modules or add new sources, re-read only certain modules, and finally start the code generation.

4.3.3 Template-Based Code Generation

Code generation is intentionally template-driven rather than hardcoded. Generation rules are written in a simple interpretive template language >>`OutTextPreparer` called ***gTxt***. This allows experienced users to adapt output to:

- Different C/C++ coding standards
- Platform-specific runtime frameworks
- Alternative target languages
- Project-specific architectural constraints

A gTxt script contains the pure text for code with place holder, e.g. in the following form (shortened):

```
##
## update a z-Variable
## @arg fb it is a FBlock with type VarZ_OFB
## @dType:DType_FBcl data type of the dout
##
<:gTxt: VarZ_OFB_upd: evSrc, fb, evin, din, d
<:if:dType.sizeArray && !dType.isScalarSizeAr
  memcpy(thiz-><&fb.name()>, thiz-><&fb.name(
<:else>
  thiz-><&fb.name()> = thiz-><&fb.name()>_d;
<.if><: >
<.gTxt>
```

In this script it is determined that `memcpy` should be used, but only for arrays, not for structured types. Last one are correct compiled with the assign operation. The operations in `<&...>` and for conditions `<:if:...>`, `<:for:...>` uses given value per identifier from the internal Java data, and can access elements and operations via Java code (using Reflection) from these internal FBcl data.

The generated part looks like

```
  thiz->ws = thiz->ws_d;
```

whereby using `thiz` for the internal data pointer (C language) and built variable names from the FBlock names etc. are determined by the script. It can be simple changed without access to the sources (Java) for the OFB translator.

4.3.4 Work flow till test on target hardware

The called scripts either with the shown `[OFBwr]` button in LibreOffice, or started via command line (or from the later planned *Repository Tree* tool) can include a complete build (make) till the executable for the target. An executable running on PC with graphic output, supported by socket communication and a ***CurveView*** graphical tool is immediately supported.

Alternatively an IDE for target compilation is recommended for detail tests even in Debug mode. Changes can be done quickly on graphic level, new translation, compile and run. The times for translation from graphic are in the same range as compilation times (less than 1 second for simple modules, a few seconds for more complex ones). With a *"Repository Tree"* tool, it is possible to dedicated translation the changed graphic sources, and hence do not re compile the whole project.

The entire development can even be done very well as a combination of graphical programming and handwritten code parts for dedicated FBtypes. Of course generated code should not be changed again on implementation level.

4.4 Event-Based Execution

4.4.1 Motivation

Traditional Function Block environments execute FBlocks cyclically using fixed sample times. This approach is common in PLC (Programmable logic controllers) and simulation systems with code generation, as well as in similarly text programmed embedded control systems.

The event driven approach is becoming increasingly important due to distributed hardware with its communication requirements and also due to the need for more complex functionality. This step was carried out by automation control software with a new IEC 61499 standard (https://iec61499.com/), which has the potential to replace the long- established IEC 61131 PLC-Standard in Simatic S7 ® and Codesys ®.

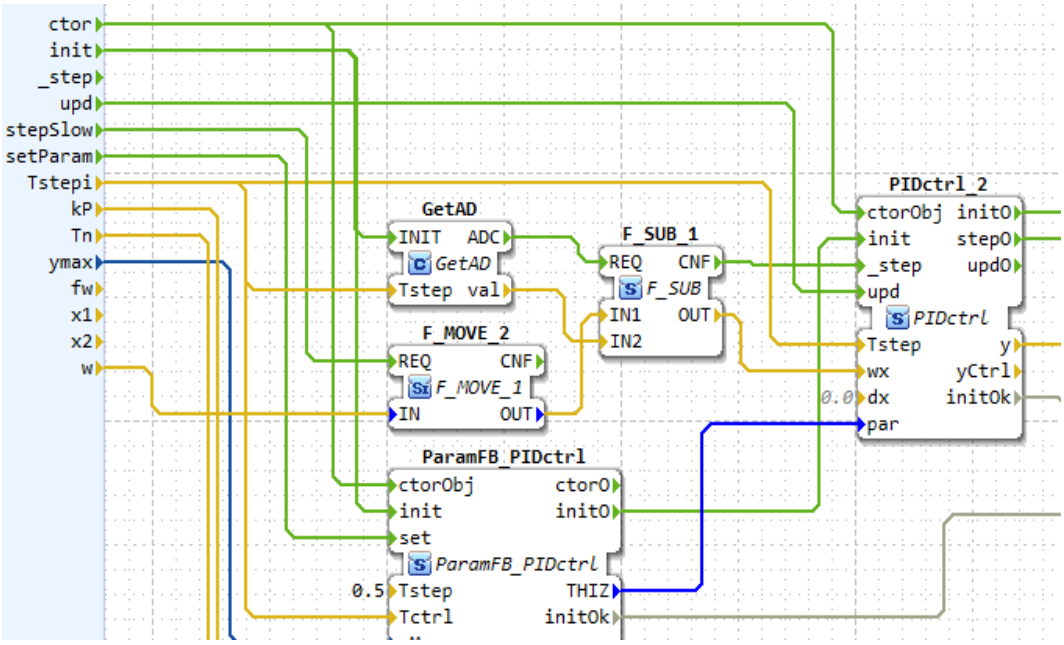


Figure 15: ExmplCtrl/4diac_ExmplGetADCtrl.png

Right side there is an example with the tool 4diac (https://eclipse.dev/4diac/4diac_ide/). The maybe interesting point is: The FBlock `GetAD` emits the ADC event, the Analogue-to-Digital Converter, used by the PID controller. This FBlock can be localised on any other device, then this event arrives via communication, e.g. via field bus. It is also possible that the event is created by hardware, from a specific driver with timer and analogue input.

OFB adopts an event-driven execution model, conceptually aligned with IEC 61499:

- Execution is triggered by events rather than implicit cycle order.
- Events may still be generated cyclically (e.g., by timers).
- The model naturally supports distribution across threads or devices.

Events therefore generalise cyclic execution rather than replacing it.

4.4.2 Advantages of Event Orientation

Event-driven FBlock execution enables:

- Explicit execution dependencies derived from data flow
- Easy distribution across CPUs or field-bus connected devices
- Reduced need for manually defined scheduling
- Clear representation of cause–and–effect relationships.

Communication between distributed blocks becomes an implementation detail handled during code generation rather than manually configured.

4.4.3 Event Determination from Data Flow (OFB Approach)

Unlike many IEC 61499 tools, OFB **does not require explicit drawing of all event connections**. In most cases: Data flow implicitly defines event flow. The translator analyses connections and derives execution order automatically. Explicit event wiring is only required when:

- No direct data dependency exists
- A specific synchronisation have to enforced
- Blocks reside on different execution domains

This significantly reduces diagram complexity without losing determinism.

This figure above is repeated from begin of this main chapter. It shows a PID controller. Here the step event comes from outside.

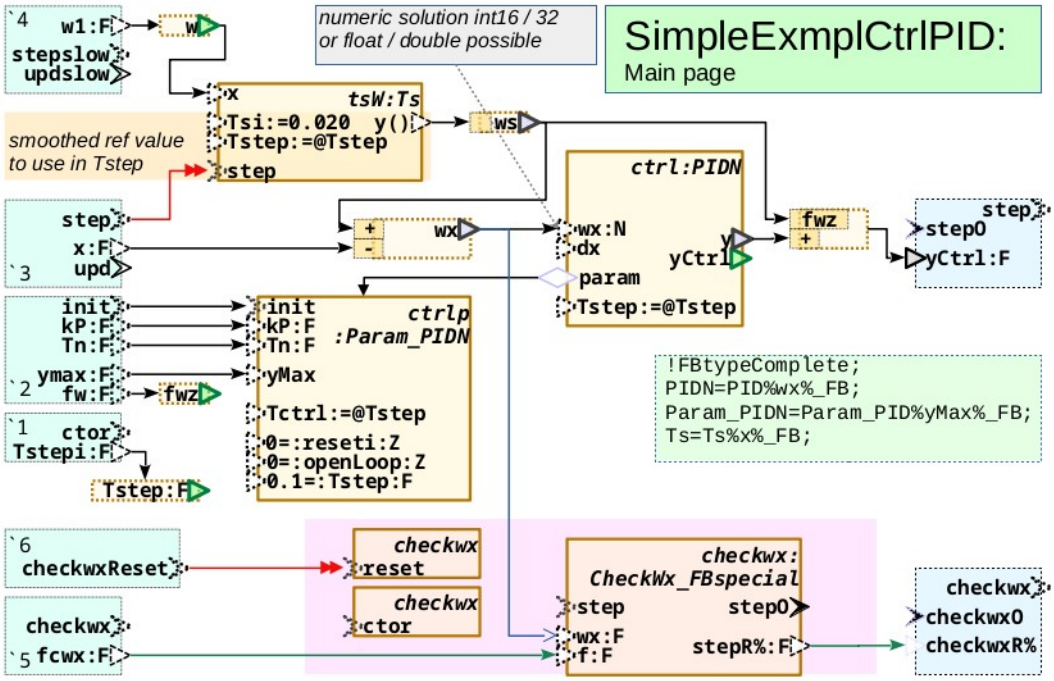


Figure 16: ExmplCtrl/SimpleExmplCtrlPID.png-2

Only the step event to a Smoothing filter `tsW` has to be drawn manually. The `PIDN` has even a `step` event input, not drawn here, but given in the library definition. This event is connected due to the data flow from the `prepO` event of the subtract FBlock before, etc.

4.4.4 Sequential Target Code Generation

Use of events suggest one or more event queues, which stores the events from different sources, and assures execution in one thread in correct order. This is the simple case.

OFB generate efficient sequential code when all blocks reside in the same execution context, instead of queue management between each FBlock. **This is essential for fast execution**, for example with sampling times of 300 µs (motion control), 50 µs (electrical control), or 20 µs (audio signals), which can still be processed by mid-range processors such as Infineon AURIX or Texas Instruments C28000 series. One event on module input with the following chain of execution is represented by one operation. The events are given here only for the superior handling.

4.4.5 Conditional Execution via Events

Conditional logic (equivalent to if / switch) is modelled by conditional event propagation rather than signal switching. This aligns control-flow semantics with execution semantics and simplifies mapping to structured target code. Here the outputs from `a1` and `a2` goes to the same input, select by the event flow, resulting in `if` statements in target code.

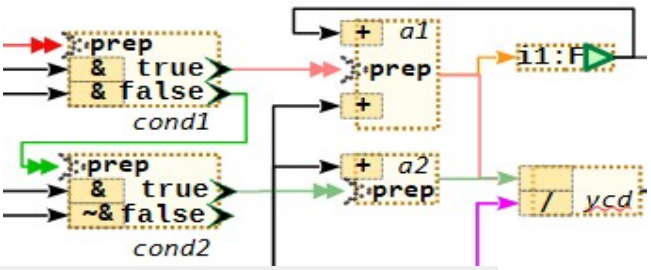


Figure 17: OFB/ExmpTrueFalseSimple.png

4.5 Capabilities of state diagrams / StateMachines - UML compatible

Thinking in states for algorithm in Embedded Control is essential. A state is also presented by given numeric values. But states in binary logic plays a specific role. For simple thinking the binary value true or false presents the state, changed with `if` and boolean logic. But thinking in **graphical StateMachine** or **State-chart** presentation is a more systematically approach.

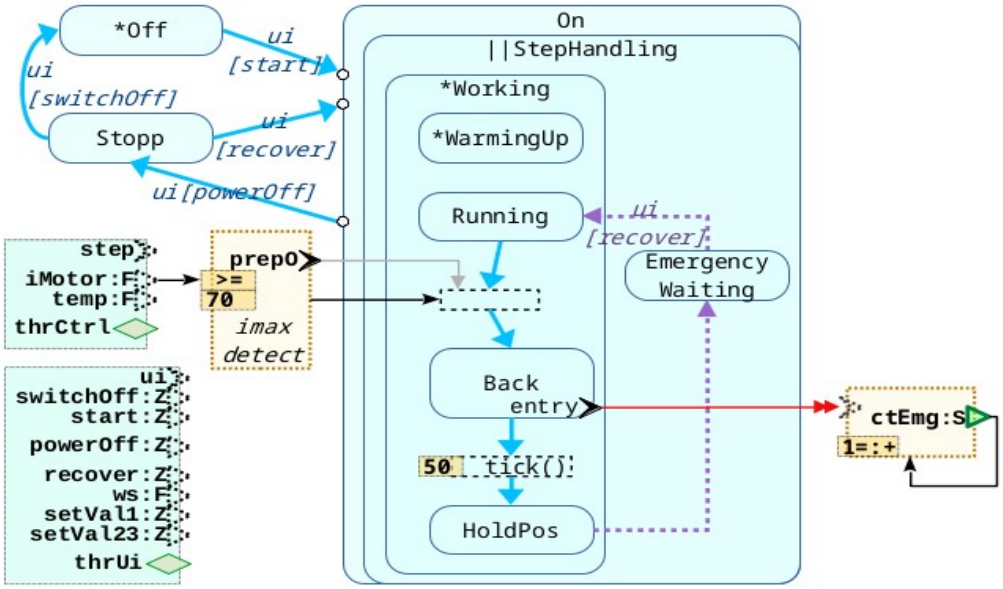


Figure 18: ExmplCtrl/Stm1

StateMachines have a long tradition, starting with the work of the known pioneer of digitisation Alan Turing (1912 - 1954), available in relay logic (FlipFlop) and some diagrams with rounding vertices and transitions. ... State diagrams are strong and well defined in UML, especially by the work of https://en.wikipedia.org/wiki/ David_Harel in the 1980th. With the tool "Statemate" from the company I-Logix, that was introduced in UML with the tool "Rhapsody", today known as "IBM Rational Rhapsody". This kind of state chart presentation is standardised in omg.org.

The UML StateMachine approach supports nested stated, a "History" transition to restore the nested state on re-entry, and parallel states with Fork- and Join-Transitions. The StateMachine is usual event-driven, but even able to declare as only "condition driven". Whereby condition driven is similar as event driven, the only one event is the occurrence of the step time where the conditional driven State Machine is executed.

The so called "Petri-nets" https://en.wikipedia.org/wiki/Petri_net usual in the 1970th are similar as parallel States and has play a role in thinking.

In traditional Function Block presentations, StateMachines are sometimes presented with specific FBlocks beside other data processing FBlocks.

For example Simulink (R) Mathworks supports also State Machines in a UML adequate kind, but sometimes with some differences, not full omg.org compatible.

StateMachines in OFB

In the OFB graphic. a StateMachine is embedded inside other FBlocks of the module. A State is even an FBlock, a transition is a connection between the FBlocks. The execution principles and functionalities of this StateMachines are compatible with UML, but slightly different in graphic appearance.

The image above shows a snippet of the states of a position control, regarding a collision detected by overcurrent.

Dispersing of state execution

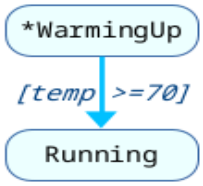
One essential capability is: Regions (parts of State switching) can be executed in different threads, or even on distributed devices. In the shown example, the position control is in the Composite State `Working`, its inner State switching is executed in a fast interrupt to recognise the overcurrent situation and react in less then one millisecond. This is a part of the event operation `step...()` in the thread `thrCtrl`, which is the interrupt. But in the `EmergencyWaiting` state, the motor is stopped and now the reaction of *"user interface"* `ui` events are done in a slower thread `thrUi` related to the event operation `ui...()`.

More devices can be arranged in one software architecture by a common StateMachine. For example one device can control the movement on a production line (conveyor) in one parallel (so called Orthogonal) Region. The superior

controller waits in a so called *Join* transition for readiness even from all other parts (*Orthogonal Regions*) in other devices, and then continues production in the next states. The field bus communication in a low cycle (e.g. 1 ms) assures knowledge of the state in the distributed devices and transports events as messages. Synchronise mechanisms are automatically inserted.

Repeated presentation of States in graphic

Figure 19: ExmplCtrl/Stm3



Even State FBlocks can be presented more than one time in the graphic of the module. Here, the State `WarmingUp` is shown in the left image only to express, that there is a warming up phase, and warming up is not necessary on recover, But the `WarmingUp` State is not shown there with transitions. This is done on another page, with specific conditions.

Additional possibilities for state drawing

Nested States do not need to be drawn inside its Super (Composite) State. This is important to support distribution of parts of Regions via some pages or areas in page. The same rules as for UML class diagrams are used here: A diagram shows only some aspects, and does not need to define the complete behaviour on one page.

The Regions (States inside a Composite State) are designated by the style of transition `ofcStateTrans` between, whereas Transitions between States of different Regions have to be designated with `ofcStateTransChgRegion`. In the image left side this are the transition to and from `EmergencyWaiting`.

The parent State can be designated even with only its name after `-`, as shown in the right side graphic for `RefValHandling-On`.

Parallel (Orthogonal) States are designated with the `||` symbol leading the name for the Super State of the Orthogonal Region. This is instead the dashed separating lines in UML.

The relationship from the parent or super State to its inner State is designated by the transition style `ofcStateTransParent`, as alternative to the nesting presentation in the diagram. Using this transition is even the so called *"Default Transition"* from the so called Initial PseudoState (UML) to the Default State. The Initial PseudoState, in UML a small dot with transition, is not drawn and not necessary here. The meaning of the transition of style `ofcStateTransParent` is the same. It may contain an action and a condition, but not an event to trigger.

If the default transition is not necessary (unconditional, without action), then the default state can be designated with a simple `*` before its name. In the image above the CompositeState SelSetVal23 has no Default State. The used State is defined by the two input transitions as `ofcStateTransChgRegion`. Both have no transitions between, but are affiliated in a common Region, with transition from its super or CompositeState `SelSetVal23` to `Setval1`.

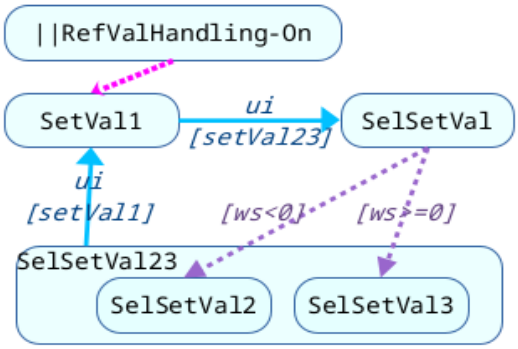


Figure 20: ExmplCtrl/Stm2

The code generation translates FBlocks presenting states in specific target language implementations (for C/C++ and other), which uses the presentation of states with bits in simple integer variables (as familiar for textual programming). Alternatively the code generation can also define `const` values for each State to determine it and possible actions and transitions using function pointer in C language or virtual operations in C++, and one pointer to the current State per Region. These are two different known patterns for implementation.

4.6 Comprehensive Example with Variants in Runtime

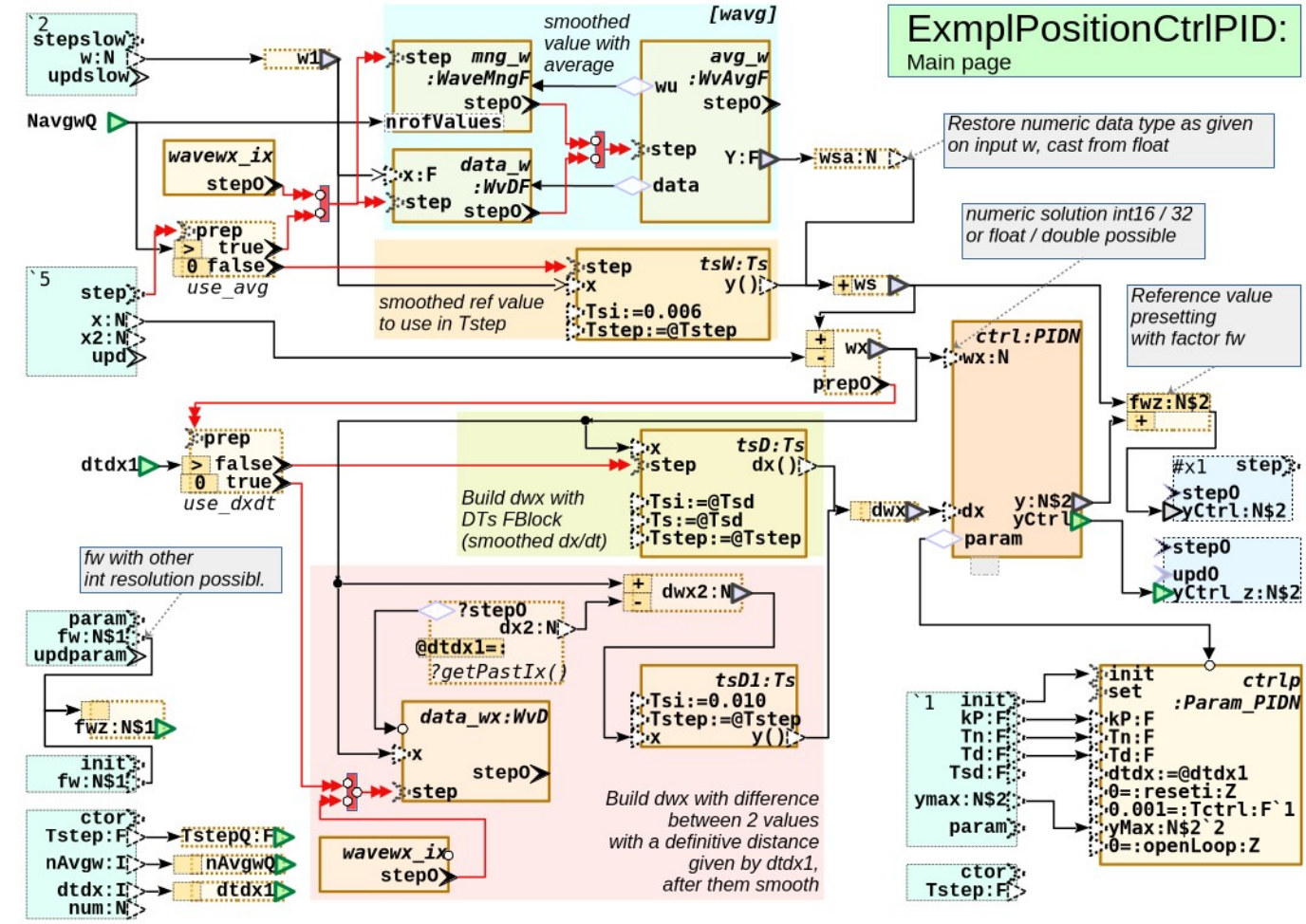


Figure 21: ExmplCtrl/Stm1

This graphic shows the important part of a PID controller solution contained in the example in the download zip file `OFB_Presentation/src/ExmplPositionCtrlPID/odg/ExmplPositionCtrlPID.odg`. See link on end.

It is drawn in A4 size. It is a half page in portrait format proper visible on a standard monitor (1980 x 1020). It is a little bit small but readable for documentation in a compact form. The size of the text on pins is 6 point.

This solution shows a few special features for PID control, drawn in OFB:

- The set value `w` comes with a slower sampling time `stepSlow`. It is stored in a local variable. There are two ways designed to smooth this value for a continuous process when changes occur. First a simple smoothing block explained already on chapter *4.4.3 Event Determination from Data Flow (OFB Approach)* page *23* is used.
- The second solution is an **average filter**. This results in a really continual value.

The decision which path is used depends on the value of the `nAvgw0`. If it is `==0`, then the event to execute the average filters is suppressed. It means no computing time is wasted on an unused value, adequate for the smoothing block.

The average filter is a solution contained in the so called ***emC*** algorithm collection (https://vishia.org/emc). It takes into account the correct calculation when specifying the mean value formation period with a fractional component. The last one is important if a signal should be filtered whose oscillation is not in integer proportion to sampling times.

The average FBlock accessed the circular data buffer `data_w` and the manager for the indices in this buffer `mng_w` via **aggregation**. Here **a classic data flow is not possible**. For this reason the event flow control is important and meaningful. The average can be built only after the other both aggregated FBlocks are calculated. To clarify, both are ready to use, the

small red Join event FBlock is used before `step` input of `avg_w`.

Classic FBlock tools have a problem because the feature of aggregation is not given. Either there is one comprehensive FBlock type which contains all, the buffer, manage and average. But then the generated target code cannot be optimised, if more as one averages should be built from the same data, or different data should be used for the same average period. In OFB this optimisation can be formulated by aggregating FBlocks properly. Some classic FBlock code generations may solve this disadvantage by automated optimising only in the generated code. But this leans towards obfuscating.

The second special feature is the **kind of building the D-part (*Derivative part)* of the controller** from the difference in time of the controller error signal . Because the D-part is a sensitive matter, it is build outside of the core controller algorithm. Here, too, two types of implementation are offered:

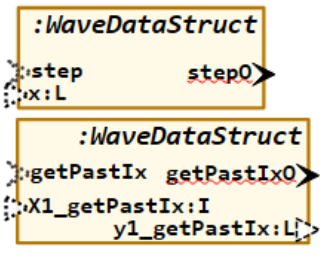
- Using a smoothing block for a DT1 D-part. This is a simple smoothed difference signal.
- Calculating a difference of the signal with a definitely period.

The last one suppressed oscillating parts in the measurement value, which would be reinforced by the D-part. Here the adequate **circular data buffer FBlock** is used to store the data of the last step times. The FBlock with the output `dx2` is an access operation to the `data_wx` FBlock, in Object Oriented sense `data_wx.getPastIx(dtdx1)`. In Object Oriented C language it is translated to

```
getPastIx_WaveDataStruct_Ctrl_emC(
       &thiz->data_wx, thiz->dtdx1, &dx2);
```

The output variable `dx2` is given per reference, due to the form of the legacy existing operation and its definition as FBtype. Last one is contained in the file `OFB_Presentation\src\LibOFB_emC\odg\LibCtrl_emC.odg`.

Figure 22: LibCtrl_emC/ WaveDataStruct-getPastIx.png

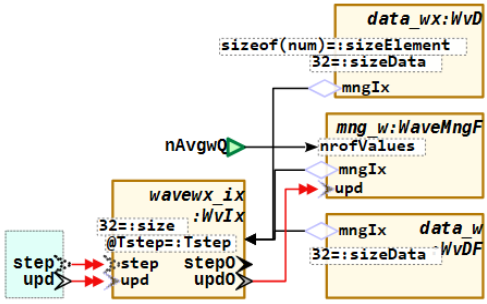


The access operation is defined by the specific input event `getpastIx` with its input and output data.

Since the buffer size is the same as for the set value, and both are used in the same thread, the same instance of the index builder `wave_ix`: is used for both. This saves calculation time and memory.

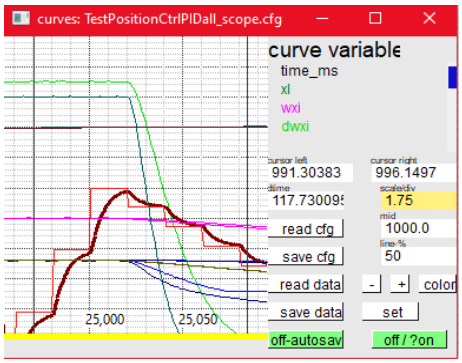
One detail is drawn with second view of this same FBlocks:

Figure 23: ExmplCtrl/ WaveMng-Aggr.png



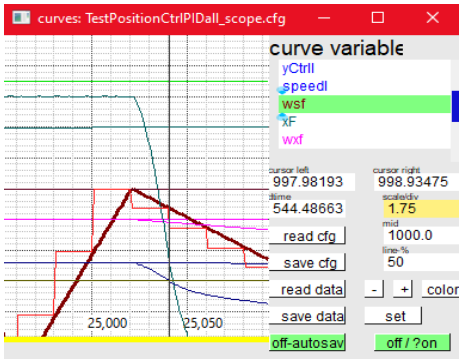
This is an overview which data buffers and average periods are used in this module and how they are aggregated and parametrised.

Figure 24: CurveView-w-Smooth-6ms.png



Right side is the resulting smoothed set value with `tsW`. The signal is rough, which can manifest itself in a rumbling noise in the servomotor. The set value input is the red line, stepwise in 20 ms.

Figure 25: CurveView-w-Average.png



This is the result using averaging. The difference is small, the graphic is zoomed. But such details may be important.

This example and some more examples and test environments are contained in a download zip file, described on the web page

https://vishia.org/fbg/html/Videos_OFB_VishiaDiagrams.html

mailto: hartmut.schorrig@vishia.de

5 Handling with OFB diagrams and LibreOffice draw

Table of Contents

empty

5.1 Usage OFB with LibreOffice, first steps

This is a kind of cookbook to get familiar with the OFB concept and LibreOffice.

5.1.1 Fundamentals on your PC and your knowledge

Of course you need LibreOffice. Which version is not so important. Also OpenOffice can be used.

The operation system Linux or Windows: The operation system Linux or Windows: The OFB converter and LibreOffice works on Windows as also on any Linux System. Mac is not tested, but should also be possible.

LibreOffice: You should be familiar with the handling of LibreOffice. Because it is a universal office tool, this is not a specialized knowledge.

Java: Your PC should have any Java installation. The OFB translator is implemented in Java. It is compiled and tested with the long term version Java-8, but also higher Java versions work. Java is familiar and used also for many other tools.

Compiler, IDE for the generated code: The examples on the `OFB_Presentation` uses an IDE (Integrated Development Environment) Visual Studio (Standard, yet not “Code”) and also CodeBlocks: https://www.codeblocks.org/. The last one is also available for Linux, it works with the known GNU compiler. Using for Visual Studio Code will be possible in the next versions. You should have installed one of this tools by your own. IDE-Projects for the examples are contained in the `OFB_Presentation.zip`. To run and view the results of the examples on PC a scope (Curve View) programmed in Java is part of the `OFB_Presentation`.

For the embedded control compilation you should be familiar with your requirements by your own.

Knowledge in Software for embedded control, compiler, UML and **Function Block graphic programming:** If you are attempt to use this concept, it should be presumed that you have knowledge of textual programming for embedded control. Usual this would be done in C or C++ target language. Knowledge of UML is not presumed, it is explain here so much as necessary, but UML should be more or less familiar.

The Function Block graphic programming paradigm may not be familiar, if you are oriented till now to target language, architecture, documentation and UML. But the FBlock programming is very familiar with some different tools also for automation programming as for such environments as Simulink. If you are not familiar, it is explained here.

Scripts, Compiler options etc: If you are familiar with embedded control in target language C or C++, you know what is an include path etc. You should also be familiar in a basic kind with scripts to control operation system execution command (batch files on Windows, Shell scripts in Linux). The scripts are explained and should be only necessary to change if you know what you want.

5.1.2 Start with the OFB_Presentation downloaded zip file

The zip file and its content is able to found on

https://vishia.org/fbg/html/Videos_OFB_VishiaDiagrams.html

There are explanations how to get it, how to get the OFB translator tool (it consists of two jar file loading from internet with check sum). This Web site contains also links to some videos with further explanations.

The examples on this presentation are multifaceted. It may be helpfully to observe the examples more or less profound.

empty

5.1.3 File tree structure of the examples

The `OFB_Presentation` examples use a dedicated file tree structure. The basic ideas are:

* Strictly separation between sources (`.../src`), generated files (`.../build`), external (downloaded) libraries and tools (`.../tools`, `.../libs`), IDE-Project environments (`.../IDE`)
* On second level separation of different components (`.../src/Cmpn1`, `.../src/Cmpn1, .../src/Templates_OFB, .../src/emC`) and parallel also in `...build/Cmpn1` etc.
* On third level separation of different tasks of the files for the component: (`.../src/Cmpn1/odg`, `.../src/Cmpn1/cpp`, `.../src/Cmpn1/makeScripts` etc).

It may be seen also for users projects to use a similar structure. There are some discussions in the internet about such topic, but without a really proven and standardized solution. Advantages and consideration about this file tree is described in https://vishia.org/SwEng/html/srcFileTree.html

5.1.4 Build your own project with OFB

To work with your own approaches, you can copy one of the given examples, and modify it. Copy the `OFB_Presentation/src/Templates_OFB/odg/OFB_DiagramTemplate.odg` is also possible.

It is recommended to create your own working space on hard disk. Merging within the `OFB_Presentation` file tree as additional component may proper for learning experience, but not for a really user project. For your own, you may use the same working tree structure as in `OFB_Presentation`. This is a proposal explained in https://vishia.org/SwEng/html/srcFileTree.html. But of course, use your own approaches, if you are familiar with path and arguments of the OFB translator.

* The path of your own working space is dedicated in the following directory presentations as `.../`. It may be located in Windows anywhere for example on a `D:` drive, in Linux in any sub directory from `/home`.
* Copy all files from one selected example component from the `OFB_Presentation`, for example from `OFB_Presentation/src/ExmplBandpassFilter/*` or from the more simple `OFB_Presentation/src/MyExampleComponent` to your `.../src/MyownCompn/*`.
* Copy the following Component directories complete from `OFB_Presentation/src/` to your `.../src/`:
  * `organizeScripts`: contains some common scripts
  * `Templates_OFB`: contains some templates as also a control FBlock library
  * `src_emC`: Contains C(++) sources for a proper runtime system. It may be recommended to used it, or replace it later by your own concept.
  * `load_tools` or alternatively copy the `OFB_Presentation/tools` directory to your working space `.../tools`. Later the tools can be placed on a centralized directory on your PC for example `C:\Programs\vishiaTools` or `/usr/local/vishiaTools` if you use the OFB approach for some more projects. It can be also delegate to an administrator for your PC. Then you need to adapt the file `.../src/organizeScripts/setJCP.batch` or `...sh` using the absolute path to the jar files. If you have experience with the tools and scripts, you know what to do.
* Rename the example file in `.../src/MyownCompn/odg/*.odg` to your file name.
* Rename the name of the copied file `.../src/MyownCompn/makeSrcipts/genSrc_Cmpnxx.bat` or `...sh` and rename the arguments in this file.

If you have done this steps, then the new `gensrc_MyComponent.bat` (or `...sh`) should generate the same results as for the example, but in your `.../build/MyownCompn/` directory.

Then you can remove the most of content in your new `odg/MyComponent.odg`, but you have the styles, and some basically GBlocks on one page. Draw your first ideas and look what is generated in the target code. The rest is step by step.

5.1.5 Consideration about generated sources for target compiling

There are three topics: **build**, **compare**, **use** proven sources for target.

But before explain this, you may following this consideration:

Often graphic programming is presented as a higher level than *“old style stupid working with C sources”* similar as *“using a high level language is modern, which guy works yet with assembler language”* in the past. But both considerations are faulty, they are slogans from tool providers. The real truth is: Some developer works or want to work with higher problem descriptions solutions, mathematics, control algorithm, graphic, or also architecture of software in UML/ SysML. But also some people in the team are responsible to the fact, that all should proper work with own knowledge, not with juristic guarantee from any vendor. In opposite, if you use a specific hardware platform, with specific development tools which are often in graphic, the vendor of these platform guarantees that it works together. But this is not so for specific embedded device solutions. If you rely on a specific predefined *“hardware support package”*, then you are giving up your own understanding of how it works.

It means, to really have knowledge about your embedded product, some people in the team should have knowledge about machine code instructions of the used controller, looking in and understand the list and map files with assembler code, machine instructions, memory addresses etc. Working direct in machine instruction level, can be necessary for specific hardware driver.

And also, looking on generated source code (C/++) from the graphic may be interesting. Also realize some parts in the target source code, and use it from the graphic level. The approach of graphical programming is not: *“save money for coding effort”*, it is: *“get a proper documentation and common understandable problem and solution description”* with guaranteed consistent textual target code to the graphic.

A next may be very important consideration is: What’s happen after some years (product responsibility for the next for example 30 years): Some times, especially by proprietary tools, the original graphic tools may not be available after some ten years, or more expectable: The specific know how for this graphic tools is lost, the developer from the old times are in pension (or dead). Whereas, working with proven lower level languages (C++) is very wide spreading and long term familiar. It may be simple to make adaptions in the C code if a hardware (sensor etc) is changed, as the effort go back to the graphic.

That’s why look on the generated sources may be important. The generated target code textual sources plays a role of “second level sources” beside primary level textual sources and should also store in a version management system.

That are the pre considerations. And now **build, compare and use**:

5.1.5.1 Build the generated sources from graphic in a temporary build directory

This is a similar approach for *“compile C/++ sources to temporary Object files”*. The Object files are not necessary to permanently stored. They can be rebuilt anytime, are only necessary as intermediate files to build the executable code for the target. Consequently, I use a RAM disk for that, which content is cleaned after any reboot of the PC. That’s why also some proposals exists in the OFB_Presentation file and description to install and use a RAM disk, and use a symbolic link (JUNCTION in Windows) for the build directory. `.../build` is beside `.../src` and not a part of it.

Now it is possible to configure a target executable (or PC executable) make system or IDE, which uses immediately the content of `.../build/CMPN/cpp/genSrc/*` files to compile. With that approach is is also possible and done for the `OFB_Presentation/src/ExmplBandpassFilter` to build and run the target executable with ***one button press approach***.

5.1.5.2 Compare the new built files with the last or persistent version of it

That's the second, **compare** topic. The generated source files as reviewed variant are stored inside `.../src/CMPN/cpp/genSrc/*` to compare, and also to use as reviewed sources.

General, the generated sources should be understandable for a normal target language (C/++) developer. This is one approach of OFB.

The comparison of the sourced newly generated after graphic changes with the established sources can be help to detect faulty inputs in the graphic, even if it's only an accidentally removed connection on a glue point of a connector, or a writing mistake. The generated source code is a second mapping of content, beside also some report files and the FBcl presentations of graphic content.

Then, after comparison, the reviewed variant of the generated source in `.../src/CMPN/cpp/genSrc/*` should be refreshed. In the OFB_Presentation (https://vishia.org/fbg/html/Videos_OFB_VishiaDiagrams.html) using of the textual diff tool Winmerge (https://winmerge.org/) is proposed to use also for Linux in a wine environment ("**win**dows **e**mulator").

5.1.5.3 Use the persistent stored generated sources or the temporary built ones

This generated sources in the persistent version should be both, **stored in the version management system beside the graphic files**, as also used for the finally target compilation for a product, available for repeated compilation after ten of years. There should not be a problem for more as one make & build system (command line execution or IDE), some uses the build... sources, some uses the src ... generated sources.

5.1.5.4 Generating different target sources for some target systems?

The answer of that is a strong NO.

The higher level languages, especially C, was developed for the approach to have a platform independent language for the several computer systems of the 1970th. Especially C and also C++ have some mechanism for adaption on different target approaches. That is not only the conditionally compilation (`#ifdef ...`). It is also set the include path to several entries and use the specific include files for the target, the macro possibility of replacement of parts in the sources via content in the (specific included) header files. The (emC concept (embedded multiplatform C/++, https://vishia.org/emc/index.html) is especially written for such approaches.

Big player for tools and compiler preach often for specialism in their compilation tools and generated sources adapted to special features of the selected platform. But this is done to secure their sales and influence, to achieve tool binding. It is not done by really academic considerations.

5.1.6 Some tips and tricks for LibreOffice

This is mentioned on the beginning of the description, because some stuff may be important. You may read and have experience step by step.

5.1.6.1 Move FBlocks and Pins with or without grouping

The is a possibility in Draw to group shapes. This may be seen as predestined for GBlocks and their GPins. But LibreOffice Draw does not allow to select a member of the group (a pin) for connection with glue point, only the whole group. The necessary action - ungroup, connect - group - is too costly for daily handling.

That's why it is recommended to let grouping be, instead catch with selecting area all shapes, GBlock with pins, shift it as necessary (may be use the cursor instead the mouse). If one element is not caught, shift it afterwards. The Pins and its GBlock are related together, if at least on edge of the GPin is inside the graphic area of the GBlock shape.

5.1.6.2 Lost connections to pins

Sometimes, on copy operations or because the mouse is slipped, a connection to its glue point of a Draw-connector is lost. This is not visible, a non connected Connector has the same graphic appearance as the connected one. This is a really disadvantage of LibreOffice Draw, also in the last current version (25.8.3).

To see this effect you should select the shape of the GPin and move it a little bit (recommended use arrow keys) and look whether the connector follows (is glued) or not. You can also select some pins together and make the same. The state of a lost connection is also seen in the generated code (a part is missing) or in some report files. See TODO chapter report files.

5.1.6.3 copy styles

You can use this drawing content in `OFB_DiagramTemplate.odg` to pick up an element, copy it to clipboard and insert it in your graphic. The associated style is also copied if it is not already existing in your destination draw file.

How to copy styles:

Unfortunately LibreOffice does not allow loading style sheets from another given odg document, only by copying the original one (see also https://ask.libreoffice.org/t/how-can-i-import-styles-from-other-draw-documents/8834).

But you can copy the internal `style.xml` file from the `UML_FB_DiagramsTemplate.odg` zip archive. This is a simple, proven workflow that has not been recommended as often, but it works:

- Copy the original `OFB_DiagramsTemplate.odg` file to `OFB_DiagramsTemplate.odg.zip`
- Open the zip file by a unzip tool.
- Copy the internal `styles.xml` for your own.
- Make a backup from your own `*.odg` file only to have it for trouble.
- Rename your own `*.odg` file to `*.odg.zip` and open it with a zip tool.
- Replace the internal `styles.xml` with the `styles.xml` from the template.
- Rename your own `*.odg.zip` file back to `*.odg`
- Check if all is proper. It should be.

empty

5.1.7 Terms used in the following description

There are many relationships between graphic, blocks and pins, UML, code generated etc. For that an overview about used terms are given here.

The terms are not alphabetic sorted, user find functions on working on pdf document. Instead they are sorted to common topics.

LibreOffice:

* ***Style***:

Blocks and Modules and there wiring:

* ***GBlock***:
* ***FBlock***:
* ***GPin***:
* ***Pin***:
* ***Module***:
* ***Event chain***:

Relations between ...:

* ***Association***: This term is used in context of **UML**. An association is a referenced other module or FBlock. In C/++ language this is usual designated as a *'pointer'*.
* ***Aggregation***: This term is used in context of **UML** similar as an 'Association'. The difference between both is: The 'Aggregation' is one time determined as relationships between modules or FBlocks and then no more changed, whereas an Association is flexible, can be changed.
* ***Composition***: This is the 3th kind of relations between FBlock and modules beside Association and Aggregation in **UML**: It is a permanent relation from one FBlock or module to another FBlock, whereby additionally the source module or FBlock is responsible to create the destination FBlock. In the OFB diagrams each FBlock in a module is a composition from the module representing intrinsic, not drawn FBlock to the member FBlock of the module.
* ***Affiliation, affiliated***: This term is used for example to describe, that a specific **pin** is **affiliated to a specific event chain**, is driven in this event chain. It is also used for a **transition** in a state machine as **affiliated to a specific region** or just event chain.
* ***appropriated***: This term is used for association **between events and** the appropriated **data** pins.

  It is also used between a FBlock and its appropriate module for code generation,

  or for the FBtype appropriate to the FBlock, or adequate for the appropriate FBlock to a GBlock.
* ***related***: This term is used to describe the association **between input and output** event or data.
* ***corresponding***: This term is used to describe the association **between prepare and update events**. and also from the events to its corresponding threads.

StateMachines:

* A ***StateMachine*** is a usual term in control logic to describe states and their changes. It has a large history, see *5.11.1 State Machines in the past, history, hardware, PLC* page *164*. In UML for a state machine diagram also the term ***'state chart'*** is in use. The omg.org writes also StateMachine with CamelCase writing style.
* ***State***: In common meaning it is the currently coherent state in a module. But for State Machines in graphic it is presented as one vertex in words of omg.org (chapter 14.2.3.3 Behavior StateMachines - semantic - vertices, page 304). This is similar as a state is presented with exact one FlipFlop in hardware in a 1-of-n presentation, if only one FlipFlop is set for each state.
* ***Region***:

empty

5.2 All Elements with their styles

5.2.1 Template file.odg for the graphic with elements and styles

The next image shows all given template elements. It is the content of the file

https://vishia.org/fbg/deploy/OFB_DiagramTemplate.odg

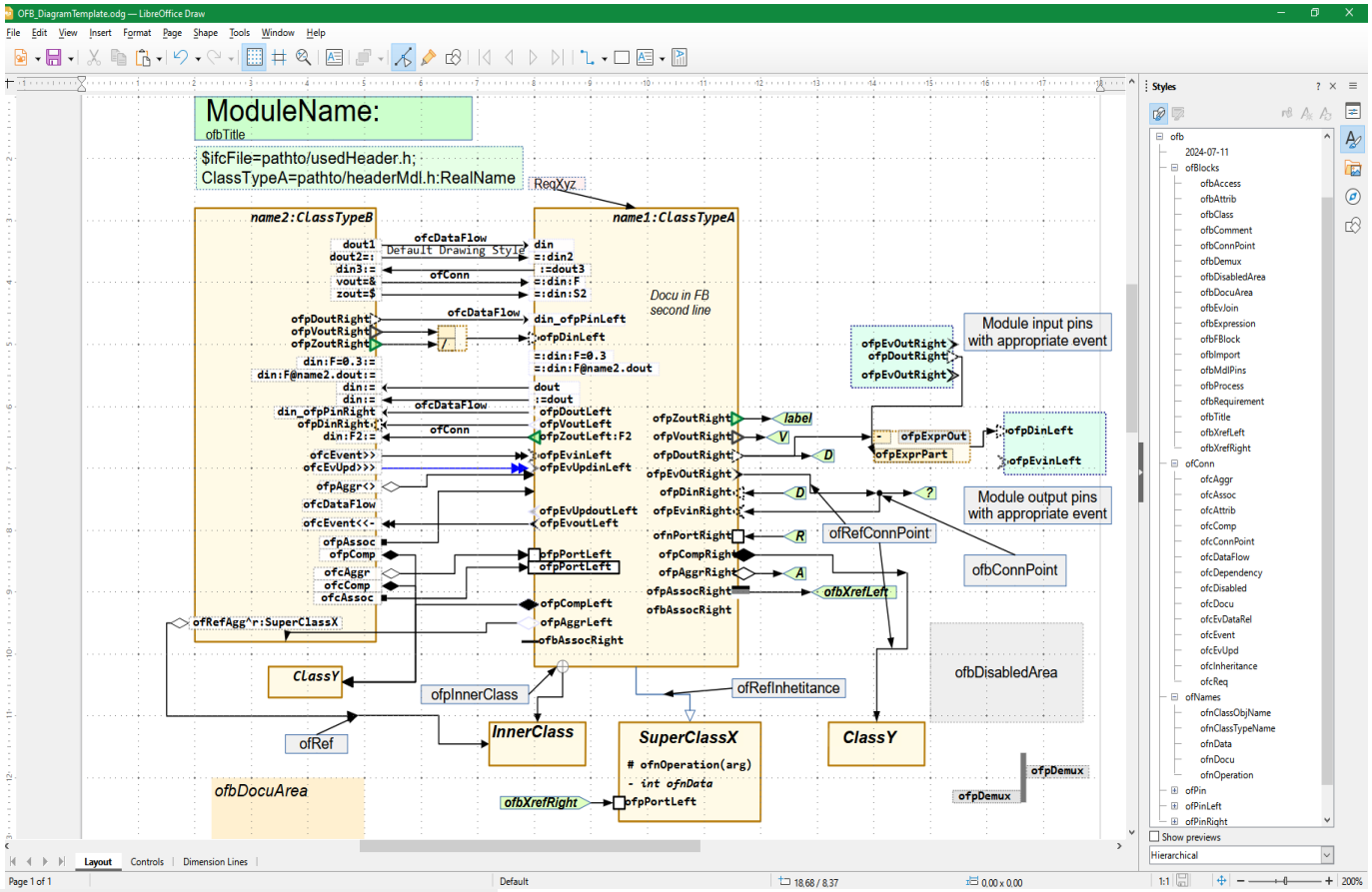


Figure 26: odg/OFB_DiagramTemplate.png

This is the whole view to the opened LibreOffice `OFB_DiagramsTemplate.odg` with the template file. Right side you see some style sheets. Activate this view with menu *"View → Styles "*. The drawing content contains some examples with its figures.

The styles can be general adapted in their outfit for your own. But remain on proven concepts. For the OFB graphic evaluation the ***names of the styles*** are essential, not any graphic figure outfit. Also some syntax in the description texts are essential. See

The class in the mid with `name: ClassTypeA` contains all connection elements for the concept described in *3.3 Using styles for graphic elements* page *8*. The identifier of the style sheet is here used also as name, only for documentation.

The class left `ClassType name` contains simple connection elements of the base style `ofpPinRight` and `ofpPinLeft`, but using connections with the specific type. Their style names are shown here as pin names. This was a first concept, maybe in future not recommended. Here the connection styles determines the kind of the pin.

The figure outfit is proper for view, but not necessary for content. It is also possible to use simple rectangles with the proper style. Then it is not so good recognizable which kind of pin it is. But handling of content (the text) is more proper. It may be recommended to use this simple rectangle forms for the amount of data pins, and use the specific form with the triangle shape for the events to see what's happen. This is in the moment growing experience. The evaluation of the graphic works with both variants, because for evaluation only the associated style is essential, not the form.

The internal data of a class can be shown, as usual in UML, with the style `ofnData`. The designation about private, public, protected should be written with a first character `-` `+` `#` as usual in UML. Writing the type of the data is recommended. The operations can be written with their argument names, if it is more informational. The operation itself, its body, should be define anyway in a programming code and not with a diagram. The association between the shown operation in a diagram and the real operation is only for documentation, should not be formalistic.

The meaning of the styles is described in *5.2 All Elements with their styles* page *42*

5.2.2 Graphic block (GBlock) styles, ofb

Als GBlock wird ein shape bezeichnet, dessen Funktion einem Function Block in der FB-Denkweise formulieren. Zu den Grapfikblocks zählen auch die hier angeführten Styles für Titel, Disable area etc.

Folgende styles definieren GBlocks:

Trennen GBlock von Managing blocks

Jeweils Bspbilder.

GBlock (*Graphic Block*) styles should be assigned to shapes that represent blocks with specific meanings, except pins. Usual that are rectangles with a little bit more size, greater then 1 cm. It is:

- `ofbTitle`: This is a shape which contains the name of the module on this page. It is necessary one time on each page. See *5.5.2 Module in odg file(s) organized in pages* page *69*

- `ofbAlias`: This is a shape which contains the association between aliases (short names) and the real used String for this names. This can be used for type names (FBtype) as also for constant strings. See *5.5.2 Module in odg file(s) organized in pages* page *69*

- `ofbMdlPins`: This is a shape which contains the pins of the module, see *5.5.2 Module in odg file(s) organized in pages* page *69*

- `ofbClass`, `ofbFBlock`: Both styles have the same semantic, because a class or FBlock is distinguished by its name and type. The element can present an instance of a class (having an instance name), that is a *“FBlock”*, or it is (only) a type / class presentation. In any case it presents a part of the properties of a class or type, sometimes as named here as *”FBtype”*. See *5.6 Possibilities of Graphic Blocks (GBlock)* page *98*

- `ofbExpression`: This is an expression FBlock or also named *“FBexpr”*, see *5.8 Expressions inside the data flow (FBexpr)* page *128*

- `ofbEvJoin`: This is usual a bar (vertical). All ending connectors are inputs, one starting connector is the output. It is a representative for a `Join_UFB` Function Block, see *5.12 Execution order, Event and Data flow, Event chains and states* page *196*

- `ofbDemux`: This is usual a bar. Either it has some ending connectors and one starting connectors. Then it is a multiplexer which joins some signals, independent of there meaning and kind. Or it has one ending connectors and some starting connectors. Then it is a demultiplexer. The order of signals is then the same as on the connected multiplexer. see *5.7 Connection possibilities* page *118*

- `ofbDisableArea`: This style can be applied to a rectangle shape which covers some other shapes. All shapes which have at least one edge coordinate inside this area of this `ofbDisableArea`shape are not recognized by evaluation of the graphic. The appearance of this shape should be a gray area which is enough transparent to see the elements.

Disablearea ist kein GBlock sondern MBlock

- `ofbAttrib`: This is usual a text field or a rectangle with text, which is associated to a FBlock or often to a class by a `ofcDependency` or also `ofcConn` connector. It declares some additional information to the FBlock or FBtype, not yet used for code generation, but maybe interesting for the diagram.

- `ofbComment`: This is a text field or shape with text which contains additional (free formatted) information which should be shown in the graphic. It is associated to any other graphical block shape (GBlock) by a `ofcDocu` connector style.

- `ofbDocuArea`: This should be used for simple rectangles which gives a color under some shapes to show one area of functionality.

- `ofbRequirement`: This is a text field containing only a requirement identification or some requirement identifications separated by comma, to assign a solution shown in the graphic to a requirement. It should be connected to any element with `ofcReq` or simple `ofcConn`. It means that the referenced (connected) detail fulfills the named requirement(s).

- `ofbProcess`: This is a text field which contains one step to execution to show process flows. It is yet not part of code generation. Should be regard in future to generate an

operation from given flows. See *5.14 Showing processes* page *206*

- **ofbConnPoint**: A connection point is usual a black circle with <1mm diameter. One connector should end there, and some connectors should start there. All connection lines starting there are then connected logically with the start point of the ending connection line.

- **ofbXrefLeft, ofbXrefRight**: It should be assigned to a shape for a Xref. The distinction between ...Left and ...Right is only for appearance, see the template file.

5.2.3 Name styles, ofn

This style can/should be assigned to text fields which are located inside a GBlock.

- **ofnClassObjName**: This should be assigned to a text field to determine the name and type of a FBlock, see *5.6.1 Difference between class, type and instance (“Object”)* page *98*

- **ofnClassTypeName**: is deprecated and the same as **ofnClassObjName**. First it was planned to distinguish a type of class and a FBlock by this specific style, but it is worse recognizable in graphic. The found solution, mark a type anytime with a leading `:` is not UML conform, but more clearly.

- **ofnData**: A text field with the name of an element in a class (or FBlock), adequate an attribute in UML class diagrams. Also the UML conform leading designation for `-`private, `~`package private, `#`protected and `+`public are accepted there.

- **ofnOperation**: A text filed with the prototype for an operation which is declared for this type, as known from UML. Also here `-` `~` `#` `+` as visibility hints can be written.

- **ofnDocu**: This is a field containing documentation for this type (FBlock).

5.2.4 Connector styles, ofc

For connectors between pins the connection style is not evaluated. The pin style is determining. Also the `Default Drawing Style` can be used for it. The style is proper only for appearance:

- `ofcAggr`: It shows a non filled diamond on the start of the connector as in UML.
- `ofcAssoc`: It shows a very small filled rectangle (0.6 mm) on the start of the connector, to distinguish from the standard connector
- `ofcComp`: It shows a filled diamond on the start of the connector as in UML.
- `ofcConnPoint`: This style is attended to use as connection to a connection point or to connect two connectors. It has a very small arrow on end (0.6 mm).
- `ofcDataFlow`: attended to use but not necessary for data flow (can be removed in future, do not use it).
- `ofcDataFlow`: attended to use but not necessary for data flow (can be removed in future, do not use it).
- `ofcEvent`: attended to use but not necessary for event flow (can be removed in future, do not use it).

The following connector styles are used to connect GBlocks. They have a proper semantic meaning and should be used:

- `ofcInheritance`: Inheritance between types also able to apply from a FBlock to a class GBlock (without name). If the referenced GBlock is a FBlock with name, the instance is not used. As familiar in UML the end is a non filled symmetric triangle arrow.
- `ofcDependency`: Dependency between types (the source type uses the destination type). As In UML a long dashed line with an open arrow on end.
- `ofcDocu`: From a `ofbComment` GBlock to the appropriate destination, a gray dotted line with a small filled arrow on end.
- `ofcReq`: From a `ofbRequirement` GBlock to the appropriate destination, a gray dashed line with a small filled arrow on end.

The following connector style is not used yet but should be necessary:

- `ofcEvDataRel`: For connectors between pins to associate event and data. Todo: If this connector style is applied at least between two pins of a FBlock or FBtype, then an automatically association between all shown pins in the GBlock is not done. See *5.6.2 GBlocks for each one function, data – event association* page *100*

Note: In opposite to UML aggregations, associations and compositions are never starting from a GBlock, only from a pin. The pin contains the name of the reference inside the source type.

Note: The aggregation and composition uses a non filled and filled diamond as arrow style on begin. This kinds of arrow was available in LibreOffice versions till 7.x and also in Open Office. In newer versions (24.x) this styles are removed. But it is possible to create own styles respectively use the diamond styles from an older version of LibreOffice. The arrow styles are contained in an XML file `user/confi/\standard.soe` able to find in the users area, in Windows `c:\Users\THE_USER\AppData\Roaming\OpenOffice\4\user\...`} , in Linux in TODO and in the portable Windows version in `LibreOfficePortable.24.2\Data\settings\user\config\…`.You can simple merge the content of this file in a newer version with the content from an older version. But you should familiar with the XML syntax.

empty

5.2.5 Pin styles, ofp

This styles can/should be assigned to pins of a GBlock. The pin styles can be used ending with `…Left` or … `…Right` or without this. for evaluation with our without `…Left` or … `…Right` has no meaning. The styles with `…Left` or … `…Right` should be used for small specific pin shapes (2*2..4 mm), the text is written left or right from the shape. Whereas `…Left` is for a pin left side with the text right side, and vice versa.

Hier mal ne Graphic

Unterscheiden Dpins, Epins Rpins, Mpins

The styles without this left/right designation should be applied to simple text fields, which has a semantic meaning adequate the pin style but also a (default) appearance, see template.

The pins can also be determined to a specific type using leading or trailing designations before and after the pin name. The also the basic pin style `ofPin` can be used, the semantic is determined by the designation, see *5.7 Connection possibilities* page *118*.

You can decide by your own using the pin style for semantic or using the here also documented leading or trailing designation, or using both. It is also a topic of appearance.

Only one of the leading or trailing designation should be used, whereas it is proper visible to use the leading one with a pin left side and trailing for right pins (near the border of the FBlock). For the evaluation of the graphic leading or trailing does not play a role. But be attentive to use the correct characters different for left and right. The characters should have a proper mnemonic.

- `ofPin`: Common style of a pin with a text field, determined by leading or trailing designation. This designation is able to use also on all other pin styles, on left **or** right side (usual on the outer side, means left on left side pins, right in right side pins) with the following meaning as the adequate pin style: …

`->name<-` same as `ofpEvin…` Event input

`<-name->` same as `ofpEvout…` Event output

`->>name<<-` same as `ofpEvUpdin…` Update Ev

`<<-name->>` same as `ofpEvUpdout…`

`:=name=:` same as `ofpDin…` Data input

`!=name=!` same as `ofpVin…` Data input as variable

`%=name=%` same as `ofpDout…` Data output

`&=name=&` same as `ofpVout…` Instance var.

`$=name=$` same as `ofpZout…` Update dout

`&<name>&` same as `ofpAssoc…` Association

`<->name><->` same as `ofpAssoc…` Association

`<%><name><%>` same as `ofpAssocRo…` Association

`<_>name<_>` same as `ofpAggr…` Aggregation

`<*>name<*>` same as `ofpAggr…` Aggregation

`<&>name<&>` same as `ofpAggrRo…` Aggregation

`<$>name<$>` same as `ofpComp…` Composition

`[&]name[&]` same as `ofpPort…`

`input=:descr` or `descr:=input` is usable for `ofpDin…` or `ofpExprPart`, whereby the last one should be have this dedicated style. See also 5.3 Texts in graphic blocks and pins page 10.

An `ofPin` without dedication is used as `ofpDin`.

`input=!descr` or `descr!=input` or `=!` or `!=` left or right dedicates `ofPin` as `ofpVin…`

- `ofpAggr`: `<&>name<&>` It is an aggregation of the type and an aggregation assignment (in init phase) for a FBlock instance. Aggregations as known in UML are valid with the initialization and cannot be changed in run time. The aggregation pin is associated with the init or ctor event in a FBlock, never to the prepare event. ***Mnemonic hint:*** `< >` is similar a diamond. But using `<>` can be confused with 'not equal' for expression terms. The `&` is the known designation for a reference.

- `ofpAssoc`: `&<name>&` It is an association of the type. An association known from UML is a temporary assignment to a specific object. Hence in a FBlock diagram it should not be wired to a specific FBlock (then it is in fact in Aggregation). Possible usages are connections to a conditional switch, a select switch or a specific port output which is volatile. The association pin is assigned to the prepare event in the same FBlock. Its value is assigned in any prepare event flow. ***Mnemonic hint:*** It is just not a diamond, only a reference.

- `ofpComp`: `<*>name<*>` It is a composition as known in UML of the type and an Allocation of the composite type for a FBlock instance. Compositions are initialized and valid with the construction and cannot be changed in run time. ***Mnemonic hint:*** It is similar a filled diamond in a textual representation.

- `ofpPort`: `[&]name[&]` A port in UML is an access point to inner instances. Here it is also the access as destination of aggregations or associations. The implementation of the FBlock is responsible to provide a proper pointer to inner data of the FBlock for code generation. The port can provide different inner instances in runtime, usable for associations. ***Mnemonic hint:*** A square `[ ]` is familiar in UML. The `&` inside should associate to a 'reference' in C/++ thinking.

- `ofpDin`: `name` or `input=:descr` or `descr:=input` Data input, without marker or this marker used inside. See 5.3 Texts in graphic blocks and pins page 10. ***Mnemonic hint:*** `=:` is the assignment operator in PASCAL and automation control languages.

- `ofpVin`: `name`, `input=!descr` or `descr!=input` Data input stored in a variable, without marker or this marker used inside. ***Mnemonic hint:*** `=!` is similar `=:`, but stronger (`!` to save).

- `ofpExprPart`: `descr` or `input=:descr` or `descr:=input` Expression input, the simple `ofPin` is not usable for that. `descr` is described in *5.8.2 Expression data input pins DinExpr ofpExprPart* page *130*.

- `ofpExprOut`: It is an output of an expression, the simple `ofPin` is not usable for that.. See *5.8 Expressions inside the data flow (FBexpr)* page *128*

.- `ofpDout`: `%=name=%` Data output, the data are locally defined.

.- `ofpVout`: `&=name=&` Data output as instance variable in the module. The data are set inside a specific prepare flow, but accessible in all other event flows or also from outside (by an inspector tool, visible in RAM which debugging in run time). ***Mnemonic hint:*** `=` anytime used for output, the `&` should associate to a referenced variable.

- `ofpZout`: `$=name=$` Data output as instance variable in the module. The data are set with an update event. It is a state variable usable in all other event flows and also usable as *"value from the last step"*, in Simulink known as ***"Unit Delay"*** regarding to the prepare event flow. But it is also seen as Simulink adequate ***"Rate Transition"***, whereby the update flow timing decides about validating.

Mnemonic hint: `=` anytime used for data assignment. The `$` should associate to a "S" for state variable. `&` is known in C/++ for a reference.

- `ofpEvin`: `->name<-` Event input used for the event flow. ***Mnemonic hint:*** should mark a `->` flow to inside or from right also to inside.

- `ofpEvUpdin`: `->>name<<-` Update event input used for the event flow. ***Mnemonic hint:*** should mark a `->>` more meaningful flow to inside or from right also to inside.

- `ofpEvout`: `<-name->` Event output used for the event flow. ***Mnemonic hint:*** should mark a .`<-` flow to outside (left) or also `->` to outside to right.

- `ofpEvUpdout`: `<=name=>` Update event output used for the event flow. ***Mnemonic hint:*** should mark a `<=` and `=>` is mor stronger to outside.

- `ofpDisabled`: A pin which is disabled for evaluation, maybe temporary disabled but just preserved in the graphic.

5.3 Text in graphic blocks and pins

The text entries in all graphic boxes and pins are built with the same syntax, because using the same algorithm on reading from the graphic. The pin designations for `ofPin` with the designation `=:` `:=` `->` etc left and right, which can be used instead the specific pin style, are not part of this evaluated text, see *5.2.5 Pin styles, ofp* page *48*.

See also >>>*Impl-OFB_VishiaDiagrams.pdf*: >>>*7.3.3.4 Evaluating Pin texts25*

5.3.1 Syntax in colored ZBNF

The simplest form, used for FBlock is:

```
name:Type
```

or exact in ZBNF syntax

```
descrType::=[<*:.{[?descr> ]
     [{{<fbSlices>?,}}]
     [<?elemDst>[[|.]<*:?>]
     [ : <*[?sType> [[<sizeArrayType> ] ].
```

`name` is the `descr`. Formal semantically `descr`, is all till `:` , `.` or `[` . This is the meaning of the syntax description `<*:[.?…`, *"all till one of the given character"*.

The `elemDst` is optional written in `[…]`. It is used to set an output element. `type` is also optional, starting with a `:` in the option `[ : …]`. Then it is all till end of the text described with `<*[?type>`. and `sizeArrayType` is a designation o array sizes or container properties.

Following a formal syntax, which contains all possibilities, is anyway correct but often not so proper understandable. Instead, given the syntax with examples is more understandable, but sometimes incomplete. If it is more complex, questions are get opened. That's why a proper way to explain text expressions is:

* explain it with examples, which are proper understandable,
* but also describe the exact syntax, as complete description.

The syntax is shown colored to distinguish between syntax control characters (in green) and terminal characters (yellow, larger). The `semantic` identifier can be an `meta-` or an `endMorphem`. The `endMorphem` identifier is used in the explanation as also in the program (Java code, same name of the variable). The `metamorpheme` is a part of text, which is described by an inner syntax. The terminal characters are texts as given. In opposite to EBNF they are not written in quotation (it's better readable). Instead, in the non colored from, conflicts to syntax control characters are solved with transliteration with backslash. `\[` is the square bracket. But in the here used colored syntax the square bracket and the other syntax control characters are immediately written as terminal `[` .

The base for the syntax is ZBNF writing style: https://vishia.org/docuZBNF/sfZbnfMain_en.html. This is similar the known BNF (Backus Naur Form) from the 1960th, still known and used, but with more possibilities. as also similar to the EBNF (Enhanced BNF introduces from Prof. Niklaus Wirth for PASCAL notations, also familiar for IEC61499 and IEC61131 automation control languages). One advantage of ZBNF is: It shows more obviously the semantic with the writing style `<syntax? semantic>`. Some hard programmed syntax control possibilities able to expressed, as *"all text till one of ..."*

`<$?…` parses an identifier.

`<#?…` parses a number.

`<*.+…?` parses all till the given characters

`<*|string|s2|…?` parses till one of strings

`<$?semantic><$-$?semenatic>`

`< terminals <*?semantic><]` parses first from right using `lastIndexOf(…)` (not in the original ZBNF from ~ 2015)

`{ forward ? backward }` Repetition

5.3.2 The complete Syntax of text for pins and FBlocks

For pins some more possibilities are given. Next shows the complete formal syntax:

```
inputDescrEtc::=
[ [<descrType>] [?<*`?special>] | `<#?nrGpos><] := <input> ]      A
|[ <input> =:                                                     B
 |[<?input>[.| [|: ]<*!~+-*%/<>=&^|?`?>                          C
 |[<?input>[.|[]<*?>                                            D
 | <*!~+-*%/<>=&^|?`?input!*@>                                   E
 |]
              [<*?`?descrType>][?<*?special>]|`<#?nrGpos><]      G
].
```

The first line **A** shows that an *input* part can be written right side after a **:=**. This is for input pins on the right side, where an input handling, comes from outside on right, it's written better right.

Bild fehlt, beginnen mit ganz einfachem!

***kFactor + := .re** This is an example for a right side input pin for an expression. From the input the real part is used with **.re**. The operation is +, but also a factor is multiplied as part of the expression using the K pin (it is not an input handling). See *5.8.2 Expression data input pins DinExpr ofpExprPart* page *130*. Spaces increases readability. Spaces in the syntax description means, this leading and trailing spaces are removed while parsing. It means the *input* used for this example contains only "**.re**" without the spaces.

The opposite is the second line **B**, where an *input* is written left of a **=:**.On pin left side the same example can be written as:

```
.re =: *kFactor +
```

Note that the **=:** or also **:=** is an assign operator in PASCAL and the automation control languages, should be known. Here it is has the same meaning, *"assign this value after input handling"*.

The 3th line **C** shows that the input handling can also be written without **=:** ,it saves space. Instead, the input starts with either **[** or **.** or **:** for an element access on input or input type cast, and goes till one of the operators for expressions or till **?** or **`**.

4th line **D** shows that if the text starts with **.** or **[** and does not contain one of the separation chars, it is also ***input***. That is for a simple input data access.

.re This is the simplest example for this variant. It is a pin description only for access to the real part of the input variable. **.re=:** is the same.

[12] Same as simple access to an array element on input, same as **[12]=:**

[12]:int16<<8| Also a cast is possible. **<<8|** is the ***descr***, the input after cast should be shifted left by 8, then or.

:W<<8 This is first the cast to **uint16** (Word) and then a shift to left. **:W** is the input.

=:[12] array access right side means set of the array in the output variable.

The 5th line **E** shows also an input without **=:** but not starting with **[** or **.**or **:**. Instead, the string till one of the operator **must contain** anywhere **the character** **@**:

fb2@?step0`2 This is a typical example for the aggregation pin of an operation FBlock, see also *5.8.11 FBoper, operation for a FBlock* page *151*. **fb2@** is the aggregated FBlock, **step0** is a ***special*** designation, used for the event. After them also the ***nrGpos*** is given.

fb2@=:?step0`2 This is the longer form.

fb@pin:float access and cast on an expression input.

fb@pin:float=:pinname Here the **=:** is necessary to separate the **pinname**.

The line **G** describes the other side, consists of ***nrGpos***, first parsed from right, a special designation and the *descrType* mentioned in the chapter before.

5.3.3 Syntax of input to a pin

The input description allows textual given connections or constants on an input, as also selection of elements and a value cast of the input value:

```
input::=
[<?constInput>'<*?>  A            B      C            D
|[ [[<$?fblockC>][{{<fbSliceC>?,}}]@[<$?pinC>] ] [<?elemSrc>[[|.]<*?>]
 | <*?constInput>    E
 ] [:<*?valueCast><]    F
].
```

The input to a pin is possible for all input pins both on FBlocks and expressions. There are four possibilities. General the input goes till a `=:` as described for *inputDescrEtc* see page before.

A If the text starts with an apostrophe `'`, then all till end is stored as ***constInput*** . It is stored in meaning of a string literal inclusively the beginning `'` if an ending apostrophe is existing. If the end apostrophe is not found, the constant is taken without the beginning `'`..This is used to mark the text anyway as constant.

If the non string literal ***constInput*** contains an identifier, it is checked whether it should be translated with the given alias in the `ofbAlias` shape, see *5.5.6 Module pins* page *78* This enable the opportunity to use a short alias for a longer text in the constant expression. How the constant is used - it depends on code generation.

The alternatives in syntax consists of:

F If not **A**, then first backward parsing till the `:` an optional ***valueCast*** will be detected, it shortens the left side of text.

B Then the remaining left side or the whole text of the *input* is checked, whether it contains a `@`. It is the optional input connection instead a wired connection. Only then the content is first parsed as identifier for ***fblock***. If an ***fblock*** is detected, Then it is checked whether either from left or immediately after the ***fblockC*** `{` follows for ***fbSlices***. More ***fbSlices*** are separated with comma. After them a `}` must be following, then the `@`. If is is not so, the parsed result is used but a *"WARNING graphic faulty connection ..."*

C After the `@` and identifier is checked whether a ***pinC*** follows, both are optional.

Examples for inputs are:

`fb1@pinX`: Access to a pin of a FBlock

`fb2@`: Access to a FBlock without pin, for example for aggregation

`fb{a,b,3}@pin`: Access to the pin of three sliced FBlocks. This builds an array type input, or can be used also as three connections for the sliced FBlock with this pin description. See 5.6.8 Sliced or Array FBlocks, Demux and array data page 35

`{fbX,fbY}@pin`: Access to this two FBlocks with this pin to use for a sliced or array input.

`@mdlPinY`: Access to a module pin

`@nameXref`: If the identifier is not found as FBlock or module pin, it is searched in the pool of Xref, see 5.7.5 Xref page 43

D After this input connection or also from left the ***elemScr*** starting with `[` or `.` is detected. This is used also on a graphical wired connection to access an element of the connected variable, maybe a structure element or an array element.

E Alternatively if neither an input connection nor an ***elemSrc*** is detected, then the input string is recognized as ***constInput***. All characters are taken. It means the syntax is not strong defined. Usual it should contain a number in a standard writing style. Remember that the ***valueCast*** is already parsed and removed from the input string before.

`fb1@pinX[2]`, `@mdlPinY.re`:, `@xref[2].re`: examples to access to an element of the array type or complex value on input.

`[2]`, `.re`:, `[2].re`: Only access an element on input, also in combination.

`123.4:F` Example for a constant cast to float

`M_PI:F+` : `M_PI` is also a constant on input till the + as operator. Because the @ is missing, the identifier `M_PI` is not a ***fblockC*** nor a ***pinC*** . The `:F` is the cast to float.

`@pin{2}` this is faulty, the {2} is report as WARNING, The `pin` is however accepted.

5.3.4 Constant input to a pin

The ***`constInput`*** of a pin is either

* a valid numeric value which's type is detected
* or an identifier which is found as alias in a `ofbAlias` box of the module,
* or it is written as `'text` with a leading apostrophe a manual given literal text, which is used in the target code without change.
* Or it is a string literal also for the target language written as `'text'` with leading and trailing apostrophe. Do not write `"text"`, though the target language needs it. The translator does it.

xxx

5.3.5 Examples for description and type

The chapter *5.3.1 Syntax in colored ZBNF* page *50* has shown the syntax as example for syntax writing. It is complete. But the examples are following here.

```
descrType::=[<*:.{[?descr> ]
     [{{<sliceFB>?,}}]
     [<?elemDst>[[|.]<*:?>]
     [ : <*[?type> [[<sizeArrayType> ] ].
```

name only the descr is used, all other optional parts are not set.

name:Type This is the typical text for FBlocks and pins on FBlocks, setting ***descr*** and ***sType***

nameArray[3] This is ***elemDst*** = **[3]**, to set an array element of the pin with array type properties. Hint: The admissibility of the writing is tested on data type propagation on translation of the graphic, not on input in the graphic itself, valid for all texts.

[0]=:nameArray[3] This accesses array element **[0]**, from the connected input and set **nameArray[3]** . The **[0]** is syntactically **elemSrc** if the input, left from **=:**.

[0]=:[3] Also this is possible, valid and sensible. It can be an expression part input for an expression to set element **[3]** in output, without operator (use the default), accessing the **[0]** from the connected input.

.re=:*M_PI+.m Here ***M_PI+** is the ***descr*** parsed till the dot which introduces **.m**. **.m** is the ***elemDst*** to set the element **m** in an structured output type variable. ***M_PI+** will be analyzed for the expression pin, see *5.8.2 Expression data input pins DinExpr ofpExprPart* page *130*.

It means using a variable **M_PI**, which is non translated used for code generation if a pin **M_PI** is not found in the module, multiplied with the input, and the input is used to add.

fbname{a,b}:Type Definition of a sliced FBlock

fbname2{1..5}: Type also a sliced FBlock with members **name21**, **name22** etc.

pin[2] Pin access to array element

pin.im:f Pin access to an element of its type, here **.im** for imagin part of the complex type. The type is also given here as **f** for float_complex, see 5.4.1 One letter for the base type page 15

This designation with **[…]** after the name is used for sliced FBlocks and accesses to output elements of connections and expressions. It is not the array definition, see *type*.

The ***elemDst*** can also start with a dot as **.elem**, here not shown, see 5.3.5 Complete syntax of Description and type page 12

The given *type* should always written with a colon before. The type is always after the name (or description). This is used also for FBlocks as also for pins with a name or just description and optional a type.

On Expressions instead the name, the expression part description is given here. This contains never a colon **:** and also never **[. ?** (see following). All other character. Especially operators are part of the description. See *5.8.2 Expression data input pins DinExpr ofpExprPart* page *130*.

- The *type* can optional have a *sizeArrayType* part, see in following text.

5.3.6 What contains descr, for expressions and pin designation for FBlocks

As shown in the syntax for `descrType` in *5.3.1 Syntax in colored ZBNF* page *50* The description is written from begin, or after the input string, all till one of `:.{`. It can be a simple identifier for the name of a pin of a FBlock, or it can be the expression for an `ofpExprPart` pin.

The possibilities of `ofpExprPart` is documented in the chapter *5.8.2 Expression data input pins DinExpr ofpExprPart* page *130*.

This chapter should explain some more general possibilities of a pin designation for FBlocks, both for the type definition of a pin as well as for the access. This chapter is separated from *5.5.6.4 The module's output* page *84* and *5.6.7 Possibilities of outputs of FBlocks* page *108* because just both have the same possibilities. That are:

descr as Pin designation for FBlocks:

- Only an identifier is the name of the pin.
- `nameR%`: This defines that pin which presents the return value of the associated event operation. The name should be `eventNameR`, but this is only a suggestion, not necessary.
- `nameR%nameVar`: or only `%nameVar` can be used as pin designation for an FBlock. The right part `nameVar` is the name of the built module variable which gets the return value. The left identifier, the type name of the FBtype pin can be omitted, if the associated event is unique. Details are described in *5.6.7.1 Reference and return output ofpDout() & ** page *108*
- `nameR&%`, `nameR*%`, are adequate for return, but for the designation return by `const` or not `const` reference. For the FBlock pin also `eventR%nameVar` or `%nameVar` should be written.
- `name*`, and `name*nameVar` or also writable as `name)`, and `name)nameVar` designates a pin which is used as reference argument for the event operation to fill on output. The FBlock needs the `nameVar` as name of the built module variable to set with the value (the reference of this variable is given to the called operation).
- `name(` or also writable as `name()` designates, that this is a pin which offers a get operation to access its value. For the access (in a FBlock) the `(` is not necessary to write, if the pin is defined before (not on demand, see 5.6.4 Predefined FBlocks or definition on demand, relation with source code page 67), But for module output pins, it should be written of course, because the pin is defined there.
- `name&(` and `name*(` are variants to access the reference to the data. It means the operation in C/++ language returns a `DType const*` or a `DType*`. This is especially to access structured variables, which can also returned by value writing `name(`.

5.3.7 type and sizeArrayType

type is either an identifier for a user defined type, or one of the one letterr type identifier due to *5.4.1 One letter for the base type* page *58* or also an array type with one letter, for example **F3** for a **float[3]**.

TODO what about pointer types for **struct** and **class** … They are aggregations! All data types are intrinsically values. It means given an association as input is call by reference, given a din as input is call by value. Same for outputs.

After the *type* which can be also an array type, the *sizeArrayType* designation is parsed. This includes also a container, whereby on a din or dout the container management **struct** is given as value, and for associations and ports it is given per reference. But the content of the container is referenced anywhere due to the container's implementation. Usual allocated RAM is used for that. For specific small container implementation for embedded control it may be also a **struct** of data without references.

sizeArrayType is a meta morpheme in *descrType* and defines array or container properties of the type. Syntactically it is:

```
sizeArrayType::= [ [**] <?arrayKeyList>
| [*] <?arrayList> | [] <?arrayFree>
| { <#?sizeArrayType> ? , } ] .]
```

See *5.4.3 Array data type specification* page *60*. Examples:

name:F2: a float[2]. This is not described here syntactically, but a special handling on short characters for the types.

name:float[2]: is the same

name:float[2,2]: is a 2-dimensional array, in C language **float[2][2]**.

name:float[*]: is a container (a List) with float values. The used container implementation depends on the code generation.

General the **=:** designates the pin as input pin. Also a **:=** inside the pin does the same, then the sides are swapped. It is for a pin shown right side in a FBlock. But a **=:** on complete right side or a **.:=** on left side designates an output pin. The mnemonic follows the 'old' assignment operator used in Algol, Pascal and also in the currently Structure text and IEC61499. In Algol and PASCAL there was written:

```
variable := expression
```

instead

```
variable = expression;
```

in the modern languages beginning with C in 1970.

The **:=** may be more obviously, because it gives a direction. The destination is on the side of the **:**. And exact this is used here for the pins. The data flow is always **src =: dest** or just **dest := src**.

On input pins a source post-processing is possible: From the connected source an element can be accessed, and a value cast can be done. This is shown in the examples left/above. The cast starts with **:** and the element access starts either with a dot **.** or with **[**.

The form starting with **@…** is proper if the connection to the pin is not given via graphic, instead via textual description.

If the expression starts left side of the =: with a number, text or other, not with **@** **.** **[**, then it is a constant input. This can be a number, an identifier for any (Macro in C etc) of the target language, or also a **'string'** designation. A variable in the graphic should accessed via **@variableName**.

The designation with **=:** can be omitted if an operator is used anyway for expression inputs, and the input pin is determined by the style or connection style. The both forms

```
:Cast +
:Cast =: +
```

does the same. Also the spaces can be dismissed. Or just, an expression input can contain only

```
+
```

5.3.8 *nrGpos*, order of pins after grave

… `123

The text can end with a grave `` ` `` and a number, This is the pin order number described in 5.5.3 Order of pins page 21. If the grave character (ASCII 0x60) is not following by a number, this text part is not used as **nrGpos**, it is part of the possible **specificDesignation**:

```
…?specificDesignation`123
…?specificDesignation with ` grave
```

The **specificDesignation** can have a special meaning. It is used for example for the event definition of an FBoperation, see 5.9 Operations to FBlocks inside the data flow (FBoperation) page 63. It can be also used for user specific data, in the **OrthBandpassFilter** example used for scope parameter.

All elements are optional. To distinguish an only one identifier between name or type, especially for a GBlock which presents a FBlock or a FBtype (class) you should write

empty

"**:nameType**" to designate it as type or class name. If you only need a value in an FBlock, write "**=value**" whereas the **value** can contain all possible characters. The **connection** must not contain a character **=** because it is the separator to the value, but a connection does not need a "**=**" inside. **name** and **type** are both identifiers as usual in most of programming languages, starting with a letter **A**..**Z** or **a**..**z** or also the "**_**", following by this letters, digits **0**..**9** and the "**_**".

The designation of **ix** and **size** must not contain (but also do not need) a "**]**" inside, so the "**]**" is the delimiter for this both parts. This is a simple and unique syntax.

This is the general rule.

For ix and size, if you have more as one dimensions, or also more as one members for sliced FBlocks, then the separator is the comma. Write "**[2,3]**" for a two-dimensional array with this size. Write "**name[A, B, C]**"

5.4 Data types

Table of Contents

In the *Error: Reference source not found* the input `x:F` is designated as float input with the letter `F`. This is very space-saving but still obvious. Other tools sometimes have only a *“Pin dialog”* where the type can be selected and can optional show the type in the graphic, but then all types destroying the overview. The idea only using one character should be seen as proper, the number of types used are not too much.

This is for the standard usual numeric types. The type of aggregations are determined by the destination class. A type name can be given additionally if necessary.

The problem on numeric and basic types is: There are a lot of designations in different programming languages and usages, but they are similar. A second approach is: Also regard non full deterministic types.

5.4.1 One letter for the base type

IEC61499 and also the automation system programming language IEC61131 knows the following definition of types, See *IEC 61131-3 Second edition 2003-01, Reference number IEC 61131-3:2003(E)*, page 32. The type `CHAR C` was later defined in IEC61131.

```
 ANY                                  A
+-ANY_DERIVED                         L
+-ANY_ELEMENTARY                      E
  +-ANY_MAGNITUDE                     M
  | +-ANY_NUM                         N
  | | +-ANY_REAL                      G
  | | | LREAL  double                 D
  | | | REAL   float                  F
  | | +-ANY_INT                       K
  | | LINT, DINT, INT, SINT
  | | int64, int32, int16, int8   J I S B
  | | ULINT, UDINT, UINT, USINT   Q U W V
  | | uint64, uint32, uint16, uint8
  | +-TIME                           T
  +-ANY_BIT                          b
  | +-LWORD, DWORD, WORD, BYTE    q u w v
  +-BOOL   bool                      Z
  +-CHAR   char                      C
  +-ANY_STRING
  | STRING                          c
  | WSTRING     (not specified)
  +-ANY_DATE                        p
  | DATE_AND_TIME                   t
  | DATE, TIME_OF_DAY              a h
```

Common reference type, used for aggregations between FBlocks, not defined in IEC61499:

```
  +-ANY_REFERENCE                   R
```

Common handle type, a simple number designation without interpretation of the number, not defined in IEC61499:

```
  +-HANDLE                          H
```

The void type for non existing data, not defined in IEC61499:

```
  +-VOID                            X
```

Complex types, not defined in IEC61499

```
  +-ANY_CMAGNITUDE                  m
    +-ANY_CNUM                      n
      +-ANY_CREAL                   g
      | CLREAL   Complexdouble       d
      | CREAL    Complexfloat       f
      +-ANY_CINT                    k
          CLINT, CDINT, CINT      j i s
```

All shown character for this types and also the names can be used for OFB:

- `D F J I S B` that are the standard numeric types which are also known with this same char in Java as return value of java.lang.Class.getName() for the primitive types `double`, `float`, `long` (64 bit), `int` (32 bit), `short` (16 bit) and `byte` (8 bit). They have its adequate in C/++ with `int64_t`, `int32_t`, `int16_t` and `int8_t` for the integers. In IEC61499 they are named `LREAL`, `REAL`, `LINT`, `DINT`, `INT`, `SINT`.

- `Q U W V` are the unsigned typs in C++ `uint64_t`, `uint32_t`, `uint16_t` and `uint8_t`. In IEC61499 they are named `ULINT`, `UDINT`, `UINT`, `USINT`. In Java there is not a counterpart, the larger signed types should be used. The used characters should have their mnemonic in *“Quad word”*, *“Unsigned”* instead `I`=`int32`, *“Word”* usual in some systems for 16 bit and `V`, it is near `W`.

- `q u w v` are the counterparts of unsigned, designated as *“Bit types”* as also in IEC61499 as `LWORD`, `DWORD`, `WORD`, `BYTE`. Distinguish between *“unsigned”* and *“bit value”* is not familiar in C/++ language, both is `uint…,` but it may be proper to distinguish it on user level of an application. In IEC61499 and IEC61131 (sometimes designated as *“safe language”*) it is distinguished. The difference for the OFB usage is: The bit types are not compatible with the common numeric type `N`.

- `Z` is for boolean, the same as in Java `Class.getName()`. What is a boolean, it should be clarified. How is a boolean presented in machine level: This is not a problem of the graphic, depends on implementing stuff. A boolean may be also possible to represent only by one bit in a bitfield. In C++ the `bool` should be used, ans also in C with (for example) a `#define bool int`. In IEC61499 it is named `BOOL`.

- `d f j i s`That are the complex types as counterpart to the real types. Complex types are fundamentally for numeric solutions, but they are not standardized in any language. General this types are structured types. For IEC61499 code generation they are named `CLREAL`, `CREAL`, `CLINT`, `CDINT`, `CINT`.

- `C c` is for one character and a String. Unfortunately the letter s or S is already used for *“short”* and T or t for *“Time”*. Whether a character has 8 or 16 bit (ASCII, UTF8, UTF16) is clarified on implementing level.

- `T` is for a current time (relative) due to the usage in IEC61499 and IEC61131 as `TIME`. How many milli or nanoseconds is represented by one step, it should be clarified by the implementation. It should be the same for all time values for the whole application.

- `t` is an absolute time stamp adequate to `DATE_AND_TIME` in IEC61499 / 61131. The format of the absolute time stamp should be clarified for the implementation. Often it is the seconds after *Jan 1th, 1970* (as in UNIX), or better seconds and nanoseconds after a dedicated base year. It is important that it is a continues value of seconds.

- `a h` is a value of the date only, the day, and the time of day (`h`our) or the question which `h`our. As mnemonic. It is also implementing specific how is it presented in machine code. It is supported also as continues value. For the human interface it is always processable as human readable format, which can also regard time zones etc or country specific presentations. This stuff should not be mixed in a core application.

Beside this one letter type designation also the known type names can be used written in style `int32` in the overview on the left page before. This is the shown designations in IEC61499, but also the here named known designations usual in C/++ or similar programming languages

The generated names for code are depending from the code generation scripts.

5.4.2 Unspecified types

Some FBtype uses unspecified types, because they are available for more or all numeric types, or the type is checked and used really on runtime. In C/++ this is often designated as `void*` also as pointer to basic numeric types. In Java there is the `Object` class as common representation of all types. But the main approach is: The type should be specified by forward or backward declaration in the graphic model by data connections.

- `N` presents any numeric type. This is formally also an unsigned type, whereby using unsigned for numerics is sometimes a prone of error. It is compatible to `D F J I S B Q U W V`
- `n` presents a complex numeric type, compatible to `d f j i s`
- `M` is any numeric presentation, not complex one and not bit values. It is `N T`
- `E` is a non referenced type.
- `L` is a referenced type. In IEC61499 and 61131 it is named `ANY_DERIVED` and distinguished from the `ANY_ELEMENTARY` . It does mean a structured type or also an enumeration defined there with `TYPE … END TYPE`. All of them can be present by an aggregation to a FBlock which contains the appropriate values. The `L` follows the `Class.getName()` in Java for the `Object` type. It is especially any reference type to a class type (a pointer) similar as the `void*` in C/++.
- `A` is a really unspecified type. This is also if the type specifier is not given.

5.4.3 Array data type specification

Arrays with one dimension and a determined length are defined by a simple number after the one-char-type, such as `F3` for a `float[3]` array. This is a concise simple style which needs less space in the graphic. The other possibility is the writing style similar in C/pp or Java: `F[3]`, or `F[2][3]` or `float[2][3]` for a two dimension float array.

Using simple one dimensional arrays is often necessary in FBlock graphics, because several values are calculated with the same procedures. It depends from the implementation whether a FBtype can really process a vector, or whether more as one FBlock is instantiated and called for the vectorized calculation. The graphic should not deal with this implementation detail. For example a FBtype to calculate the complex representation from a 3-phase voltage in a grid has of course an input `:F3` for the three phase values, and hence an output `f` as complex, and also an output `F` for the so named zero sequence value which is often `0.0`.

As unspecific array, also possible as scalar or as container, can be written as `type[]` . Then the data type propagation (see *5.4.6 Data type forward and backward test and propagation* page *63*) determines the array size, or also determines the data type as scalar.

On expressions, using array types means that the expression is executed separately for all array elements. If inputs are scalar for the same expression, this scalar values are used for any of this expression:

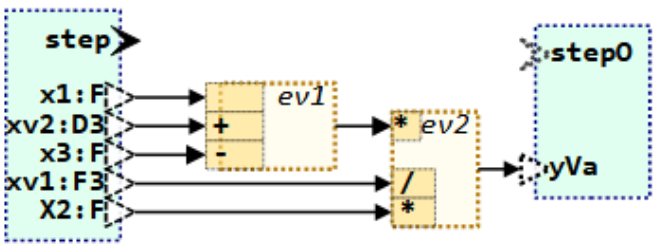


Figure 27: OFB/ExprVect-A.png

Because two inputs are arrays, with concerted sizes, the expression is executed 3 times, and the yVa is also an array output of type `D3` or `double[3]`.

For FBlocks, which are marked with type[] on any input, it is adequate. To select the correct implementation features of C++ with its template concept can be used. But then, the module is oriented on the target language C++ or another language which supports the adequate template concept.

For FBlocks, which's all inputs are marked as scalar (without `[]` in the type), but which are connected to array inputs, this FBlock instance is implemented as array instance and hence also called for any array element similar as for expressions.

5.4.4 Container type specification

A container is known in higher programming languages, for example in Java as `java.util.List` or as sorted container as `java.util.Map`. Also an array with a non limited size is a container.

In UML the `*` is familiar to designate an aggregation with more possible destinations. This is also a quest of container: The aggregation (or also association and composition) has a multiplicity. Whereby the possibility to select exactly between 1.. or 0.. or 0..2 members or such is not supported in this granularity. It is possible also to have an array of a dedicated size also for aggregations. But whether this elements are set or they are nil, this should be checked by the implementation.

- Write a `*` after the type specifier or also on place of the type specifier (`name:*`) it is designated: Any container. The implementing level decides about the implementation of a container. A container refers or contains any number of elements, sorted in order of input. Such a linear container can also implemented by an array in a free size.

- `**` after the type designates a sorted container. The sorting key is implementation specific or specific from the creating and using FBlocks. Often the name of an element is the sorting key (it's a `String`).

- `[99]` after the type designates an array with variable size but possible with a given maximal size. `[]` is a free variable size.

- `[1..4]` after the type designates an array with this possible range of size. It is similar the number of associations in UML

What about more dimensional arrays … should be clarfied in future. Writing style dimensions separated by comma such as `[9,3]` or `F2,3` for an array of 2 element which each 3 elements. All rows and columns have an equal length. It should also be possible to use `[][]`, then the rows and columns or more dimensions can have each any different length, such as arrays in Java language.

5.4.5 Structured type on data flow

A structured type for data inputs and outputs is an instance of a FBtype. This instance comes from the data output provided to the data input. The difference to an aggregation is: The aggregation is a stable connection from one instance to another one, the using FBlock can access the currently data from the aggregated FBlock. For that also problems of data consistence (mutual exclusion on access changed data) should be considerate as known in Object Orientation and UML.

The data flow with instances of FBtype presume constant instances, which are not changed after delivering on the data input. This approach comes from the IEC61499. It is often also used in ordinary programming, but not so obviously. The common solution is: The data are binding to the event instance. Or, the event instance contains the data.

Often, for such approaches, dynamic allocated memory is used. This is the simplest form. But for frequently used dynamic memory the problem of fragmentation exists. In Java Runtime Systems this problem is solved by using the Garbage Collector. Another possible solution is: Using only memory blocks with equal sizes.

The other often simple solution is: Using a pool of event data. The event flow is usual deterministic in amount. It doesn't make sense to shoot around with events. An event should be created (using a member of the pool) only if it can also be processed, and if the pool is empty, there are obviously too much events in queues, not processed, and more events are only disturbing. Hence, the pool of event data is often a possible and proper solution for implementation.

Designation of the data type:

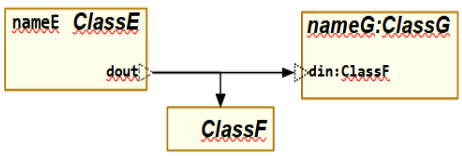


Figure 28: OFB/DflowStructData1.png

The shows two possibilities to dedicate the type of the data flow:

- If you have a connection from a dout or din pin to a class frame of style **`ofbClass`** or to a FBlock frame, style **`ofbFBlock`** without instance name, then this defines the type of the data pin.
- The second possibility is, use the type name after colon.

You can define the data pin type also in an extra diagram:

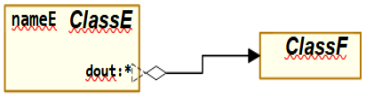


Figure 29: OFB/DflowStructData1.png

Here the connection is used as Style **`ofRefAggr`** which shows the non filled diamond as in UML. Additional for the type an **`*`** is written. This means, as also for other types, The type is a container. Also an array size can be used there, or the **`**`** for a sorted container or **`[]`** for an array of not variable size. This is also possible of course for a immediatelly type specification as in on **`ClassG`**.

5.4.6 Data type forward and backward test and propagation

emtpy

5.4.7 Using a module with non deterministic data types

Data types should be determined on inputs and outputs of the module's pins and on the pins of called FBlocks. They are often not declared on expressions. But knowledge of the data types are internally necessary for all pins for exact code generation. This is solved by the 5.4.6 Data type forward and backward test and propagation page 45

Sometimes a module has not full determined types. For example a math algorithm in a module can be executed at least on controller in float, double or also integer arithmetic. The graphic of the module should not be changed because of this implementation detail. Determining the used data type should depend from the usage in the superior module or from settings from outside given on translation.

Sometimes also the destination language of code generation supports variable data types, for example C++ with the `template<type>` language feature or also C with a well-thought-out system of `#define` data types. But often it is necessary to use determined types for code generation.

For these challenges both are necessary:

a) Omit data types on pins, replace them with a handful of information in prominent places by data type propagation.

b) Use the non full qualified data types if flexibility is necessary.

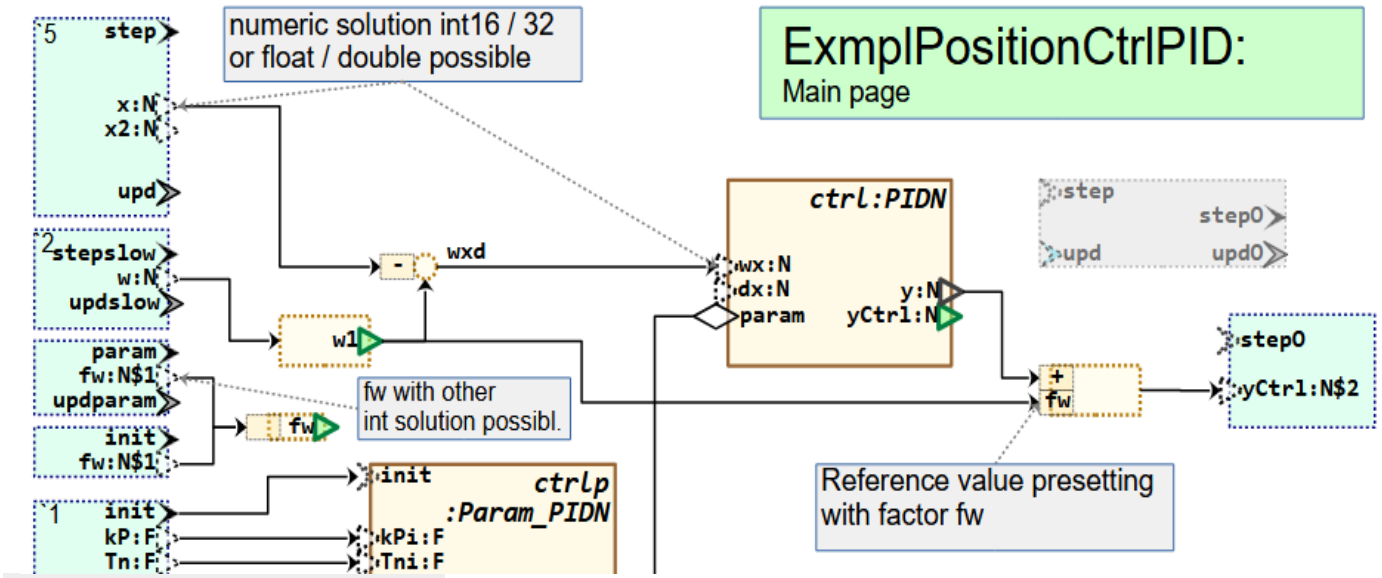


Figure 30: odg/PositionCtrlPID_1.png

The image above shows a simple position control functionality using an PID controller. The PID controller is given as (legacy) C-code implementation, for float and also for integer with 16 or 32 bit resolution. The calling conventions for all three variants of PID are near equal (not exact at all). Hence it is proper to use **only one graphic presentation** for all three (or possible four, also *double*) numeric resolutions. That is expressed in the FBlock using the '`N`' data type (`ANY_NUMERIC` in IEC61499). Also the inputs and outputs of this module are marked with the '`N`'.

The real used type of the PID FBlock (access legacy code) is declared in the `ofbImport` shape:

```
$mdlType=ExmplPositionCtrlPID_%x$step%;
$ifcFile=emC\Ctrl\PIDi_Ctrl_emC.h;
PIDN=PID%wx%_Ctrl_emC :emC\Ctrl\PIDf_Ctrl_emC.h;
Param_PIDN=Par_PID%yMax%_Ctrl_emC;
$ifcFile=Adapt_PIDI.h;
```

Figure 31: odg/PositionCtrlPID_ofbImport

The 3th line defines the alias PIDN with `PID%wx%_Ctrl_emC`. The `%wx%` describes the name of the pin `wx`. To build the used name the Data Type (DType) of this pin in the used situation (depending from the outer Dtypes) is taken to replace the `%wx%` part in this text. Result is here `PIDF_Ctrl_emC` or `PIDI_Ctrl_emC` or `PIDS_Ctrl_emC` depending from the DTypes outside:

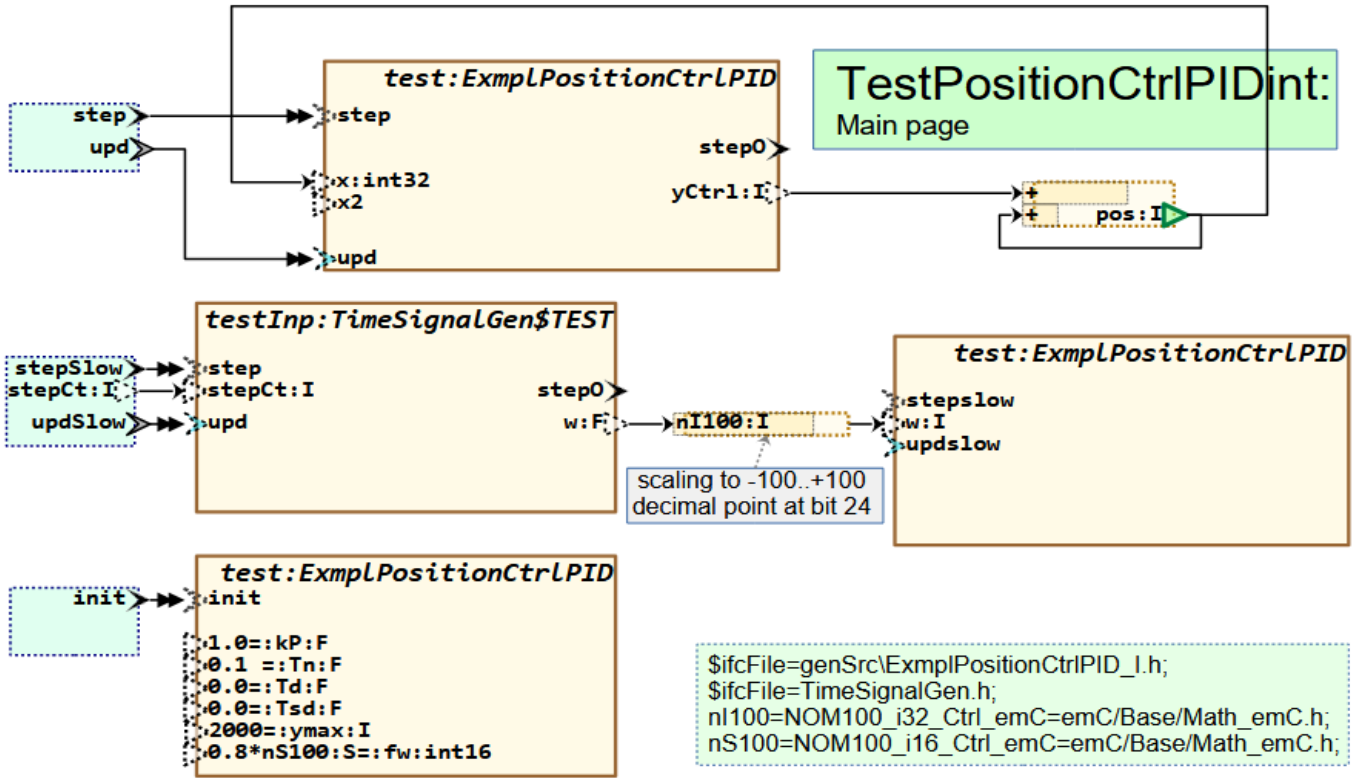


Figure 32: odg/PositionCtrlPID_Test_int32

This is now a usage of the `ExmplPositionCtrlPID` with integer arithmetic. There are three FBlocks for the tested module for the different step times or just events. Above the step is shown, which is a closed loop with a simple increment or decrement of a position `pos` depending on controller output (`yCtrl=0` means the `pos` is not changed).

The inputs of the FBlock incarnation for this non deterministic `ExmplPositionCtrlPID` type module defines the type `I` (or `int32` in C++) on the relevant inputs. Together with the `$mdlType=ExmplPositionCtrlPID_%x%` in the `ofbImport` shape in the module (Figure 18: odg/PositionCtrlPID_1.png) due to the DType of input x with event step, the built module identifier for code generation is `ExmplPositionCtrlPID_I`. Hence from the given module with the name `ExmplPositionCtrlPID` source files are generated: `ExmplPositionCtrlPID_I.c` and `ExmplPositionCtrlPID_I.h` . This files contains the code for the integer variant from the module with non deterministic (`ANY_NUMERIC`) data types.

To do so, all DTypes in the module accesses on code generation the given DTypes on the FBlock inputs. That are `I` on `x`, `w` input and `S` on the `fw` input for the `init` module, last pin. This inputs are designated with `N` and `N$1` in the module `ExmplPositionCtrlPID`. In the module the **Dependency** of the DTypes are important: All non deterministic DTypes without `$1..9` have the dependency designation `$0`. It means **all `N` have the same DType from outer**. This is also valid for derived DTypes (here not used). For example a used DType `N[3]` also writable as `N3` has also the given integer designation `I[3]` or just `int32[3]` for usage.

But the DType designated with `N$1` with another dependency designation is independent of the `$0` (without dependency designation) DTypes. Here it is used with `S` (`short` in Java, `int16`). You can write `int16` or `S`, it is the same.

Now look inside the `ExmplPositionCtrlPID` for code generation. This module is repeated shown here:

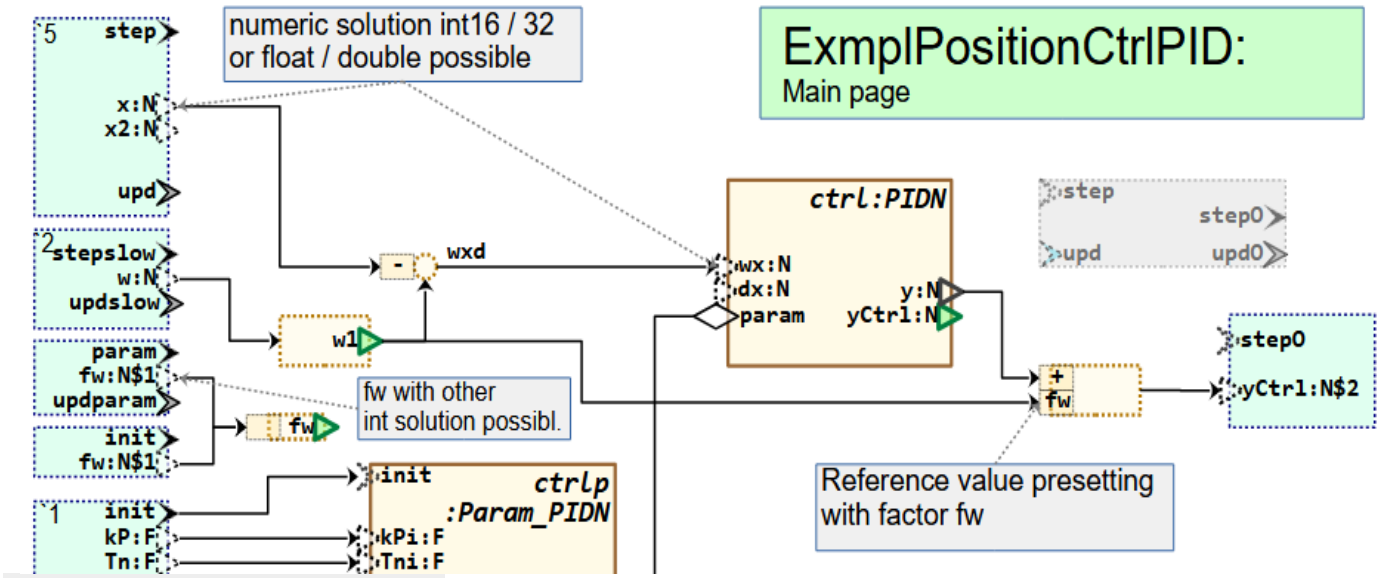


Figure 33: odg/PositionCtrlPID_1.png

For code generation, we know now that `x`, `w`, etc, have the DType `int32`.or `I`, and `fw` has `int16` or `S`. The DType on x is propagated to the `ctrl.wx` as also `ctrl.dx`. More exact the N on input x was propagated with the same internally instance of DType to the modules internal pins. So using `I` or `int32` outside uses automatically also `I` for the `ctrl` FBlock.

As described on page *63* before, the name of the called PID implementation results in `PIDI_Ctrl_emC` in C generated code. But this is not the given (legacy) name of the PID variant for integer, the name is unfortunately `PIDi_Ctrl_emC` respectively `PIDi_Ctrl_emC_s` for the `struct` type itself. That's why a specific header file is also include in code generation for adaption automatic generated code with the rules of this OFB tool to the given legacies. It contains:

```
#define PIDI_Ctrl_emC PIDi_Ctrl_emC
#define Par_PIDI_Ctrl_emC_s Par_PIDi_Ctrl_emC_s
#define ctor_Par_PIDI_Ctrl_emC ctor_Par_PIDi_Ctrl_emC
#define init_Par_PIDI_Ctrl_emC(thiz, Tctrl, yMax, kP, Tn, Td, Tsd, reset, openLoop) \
        init_Par_PIDi_Ctrl_emC(thiz, Tctrl, yMax, 32, kP, Tn, Td, Tsd, reset, openLoop)
#define set_Par_PIDI_Ctrl_emC set_Par_PIDi_Ctrl_emC

#define PIDI_Ctrl_emC_s PIDi_Ctrl_emC_s
#define ctor_PIDI_Ctrl_emC ctor_PIDi_Ctrl_emC
#define step_PIDI_Ctrl_emC step32_PIDi_Ctrl_emC
#define init_PIDI_Ctrl_emC init_PIDi_Ctrl_emC
#define upd_PIDI_Ctrl_emC upd_PIDi_Ctrl_emC
```

It means, using the C internal MACRO replacement, the generated names are replaced by compilation by the necessary legacy given names. This includes also the fact that the PID controller is not given as 16 bit variant. The 16 bit variant has the same data as the 32 bit implementation, only a `step16...` and a `step32..`. core operation should be called, here the `step32..`. variant. Another interesting detail is: The parameter are float also for the integer controller variant. The reason for that is presented in the legacy controller description, see vishia/emc/Ctrl/PIDctrl(www)

5.4.8 Integer Data types and their scaling and decimal point

In embedded control with small processors (less power consumption, cheap) floating point arithmetic is often not on chip, but multiplication with 32 bit fix point may be available. Floating point can be implemented with software operations. To implement controller algorithm the focus may be on using fix point arithmetic. The scaling and multiplication should be clarified and mapped to the graphic.

But there is another reason to use fix point data presentations: The inputs and outputs to real signals are limited in range, and also limited in resolution. It is nonsense to present a position with the value of 1.234567 mm in this fine resolution (1 nm), if the range is for example 2 m. Floating point presentation has no advantage for that. But for the pure mathematics, floating point using is sensible if it is given. Often floating point arithmetic are really faster than adequate fix point arithmetic, which needs sometime additionally shift operations, which is done in floating point by the hardware arithmetic.

There is one reason more to use fix point: Often a double arithmetic (48 bit mantissa) is very slow on embedded controller. 24 bit mantissa length is too less for integration of small increases. For that reason using a 32 bit integer number is better, especially if the integration range is limited, in combination with floating point arithmetic to build the amount of increas.

For imaging a proper range of values on graphical level the fix point format may or should have a decimal point on a determined bit position. For example, a sensible decision is: Use values in range of -100..100 which are the percent value from a nominal presentation. Then it is very sensible and simple to set the decimal point after the first byte, on bit position 24. Then, also in debug mode with hexa presentation of register content, you can simple estimate the value. Any machine code oriented programmer knows, that the value `0x40` is 64, `0x64` is 100, looking on the highest byte. You have also the advantage that you have a sensible overdrive to -128...127.999 which is similar as usual in analog technique.

But also for natural, not nominal values a sensible fix position in bits for the decimal point is possible. In the `ExmplPosiitonCtrlPID.odg` there is a position, measured in mm, in range -2000 … 2000 which are +- 2 m. It is proper mapping to 12 bit, the decimal point is on bit 20.

For that reason the integer types have an additional information about the decimal point In the OFB graphic, and can have also the number of used bits of the integer part. It is written in form `S8.4` or `I.20`. The first examples describes an `int16` value (`s` for Short) with in sum 12 bit, 8 bit for integer, and 4 bits for fractional. it means a value from -128...127.94 is able to present, the highest 4 bits are declared as ‘not used’ (either 0 or 1111 due to the sign). The second example is an `int32` with 12 bits integer and 20 bits fractional, just as used in the position control example. But to be honest, the resolution of 1 nanometer with the 20th bit is not really used.

The syntax of the type text on pins is described in *5.3.7 type and sizeArrayType* page *56*. For type identifier, also a dot is accepted. The parsing of the type is then done in the operation `org.vishia.fbcl.fblock.DTypeFBcl.parseDType(…)` in the code of the translation.

5.5 Modules, Inputs and Outputs, file and page layout

Table of Contents

5.5.1 Modular structure of software

Software and also hardware has usual a modular structure, the OFB graphic with code generation is in mid of them.

* Used modules can be given as ready (or preliminary, not ready) files in the target language (C/++), used with its interfaces (header files), and linked by the compilation tool chain. The OFB graphic adds one or more using target language files after translation to the process.

* Used modules can be given also in the same OFB graphic project. One used module is present as FBtype, as interface definition for the using (superior) module. After translation, both modules, the used and the using, are translated to target language files, and linked together with the target language compilation tool chain.

* Used modules can be given also in another OFB project, translated before or parallel given, independent (as collaboration between different developer teams). It is possible to include the interface definition, the FBtype of a used module via its textual presentation, the fbd file in IEC61499 syntax. The developer teams collaborate with interchange of these fbd files. They can also be created with other (graphic) tools.

* The most superior OFB graphic module can be used via target language code generation in another development environment. It is presented for using in the target language interface (header) and implementation files.

The other approach of software modularity using the OFB graphic is: Use the graphic for the architecture of the software, showing inter communication between modules, first without code generation, later possible after all with code generation. For this approach also hardware can be shown in the OFB graphic.

This chapter describes, how one OFB project with one or more input (graphic) files can be used. The order of the files, the definition of the module interface, the usage, and the rules for translation into the target code are considered.

5.5.2 Module in odg file(s) organized in pages

One odg file can or should contain one module, but can contain also more as one module. It should be possible to distribute one module to more as one odg file (do in future). But then all these files must be processed with one translation step.

Any page must have a shape with style `ofbTitel`:

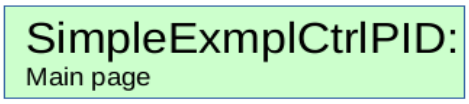


Figure 34: og/ofbTitle-1.png

The first word separated with colon is the name of the module, need to be an identifier. The text after colon is only comment in the graphic. It is not used for code generation or other content evaluation.

If you write a sharp as first character `#Modulename:...`, then this page is commented out, not used for evaluation.

You can have more as one page in one file with the same `Modulename`. Or just more as one file. The pages are count in order of the files and in the file. Pages for one module need to follow one after another. Each page must contain the `ofbTitle` with the module name.

If the page contains an area with style `ofbDisableArea` then all shapes which are inside or only touches this area are not evaluated. This is a simple and proper obvious possibility to deactivate parts of the graphic without removing in the graphic, similar as commented parts in textual sources.

5.5.3 Alias usage as type identifier in a module and OFB project wide

The first page of the module, or optional also other pages can contain an `ofbAlias` shape:

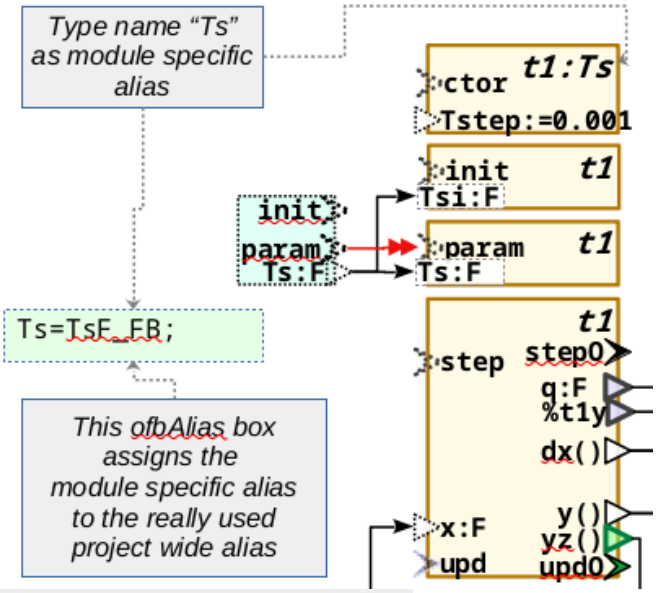


Figure 35: Alias_Mdl_Prj_assign.png

This image shows the usage or maybe the definition on demand of a FBtype, which is named `Ts` in this module. It's very short for *"Smoothing block"* The designation `Ts` is only valid in this module (can be used in the same way in other modules, but does not need). The project wide designation is `TsF_FB`. It is a float smoothing block. `t1` is the instance name.

This simple smoothing block is implemented in C language by the type `T1f_Ctrl_emC` defined in the header file `<emC/Ctrl/T1_Ctrl_emC.h>`. This is target specific, for another target language or another running system environment it can be another designation. That's why the association between the OFB project wide designation (`TsF_FB`) and the target language designation (`T1f_Ctrl_emC`) is contained in a file given as argument `-cfg` for the translator (see *6.1.3 Common script for translation odg to target code* page *212*):. This file contains for this example one entry:

```
TsF_FB = T1f_Ctrl_emC,
h=emC/Ctrl/T1_Ctrl_emC.h;
```

Another aspect of the `ofbAlias` box is the association of the used numeric type to the used name. This example is related to the description of *5.4.7 Using a module with non deterministic data types* page *64*:

```
PIDN=PID%wx%_FB;
Param_PIDN=Param_PID%yMax%_FB;
Tsi=Tsl_FB;
TsF=TsF_FB;
```

Figure 36: odg/ofbAlias.png

The simple used module related type designation is `PIDN`. This comes from the example `ExmplPositionCtrlPID`. The project wide used name ***depending from the type on the input*** `wx` is `PIDF_FB`. if `wx` has the data type `float` or just `F`. And now, the used type identifier in the target language for this example due to the config file `OFB_Presentation/src/Templates_OFB/makeScripts/LibCtrl_emC.alias.cfg` used for translation is:

```
PIDF_FB= PIDf_Ctrl_emC,
h=emC/Ctrl/PIDf_Ctrl_emC.h;
```

The The `ofbAlias` box can contain: box can contain:

```
alias = used_name;
```

associates an alias name, used in the module's graphic, to the used name for OFB data.

```
alias = constant_value;
```

Also usable for constant values.

```
!ModuleComplete;
```

This entry controls, that the interface of the mdoule (its FBtype) is complete after reading the module pages in the first translation pass, cannon changed by FBtype modifications of this module afterwards. This is important for modules which are defined with non deterministic data types or for really completed modules.

```
!FBtypeComplete;
```

This entry controls, that all FBtypes defined inside this module are completed (no more changeable) after parsing this module. It is important for FBtypes with non deterministic data types to prevent faulty deterministic in the FBtype on usage, or also to prevent mistakes.

```
$mdlType=Module_%pin%_name;
```

This entry defines the name of the module itself, here in the example with `%pin%` whereas `pin` is a name of a module's pin. Then its type short designation (one character) is used for this position. If the pin is `float` or `F`, then the module's name is `Module_F_name` for this example.

Each entry ends on the semicolon, also the commented entry in the last line.

5.5.4 Order of modules in the project files,

when to define a module's interface (predef / on demand)

The modules are translated in order of the files in the `-i:path/to/input` argument (see *6.1 Converting the graphic – scripts for source code generation* page *208*. Whereas, a module is pre-translated from graphic with its module pages, but later usages of the module's FBtype can complete some properties, for example defined but not yet implemented operations, or data types. The translation to target code will be done in a second pass after the first pass pre-translation.

A module's interface is a ***FBtype***, the type to a used ***FBlock***, or a specific defined FBtype. A FBtype is a ***class*** in UML wording. The image right shows a used and here completely defined "smoothing FBlock" which is implemented in C language as given or "legacy" target code.

A module interface is a well defined FBtype containing in a 'definition' module. This FBtypes can be the interface also to immediately programmed target code, as well as to later defined module content. It is also able to see: A defined module content in graphic can be used later or anywhere other in form of ready to use target code, without knowledge of its implementing graphic. This allows dispersing of work and know how. The module's interface in form of a FBtype is the same in all situations.

5.5.4.1 First define a module, then use, or first use, define the FBtype only

There are two general different approaches for definition of modules and their usage in graphic programming:

a) First define a module completely. then use it.

b) Define only the interface to a module, maybe actually on demand wile usage, or with predefined FBtypes, and later implement the module.

For ordinary FBlock graphic tools usual the variant a) is only given: A module to use must be completely defined, usual in a library module, or in prior to the using module defined modules.

But for practical work and also for thinking in software architecture this strong approach a) is not practical. Any module can be used as ***"Black Box"*** with knowledge of only its interface.

The interface is the FBtype presentation of a module. Indeed also an only partially defined module can be used. This is also usually for working with C/++ header and implementation files: For usage another compilation unit only the header file is necessary. And also for languages, which does not know the separation into header and implementation, for example Java, or for Pascal language of the 1980th, this approach is used: In at least two translation passes, first all interface relevant parts are read and translated from the source file, then this information can be used to translated also the other sources in a second pass with cross knowledge of all interfaces. The separation in *"Header"* and *"Implementation"* in C language is intrinsically only a concession to the capability of compilation tools of the earlier 1970th. It was not possible to compile a lot of files one after another, with storing the interface parts only, then read the files again.

See an example in OFB graphic from `OFB_Presentation/src/BasicTest/odg/BasicTest.odg`:

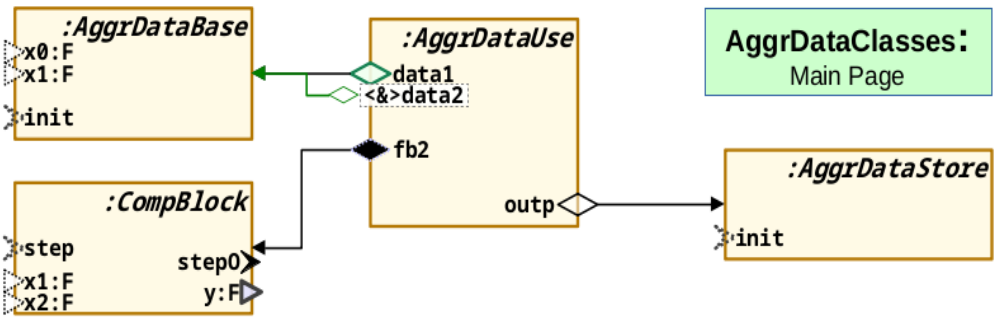


Figure 37: OFB/ExmplAggrDataClasses.png-1

This is like a class diagram in UML, it shows only FBtypes and there usage.

This module defines properties of the FBtype or class `AggrDataUse`, with its three aggregations of the definite type of the destination class, and one composition.

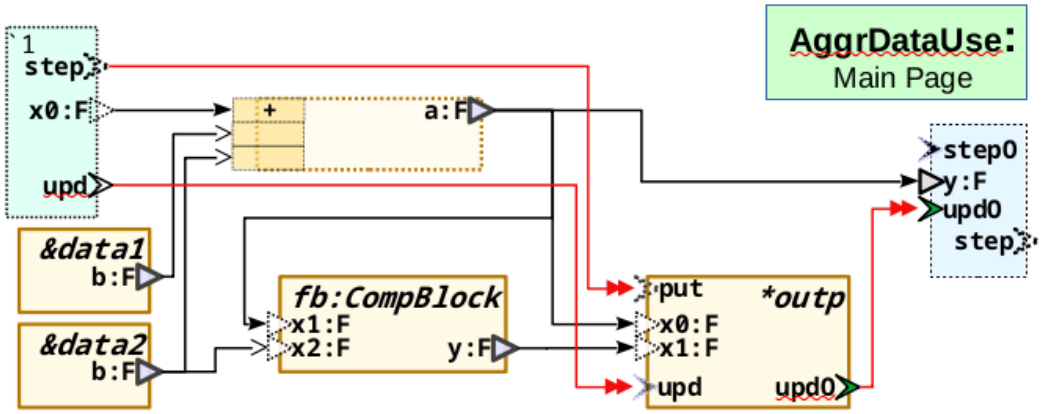


Figure 38: OFB/Exmpl_AggrDataUse-short.png-1

This next page (image left side) contains already the implementation of this module. Because the FBtype of the aggregations are clarified, and also the aggregation pins are clarified, it is not necessary (but possible) to repeat this pins in the module implementation. Compare also the explanation for using aggregations with proxy on *5.5.8 Aggregated modules, associations and composite* page *94*.

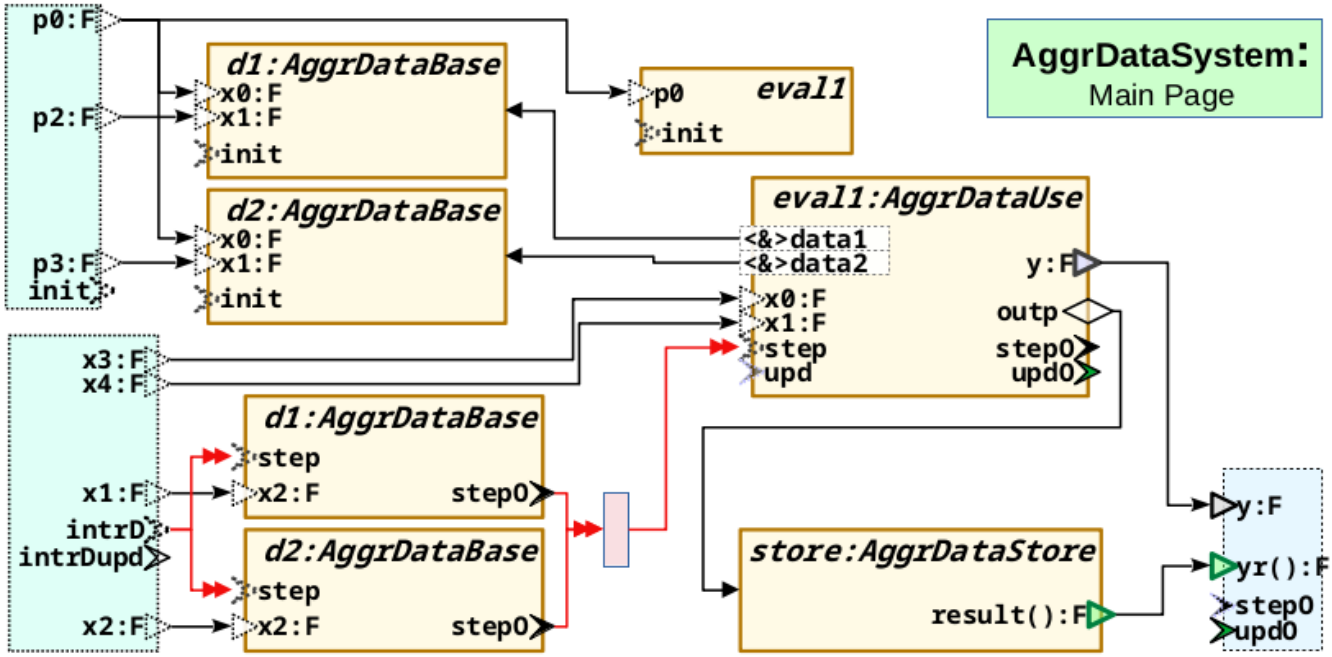


Figure 39: OFB/Exmpl_AggrDataSystem.png-1

This next page shows the usage of the module `AggrDataUse`. But here two additional features are added: first, the `init` has an additional input `p0`, beside the inputs to set the aggregations. The next is a second input `x1` for the step operation, which is not mentioned in the class diagram nor in the module implementation. ***This in an example. In practice, the two inputs may not be considerate in the module's implementation, because yet the functionality is in development, not completely finished. But the usage knows, a second input is desired for future enhancements, hence drawn already.*** This using module for `AggrDataUse` may be also presented before the implementation of `AggrDataUse` is contained in the graphic files, but it should be also possible to arrange it after the implementation of the `AggrDataUse` module. Both should be possible. The interface can be enhanced also without yet given implementation. If an output is given additionally, it delivers 0. An input is yet unused. That's possible as intermediate version (and may be also in a delivered version).

This behavior can be disabled by the entry `!ModuleComplete;` in an ofbAlias box in the module graphic, see *5.5.3 Alias usage as type identifier in a module and OFB project wide* page *70*.

That is supported by the three-phase-translation of the graphic:

* First pass: Reading the content of all modules, define all used FBtypes. Data type propagation to fulfill not completely given data types, for example the `p0` in the image above is `F` or float as data type because of the input. Reading the content of the module, define the module interface to the module's FBtype.
* Second pass: Data flow to event chain process, complete the module, regarding all also later defined changes in FBtypes.
* Third pass: Translate to target code.

The second and third pass may be combined together, but they are separated because of internal reason.

Look on the generated target code files, for the effectiveness of the later defined completions of the module's interface, your will find these additional arguments in the operations.

5.5.4.2 Definition of FBtypes for given target language implementation code

The given target language code can be legacy code, maybe adapted for some reasons by macros, deterministic manual written target code (for example for hardware access or because it is better to overview), or also generated code from other OFB projects or other tool chains.

There are well-founded reasons for manually programming certain parts of the software. That are sometimes hardware oriented parts, and also core functions that are easy to understand in the target language.

The right side image shows a smoothing FBlock which is given in C language in the following form:

include:../../src_emC/cpp/emC/Ctrl/T1_Ctrl_emC.h::T1f_Ctrl_emC::45

```
/*@CLASS_C T1f_Ctrl_emC @@@@@@@@@@@@@@@@@@@@...

typedef struct T1f_Ctrl_emC_T {
  float Ts, Tstep;
  float dx;
  float q;
  float qz;
  float fTs;
} T1f_Ctrl_emC_s;

extern_C T1f_Ctrl_emC_s* ctor_T1f_Ctrl_emC...

extern_C bool init_T1f_Ctrl_emC(T1f_Ctrl_e...

extern_C void param_T1f_Ctrl_emC(T1f_Ctrl_...

extern_C void paramExp_T1f_Ctrl_emC(T1f_Ct...

static float step_T1f_Ctrl_emC(T1f_Ctrl_em...

static float get_dx_T1f_Ctrl_emC(T1f_Ctrl_...


//tag::step_T1f_Ctrl_emC[]
static inline float step_T1f_Ctrl_emC(T1f_...
  thiz->dx = thiz->fTs * (x - thiz->q);   ...
  thiz->q += thiz->dx;
  return thiz->q;
}

static inline float y_T1f_Ctrl_emC(T1f_Ctr...
  return thiz->q;
}

static inline float q_T1f_Ctrl_emC(T1f_Ctr...
  return thiz->q;
}

static inline float yz_T1f_Ctrl_emC(T1f_Ct...
  return thiz->qz;
}

static inline float qz_T1f_Ctrl_emC(T1f_Ct...
  return thiz->qz;
}

static inline float dx_T1f_Ctrl_emC(T1f_Ct...
  return thiz->dx;
}

/**If upd is executed before step (or afte...
 * then y is the value on start of step time.
 * q may be not used if strong step/upd co...
 */
static inline void upd_T1f_Ctrl_emC(T1f_Ct...
  thiz->qz = thiz->q;
}

static inline float get_dx_T1f_Ctrl_emC(T1...
  return thiz->dx;  //regard step is execu...
}
```

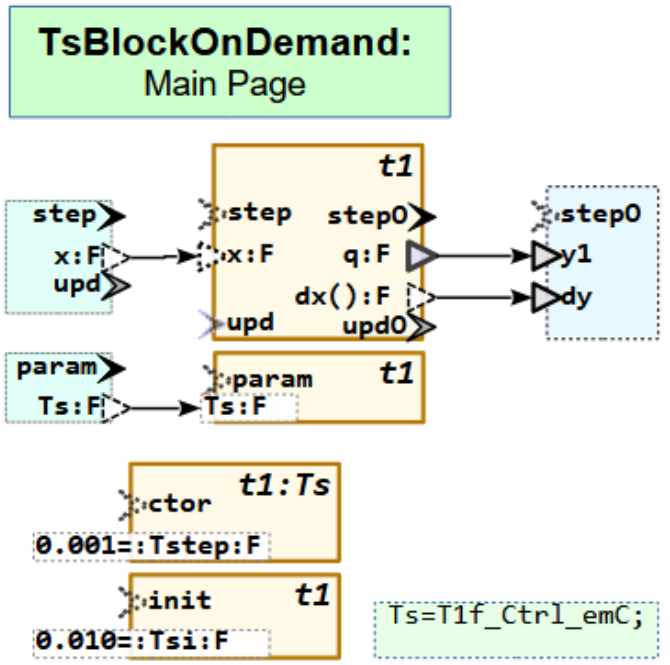


Figure 40: odg/TsBlockOnDemand.png-1

This is a ***definition on demand***. The OFB graphic regards the given target code, uses the same identifier names and types.

* Operations in target code are presented by the events.

* Output pins which are simple `ofbVout` or `ofbZout...` are presented by variables in the `struct`.

* Operations as getter are pronounced with the `()` in the graphic, see *5.5.6.4 The module's output* page *84*.

* Not all parts given in the legacy target code should be present, only the used ones.

* The target language should be matching to the target code generation rules, see *5.13 Drawing and Source code generation rules* page *200*, adaptions in target language with macros may be used if it is not matched.

But the definition on demand may prone for errors while using module specific. Hence it is better to define the target language given FBtypes in an extra module in OFB, which defines only parts of given target language stuff. This can be administrate by a responsible developer, who knows all details. This is done for exact this FBtype in the following form, contained in the `OFB_Presentation` in the file `OFB_Presentation/src/Templates_OFB/odg/LibCtrl_emC.odg` right side in the following form, containing all given features in the target code. Here also a non data type deterministic variant is defined, because effectively this FBtype exists also for int32 and int16 in the target language C, also as C++ class.

This is the ***predefined variant of FBtypes*** for given target language modules.

This module to define the FBtypes is marked with `!FBtypeComplete;` in `ofbAlias` box. It means that the FBtypes here defined are fix, complete or *"freezed"*

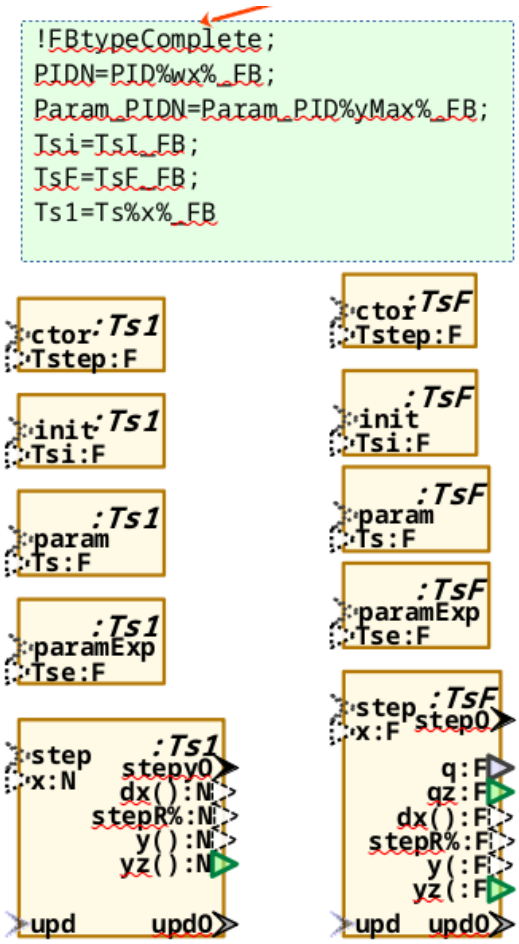


Figure 41: OFB/Lib_Ctrl_emC-Ts1.png

5.5.5 Import or include header files in target code

5.5.5.1 Used external FBtype and the configuration file for header files

The target code needs usual an import of the used modules, in C/++ language this is the `#include` of header files.

As already mentioned in the chapter before, any used FBtype in the graphic has a maybe module specific alias designation, which is translated to the OFB graphic specific designation with the `ofbAlias` box. But the real header file names in code generation for the FBtype names are determined by a textual configuration file for the translator. It is given with the argument `-cfg:path/to/file` The detail description for this file is contained in *6.1.5 Handling of include in C/++ or import and real used type names* page *224*.

That file clarifies the used name and the necessary import / #include in the target language. This configuration file given with the `-cfg` argument can contain for example a line:

```
TsF_FB = T1f_Ctrl_emC,
h=emC/Ctrl/T1_Ctrl_emC.h;
```

It means, if a FBlock with the OFB width type `TsF_FB` is used in the module, (maybe only designated with `Ts` in the graphic module, but translated to `TsF_FB` with the `ofbAlias` box of the module), then the header file

```
#include <emC/Ctrl/T1_Ctrl_emC.h>
```

is included in the target code's header file for the module, to use the `struct T1f_Ctrl_emC_s` or `class T1f_Ctrl_emC` in the module's `struct` or class definition.

If a FBtype is used and maybe mentioned in this config file with a target specific alias translation, but without a header file contained in this line, then no include is generated because of this FBlock. This is sensible if other FBtype are used, which creates anyway the necessary import or `#include`, but nevertheless it is recommended, that any FBtype name has a header file assignment in this config file.

If one header file is selected to include, it is not included more as one time because of another FBtype which is assigned to the same header file.

This is one source of the import or just `#include` statements.

5.5.5.2 Used FBtype, translated as module in the same graphic assembly

The other source of import or `#include` is: Use a FBtype which is translated as module in the same project, means in the same file or another file of the same project, which contains several modules. If this module has not non dedicated data types, hence it is unconditionally generated as target language source or will be generated in this translation step, then its module name, which is also the name of the output target file (header and implementation) is used to import or `#include`.

```
#include "ModuleName.h"
```

The header is generated in the same directory as the used header, hence a path before is not necessary.

Internal hint: The used FBtype is found in >>`Prj_FBcl#idxModuleRdyToGen`

If the included module is defined with non deterministic data types (see *5.4.7 Using a module with non deterministic data types* page *64*, then that module is translated only when it is used in a defined data type environment to make it deterministic. Then also the module name depends on the used deterministic data type, and this name is used for include. For example

```
#include "ExmplPositionCtrlPID_F.h"
```

for the example `ExmplPositionCtrlPID.odg`. The intrinsic module name is `ExmplPositionCtrlPID_%x%` defined in its `ofbAlias` box, the data type of the input `x` determines the `F` (for `float`).

Internal hint: >>`Module_FBcl#idxMdlIncluded`, >>`WriterCodegen#iterImportHeader(...)`.

empty

5.5.6 Module pins

Table of Contents

The image above is a test example for module and FBtype pin capability. It is used here to explain the module pins topic.

Module pins should be contained in a shape or graphical block (***GBlock***) with the style `ofbMdlInp` respectively `ofbMdlOut`.

Module inputs should be contained in a `ofbMdlInp` GBlock. It should contain **input data** with the pin styles `ofpDin...` or `ofpVin...`, or as simple rectangle pin only `ofPin` or `ofPinLeft` / `...Right` with the proper textual pin designations `=:` and `=!` see 5.2.4 Pin styles, ofp page 38.

Similar it is for **Module outputs** The module's output should be contained in a `ofbMdlOut` box, which is shown with light blue in this formatting style in the image left.

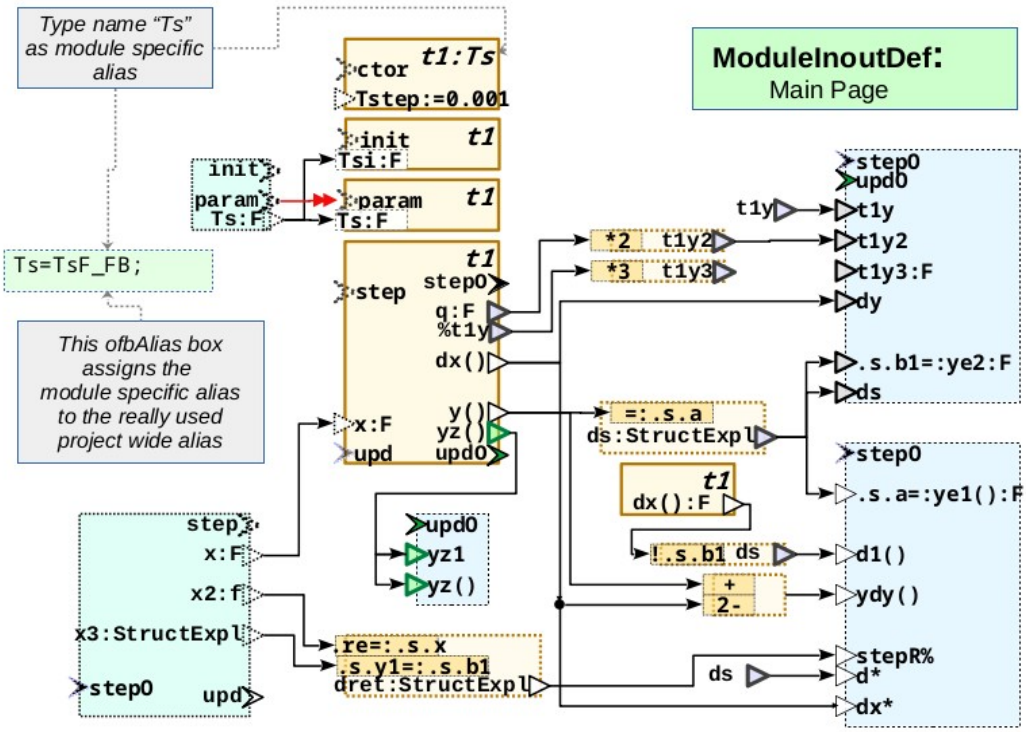


Figure 42: OFB/Exmpl_ModuleInoutDef.png

The pin styles should be the output pin styles `ofpDout...`, `ofpVout...` and `ofpZout...` or also the simple `ofPin` or `ofPinLeft` / `...Right` in a rectangle pin box with the proper textual pin designations `=%`, `=&` and `=!` see 5.2.4 Pin styles, ofp page 38. Module outputs can have more functionality, see 5.5.6.4 The module's output page 72.

If the module contains FBlocks which have `ctor` or `init` event inputs, ***then automatically***

`init` and `ctor` also ***for the module will be created*** without the necessity that it is drawn. It means the module's interface FBtype gets this event inputs if necessary.

5.5.6.1 Events and appropriate Data in the Module interface

A module input box `ofbMdlInp` must also contain at least and usual one ***appropriated event for the input data*** if the `ofbMdlInp` defines data, beside possible the correspond EvUpdin and the related Evout to assign it, see next chapter.

With the assignment of data to events the data are belonging or ***appropriate*** to this event, or in other words, it builds the arguments to the event operation in the order given from top to down. Or, for event queue oriented processing, this data will be stored with the event in the event queue. The data are assigned to the Evin, not to the EvUpdin, see next chapter.

The assignment of data to the prepare input event (here `step`) and also corresponding to the update event (here `upd`) in the module's pin block is essential for **build the event flow due to the data flow**. The event flow is first build for the prepare event, but all reached FBlocks are associated then also to the given update event, if they have an update event input.

A relation between the prepare (`step`) Evin `ofpEvin` and the related Evout `ofpEvout` in the `ofbMdlOut`, if the data flow triggers immediately the Evout after processing the data. This is a ***data flow through***. This is not so, if the module contains a state machine. Then the state machine behaviour determines when an Evout comes.

For the data flow through behaviour without state machine, the Evout should be related to the belonging Evin. This can be either clarified by an event connection between Evin and Evout in the graphic. ***But it is possible***, more obvious and more simple ***to draw this assignment with the related Evout in the*** `ofbMdlInp` ***GBlock which contains the related Evin***, or also draw the related Evin in the `ofbMdlOut` GBlock which contains the Evout.

The output GBlock must also contain one correspond event Evout for the outputs. If update outputs are given (`ofpZout...` or `=$`), then the correspond EvUpdout must be given.

At least one `ofpMdlOut` GBlock must contain both, Evout and EvUpdout to assign it together.

It is possible to draw the correspond Evin in the `ofbMdlOut` GBlock instead the Evout in the `ofbMdlInp` GBlock to assign both together.

In the shown example without state machine, it is clarified that the `stepO` follows the `step` for the interface already, because `stepO` and `step` are in the same `ofcMdlInp` GBlock. This is given without consideration of the inner content of the module.

The update evin is never related to data, the update event operation have no data arguments. It updates the internal data. But updating delivers also the Zout data on the modules output. This data are used from the evin of connected modules, not from the evUpdin.

Also the `init` and `param` Evin are given, but there is no counterpart as `ofpEvout`., `initO` and `paramO` is not existing. If parameterize is done, it does not need any more action for this example. But commonly a `paramO` may be existing. This module has no `ofbMdlInp` GBlock for the `ctor` drawn. But because the module has FBlocks with `ctor`, as shown, the `ctor` Evin (means a `ctor` operation) is created though it is not drawn. The same would be done for `init`. An `ofpEvin` for `init` is here necessary, because `init` has appropriated data.

hint: The data input are of Java-Type >>`Din_FBcl` for the FBtype view of the module, but of Java-Type >>`Dout_FBcl` as representation of the inner module pins, they are mirrored. The Kind of the data are >>`PinKind_FBcl#DinMdl` or >> `PinKind_FBcl#VinMdl`. The Evin are >>`Evin`_FBcl for the FBtype and >>`Evout_FBcl` from inner view, with the adequate >>`PinKind_FBcl`, see Javadoc.

5.5.6.2 Update event and Thread in the Module interface

Events plays a fundamental role in the OFB graphic, as presented in chapter *4.4 Event-Based Execution* page *22*. The events for the module are defined at the module pin boxes or GBlocks of style `ofbMdlInp` and `ofbMdlOut`. Even the appropriate data for each event of the module are defined there.

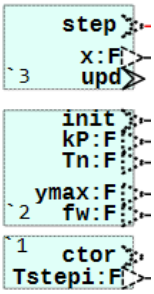
An **event as module pin** presents an **operation**, ready to execute if the event comes from outside. The functionality of the **event operation** is built by the inner FBlock design on the module related to this event. The **data** appropriate to the event **are the arguments** of this operation.

What about the **update event**: The concept is explained in chapter *5.13.2 Life cycle of programs in embedded control: ctor, init, step and update* page *201*. The **update event operation** copies the data from the last execution of the corresponding event operation(s), (from the step time before for controller algorithm) in the so named Z-variable. The term *"Z-variable"* is derived from Z-transform theory, in which the values from the previous calculation step are referred to as the *"previous value"* with designation `z-1` or or `1/z`. In Simulink ((R) Mathworks) this is the *"Unit Delay"* standard library FBlock.

What about **Thread** determination: This indicates several **event operations** that are **intended to be executed in the same thread**. *"Thread"* means one thread in an Real time multithread Operation System (RTOS), as even one interrupt and the main loop in an simple execution system on a controller. Thread ownership is closely linked to the concept of mutual excluded access (mutex) to data. Events which are corresponding to the same Thread can access data without mutex mechanism because they should never be executed concurrently. But this is a contract to the usage from outside. It is not assured that the access is never done in a faulty thread, it can be only checked with certain algorithm built in in the event operations. In opposite, it means, event operations corresponding to different threads should use, and hence generate mutex access algorithm.

Well then, data appropriation to events and the corresponding Thread and the update operation is clarified with the module interface pin arrangement in the module interface boxes `ofbMdlInp` and `ofbMdlOut`.

Figure 43: ExmplCtrl / fbMdlInp_StepUpd.png



This is a simple example for three module input GBlocks.

The designation with `` `1 `` ... `` `3 `` determines the order of pins (of GBlocks) in the target code. First comes the `ctor`, which has one data input. It is the constructor called on startup, without output event and without update event. It is called ones only. Similar it is for `init`. The `step` is the cyclic operation, with `x` as input and the `upd` pin for updating the values from the step time before.

The complete module **example right side** shows three prepare events: `step`, `angle` and `param`, which are all associated to the same thread. This is expressed by the common `upd` pin as `>>PinKind_FBcl#evUpdinMdl`, or with style `ofpEvUpdin...`. Let explain why, following the example:

It shows the preparation of 3-phase electrical values (voltage or current). The `angle` event delivers a reference rotating angle, first set in the same thread, which may be a fast interrupt. For such applications a step time of 50 µs is recommended, possible faster. The reference angle is stored in a simple variable `g`.

Later, but in the same thread comes the step event with the input 3-phase values. First there are transformed to a complex value, which is named *"Cleark transformation"* (Edith Cleark, 1983..1959). Then the reference angle is used, both are transformed with a complex multiplication with the `vrback` operation in to a non rotating value `Vab`. This is the *"Park-Transformation"* (Robert H. Park, 1902..1994), which is the base of the so named *"Field Oriented Control"*.

But also the third prepare event `param` corresponds to the same thread, though it is more rarely executed because of calculation effort and it's non necessary to tune this value in any step time. It is an adjusting of the angle because of possible measurement delays.

The “trick” to do so is: ***Reduction ratio of execution.*** This algorithm is executed in the same fast thread (interrupt) but not in any step times, instead for example only each 16th or 100th step time. In the other step times other longer running stuff can be arranged. With this “trick” the data consistence is ensured at all times without the effort of data synchronisation with mutex mechanism as usual RTOS, which is not possible for interrupt execution. Hint: Even Simulink ((R) Mathworks) uses this capability.

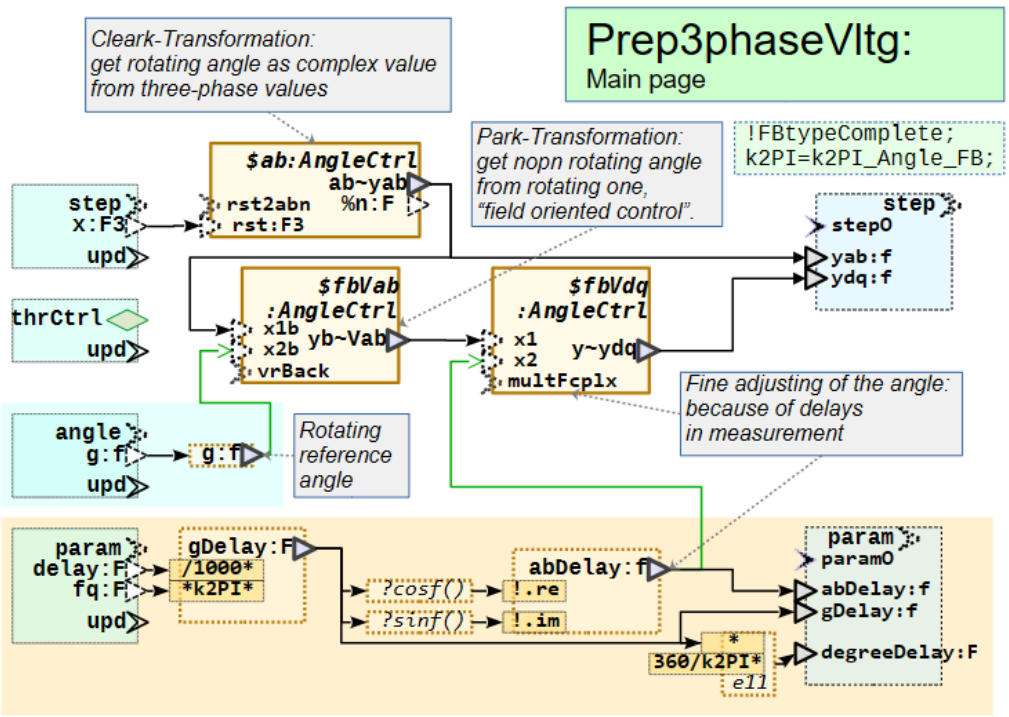


Figure 44: ExmplCtrl/Prep3phaseVltg

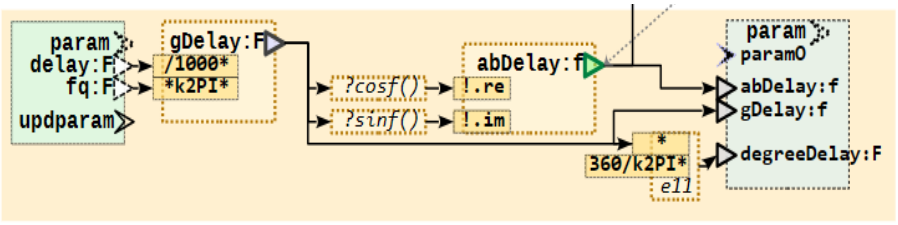


Figure 45: ExmplCtrl/Prep3phaseVltg_extraParam

The alternative possibility is: Execution of the `param` operation in another thread, or in the back loop, and use an specific `updparam` with a Z-variable. Either the appropriate `updParam` operation should be executed with *“interrupt disable”* for the fast thread, it cannot interrupt the set process of the two related parts of the complex value (it is a form of atomic write). Or better execute the `updParam` operation as part, on begin or end of the interrupt itself if necessary. The `param` prepare operation can be executed independently in a slower thread, possible interrupted by the fast thread of `angle` and especially `step` and angle, without conflicts.

Look mindful: The data connection between the separated event operations are drawn with a `ofcDataGet` style, here drawn in green. This breaks the elsewhere faulty automatic data flow to event flow propagation. The `ofbMdlInp` box with `thrCtrl` is not necessary but possible to get a reference to some thread information for managing of special cases.

As shown here, the **update operation is responsible to** all, possible **more as one event operation**, executed before the first event operation for one thread is started, it means it **is thread related**.

The update event operation forwards data from the inner Zout variable or FBlock pins to its `ofpZout` pins of the `ofbMdlOut`. This data can be used beside the current `ofpEvout` related data for the data connection of the module outside. The Zout data are the data of the last step time (thread execution), delivered possible with current data on non `ofpZout` pins valid with the appropriate `ofpEvout` in the module’s output box.

The `ofpEvUpdout` pin on a modules `ofbMdlOut` is only necessary to forward Zout data from the module’s output to the module’s output of the superior module, to use in the level above, in there event operations.

The relation between EvUpdin and EvUpdout is given by the relation of the corresponding Evin and Evout and the relation between Evin and EvUpdin and Evout and EvUpdout.

5.5.6.3 The Module's Input

5.5.6.3.1 Usage Din, call by value for scalar and structured data types

Each GBlock for `ofbMdlInp` should contain usual only one event or the prep or step event and the appropriate update event, style `ofpEvin...` and `ofpEvUpdin...` if this Module Input defines also the appropriate input data. **If more as one `ofpEvin` is used**, then the data are related with the same name to all listed event inputs. This is for example sensible for data for `init` and `param`. But only the data listed here are appropriated. More data can be different for e.g. `init` and `param`.

Using the symbol `ofpDin…` in a module's input GBlock with style `ofbMdlInp` is the simplest form. This pins are translated for target code generation to formal arguments of the operation, and hence actual values on call, also as result of an expression.

If the data type of the pin is a structured type, then a call by value is done. For example even a complex numeric type is a structured type. The following image is a part of the `BasicTest` - module `ModuleInoutDef`, also shown on page 78 on start of this main chapter:

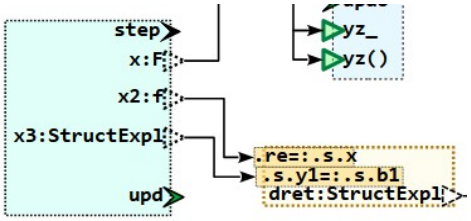


Figure 46: BasicTest/ ModuleInout Def_Inp.png

The pin `x2` is designated with the type char `:f`, which is a complex float. The input `x3` is the named structured data type. The generated code for the forward declaration of the step operation is:

```
StructExpl_BasicTest_s step_ModuleInoutDef
( ModuleInoutDef_s* thiz
  , float x
  , float_complex x2
  , StructExpl_BasicTest_s x3
  , StructExpl_BasicTest_s* d
  , float* dx
  ) { // ##oper_step
```

whereby `d` and `dx` are for output references, see next chapter.

On usage for scalar inputs even an expression can be connected. But the input with structured data type needs a variable connected. This is shown in the usage module `ModuleInoutUse` shown in the following image:

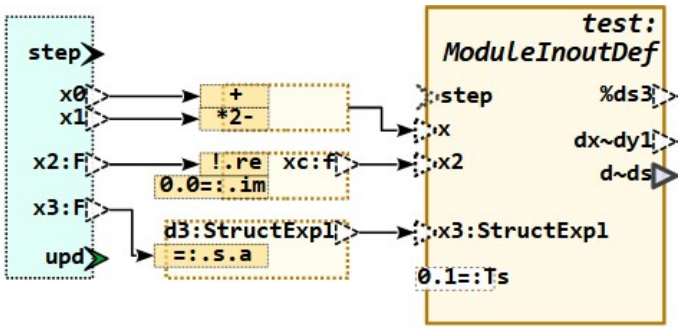


Figure 47: BasicTest/ModuleInoutUse_Inp.png

The generated code of the call is:

```
ds3 = step_ModuleInoutDef(&thiz->test
, (x0 - (x1 * 2) ), xc, d3, &thiz->ds, &dy1);
```

The expression appears immediately in the argument value. Whereas for the complex and structured data the e.g. local variable after expression (`ofpDout`) is connected, also a module input can be used, which is also a local variable in generated target code. The argument is delivered on call as *"call by value"*, which results usual in an internal `memcpy`.

5.5.6.3.2 call by reference

It is interesting for such situations also used a call by reference. In this case the inputs `x2` and `x3` of `ModuleInoutDef` are intrinsic associations in UML wording. The assignment of the structured variable in the module delivers the reference to this instance in the calling environment as usual. The access from the inner of `ModuleInoutDef` is adequate the access to an associated instance, to its inner data in this case (commonly also an access via operations is possible, but not for the here given simple data `struct`).

But this feature should be done in near future (2025-07).

TODO:

also with the adequate `ofp...Left` styles, but the pins are usual right. For a simple rectangle box for the pin with style `ofPin` the pin should be marked with `=:` on end, or if left assigned starting with `:=` instead `ofpDin`, and adequate `!=...` and `...=!` for `ofpVin`, means `name:DType=:` as Text in the pin. Din pins are used in target code as arguments of called operations, or hence as calling arguments for a stored event, whereas `ofpVin` data are used as (public) variable in the module's data on target, as `struct` or `public class` member in C/++.

5.5.6.3.3 Input variables as global variables to set

Using `ofpVin` for input pins creates simple global variables in the data of the called FBlock. Using this capability may be an interesting and simple feature. But this is not Object Oriented and disregards rules of data encapsulation. But often used in pure C programming.

From event driven view: It gives the possibility that one event sets data, which are processed with another event. The data to event association is more freewheeling, sometimes desired. But exact from this reason it is not a good style. It is the counterpart to access immediately the output data.

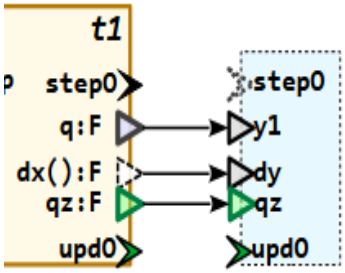
This feature, using `ofpVin` pins, is just not implemented (2025-07), but possible to implement if necessary.

5.5.6.4 The module's output

5.5.6.4.1 Using public variable for the output

This is the simplest form to access data and usual in embedded control for simple algorithm in C language. Also possible in C++ classes or other languages. using public variable. It is not recommended for a strong safety code, because the access to inner variables is possible by manual written code, which is not automatically checked by the standard compiler. But it is a usual approach.

Figure 48: odg/MdlOut PublicAccess.png



Public variable for outputs are simple `ofpVout` or `ofpZout` variable with a simple name in a `ofbMdlOut` GBlock.

The image left side shows two `ofpVout` and one `ofpZout` connected with a FBlock, For `y1` the value is copied from the t1 inner data. For `dy` the value is gotten and stored via an operation, and `qz` is also copied.

The inner code looks like:

```
void step_TsBlockOnDemand ( ...) {
  thiz->mEvout_step |= MASK_step_step0;
  thiz->y1 = thiz->t1.q;
  thiz->dy = dx_T1f_Ctrl_emC(&thiz->t1);
}
void upd_TsBlockOnDemand ( ... ) {
  thiz->mEvout_upd |= MASK_upd_upd0;
  thiz->qz = thiz->t1.qz;
}
```

The access code to this module with name `test` looks like:

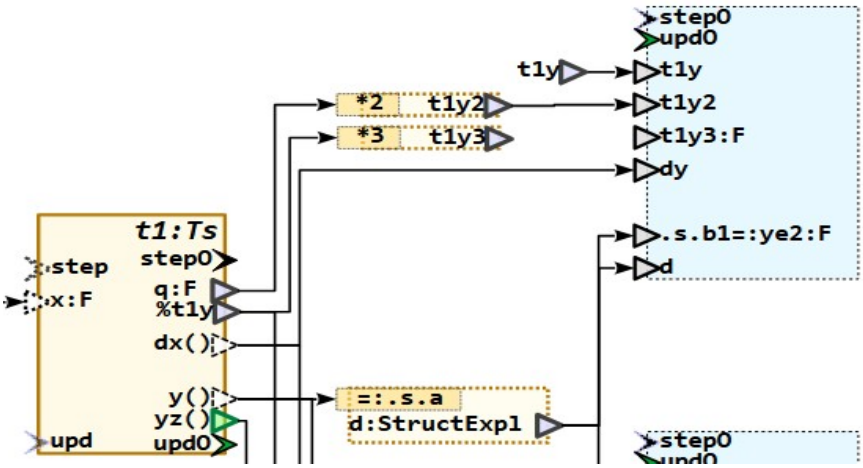
```
  .... + test->y1 + ...
```

5.5.6.4.2 Access inner variable of the module for output

The inner variable are usual in the same `struct` as the public output variables. This is a usual old or simple programming style. The difference between declared output variables and inner variables is only: the first one are declared or described formally maybe only textual. But in C++ or other languages a difference can be given: public or private declaration. The real inner variables are private (or maybe protected) whereas the variables declared for outer access are public.

In OFB graphic it is simple to draw inner variables as “variable after expression” (see *5.8.7 Output possibilities, variable after expression* page *148*) or also as variable as call by reference or return destination of FBlocks (see *5.6.7.1 Reference and return output ofpDout() & ** page *108*). Also, inner variables can be better designed as (see *5.4.5 Structured type on data flow* page *62*).

Figure 49: odg/MdlOutVariableA.png



For that reason it is proper to define an inner variable of the module as accessible for output. The image right side shows some examples for that:

The `t1y` is the variable assigned as return variable to the `t1` FBlock in the mid of right side. It is a module variable because the output of the FBlock is designated with `name%`, then the drawn pin is created as module variable and the return value of the associated event operation is stored there. See *5.6.7.1 Reference and return output ofpDout() & ** page *108* It creates a code line:

```
thiz->t1y = step_T1f_Ctrl_emC(&thiz->t1, x);
```

It means, The `struct` variable `t1y` is used immediately as public output variable:

```
typedef struct ModuleInoutDef_T {
  ....
  /*public: */ float t1y;      //dtype:F
}
```

Because of the module's output variable has the same name `t1y`, this variable is used. That's why the module variable `t1y` is designated in the data `struct` with `public:`, here as comment for CC-language. The connection between `t1y` before the module output (which is the repeated drawn module variable) to the `t1y` output is formally only important for the data type forward propagation. It is `float` or `F` here because the `t1` FBlock is designated with `x:F` on its input, and the float variant is used. `t1y` as module variable is then automatically data type propagated also to `F`, and the formal existing output variable also because of their connection.

It may be an interesting information about the inner data of the OFB translator: Formally the module's output is another instance than the module variable with the same name. The module output is designated with `UseVarMdl` in its internal field `DinoutType_FBcl#accTargetCode` and hence ignored for code generation. As counterpart, the module variable `t1y` is designated with `FieldPublic` in this same field, and hence placed accessible in code generation. So a calling routine accesses immediately the module variable with the given name of the output.

The same is also done for `t1y2`, which is a module variable as variable after expression after `*2`. It is also done for `t1y3`, but here the connection is omitted, instead the type information is given immediately on the output, which results in the same generated code.

`dy` is a normal output public variable filled with:

```
// Module outputs due to the event stepO:
  ....
thiz->dy = dx_T1f_Ctrl_emC(&thiz->t1);
```

Also the module outputs `ya1` and `ya2` are not module variables, `ye2` is similar as `dy` an public output variable and `ye1()` is an access operation (getter). The interesting fact is, that an access to elements of the connected `struct` variable `d` is generated.

But the output variable `d` is again represented by the module variable `d`, as described for `t1y` etc. Here it is additionally interesting, that `d` is a `struct` variable. If instead `d` an output variable would be used and connected, maybe `d1` or `dout` on output, it results in twice memory consumption and a `memcpy` to copy the values. That is stupid, because the same data exists as module variable. If data consistence is interested, the a `ofpZou`t variable `d` may be used which is set by `upd`. But the consistence is also guaranteed, if the `stepO` event is regarded, which is so on code generation. If `stepO` comes, all parts of `d` are set.

The same copy effect is given also for the other variables with `UseVarMdl`, but just only for 2..8 byte.

5.5.6.4.3 Operation for outputs access ‘getter’

A getter for output is the encapsulated access to possible private data, as usual or recommended in Object Oriented languages. The advantage using getter is also, on debugging the target code, a breakpoint can be set in the get operation, to see when the access occurs. A breakpoint set to data access itself is sometimes possible but not supported in any case and more complicated to deal.

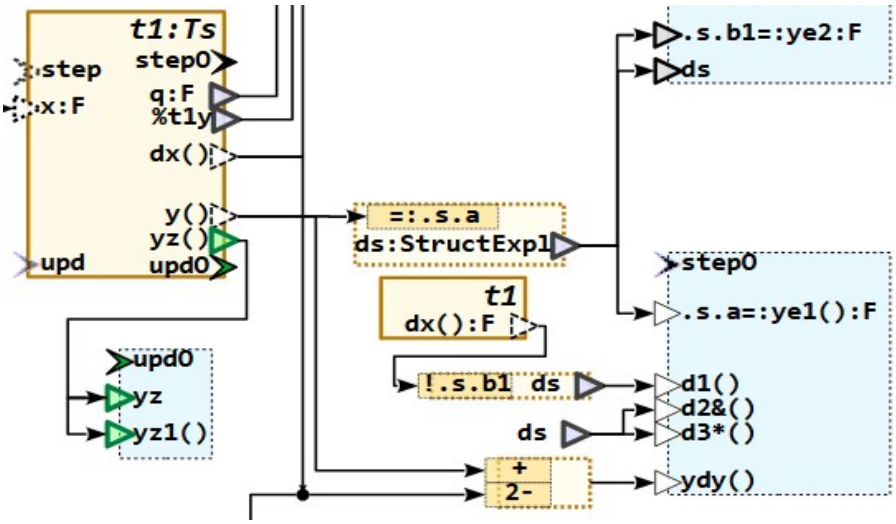
Another important advantage is: An operation may contain not only a simple access to elsewhere public data, it can be also the execution of a more complex expression, maybe also executing a longer expression in hard coded target language. For example a `atan2()` operation is a little bit longer, important in fast step times (50 µs or faster). Without getter, for example a library module calculates anyway this `arctan` in an output variable, also if this value don’t need to be used from the more universal library module. If this operation is part of a getter, and this output is not connected to the library FBlock, then this algorithm is not executed. The getter is only called if necessary. It saves calculation time.

A general question is: Using a getter to encapsulate data instead immediately access to the (public) variable - is this an extra effort for machine code execution. The answer is NO. Also in C language from C99 `inline` operations are possible. For calculation time efficiency an inline get operation is often reduce to the simple and fast data access in machine code. The compiler optimizes the machine code. The inline operation does not produce additional effort for the call.

Anyway, a complex (longer) get operation should not be called on demand more as one time. Its output (if necessary) should be written in a (local, stack) variable, which is accessed more as one time if necessary. The local (stack) variable is not an additional effort, because on optimizing a register is used, and non optimized also registers are used to store values without using a variable. This explanation is written for old-style C machine near programming, modern compiler optimize and modern programmer knows this.

It means using get operations in all cases should be seen as recommended. It may be also possible to adapt the code generation in that way, that always instead access to output variable an operation call for a *“getter”* is generated.

Figure 50: odg/MdlOutOperationA.png



The figure right side shows some variants of getter, beside the output to `struct` variables explained in the chapters before.

Generally, for the operation Dout the style `ofpDout...` should be used. Only for operations which accessed updated data the `ofpZout` need to be used, as here for `yz1()`.

The name of the operation outputs should be differ from data outputs, though they are formally differ because of the operation output property. The one reason for that is, the outputs in FBcl or IEC61499 writing style are not distinguished with same names. The formal built writing (connection) of FBlocks does not regard the operation property.

The `(` after the identifier dedicates this pin as operation access, writing `()` is possible or maybe recommended, but the closing parenthesis is not necessary. An `&(` means: return a `const` reference instead the value.

A reference means general, as also explained for immediately access to inner data, the data may be changed between several accesses,

the consistency is not guaranteed. But if the data are immediately stored on calling side, it is proper.

General, the operation outputs are associated to the shown event. It means the access should be done, is valid, if the output event, here `step0` was coming. Then if one access occurs, and the data returned by reference are stored in another instance, then the data are matching together, consistently. But if the reference itself is transported to other locations, maybe written as pure target code, the user is responsible for that.

The `*(` means, return a modifiable reference. This is primary a prone of error, because it is a impact to the black box principle. If access to inner data should be possible, then use an aggregation. Nevertheless this possibility is given here.

General, via getter only instance data can be accessed, means Variable as `ofpDout` cannot be accessed. It should be `ofpVout` or `ofpZout`. But all operation outputs, here shown for `yz()` to the `yz()` of the `t1` FBlock, is possible.

The `ye1()` is the access to a part of a structured data inside the module. The structured data is `d`, defined the mid, and the output operation `ye1()` does the access to the member `s.a` of this `struct`. The writing style of the element access follows the access in target C or C++ language (it is not translated), because this writing style is usual:

```
static inline float ye1_ModuleInoutDef
(ModuleInoutDef_s const* thiz) {
  return thiz->d.s.a;
}
```

The `d1()` output is the access to the complete `ds` `struct`, as ***return by value***. It means, the values are copied to a given output variable or also copied maybe in a temporary location. This has the advantage, that the data are consistent stored in the new location:

```
static inline StructExpl_BasicTest_s
d1_ModuleInoutDef (ModuleInoutDef_s const*
thiz) {
  return thiz->d;
}
```

But this return by value needs double memory space for the maybe comprehensive data. That's why an access per reference may be better. This is done with `d2&()`. The `&` before `(` symbolizes the return by constant reference:

```
static inline StructExpl_BasicTest_s const*
d2_ModuleInoutDef (ModuleInoutDef_s const*
thiz) {
  return &(thiz->d);
}
```

A returned reference is an association to the inner of the module, and this is a port in UML wording. The association can be used to access a part of the module, as usual for associations. But here only readable.

In 2025-07 this topic is yet not complete. An important change may be: Instead the `ofpDout` style and symbol `ofpPort` need to be used. A second topic is: For associations, aggregations and also compositions the state read only or read/write should be discussed. This is also not clarified in UML.

The `d3*()` produces the same code, but only without the `const` modifier, an read/write association..

The `ydy()` calculates an expression as return value. Only calling this operation (using this pin) forces this calculation effort. And, because it accesses pins of `t1` via operation, also there the effort only occurs on calling:

```
static inline float ydy_ModuleInoutDef (
ModuleInoutDef_s const* thiz) {
  return (y_T1f_Ctrl_emC(&thiz->t1)
  - (dx_T1f_Ctrl_emC(&thiz->t1) * 2) );
}
```

The `yz1()` is marked with `ofpZout` as pin style. It means that the pin is asscociated to the `updO`, the value is valid after the `upd` operation of this module. It accesses the `yz()` operation of `t1`.

```
static inline float yz_ModuleInoutDef
(ModuleInoutDef_s const* thiz) {
  return yz_T1f_Ctrl_emC(&thiz->t1);
}
```

In a similar way the output variable `yz_` is set, written with underscore to distinguish from `yz()` as pin name `yz`. the `t1.yz()` operation is here called twice (non optimal)

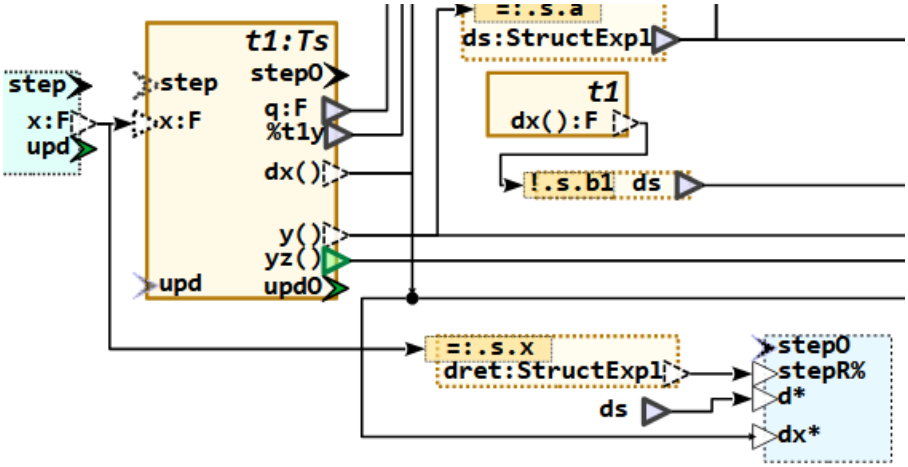
```
void upd_ModuleInoutDef ( ModuleIno ...
  upd_T1f_Ctrl_emC(&thiz->t1);
  //
  // Module outputs due to the event updO:
  thiz->mEvout_upd |= MASK_upd_updO;
  thiz->yz_ = yz_T1f_Ctrl_emC(&thiz->t1);
```

But follow also the next chapter

5.5.6.4.4 Event operations with return value and / or output variable by reference

This is a second approach to use an encapsulated style with operations, with the additional advantage that local (stack) data can be transported to outside, with saving memory and guarantee consistency.

Figure 51: MdlEventOperReturnRef.png



The return output of the event operation is designated with `%` after the name. The name should be built with event and `R`, as shown, but this rule is not necessary. It means, `stepR%` is the output for the return value of the `step` operation.

All outputs designated with `*` after the name (or also `name)` is possible) are output arguments called by reference of the event operation.

For this example the return value is the local (in stack) stored instance `dret`.

For this example it has the same DType as the module instance `struct ds`, but it can be of course differ.

It means this event operation has the header and implementation:

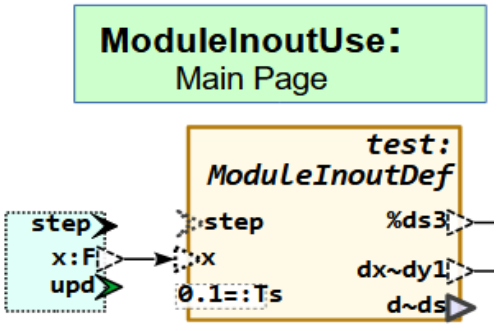
```
StructExpl_BasicTest_s step_ModuleInoutDef ( ModuleInoutDef_s* thiz
  , float x
  , StructExpl_BasicTest_s* d  // output per reference in otx: EventOperBody
  , float* dx  // output per reference in otx: EventOperBody
  ) { // ##oper_step

  StructExpl_BasicTest_s dret;  // #FBevin_dret_X_prep @23'130(130..132, 84..86)
  .....
  dret.s.x = (x);
  thiz->ds.s.b1 = (dx_T1f_Ctrl_emC(&thiz->t1));
  thiz->ds.s.a = (y_T1f_Ctrl_emC(&thiz->t1));
  .....
  *d = thiz->ds;  //otx: EventOperBody-doutRefer
  *dx = dx_T1f_Ctrl_emC(&thiz->t1);  //otx: EventOperBody-doutRefer
  return dret;
} // step_ModuleInoutDef
```

Figure 52: odg/ModuleInoutUse.png

The calling environment in C target language looks like

```
  float dy1;
  StructExpl_BasicTest_s ds3;
  ds3 = step_ModuleInoutDef(&thiz->test, x
  , &thiz->ds, &dy1);
```



For the calling environment see *5.6.7 Possibilities of outputs of FBlocks* page *108*. The same rules are valid for defining a module with the module's in- and output as described here and for defining the prototype or interface to a module as described in *5.6.4 Predefined FBlocks or definition on demand, relation with source code* page *102*.

The return per value means in C/++ language, the content is copied in data which are given by the calling environment. it is an *internal* `memcpy` This calling data can be also local (stack) variable, but in the stack area of the calling operation. Then it is memory optimized, no extra data are necessary. And the consistence is also guaranteed.

In the example, the output of the used modules are set with a stack local variable `ds3` on the

return pin (the designation with `%` is enough to designate the pin on using as return pin). The both reference variable to the step routine are designated as shown, with `~` as separator between left the inner name, and right the name of the used variable outside. `dy1` is a stack local variable as destination, and `ds` is defined in the instance of `ModuleInoutUse`.

If you follow the C language target code with knowledge of C(++) inner mechanism, you see that the return value is transported per inner `memcpy` from the stack local variable `dret` to the stack local variable `ds3`.

The data consistence and alive conditions of the data are considerate and proper, no external memory outside of the stack is necessary.

The same is for the reference variable `dy1`, which is only a `float`, but can be also a data `struct`. Whereas the `ds` on calling is filled also with an inner `memcpy` with the assignment `*d = thiz->ds`.to the given pointer to the outside existing `thiz->ds`, written as assignment which is a `struct` copy. Both variable in this instances have the same name.

This is only an example to test and demonstrate the code generation, without more sense.

5.5.6.4.5 Return a reference or variable by double reference

Return per reference is also possible. This is an returned ***association*** to inner data, which is presented by a port symbol (`ofpPort…`). As described in TODO, associations and also aggregations and compositions are either read only (with `const*` in C/++), or their are writable (a pointer in C/++).

The same is for reference variables on call. If they are drawn with `ofpPort…` or `ofpPortConst…`, then the code generation generates `DType**` or just `DType const**` for the formal argument and defines a pointer variable as destination.

This feature is not implemented or full tested in 2025-07, should be done step by step.

5.5.6.5 Order of module pins

The order of the pins is important both for the generate fbd file (IEC61499 presentation) as also as argument order in the operations especially for the generated code. If you think on reproducible build, then it is important that a repeated generation from graphic to IEC61499 should create the same text if the determining conditions are not changed. For example if a graphic position of a FBlock was moved to a slightly other position, or one connection is new routed in graphic, but is unchanged in functionality, then the generated code should be unchanged. The order of the pins should be determined. This can be done in two ways:

a) Sort the pins by its graphic position of first used GBlock.

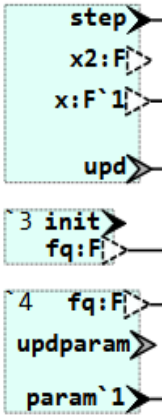
b) Determine the pin order in the first used GBlock by a sort number, named `nrGpos`. This number is written as last information in the pins text as `` `123 `` The character before the number is the *'grave'* (hexa 0x60), usual able to found on a US keyboard left top, on a German keyboard right top.

If the pins are used furthermore, in other pages or in the same page twice, the pin graphic order is not relevant. The first detection in graphic determines.

This is valid not only for the module's pins, also for the pins in GBlocks.

For the module's pin order also the graphic position or just a `nrGpos` can be written in the `ofbMdlPins` GBlock as text in form `` `1 `` for the first GBlock to use. Non dedicated with `nrPos` Pins and GBlocks are sorted after the dedicated with `nrPos` in graphic order from to to down and then left to right in columns.

Figure 53: odg/ofbMdlPins-2.png



This image shows the first page of module pins of the example `OrthBandpassFilter`. Normally the order of pins is first the order of GBlocks from top to down, and then in rows with a distance of at least 1 cm from left to right. It mean, the GBlock with `step` would be the first. `x2` is the first pin.

But on this page below and even on the second page there is a GBlock with a designation `` `1 `` and `` `2 `` for the `ctor`, not shown here. The two GBlocks below are designated with `` `3 `` and `` `4 ``, they are used first to sort the module pins after the `ctor`. In the GBlock with `` `4 `` (bottom) the pin order is normally top to down. But because param is designated with `` `1 `` it comes first before `fq` and `updparam`.

The non designated GBlock with `step` comes after all numbered GBlocks, then all GBlocks without number designation from top to down and left to right, then in order of pages. But also in the GBlock with page the `x` is designated with `1 and hence comes fist before `x1`.

As result the following order in the IEC61499 fbd file occurs:

```
EVENT_INPUT
  ctor WITH Tstep, ...
  init WITH fqi;
  param WITH fq;
  updparam;
  step WITH x, x2, ...
  upd;
END_EVENT
EVENT_OUTPUT
  paramO ...
END_EVENT
VAR_INPUT
  Tstep : REAL;
  fqi : REAL;
  fq : REAL;
  x : REAL;
  x2 : REAL;
```

For the graphic position GBlock order, internally a number is build consist of the page number on a high position (bit 22), the x position from bit 11 and the y position. The positions have a resolution of 1 mm, hence 2047 mm or 2 m * 2 m area can be used for the graphic, and ~ 1000 pages. But the x position is filtered to columns: When two GBlocks are almost under each other, but not exact, they should be related together in one column. For that a distance of +-9 mm is accepted as the same x column. Whereby not the first found shape determines the common x position, but the mid value of all. the GBlocks are on the same x position rights side but not left side. But all are accepted to be in one column. It means the order is as you see.

A GBlock more right comes in order after the last GBlock on bottom more left. But the distance of +- 9 mm of the column width should be proper to a normal size of a GBlock (10..20 mm width) and a proper column association.

The pin order in a ***GBlock*** is first left from top to bottom with x1 left of or exact on the border of the GBlock area, then on top (y1 less or equal the GBlock area), then right side with x2 right or equal to the GBlock border, and then bottom side from left to right. At last also Pins which are only inside the GBlock are regarded. in order of first left to right, then (the fine order) top to down, in 1 mm rounded positions.

The given number after the grave character `` `1 `` is internally converted to a negative number for sorting in range -9999… That's why this shapes are sorted first.

The same is done also for ***FBlocks***, which can have more as one GBlock for one FBlock. Also here the order of the same FBlock instance (same name) is used as first order, from page, x-column +- 9 mm and then y-position. Then the pin order inside each of this FBlock is build with the same rule.

Also the same is valid for ***FBexpr***, the expression GBlocks. Whereas FBexpr are always present by only on GBlock. The order of arguments of the expression is left side from top to bottom etc.

5.5.7 Inheritance of modules

This is sometimes considered, but not yet full elaborated. The idea is, a module has a derived interface, and should fulfil this interface definition, enhanced with some more aspects. On module definition the interface should be presented and tested whether it is all correct.

empty

5.5.8 Aggregated modules, associations and composite

5.5.8.1 Class diagram of a module with aggregations

The following image shows similar as in UML a class diagram:

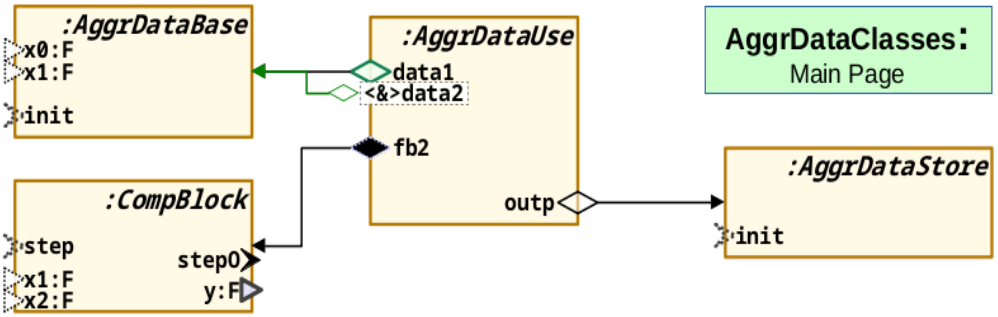


Figure 54: ofb/ExmplAggrDataClasses.png

There are three modules which corresponds:

* `AggrDataUse` is the module which has three aggregations:

* `AggrDataBase`: A read only aggregated module from `AggrDataUse`, which, for example, may produce data to process. It is aggregated with `data1` and `data2` with a read only aggregation.

* `AggrDataStore`: A read/write aggregated module from `AggrDataUse`, which may store the processed data, aggregated via `outp`.

The differences to a standard UML are:

* The aggregations are related to its source class. In UML class diagrams aggregations are ***between classes***. The reason is, supposed, UML has the focus to the software structure (architecture), and there is an aggregation between classes. But in all programming languages, aggregations are built as reference (pointer in C, or just also other mechanism as a handle and an appropriate memory address table), which is ***a property, an element of the class which aggregates***. Hence for code generation and also for understanding it is better to consider the aggregation as property of one class. The other class determines only the Type of the aggregation, or also the aggregated instance if FBlocks are used (see next).

* Corresponding with the point before: Aggregations are always directional, never bidirectional as often used in UML. UML knows both possibilities, but often the bidirectional variants are first offered from tools. But this is not the world of running code, and it may not be the real world of architecture. An aggregation means, another instance of type of a class is need to work. This instances is outside (in opposite to a composition, where it is inside), and it is used stable from beginning (in opposite to an association, the associated instance can be changed).

* If a bidirectional association is really necessary, both instances should work together and should known each other, then the question is: Should the aggregation in maybe one direction go via an interface? To have flexibility in architecture? The bidirectional aggregation covers and prevents this consideration.

* If really a simple bidirectional aggregation is necessary, use two separated ones.

* UML does not know primary the distinction between read/write and read only aggregations, maybe only in some properties or stereotypes of the aggregation. But this distinction is essential! It is the quest, does the aggregating class change data in the aggregated class. This is essential for test, for understanding what's happen. The read only property is mapped in C/++ language to a `const*` pointer.

As described in *5.2.5 Pin styles, ofp* page *48*, the aggregations can be presented by the here shown more as one pin drawing possibilities.

The composite reference (filled diamond) can be drawn and hence shown here, but the composite is then inside `AggrDataUse` as the FBlock `fb2` with this type.

5.5.8.2 Defining the aggregation destination in an Object Diagram

A OFB graphic is similar as an Object diagram in UML, if FBlocks are used. FBtype are classes, and FBlocks are instances or **Object**s. The OFB diagram or also an Object Diagram in UML clarifies, which instances are aggregated.

In the example it is:

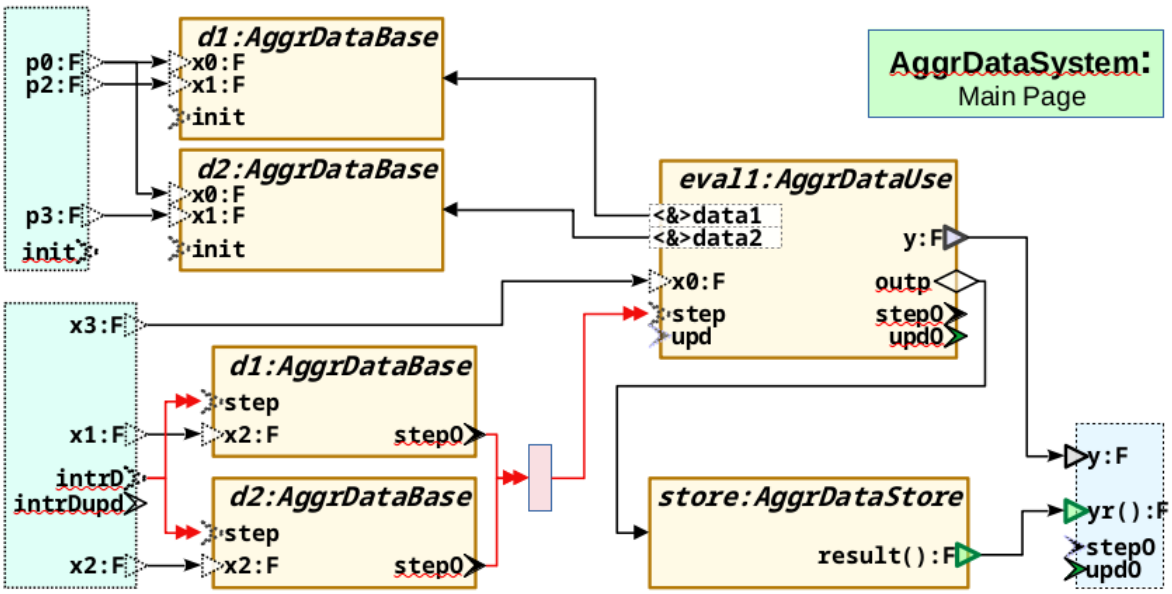


Figure 55: ofb/AggrDataSystem.png

This OFB graphical presentation defines the aggregations as well as the class diagrams in the *Figure 54: ofb/ExmplAggrDataClasses.png* page before. The class diagram may be unnecessary for OFB. It is helpful (beside its documentation character) if the aggregation goes via an interface, which is presented in the class diagram view. The FBlock OFB diagram would assign the type of the aggregation with the FBtype of the reference FBlock, but also here the interface can be drawn between (TODO test and explain).

The main task of the FBlock view is: Clarify which instances are aggregated. For this example the both aggregations `data1`, `data2` goes to the instances or FBlocks `d1` and `d2`. And the other aggregation `outp` goes to `store`.

But beside the aggregation clarifying, the diagram shows also the data processing. Here the `intrD` event (may be an interrupt from hardware) triggers data processing in `d1` and `d2`, and then, if both are done, after the light red `ofbEvjoin` FBlock, of FBtype `Join_OFB`, the event process triggers or is continued in the `eval1:AggrDataUse`. The `store:AggrDataStore` seems be uninvolved, but its result is used and outputted. See next chapter.

All FBlocks are compositions of the FBtype `AggrDataSystem`, the here defined module. This relation may be also presented in a OFB FBtype diagram, but it's not necessary.

5.5.8.3 Using aggregated FBlocks - proxy FBlock

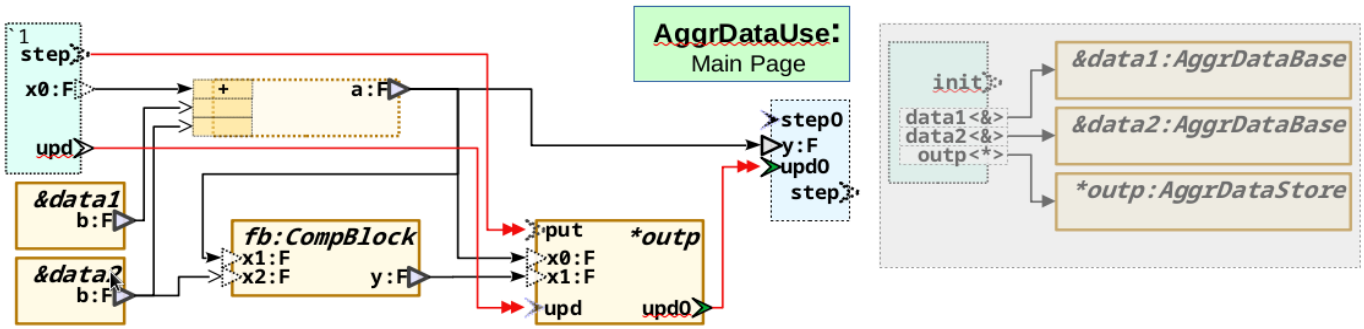


Figure 56: OFB/Exmpl_AggrDataUse.png

This is the essential graphic to explain how to deal with aggregations in Function Block diagrams: Using the aggregations.

The module's FBtype `AggrDataUse` is sufficiently proper described by the modules before, the using module definition `AggrDataSystem` and also the class diagram in `AggrDataClasses` on pages before. This modules are all part of the translation process, contained in one file (here `BasicTest.odg`) or in the adequate file assembly for translation. Hence it is not necessary here to show the aggregations as inputs of the module. They are already well defined in the both other modules. But it is possible. This is suggested by the gray box right side (style `ofbDisableArea`). If the module is translated alone without environment definition, because for example it is used outside of the OFB graphic or in another OFB translation process, the module is completely presented on demand with the right grayed part.

But first the quest, how to use aggregated FBlocks. They are not really a part of the module (as expected for ordinary FBlock diagrams), they are outside. In C/++ language (or also in other ones) they are referenced by a pointer. But in the OFB module they are presented as FBlocks as all other? No, they are **proxies**. This is expressed by the `*` or `&` (or also `-` or `%` for associated FBlocks) written before the name.

With them, also the FBtype of these proxies are clarified, it is the data type of the aggregation (or association) pin. This data type is not shown here in the module definition graphic for `data1`, `data2` and `outp`. They are predefined, for this example, both in the using module `AggrDataSystem` page before, as also in the class view `AggrDataClass`. It means, the type with colon after the FBlock name is not necessary. The FBlock names `data1`, `data2` and `outp` must follow the names of the aggregation pins as they are defined in the diagrams before.

With the grayed part right side, exact this is given, if the using modules or other predefinition of the module's FBtype are not present. The pins in the `init` `ofbMdlInp` GBlock are aggregations, `<&>` as read only and with `<*>` as read write access. Hence the connection to the proxy is an aggregation, which assigns the FBtype of the aggregated FBlock to the aggregation pin, here `AggrDataBase` and `AggrDataStore`. Hence, the aggregation pins are well defined, instead in the using module or as any predefined FBtype for this module.

page break

5.6 Possibilities of Graphic Blocks (GBlock)

This chapter should show all possibilities for Function block shapes (FBlocks).

Table of Contents

5.6.1 Difference between class, type and instance (“Object”)

In ordinary Function Block Diagrams usual any FBlock is an instance. The term *“class”* is not usual. If a FBlock is derived from a FBlock in a library, the FBlock in the library can be seen as *“type”*.or just *“class”*. The library FBlock contains the inner functionality, and defines the interface to the FBlock. The own diagram *“uses”* it and builds an instance with own inner data..

In UML (Unified Modeling Language) the term *“class”* as synonym for a *type* is usual, and instances (incarnation of the class type), sometimes denoted also as *“object”* are more rarely used in diagrams.

The OFB (*Object oriented Function Block graphic presentation*) uses any FBlock also as presentation of the type (*class*). If the FBlock have an instance name, it is also an Object or ***FBlock***. The type is presented by all FBlocks with the same type name, also if they are several instances. But also the ***same FBlock*** (same instance, same instance name) ***can be presented more as one time*** in several graphic shapes (*GBlocks*). It means a class or a FBlock can be shown in different contexts, see also *Error: Reference source not found* page *Error: Reference source not found*

Name and type designation:

The name of a FBlock and the type can be written in the text of the rectangle shape for `ofbFBlock` which is used for the FBlock, and also for a class in UML thinking. The original style of `ofbFBlock` expects the text in the right top corner, see following image. But sometimes this works not properly, then either “*Format – Clear direct Formatting*” on the shape helps, or Menu “*Format – Text Attributes*” and adjust it.

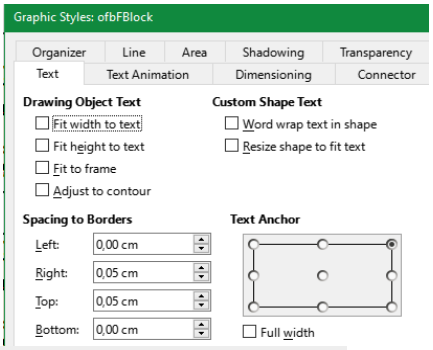


Figure 57: odg/ofbFBlock-TextStyle.png

You can use also the direct formatting to put the name and the type in the mid, to another corner, or at a desired position. But right top is often a good decision because the FBlocks have often more inputs (left side) then outputs.

- By the way, inputs do not need positioned left side, they can be also right or rotated on top or bottom, same as outputs. The drawing style have more possibilities than some commercial tools, you can use it for your own.

The other possibility for name: type is a text field marked with the style `ofnClassTypeName`. This text field can be positioned anywhere inside or touching your FBlock shape. If you want to describe only the class (type), then you need to write `:typeIdent` with the colon. This is not UML-conform, but unique.

If you omit the type name, but the classification of the named instance is done in another FBlock with the same name, it is admissible. It may simplify the diagrams. If the type is never associated, an error message is given on translation.

The shows an example which contains 3 FBlocks which define the type or class `Bandpass`.

Two of them are only for type definition, here the association of data inputs and outputs to events are defined, and also the aggregation `param` associated to the `init` event. The `h3:Bandpass` is an instance definition which contains constant values for two inputs and connections for two other ones. Similar, this is a type definition because here the inputs for `kA`, `kB` etc. are defined as associated to the `ctorObj` event. It is for construction. The type `WaveMng` is defined with also 3 FBlocks, but all with the instance `wf1mng`. One of these FBlocks has no type definition, but the type assignment to the instance is given on two FBlocks with `wf1mng:WaveMng`, one association would also be unique, both associations should be congruently. The more as one FBlocks are necessary because the event and data association should be clarified each on one graphic FBlock instance.

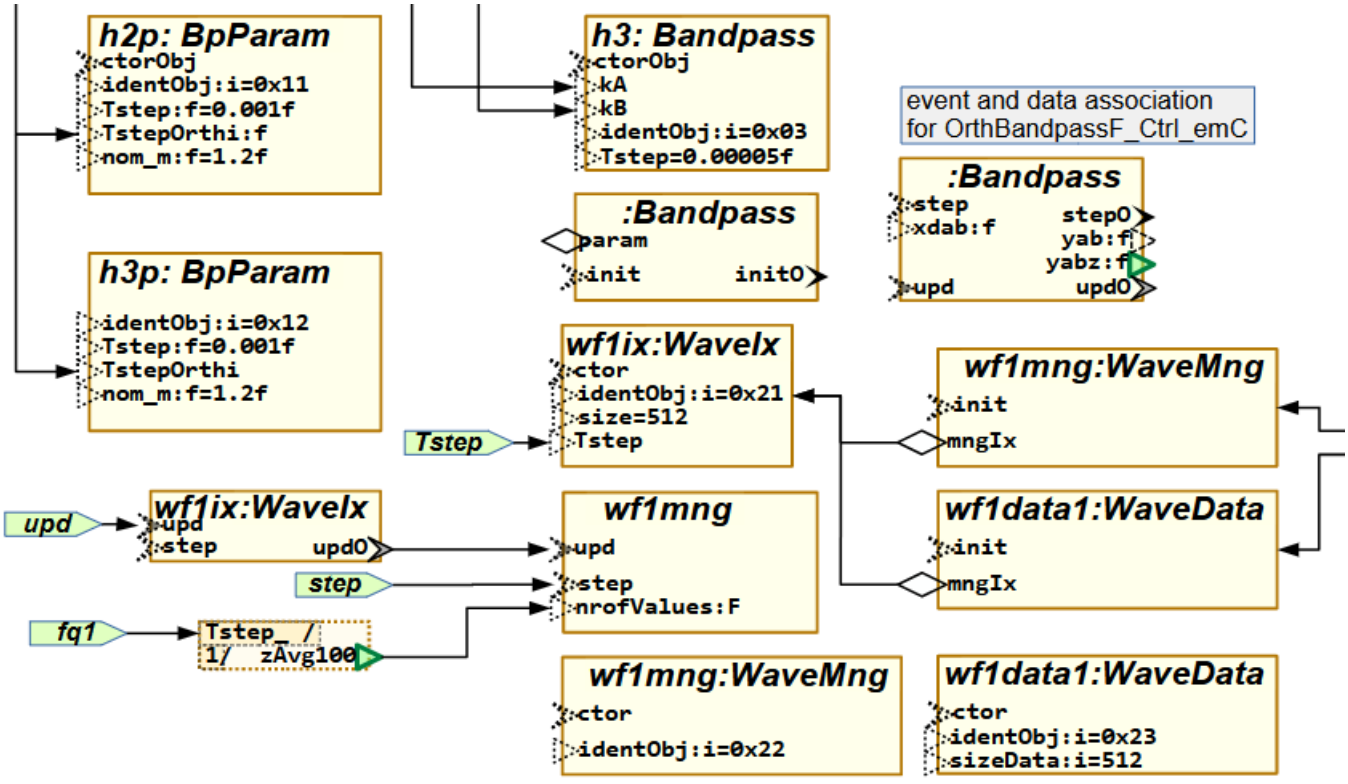


Figure 58: odg(ExmplFBlocksTypes.png

--

5.6.2 GBlocks for each one function, data – event association

In this chapter and also following the following terms are used:

- *Association* between data and events. Also in IEC61499 the term *association* is used in the same manner. The meaning of *association* in UML kind is not related to this.

- *Aggregation* is here the term of UML, used for aggregations shown in the graphic. In implementation these are usual references (containing addresses of the aggregated data with determined type or just pointer).

- *corresponding* events for input and output and for prepare and update (see also *5.13.2 Life cycle of programs in embedded control: ctor, init, step and update*

- The terms *<:n:"operation".> <:n:"method".>* and *<:n:"function".>* means all the same. *<:n:Method.>* is the first used term for Object Orientation. *<:n:".><:n:0.><:n:peration".>* of a class means the same, the implementation in C language is named *<:n:"function".>* (may / should have a reference to the data for Object Orientation) and *<:n:"function".>* is also a common understanding what is done (execution of any functionality).

In ordinary Function Block Diagrams one graphic FBlock presents one instance of a FBlock, and each FBlock has often only one function internally, maybe completed with corresponding construction and init functions. No more. But usual programming in C language (object oriented), more as one function or ***operation*** can be used with one data `struct`, and in object oriented languages (C++, and more) any class has of course more as one *"method"*, *operation* or just *function*.

The non-consideration of the object-oriented concept with several operations per class may be one of the reason of the divergence between graphical programming (often used, non object oriented, specific user-bubble, specific tools with code generation) and the frequently object orientated text coding (other bubble of engineers).

One of the goal of OFB is: bringing it together.

But first, discuss about the event thinking:

The idea of event driven thinking of the here used IEC61499 textual presentation of the graphic is not in contradiction to the object oriented thinking with operations, as explained following.

If you look in on the last page, or just in,

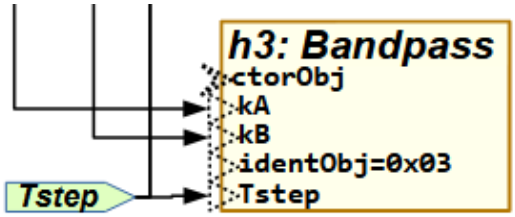


Figure 59: odg/FBlock_ctorObj.png

you see the `h3` FBlocks with the `ctorObj` or the `ctor` event. That calls the `ctor…` operation for this instances with the given constant or wired input data.

Figure 60: odg/FBlock_stepUpd.png

shows the same FBlock instance h3, but here with the `step` event with `xdab` as data input and some outputs. It defines that in `:Bandpass` the `xdab` data input is associated to the `step` event, or just as input argument for the step_… operation. The other `step0`, `upd` and `upd0` events are also corresponding to `step`, as its output (which operation follows) and as corresponding update event.

It means, any FBlock appearance (it is a graphical Block, GBlock) describes one operation of the FBlock in its context (calling the operation) or just seen as class or type, one operations with its arguments. But also several GBlocks are possible for several arguments of the same operation (presented by the events).

That is newly also for FBlock diagram thinking as also for UML.

The following rule is used:

- If a graphic FBlock has exact one prepare event input (style `ofpEvin…`), then it defines all data input associated to this prepare event.

- The only one update event input (style `ofpEvUpdin…`) is then the correspond update event input.
- The only one `ofpEvout...` is the corresponding output event to the `ofpEvin`.
- All data outputs are associated to the `ofpEvout`.
- The only one `ofpEvUpdout…` corresponding to the only one `ofpEvUpdin`.
- If more as one `ofpEvin…` is given in the graphic FBlock, or more as one `ofpEvout…` or neither an `ofpEvin…` nor an `ofpEvout...`, then this graphic FBlock does not define associations between data and events. The FBlock can be used instead as overview over more as one events, over all or parts of non formal event-associated data but showing commonly relationships of data.
- If more as one update events are given, it is shown as error, only the first update event is used (`ofpEvUpdin…` or `ofpEvUpdout...`).
- The data associated to the events and the corresponding events may not be complete. data-event-associations and corresponding events can be dispersed over more as one graphic FBlock. It means the conclusion *<:n:"that's all".>* cannot be done. But it should be recommended to show things as complete.

It means, ***a graphic FBlock instance represents*** (a part of) o***ne function, operation or method***of the assigned instance with its type. In this manner the term *"Function block"* for one function (*operation*, *method*) of a type is proper. The association to one type is given with the type designation, and the assignment to the same instance data are designated by the instance name.

Thinking in these FBlock approaches is related to Object Oriented thinking.

5.6.3 Aggregations are associated to ctor or init events

If aggregations are merged in a graphic FBlock instance between data and events, the aggregations are ignored for correspond event-data assignments. See

Figure 61: odg/FBlock_initAggr.png

But if the `ofpEvin…` event **starts with** `ctor` or with `init` as in , then the aggregations are associated to this given event. It means aggregations can be set only in such operations which names starts with `ctor` or `init`. That are usual used for the constructors and the `init` operation. See also chapter *5.13.2 Life cycle of programs in embedded control: ctor, init, step and update*.

It means, the opportunity is given to show aggregation ordinary in diagrams for understanding of relations between FBlocks (instances or classes) between important data connections with there event – data associations (in IEC61499 terms). The data connections regarding its events are used for code generation as arguments of the operation, the aggregations are also regarded as connection between instances, but not related to the shown events.

If the aggregations **are never shown together with an** `ctor-` or `init-`event, then they are automatically associated to an event with name `init`, or just to the `init_Type(…)` operation. This simplifies drawing diagrams.

This rule is effective for code generation. The generation scripts can be indeed adapted to call any specialized operation, for example to use the identifier part after `init…` as name for the function, but it may be more simple to adapt the called code for example by a macro or inline operation named `init_…(…)` which calls then the original one.

5.6.4 Predefined FBlocks or definition on demand, relation with source code

For simple usage a FBlock can be defined on demand: As shown in the chapters before it can be drawn with the necessary pins, and the existence and order of pins defines the generated code.

Let's demonstrate this on a simple smoothing FBlock or *"Low pass filter"*. Such a functionality is described for example in https://en.wikipedia.org/wiki/Low-pass_filter. In C language it is very simple. The core algorithm is :

```
static inline float step_T1f_Ctrl_emC
(T1f_Ctrl_emC_s* thiz, float x) {
  thiz->dx = thiz->fTs * (x - thiz->q);
  thiz->q += thiz->dx;
  return thiz->q;
}
```

The filter has an additional state value `dx` which can be used for a DT1-Functionality, a Differential FBlock which smooths the differential. A step response for that is not an infinite value (Dirac impulse), or for discrete systems an impulse of width=`Tstep` and height=1 or x, instead it is the ***area*** of the `dx` output related to the step difference. `dx`.It builds a high-pass-filter.

The filter has a update operation:

```
static inline void upd_T1f_Ctrl_emC
(T1f_Ctrl_emC_s* thiz) {
 thiz->qz = thiz->q;
}
```

The update operation is only necessary if the state value of `qz` before step should be used for other functionalities. Often in pure C programming it is not used.

The filter factor `fTs` is calculated as:described in the Wikipedia article.

```
void param_T1f_Ctrl_emC (T1f_Ctrl_emC_s*
thiz , float Ts) {
 thiz->fTs= thiz->Tstep / (thiz->Tstep + Ts);
}
```

The constructor and init

```
extern_C T1f_Ctrl_emC_s*
ctor_T1f_Ctrl_emC(void* thiz);

extern_C bool init_T1f_Ctrl_emC
(T1f_Ctrl_emC_s* thiz
  , float Ts_param, float Tstep);
```

completes the system.

```
static inline float get_dx_T1f_Ctrl_emC
(T1f_Ctrl_emC_s const* thiz) {
  return thiz->dx;
}
```

returns the value for the high pass.

To use this C routines in graphic on demand the following graphics are necessary:

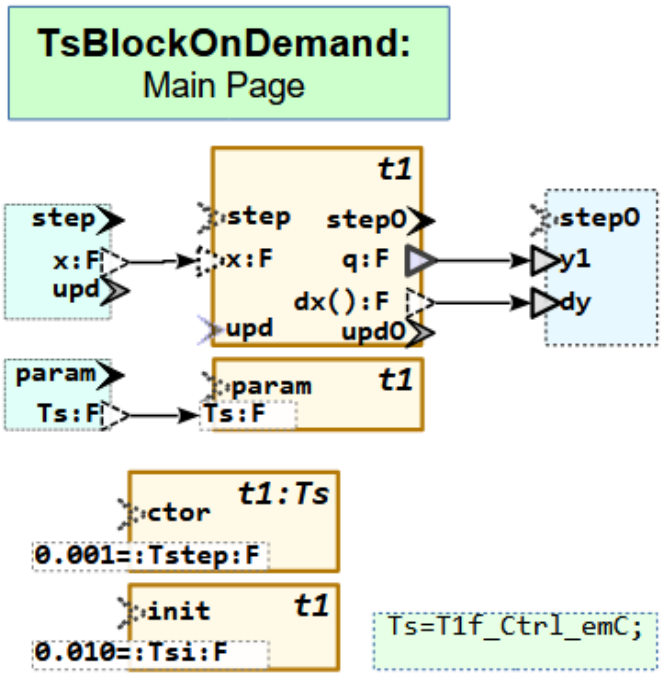


Figure 62: odg/TsBlockOnDemand.png

This is one page in the BasicTest.odg. Any GBlock determines with its event one operation, `step` with `upd`, `param`, `ctor`, `init`. The order of Din pins top to down and left to right should be the order of arguments in the existing operations, and organizes the call of the operation in this order of arguments. For that the Din pins uses the style `ofpDin...`.

The output `q` is the `struct` variable and hence a pin with style `ofpVout...`, But the `dx()` is given as access operation (*"getter"*), drawn with style `ofpDout…` but marked with `()` to determine, it is an access operation. So the code generation knows how to generate the code to call this FBlock operations.

Note that for the correct including of the header file the `-cfg:makeScripts/local.aliasHeader.cfg` should contain a line:

```
T1f_Ctrl_emC = T1f_Ctrl_emC,
h=emC/Ctrl/T1_Ctrl_emC.h;
```

See *5.5.6 Module pins* page *78*.

Better to have predefined FBlocks

But if you have differing orders of pins due to drawing mistakes etc, the code is confused, and there is no way to prevent or see this mistakes exclusively ask the compiler for the generated code. That's why the definition on demand is only proper to use for simple FBlocks, only used ones, only have a few events.

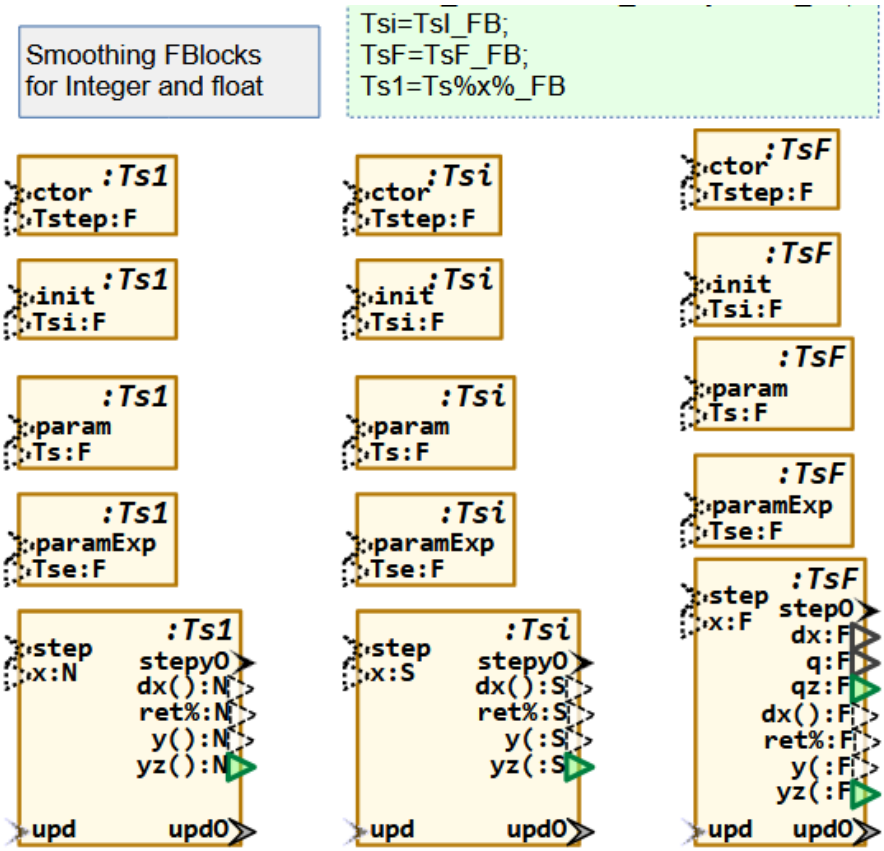
It is better to have prototype definitions or just predefined FBlocks. A module diagram with this predefined FBlocks describes only and exact the interface to the given legacy code, or just the interface to another translated module.

There are two ways to get predefined FBlocks for the current module in one translation action:

a) Given textual. This is sensible if a graphical module was translated outside of this translation session. It is the fbd file given by translation. It is also sensible if a module should be used for different projects, and the effort to write this code is only one time, together with writing the C/++ code of implementation.

b) Given graphical and translated in the same session or project. This is sensible if the overview over called functionality should be given in own diagrams.

Figure 63: odg/PIDctrl_TsModulDef_Ts.png



This image shows the complete definition of three variants of the Smoothing blocks. `Ts..FB` FBtypes (classes) in the module `PIDctrl_TsModulDef` which is contained in `src/Templates_OFB/odg/LibCtrl_emC.odg`.

The GBlocks are FBtype, or classes in UML view. It is necessary to write the colon `:` before the class name.

The relation to code generation is given by the green box of style `ofbAlias`. As interesting additional feature here non determined data types are used (`N`, `ANY_NUMERIC`). In the `ofbAlias` it is also clarified that the `%x%` is replaced by the one-char-identifier of the data type on the pin `x` of this FBtype on usage. See *5.4.2 Unspecified types* page

If such a pre defining module graphic (as a library) is given, then the using module do not need to contain all information. Especially the order of pins does not play a role.

Look on the image right side. Here only the event step is mentioned, because it is the triggering one to smooth the value w1 from another step time. ctor, init are not necessary to draw because the association to it from the shown data pins are given and unique.

Lets look on an example:

Figure 64: odg/T1_appl.png

It's also possible to draw more as one Graphic block for this instance, for example to show the data flow to the parameter pins. Of course the alias is written a little bit other: `Ts=Ts%x%_FB`.

5.6.5 State Machine GBlocks

To present State Machines in specific graphic blocks two styles can be used:

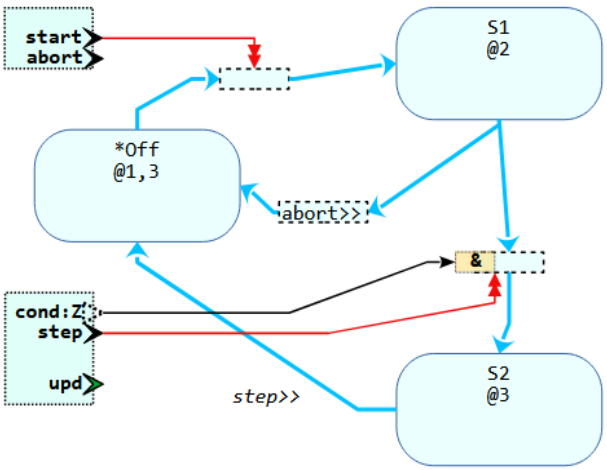


Figure 65: Stm/SimpleStM-OFB.png

The states are presented by the style **ofbState**. The rounded corners are not part of the style, they are a property of the shape. It can be shown as desired, but rounded corner are usual for states.

Transitions between states are either presented simple with a transition connector of style **ofcStateTrans**. Then the event and condition can be written as text to the connector, as usual also in UML state diagrams, seen in the transition between S2 and Off. But in OFB diagrams it is also possible and recommended to draw an extra GBlock in the transition. The style should be used is **ofbStateTransCond**. This GBlock is similar an expression GBlock, intern it is mapped to a **>>FBlock_FBcl** of FBtype **>>StdLibFBlocks.fbtExprEv_OFB** which is also used for event controlled execution, see *5.6.10 Conditional execution with boolean FBexpr* page *112*. Hence this GBlock can have an event input, which is the **prep** input of this expression. Some more conditions as & condition can be used as guard for the transition. Or wired conditions should be built with an extra FBexpr. This is shown for the other transitions.

The transition FBlock can have the event and guard also written in text form. Also a combination of event on the transition itself and the condition textual or wired on the **ofbStateTransCond** is possible.

To show nested states in the OFB diagram the following writing style is possible:

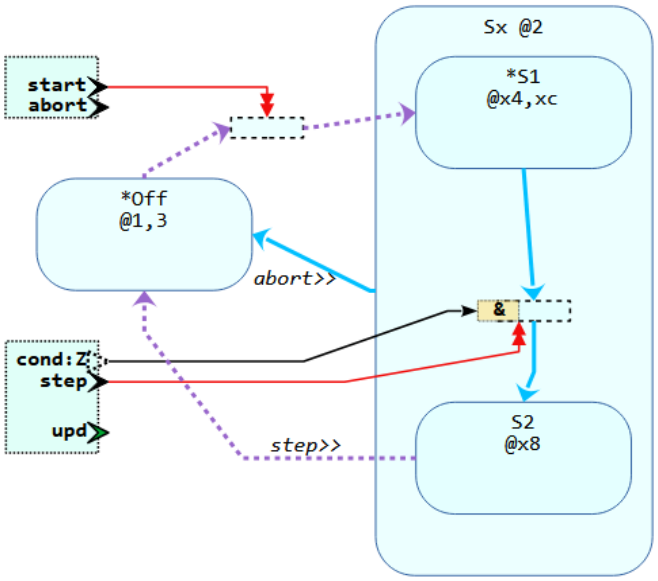


Figure 66: Stm/NestedStM-OFB.png

The difference here is, the transition to **Off** with **~cond** as guard works also for the **S2** state.

The https://www.omg.org/spec/MOF/2.5.1/PDF says in chapter *14.2.3.9.4 Firing priorities*: *“a Transition originating from a substate has higher priority than a conflicting Transition originating from any of its containing States.”.*

empty

5.6.6 Possibility of inputs of FBlocks

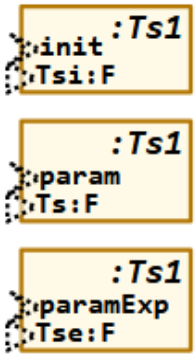
5.6.6.1 Inputs as local arguments of the event operation ofpDin

As shown in chapter before, the inputs are drawn with a figure of style `ofpDinLeft` or `ofpDinRight`. The difference of both is only appearance, the text is organized left or right. For input pins shown as rectangle shape also `ofpDin` can be used. The effect for code generation is the same for both. That's why the documentation contains usual the style designation `ofpDin` or `ofpDin...`.

These pins on FBlocks are arguments of the event operation for code generation. They are existing as stack (local) variables in the target code execution. It is similar as for a module variable with the style `ofpDout`, which is also a Stack or local variable. The access to these variables can only be done in the same operation execution, or from view from graphic, in the same event chain.

The names of the `ofpDin` variables on FBlocks are not related to target code for C/++ or also Java code generation, because the names of arguments does not play a role for the call of an operation in these languages. Other than in languages such as Structure Text (used in automation computation). The style comes from PASCAL known on end of 1980th. Here the actual arguments are associated by the argument name of the arguments of the called operation. Because it may be that a graphic would be translated to such languages, and also for well documentation, the name of the `ofpDin` in a FBlock should follow given names in target code.

Figure 67: odg/TsBlock_ArgNames.png



If different operations, which are different events in an FBlock, have similar arguments, and the argument names are usual the same in target code, then the `ofpDin` names should be different! Look on the image right side.
All three Din presents the smoothing time constant, and they may be named equal in target code:

```
void init_T1... (... , float Ts);
void param_T1... (... , float Ts);
void paramExp_T1... (... , float Ts);
```

The different names are first necessary to distinguish the pins from data flow to the event association. If may be possible that all three argument values are built in the same manner (it is the smoothing time maybe comming from the same input). But, really not from the same input, often locally from an argument of the event chain.

Hence it is recommended to name this arguments also different in the legacy target code following the graphic appearance:

```
void init_T1... (... , float Tsi);
void param_T1... (... , float Ts);
void paramExp_T1... (... , float Tse);
```

Inputs of FBexpr are more complex. The `ofpExpPart` pins contains sometimes expression terms, they are not designated to memory locations.

*5.6.6.2 Call by value or call by reference ofpDin& **

For simple variables as arguments of course the value should be given. But if the variable has a struct type then also it is sensible in C/++ language to deliver a pointer to data referencing the argument value. Then of course the data consistence should be regarded, and the accessibility / existence for the data. Local data (in stack) can be accessed via reference, but only in the current event chain (in the same operation for target execution).

Arguments provided by reference are drawn with the style `ofpDin` but have a `name&` designation for a `const` reference or a `name*` designation, if the data should be backward able to change. The last one may be problematically for a proper design but it is usual in manual C/++ programming.

This feature is not yet implemented 2025-06

5.6.6.3 Instance variable for inputs ofpVin

Their is a second possibility for inputs to FBlocks: Using the style designation with `ofpVin...`. This describes a variable as member of the `struct` or `class` of the FBlock, which can

be set immediately (public access). This allows to set the variable in any data flow (or event chain) independent of the event call respectively with another (related) event. It is a simple possibility, in response to the user. The event association describes, which event uses this pin. It can be used by more as one event. In 2025-06 this is not complete implemented, hence only mentioned here.

*5.6.6.4 Instance variables as reference ofpVin& **

This are intrinsically associations to any inner data ports of the source FBlock. But for more simple understanding it can be drawn also as data flow.

Write `name&` or `name*` for a `ofpVin` pin. Then the data flow delivers a `DType const* name` or `DType* name` as reference stored in the destination FBlock for the data flow.

This feature is not yet implemented 2025-06

5.6.7 Possibilities of outputs of FBlocks

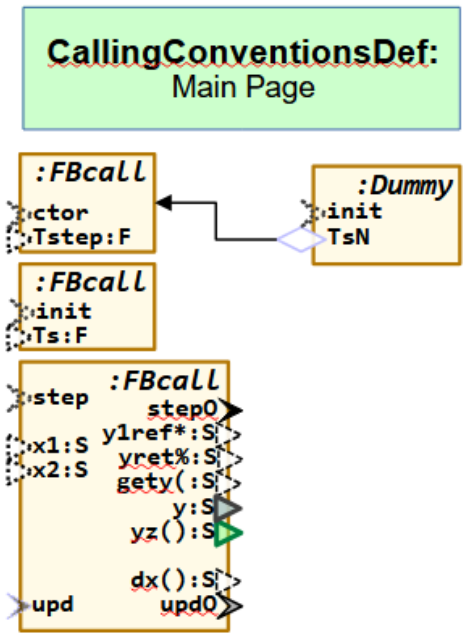


Figure 68: BasicTest/CallingconventionsDef.png

The image above shows an example, a little bit similar to the Ts FBlock in the chapter before, but with more nuances. It's a constructed example.

The step and upd operations have some outputs. Note that two output events are only admissible for `ofpEvout` and the associated `ofpEvUpdout`. adequate to `ofpEvin` and their related `ofpEvUpdin`. All the shown outputs are related to the output event(s) in the same GBlock. It means, if it comes (usual after the evin, but possible also from a state machine), then the outputs are set already and can be used.

*5.6.7.1 Reference and return output ofpDout() & **

The first and 2th shown output `y1ref` and `yret` are **immediately related to the evin operation**: It is an **output by reference** and the **return value**. Hence the step operation should be defined as:

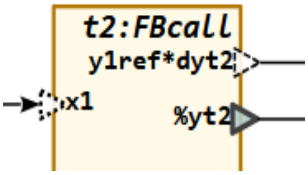
```
int16 step_FBcall_... (FBcall_..._s* thiz
, int16 x1, int16 x2, int16* dx);
```

Reference outputs are designated with an asterisk after the name: here `y1ref*`. All reference outputs are assigned after the inputs in the order as defined in the graphic.

The **return value** is designated with an percent after the name, here `yret%`, or an ampersand, here not shown: `name&`. The second form is to return a complex type (struct or instance) per reference.

On call of this operation the reference or return outputs need to have a variable of the module. This is done by the following graphic:

Figure 69: BasicTest/Calling ConventionsUse _DoutRefRet.png



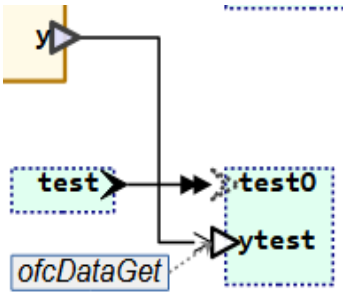
The kind of the Dout in the using GBlock determines the kind of the variable. Here `ofpDoutRight` is used for `y1ref`. it builds a local (stack-) variable, and for the return variable a `ofpVoutRight` is used which builds an instance variable.

The name for these variables are related to the module. It can be used free, here `dyt2` and `yt2`. But the name of the pin in the FBtype should also be given, because the graphic order is not relevant for predefined FBtypes. The name of the FBtype pin is written before one of the characters `~ * % &`, the name of the module variable to the pin is written after them. For return variables the FBtype pin name can be omitted, if it is unique, only one return pin exists. That is used for `%yt2`. `>>OdgNameTypeArray#OdgNameTypeArray(...)`

5.6.7.2 Instance variable with public access ofpVout

The simplest kind of output is, set a **variable** in the `struct` or `class` of the FBlock, which can be **used with direct (public) access afterwards**. This is designated by a `ofpVout` pin, as shown for `y` in the image above. The data to event flow translation assures, that the variable is used only after the output event.

Figure 70: BasicTest/ CallingConventionsUse _Vout_ofcDataGet.png



But the variable can be used also in any other event chain, in response to the user. This can be done by using a connection of style `ofcDataGet`. It is possible because the

variable is accessible as instance variable in any operation. The responsibility for data consistency lies with the user.

5.6.7.3 Output access via operation ofpDout()

This is a typical *"getter"* But for target code the operation can be manual written in a more complex kind. The approach "make data private" is not the only one reason to do so. Another reason to use operations for data access is also the ability to set a break point in the access operation for debugging (track when it is accessed).

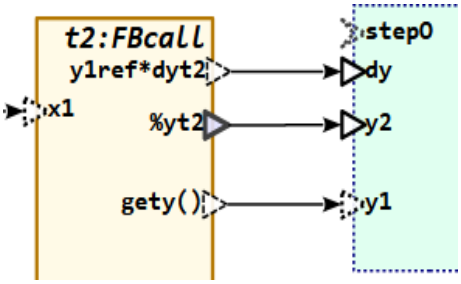


Figure 71: BasicTest/CallingConventionsUse_DoutOper.png

In the image above as simple example is shown from the test module `BasicTest#CallingConventionsUse`. The output `gety` is designed as operation access (see Figure 36: BasicTest/CallingconventionsDef.png left side).

The access is done backward from the using output `y1`. It is also possible that the output is used as part of an expression. The access in target code is similar as the access to a instance variable, only the operation is called instead access to the instance variable:

```
thiz->y1 = gety_FBcall_BasicTest(&thiz->t2);
```

Inside a target code getter operation for example a more complex access can be written. In this example the gety returns a int16 value, but operates internally with int32. It adapts the inner value:

```
static inline int16 gety_FBcall_BasicTest
(FBcall_BasicTest_s const* thiz) {
  return (int16)(thiz->y >>16);
}
```

5.6.7.4 Operation access returns the value or the reference ofpDout*()

For simple variable access via getter operation of course the value should be returned. But if the variable has a struct type then also it is sensible in C/++ language to return a pointer to data referencing this value. But then the data should be persistent in memory, stored in an instance variable (adequate `ofpVout` or `ofpZout`), never in a local variable (`ofpDout`) Secondly the problem of data consistence is to regard.

To mark a return by reference for an operation access `name&()` should be written. The access operation should be defined in form

```
DTypeVar const* name(FBtype const* thiz);
```

It means the returned reference should not be used to modify this data, only to get it.

But if it is written `name*()` then a non const pointer should be returned:

```
DTypeVar* name(FBtype const* thiz);
```

The const* for thiz means, the operation does not change the FBlock instance data itself inside the called operation. That's correct.

Hint/TODO: this feature is not yet implemented 2025-06

5.6.7.5 Access Zout values ofpZout

Zout values are values which are set with the update operation. This is a general concept, see 5.12.2 Life cycle of programs in embedded control: ctor, init, step and update page 121.

Update outputs needs the style `ofpZout...` in the FBtype definition or in the FBlock with definition on demand. The designation with `()` or `*()` after the name to access via getter is also possible and follows the same rules for access via operation in the chapter before.

How this is stored in the fbd file (IEC61499):

For the FBcl file (Function Block connection) primary the pure functionality is important. But the given property how to generate it in code is important for the target code generation. This is a property of the interface of the FBlock, written in the comment field in the interface definition:

5.6.8 Expression GBlocks

Expressions are elaborately described in the next chapter *5.8 Expressions inside the data flow (FBexpr)*. The difference between expressions FBexpr and ordinary FBlocks is: FBlocks have an inner structure, may be there are implemented specifically in the target language, or described also with an OFB module or with another source in IEC61499. Whereby FBexpr and also FBoper or completely described with its graphic appearance in the module itself.

Expressions are presented in other FBlock graphic languages usual with specific library FBlocks for different operations, such as AND, ADD, MULT maybe also with different FBlock types for the variants of number of inputs, or also with specific FBlocks for a multiplication of a signal (it's a "gain" in Simulink), or adequate operations, and for specific FBlock to access elements of a structured type or array. This causes a lot of standard library blocks and confusion.

The better variant in OFB graphic is, have only a small set of different block kinds, and use familiar textual notation of the pins to dedicated the operation.

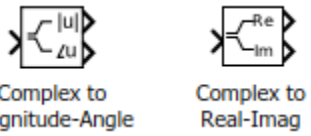


Figure 72: Simulink standard library blocks SmlikLibCplxMagnAngle_CplxReIm.png

The Figure above is an original snapshot from the Simulink System Library *Math Operations*. The both mathematics blocks looks very similar and simple. But the right block is really a simple access to the components of the complex, and the left block is a specific operation to get the angle via an arctan call and to get the magnitude via the square root of its square of the components. Both are expensive operations, very expensive if the controller has not a specific mathematics support for that.

The OFB is more implementation oriented.

For really specific simple functions you can use an FBexpr with a specific operation name in its text. This operation can immediately called with the input and output pins as arguments, implemented in the target language. Or also, the specific operation can be part of the code generation (the otx script) and generates then a simple but specific target code. Instead a lot of specific library function blocks, you have the expression with the specific operation name.

That opens also the capability to influence the operation name and hence specific adaptions only while translating to code generation.

5.6.9 GBlocks for operation access in line in an expression - FBoper

See also *5.9 Operations to FBlocks inside the data flow (FBoperation)* and *Error: Reference source not found*

This is a contribution to the Object Orientation. In ordinary FBlock diagrams one FBlock instance presents an instance (of a class) but only with one operation, or some only specific operations. For example, in Simulink S-Functions, *sample time* associations to pins are mapped to several operations). But the object-oriented world has more than one specific operation in addition to simple getter accesses as operations in one instance (class).

This approach, more as one operation for one FBlock, is settled by different events given in more as one FBlock presentation, as described in *5.6.2 GBlocks for each one function, data – event association*. The specific event maps to the operation, the associated data are the arguments of this operation. But an operation with return value, usable in line in an expression is not settled with that. Also outputs of an operation *"called by reference"* to given variables are not settled.

For that a specific expression presentation is used, the FBoper (Function Block operation):

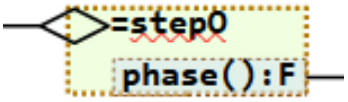


Figure 73: odg/FBoperGetter.png

The right figure shows a simple getter possible as part of an expression. The aggregation refers the proper FBlock, see also . The `=step0` means, that the operation (getter) can be called only after the `step0` output event of the referenced FBlock. It means the data to get are prepared after finishing the correspond step event. In ordinary textual languages such things are given by the line sequence (calling order). For graphical programming the events determines the order.

This getter ***FBoper*** can be used more as one time in the graphic. It is not an only repeated graphic presentation (due to *Error: Reference source not found*), it is really each an operation call for each graphic presentation.

That fact is more able to explain with the following example:

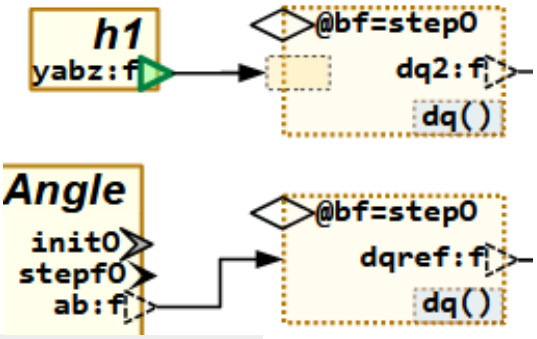


Figure 74: odg/FBoperInOut

Here two times the same operation of the same instance is called, but with different input values. The instance is in both cases the `bf` instance, textual given with the `@`connector (see chapter *5.7 Connection possibilities* page *118*).

It means, the same operation for the same instance is used twice, but with different input values. That's why it is important that the operation itself do not change internal data in the aggregated FBlock with name `bf`, given in the aggregation as connection.

The called function should be designated in C language as

```
void dq_Bandpass(Bandpass const* thiz
  , float_complex x, float_complex* y1);
```

or just in C++

```
void Bandpass::dq(
  float_complex x, float_complex* y1) const;
```

The reference to the type (to the data) `Bandpass*` is `const`. , also in C++ language given with the `const` on end of the operation declaration, regarding to the implicit `this` pointer. In Java language unfortunately an adequate designation does not exist (`final` does others). This `const` designation can be seen as contribution to the ***Functional Programming Approach***. It means, the output is only determined by the input (also the referenced data of input pointers, means the data of the instance), but no side effects occurs. This is also the approach for this ***FBoper*** constructs in OFB.

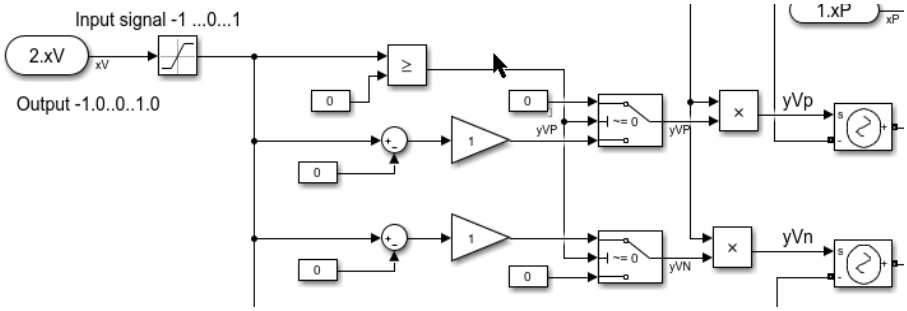
Also here, `=step0` on the aggregation means, that the FBoper can be executed only after valid `step0`, it means after `step` was executed. In source code programming this should be regarded by the line order, call `dq..()` only after `step..()`. Here for graphical programming it is deterministic in this kind. After the evaluation of the graphic it is really a ***event-Join-FBlock*** with one input of the `fb.step0` to the expression prep input. The other input to Join comes from the data input before. But because the first FBoper is feed by a `ofpZout` pin which has valid data outside the event flow, here only the `fb.step0` is connected to the FBoper. This can be seen in the produced fbd file, for this example:

```
EVENT_CONNECTIONS
bf.step0 TO dq2_X.prep;
bf.step0 TO JOIN_dqref_X_prep.J1;
gref.stepf0 TO JOIN_dqref_X_prep.J2;
JOIN_dqref_X_prep.J TO dqref_X.prep;
```

5.6.10 Conditional execution with boolean FBexpr

In textual languages the `if-else` and also `switch-case` are one of the important control structures. In the FBlock diagram world this is not simple to map.

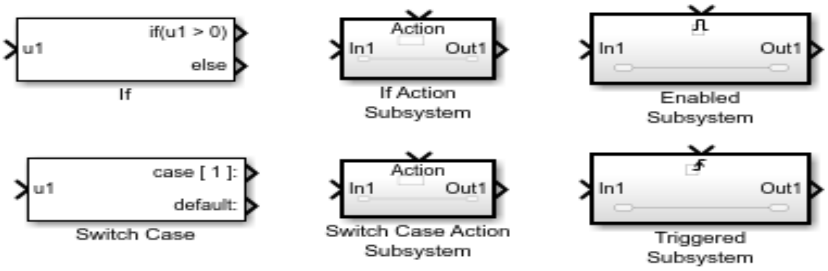
Figure 75: smlk/Exmp_if_switch.png



For example in Simulink a switch block can be used to determine that a signal is built in the one or other kind. The control input of the switch is the condition. The thinking is here backward, from the output:This example shows building a signal for `xV >=0` and another signal for `xV <0`:

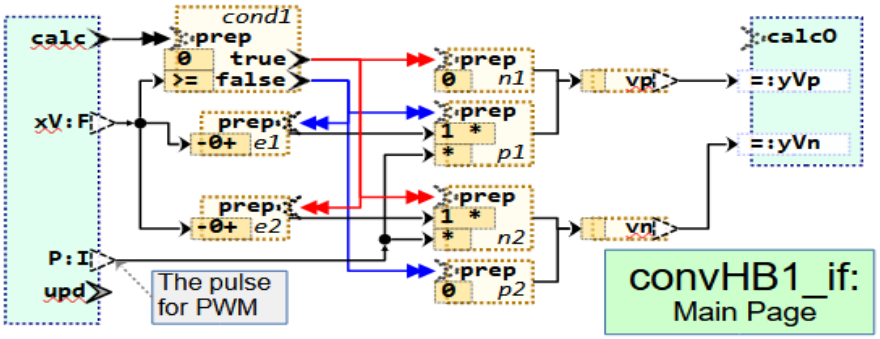
```
if(xV >=0) {
  yVp = 0;
  yVn = P * (xV-0) *1; // (P: line from top)
} else {
  yVp = P * (xV-0) *1; // (P: from top)
  yVn = 0;
}
```

Figure 76: smlk/SmlkLibCondFBlocks.png



Simulink offers some other possibilities also for conditional processing: The enabled and triggered subsystem. The internal function is only executed with a condition outside.The image above shows some specific *'Subsystems'* for conditional operations.

Figure 77: OFB/exmpTrueFalse.png



In the OFB graphic with its event orientation the conditional execution (if-else-construct) is simple. The right image presents the same functionality as the shown Simulink solution in Figure 17: smlk/Exmp_if_switch.png above, also with the not useful (for experience) some added 0 values, to compare this solutions.

The FBexpr `cond1` checks the condition. If it is true, then the `true` event triggers following the prep input event, if it is false then the `false` event triggers. Both are connected in different ways, here shown with red and blue connections. It means either the following FBlocks either the red connection are used, or the other ones. Both delivers a result on the input of `vp` and `vn` (right). It means this FBexpr data input has two concurrent driving signal, but only one is the active adequate one of the event flow. In opposite to the Simulink solution here a forward thinking is appropriate.

The code generation order is defined evaluating the event connection order, shown in a log file `convHB1_if.evTree.txt` which is generated with the option `-dirFBcl:path`

```
== calc =====
```

```
calc =>> cond1.prep   (*  :  *)
  cond1.true =>> vp_X.prep   (*  | 2 : 2 *)
  cond1.true =>> vn_X.prep   (*  | 2 : 2 *)
  cond1.false =>> vn_X.prep   (*  | 1 : 1 *)
  cond1.false =>> vp_X.prep   (*  | 1 : 1 *)
JOIN_calcO.J =>> calcO   (*  :  *)
```

The event flow is evaluated as following:

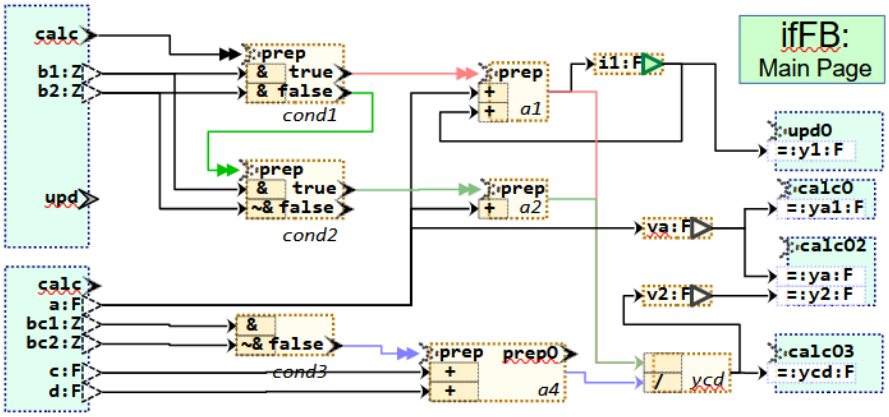
```
EVENT_CONNECTIONS
  calc TO cond1.prep;
  calc TO e1.prep;
  JOIN_calcO.J TO calcO;
  cond1.true TO e2.prep;
  cond1.true TO n1.prep;
  cond1.false TO e1.prep;
  cond1.false TO p2.prep;
  e1.prepO TO p1.prep;
  e2.prepO TO n2.prep;
  n1.prepO TO vp_X.prep;
  n2.prepO TO vn_X.prep;
  p1.prepO TO vp_X.prep;
  p2.prepO TO vn_X.prep;
  vn.prepO TO JOIN_calcO.J2;
  vn_X.prepO TO vn.prep;
  vp.prepO TO JOIN_calcO.J1;
  vp_X.prepO TO vp.prep;
END_CONNECTIONS
```

The generated code is similar as shown above:

```
  cond1 =  (0 < xV) ;  // otx: ExprEv_OFB @10'0(49..59, 23..31)
  if( cond1 ) {   // otx: exprCondIf
    vp = 0;  //cond1.true --> vp_X.prep  genExprOut(...) in otx: setVar_FBexpr
    vn =  (((xV - 0)  * 1)  * P) ;  //cond1.true --> vn_X.prep  genExprOut(...) in otx: ...
  } else {   //else  (0 < xV)   otx: exprElse
    vn = 0;  //cond1.false --> vn_X.prep  genExprOut(...) in otx: setVar_FBexpr
    vp =  (((xV - 0)  * 1)  * P) ;  //cond1.false --> vp_X.prep  genExprOut(...) in otx: ...
  } // endif   // otx: exprEndif fbx=<null>
```

But this is not the only one possibility of condition. It may be more complex:

Figure 78: OFB/exmpTrueFalse Complex_ifFB.png



The image right shows a more complex conditionally exectution. There are three conditional events in cond1, cond2 and cond3. The FBlock ycd joins signals, whereby also here the inputs comes from more as one sources. But the ycd has one input more, also conditional. It is only an example.

Look on the generation code, then it may be more understandable for a source-code C programmer. The code is original from code generation but here a little bit shortened for better explanation and presentation:

```
void calc_ifFB ( ifFB_s* thiz ...) {
  bool cond1, cond2, cond3;  // for the cond.
  cond1 =  (b1 & b2) ;  // the condition
  if( cond1 ) {   // otx: exprCondIf
    thiz->i1 =  (a + thiz->i1_z) ;  //
  } else {   //else  (b1 & b2)
    cond2 =  (b1 &  !b2) ;  // the cond.
  }
  cond3 =  (bc1 &  !bc2) ;  // the condition
  if(cond1 && !cond3) {        // otx: exprC
    //Module outputs due to the event calcO3
    thiz->mEvout_calc |= MASK_calc_calcO3;
    thiz->ycd =((a + thiz->i1_z)/(c + d) );
    thiz->v2 = 0;  //ycd.prepO --> v2_X.prep
  } else if(!cond1 && cond2 && !cond3) {
    //Module outputs due to the event calcO3
    thiz->mEvout_calc |= MASK_calc_calcO3;
    thiz->ycd =  (a /  (c + d) ) ;    // otx
    thiz->v2 = 0;  //ycd.prepO --> v2_X.prep
  }  //Condition Bits
.....
```

5.6.11 Data flow event related – or persistent data

Primary a Function Block Diagram shows the data flow – from input to output. But some values are used as states, read from stored variable:

a) from the step time before (in Simulink this is a Unit Delay)

b) from another data flow, or another operation, another sampling time (in Simulik this is a Rate Transition).

The used values comes from another event chain, they are not in the own flow. If you think in relations of *“Functional Programming”*, only the flow with the own data are proper to this concept.

In ordinary text line programming such things as *“using values from the step time before”* are solved in a simple way:

* The values are stored in instance variables after calculation.

* A value from the last step time is used, because the using code line is executed before the variable is set newly.

* For values from another operation it is similar: The values are set in the other operation, and used by access to this instance variable.

```
thiz->a = (x - thiz->a) * thiz->fa + thiz->a;
```

This is a simple PT1 algorithm, a low pass filter. `Thiz->a` is the own state variable for the filter output. The value of the last step time is used in the same line by access to `Thiz->a` in the line, and set the new value on end of this calculation in only one line. This is simple ordinary C programming. You can also write

```
thiz->a += (x - thiz->a) * thiz->fa;
```

- looks rather short and smart.

But what about a low pass filter second order. The simplest form is:

```
thiz->a1 += (x - thiz->a1) * thiz->fa;
thiz->a2 += (thiz->a1 - thiz->a2) * thiz->fa;
```

The timing values are the same (same `thiz->fa` for this example). There is a small mistake: The second filter do not use the

empty

5.6.12 Sliced or Array FBlocks, Demux and array data

In FBlock graphics usual one GBlock (graphic Block) is one FBlock. But also Simulink knows a "slicing". To explain it, look first to a simple example:

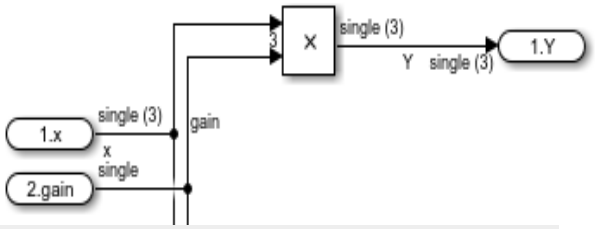


Figure 79: smlk/Exmp_Multiply_Vector_Scalar.png

Above, very simple, the Multiplier calculates a float[3] vector with a scalar gain, resulting in again a float[3] output `Y`. The graphic detects automatic the scalar of one of the inputs. From the scalar view this is a slicing. Three multiplications.

The same is done adequate in OFB graphic:

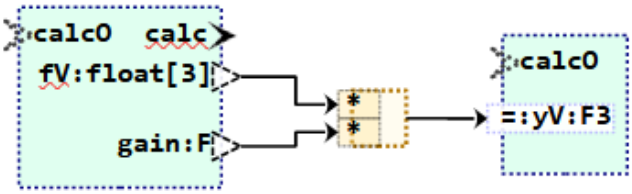


Figure 80: OFB/Exmp_Multiply_Vector_Scalar.png

The multiply expression is dedicated in the FBcl file as:

```
FBS
  d_1 : ARRAY[0..3] OF Expr_OFB( expr:....
```

It means it is an array FBlock. This is the internal information, done automatically because the connected data types.

But what about, if that isn't a simple expression (the vector-scalar calculation can be seen as a standard behavior). Instead: An only scalar defined operation or FBlock should be used with the vectored inputs. Then, thinking in source line programming, you need three or more operation calls with the appropriate instances, maybe organized in a for - loop.

Simulink has the solution of a *"For Each Subsystem"*, looks like:

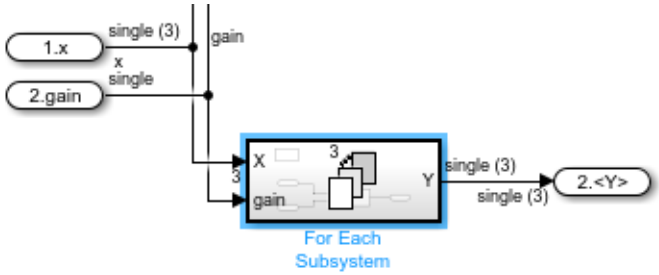


Figure 81: smlk/Exmp_Multiply_Vector_Scalar.png

From outside it is an *FBlock Subsystem* with the vector and the scalar input, and the vector output, as necessary.

Internally this specific "*Subsystem*" has for-each pins for X and Y, which are outside vectors. It looks like:

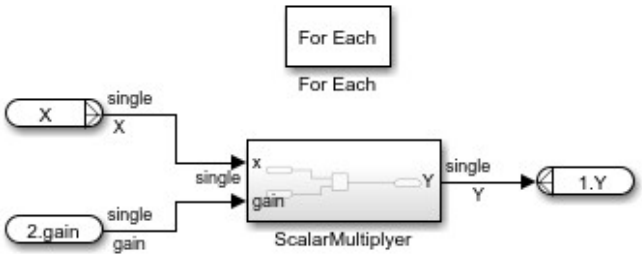


Figure 82: smlk/Exmp_ForEachSub_InnerScalar Mult.png

Internally the FBlock which should be used three times, or more times depending from the vector size, is contained only one time. In Simulink there is a dialog box opened in the '`For Each`' Block. The dialog determines (in several kinds) what should be happen with the specific pins `X` and `Y`. The '`ScalarMultiplyer`' FBlock is only an example for a more comprehensive only scalar FBlock used with vectors. The code generation creates more as one instance of this FBlock type, and organizes calling in a for-loop or one after another (depending on some settings).

In OFB graphic a similar but more user-simple and obvious solution is given:

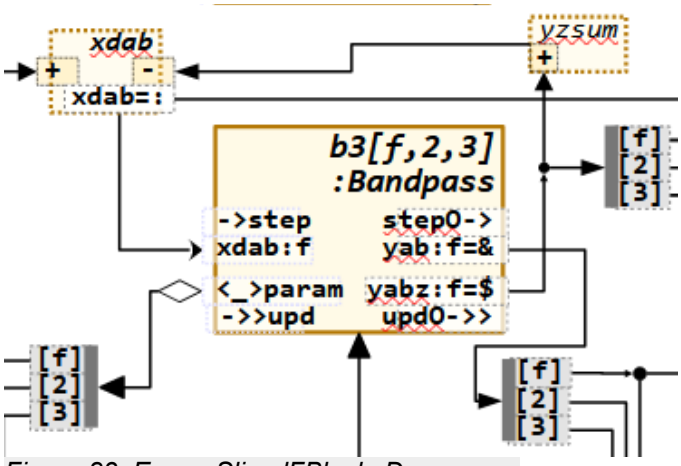


Figure 83: Exmp_SlicedFBlock_Demux.png

In opposite to the Simulink approach to encapsulate the 'For Each Subsystem', here all is organized in the module level. You see the inner implementation (in *Figure 82: smlk/Exmp_ForEachSub_InnerScalarMult.png* the '`ScalarMultiplyer`' immediately instead additional wrapping. The image above shows one Graphic Block which presents three FBlocks with name b3f, b32 and b33. The writing style of the name is described in 3.3 Texts in graphic blocks and pins page 8. It is not a vectored FBlock instance, but three named instances. Also a vectored instance is possible here, for example designated as `b3[3]`. But the named instances are the user decision, it works.

(empty)

The output `yabz` for the three instances is a vector. Hence the input of `yzsum` is a vector. But the output of this expression is scalar, because of back propagation of `xdab` input, which is scalar. Hence this input '`+`' on `yzsum` presents three inputs which are added together.

The `xdab` output is a scalar, because the incoming + input on this expression is scalar. This scalar value is applied to all instances of the b3[f,2,3] graphic block with the same value.

The `yzsum` expression gets on its `+` input pin three signals, from the three instances of the sliced FBlock `b3f.yabz`, `b32.yabz` and `b33.yabz` via three input connections. Because this pin is a multiple pin (see *5.7.12 More outputs to one input* page *126*), the three connections means three independent `+` inputs.

Furthermore, you see multiplexer and demultiplexer, here only demultiplexer. The output `yab` or also `yabz` is used for all instances in a different way, and the demultiplexer organizes the access to the correct FBlock of this drawn GBlock.

The aggregation `param` goes to three different parameter FBlocks. In the current implementation there may be also a sliced GBlock for parameter, hence the demux is not necessary, a simple connection between to sliced GBlock means the 1:1 connection of each FBlock representing the sliced GBlock.

5.7 Connection possibilities

Table of Contents

5.7.1 Pins

Connections between FBlocks (or first between GBlocks in the graphic) are drawn using Connector in LibreOffice draw with a dedicated style. The connections are connect to glue points in Office draw either to pins or to the GBlock frame. Connections to GBlock frames forces default pins '`pinFBsrc`' and '`pinFBdst`', which are mapped to real pins in the FBcl data.

The pins are either formed shapes or simple rectangle with a dedicated style. The pin appearance itself does not play any role for the interpretation and converting of the graphic, this is essential only for manual view. For interpretation the associated style is essential. Also in different situations the style of the connector is essential if the pin is not complete dedicated.

Compared with UML class diagrams, there are no pins, only connections between the class blocks as relation of the classes (aggregation, inheritance etc.). Here only the connector style determines the existence of the relation ***between*** classes. This is other than in ordinary programming languages, where the fact of an association to another class is given as property of one class by the definition of a pointer variable with the appropriate type. Whether it is an aggregation or association or composition, is given by the context (final variable in Java are never associations, there are aggregations if they are set in the constructor from arguments, or just compositions if the instances are created in the constructor). The showed relation between classes in UML is intrinsically only a kind of shown documentation. In OFB the pin play the role of define an aggregation, composition etc with the given type, also without showing the relation between (means more exact to the) destination class. For that look on Figure *Figure 84: odg/FBpin_ofPinOnly.png*, on the pin `param`. Without connection it is already designated as aggregation due to the <_> on start of the pin description text. But here the type is missing. The type is possible also in the description text (see 5.3 Texts in graphic blocks and pins page 8), but here it is given with the connected destination class. Because the connection style is an aggregation (`ofcAggr`), the <_> in the pin description is not necessary, but possible.

For the pins the simplest variant is, have a text field with the common style `ofPin`. Then the kind of the pins is determined by specific leading a d trailing pin kind designations, as able to see in the next figure, or also by the kind of the connection.:

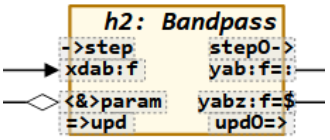


Figure 84: odg/FBpin_ofPinOnly.png

The pin kind designations are described in *5.2.4 Connector styles, ofc* page *46*. But it should be understandable. The events are designated with arrows `->` `=>` because it's the meaningful execution flow. The outputs have a `=` in the last but one position and a `$` in the last for a *"State"* variable. Aggregations have the `<>` as a diamond (UML) and the `&` know as reference designation in C/++.

The diamond on the aggregation connection is for viewing, it is twice here, the `<&>` cannot removed. But see next image:

Data connection:

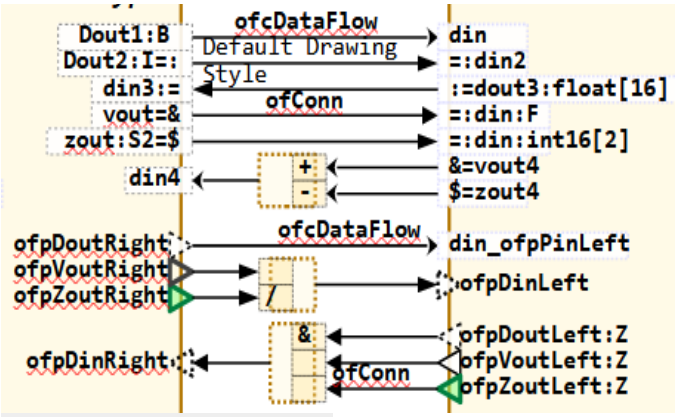


Figure 85: PindefDinout.png

The image above shows a detail of the https://vishia.org/fbg/deploy/OFB_DiagramTemplate.odg (*5.2 All Elements with their styles* page *42*) for data pins. The first (top) two pins left and right are **determined** as `Dout` and `Din` **due to the connector type** `ofcDataFlow`. The rectangle for the pin has the style `ofPinLeft` or `ofPinRight` or `ofPin`. The difference of this styles are only in appearance, bounding the text left or right side or in the mid of the pin, whereas for ofPin the bounding can be clarified by direct formatting in the *"Format – Text attributes"* dialog (recommended using key F3, see *3.8 Appearance of the GUI in LibreOffice draw*.

For the first pin left the **data type** `:B` is given, which is `Byte` or `int8`. Right side the data type is propagated by the connection, hence not necessary but possible to draw..

The next both pins `Dout2`, `din2`, `din3`, `dout3` are connected with a default connector style, which is adequate to the `ofConn` style in the below following `vout` and `din` connection. The connector style has no contribution. Using this style is more a fast choice. **The kind of pins are determined by** `=:` and `:=` whereby the `dout` is dedicated by `=:` on right side or `:=` on left side. It is equivalent to the assign operator known from Algol, Pascal and Structure text, the `:` is on the side of the assignment. If the `=:` or also `:=` is in the mid of the pin text, then it is always a `Din`. The `=:` from left or `:=` from right separates then input data preparation from the name and type information, as described in *5.3 Text in graphic blocks and pins* page *50*.

The below following pins `vout4` and `zout4` are determined as `ofpVout` and `ofpZout`. This pins are variable in the structure context of the FBlock (`Vout`) or a state variable (`Zout`). A state variable is updated by the update event, hence have the value from the step time before. A `Vout` variable can be accessed also in another event chain (other operation) but then without guaranteed consistence to other data. Use `Vout` variable if they should be monitored from outside.

The pin appearance below is the alternative. The triangle figures symbolized the pin itself, the text to the pin is written outside, left or right beside. This is a little bit sophisticated in LibreOffice, but possible. Here dedicated pin styles can be used (as shown as pin texts) because the specific appearance is only related to the small triangle. Hence the textual dedication with `=:`, `=$` etc. is not necessary.

Figure 86: odg/FBpin_ofp.png

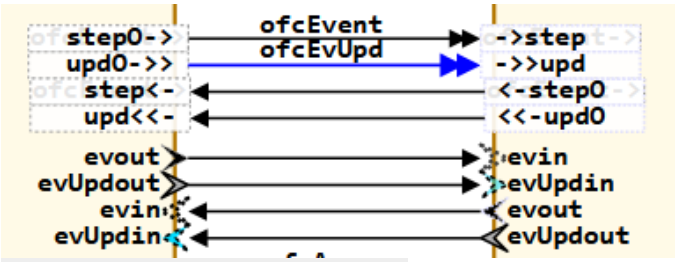


Figure 87: OFB/PindefEvinput.png

That are pins for events from the template. As also for data pins first the rectangle variant with `ofPin` style and the textual designation of the pin kind. The textual designation is not necessary if the designated connector styles are used, but it is though recommended. On removing the connection elsewhere the pin kind is undefined. Furthermore, often event pins are not connected because the connection

is automatically found (see *5.12 Execution order, Event and Data flow, Event chains and states* page *196*

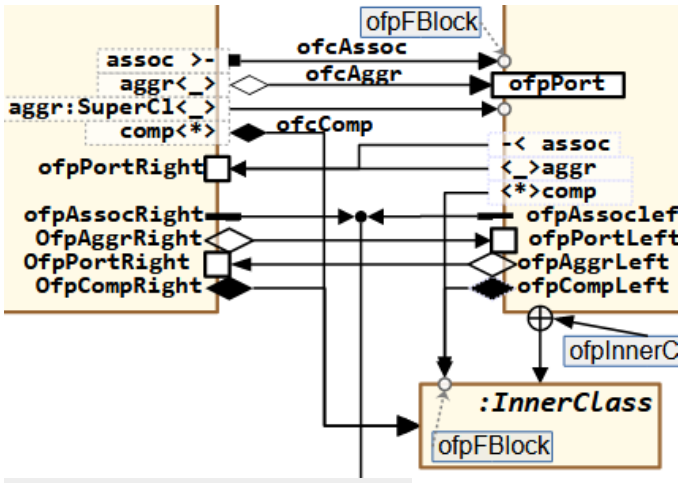


Figure 88: OFB/PindefRefPort.png

This immage above shows references between instances, ports and an inner class.

`Assoc>-` This is a pin with **`ofPinRight`** style. It becomes an association because of the **`>-`** right side in the text. It becomes an association also because of the style ofcAssoc of the connector.

`OfpFBlock` This is a small pin which is an alternative to a glue point of the FBlock. In this example the association is initialized with this FBlock instance. It means the connection should go to the FBlock itself. Target glue points are in the mid of the edges. Additional glue points are possible. But the disadvantage of glue points is, they are oriented to the FBlock rectangle in a relative metric. Changing the rectangle shifts the glue points. With the **`ofpFBlock`** this disadvantage is prevented. The **`ofpFBlock`** pin is independent of the FBlock rectangle (but shut manually positioned on the edge). To copy and move this very small **`ofpFBlock`** shape capture it with a lasso and move it with cursor keys.

`aggr<_>` The pin text **`<_>`**, which symbolizes a non filled diamond, determines this **`ofpPin…`** as aggregation, should be written left or right side.

`Comp<*>` The **`<*>`** should symbolize a filled diamond. A composition (UML) should never refer an instance but a type. Here all compositions goes to the **`InnerClass`**. Also another class as the own outer class can instantiate an Inner class of another type. See *Error: Reference source not found*.

`ofpPortRight` The small square symbol as also the same style **`ofpPort...`** as rectangle with internal text describes a port of an class or instance as in UML. It is the destination for references, beside the whole class or instance. It describes an inner instance in a FBlock whose reference is used. See *Error: Reference source not found*.

`ofp…` styles and symbols: This are the pin symbols with its styles who can be use instead the rectangle boxes of style **`ofPin.…`** Because the pins are always determined in its meaning, a simple connection **`ofConn`** can be used to connect. Note that for the **`ofpPort`** a textual dedication for a simple rectangle with **`ofPin…`** does not exist. Use always the **`ofpPort...`** style also for a rectangle pin.

`ofpInnerClass`: This is a pin symbol also used in UML to dedicate a relation from a class FBlock to its inner class Type as FBlock. An inner class can be referenced as a port of the outer class. In this example the inner class is instantiated and referenced by the compositions, but also aggregations or associations can refer it (via port). It means the relation with this **`ofpInnerClass`** style symbol is not a reference on runtime, it is a relation between the types of the FBlocks. An inner class is usual defined in the name space of the outer (environment) class and can also access private members of its outer class. See *Error: Reference source not found* how it is mapped to programming languages.

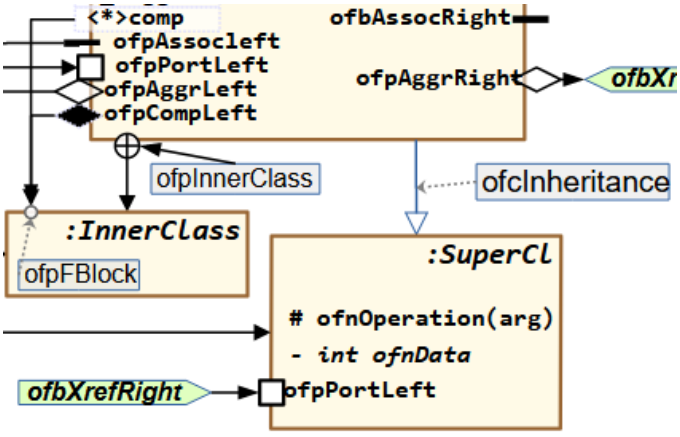


Figure 89: OFB/PindefRefPort.png

This image above shows the right continuing of the https://vishia.org/fbg/deploy/OFB_DiagramTemplate.odg (*5.2 All Elements with their styles* page *42*)

`ofcInheritance`: This is the inheritance relation between the two types shown as FBlock as used in UML. It uses the **`pinFBsrc`** and **`pinFBdst`** of an FBlock for the connection in FBcl. On the connection also the symbol with ofpFBlock can be used instead connecting immediatelly to the FBlocks with glue points. Note that the instance of a super class (from inheritance) is always the same instance as the defined Object of the inherit FBlock.

To edit the text in a pin select the pin and press <F2>. It is the same as *“Insert Text box”*.

To modify the pin text placing you can use the following dialog *“format – text attributes”* or maybe set to <F3>:

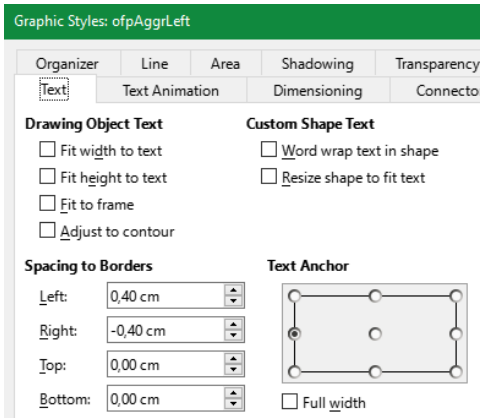


Figure 90: odg/Fbpin_ofpStyleText.png

The figure above shows the necessary settings to place the text right side to the shape of length 0.4 cm.

5.7.2 name : Type on pins

See also 5.3 Texts in graphic blocks and pins page 8. A type on a pin is necessary one time in usage of the pin in the graphic. If the same pin is used in several GBlocks (for more as one FBlock instances of the type) it is sufficient to write the type only one time. The type of the pin is stored in the `PinType_FBcl` instance due to the `FBtype_FBcl` one time for the type definition.

The type information can be given in another graphic (for another module) or also in a read FBcl file read before.

Also, the type is propagated due to the data flow, see *5.4.6 Data type forward and backward test and propagation* page *63*, and in this kind stored in the `PinType_FBcl` for all usages.

The name of a pin should be an identifier as usual in programming languages, also due to the rules of the target language.

There is a special feature: If the name ends with 1999 or 0999, the pin is a so named **multiple pin**. If more of this pins are used in the instance, pins from X1 or X0 counting up are used in the instance, and enough pins are built in the FBtype. This is especially used for expressions.

5.7.3 Connectors

It is very simple to draw a connector from an output to an input using the

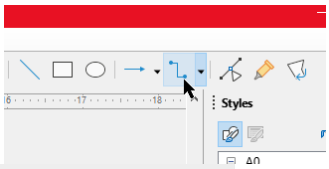


Figure 91: odg/Connector-Icon.pdf

The handling with the glue points is a little bit sophisticated in LibreOffice draw. Press the mouse in the near of the source glue point but outside of the appropriate pin, and release the mouse also outside near the glue point. The used shape for glue is highlighted.

Select the necessary `ofc...` style after glue.

See *5.2.4 Connector styles, ofc* page *46*.

It is also interesting to have a line connector:

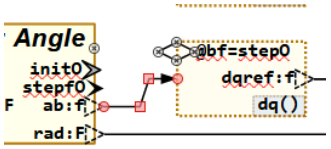


Figure 92: odg/LineConnectorExmpl1.png

This gives sometimes a better appearance of the graphic as only the known rectangle connectors as in other tools. The line connector is a given feature in LibreOffice as also the Curved and the Straight connector.

5.7.4 Connection points

One fast usable possibility is to organize the connectors from the source with proper positioning:

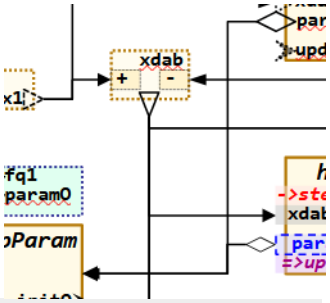


Figure 93: odg/LineConnectorExmpl1.png

The figure above shows three overlapping connectors, twice from `par…` to the destination FBlock, three times from `xdap` output, and twice from left top `x1` output. The lines are proper overlapped so that the graphic is proper visible. The grid snapping of 1 mm helps to get proper lines.

But an also proper sometimes better variant is using connection points:

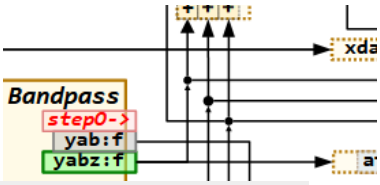


Figure 94: odg/ConnectionPoints1.png

From yabz two connections goes out overlapping, but one of them goes to a connection point. This is a filled circle with the style `ofbConnPoint`. The mid connection point has a diameter of 1 mm, the other both have 0.8 mm, maybe better. The incoming connector has the style `ofcConnPoint`, which results in the viewable very small but visible arrow (size 0.6 mm). The positioning of the connection point should be in the 1 mm grid. For that the position dialog should use the mid point:

The position can be tuned simple with pressing <F4> with the standard key settings in LibreOffice. You should select the Base Point in mid, then adjust values smoothed to 1 mm. Then the resulting connected connectors are also in the 1 mm grid as seen in .

The connection points are too small to move it with the mouse (unfortunately, should be improved in LibreOffice). But it is simple possible to move it with the arrow keys after copying from a smoothed position. This works fine, better as in some other tools.

It is also possible to connect connectors on its end. Sometimes this is only necessary to draw connection lines in a more complicated kind. See also *3.5 Connectors of LibreOffice for References between classe* page *10*

5.7.5 Xref

This is already described in *3.7 Diagrams with cross reference Xref* page *12*. A Xref shape is from type `ofbXrefLeft` or `ofbXrefRight`. Left and Right are only for the appearance, the text position. The shape form can be copied from the template or other given odg files. But the shape form is only for viewing. Any rectangle or text field can be used.

The incoming connections to a Xref are connected with the outgoing connections similar as in a connection point. All Xref with the same name are existing only once in the graphic data (only one `OdgXref` instance for several GBlocks). The Xref instances are only existing in the odg data map, in the data for code generation they are dissolved already.

5.7.6 Using GBmux and GBdemux for connections

A GBmux is a first graphic block to assembly different signals, often referred as "*multiplexer*". The opposite GBlock is the GBdemux for demultiplexing. Whereby the term *"multiplexing"* is a reference to hardware solutions, where different signals are transmitted via one line. This is not really similar. The demux pins are only designations for the signals that are connected in the graphic with only one line or with only one Xref.

As described in *5.6.12 Sliced or Array FBlocks, Demux and array data* page *116* or more detailed 5.10 FBlocks in slices, access to slices page 106 *GBmux* are necessary to offer signals for sliced FBlocks and *GBdemux* to get signals from slices. But this blocks can also be used to simple assembly signals to have only one connection line for it. In Simulink Buses are used for that, also in other graphic tools buses or multiplexed signals are usual.

5.7.7 Connections from instance variables and twice shown FBlocks

Instead necessary using of Xref to connect stuff over some pages, the possibility to show the same FBlock with a second GBlock may be more proper:

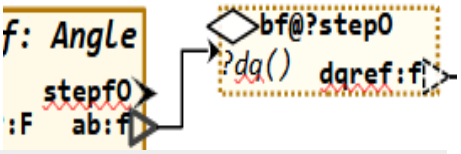


Figure 95: odg/ConnectionFromFBlockOut.png

The figure above shows the FBlock with the name h1 only because its output is used. The viewer of the diagram may better recognize which factual context is given. One should not take the detour via the Xref. But this is only possible for outputs of existing FBlocks, not for outputs of expressions, because they cannot be shown twice.

It is more simple to show only the variable as shown in the next example:

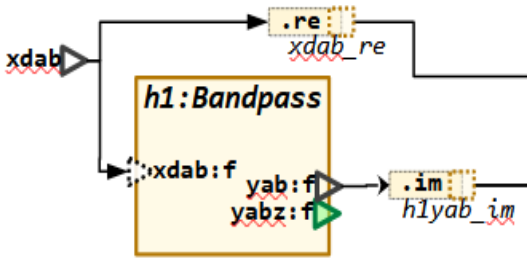


Figure 96: odg/ConnectionFromVariable.png

The variable xdab is an output variable from an expression. An expression cannot be shown twice, but the variable can.

It is also possible to lets start a connection not from its output, but from any input which is connected with an output. This is also an interesting possibility. It is in the as start the connection on the input xdab from h1, instead giving the expression output variable. Because the connection from the expression output xdab to this input is already given on another page, see page *116*

5.7.8 Textual given connections

It is also possible to write the connections simple as text:

Figure 97: odg/ConnectionFromText1.png

The image above is a showing example. Instead the immediately connection exact the expression output variabel fq3 is used in `fq@fq3` . After the `@` after the input variable name either a `Fblockname.pinName` can be written, or the `varname` of an output variable from an expression, or also the `label` from a Xref. The translator searches the proper element and connect the input in the same manner as using a graphical connection.

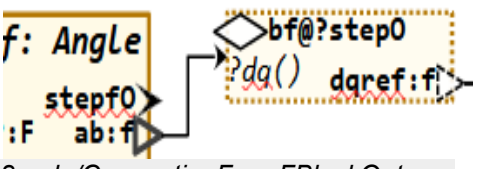


Figure 98: odg/ConnectionFromFBlockOut.png

This image shows also the connection from FBlock output but also the textual connection for the aggregation. The aggregation itself hasn't a name, not necessary. But the `@bf` describes the connection to the FBlock with name `bf` as aggregation for this FBlock operation. The `=stepO` is the here necessary designation of an event order, see *5.6.9 GBlocks for operation access in line in an expression - FBoper* page *110*

The graphical connected variant for an adequate approach is shown in:

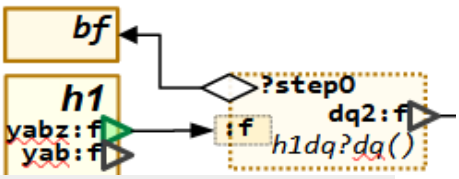


Figure 99: odg/FBoperGetterAggrConn.png

Here the `h1` FBlock is aggregated and shown immediately in the graphical context.

5.7.9 Admissibility check of connections

On drawing in LibreOffice an admissibility check is not done, it is not a feature of LibreOffice, and secondly faulty connections should be firstly possible before correction of it. (It is a non proper behavior of some tools to strong forbid faulty inputs as intermediate state). But the admissibility of connections is checked on translation of the graphic.

Hence it is recommended to translate the graphic (needs only a few seconds) during editing time to time. Also the translation result can be checked (compare with result before) to see what is happen for different graphic inputs.

So an error or misunderstanding can be early seen.

Connection ends bound to an input are always an error. A connection start on a input is admissible, because it is associated to the output driving the input. Connecting on the input is only a more simple drawing possibility.

Faulty usage of pin kinds (styles) and connector kinds (styles) is reported, should be simple correct. Also connecting of faulty pin kinds is reported as error, for example using a data input for an aggregation destination.

5.7.10 Data type test and conversion on inputs

Other than some other FBlock tools, connection of outputs with another data type as the given input is admissible. On target code generation an automatic value casting is inserted, so that the data type casing is also obviously visiting the target code.

The reason to do so is: It saves FBlocks only for converting the data type, the graphic is more clearly arranged, not overloaded with maybe formally stuff. If a data type adaption is really necessary and should be obviously in the graphic, then an expression can be inserted with the necessary cast output data type, if necessary with an additonal local output variable or also as `ofpExprOut`.

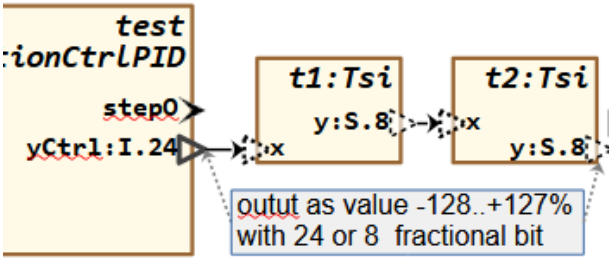


Figure 100: odg/ExmplAutoCastData.png

The image above shows data connection between an int32 output with 24 fractional bits and an int16 input also with 7 pre-factional bits, presenting the documented value range. The input x has the same data type as y, as known by the properties of the FBlock, do not necessary to show in the graphic twice. But also the data type S.8 can be documented on the input. That's all, the situation is proper and sufficient shown in the graphic.

The code generator creates of course a cast and shift to adapt both data presentations:

```
(int16)((test->yCtrl >>16) & 0xffff)
```

5.7.11 The direction of references and the data flow

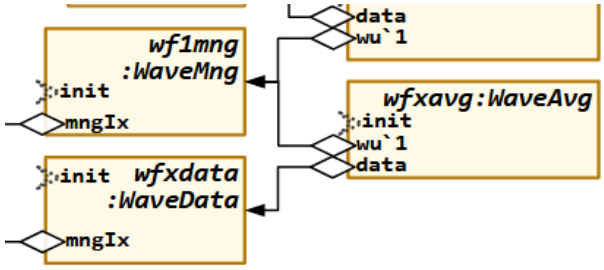


Figure 101: odg/ConnectionAggregation.png

In the UML a reference (association, aggregation, composition) is drawn from the using class (has a reference) to the used class (represents the type). But the data flow: Set the value of the reference to the referenced FBlock, is in the opposite direction.

The data flow is against the arrow of the aggregation. The image left shows some references, from a FBlock `wfxavg` of type `WaveAvg` to build average values, to its management FBlock `wf1mg` and to the data containing FBlock `wfxdata`. That are UML thinking and also *"reference"* thinking *"… the average FBlock needs and hence references the data FBlock"*. But the data flow for the C code generation goes in the opposite direction *"the `wfxdata` FBlock gives its reference to the using FBlock for the average."*

5.7.12 More outputs to one input

Usual in FBlock Diagrams an input can be driven by only one data output. But for more flexibility this is not the strong rule for OFB diagrams, there are possibilities:

Conditional connections: The data output can be used conditionally. This is described in *5.6.10 Conditional execution with boolean FBexpr* page *112*. One data input can have more driving sources, each for each condition.

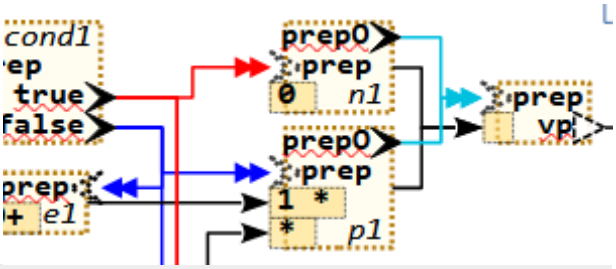


Figure 102: OFB/ExmpTrueFalseConnConditional.png

The image above shows two event and two data inputs to the FBexpr `vp`. In the evout pin the specific condition is stored. On code generation this both event chains are separated, the two `prep0` from `n1` and `n2` triggers or arrives the one `prep` evin of `vp` in different branches. On any trigger the correspond data connection either from `p1` or from `n1` is found due to the condition, which is stored also in the `prep0` evout as also proper to the data via its associated evout (the same). For this example look also on *5.6.10 Conditional execution with boolean FBexpr* page *112*.

But as in the next image shown, the variable vp can also drawn twice instead have two connections to the variable:

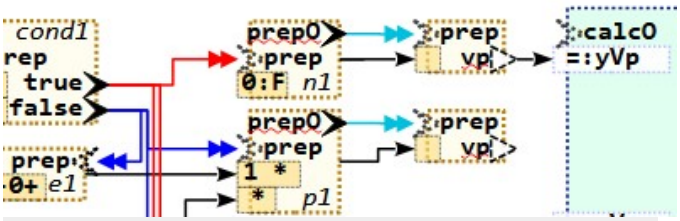


Figure 103: OFB/ExmpTrueFalseConnConditional1.png

The effect is the same. But here the expressions can be contain different things. The data flow is joined only on the vp variable, it is one and the same variable. That's why the connection to the output yVp may or should be drawn only one time.

Hint: the light blue-cyan event connections are not necessary to be drawn, because they are determined already by the data flow. It is only here drawn for understanding, it can be drawn.

The next should no more supported:

Variant connections: It is possible that one `evin` is driven by different `evout`, and also different associated data are driven by these events. For example a FBlock as part of a user defined class can be called with different inputs, and also different usages of outputs. Or the values of a structured data or of `doutMdl` pins are set with different values by different events. In this cases the connections are marked with a variant:

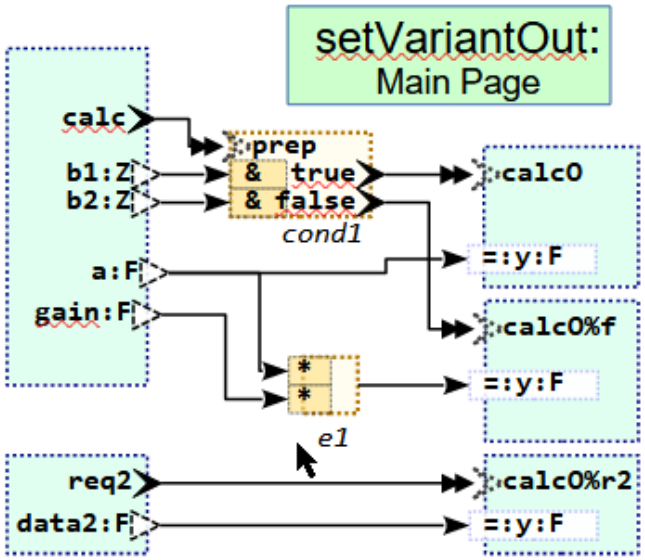


Figure 104: OFB/setVariantOut.png

This is an example. The output y associated wit the event calcO is set in three variants:

* If cond1 is true, then input a is output.

f: If cond1 is false, the expression result of e1 is output. a) and b) are similar a condition as shown in the image left.

r2: If another event comes with other data, here req2 with data2, the output is set instead with this data. This is a usual normal practical approach: In text line C language or any other language you can set data as you want. In many cases it is sensible. Data are delivered under different conditions, but the output data are all related to the `calcO` event.

This is slightly different to the conditional connections in Figure 43: OFB/ExmpTrueFalseConnConditional.png, because the connections are not able to associate to a determine conditions. They are independent, anyway the r2. Hence, they should be marked on user level. Automatically distinction is not possible. For that the output blocks for the module outputs (or also FBlocks if the same FBlock is triggered in this way) are separated graphic blocks. That is the first one. The second one is: The data and events which are associated should be marked. This is done already if only one data or event is marked, favored the event. The mark for the variant is the `%f` or `%r2` after the event name.

The syntax for variant designation is:

```
nameVariant::=<$?name>[%<*?variant>]
```

This is ZBNF syntax, see 5.3.1 Syntax in colored ZBNF page 10. The name is an identifier. The % should follow without spaces. The variant is all text of the descr part of the pin text. It means the variant can contain any character for descr. But it is recommended to use also only an identifier also for that.

If one pin of the GBlock is designated with the variant, the GBlock is the variant, all pins (more exact the connection to the pins) have the information of the variant. That enables the correct association of the data for the given triggering event for code generation.

The code for this example in C language is:

*include:../../+BasicTest/cpp/genSrc/setVariantOut.c::2'bool cond1'*2!1'}'::43::-//::*

```
  bool cond1 = false;
  cond1 = (b1 && b2);
  if( cond1 ) {
    thiz->y1 = (a);
  }
  if(!cond1) {
    thiz->y1 = ((a * gain));
  }
```

include:../../+BasicTest/cpp/genSrc/setVariantOut.c::'Operation req2(...)'!0'}'::43::-//

```
/**Operation req2(...)
 */
void req2_setVariantOut ( setVariantOut_...
  , float data2
  ) {
  thiz->y1 = (data2);
}
```

More Sources for a multiple pin: For all multiple pins (see *5.7.2 name : Type on pins* multiple connections to one pin are the same as more pins. Especially for expressions more sources for an expression input are the same as more drawn expression inputs with the same designation.

TODO OFB diagram examples.

Empty yet_A

5.8 Expressions inside the data flow (FBexpr)

Table of Contents

Expressions are written in textual languages usual with a longer line ending with an assignment, for example:

```
var = a / (b - c) * sin(2 * w + dw);
```

Expressions are also used for arguments for operations, as in the example for the `sin(...)`. It means one expression in this example is `2 * w + dw`, the other expression is `a / (b - c) * sin(...)`.

This presentation use the wording *“Expression Part”* or short *ExprPart* with the style `ofbExprPart`. Related to this example one part of the expression is `a`. The other part, for division, is `(b-c)`, which is an inner expression. And the third part is the `sin(...)`. It means each ExprPart is related to the expression as one contribution. The whole expression is here an `MULT` and `DIV` expression, the inner expression is `ADD/SUB`. The expression for the argument of the `sin(...)` consists of an `ADD` expression with `2 * w` as one ExprPart, presenting an inner `MULT` expression, and the other part `dw`. Note: `w` should be a synonymous for *“omega”*, an angle.

Each ExprPart has an an operator. In an `ADD/SUB` expression this can be `+` or `-`, a `MULT/DIV` expression can have `*` or `/` in a non deterministic order. But you should regard that the order of operation may be have effects in the target execution. If you calculate (`a / 2 * 3`) in in integer artithmetic, you get `0` for `a = 1`, whereas `(a * 3 / 2)` is `1` for `a = 1`. This is an effect in the numeric calculation in the target processor. The commutative law in mathematics is unfortunately not valid for integer numeric calculation.

Why this preliminary considerations? In Function Block graphics the parts of an expression are usual sorted in this order. An expression FBlock for this arithmetic is either an `ADD` or a `MULT`, or some more possibilities as also for OFB available.

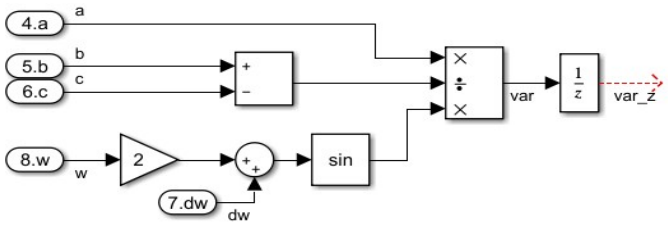


Figure 105: smlk/Exmpl_ExprAddMul-Smlk.png

The image above shows the graphic in the tool *Simulink* (R) *Mathworks*. The `MULT` expression is right with [x], `ADD` expressions are usual in two kinds, as rectangle or as circle. The `MULT` with factor `2*w` is shown as Gain FBlock.

For OFB a similar approach is used. But there are not different library elements for the kind of expression, OFB knows only one GBlock with the style `ofbExpression`. It has input pins as `ofpExprPart`. and possible outputs either as variable or as `ofpExprOut` for specifics. The functionality is determined by operators and by given operation names with parenthesis following.

The same functionality in OFB graphic is presented as:

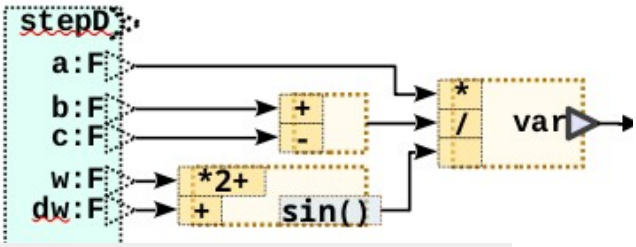


Figure 106: OFB/Exmpl_ExprAddMul.OFB.png

Also here expression GBlocks are strongly separated for `ADD/SUB` and `MULT` including `DIV`. Both cannot be mixed, it would be elsewhere confused by the application of the precedence rules for this both operation types. Hence the inner expression for `ADD/SUB` is an extra GBlock or FBexpr.

As it can be seen in the expression for `sin()`, the name of specific mathematics functions is written with `()` in the pin with style `ofpExprOut`. The translation to the used operation of the target language, in C/++ `sinf(float arg)` is necessary, is controlled by specialized script entries for all possible functions in the translation templates (gTxt). These scripts can regard the current data type to create the correct function name.

As also in this example GBlock, the expression part can contain the necessary inner expression, here `2 * w`, if the factor is a constant or also a variable which is textual referenced.

The expression FBtype `Expr_OFB` for all expressions has a second input `K...` for each `X...` input and a designation about the used operator on its `expr` input. This allows one additional factor or summand per input, per ExprPart, which makes typical expressions of kind `a*x + b*x2` more simple and obvious in graphic, it saves the effort of individual FBlocks. The expression should be presented or read in the graphic in data flow direction. From input `w`, take it, multiply with `*2` and offer it for addition in the ADD/SUB expression `+` The other part is then, input dw, offer it for addition `+`, and as last, output action build the `sin()` from it.

The resulting generated target code for C is:

```
var = (a / (b - c) * sinf (((w * 2)  + dw)));
```

Furthermore, expressions for comparison can create true and false events to control the execution flow. But refer the further documentation in this main chapter.

The general difference between Expressions (FBexpr) and FBlocks is: FBexpr have no state. There are always calculations from input to output. The other difference is: The code generation is completely done only from the information in the expression in graphic level. It is complete. Whereas FBlocks have their inner functionality either given by a graphical (sub-) module or in the implementation language.

5.8.1 Expression as rectangle and input pins as rectangle `ofpExprPart`

Expressions for data flow are presented by a figure (here a circle, but usual also a rectangle) of the style `ofbExpression`. This figure can immediately be connected by `ofcDataFlow` connectors or simple `Default Drawing Style` or `ofConn` for input and output, whereby the input connector can have a text for the expression.

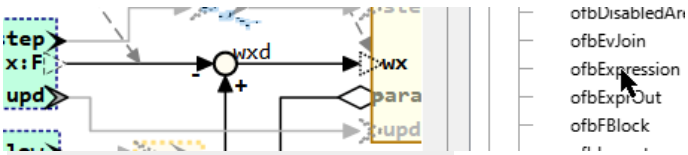


Figure 107: odg/ExpressionExmp.png

In the figure above, the name `wxd` is the text on the circle itself. It should be placed proper using the Dialog in LibreOffice: "*Format – Text Attributes*".

This is the form known also from other FBlock graphic tools. But writing a text to a line with some inflection point is a little bit sophisticated in currently LibreOffice versions.

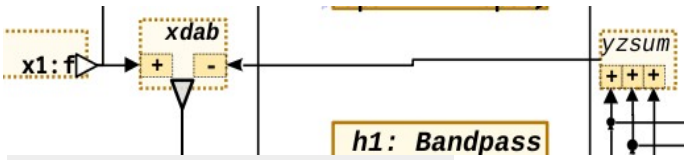


Figure 108: odg/ExpressionExmp.png

The other possibility is using a rectangle box with the style `ofbExpression`. The original outfit of the style is a dashed line as border. Small inner rectangle shapes with style `ofpExprPart` can be used for the expression inputs. The image comes from the `ExmplBandpassFilter.odg`.

5.8.2 Expression data input pins DinExpr `ofpExprPart`

Expression data input pins are designated with the style `ofpExprPart`, as part of the expression. They are designated here in the description with the shortened word ***DinExpr***.

The internal type of an DinExpr is `>>DinExpr_FBcl`. This is for further internal information, the link goes to the Javadoc of the OFB Translator sources.

5.8.2.1 Possibilities and syntax of the text of DinExpr

The text in an DinExpr is well sorted but has many possibilities. That's why here first an overview should be given to understand the possibilities with there syntax. General the syntax of all excluding [*KinExpr*] [*OpExprPart*] [*castExprPart*] is the same as also for pins on FBlocks, see 5.3 Text in graphic blocks and pins page 42.

- *OpExprPart*: The central meaning of the text in DinExpr is the **operator for the ExprPart**. The following characters respectively combination are used as operators: `+ - * / % >> << & | v ^ ~ < > <= >= == <> !=`.
- Another important element is the **assign operator** `=:` or also right side as `:=` for inputs and its handling.
- Operators of the ExprPart and the assign operator ***can be omitted*** if the meaning is dedicated without it, and the syntax is clarified. This is in definitely but often used cases, see examples.
- Left side of the `=:` or also right side of the `:=` some **input handling** can be textual given.

All is optional. The next line shows an overview, defining also the order of this text parts:

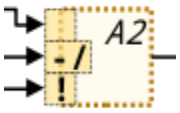
```
[Input] [InputAcc] [castInput] =: [KinExpr] [OpExprPart] [castExprPart] [setElemOutput]
                   [?specialDesignation] [`PinOrder] := [Input] [InputAcc] [castInput]
```

Instead the name of the pin in FBlock pins as *descr* here the *descr* is:

```
descr::= [KinExpr] [OpExprPart] [castExprPart].
```

OpExprPart: Operators

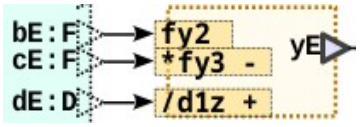
Figure 109: OFB/DinExpr1.png



- A pin without any text should have a connection, and contributes to the expression with the default operator of the expression. It is + for ADD expression, * for `MULT/DIV`, etc.
- `-/` One or the right side operator defines how the DinExpr contributes to the expression. See *5.8.2.3 Operator combinations of DinExpr determines the type of the expression* page *133*. A **second operator before** (left side) is the **unary operator** to the input. It is written in data flow direction: First negate, then contribute as divisor.

KinExpr: Additional textual Input as simple inner expression, factor or weighting value

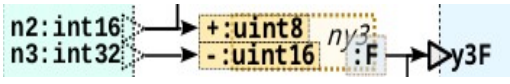
Figure 110: OFB/DinExprFactor.png



This is a simple possibility to include a factor for an expression part to realize such expressions as `y = m*x + b;` It spares effort in graphic because it needs only one FBexpr GBlock, with `m` as factor for this example. The factor (or other extra weighting value) should be found as variable in the module, from another expression as usual `ofpZout...` pin. There are some more possibilities with the operator for the extra values. See 5.8.2.4 Operation on expression input: factors in Add expression, variables page 125

castExprPart: Value casting of ExprPart

Figure 111: OFB/DinExprPartCast.png



Right side of the operator after the colon `:` a data type can be given. Then the resulting ExprPart is cast to this type for the expression. See *5.8.3 Data Type specification and value casting in expressions* page *140*. Here it results in:

```
thiz->y3F = (float)((uint8)(n2) - (uint16)
(n3));
```

`castInput`: Value casting of the input

Figure 112: OFB/DinExprPartCastInPart.png

In opposite to `castExprPart` the `castInput` casts the used input value before an `KinExpr` is executed. In this example the `int32` value `xb2` is taken, cast to `uint16` (`W`), This means for machine code level: Store and process it only in a 16 bit register. Then it is shifted, and for the AND operation cast to uint8, means only handled in a 8 bit register as also xb3 is. The generated C code with the proper target compiler does it:

```
thiz->yb2 = ((uint8)(~((uint16)(xb2) >> 8) ) & xb3);
```

Note: It would be better write it left of the OpExprPart hence also ~ for the casted one.

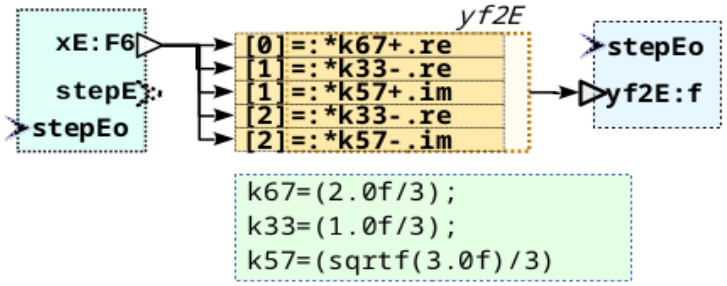
`InputAcc`: Access to an element of the input variable

Figure 113: OFB/DinExprPartAccInput.png

This is a similar example as for type casting, but here the input is offered as `int8 xa[10]` array. `B` is the designation for `int8`, and the `10` after describes an array size, see *5.4 Data types* page *58*. The first part accesses element 3 (index counted from 0), the next part the other element. `[4]` is cast to `uint16` to make the shift operation sensible, then both are or combined to a 16 bit value at least on output declared as `int16`:

```
thiz->ya2 = (int16)(((uint16)(xa[4]) << 8)  |  (uint8)(xa[5]));
```

Similar the access to elements of a data structure on input can written with `.name`, and both can be combined as possible. See *5.8.2.5 Access to elements of the input connection to use* page *136*



`setElemOutput`: Designation of an element to set in the output variable

Figure 114: OFB/DinExprPartAccInOut.png

Here both, the access to an array element of the input and the set to an element of the output is used. The output is a `float_complex` with `re` and `im` as parts. This parts are addressed by `.re` and `.im`. The expression is an ADD expression, but the output parts are separated. It is a Clarke transformation, 3~ to vector presentation of electrical grid current or voltage. The code generation shows with the given factor alias definition:

```
thiz->yf2E.im = ((xE[1] * (sqrtf(3.0f)/3))  - (xE[2] * (sqrtf(3.0f)/3)) );
thiz->yf2E.re = ((xE[0] * (2.0f/3))  - (xE[1] * (1.0f/3))  - (xE[2] * (1.0f/3)) );
```

`specialDesignation` is used only for special cases.

`PinOrder` is explained in 5.3.7 nrGpos, order of pins after grave page 49

`Input` is the possibility of a textual connection see *5.7.8 Textual given connections* page *124* or also constants see 5.3.4 Constant input to a pin page 45.

5.8.2.2 Connection possibilities of DinExpr

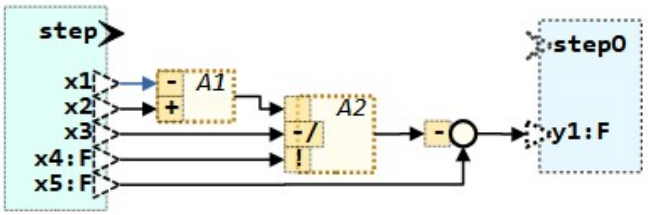


Figure 115: OFB/ExprExmpCombi.png

```
thiz->y1 = (-((-x1 + x2) / -x3 * x4) + x5);
```

* The DinExpr can have a **connection** of style `ofcDataFlow` or also of a common connection style `ofcConn` or `Default Drawing Style` **as destination from any data output as source**.

  * The data output can be also an FBexpr output, which creates an inner expression in parenthesis in the generated target code. In the image above this is A1 as inner expression from A2which contributes `(-x1 + x2)`:

  * The data output can be a module input or a variable in the module, which

value is then used, respectively the name of the variable is used in the expression term. In the image above they are all values `x..`

* The data output can also be an operation output (written with `name()`) from any FBlock. Then this operation contributes for the expression.

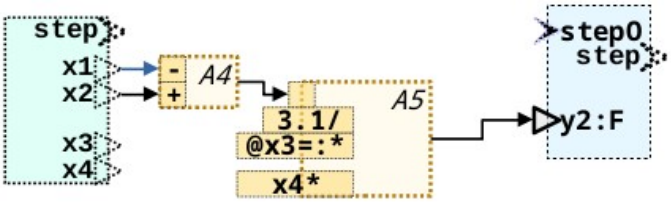


Figure 116: OFB/Exmpl_ExprConnConst_OFB.png

* The DinExpr can have a **constant value as input**, with several writing styles, see *TODO*. This is shown on the right image for the input `3.1/`

* The DinExpr can have a textual connection instead wired connection. This is shown here on `@X3=:`. Note:

The x4 is not a textual connection, it is an immediately constant and this is very confusing. TODO change it. It is fixed meanwhile!!! TODO work on document

* The DinExpr can contain the **operator for this input** as often last right side or only one text part. But some special entries can be **right side after the operator:**

  `[` as first character of a array index for the output variable, see TODO

  `.` as designation of set of a structure member of the output variable, see TODO

  `?` following by really special entries see TODO

  `` ` `` following by a number as order of inputs, see TODO

  `:=` as assignment operator with a constant or variable input right side instead left side, see 8.8.3.2 Constant inputs on DinExpr page 122

  * The operator can be omitted, then a default operator is valid, see next chapter.

* The DinExpr can contain a second input with an operator before for a **simple inner expression of this ExprPart** written left side of the operator. The second input can be a constant value or a textual given variable name or a textual given connection. See TODO

5.8.2.3 Operator combinations of DinExpr determines the type of the expression

- Operators are `+ - * / % >> << & | v ^`. Its result is the same data type as the inputs. See *5.8.2.3 Operator combinations of DinExpr determines the type of the expression* page *133*
- Unary operators are `- ~`.
- Comparison operators are `< > <= >= == <> !=`, its result is boolean. See 5.5.6 Compare Expressions page 137

`!` The exclamation mark is only a place holder for the operator with the same meaning as if no operator was written. It is necessary if the operator position is necessary for syntax or designation, see possibilities below.

, but also a factor as constant or as variable and also a type casting, see *5.8.2.6 Description of all possibility, syntax/semantic of DinExpr* following. The simple form to add and sub is shown in the image above.

In opposite to the circle with lines, here is enough place and clarity to write a text associated to the expression input. This can be one of the operations known from mathematics and logic in the following groups:

Unary operators: They should be written before the binary operator to this pin. The binary operator needs to be written if an unary operator is given.

- `- /` are numeric unary operators. The `/` means: build the reciprocal before operate. It is proper translated in destination source code `(1.0 / (input))`.

- `~` is the bit negate for bit operators (data types `...WORD`, `BYTE` in IEC61499, type chars `q u w v`). It depends on the code generation whether it is applied also to numeric types. Note, that numeric and bit types are not distinghuished in C/++ or some other languages, but in IEC61499 and also IEC61131 for automation control.

- `~` is also the boolean negate (do not use `!` or other).

Binary operators: This are the operators for the input in relation to the input before respectively the result of the inputs before. The first pin has not a binary operator, hence the operator given is a unary operator with the same meaning. It is important that one FBexpr can handle only binary operators of one group. But especially usable for an ADD expression the inputs can be modified by usual a factor before operation written textual in this `ofbExprPart` pin, see following *5.7.2 More possibilities of DinExpr*.

- `+ -` numeric ADD FBexpr. Unary operator `- /` possible written before.

- `* / %` numeric MULT (DIV, Modulo) FBexpr with unary operator – possible. The `%` is the modulo operator. If the first pin has a `/` then the reciprocal is build from the input, or it is the binary operator with “1.0” as first operand. Both means the same. Unary operator `- /` possible written before.

- `&` boolean or bit wise AND, with unary operator `~` possible before for bit wise negate. At least one input (recommended the first) should have the `&`, the others are `&` inputs also without designation.

- `| v` boolean or bit wise OR, with unary operator `~` possible for negate. The `v` may be better readable as `|`, hence recommended.

- `^` boolean or bit wise XOR, with unary operator `~` possible for negate. Note that also `==` and `<>` can be used for a boolean exclusively OR and NOR.

- `<< >>` Bit shift operators. It can applied for numeric or bit values. `-` as unary operator before is admissible. Negative values means shift in the opposite direction. This is important if a non constant value is on input.

- `== != <> < <= > >=` For numeric, boolean or bit wise comparison, with unary operator `~` or `-` possible for bit wise negate or numeric negate. More as two inputs can be used, then the relation The result is always a boolean value. Hence only two inputs are admissible for the comparison. The compare operator can be written on any of the both inputs. For greater and lesser the first input is at left side of this operator.

`<>` is defined for ‘not equal’ in IEC61499 and also Structure Text, which is translated to `!=` in C/++. If more as one input is used with `==`, all

should be equal. Also `<>` means, all are not equal together. Elsewhere the relations are valid in comparison to the input before, or in comparison to the first input. The first input should have either the `==` operator or given without operator.

Mixing faulty operators cause an error while evaluation the graphic.

Look on the following examples:

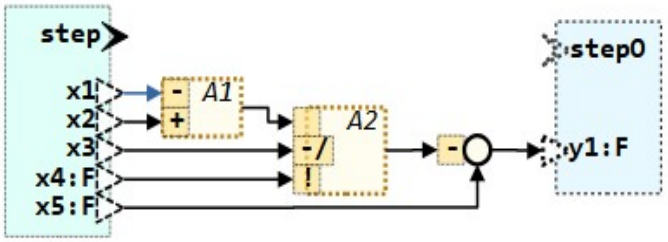


Figure 117: OFB/ExprExmpCombi.png

It shows a combinatorics, the expression is

```
y4 = -((-x1 + x2) / (-x3) * x4 ) + x5;
```

The last expression block has the `-` as *DinExpr* immediately near the circle which is an `ofbExpression`. This is an alternative instead write the `-` on the line. But of course in the translated source expression line the `-` appears before the representing `(…)` of the expression before.

In the middle FBexpr the `*` on the 3th input is omitted because it is default, the expression is detected as multiply expression. Also the `*` on the first input can be omitted because the `/` is enough concise to determine this FBexpr as Multiply expression with this operand to divide. The `-` after `/-` is the unary `-` for the `x3` input. All of this should be intuitive understandable.

But to reinforce it look on a boolean example:

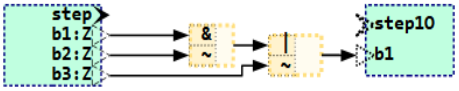


Figure 118: any image

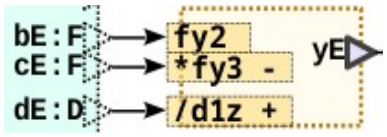
This is in C/++ Syntax:

```
yb1 = (b1 & !b2) | !b3;
```

Because the data types are boolean in C/++ the `!` should be used for negation (NOT). If the data types would be `u w v` then the ~ will be proper. The code Input generation designates it automatically.

5.8.2.4 Operation on expression input: factors in Add expression, variables

Figure 119:



OFB/DinExprFactor.png

fy2 This is a simple possibility to include a factor for an expression part to realize such expressions as **y = m*x + b;** It spares effort in graphic because it needs only one FBexpr GBlock, with **m** as factor for this example. The factor (or other extra weighting value) should be found as variable in the module, from another expression as usual **ofpZout...** pin. There are some more possibilities with the operator for the extra values. Elsewhere an ERROR is reported. This is an extra input for the expression. For implementation (IEC61499 coding) each DinExpr designated with **X1..** can have an correspond input **K1..** which is internally a **>>DinExprK_FBcl** for this weighting value.

The factor can be used alone, then the default operator is used for the expression part, here **+**, and the extra value is used with the opposite basic operation, here multiplication *****. For a MULT/DIV operation the opposite basic operation is +, for AND it is **OR** etc. See chapter *5.8.2.3 Operator combinations of DinExpr determines the type of the expression* page *133*

***fy3 -** Because the expression is an ADD/SUB expression, the operator for the additional evaluative input is *, but here explicitly given. The operator for the ExprPart is also explicitly given, but here **-** to contribute for SUB.

/d1z + This is not a factor but a divisor for the input.

The graphic is from **BasicTest.odg**, module **TestCombinatorics** (page 3). Resulting is:

```
thiz->yE = ((bE * thiz->fy2_z)  - (cE * thiz->fy3_z)  + (dE / thiz->d1z_z) );
```

But there are more possibilities using **ofpExprPart**:

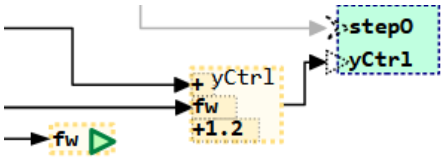


Figure 120: odg/ExpressionExmpK2const.png

This figure shows an add expression, but the second input is also multiplied with the variable **fw** and the 3th input is a constant with the given value be added.

The variable **fw** should be able to find in the state variables of the module. It is wired to the **K2** input in the FBcl textual presentation. The constant value of the 3th Input is a constant on the **X3** input.

The operation for the three inputs are written right side, or they are omitted as default for the operation type. The operation type is ADD (not MULT, not AND …) because the first operation is a **+**. Then all others are also **+** if not given.

There is also a possibility to write two variables in the expression input, but only if the input is not connected:

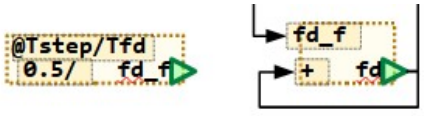


Figure 121: odg/ExprExmp2Vars.png

Left side it is a FBlock which should only built a proper adding factor **fd_f** for the right side integrator. This factor depends from the step time given in the module with **Tstep** with the **init** event, not shown here. The connection is omitted, because **Tstep** is well known in the context. It is drawn as a module variable in another page.

The factor is **Tstep/Tfd**. **Tfd** is a parameter loading on **init** or also able to change with the **param** event, not shown here because also recognize as such. The interesting detail is, how to build this variable for the integrator growth. The variable **fd_f** is an internal factor, but stored as state variable (**VarZ_UFB**) in the module. This factor is additional divide by a number, here **0.5** which means multiply by 2. But the value is an important manually found additional parameter with the technical meaning (here it is a magnitude relation) known by the developer (hence not an outside tunable parameter.

Right side a numeric integrator or += operation in C thinking is shown. The input `X1` is added and before multiplied with the factor `fd_f`. This may be done in a fast cycle, means should need only less calculation time. The factor is the left calculated variable, it is a time factor calculated as shown with the left FBexpr as described. The factor `fd_f` is calculated in another, a slower cycle because the `Tfd` value does not change so fast (possibility) and the division needs more calculation time (necessity to calculate not in the fast cycle).

The connection between the output `fd_f` and the input for multiplying in the right FBexpr can be drawn here with connections. But, the calculation of the factor may be placed on another page, the factor may be used more as one time, it may be more obvious if both are separated.

This is the here shown example, typical for controlling algorithm.

The variables are used in textual form. They should be known and locate on other pages on the graphic. A wiring is not necessary, it is more confusing than helpful. Where to find this variables? Of course either as input values of the module or as output of a parameterize FBlock. You can use ctr-F in the LibreOffice graphic tool.

5.8.2.5 Access to elements of the input connection to use

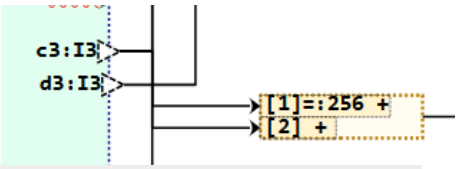


Figure 122: ExprInpArrayAccessMult256.png

The image above shows an expression which has its input both from the array `c3`. It gets each the both indices. But it multiplies the array element `[1]` with 256.

This may be a specific built 16 bit value with big endian, but read each byte from an int32-array. Only as example. Both are added then.

Adequate can be done for access to elements of a structured data type. Then the input starts with a dot and `.elem` with the name of the accessed element in the input structured data type. For example `.re` and `.im` can be used to a complex value's components.

5.8.2.6 Description of all possibility, syntax/semantic of DinExpr

See also chapter >>>*Impl-OFB_VishiaDiagrams.pdf*: >>>*7.3.7 Preparation of Expressions from odg33*

The syntax of the text on `ofpExprPart` follows the description on 5.3 Texts in graphic blocks and pins page 36 valid general for Din pins. Here examples are shown. Any part is optional.

Textual given connections:

- `fbSrc@pinSrc`: This is a textual connection, see also 5.7.8 Textual given connections page 76. If the String contains a `@` then it should not have a connection. Instead the connection is given textual. After the `@` the pin name is written, before the `@` the FBlock name. A module pin is named with `@pin`. A module variable is named as `@varName`. Access to a Xref label is named `@label`. If the pin has also a connection, it's the same as twice connections to the same pin, which is admissible in special cases, see 5.7.11 More outputs to one input page 78. A connection can be also an outgoing connection to another input

- `fbSrc{a,b}@pinSrc`: The index is regarded to a sliced FBlock, See *5.6.12 Sliced or Array FBlocks, Demux and array data* page *116* or also 5.10 FBlocks in slices, access to slices page 106.

Access to elements of the data source:

- `.element`: This is an access to an element of the connected source. If the pin has a connection, and hence the text of the pin does not start with `@`, this is the access to an element of the driving source. As well as it is possible to write `@varName.element` to access a variable (or FBlock output, or module pin) which is a structured variable, and then to the structure element. It is for example to access to `.re` and `.im` for a complex value

- `[0,2]`: This is an access to an element of an array of the connected source. It can be combined with element in all possibilities, but of course depending of the used data types. For example `.myArray[3]` accesses the element `myArray` in the given structured data type, and there the given element in the array. Otherwise `[3].myDetail` accesses in the third element in the given array type of a structured type, and there the element `myDetail` in the structure in the array element. It can be also combined with the connection given for example in the form `@fb@pin[3].detail`. or `@fb@pin.arrayElement[3]`.

Value cast of input:

- `:valueCast` this is a value cast, It is written as last operation of the access description before the `=:`, for example `@fbSrc[1]@pinSrc.element:int16=:...`. The given data type have to be one one of the standard types see chapter *5.4 Data types* page *58*. For example `:w` to cast to a 16 bit value `WORD` in IEC61499. Also `:uint16` is able to write, where this is the `:W` (upper case) which is `UINT` as numeric (not bit) value in IEC61499. In generated C language there is no difference for that. But the data type check in the graphic regards it.

The value cast type determines the type of the expression. All value cast types should be the same, differences are not admissible. The difference using the value cast to define the type of the expression with the output variable is: With value cast a cast is definitely done with the input value before it is modified by the K input.

If a `:valueCast` is used, the input type on the connection is free, determined by the input, not tested.

input =: operation

- `=:` This designation with the meaning *"It's an data input pin"* is necessary as termination of the input access description as shown before. After them as following described, the modification values comes, or the operator for the expression pin, may be able to omit. For a Din of a FBlock the name of the Din follows. For example `[1]=:` describes only the input access. The operator for the expression is not given, able to omit if the expression operation is described by operators on other pins.

Modification operation for input

The next elements are specific only for expressions:

- `*-factor` or also `+/bias`, `& ~mask`, `<<shift`, : This is a modification of the input value with a textual given operation and a possible unary operation of the modification value.

- If the modification value itself is an identifier, then it is searched as variable in the module. If found, the access to this variable is generated. It is possible that it is an instance variable for example with access using `this->` in C++, `thiz->` in C language.

- If the modification value is not found as variable, or it is a number string, then it is used for code generation as given. For example you can use identifiers, which are given in the generated code environment only (as Macro in C, as static variable in Java etc.). For example write `<<BITPOSXY` if `BITPOSXY` is defined in your generated code environment as Macro.

- The operator for the modification can be `+ - * / & | v ^ << >>`. The `v ` should be written with a space after, it is a OR operation as well as `|` but may be better readable. `^` is XOR. The space after the operator is optional.

- The operator for the modification value can be omitted if the DinExpr string starts immediately with the value or a given input access is finished with the `=:`: `@fb@pin[3].detail=:`. The omitted operator is a

  - `*` (multiplication) for ADD expression
  - `+` (addition) for MULT expression
  - `&` (Bit AND) for OR expression
  - `v` (Bit OR) for AND expression

- After the operation for the modification an unary operator for the modification value is admissible. This is `- / ~` for numeric negate, reciprocal and bit wise negate.

- There are two modification values possible necessary for example for bit shifting and masking `&MASK<<BITPOS` or also `+bias*factor` if necessary, for example `+1*adjust` if the adjust value is in range arround zero, but it should be multiplied with `1.0` if `adjust == 0`. This is sometimes necessary and here possible.

The modification values and operators are either a constant on the appropriate `K…` input to the `X…` input pin of the `Expr_UFB` in the fbd or

FBcl presentation (IEC61499), or it is written as String expression in the `expr` input of the FBlock presentation if a module variable is used. Then the module variable is connected to the `K…` input and presented as `$` in the expr String. That is sufficient for the adequate code generation with this `Expr_OFB` FBlock or just also able to interpret. But this means, only one value for the modification can be a module's variable, the other should be either a constant or an identifier not found in the graphic, instead found in the target language (MACRO constant definition or such).

Operator for the expression input

- On end of the expression the operator for the pin is written. The combination of the pin's operators are explained in the chapter before.

- Before the pin's operator also a unary operation for the value can be written.

A complete example for a `ofpExprPart` String is:

```
@fb@pin[3]:W =: <<BITPOS & BITMASK v
```

This example gets an array element form the named pin, may be a byte type, cast it to `WORD`, used for a bit wise OR with the v operator, but before mask and shift the incoming value.

Formally syntax:

A constant or a variable in the DinExpr plays often the role of a multiplier, but can also be used to divide, to add and subtract or to mask for bit operations. That's why the syntax of the DinExpr should be exactly presented:

TODO this syntax is yet not actually

```
DinExpr::=[\.<$?componentAccess>
| \[ [<$?arrayIndexVar>|<#?arrayIndex>] \]
|[<$?variableX>|=<#?number>|='<*'?string>'|]
 [<opK> [<unaryOpK>]]
 [<$?variableK>|<#?numberK>]
 [[<unaryOpX>]<opX> ]
].
```

The syntax is given using ZBNF-Syntax: The meta morphemes are written in `<morpheme>` or `<..?semantic>` whereby `$` as morpheme means: any identifier, `#` is any number, `*'`means any String till the end character `'`. The semantic helps to explain. Plain text is written immediately without quotations. Special symbols `<>[]{}.` are used for syntax expressions. If they are necessary in the plain text, a `\` is written before. `[…]` is an option. `[…|…]` is an alternative. `[…|…|]` is an alternative option.

- The `DinExpr` can be empty.

- If the text in a `ofpExprPart` shape starts with a dot as `.name`, then it is the name of a component of the variable on output of this expression. See *5.8.8 Set elements to a array of structure variable*

- Similar as dot, if the text starts with a `[` then it is an array store input. The text designates the index either numeric `[0]` or via a variable `[ixVar]` or also via the second input if only `[]` is given.

For the next three possibilities the following is valid:

If the pin has an input connected, the constant is the multiplier and assigned to the `K..` input. Then continue on `variableK`. If the pin has no connection, the constant or also a variable is wired to the `X..` input as `variableX`. or `number` or `string`. It means one FBexpr supports also multiply its inputs with numeric state variables, which is often proper usable. Also for comparison constant values are proper usable.

- `variableX`: An identifier on first position can be the replacement of the non connected input. But if the input is connected it is the `variableK` after the omitted `opK`.

- `number`: The same is with a given number. If the input is not connected, it is a constant on the X-input. If the input is connected, then it is the `numberK`. The number can be given hexadecimal. A numeric given number is converted in the proper form due the type for code generation. For example writing `13.0f` instead `13.0` for a float operation.

- `string`: A String in apostrophes is notated as String as given in the IEC61499 representation. For code generation, it is used as is. That makes it possible to write for example `'M_PI'` to address a `#define-Makro` given number. Without apostrophes it would search a variable named M_PI, not found, produce a warning but let this identifier in the code. That is dirty. Also a complex expression can be written for code generation uses as is.

- `opK`: The second operand which is connected to the input K… can be operate with this operators with the input.

```
operatorK::=+|-|*|/|%|&|^|
```

The compare operators are not admissible, because for this comprehensive expression form they change the type to boolean.

- If the `opK` is omitted, the default is `*`. `factor+` or only `factor` means, the input is first multiplied with the factor, then added. Also in a MULT term `factor*` means, the input is multiplied with factor, then both are multiplied with the rest of the expression term. Whereas `+factor*` means, the factor is first added with the input, then both are a multiply input in a MULT term.

unaryOpK::=-|/|~.

- `unaryOpK`: Also the second operand can have an unary operator after the given operator.

- `variableK`: The second operand can be either a variable of the module given as identifier which is connected to the K… input in the IEC61499 presentation.

- `numberK`: The second operand can be a number which may be converted by code generation to a necessary form. Also `0x1234`, a hexa number is accepted, but not converted.

- `stringK`: If the second input is given in apostrophes, it is designated as character string literal on the K… input as constant used as is for code generation. If the expression is a string expression (concatenation) then the code generation writes this `"string"`.

- `unaryOpX::=-|/|~.` The unary operator is regarded to the whole input for the expression term after a possible K input. For using an unary operator the `<operatorX>` should be written after. For example a simple `/-` means, that the input is subtract in an ADD expression, but before subtract the reciprocal is built as unary operation with the whole input. `var/-` means the input is multiplied by var, then the reciprocal of both is built, and the result is subtract.

- `opX`: Operator for the input:

```
opX::=+|-|*|/|%|&|v|^|>|‹|>=|‹=|=|==|‹>.
```

The operator for this expression is written at least right side. The syntax presents all possible operators. But as shown in *5.8.2 Expression data input pins DinExpr ofpExprPart* only determined combinations are admissible. Note that a `\<` in ZBNF presents a single `<`.

The operation with X and the second input is always done with more precedence, it is in parenthesis for the generated code.

(see `FBexpr_FBcl#setOperatorToPins()`)

5.8.3 Data Type specification and value casting in expressions

In the texts of the expression inputs and outputs (`ofpExprPart`, `exprOut` and also in the pins on output `ofpDout`..., `ofpVout`... `ofpZout`…, the text on the pin can contain some data type designations, written as `:Type`:

- `:F=:...` or `:W<<8` ***Left side on input before*** a `=:` or also before a detected operator in an expression part as shown in *Error: Reference source not found*: This is a ***value cast*** of the input data before the operation and also before an operation on expression input.

In C/++ target code this is `…((float)inputTerm)…` or `…((int16)input)<<8…`. ...

- `+:D` `<<8:S`: ***Right side on input after the operator***: This is a value cast after the the operation on input. If you have not an operation on input, left and right side value casts have the same effects. In C++ target code this is `…(double)(inputTerm)…` or `…(int16)((input)<<8)…` or just in combination for `:W<<8:S` as `…(int16)( ((uint16)(input)) <<8)…`. Note: Inside the data in the OFB converter this is stored in `fblock.DinExpr_FBcl#dTypeIxPart` and also `...#bCastToDTypePart` is set.

- ***Given DType on the expression output***, as shown in the figure right side for `ya2`, or also as shown in the Figure 73: odg/ExprArray.png below after the variable name on output. Then the expression term result is cast to this type, respectively the variable have this type.

- ***Resulting DType off all parts***

The Resulting DType of all parts is the numeric highest of all inputs, as also given in usual target languages maybe implicit in expression terms. It is not identically with the output type. Look for the example:

```
int16 n2; int32 n3; float y3F;
y3F = (float)(n2 + n3);
```

Here the operation itself produces an `int32` result after addition, whereas addition of `int16` to `int32` can be optimized by to compiler by generating an ADD16 in assembler and only increment the high part of the result on carry. The resulting DType of all parts in C language is `int32`. It is automatically converted to `float` or better should definitely cast afterwards to store in `y3F`.

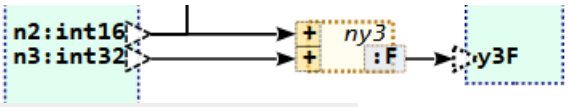


Figure 123: odg/ExmplAddFSI.png

Note: Inside the data in the OFB converter this is stored in `fblock.FBexpr_FBcl#dTypeAllDin`. If you look in the report file for this example, it is `src/+BasicTest/genSrc/report/TestCombinatorics.dTypeUsg.txt`, then you find:

```
==== FB: ny3:Expr_OFB
  =:ny3.dTypeAllDin::I
  =:ny3.expr::-c
  =:ny3.X1::-S
  =:ny3.X2::-I
  %=ny3.y::0`F  @0 mIO=3F mDEv=1
```

The data type designation follows chapter *5.4 Data types*. The data types from the inputs `n2` and `n3` are propagated to the `X1` and `X2` inputs of the expression as `S` and `I`, which is `int16` and `int32`. The detected common inner type of the expression is `I` `int32`, reported as `dTypeAllDin`. The output is `F` `float`. The generated code for C language is exactly as shown in the code example above, second line.

Look to another example:

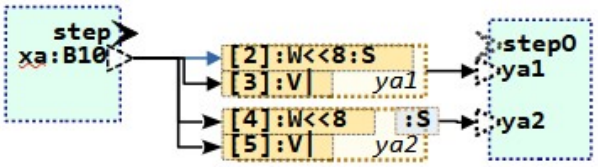


Figure 124: odg/ExmplShiftOr16.png

The mission of this code is, take bytes from an array (`B10` is `int8[10]`) for example from a received raw datagram, and combine it to 16 bit values. Here for the first output ya1 the cast to `S` `int16` is done only for the first input, after cast to `W` `uint16` for the shift operation. Hence the OR operation combines `int16` and `uint8`, which is intrinsically faulty but without problems in C language. Because the `int16` inner DType of the expression, the output ya1 is `int16`.

```
thiz->ya1 = ((int16)((((uint16)xa[2]) << 8) )
           | ((uint8)xa[3]));
```

The cast first to `W` `uint16` (`S` `int16` may be also possible, it's an example) is necessary, because elsewhere `<<8` on a `int8` would be done, which isn't sensible.

The second expression `ya2` is more clean. Preferred, use this pattern. The first operand is cast to `W` `uint16`, then OR with `V` `uint8`, which may be optimized by the compiler only by

loading the 8-bit Lo and Hi part of a 16 bit register. The inner data type (imaginary type of the register) is `W uint16`. That is clean. After them, the result is cast to `s int16`, which is the interpretation of the combined bytes. This is a clean operation.

```
thiz->ya2 = (int16)( (((uint16)xa[4]) << 8)
                   | ((uint8)xa[5])
                 );
```

In assembly language registers have not a dedicated data type, only a bit length. The machine code operations decides which is done, whereby signed or unsigned addition is primary only a quest of evaluation of overflow and carry flags, but secondary a quest of saturation, see next chapter. C language has inherited this way of thinking. The decision between `int` and `unsigned int` is only a hint to the assembler, and was not thought in the 1970th to influence a clean programming. For the graphic level it should be clean. It means the first construct for `ya1` in the image may be forbidden or at least should produce a warning, because signed and unsigned are combined with OR. In opposite, the cast on input and output is a definitely and hence correct cast, the only one possibility to deal with near machine code given stuff.

Quest of strong data type usage:

Traditional Function Block Graphic tools, for example Simulink but also Automation Control tools requires a strong type determinism. The operation in C language can be presented in Simulink with:

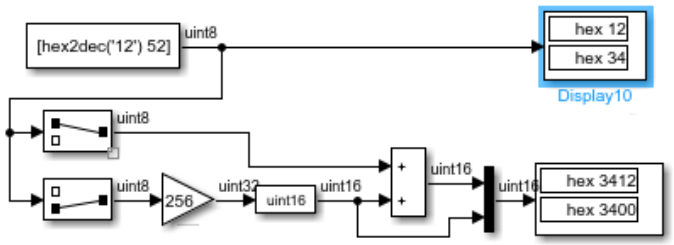


Figure 125: smlk/Exmp_shiftAdd8bit.png

The image above shows an example how to work in Simulink. The value casting to int16 is not necessary, if the gain is parameterized with output `unit16`. With *“inherit via internal rule”* it means to use `int32`, why ever. For shifting in Simulink, here a multiplier with 256 is used. The existing shift Blocks are too sophisticated, because they generate a specific data type “`ufix16_E1`” which is not usable nor can be converted to `uint16`. Ask Mathworks how to deal with it.

To have full control about the data types in OFB, there are two possibilities to determine the data types on expression input:

a) cast the input value

b) cast the expression part output

Both are separated because between both the modification operation is additional possible.

But: The addition operation can be optimized by the compiler, for example for a 16 bit Processor with only a 16 bit addition, and regard and overflow to increment only the high part of the copied x2 in the extra 16 bit high part register. Such optimizing possibilities are hindered, if a conversion of all inputs to the same data type, here `int32`, is necessary, done outside of the intrinsic expression FBlock, as given in some traditional FBlock programming tools. Then the compiler does not may know, that x1 has really only 16 bit:

```
int16 x1; int32 x2; float y;
int32 x1a = (int32)x1;
  // later in code
int32 ya = x1a + x2;
y = float(ya);
```

This may not able to optimize afterwards.

5.8.4 Data types with fractional bits in expressions , using saturation

Integer data types presents real values, for example after a ADC (analog to digital converter). If your controller has full floating point support, then often the ADC result is converted to float, scaled, and furthermore calculate with float. But at least on the output, before DAC (Digital to Analog Converter) or also before a PWM (Pulse Width Modulation) you need again an integer presentation. Some processors have not float support, only `int16` multiplication, or `int32`, or only multiplication by software.

The world of embedded control is variegated.

Also if a float arithmetic is present, it may be necessary to use `int32` for integral parts in control algorithm, because float has a resolution of only 24 bit. This may cause hanging problems, an integral part does no more integrate if the increment value is too less. Using an integer `int32` integral value helps, because anyway the value is a physical and hence limited in range value.

There are some reasons to work with integer arithmetic instead of float. This topic is also discussed in https://vishia.org/emc/html/Ctrl/Fixpoint_float.html.

5.8.4.1 Example - How is it done in pure C programming

In source text programming in C language the unit and the scaling of values, and a possible position of a decimal point in an integer value is only thought in brain, and maybe mentioned in the comment lines. The programmer should think about shifting of values. And also regard limits, use saturation arithmetic.

Follow an example. An ADC inputs a 14 bit value. This value is shifted to 15 bits, the LSB of a `int16` value is the sign, which should be left zero.

But now for tuning the accuracy the value is multiplied with a factor near 1.0, but with them scaled to a 100% value presented in the high byte. Now this value should be subtract from a reference value. The result is a signed value. The difference should be multiplied with a factor in range 0.01 till 100, as gain for a simple P controller. The C programmer writes the following code:

```
uint16 adcInput = readADCRegister & 0x3FFF; // uses bit 13..0
uint16 gainInput = (uint16)((1<<16) * (1.01f/1.28f)); // scale the ADC to nom 100 in hig byte
int16 xAdc1 = adcInput + offsAdc;           // maybe <0 if offsAdc is negative
if(xAdc1 < 0) { xAdc1 =0; }                 // saturate it, never < 0
uint16 xAdc = (((uint32)adcInput * gainInput) >> 14;  // hi byte is 0..200 nominial.
uint16 ref = getReference();
int16 wx = (int16)ref - (int16)xAdc;        // now needs signed int
uint16 kP = param->kP;  // 0x0400 presents 1.0, max 63.999, resolution 0.001
int32 ymult = (int32) wx * kP;
if(ymult > 0x01ffffff) { ymult = 0x01ffffff; }   // saturation check. Regard >>6
else if(ymult < 0xfe000000) { ymult = 0xfe000000; }
int16 yCtrl = (int16)(ymult >>6);                // limited output wx * kP as 16 bit value
```

For the offset adjustment, a negative value may occur though all is unsigned here. But then the xAdc1 is limited or saturated to 0. Imagine a value which's 0 and end point is out of interesting, the ADC value is used and adjusted on maybe on 10% and 90% of its range.

For multiplication with `gainInput` temporary `uint32` is used, because both factors have 16 bit. But the result is shifted back to `uint16`. The output presents now a nominal range between 0 and 200.

Now two positive values are subtract, and the result, the difference `wx` is signed. In the algorithm a possible overflow is forgotten. 170 - 30 should be 140, but 140 in the high byte is 0x92, and this is negative. Here a comparison is necessary, or using processor specific saturation arithmetic, or simple prevent using the bit15. But this is a loss of accuracy of the scaled ADC signal, important to build small differences for a possible D-Part or the controller.

The `kP` factor, the gain for the controller is oriented to a range from 0.001 till 63.999 and uses 6 bit before decimal point. The given float number is proper converted on compile time, no floating point arithmetic on a cheap

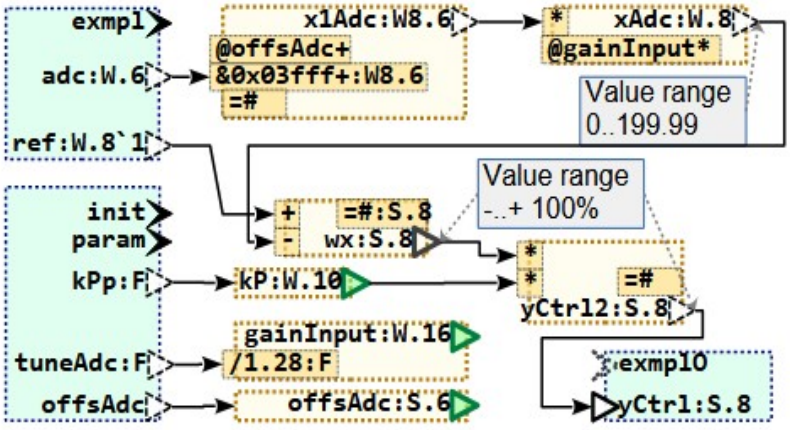
controller. On multiplying here, the value range of the difference `wx` is multiplied with the `kP` gain. It means on gain=10.0 means 10% difference between reference and measurement value thought in 100% = high-Byte = 100 forces and output of 100%, value 100 in `yCtrl`., with possible overdrive to 127%.

For the overdrive here a check of the 32 bit multiplication result is done, and limit or saturate the result.

But do you overview this bit shift and test stuff? Did a consumer with physical and control knowledge overview this peace of code, beside other similar stuff? Does the programmer overview all? Do you attempt to use a more costly floating point processor because the integer arithmetic is too confuse for programmers?

5.8.4.2 Same Example graphical

Figure 126: odg/FractionalBitsExmpl1.png



The data types are marked with the number of fractional bits, optional also a number of pre-fractional (integer) bits, if there are lesser then the value size. The `adc` value comes in with 16 bit unsigned (`W uint16`), it contains the ADC value itself and can contain other bits, for example binary inputs in bit 15..14. Hence it is masked with `0x3ffff`, but then declared as `W8.6`.

It means 14 bits are used, 8 bit pre-fractional, 6 bit as fractional, hence a value range from 0...255.98. To this value an offset is added, usual in a small range, but maybe negative. The offset is scaled as `S8.6`, with the same number of fractional bits. It means it is a simple addition, builds a signed result. But because the output is scaled as `W8.6`, unsigned, a saturation to 0 for negative values is done. For that the input pin with `=#` is given. It is a limitation or saturation designation.

For the `gainInput` the same is done as in the C routine, because of the data type designation. The `ganInput` is `W.16`, means unsigned, and it is also scaled from a float value, input `tuneAdc` with the nominal value 1.0 but divided by `1.28`. If the gain is never >1.27, it is proper for this value range, nominal `0xc800` for `tuneAdc = 1.0`. `200 == 0xc8`. This is explainable with the graphic if the listeners have a conception of value ranges in their hexa presentation. To understand possible errors for overflow of the solution. The output is presented as W.8 or same as W8.8, hence in a range from 0.0..255.996, but because of the `gainInput` scaled to a used range till 200.0 nominal. This range may be a part of concept, for example a position measurement between 0 and 200 mm.

To build the control difference `wx` the output is signed, but also with 8 fractional bits (no accuracy lost), but yet only with 7 pre-fractional bits, and with saturation symbol `=#`. The internal arithmetic result is also `S.8`, marked on the saturation input pin. Here an understanding of the implementation of the graphic is necessary. As described in *5.8.3 Data Type specification and value casting in expressions* the chapter , the automatic built internal value is W.8 with saturation to 0 and 0xffff because both inputs are W.8 and an unsigned operation results. But this is not desired here, the result should be presented as signed. The signed designation on the output is not sufficient, because it is only the output designation, cast after the operation. At least one input should be designated as `S.8`, done on the saturation or limitation input. Now internally an operation

```
{ int16 _expr_;   // otx: exprSum
  SUBSWW_SAT_emC(_expr_, +ref, xAdc, 0);
  thiz->wx = _expr_;
}
```

is produced by the given target code generator. The `SUBSWW_SAT_emC(...)` is a macro defined in

`emC/Base/Math_emC.h`, which can be optimized defined for a target controller which has saturation instructions, using `__asm` syntax, For example for ARMv3M instruction set this is:

```
#ifndef SUBSWW_SAT_emC
/**Subtraction of unsigned to signed builds...
static inline int16 subSWW_SAT_emC(uint16 a...
  int32 R; int32 A = a; int32 B = b;
  __asm("SUB %[R], %[A], %[B]\n" : [R] "=&r...
  __asm("SSAT %[R], 16, %[A]\n" : [R] "=&r"...
  return (int16)R;
}
#define SUBSWW_SAT_emC(R, A, B, SH) { R = s...
#endif //SUBSWW_SAT_emC
```

It is truncated here, see the original header in `src/src_emC/cpp/emC_inclComplSpec/cc_ARM/ARMv3M_Math.h`. The subtraction itself is done by the 32 bit SUB, but afterwards it is saturated to 16 bit, the result is interpreted as signed int16.

The default implementation in pure C, which may be optimized by a target compiler, looks like:

```
#ifndef SUBSWW_SAT_emC
/**Subtraction of unsigned to signed builds...
#define SUBSWW_SAT_emC(R, A, B, SH) { \
  uint16 _A_ = (A); uint16 _B_ = (B); \
  int16 _R_ = _A_ - _B_; \
  if(_A_ > _B_ && _R_ < 0) { R = 0x7fff; SE...
  else if(_A_ < _B_ && _R_ > 0) { R = (int1...
  else { R = _R_; } \
}
#endif
```

Other than in the ARM implementation an int16 subtraction is done here. For a 16 bit target processor it is more applicable. The result is checked in faulty sign and values. If it is overflowing, the result is not as expected, detected by comparison.

Adequate it is done for the multiplication of `wx` with the `kP` with different length on fractional bits. For a multiplication shifting the result is necessary and sufficient. Before shifting the possible overflow should be checked. The multiply to get `yctrl2` is performed by the following generated target code (shortened):

```
{ int16 _expr_;   // otx: exprMUL
  MULhiSSWshlSAT_emC(_expr_, thiz>wx, ...
  yCtrl2 = _expr_;
}
```

Also here is a macro for multiplication is used, which can be implemented for example for the ARM:

```
#define MULhiSSWshlSAT_emC(R, A, B, SH) { \
  int32 _A_ = A; int32 _B_ = B; int32 _R_; \
  __asm("MUL %[_regR_], %[_regA_], %[_regB...
  __asm("SSAT %[_regR_], 16, %[_regA_], AS...
  R = (int16)_R_; \
}
```

This is a pure macro which calls the specific multiplication and the SSAT saturation instruction with shift for ARM Cortex M3 machine code. The default implementation is also given:

```
#ifndef MULhiSSWshlSAT_emC
  //int16 mulhiSSWshlSAT_emC(int16 a, uint16
  static inline int16 mulhiSSWshlSAT_emC(i...
    int32 res = ((int32)a) * b;
    int32 m = ~((1LL<<(31-sh))-1);      // sh
    if(res <0 ? (res & m) == m : (res & m)...
      if(sh <16) return (int16)(res >> (16-sh
      else return (int16)(res << (sh-16));
    } else {
      SET_SAT_Math_emC()
      return res < 0 ? ((int16)0x8000) : 0...
    }
  }
  #define MULhiSSWshlSAT_emC(R, A, B, SH) R =
#endif  //MULhiSSWshlSAT_emC
```

This macro implementation calls an inline operation which has more opportunities in writing style. For saturation, a mask `m` is created, used to test the 32 bit multiplication result of the 16 bit factors, before shift. See also `.../Math_emC.h`.

5.8.4.3 Why saturation or limitation is neccessary

In float arithmetic you get an overflow in numeric range only for values >= 10^{38}. It's very much, maybe greater than the universe. In integer arithmetic overflow is usual a problem to handle. Remember that the Ariane 5 has crashed because of an overflow:

https://en.wikipedia.org/wiki/Ariane_flight_V88

There are some things, not only one, wrong. If you work on processors machine level, an overflow flag set if the operation result leaves the numeric range, or just a carry flag for unsigned arithmetic, as also known for manual school mathematics. Assembler programmer should know and regard it. But the programming language C, founded in 1970, does not regard such flags. Why? Speculation:

C language should be machine independent, but the flags are machine specific. The importance of such flags were not seen by the developer of the language C in 1970, were not in focus. They had thought (speculation), it is not important. Maybe, controller algorithm were not in focus running in C, only data processing things, which have not a frequently overflow problem. Since 1970, nobody has solved this gap, also not C++. (has Rust it solved?).

The next thinking error for that problems is: On a numeric error a hardware exceptions should be thrown. But this thinking is also very wrong. Because: On embedded control the show must go on. You cannot abort or delay the execution of machine code because of a long exception handling. Especially on startup, if not all values are already proper, also a division by zero may occur, because a parameter doesn't may be known and has a default of 0. This division should not have impacts (please do not throw an exception), because the result is not used anyway.

The result of division by zero is well defined in float arithmetic: It is a special coding of "NAN" – not a number, possible to check afterwards with a simple comparison.

In C language in `int16` resolution, addition of 32700 + 100 result in a value of -32736, it is negative. Because the value range ends on 32767. It you do not do nothing more, this negative value is the result. And this result is faulty and may be dangerous.

The ARM controller technology have made as first a proper solution in machine level: the saturation arithmetic. This ensures that 32700 + 100 result in 32767, which is the greatest value able to present in `int16`. See en.wikipedia.org/wiki/Saturation_arithmetic. This can be used also in a compatible way in C language, see https://vishia.org/emc/html/Ctrl/Fixpoint_float.html#truesaturation-arithmetic.

But using an overflow handling or not should be a decision of the programmer, which is in the compatible form also possible to use in C language. Why: Overflow handling needs a little bit more calculation time. If an overflow is excluded by the input value ranges and the calculation time is rarely, it should not be done.

And this decision should be transferred also to the graphic programming level.

5.8.4.4 Limit or saturation input(s)

Any FBexpr can have one or two inputs for limitation. The possible operators on this inputs are:

`=#` for a symmetric + - limitation, only one input

`=^` limitation to maximum

`=_` limitation to minimum

This inputs can have all capabilities as other inputs, constants, wiring, pin expression. It the input `=#` has no input value, no connection or constant, then it is only an overflow limitation. It means the saturation operations are used.

If the other operations on the pin are given, or a value is given on `=#`, then also saturation operations are used, but additional the result is limited to the given value.

If the FBexpr with this inputs have only one other input without operation, then of course no arithmetic is used, it is only to limit the input.

This is adequate the Simulink the Library FBlock *"Saturation"* or *"Saturation Dynamic"* in the Simulink standard Math FBlocks Library, *"Discontinuities"*.

If FBexpr do not have such inputs, especially not the simple `=#` input, then ordinary wrap around arithmetic is used as usual in most programming languages (especially C) without any overflow detection. That is in response to the user.

Note: For angle integration the wrap around is the base of an integer presentation of the angle in range -180...+179.999 degree. This is the usual only one sensible use case for the wrap around addition.

5.8.4.5 Condition on overflow

This is in the moment a TODO. Planned is: If an saturation occurs, a specific event output is possible. It triggers (see 5.6.8 Conditional execution with boolean FBexpr page 66 on overflow situation, parallel to the normal event output. It means the normal calculation with the saturated value continues, but additional a specific overflow handling is possible, for example to create a log message, control some state machines or etc.

5.8.5 Compare Expressions

TODO

5.8.6 Any expression in FBexpr

The `ofpExprOut` shape or also the text of the `ofbExpression` can contain both a function ***written with parenthesis***, for example `atan2()` or any expression written in the target language using `X1`, `X2` etc. for the inputs. The source code generation inserts this function or expression either as written or with an adequate derived code, see next. Some functions should be well known for graphical level. Specific maybe complicated functions can be written in the implementation level and called here immediately.

Look on a first basically example:

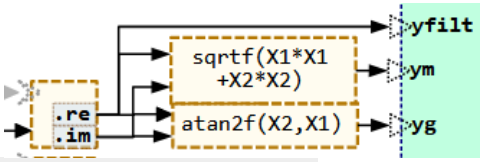


Figure 127: odg/ExprAnyX1X2.png

The `ofbExpression` shape or block in this example has not any `ofpExprPart` or `ofpOut` pins, it is not necessary. Input and outputs are immediately bonded to the expression block. The inputs are counted from top to down, and then right side from top to down, or also from left to right first top, and at last on bottom side, if necessary. The input pins has in this order the names `X1` .. `X99` so much as given.

While code generation, the identifier `X1` … etc. are replaced by the values which are connected to the inputs using the .code template scripts, see chapter *5.8.11 FBoper, operation for a FBlock*.

Because often target languages such as Java or C/++ are very similar in expression writing, the expression notation in the graphic is compatible with some languages. With an adaption table function names can be replaced for a specific destination language. For example the here shown `sqrtf()` is known for C/++ language, for float calculation. For Java source code it can be adapted with `(float)Math.sqrt()`. This is done as part of the translation template.

Also for this possibility input `ofpExprPart` can be used to influence the inputs also with factors, or using constants or negate the input values.

5.8.7 Output possibilities, variable after expression

All shown expression examples till now have its outputs on the expression box. In this kind the expression is not represented with a variable, it is an inline expression. The value is stored or used from the input pin after.

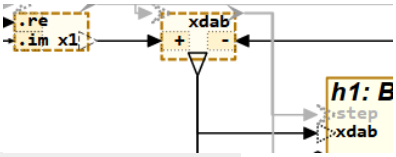


Figure 128: odg/ExprOutpin.png

This example shows two expressions with a pin symbol on output. A pin symbol or any other shape form of style **`ofpDout...`**, **`ofpVout...`**, **`ofpZout...`**, forces creation of a variable in the generated code. Especially on forking the data flow (using for more as one input) as here for **`xdab`** it is sensible. The left output has the style **`ofpDoutRight`** which is a normal data output. This forces a stack local (temporary) variable in the code. Here the variable is also necessary to collect the both parts of the complex value. If the expression is only used in one event chain, it is always ok.

The second expression **`xdab`** uses a style **`ofpVoutLeft`**, here the shape is rotated to 90°. This forces an instance variable in the **`struct`** or **`class`** of the module. One additional advantage is, it can be better visited in debugging on runtime. The variable can be used also in more as one event chains, which are more as one operations, but the data consistence is not guaranteed then, as usual in such situations.

The name of the output pin determine the name of the expression. If the output pin has not a name as for **`xdab`**, the name of the expression is the text in the **`ofbExpression`** shape box.

In the built data from the graphic or also in the FBcl representation (IEC61499) (see chapter *Error: Reference source not found*, page *Error: Reference source not found*) the expression itself is a FBlock of type **`Expr_UFB`**. The variable on the expression output builds an additional FBlock with type either **`VarL_UFB`**, **`VarV_UFB`** or **`VarZ_UFB`** for this tree possibilities.

The next figure shows the sensibility of a **`ofpZout...`** or **`VarZ_UFB`** variable:

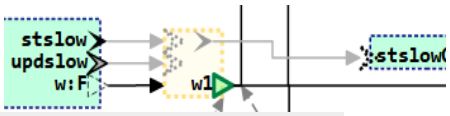


Figure 129: odg/ExprOutStateUpd.png

The output has the style **`ofpZoutRight`**. The letter **`z`** is derived from the https://en.wikipedia.org/wiki/Z-transform which is used for calculation, z is the stored (state) value. Hence it is set with the *update event*, here **`updSlow`**. The image shows the prepare and update events in gray, because there are automatically built. The input of the expression is here only one value **`w`**, the expression can have more inputs as shown in the chapter before *5.8.2 Expression data input pins DinExpr ofpExprPart*. The expression is calculated with the prepare event, here **`stslow`**, due to the data flow. But the output of this prepared value, setting of the variable is done with the associated update event, it means after (or before the next) preparation calculation. It means all Zout variable have the state of the last step for the next preparation. In Simulink those are 1/z Blocks, so named *"Unit Delay"*, or also so named *"Rate transition"* FBlocks, from view of another event chain (means another sample time, or another operation in implementation. If the update operations are atomic, non interruptable, then all Zout data are consistent.

5.8.8 Set elements to a array of structure variable

A variable after expression can be generally from a structured type. The simplest case is a complex or array data type.

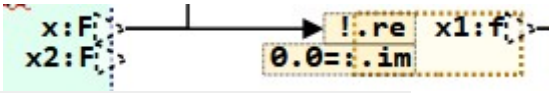


Figure 130: odg/Exmpl1SetElemvar.png

The image above shows a simple case. The variable is a `float_complex` in C language with the elements `re` and `im`. To set both (or also one of them) the name of the element is written in the `ofpExprPart` right side after the operation, see *5.3.2 The complete Syntax of text for pins and FBlocks* page *51* and 5.8.2.3 Description of all possibility, syntax/semantic of DinExpr page 85. Because the input has not often an operation, for the `!.re` the `!` is given as “set” operation symbol. If you omit the `!`, the simple `.re` would mean *“access to the element of input, left side”*.

For the second pin `=:` is given as separator between access or the constant value left side, and the expression part right side, hence it is sufficient also without operator. But also here `!.im:=0.0` is possible to write, with the right side written constant assignment.

Generally variables as expression output can be drawn more as time with or without an `ofbExpression` block (FBexpr).. If the expression has no input, then this variable can be accessed, not set. with more as one FBexpr, different elements can be set to the same variable, on different positions (also pages) in the graphic. The variable is only existing one time. The type need to be given only one time. If the type is given more as one time, it must be the same.

Here you see a lot of vector access. The principle over all is: Note that there are also access to vector element of the input variable from v1, the access to vector elements is written left side efore the `=:` or before the operator.

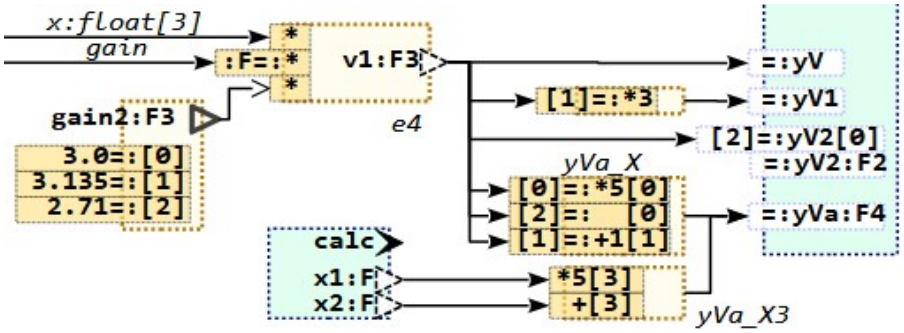


Figure 131: odg/ArraySlideDemux_VectorAccExpr.png

Whereas, set of elements, here vector elements should be written after the operator or `=:`. The output of the expression must be a variable after expression as in `gain2`, or also as shown in the image for `yVa_X` and `yVa_X3`, the variable may be follow on output.

With the `[…]` designation in the `ofpExprPart` the dedicated element on the output variable is set.

The variable after such an expression can be filled with more as one expression, one for each or one for more elements. Also an operation for each one element can be done as here shown for `yVa[0]`. Note that the element `yVa[2]` is here never set.

For `yV2` there is used another possibility: The variable in the `ofbMdlPins` is written as vector element. But then the variable itself need to be exists also, shown and defined below (possible also anywhere other). Additional the example shows the access to `v1[2]` to assign to `yV2[0]`.

```
thiz->yV2[0] = v1[2];
```

That are obvious possibilities to deal with vectors. Look on the generated code in the example `BasicTest.odg` Module `ArraySlideDemux`.

5.8.9 Output with ofpExprOut

TODO This shold be no more supported,

The graphic style `ofpExprOut` can be used to define an output for an inline expression, but with a called function. This results in the same as shown in *5.8.6 Any expression in FBexpr*, this text can be also notated as text in the `ofpExpression` shape. The difference is better handling in graphic.

In this case the name of the FBexpr FBlock in the IEC61499 presentation can be given as identifier in the expression FBlock.

The function designation can also contain a type for the output and also specific types for the inputs, writing after `:`, see next chapter

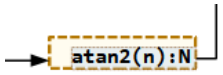


Figure 132: odg/ExprAtan2.png

The shows an atan2() operation which takes a complex value as input and outputs a scalar number.

To translate it, firstly the type letters for maybe non full specified values are replaced by the forward propagate types, for example results in `atan2(f)=F`. With this text the source code generation searches a proper translation, exact this String is used as identifier for a OutTextPreparer sub script which is then used for code generation. This sub script can be

```
<:otx: atan2(f)=F : fbx, cacc>
<:set:dinVar=genValueDin(fbx.din[1],'')><: >
atan2f(<&dinVar>.im, <&dinVar>.re)<.otx>
```

which results in generated code for example to `atan2f( cvar.im, cvar.re);` which calls the atan2() as given in C/++ destination language.

The designation of the output (here `N` as any numeric) is important, elsewhere the type propagation forwards the input type to the output. It does not know that the atan2() operation outputs a scalar.

5.8.10 FBexpr as data set

This is a snapshot from the BandpassFilter example which have on input but needs internally complex values.

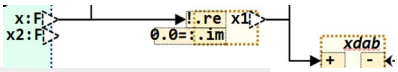


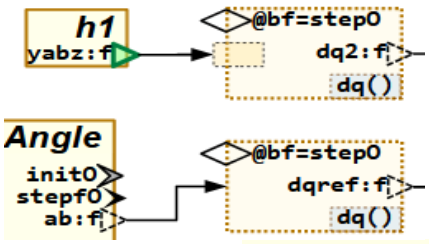
Figure 133: odg/ExprOutReIm.png

The expression inputs have a designation `.re` and `.im`.on end of the input. This means the so named data elements of the necessary output variable are set. This variable collects the real and imagine part and delivers a complex value.

The same as for `.re` and `.im` can be done for elements of an array. On right side in `ofpExprPart` it should be written in form `[2]` where as the `2` is the immediately constant index to the array. But also a variable index is possible, write `[X2]` where `X2` is the value on the second `K` input of the expression. (TODO in software ?) The size of the array variable on a collect expression should be dedicated, given with the type specifier, see next chapter.

5.8.11 FBoper, operation for a FBlock

The FBoper as shown in the following Figure can be seen also as part of the expression flow, hence it is here mentioned. But such an FBlock is intrinsically a concept of the FBlock and classes.



See chapter *5.6.9 GBlocks for operation access in line in an expression - FBoper* on page *110*

empty

5.8.12 How are expressions presented in IEC61499?

The IEC614499 does only know FBlocks and their types. Expressions are built from a lot of variants of standard FBlocks, as mentioned in the chapter *5.6.8 Expression GBlocks* page *110*. That is not the approach in OFB. For OFB one expression FBlock exists, which properties are described by textual qualifications.

But it is proper to map the OFB to the IEC61499 style by using a set of universal FBlocks for expressions and variable access as well as the following FBoper which are determined by String given parameterize of the operations.

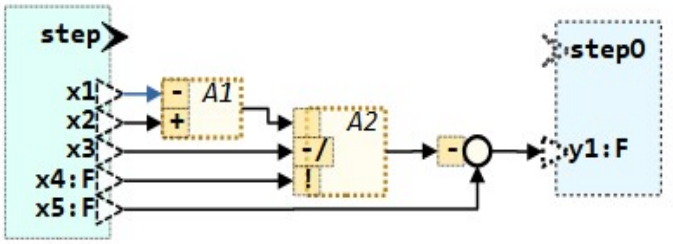


Figure 135: OFB/ExprExmpCombi.png

Have look to the fbd code for this file:

```
FBS
  A1 : Expr_OFB( expr:='~+,-,+;,,;;' );
  A2 : Expr_OFB( expr:='~*,*,/-,*;,,,;;' );
  .....
  d_4 : Expr_OFB( expr:='~+,-,+;,,;;' );
  .....
END_FBS
```

This is the definition of the FBexpr FBlocks. All three have the type `Expr_OFB`. The operation is defined by the string `expr`.

The FBtype `Expr_OFB` is defined as prototype only:

```
FUNCTION_BLOCK Expr_OFB
EVENT_INPUT
  prep WITH expr, X1999, K1999
END_EVENT
EVENT_OUTPUT
  prepO WITH y;
END_EVENT
VAR_INPUT
  expr : STRING;
  X1999 : ANY_ELEMENTARY;
  K1999 : ANY_NUMERIC;
END_VAR
VAR_OUTPUT
  y : ANY_NUMERIC;
END_VAR
END_FUNCTION_BLOCK
```

The input designation X*1999* means they are any number of inputs start with X1, and also any number start with K1. It depends on the connection. The `K...` can be connected to variables if necessary or holds a constant.

The `expr` is an input which controls the operation.

- It consists of three parts, ended with semicolon `;`. Each part contains information to the pins, separated with comma `,`.
- The first part describes the operation for the expression and for each data pin.
- The second part describes additional K-inputs (multiplied constants)
- The third pin may contain a specific operation defined in the expression, see chapter 5.8.3 Any expression in FBexpr page 50.

The first part starts with the common information to the expression:

- **The first char**acter in `expr='~+,.'` is the access kind of the expression:
  - `~` means an pure inline expression, as here in the example..
  - `=` designates an expression with a following variable. Hence it is generated to source code as statement to set the variable. The variable is a following extra FBlock, see todo
  - `&` designates an inline expression, but with some additional variable as output (call by reference). This occurs only if a specific function is given.

  This designation is determined based on the arrangement of the expression term in the data flow. It is used as input information for the code generation.
- **The second char**acter in `expr='~+,..'` is the operation type of the expression.

  `+ * & v ^ h =` are used for ADD, MULT, AND, OR, XOR, SHIFT and CMP

  `!` means, the expression is textual given, see 3th part of `expr`.

  `(` means, the expression is a operation call, whereby the operation is given, in the 3th part of `expr`.

- **The third char**acter in `expr='~+,..'` is always a comma `,` as separator between pins.

From this comma to the next comma each pin description is stored:

- If the first character of the pin description in `expr='~+,C…,C…;···` is a `C`, it means the code generation should insert a definite type casting to the type of the pin.

- After the first `C` or as first character the operator is written for the pin. It is one of

```
opX::=+|-|*|/|%|&|v|^|>|<|>=|<=|=|==|<>.
```

The operators with two characters are specifically tested.

- After the operator the unary operator is optional stored if given. It is one of `-` `/` `~`.

- After the operator optional an access to the source data is stored, as it is given as ***elemSrc*** in 5.3.4 Syntax of input to a pin page 12. This element starts with `.` Or `[` and goes either to the closing comma or the `@`, see following.

- If this ***elemSrc*** is an increment or decrement operation of the source variable, it is written with a starting `^` following by the operator `++` or `--`, or also possible another operations (textual given).

- As last, optional beginning with `@`, the ***elemDst*** is stored as access information to the destination of the expression, see 5.3 Texts in graphic blocks and pins page 10. This both information are also part of the data connection and are here twice, but only one time if constant values are on the input.

With this description all possibilities of the ordinary expressions can be mapped. For execution of the IEC61499 code in another environment as the here used OFB code generation the `expr` input should be proper interpreted or proper translated to a specific FBlock only existing in the generated code.

An example for usage that expression is shown next:

```
FBS
d_14 : Expr_FBUMLgl( expr:='~+,+,+;,,;;' );
  ...
DATA_CONNECTIONS
  ...
bf.yabz TO d_14.X1; (*dtype: f*)
```

This is a simple expression to add two values, which is adequate a F_ADD in the 4diac-tool for IEC61499.

For the other kind of expressions similar common FBtype are used, see the describing chapters and also the implementation hints in chapter >>>*Impl-OFB_VishiaDiagrams.pdf*: >>>*7.1.6 FBexpr_FBcl: FBlock for expressions, presentation in FBlock_FBcl12*.

5.8.13 FBexpr capabilities compared to other FBlock graphic tools

Compared for example with the known IEC61131 FBD diagrams for industiral automation programming the last one contains usual a lot of FBlocks for specific operations, for example ADD3, ADD3, SUB2, AND with two inputs which can be cascade etc. In comparison to the possibilities of OFB it needs some more FBlocks in the diagram, the diagrams will be more voluminous but not more clearly. It is a entanglement in details. Often a textual written expression is more proper understandable then a lot of wiring.

Expressions in the FBexpr blocks are related to the target language. This is an advantage for programming, it's clear what's happen. The expressions in a familiar target language are quite easy to understand from a customer level (with focus on mathematics). In opposite using a specific formula writing style of any specific tool needs also the understanding of this tool, sometimes it is more specialized as the familiar used programming languages.

Also a lot of specific numeric function blocks for sin, cos and whatever are lesser helpful as a simple written `sin()` in the graphic box.

Some graphic tools have also some parameters for expression blocks, which are hidden (not shown) in the graphic. They are editable in a ***"parameter dialog"***. Often this is for the data types. Here also the types are shown with its simple short designation.

(empty page)

5.9 Operations to FBlocks inside the data flow (FBoperation)

Table of Contents

5.9.1 void Operation with input(s) and reference output

As shown in the overview chapter *5.6.9 GBlocks for operation access in line in an expression - FBoper* this is the possibility for operations to FBlock instances in the Object Oriented kind. Familiar FBlock tools does not support Object Orientation. The reason may be that the FBlock graphic was already founded in the 1970..80er where the ObjectOrientation was not familiar for embedded control in that time. Today, object orientation has still not been used extensively in the embedded control. But a look at ObjectOrientation can also be helpful there in understanding and systematizing algorithms. That's why this contribution to the FBlock world in OFB may be an important contribution for software technology.

Look first to an example:

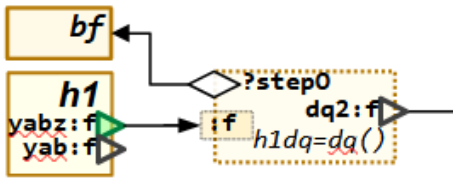


Figure 136: OFB/Fboper_h1dq.png

The FBlock `bf` is type of `OrthBandpassF_Ctrl_emC`, as defined in the OFB graphic on another page. Here the FBlock is only repeated to have a short way for the aggregation connection. The `h1dq` is the FBoperation. The text '`?step0`' on the aggregation is the hint to connect this FBoperation after the `step0` event with the aggregated `bf`. It saves effort to draw also the event connection. But the event connection cannot be found automatically.

The '`=dq()`' describes the name of the operation `dq`, in full C code generation it is `dq_OrthBandpassF_Ctrl_emC`, or just in C++ `dq` as class operation.

The expression input (style `ofpExprPart`) with '`:f`' is the first and only one input value as first argument of `dq(…)` operation. The variable `dq2` is a *"variable after expression"* with style `ofpVout…` .

Hence, in C code generation with step as input event in the module it is:

```
void step_OrthBandpassFilter ( OrthBandp...
  .....
  step_OrthBandpassF_Ctrl_emC(&thiz->bf, ...
  .....
  dq_OrthBandpassF_Ctrl_emC(&thiz->bf,
                thiz->h1.yabz, &thiz->dq2);
```

and adequate in C++:

```
void OrthBandpassFilter ::step (...
  .....
  bf.step( ...)
  .....
  bf.dq( this->h1.ybaz, &this.dq2);
```

By the way: The C++ code is shorter, maybe better readable. But the C code is more obviously, nothing is hidden. Both codes may produce exact the same machine code.

The called operation has the following prototype (in C):

```
void dq_OrthBandpassF_Ctrl_emC (
  OrthBandpassF_Ctrl_emC_s* thiz,
  float_complex ab, float_complex* ydq_y);
```

5.9.2 What is stored in the IEC61499 FBcl.fbd file:

```
FBS
  .....
  dq2 : VarV_OFB;
  dq2_X : FBoper_OFB( expr:='$(,(;,;dq;' );
```

The FBoperation is the `dq2_X`, `dq2` is the variable after operation. The operation has an `expr` input as also FBexpr. As described in *5.6.9 GBlocks for operation access in line in an expression - FBoper* page *110*.

The first character '`$`' is the access, it means:

`@`: it is inline in a expression term as FBoperation (access with THIZ to a FBlock), without more outputs, but possible input arguments.

`%`: inline in an expression term as FBoperation, but with some additional outputs necessary as call be reference in C/++. The additional output variables may be `ofpDout`, `ofpVout`, `ofpZout`.

`$`: an FBoperation which sets all outputs to output variables, hence it is called as statement.

The second character is always '`(`' as designation as FBoperation.

The `expr` input is the same as for FBexpr, especially input variants have the same possibilities as in expressions, see *5.8.2 Expression data input pins DinExpr ofpExprPart* page *130*. The name of the operation is written after the second semicolon.

Furthermore the event and data connections are important, for this example:

```
EVENT_CONNECTIONS
  .....
  bf.stepO TO dq2_X.dq;
  dq2_X.dqO TO dq2.prep;
  dq2.prepO TO .....
```

The first connection is from the stepO to the `dq` event input to the FBoperation. It clarifies the execution order, `dq2` is executed after the step operation of the instance `bf`. The second line is the event flow from the FBoperation to the variable after expression (which is set implicitly with the execution of the FBoperation).

```
DATA_CONNECTIONS
  ......
  bf.THIS TO dq2_X.THIZ;
  h1.yabz TO dq2_X.X1_dq;
  dq2_X.y_dq TO dq2.X;
  dq2.V TO .....
```

The `bf.THIS TO...THIZ` is the aggregation which clarifies implicitly the type of the FBoperation respectively the FBtype of the associated FBlock, it's the type of `bf`.

The input and output pin types of the FBoperation are defined in the FBtype of the associated FBlock, here `bf` defined as:

```
FUNCTION_BLOCK OrthBandpassF_Ctrl_emC
EVENT_INPUT
.....
  dq WITH X1_dq;
EVENT_OUTPUT
 dqO WITH y_dq;
VAR_INPUT
  X1_dq : CREAL;
VAR_OUTPUT
  y_dq : CREAL;
```

For the internal data mapping also the PinType_FBcl instances are contained in the Fbtype_FBcl data as member. There is no specific FBtype for the FBoperation instance, instead the FBoperation instance (Graphic Block) is always associated to the FBlock with the representing FBtype. The name of the inputs and outputs regarded to the FBoperation are denominated as `X..oper` and `Y...oper` inside the FBtype with `X` and `Y` starts from `1`.

That's the view to the internal textual data mapping as bridge between graphic and generated code – for this first example.

5.9.3 Operation with return value and reference outputs

Now look to a more complex example for usage of a FBoperation.

For this example the prototype of the operation is given in C as:

```
float getnmOscil_Angle_abgmf_Ctrl_emC ( Angle_abgmf_Ctrl_emC_s* thiz
, float m, float nm, float_complex* ab, float_complex* anb);
```

The graphic application looks like:

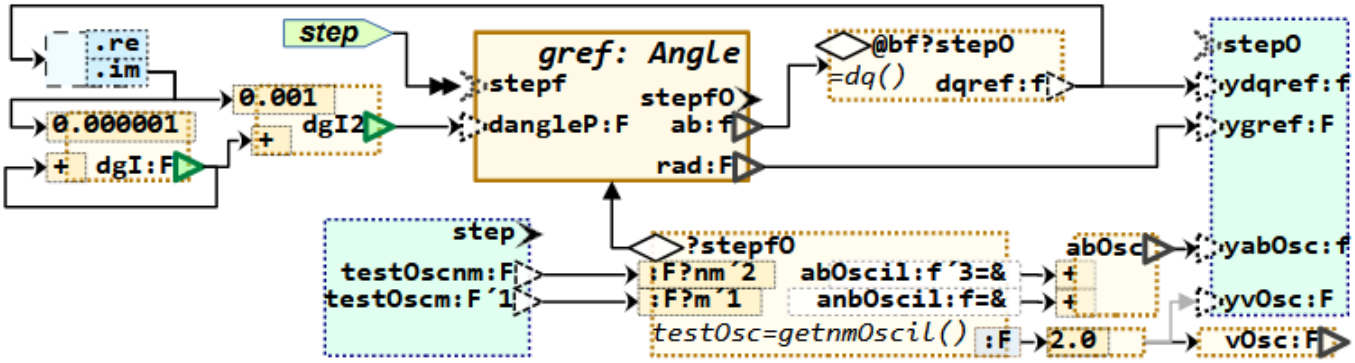


Figure 137: OFB/FBoper_getnmOscil.png

The FBoperation on bottom `testOsc` has its known aggregation with step event input, and two input values as float. The outputs are provided adequate to the prototype via two reference variables or for the graphic, with variables after expression, and one return value.

Only for interesting: This part of the module is a PI-control algorithm for the frequency for the measurement signal on `bf`, similar as the *"AFC"* in a analog FM receiver, but usual for electrical grid frequencies.

The return value builds an inline data flow. A return value is designated with an `ofpExprOut` pin. It is marked here with the type `:F` for `float`. No more is necessary.

If the designation with ***´nrPin*** is not used (see *5.5.6.5 Order of module pins* page *90*), then **the graphical position of the pins determines the order of arguments for the generated code**. The order is **left from top to down, then right top to down**. It is also usual in C/++ to organize outputs more right in argument order. If it is not so (legacy code) you can either write a macro-wrapper or position outputs also left in the graphic.

For this example the ***´nrPin*** **designation is used for three of the pins, only for test here. It means the order is defined by `´1` first, then `´2` etc.**

The execution order depends here not only from the event connection (`stepf0`) but also from the necessity of the return value. The FBoperation testOsc is called only if the return value is needed in an expression. If the output after the 2.0 expression is wired to the output variable yvOsc (shown in gray), then this FBoperation is called at least, if all other values for the stepO event are also prepared. If the return value feeds a variable, as shown here, feeding the variable does not depend from other values. Hence this FBoperation is called earlier. If the expression to the variable depends in other values, after the FBoperation itself, the FBoperation is only called if all other data are ready.

If the return output is not connected, though the other variables are connected, the FBoperation is never called.

This should be all regarded. The simplest case is a short connection to any variable. Providing a return output is often usual by given C/++operations (also using legacy code). If it is desired to embed the FBoperation in an inline expression, it is the proper way to do.

For this example the code generation looks like:

```
thiz->vOsc =(
getnmOscil_Angle_abgmf_Ctrl_emC(&thiz->gref
, testOscm, testOscnm, &thiz->abOscil
, &thiz->anbOscil) * 2.0) ;
```

How the FBcl files (IEC61499) looks like:

```
FBS .....
testOsc : FBoper_OFB(
     expr:='%(,(,(;,,;getnmOscil;' )

EVENT_CONNECTIONS .....
gref.stepfO TO testOsc.getnmOscil;
.....
testOsc.getnmOscilO TO abOscil.prep;
testOsc.getnmOscilO TO anbOscil.prep;
testOsc.getnmOscilO TO d_22.prep;
d_22.prepO TO vOsc_X.prep;
vOsc_X.prepO TO vOsc.prep;
```

The first event connection is the first time or condition to execute this FBoperation. The others are from the outputs.

Because the variable `vOsc` is no more connected, the variable is set but not used. If the gray connection to `yvOsc` is used, then the `d_22.prepO` will be inputted in a `Join_OFB` FBlock with the other events feeding the `stepO`.

```
DATA_CONNECTIONS ....
gref.THIS TO testOsc.THIZ;
```

… the aggregation connection.

```
testOscm TO testOsc.X1_getnmOscil;
testOscnm TO testOsc.X2_getnmOscil;
```

… the both inputs.

```
testOsc.abOscil_getnmOscil TO abOscil.X;
testOsc.anbOscil_getnmOscil TO anbOscil.X;
testOsc.y_getnmOscil TO d_22.X1;
d_22.y TO vOsc_X.X1;
vOsc_X.y TO vOsc.X;
```

… the outputs of the expression.

5.9.4 Join_OFB for inputs for calculation order

The *Figure 137: OFB/FBoper_getnmOscil.png* shows a second FBoperation on mid top: dqref. It is similar is the example on the page before for h1dq, it is the same FBoperation, used a second time. The aggregation is here textual given with '`@bf?stepO`'. This operation is called independently with the h1dq, with other input data, other output, but the same operation of the C/++ struct or class:

```
dq_OrthBandpassF_Ctrl_emC(&thiz->bf
, thiz->gref.ab, &dqref);
```

The event input of this FBoperation instance has a Join_OFB before, because both, the bf FBlock should be finished, as also the data on ab of gref should be given:

```
EVENT_CONNECTIONS .....
bf.stepO TO JOIN_dqref_X_dq.J1;
gref.stepfO TO JOIN_dqref_X_dq.J2;
JOIN_dqref_X_dq.J TO dqref_X.dq;
```

That is the only one difference. The Join_OFB will be automatically inserted due to *5.12.1 Event and Data flow* page *196*.

5.9.5 A FBoperation as simple getter

General a simple getter FBoperation is the same as the access to an `Dout` output pin of the aggregated FBlock. But if the getter FBoperation does more as a simple access, a longer calculation or, which is possible, change of data on access, then it is more obviously to use an extra FBoperation for that.Here a value for the phase deviation from another FBlock `h1` is necessary as input for the next FBlock `wf1data1`. The prototype in C language it is:

float phase_OrthBandpassF_Ctrl_emC (

```
  OrthBandpassF_Ctrl_emC_s* thiz)

<:@image:./../img/OFB/FBoper_phase()-
getterl.png :: id=__Img_OFB_FBoper_phase()-
getterl ::
title=Figure 60: OFB/Fboper_phase()-
getterl.png :: style=ImageCenter ::
size=9.0cm*2.54cm :: px=365*103 :: DPI = 103.>
```

The call of this operation is very simple mapped to the graphic.

5.10 FBlocks in slices, access to slices

See also the overview chapter *5.6.12 Sliced or Array FBlocks, Demux and array data* page *116*. That chapter shows also a small comparison with Simulink © Mathworks.

5.10.1 Vectors in expression

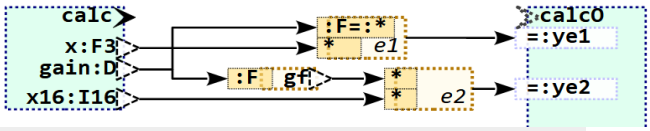


Figure 138: odg/ArraySlideDemux_VectorExpr.png

This image shows an example for vector multiplication. The `gain` is scalar as `double`. The both `x`, `x16` inputs are vectors `float[3]` ind `int32[16]`. The code results in:

include:../../+BasicTest/cpp/genSrc/ArraySlideDemux.c::'gf = '#13::45::2:-::6:+

```
  float gf = { 0 };  // #FBevin_gf_X_prep ...
  // todo thiz->mEvout_calc = 0;  // @5'20...
    ......
  v1[2] = (x[2] * (float)(gain) * thiz->ga...
  step_FBx(&thiz->fb1a, xa, (float)(gain),...
  step_FBy(&thiz->fb2a, thiz->fb1a.y2);  /...
  step_FBx(&thiz->fb1a2, xb, (float)(gain)...
  step_FBy(&thiz->fb2a2, thiz->fb1a2.y2); ...
  step_FBx(&thiz->fb1b, xc, (float)(gain),...
  step_FBy(&thiz->fb2b, thiz->fb1b.y2);  /...
  step_FBx(&thiz->fb3[0], xa, (float)(gain...
  step_FBx(&thiz->fb3[1], xb, (float)(gain...
  step_FBx(&thiz->fb3[2], xc, (float)(gain...
```

It generates, as expected, one line per vector element.

For the expression `e2` there is a nuance: The casting to the `(float)` is done with a local variable. Then this `gf` is used. It optimizes code before C language. Maybe the compiler itself can also optimize the repeated `((float)gain)` castings.

It is not realized yet but planned that more as a parameterized number of repeated similar lines for vector elements should be produce a for loop in code, for optimizing machine code size. The repeated assignment lines optimizes calculation time.

The intermediate FBcl language shows:

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'FBS'#4::45::2:-::3:+

```
FBS
  JOIN_calcO : Join_OFB                    ...
    ......
  e1 : ARRAY [0..2] OF Expr_OFB( expr:='~*...
```

Both expressions are arrays. The code generation regards data flow with vector inputs of the same size: each inputs gets one element. Or just data flow with scalar inputs or also vector inputs of a lesser dimension, then any input gets the same value as shown for gain. The data flow is shown in the FBcl file as:

```
DATA_CONNECTIONS
```

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'TO e1.X2'#1::45

```
  x TO e1.X2;  (*:F3  evChain=0->;   src:@...
```

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'TO e1.X1'#1::45

```
  gain TO e1.X1;  (*  evChain=0->;   src:@...
  gain$calc TO e1.X1;  (*  m_evinMdl=0x1  ...
```

5.10.2 Vectors and scalar FBlocks

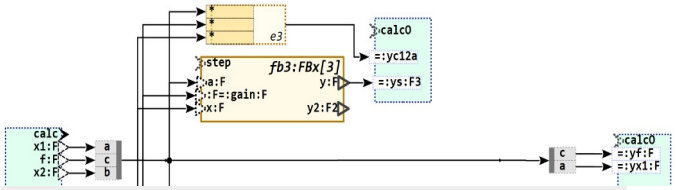


Figure 139: odg/ArraySlideDemux_VectorFBlock.png

This example shows an ordinary scalar FBlock, a T1 smoothing block, which is used three times by connections with vectors. The FBlock has an array designation after the type, it is exists 3 times. The input `x` is `float[3]`, proper each element to each T1 block. The data type of output `yts1` is automatic calculated also with `float[3]` or `F3` due to the output `Ts.y` as `float` (`F`) and the `[3]` of the instances.

The smoothing time or factor is set in the init, same value for all three instances. For that you can omit the [3] designation. It is known that the ts1 is instantiated three times.

The header file for this part looks like:

include:../../+BasicTest/cpp/genSrc/ ArraySlideDemux.h::'Ts1FiltSimple_s'#1::45

```
    Ts1FiltSimple_s ts1[3];  // ts1:Ts1Fil...
```

See the definition of a vector of struct.

The FBcl file contains also this three instances:

```
FBS
```

include:../../+BasicTest/cpp/genSrc/FBcl/ ArraySlideDemux.fbd::'Ts1FiltSimple'#1::45

```
  ts1 : ARRAY [0..2] OF Ts1FiltSimple( fs:...
```

(empty)

In the generated C-source there is:

include:../../+BasicTest/cpp/genSrc/ ArraySlideDemux.c::'step_Ts1FiltSimple'#3::45

```
  step_Ts1FiltSimple(&thiz->ts1[0], x[0]);...
  step_Ts1FiltSimple(&thiz->ts1[1], x[1]);...
  step_Ts1FiltSimple(&thiz->ts1[2], x[2]);...
```

... called three times. It should be also possible to create a for-loop for that (Req)

The output signals are gotten in a for-loop for the vector elements, due to the code generator:

include:../../+BasicTest/cpp/genSrc/ ArraySlideDemux.c::'step_Ts1FiltSimple'#3::45

```
  step_Ts1FiltSimple(&thiz->ts1[0], x[0]);...
  step_Ts1FiltSimple(&thiz->ts1[1], x[1]);...
  step_Ts1FiltSimple(&thiz->ts1[2], x[2]);...
```

For optimizing code size / calculation time here the selection between for-loop and more lines should be also possible (Req).

Of course, this FBlock needs an update, generated due to the existence of the upd pin and the connection of the Zout variable:

include:../../+BasicTest/cpp/genSrc/ ArraySlideDemux.c::'upd_Ts1FiltSimple'#9::45

```
  upd_Ts1FiltSimple(&thiz->ts1[0]);  // #F...
  upd_Ts1FiltSimple(&thiz->ts1[1]);  // #F...
  upd_Ts1FiltSimple(&thiz->ts1[2]);  // #F...
  // TODO think about set bit for evoutMdl...
  // Module outputs due to the event updO:...
  //todo: thiz->mEvout_upd |= MASK_upd_upd...
  for(int ix = 0; ix < 3; ++ix) {
    thiz->zts1[ix] = thiz->ts1[ix].yz;    ...
  }
```

The FBcl file contains the proper data and event connection to have the bridge between graphic and generated code.

5.10.3 Slices of named FBlocks

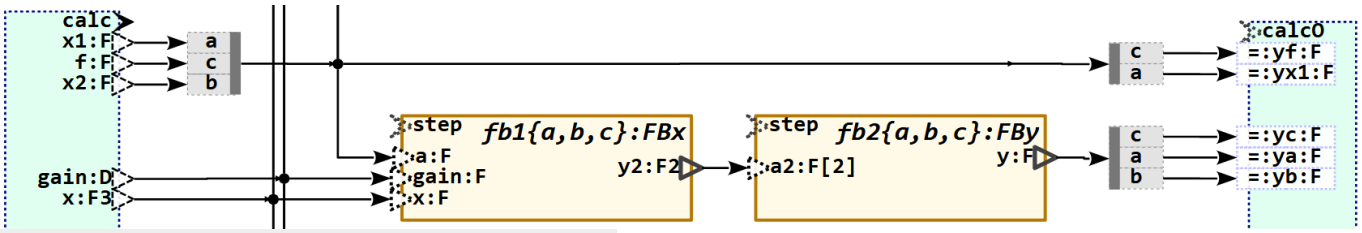


Figure 140: odg/ArraySlideDemux_DemuxFBlock.png

In opposite to the vector FBlock in the chapter before, the two graphic blocks (GBlock) presents each three different named FBlocks with the built name `fb1a`, `fb1b` and `fb1c` and `fb2a`, `fb2b`, `fb2c`. They are not defined as vectors. This may have an advantage for code and documentation. It are independent FBlock from the view of the source code. But they have equal or similar connections, so that space is saved in the graphic and (more important) the functionality in the graphic may be more clearly arranged.

Look firstly in the FBcl files for the `FBS` definition, you see independent FBlocks:

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'fb1a : FBx'#6::45

```
  fb1a : FBx_FB                                (*...
  fb1a2 : FBx_FB                                (...
  fb1b : FBx_FB                                (*...
  fb2a : FBy_FB                                (*...
  fb2a2 : FBy_FB                                (...
  fb2b : FBy_FB                                (*...
```

And the header file also:

include:../../+BasicTest/cpp/genSrc/ArraySlideDemux.h::'fb1a'#6::45

```
    FBx_s fb1a;  // fb1a:FBx_FB@5'60(59..8...
    FBx_s fb1a2;  // fb1a2:FBx_FB@5'60(59....
    FBx_s fb1b;  // fb1b:FBx_FB@5'60(59..8...
    FBy_s fb2a;  // fb2a:FBy_FB@5'90(92..1...
    FBy_s fb2a2;  // fb2a2:FBy_FB@5'90(92....
    FBy_s fb2b;  // fb2b:FBy_FB@5'90(92..1...
```

In such cases the inputs may not be vectors, they are here different signals `x1`, `x2` and `f` from inputs. To build one connection to the GBlock for all three FBlocks, a Multi- / Demultiplexer is used. This is a shape of style `ofbDemux` presented with a small gray bar. The pins of the Demux are built from shapes of `ofPin` style `ofpDemux`. Depending from incoming or outgoing data connections this are the signal names to multiplex to a ***bus***, or demultiplex from bus, or better a ***wiring loom***. The bus itself is not named (as also connection lines have no names), but can use a Xref to connect also between pages.

The names of the Mux and Demux pins have no relations to the connected signals. For the graphic in the image above the names of the multiplex pins are the same as the input signal names. This is sensible, but not necessary.

The order of signals for Multi- / Demultiplex is not important for the signal selection. But for usage a Multiplexer output as vector, it is important. See next chapter.

The names on the Mux pins are essential for the assignment signals to the named slices, here `{a, b, c}`, The input `x1` is assigned via the Demux pin `a` to the `fb1a` etc. Same is with the Demux, the pin `c` gets the `fb2c.y` to connect to `yc`. In the FBcl it looks like:

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'TO fb1a.a;'#1::45

```
  xa TO fb1a.a;  (*:F  evChain=0->;   src:...
```

and in C/++ it appears as

include:../../+BasicTest/cpp/genSrc/ArraySlideDemux.c::'step_FBx(&thiz->fb1a'#6::45

```
  step_FBx(&thiz->fb1a, xa, (float)(gain),...
  step_FBy(&thiz->fb2a, thiz->fb1a.y2);  /...
  step_FBx(&thiz->fb1a2, xb, (float)(gain)...
  step_FBy(&thiz->fb2a2, thiz->fb1a2.y2); ...
  step_FBx(&thiz->fb1b, xc, (float)(gain),...
  step_FBy(&thiz->fb2b, thiz->fb1b.y2);  /...
```

include:../../+BasicTest/cpp/genSrc/ArraySlideDemux.c::'thiz->yc = '#3::45

```
  thiz->yc = thiz->fb2a2.y;     // #genDinA...
  thiz->ya = thiz->fb2a.y;     // #genDinAc...
  thiz->yb = thiz->fb2b.y;     // #genDinAc...
```

It means, the slice definition is no more seen, neither in the FBcl code nor in C/++. It is dissolved, it is only in the graphic. That is other for vector FBlocks, where the definition of vectors is in the code.

5.10.4 Mux and Demux, build vectors with Mux

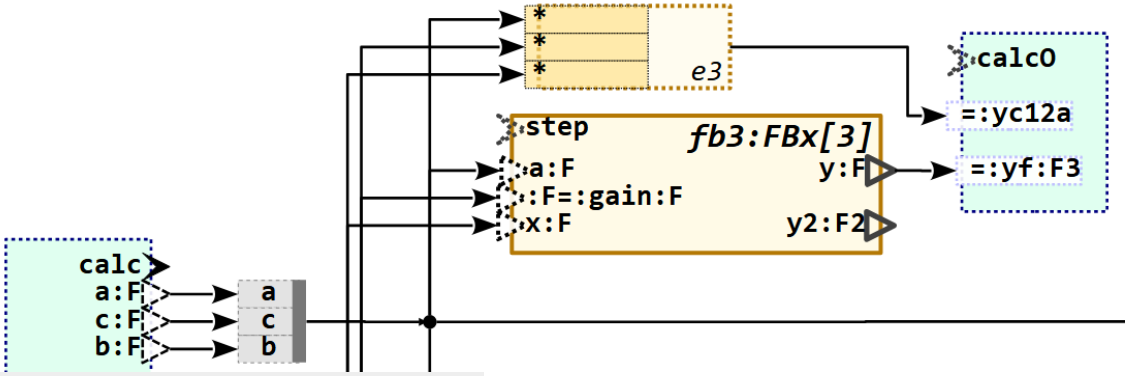


Figure 141: odg/ArraySlideDemux_VectorFBlock.png

The multiplexed signal bus wiring loom can be used for vector inputs also. Look to the image above. Here the same signals as in *Figure 140: odg/ArraySlideDemux_DemuxFBlock.png* are used for the vector multiplication expression `e3` as also for the vectored FBlock `fb3`. Here the order of signals from top to down (or left to right if the multiplex bar is horizontal) determines the vector elements `[0]` till here [2].

But a Demultiplexer bar cannot be used for access to vector elements (in Simulink it is possible). This is too unspecific. It is better to use the specific constructs in the next chapter.

5.10.5 Build vectors with elements, access to vector elements

5.11 State Machines in OFB Diagrams

5.11.1 State Machines in the past, history, hardware, PLC

A ***StateMachine*** (**StM**) is an execution mechanism to changes values, which defines the state of a module. The state can influence the kind of functionality and can be proper shown to outside. The state as a whole is representing by a set of given values.

History and other Implementations

State machines have its base on the theory of digital automates, for example forced by pioneers of digitisation as Alan Turing (1912 - 1954). Already in that time similar graphic representations of state behaviour becomes familiar as here drawn in OFB graphic:

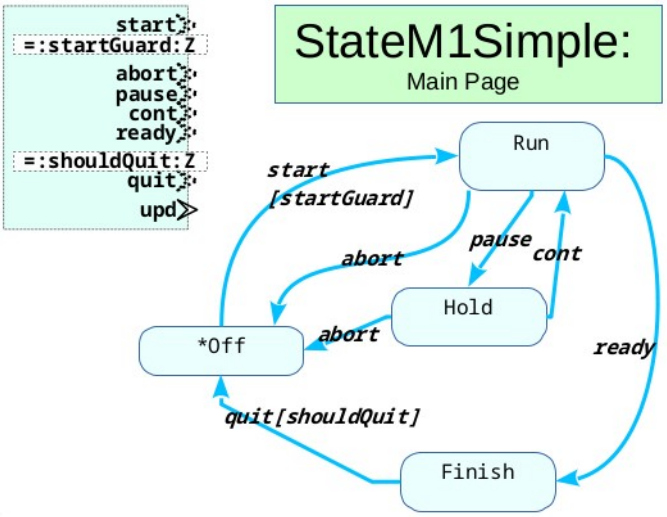


Figure 142: stm/StateM1Simple-OFB.png

But for implementation first relays and switches are used. A state implemented with a relay as self locking circuit is named *"FlipFlop"*, because its sound on set with *"flip"* and on release with *"flop"*. Adequate graphical circuit diagrams were familiar in hardware wiring and also in the *"ladder graphic"* for PLC = *"Programmed Logic Control"*.

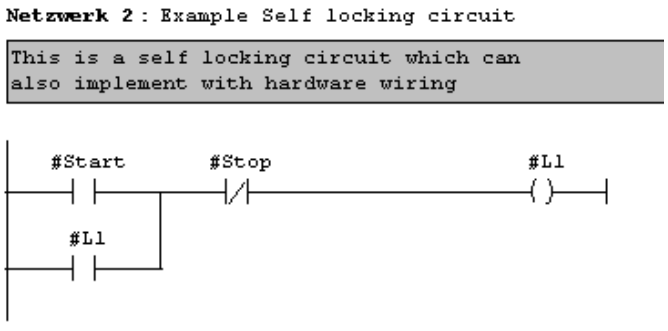


Figure 143: S7/LadderExmplSelfLockingCircuit.png

This image shows such a self locking circuit, the simplest state machine, drawn in Simatic S7 tool as so named "ladder graphic". The ladder graphic is one diagram type of the *"Programmable Logic Control"* (PLC), standardised in the norm IEC61131-3, created in the end of 1960th, currently still familiar, but with some more elements which does only work in software implementations.

In this simple presentation, it can be view as a hardware wiring. `#L1 -()-` is a coil of a relay, `#L1 -| |-` is a switch closed if the coil has current, `#Start -| |-` is an closing on press switch, and `#Stop -|/|-` is an opening on press switch. If the Start switch is pressed, then the coil `#L1` gets current, closes the `#L1` switch and holds its state though Start is released. If Stop is pressed as opening switch, the coil goes off.

The logic shown as StateMachine graph left side can be built also in hardware with Integrated circuits as so named ***RS-FlipFlop***, RS means "Reset" and "Set". The next image right side shows this state machine with RS FlipFlop in the so named *"Function Block Diagrams"* (FBD) of a Simatic S7 project. This follows hardware wiring ideas, though it is implemented in the *PLC*, the ***Programmed Logic Control***. It is more simple for the hardware oriented engineer of the 1970th area and later (till now) to understand what it does.

The other possibility to implement FlipFlop Logic is: Using so named *D-FF* which is given especially as 7474 as veteran, still in use, in the TTL series of basic logic integrated circuits. This kind of FlipFlop is gotten familiar against simple R-S-FF (Reset/Set). It works with a clock. On the clock edge the state on the D-Input is taken over to the Q-output. State machines can also be shown graphically with a combination of such D-FF and boolean logic. This D-FF logic is also familiar in FPGA circuits (*"Field Programmable Gate Arrays"*). But the FPGAs are today usual programmed in the textual *VHDL* or *Verilog* language, using boolean expression and assignment thinking. Only the view on implementation level shows these D-FF.

TODO show same example with FF Logic

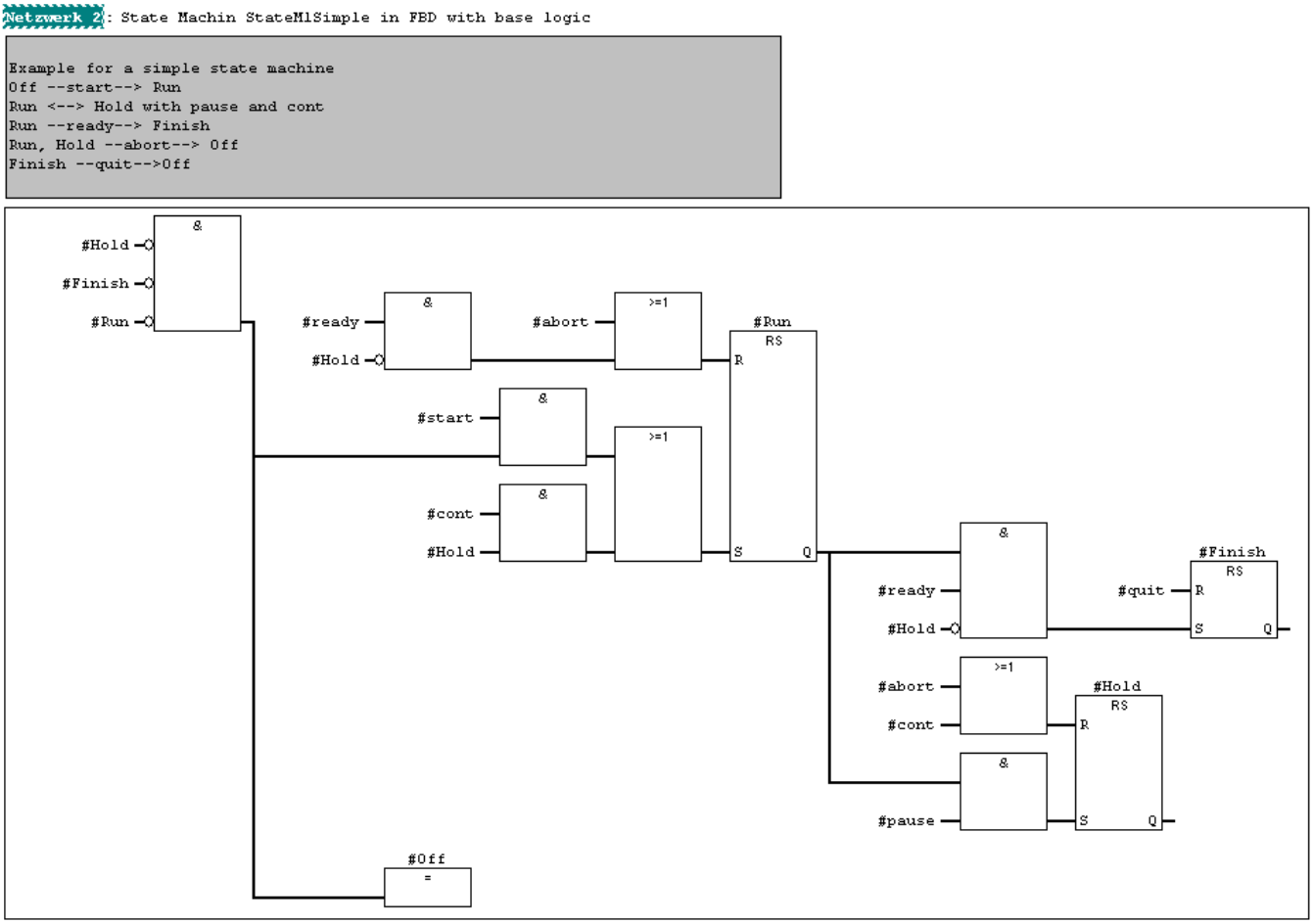


Figure 144: Stm/StateM1Simple_S7FBD.png

Runtime effects and D-FlipFlop in hardware

In the image above the signal to switch on the `Run` state is very short. Because, if the `Run` state gets active, the Set input on the RS-FlipFlop gets immediately inactive because of the AND-relation of `start` and `NOT Run` on the input. The AND relation is necessary because start should not Set the Run FlipFlop if another state is active. If this is realised in hardware, negative effects of **hazards may be possible.** That's why for hardware logic the so named D-FlipFlop were developed. The IC 7474 contains two D-FlipFlops. A D-FlipFlop has a D-input and a clock-input. If the clock comes with a Low-High edge, then the value on D is transferred to the Q output. Whereas a jittering value is sampled with the clock edge in an internal intermediate FlipFlop to ensure a stable state though jittering. The inner circuit is the following, the source of this image is https://de.scribd.com/document/45032899/7474-Datasheet. You may detect three crossing lines for the known RS-FlipFlop combinations.

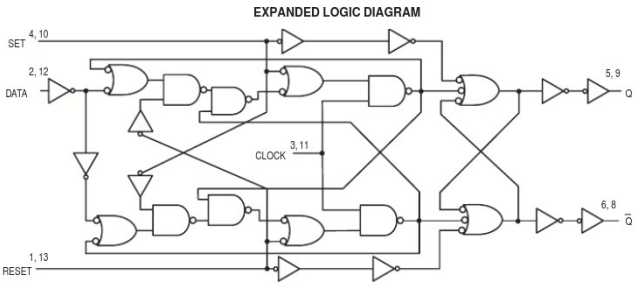


Figure 145: Stm/D-FlipFlop_7474-inner.png

In FPGAs (Field Programmable Gate Arrays) exclusively D-FlipFlop are used to ensure an exact timing. - Without detailed explanation - the idea of clocking the processed signals in hardware is allied with the using of step and update for software data processing.

Event control

The idea of event control is substantial for the UML State Machines as well as for the processing of functionality in the OFB diagrams and in the IEC61499 norm.

All these things are topics of State Machines.

5.11.2 Terms related to state machine technology

This chapter denominates some terms around State Machine technology in order of usage one after another. An alphabetic sorting would not get an overview. Use searching operations (ctrl-F) to find a term in digital read documents.

The terms which are defined here are written starting with an upper case letter, because they are proper names, for this denomination. Additional the *CamelCase* writing style is used as also in the OMG UML Document for example for the term **StateMachine**. Attributes which are not combined in the CamelCase word is not a part of the term, but an explanation, just an attribute, written with lower case inside a sentence. For example *orthogonal CompositeState*. is a *Composite-State* which has *orthogonal* Regions.

The standard for UML conform StateMachines is managed by the ***OMG***, an organisation *"Object Modelling Group"*. The web side is omg.org. The current standard is downloadable under https://www.omg.org/spec/UML/. For this document the version 2.5.1. is referenced, named here **OMG UML document**.

Furthermore another reference for UML StateMachines is also the https://en.wikipedia.org/wiki/UML_state_machine.

The **OMG UML document**. has not at all a really proper definition of terms. This document want to be follow the **OMG UML document**, but a precise definition of the terms is necessary here.

- A ***StateMachine*** (**StM**) is an execution mechanism to changes values, which defines the state of a module.

* Sometimes the term ***Finite State Machine*** (***FSM***) is used in the literature. The *"Finite"* comes from theoretical considerations especially from the time of Alan Turing and himself. it is not used here because all State Machines are finite.

* ***UML*** is the ***Unified Modelling Language***, used for Architecture and Design and also for implementation of software.

* A ***state*** as common term is one stable or volatile state presented by given data. The graphic appearance of a state is denominated in the **OMG UML document** as ***vertex***. For simple usage the graphic appearance, the vertex, is denominated here also as ***State***, written with starting upper case as proper name. Often quests `isinState(...)` are used in in textual languages to ask whether a definitely State is active. The state as a whole from the StateMachine can be presented by more as one active States in different ***OrthogonalRegion***s.

* A ***Transition*** is the switch or ***edge*** in graphic from one to another state. A transition can be triggered by an ***event*** and can have a ***guard***, or can quest ***condition***s.

* An ***Event*** is from the pure word meaning an occurrence of anything. In common meaning the information that an event will be occur is a ***message***. But for StateMachine in in software the Event is usual used also in the meaning of message. A message can be stored in a queue, and the fact that an event will be occur can also be stored. But usual this information is also denominated as ***Event*** in the queue. An event ***triggers*** an ***action***. For StateMachines this is especially triggering a ***Transition*** switching from the current active state to its destination state.

* A ***Guard*** is the condition on a ***transition***, which is tested with the event, which should switch the transition. The transition does not switch and the event is not consumed, if the guard results in false.

* A ***Condition*** is used for a transition, which is not driven by an event, hence used for Run to complete.

* ***Run to complete*** is a paradigm used for switching in StateMachines, well described in the **OMG UML document**. It means that after switch a transition triggered by an event, all transitions of the following activated states which are not event triggered are tested and used to switch, till no more transitions can be switch. Only then the next event (from a queue) is applied to the StateMachine.

* ***Region*** is a part of the StateMachine, which is either the top level behaviour or

which is inside a *CompositeState* as sub behaviour. See ***OMG UML Document*** 14.2.3.2 Regions page 307.

* ***ToplevelRegion*** means the region from the whole state machine's top level. Each StateMachine has at least one Region, which is the ToplevelRegion.

* ***Orthogonal Regions***: That are more as one Regions which have its own separated behaviour, switching independent. In original UML the orthogonal Regions are drawn in one CompositeState, which has divisions separated with dash lines. To set specific States in each of the orthogonal Regions a Fork transition is possible. To check specific States in each of the orthogonal Region to switch out of the orthogonal CompositeState, a Join Transition is possible.

* ***CompositeState*** or also denominated as ***SuperState*** is a state which contains some other states in its inner, one ore more orthogonal ***Regions***. A *CompositeState* is used as a State as each other from the focus of its own Region, where it is a part from. It can have its own Entry- and ExitActions as any other State. In explanation of a behaviour, the inner content of a Composite or SuperState can be abstracted from.

  The ***OMG UML Document*** uses the denomination ***CompositeState*** in chapter 14.2.4.4.1 Composite State page 321 especially for the **Parent State of all orthogonal Regions**. But for example in *figure 14.7 Composite State with two States* also a state without orthogonal regions is shown. It means all not simple states are composites.

* ***OrthogonalCompositeState*** or shorter ***OrthSuperState*** is a more fine denomination that the *CompositeState* contains orthogonal Regions.

* ***RegionState*** is the state which is responsible as parent for one Region. If this Region is not orthogonal, a simple nesting of States, then this is also the ***Composite*** or ***SuperState***. If the Region is orthogonal to more Regions, this is the ***OrthRegionState***, see next.

  OrthogonalRegionState or short ***OrthRegionState*** denominates the comprehensive state which contains one Region of some orthogonal ones. This parent is intrinsically not a State by itself, the *orthogonal CompositeState* which contains all orthogonal Regions is the intrinsic used *SuperState* seen from outer. But the *OrthRegionState* has a name, and has in generated code a variable for the current State in the Region. In UML due to the ***OMG UML Document*** the o*rthogonal CompositeState* and its containing *OrthRegionStates* are drawn with one State (with rounded corners), containing dashed lines to divide its inner area. This divisions presents the OrthRegionState. It contains the sub States of the Region. See *Figure 14.9 Composite State with Regions* page 322. The **OMG UML Document** has not a definitely name for these *OrthRegionStates*.

* ***Nested States*** are these states which are inside a *Region*. They are the members of the *Region*. Seen form a *SuperState* (*CompositeState*) these are ***SubStates***.

* ***Fork*** transition or ***Fork Pseudostate***: The Fork transition is used to activate dedicated states in ***Orthogonal Regions***. In the OMG UML Document it is denominated as *"PseudoState"*, because it is between Transitions. An event is notated to the only one incoming transition to switch over the Fork to all destination state in the Orthogonal Regions. A missing Orthogonal Region for transitions starts on its default state.

* ***Join*** transition or ***Join Pseudostate*** has more as one incoming transitions from dedicated states of Orthogonal Regions (Parallel Nesting States). An **event is notated on the outgoing transition** to switch over the Join to the destination state. A missing Orthogonal Region of incoming transition is aborted in its current state.

* ***PseudoState***: This is a term used in the **OMG UML document**. for some vertexes necessary for organisation of switching.

5.11.3 State Machine in Embedded Control, UML and OFB

This chapter presents the features and properties of StateMachines in UML and the kind how this is drawn and implemented in OFB. For Details of OFB graphic refer the next main chapter *5.11.4 Details of StateMachines in OFB* pate *174*. For details execution principles refer chapter *5.11.5 Organisation of execution of the StateMachines* page *180*

5.11.3.1 History

First, in the 1960th Petri-Nets become familiar, developed from Carl Adam Petri, a German engineer (1926 - 2010). With Petri-Nets especially parallel processed states can be modelled. In the 1980th ***David Harel***, an Israeli developer (born 1950), creates a kind of state graphs, named as ***"State Charts"*** and used first for the tool *"Statemate"* from the Israeli company *"i-Logix"*. The same company has contributed in the 1990th to UML with its tool ***Rhapsody*** (today *"IBM Rational Rhapsody"*), and has introduced the ideas of the *Statemate* StateMachine principles in the UML. This type of State Machine presentation and execution is event based.

The UML kind of StateMachines are standardised in UML and also familiar in tools outside UML. For example Simulink knows an implementation of State machines, looks similar, but with unfortunately small differences in behaviour.

5.11.3.2 Manual programmed StateMachines, conditions and events

In Embedded Control simple state machines are often familiar by manual written code. Often a state variable as `enum` or a simple `int` variable in C holds the state identifying value. The core operation of state switch is a `switch-case` algorithm which selects the current state, checks any condition as transition, if it is met calls other actions on state switch, and sets the state variable with the destination state.

This manual programmed state machines uses often simple boolean conditions, as results of comparison, for switching the transitions. This seems to be in contradiction with the UML State Machine presentations with Events and State Charts. But is it really?

In opposite, The OFB concept regards completely the UML kind of state machine presentation, described in the omg.org standard. Some specific implementation goals ensure a similar handling of state switching without explicitly event usage to manual programmed state machines to use it also in fast cycle (interrupt) execution. See more explanation in *5.11.3.11 Execution principles of state machines - and OFB* and details in the chapter *5.11.5 Organisation of execution of the StateMachines180*

5.11.3.3 The UML StateMachines

The ***omg.org*** standard for UML describes only a few but important considerations how to deal with states:

* Events and Guards on transition
* Intention of actions on transition, entry- and exit actions.
* Nested States with the default state.
* The history pseudo state
* Parallel state execution with fork and join transitions.
* Rules how events are applied to levels on nested and to parallel state transitions.

The standardisation document see https://www.omg.org/spec/UML/ (yet the version 2.5.1. is referenced, named here **OMG UML document**) contains all these details, here explained as overview.

This all is also valid for the OFB state machines, maybe with some deviating in the graphic but not in functionality.

5.11.3.4 Nested states or Regions

If you look on the image of a simple StateMachine *Figure 142: stm/StateM1Simple-OFB.png164* in the first chapter, repeated here,

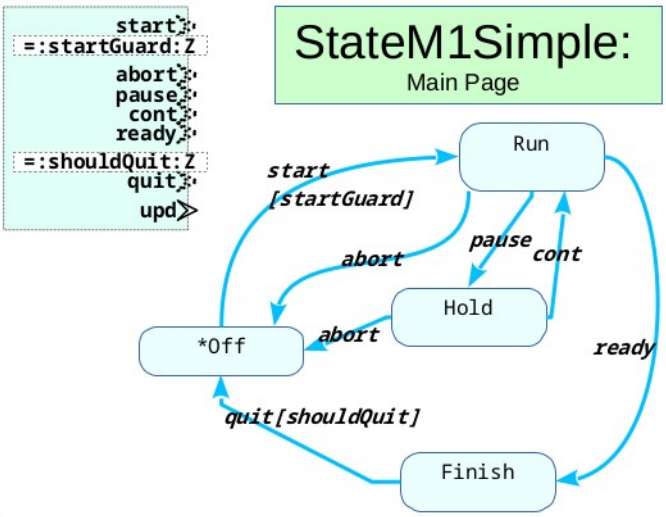


Figure 146: stm/StateM1Simple-OFB.png

it may be obviously that the states Run and Hold are more affiliated together. Especially from both the abort transition is drawn. It is better to draw the affiliated states in one superior state:

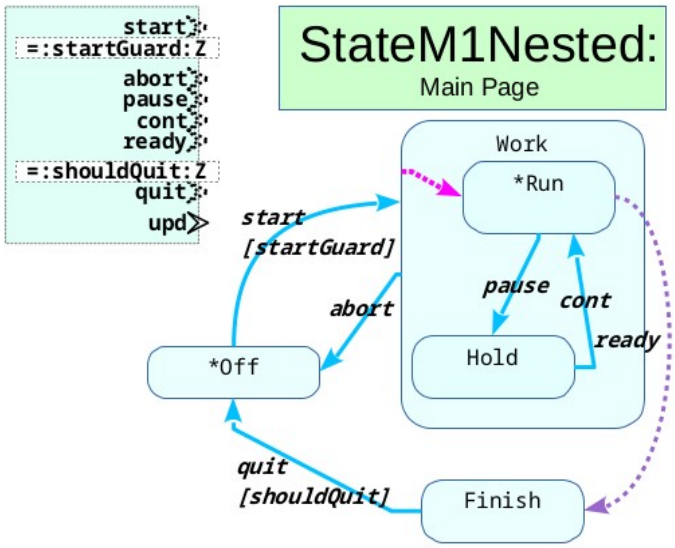


Figure 147: Stm/StateM1Nested-OFB.png

TODO image with Enterprise Architect

The image above have exact the same state behaviour. But here it is more able to see, that Run and Hold are an inner pair of states. Secondly the abort transition is only drawn one time. The start transition goes to the whole state `Work`, because `Run` is the default state (designated with asterisk before the name).

If states are more complex, it is also possible to show the behaviour outside a nested state with last one as black box, and show the behaviour of the nested state(s) on other diagrams. This gives more overview, it is a kind of structuring.

In the OFB kind of drawing, the transition from inside a nesting level to outside is additional designated with another style (`ofcStateTransChgRegion`), with is more obviously as in familiar UML state graphics, it is an additional graphical designation only, does not force another behaviour as standardised in UML omg.org (**OMG UML document**).

If the nesting state is activated from outside, here with the `start` event, its inner default state is activated. This is shown in UML with a transition from a so named ***InitialPseudoState*** to the default state, see in the Rhapsody diagram on next page. In the OFB diagrams it is the same functionality and behaviour, but the default state can be designated with the `*` before the state name, or also with the specific transition kind `ofcStateTransDefault`, shown here in ping colour. In OFB it is not necessary to draw the nested states inside the nesting state. The nesting - nested relation is also well defined by the `ofcStateTransDefault`, and by all transitions of style `ofcStateTrans`, which are only used between states of the same level, also designated as ***Region***. A transition from one to another nesting level or Region or from and to outside must have the style `ofcStateTransChgRegion`.

5.11.3.5 Parallel state execution, OrthogonalRegions, fork and join

The idea for the parallel states may come from the Petri Nets, see introduction to this main chapter, it is similar as in Petri Nets. But the latter have pellets, the Parallel State Execution in UML haven't. Also in the ***OMG UML Document*** Petri nets are mentioned, see page 313 in cohesion with the join transition.

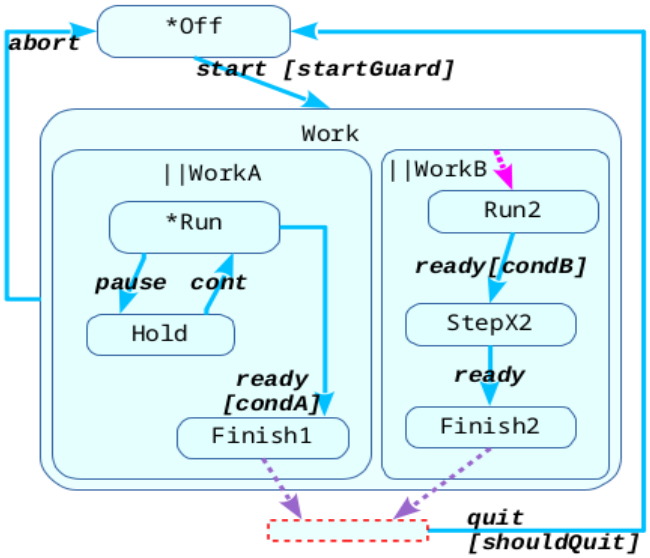


Figure 148: Stm/StateM1Parallel-OFB.png

The image shows an example for a parallel state machine execution in OFB graphic.

The same is also drawn in the UML Tool Rhapsody.

The parallel states have so named ***Orthogonal Regions*** (see ***OMG UML Document*** 14.2.3.2 Regions page 307).

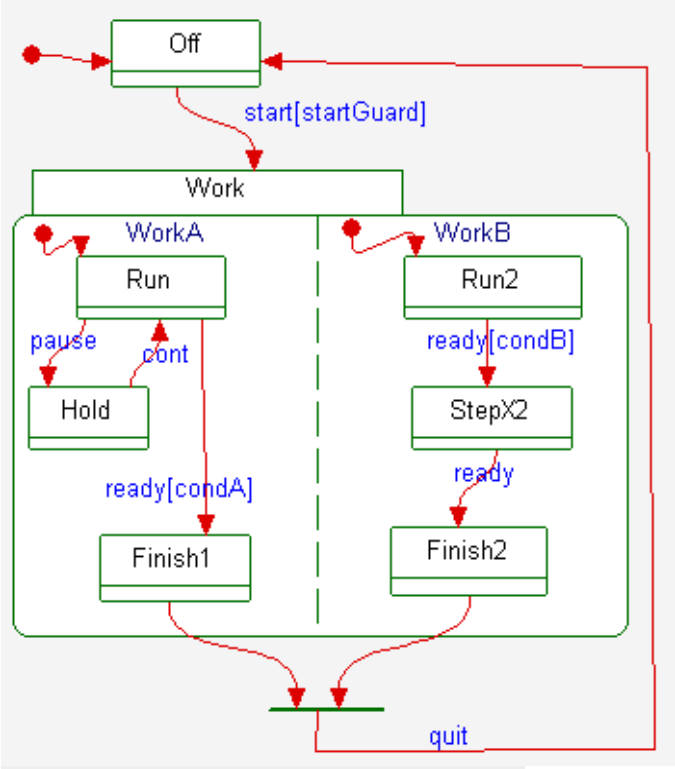


Figure 149: Stm/StateM1Parallel-Rhaps.png

There are only formally differences in the graphic, not functional ones. The parallel states are drawn in OFB with CompositeStates starting with the `||` parallel symbol, instead the dashed middle line. The default States of each Region can be drawn with a transition of style `ofcStateTransParent`, but from the here definitely existing enclosing (parent) state `WorkB` instead of the InitialPseudostate as shown for `Run2`, or more simple with the asterisk `*` before the name of the state `Run`, as well as for `*Off`.

5.11.3.6 Regions are independent

Regions are that part of State switching which are inside a CompositeState or also as OrthogonalRegions in Parallel States. The ***OMG UML Document*** page 316 chapter 14.2.3.9.1 The run-to-completion paradigm says: "*Due to the presence of orthogonal Regions, it is possible that multiple Transitions (in different Regions) can be triggered by the same Event occurrence. The order in which these Transitions are executed is left undefined.*" If the order is undefined, then the orthogonal regions can be switched independently. The same is for non orthogonal Regions. If they are not exited, the superior States are not switching.

As conclusion, Regions can be processed in different threads, or also can be located in different, distributed devices. If this independent regions are exited, this is especially in a Join transitions for orthogonal regions, then synchronisation mechanism are necessary, which ensures the correct "Run to completion" behaviour.

5.11.3.7 Applying events to transitions

There are definitely rules how events are applied to transitions. This can be explained in the best kind on parallel states.

The following rules are valid:

* An ***Event*** is applied first on the most inner active state of a composite (nesting) state. If this inner state does not have a transitions using this event, nothing is changed (nothing switches), then the event has to be applied to the more outer, the NestingState and so on till the top Region of the StateMachine.

* If ***Orthogonal Regions*** are active, the event is applied independently to all orthogonal Regions to switch transitions there. Only if is nowhere consumed, it is applied to the outer Regions. This behaviour is mentioned in the **OMG Document** only with one sentence in the chapter **14.2.3.9.3 Conflicting Transitions:** *"Only Transitions that occur in mutually orthogonal Regions may be fired simultaneously."*

* If any transition switches, then all other transitions from reached states are tested whether they can switch because of conditions without event. This is first done before the next Event is applied. This is the ***Run to Completion*** paradigm.

* If an event is not consumed, it is ***removed***, it is never repeatedly applied.

* ***Deferred events***: A state can store dedicated events to apply it for following switches, after it is left. This is not in contradiction to the principle *"remove the event if not used"* because it is deterministic used and stored. The stored deferred events are not applied to the outer Regions, they are used (to defer). See OMG UML document chapter 14.2.4.8.6 Deferred triggers page 333 and mention of *Deferred Events* on page 309.

* For events from any state outside to one or some states in ***OrthogonalRegions***, the following is valid: The event is applied to the incoming transition of a so named Fork Pseudostate, often more simple denominated as *"Fork Transition"*. Then all States in the orthogonal regions are activate, only one state per region. If an Orthogonal Region is missing, then it starts on its default state of this OrthogonalRegion.

* An event to exit OrthogonalRegions is either applied to only one single transition from the CompositeState of all Regions (Parallel-ParentState). Then it is similar to a transition starting on a single CompositeState (RegionState), the current state is left on switch, maybe remember in existing History state.

* The ***Join*** Transition or a ***JoinPseudostate*** is the other possibility. The event is notated to the outgoing transition, as shown in Figure 7: Stm/StateM1Parallel-Rhaps.png. But all incoming transitions switches with this event, if the source states in all regions are active. If a OrthogonalRegion is missing, then this region is left on its current state maybe remember in a HistoryPseudoState.

In the example for the parallel state execution left side, the `ready` event is used one time in the `WorkA` and two times in `WorkB`. If it occurs, it is applied to both of the parallel state execution, inside the independent (orthogonal) regions.

The second transition from `StepX2` to `Finish2` uses the same `ready` event. But for that the following rule is valid: If an event is used for one switch in a region, then it is consumed, not able to apply furthermore in the same region or in a parent region. (In the other Orthogonal-Regions it can be applied independently).

In the images on this pages a so named ***fork transition*** is not drawn, it is necessary to start specific states on entry in the orthogonal regions. It is drawn in the similar kind as join, only with one input and more outputs for each destination state.

The both *Figure 148: Stm/StateM1Parallel-OFB.png* and *Figure 149: Stm/StateM1Parallel-Rhaps.png* shows the same StateMachine with Orthogonal Regions in UML and in OFB, the drawing style is different, the behaviour is the same.

5.11.3.8 Start or InitialPseudoState and DefaultState

Also the top level as any nested level (or region in the OMG document) can or should have a default state. It is the state on initialising the state machine respectively the initial state of a nesting level if the transition goes only to the whole nesting state, not to a specified state.

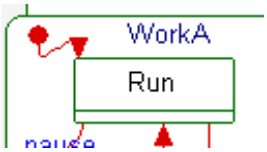
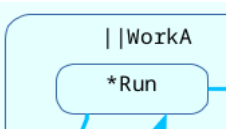


Figure 150:

Stm/InitalPseudoState_Rhaps.png

In UML this is designated by the transition from the initial pseudo state, a small round point, to the default state. This start transition cannot have a condition nor an event as trigger, but can have an action on transition, which is independent of the entry action of the default state, only the last comes if the default state is immediately the destination from the transition from outer.

It is even independent of the entry action of the whole nested state, which comes on any entry from outer. Differentiating this may be important for some applications. See *OMG document 14.2.4.7 Pseudostate and PseudostateKind* page



For the OFB State Machines the extra small point of the initial pseudo state is spared, instead its transition goes from the superior nested state to the default state with style `ofcStateTransDefault`, seen on the left image from Work to Run. This is helpful the other supported draw possibilities, see 25.11.6 Nested State Machine page 9.

5.11.3.9 HistoryPseudoState

Imagine, that the state `Run` in the image is also a complex nested state with some much more sub states. Then the `cont` transition from `Hold` should activate the last state which was left on `pause`. For implementation in the target level this is not a problem, because usual the state in the nested level can be stored in an extra variable or an extra part if a bit mask. It can be reconstructed. For this action, entry to the left state, the History Pseudo State was thought by David Harel and it is defined in the UML standard. But there are two types of History Pseudo State:

* Deep History: Go to the real left state also in deeper levels of nesting.
* Shallow History: Go only to the left state of the first nesting level, and for deeper nesting levels start on the default state.

TODO a diagram and example,

5.11.3.10 FinalPseudoState

The FinalPseudoState has the meaning, a Region is still active, but inside the Region no State is active. This Situation is important for multi threading of Regions, it is the temporary situation for state switching going out of a Region.

5.11.3.11 Execution principles of state machines - and OFB

There are two different principles as state of the art:

a) Typically manually programmed state machines executed in a maybe fast thread or interrupt **only with condition quest**, using **switch-case** constructs.

b) Typically **generated StateMachine** code uses events with an **event queue**.

Of course, manual programmed state machines can be used also events, as well as there are some generating approaches in UML tools for conditional driven state machines.

But independent of details, for both a principle seems to be valid, which, on closer inspection, is incorrect:

* The complete state machine **shall be executed only in one thread** to prevent conflicting state changes.

This principle is usual fulfilled with b), using events and a queue, and also in a) because it is only one thread.

Principles a) and b) seems to be not able to merge. Hence StateMachine using is dispersed in teams and parts of the software. Intrinsically, the whole behaviour should be presented and even designed together, certainly with more as one StateMachines, but with same principles and communication between. The OFB with its graphical programming should close this gap.

Examples are, a fast reaction is necessary to handle emergency situations. This is e.g. in **electrical control an over current situation**, or for **mechanical control a collision**. That should be handled immediately in the current (fast) thread which is often an interrupt execution. If the principle b) is used, then this reaction is programmed often in a manually way outside of b) and hence outside presenting in the (elaborately and complete) state machine description. But it is, or should be an integrated part of the state behaviour for the complete overview for a good software design and documentation.

The solution offered in OFB **is: is:**

* Only each one Region (nested or orthogonal) needs to be executed in one thread, to prevent state switch confusing.
* Other Regions can be executed in other threads. On entry and exit the Regions, synchronisation mechanism are necessary between this threads (between the executing operations).
* State switching in a Region does not need an event queue. It can be done immediately in the called **event operations**. It is fast.
* An event queue is possible if the organisation of execution should be require it. But it is not a basic principle.

This solution allows working with a comprehensive graphical drawn state machine, and the fast execution of determined parts of it. See the example in chapter *5.11.5.5 Fast execution in one thread (interrupt) in a specific region* page *186*

Another question is, **how states are mapped to the data** in memory.

1) The most known solution is: Using one simple integer variable for the current state, and define its values with `#define` constants or better use an `enum` definition. The switch of transition is then often a switch-case statement to test the state, test the event and condition, call the actions in this branch and set the state variable newly.

2) Another solution is: Use a complete description of the State as a class definition, containing possible transition switch conditions (table of events and conditions) and also the actions as virtual overridden operations. This is usual possible in the known object-oriented languages (C++) but also in C, if function pointer are used in a well formed structure for the actions. The effort to write this is higher, but the execution time may be lower because the reference to the current state is immediately given. It may be important that the constant state data are really `const` possible location in the Flash ROM of embedded controllers.

For both, 1) and 2), independent regions of the state machine need to be stored in independent variables. This is the base consideration first for use the history state principle, but also use the independent execution of regions in OFB.

5.11.4 Details of StateMachines in OFB

5.11.4.1 Merge StateMachines with other FBlocks in the module

In UML usual a few classes have a “behaviour” with a StateMachine, and the rest haven’t. Then the StateMachine working is delegated to instances of this classes in a module. But a class with state machine behaviour has also its data and operations – as members of the class and as manual written content of the operation bodies.

A similar situation is given in IEC61499 graphic programming for example with the tool 4diac. There are so named *Basic Function Blocks*, and only these can contain an ***ECC*** *“Execution Control Chart”*, and also data and operations, but not as graphic, only as textual operations

Graphic FBlocks are contained only in a *“Composite Function Block”* Type, and this cannot contain an ECC by itself.

There is no really reason that State Machine behaviour cannot combine with graphical Function Blocks. The ECC approach in the IEC61499 is not be used for OFB. Supposed, an ECC containing function block is possible in OFB but not the general solution.

The event orientation of the design used in the IEC61499 is the ideal idea for implementing state machines merged with FBlocks. See a simple example:

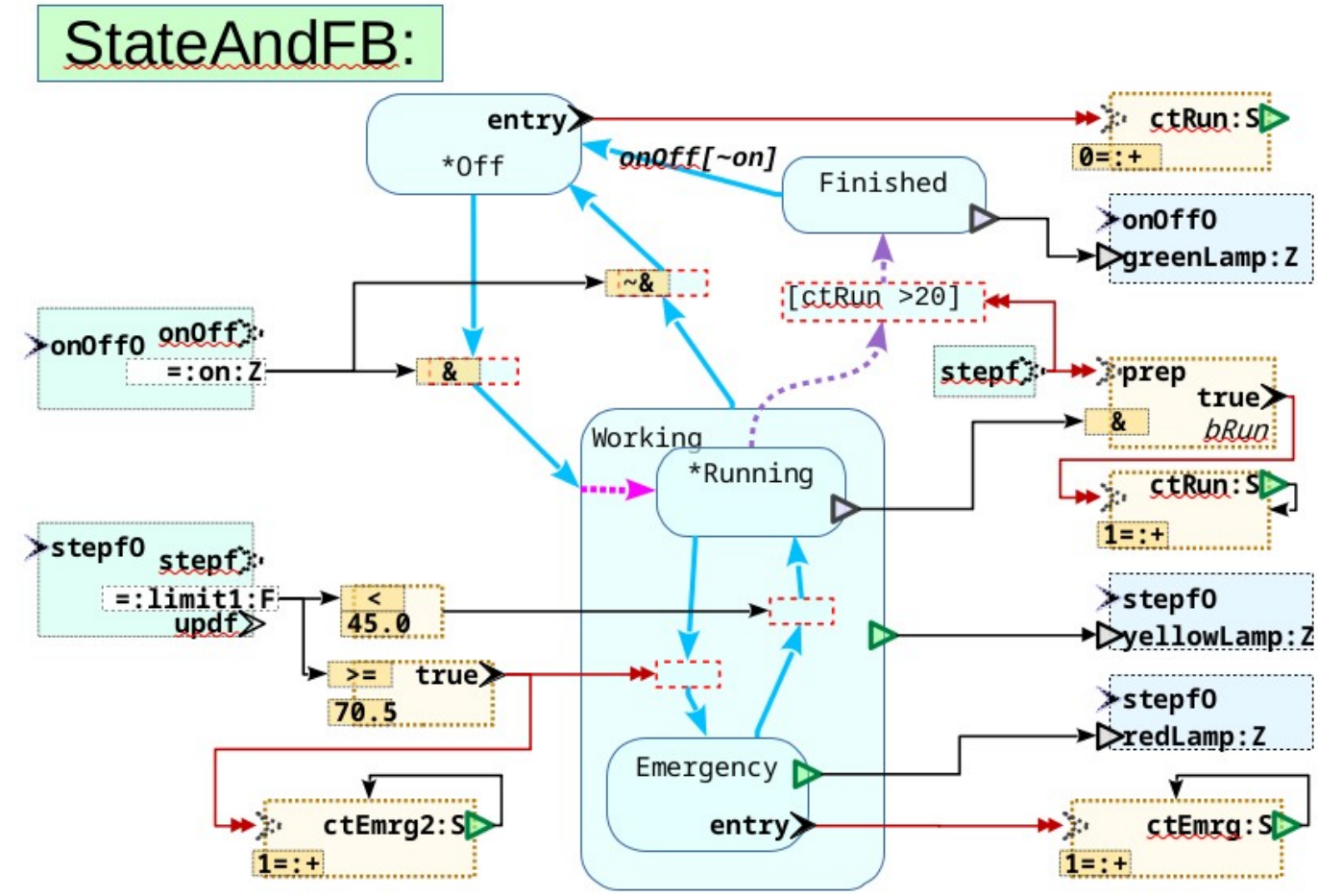


Figure 152: Stm/StateAndFB.png

In OFB a StateMachine as part of a module can be mixed with other graphical FBlocks. The difference is not, that the StateMachine is now applied to a module. Because: A *Basic Function Block* in IEC61499 with ECC can also be seen as a module. As well as a class in UML with a StateMachine can be seen as a module. The real difference is: The FBlock (or module) containing a StateMachine can contain also other FBlocks in graphical presentation. The advantage is, there is not a separation between “behaviour” of a module in specific FBlocks or classes, and the rest of functionality with data flow. Instead, it is a common behaviour, data flow and StateMachines functionality as a whole.

5.11.4.2 Elements in OFB graphic for StateMachines

The OFB example with StateMachines contains a lot of elements, which are now explained using the left side image. The principles and capability are already presented in the main chapter before: *5.11.3 State Machine in Embedded Control, UML and OFB* page *168*

5.11.4.2.1 State, name of the States, default or orthogonal

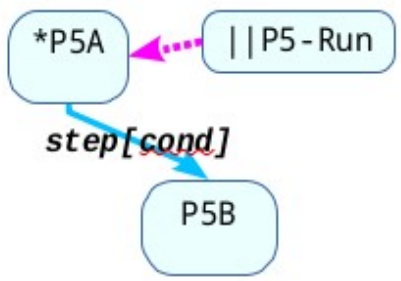


Figure 153: StMParallel-NameStates.png

1) States are **rectangle shapes**, usual drawn with rounded corners. The style has to be `ofbState`. To copy the shape use a given example or select a rectangle with rounded corners from standard shapes. Whereas, the form of the shape is not relevant for translation, but it is for readability of the graphic. The radius of the rounded corners may be necessary to adapt.

2) The **name of a State** is determined by the text of the state shape. It can be placed using the menu *"Format - Text Attributes"* (F3) in the mid of the shape, or on top, or ... it is free. Also the text can be broken in more lines, and can have spaces on start and end (that are *"white spaces"*).

3) The state **name is an identifier** as usual in programming languages, possible containing or also starting with the underline `_`.

4) Before the name an **asterisk** `*` can be written. Then this state is dedicated as ***DefaultState*** of the ***Region***. In the Image above `*P5A` is such an ***DefaultState***.

5) Before the name `||` can be written. Then this state is the ***RegionState*** of one ***OrthogonalRegion***, means parallel state. The UML standard writing style with dashed lines is not used here, because there is often not enough space in the graphic, and the graphic appearance can be designed more freely. The characteristic with `||` is unique.

6) The state text can be continued with the characteristic and name of the parent states, means **name of the RegionState** and also more outer level states, following with `-` after the name or also following with * or || as characteristic of the following parent states, see 4) and 5). This possibility saves graphic effort to define the environment of a state, especially if a part of the StateMachine is presented on other pages, and related to other graphic drawn behaviour of the module.

The left image shows, that the `P5A` and `P5B` is a part of a OrthogonalRegion of the RegionState `P5`, and the parent of this OrthogonalRegions is `Run`. Because of that, the `Run` state may be remembered as important on view to the graphic, and the graphic is understandable. It is a question of context in the whole module.

7) Due to the general rule in OFB, FBlocks can be **presented more as one** time with different **GBlock**s (*graphic blocks*), also the States can be drawn more as one time in the same module. This allows dispersing the StateMachine's diagram over some pages in the module.

See the image on next side, this may be an overview image for the OrthogonalRegions. `||P5` is here drawn in its context as overview, and in *Figure 153: StMParallel-NameStates.png* it is used. In this image it is not necessary to dedicate the P5 state as OrthogonalRegion, because it is defined already in the other *Figure 154: StMParallel-1.png*. But it is helpfully to do so. If confusing dedications are done, an ERROR message is logged and the confused information is not used.

5.11.4.2.2 Nested drawn states and nesting Regions

As already shown in some images before, and also familiar in UML, nested Regions can be draw in a nested kind in the StateMachine's graphic presentation (State chart). Looking on the image *Figure 152: Stm/StateAndFB.png* on page before, `Running` and `Emergency` are nested States of `Working` as the ***Nesting*** or ***CompositeState***. Looking on the image right *Figure 154: StMParallel-1.png*, the state `Run` is the container state for OrthogonalRegions `P1` ... `P5`. The nested drawn OrthogonalRegion states are similar in appearance as the doted line in a OrthogonalRegion container State in UML. Also here inner content of the OrthogonalRegion can be drawn, shown for `P1`.

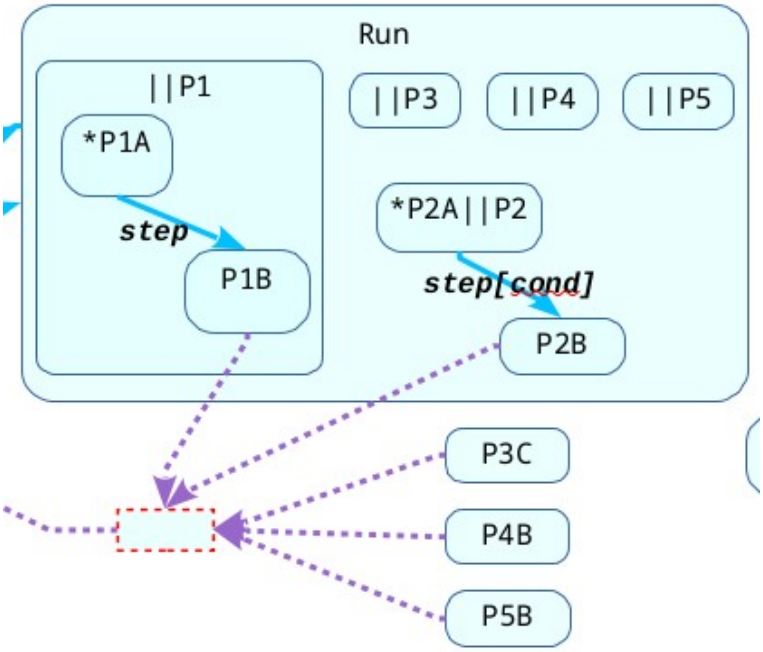


Figure 154: StMParallel-1.png

Following rules are valid:

11) If the states are drawn in a nesting view, as shown here for `P1A` and `P1B` nested in `P1`, then it is a relation between the ***NestingState*** or ***CompositeState*** here `P1`, with its nested inner states here `P1A`, `P1B`.

Note that the nesting draw kind is not necessary to designate the nested situation. It is also possible to use a transition of style `ofcStateTransParent` from the ***NestingState*** to the ***DefaultState*** of the nested Region is sufficient. Which states are part of the nested region is defined by using transitions between of style `ofcStateTrans`.

12) Due to the rule 7) **States can be presented with more as one GBlock**, the whole region of a ***NestingState*** consisting of a lot of States of the Region can be dispersed over some pages. The adhesive region is also detected in the sum of `ofcStateTrans` transitions between the appropriate states, maybe driven more as one.

13) The ***States* affiliated to one Region** with the ***DefaultState*** must only be connected with transitions of style `ofcStateTrans`, that are the light blue transitions in the state images. With this dedication, a Region is well defined. This style for transitions must not use for transitions between Regions, use `ofcStateTransChgRegion` for that.

14) The comprehensive drawn State containing States in a ***Region*** is the ***CompositeState*** for this Region. In the image above it is `P1` for States in the Region `P1A` and `P1B`. If the Region is a OrthogonalRegion, the CompositeState is orthogonal in its parent, as here in the example.

15) The comprehensive drawn State containing only one or more ***CompositeState*** which contains orthogonal (parallel executed) Regions is the ***OrthogonalCompositeState***. This is `Run` in this example. The **OMG UML document** page 208 *14.2.3.4.1 Kinds of States*. does not contain a dedicated designation for that state, it writes only *"A composite State can be either a simple composite State with exactly one Region or an orthogonal State with multiple Regions (isOrthogonal = true)."*

The drawn State `*P2A||P2` presents two states, the state `P2A` which is the ***DefaultState*** of the *Region* with here also shown State `P2B`.

And the State P2 is the ***CompositeState*** which contains this Region. It is one member of ***OrthogonalRegions*** of Run as ***OrthogonalCompositeState***. See also 5.11.2 Terms related to state machine technology page 4

5.11.4.2.3 Transition

20) For **transition lines** three styles are used:

21) `ofcStateTrans`: This are transitions only used between states of the same Region or nested level. All states connected with this kind of transition builds a Region. It is not necessary to draw the Region inside the RegionState, the kind of transition determines the Regions.

22) `ofcStateTransChgRegion`: This are transitions between states of different Regions, means nested states and its outer states, or also via Fork and Join transitions to and from the orthogonal Regions. Note that between States inside different OrthogonalRegions transitions are not allowed. In the image it is used between `Running` and `Finished`.

23) `ofcStateTransParent`: This kind of transition is used from a CompositeState to either the DefaultState of the Region or to its OrthogonalRegionStates (the CompositeState of the OrthogonalRegion content). In the image it is used between `Working` and `Running`. it is not necessary that the RegionState and the DefaultState respectively the whole region should be drawn in nested form in graphic. Also without nesting this transition kind with this style declares the source state as the RegionState and the destination state as the DefaultState of the region.

24) **Transitions** can be drawn either between States, or **to and from an FBlock of style** `ofbStateTrans` from and to the states. This builds a specific FBlock for the transition. Also if a transition goes immediately between states, internally such a FBlock of type >>`StateTrans_FBcl` is built. The **transition FBlock** can have boolean inputs or build a boolean expression used as ***Guard*** or ***Condition*** from the now possible graphical drawn data flow for this transition. Also some specific operations written as `name()` in the transition FBlock text are possible.

25) If Transitions are drawn without an explicitly transition FBlock, then the text on the transition is valid for event and condition. The writing style is the same as usual in UML: `event[guard]`. The **event** should be the identifier of an event in the module. The guard can be either a boolean variable in the module as possible given as input on pins written as @var, or it can be a simple comparison expression with a constant value. This expression is evaluated, and the internal transition FBlock is created in the internal Java operation >>`OdgMdlStates#createTextualConnectionsInStateTrans(...)`

26) If the **transition FBlock has an event input**, then the transition is event driven. But if this event is an inner event in the event chain build with some more logic, then the start event of this chain is the relevant event to trigger. The inner event is only sensible for the sequence of data flow, it occurs only internally following the start event of the chain. In the image left side the start event is the `stepf` on the module's input for the transitions between `Running` and `Emergency`, and the outgoing transition from `Running` to `Finished`. The event `onOff` is the start event for the transitions from and to `off`.

- The transition between `Emergency` and `Running` in the example image left is also event triggered, though an event is not drawn. The event results from the data flow. A transition is only then non event triggered, only then used for **Run to completion**, if it has only data inputs from updated values (from `ofpZout` pins) or a definitely connections with the style `ofcDataGet`. from non `Zout` pins. To better show this situation in graphic, these transition FBlocks can or should have the style `ofbStateTransR2Cpl`. Then it is definitely checked, that the transition FBlock has not a resulting event input by data flow, producing a WARNING if it is not true.

- A Region of states and its transitions is immediately affiliated (associated) to the given event chain, if at least one transitions has not a drawn nor a via text given event input, though they have an event input because of the data flow. That is here between `Emergency` and `Running`. If then different transitions in the same Region are affiliated to different event chains, an `ERROR graphic - more as one event chain for only data flow drawn transition in the same region are not possible` is given, and the generated code may be faulty.

- If this situation of the immediately data flow affiliation between Region and EventChain is not given, then the Region is affiliated to that EventChain which drives the bulk of transitions of this Region. This influences event usage in code generation, not the functionality.

- Also shown in the image page before, states can have **data outputs (Dout)**. The data output of type `ofpZout` presents the state after update, usable for non data flow relevant other data inputs, or just for Run to completion conditions.

- Whereas the Dout of style `ofpVout` or `ofpDout` presents the state immediately after switch the state in data flow order as part of the data flow. It means, if this state is reached by a switching transition, then data using FBlocks are executed after this switch, before further Run to completion switches occurs. It is .

empty

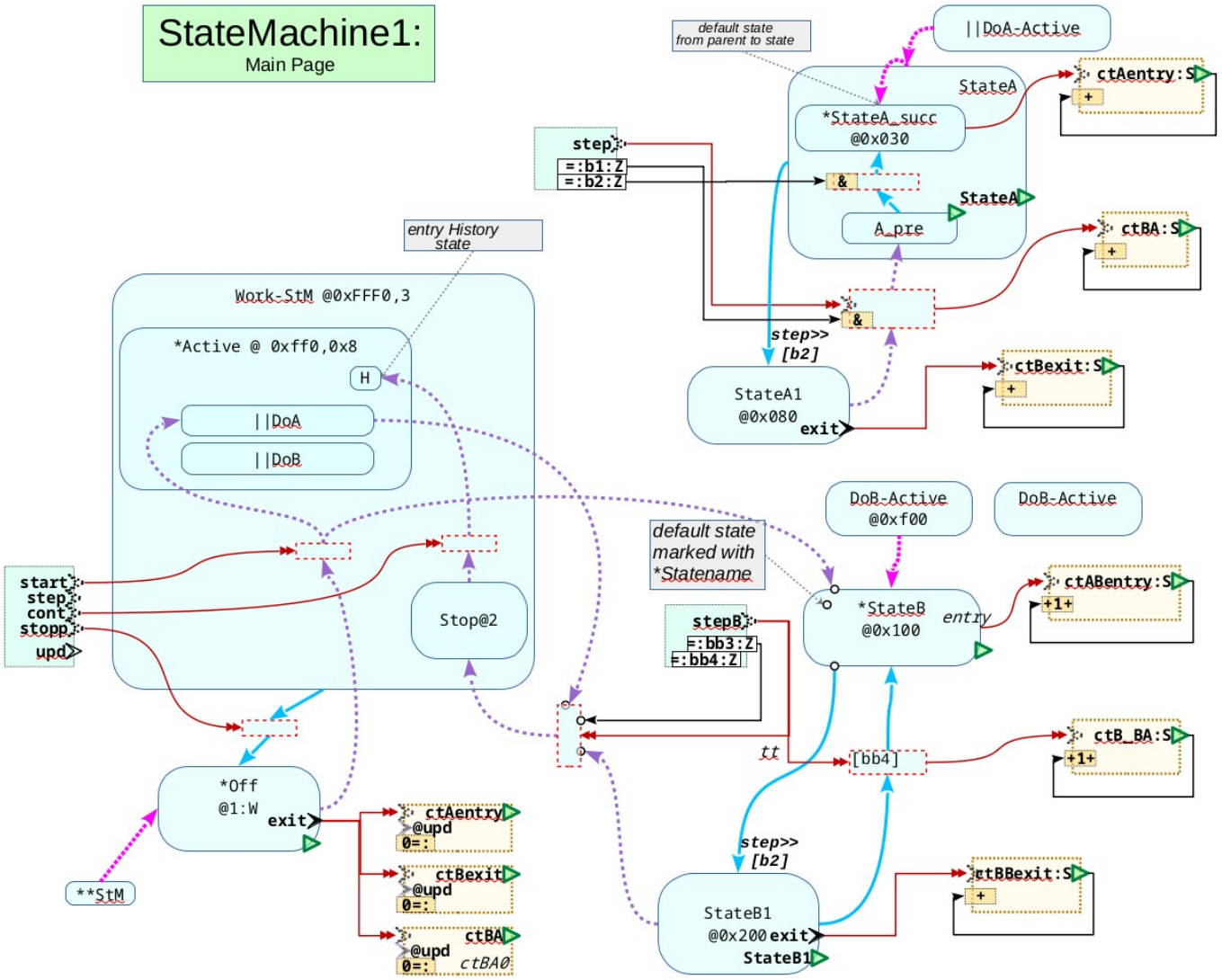


Figure 155: Stm/StateMachine1.png

This is not a working example, only a example which shows a lot of possibilities.

Das war das erste Beispiel, an dem ich mir überlegt habe, was die StateDarstellung alles können sollte. Insbesondere Schachtelung oder nicht, und Namensbildung.

Vielleicht sollte man dies Example nicht bringen und dafür adäquate mehrere kleinere neue. Die das Prinzip z.Bsp. der State-Namensgebung zeigen.

5.11.5 Organisation of execution of the StateMachines

See also *5.11.3.11 Execution principles of state machines - and OFB* page *173* and some fundamental estimations in *5.11.3.2 Manual programmed StateMachines, conditions and events* page *168*

General the execution depends on the kind of code generation, and this is able to adapt by using adapted ***gTxt*** scripts. The base data for the state machine's code generation are contained in one instance per state machine of each module of class >>`ModuleStateM_FBcl`. The state machines (often only one) are referenced by >>`Module_FBcl#stm`. The content of the `module.stm` instance can be used to generate any target source code with this information. If the idea of a repository for all model information are thought, that are the repository entries for each state machine inside a module.

The data of each State Machine as internal Java data of the OFB translation is written in each one file per module: `build/<&Component>/genSrc/FBcl/<&Module>.stm` or also in a file `build/<&Component>/report/<&Module>_StM.html` to view the data content as html page.

5.11.5.1 State variable or classes for states

For really complex state implementations often in C++ (or in Java or other Object Oriented languages) any state can be described by a class which contains all conditions and actions. All state classes have a common base class. All states are represented by instances of this classes, maybe as `const` data (possible stored in Flash). The current state is then a reference variable (pointer) of the type of the common state class. This is also partially known and used by manually programming.

The currently given implementation uses a bit range in one variable array per region of the state machine for regions which are executed in the same thread. Regions executed in different thread uses different elements in this array. The array is not locked as a whole for mutex access, instead the code generation uses only the determined positions.

5.11.5.2 Different threads or devices for each Region of the StateMachine

Another question is: Distributed StateMachines, distributed to several devices on different locations, coupled with Field Bus Communication. Is that conceivable with the StateMachines theory?

The consideration is, Orthogonal Regions works independently.

But if a Join transition should switch, the separated orthogonal Regions are left. This is not independently.

How does a Join transition works?

All Orthogonal Regions should be in the definitely State for join. General, the current state of all devices is interchanged frequently via the Field Bus. If the superior control detects this state situation, then it sends a "leave" request to all Regions, to all the devices. If all devices have left the Region (it is: left its state activity), and all have communicate it, then the superior controller can switch.

But what's happen, if one device reaches the requested State for Join, but the join or leave request to the device does not come. Either because the other devices are not in the Join State, or the Join event in the superior controller has not occurred. The answer of this question depends on the StateMachine's (the Region) programmed behaviour:

a) The Region remains in this reached Join State till the leave event comes, does never more, means waits for ever exclusively for the leave event.

b) Additional to a) any specific event to continue with any other state transition in the Region / device may be possible. But this event comes only from the superior controller, because another situation.

c) The device works furthermore in its Region, and leaves the Join State uncoordinated to the other Regions or devices.

The cases a) and b) are well. In b) the continue event in the Region comes only alternating to the expected join state switch from the superior control.

The case c) is problematically. Because the superior control may have detected the Join state situation for all, sends the leave events to all Regions, but one Region / devices is meanwhile disappeared. Then the other Regions / devices may have received already the event to leave, and this is no more valid.

This situation can be managed by the superior control. It detects that the requested Join state of all is no more valid, and can be send any other event to the Regions / devices to continue. Do not leave for join.

To master this situation, the Regions should be remain in its Join relevant State, understand the event to leave first not for “leave”, instead wait for leave, do not take other activities, exclusively react to the recover-leave event. The leave will be done with a second event which comes after the Join transition was successful for all.

The situation c) is able to prevent with a proven state switch design. If the Join-possible State is reached, either the other Regions / devices can be checked whether they will be ready in their Join State in the next time (for example are active in the State before), or a minimum time should be waited before starting another activity. It is as in the daily life: If you have assigned a worker a task and he has finished it and is bored, he may start something else on his own that is potentially useful and that he cannot be interrupted from so quickly.

With this considerations the problem of dispersed State behaviour over more Regions or devices is solvable.

5.11.5.2.1 Detail: When is a Region active, activate / deactivate it

If a Region has a History entry possibility, then the Region needs a Region-related **boolean variable or one bit** `regionActive` whether it is active or not. This boolean should be set with state switch. The State bits cannot be set to 0, or the pointer set to null if the Region is inactive, because the last active State is necessary for the History entry.

* If the StateMachine works in implementation with state bits, then one of this bits is the Region bit.

```
int regionVariable;
#define mStateRegion_P1 0x0001
#define mState_P1 0x000f
#define kState_initial_P1 0x0003
#define kState_P1A 0x0005
#define kState_P1B 0x0007
```

  The state numbers are all odd, because the last significant bit inside the mask is set as the Region bit.

* If the StateMachine works in implementation with a pointer to the current State of the region, a boolean variable is necessary which is set consistently with a State leave and entry the Region.

If the Region was never used before, the `regionVariable` is `0` or the pointer is `null`.

* If the Region is set first active to the initial State or to a dedicated State, maybe in another device, then the destination State should be set in a dstState variable, and the boolean variable `regionActive := true` as one event action. Set the region active can or should be done in the thread of the superior region, or immediately from field bus communication on dispersed devices. The destination state must not be set Immediately, because in the Region’s own thread first an entry action to the whole Region should be executed. For that the dstState variable is responsible to. The Region’s own thread is informed to get active.

* If the Region was active before and it is now inactive, only the boolean variable for the Region's activity is set from the Region's own thread to `regionActive == false`. The last State is remaining for history entry. This should or need be done by the Region's own Thread. The superior Region's thread (or just field bus communication) detects the `NOT regionActive` Situation and can switch in its own variable backward from the Region State. For a small time the situation is: The Region State is the active one, but inside the Region no state (or just the final PseudoState) is active.

* More simple, on history entry only the boolean variable `regionActive := true` should be set, or more simple, only the bit `mStateRegion...` should be set.

But the problem is, if there are more Orthogonal Regions on dispersed devices, some works, some does not work yet because communication is delayed. What is happen:

If a Region is not active yet, then a definitely State for a Join transition is not set, cannot be detected. Hence it is not a problem, if one Region dawdles a little.

If the OrthCompositeState should be left as such, then all inner Regions should be receive the signal to left its activity (set `regionActive := false` or reset their `mStateRegion` bit. This should and can be done independently of its state or its activity. The current state remains as history.

It means in conclusion, **the Orthogonal-CompositeState can be set as active in the superior Region (on another Device) though not all Regions have started their activities yet.** The OrthogonalCompositeState can be set first independent of feedback from its Regions.

What's happen on deactivating all regions with leaving the OrthogonalCompositeState?

This should be done in two stages:

* First, if the superior control recognises the possible switch, then it sends the "leave" event to all Regions / devices from the relevant Join source State. It means the information which State should be left needs to be a part of the leave event. Only the expected Join State should be left.

* Then it must check and wait till all devices are in the correct "exited Join State" situation. It means the Join State should be detected as currently, but with the `regionActive == false`. or reset `mStateRegion...` bit. Only then it can left its own OrthCompositeState.

* If this condition does not met, the only one possibility is: cancel this execution action of the Join transition, recover all Regions as History recovering. Because not all state conditions for Join were met.

The last one occurs only, if a state behaviour is not friendly designed. It is c) in the chapter before.

5.11.5.3 Merging the idea of event and condition driven state switch

What is the really difference between event and only condition driven StateMachine behaviour:

The condition driven transitions are mentioned in the *"Run To Completion"* paradigm. The primary event does trigger, then all following conditions in transitions outgoing from the reached destination states are executed till nothing more is conditional given. But, if no other event occurs, and meanwhile the condition is changed to true for a possible transition, nothing occurs. The trigger does not come. It means, the state is not left, though the condition to left is given. That is the defined behaviour of the event driven execution.

The condition driven execution seems to be in opposite, the transition is switched because of the condition, without necessity of an event.

But is it so? Are the approaches really in opposite?

The fast control execution is called cyclically. And this call can be interpreted as a cyclically event. It is an event in the common meaning of the term 'event', triggered for example by a timer interrupt. And it can also be considered as an event for the StateMachine. Formally this event is (implicitly) applied to any state, as inner trigger of each state (without exit and entry, do not left the state). Such a transition is defined as ***internal*** due to omg.org chapter *14.2.3.8.1 Transition kinds relative to source.*

And thus, with the new step cycle, and the supposed inner event, the Run to completion is again started with the possible changed condition.

There is no difference, it depends on the interpretation of the situation, it does not need other thinking or implementation.

5.11.5.4 Where is the event queue located, the StateMachine's thread

The IEC61499 as programming environment especially for automation control works intrinsically with events. To show this, the next image is used as example:

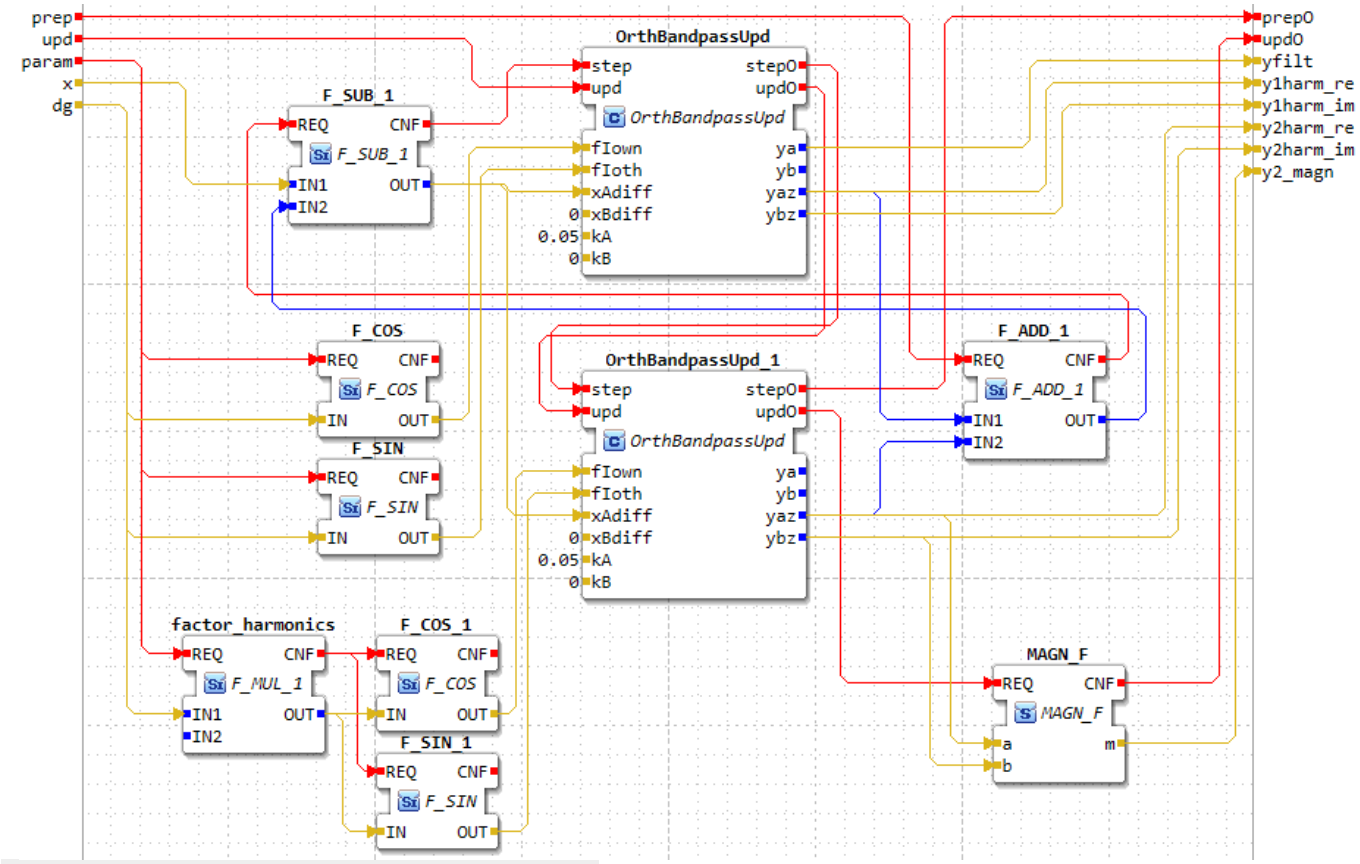


Figure 156: 4diac/OrthBandpassFilterApplUpd.png

The red lines are events. Some FBlocks are simple mathematics, the `OrthBandpass...` are composite, complex FBlocks.

In the original automation control processing the event flow is organized **by event queuing between each FBlock**. It means, on input the `prep` event comes. Because of that, a new event addressing the FBlock `F_ADD_1` It is stored in the module's event queue.

Then, if the first dequeuing is done. The `F_ADD_1` is executed, and it puts the next event for `F_SUB_1` to the queue, and so on. The both `OrthBandpass` can be also executed parallel, means the `F_SUB_1.CNF` can be connected with both `OrthBandpass.step`. Then two events are put in the queue, and if the dequeuing works strong sequential till the FBlock is calculated in the same thread, then both are executed one after another. But here also parallel processing may be possible. Means the execution of `OrthBandpassUpd` and `OrthBandpassUpd_1` is only prepared in this dequeuing thread and both are calculated independently in other threads. But then a so named `REND` event join FBlock is necessary before the output `prepO` event is created and given to out. Drawn it sequential is more simple and usual adequate.

There is an additional possibility also explainable here: Presumed, the both `OrthBandpassUpd` are located **on different devices**. That is the possible concept in IEC61499. Then it means dequeuing of this events for the `OrthBandpassUpd` in the current thread does only **prepare a communication (field bus)**. Then the queue is empty and the **dequeuing thread goes to sleep** because nothing more is to do. Later the other devices have execute their stuff, and send both their `OrthBandpassUpd.stepO` event to the communication via field bus. The receiver put both (independently) in the own queue for the module on this device and **wakes up the dequeuing thread**, and after the maybe possible event join FBlock the `prepO` event of the module is created, or any next FBlock is processed.

If the both `OrthBandpassUpd` contains a StateMachine, they works independent anyhow

if they are dispersed on different devices, and just one after another on the same device, processing its `step` event.

The next question is, **what happens if** all three events of the module **comes uncoordinated**. For example the inner module's event queue has just the `OrthBandpassUpd-step` event in the queue after execution of `F_SUB:1:CNF`, and unconditionally the `param` event comes. Then, after execution `OrthBandpassUpd` in the here shown sequence first the `factor_harmonics` comes between etc. This causes that both `OrthBandpassUpd` works with different values for the parameter. This is supposed not desired.

That's why for this application, it is outside clarified that the `param` event does not come if step is still active. To be precise, this simple example has missing a `param_evout` pin, it is necessary to organize this sequence. And more better: An `updparam` event of the module comes in sequence to `prep`, to takes over the new parameter values after calculation of them. Because the prep event should be executed hard time cyclically.

It means there are outside given sensible rules for the timing of the events.

In IEC61499 StateMachine's execution are separated from the module's graphic FBlock execution. A FBlock which contains a so named ***ECC*** *"Execution Control Chart"*, is a ***Basic FBlock*** and can contain only operations, so named *Algorithms*. Such an ECC can have the following graph:

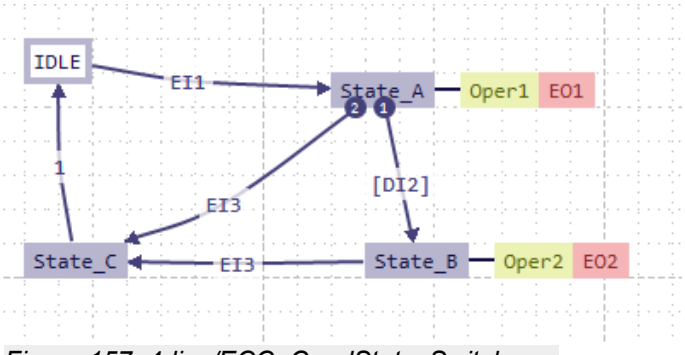


Figure 157: 4diac/ECC_CondStatesSwitch.png

This FBlock has at least three event inputs `EI1..3`. The FBlock seen as module has also one event queue, which is fed from the input events in the outside determined order. The one event queue in the FBlocks inner module data stores the events in this incoming order, and executes one after another the state switches in the only one thread of state execution, also executing the operations (yellow boxes) as entry actions of the states, with creating the output events `EO1, 2` in this operations.

It means there should be only one execution thread for the StateMachine.

If the superior module, which calls this ECC FBlock, is also organized in one thread for the dequeuing of its event queue, and on dequeuing the FBlock with this ECC is called one after another with an implementing operation, which does not save the events again, but executes the StateMachines transition in this three event operations, then also all is proper. Then, for the module with the ECC, it is clarified that the next event is executed only if the execution to the event before is finished.

But it is also possible that the events come from different sources independently in timing. This is thought in focus of a more free designed embedded programming with graphic and OFB.

The conclusion is, **a module in OFB should determine by its design, whether the StateMachine switching events have to come all from one event queue and its dequeuing thread outside, or not**. If yes, then StateMachine switching can be done each in each event operation.

If this is not guaranteed, then one extra event for the state machine is necessary. It **has the name** `evStm` and `evStmO`. The other input events should put the event for StateMachine switching in the inner queue of the module, which gives an information to activate the StateMachine switch operation by the underlying operation system. Then it can do any other work determined by the event chain with the other FBlocks, non influence the state machine, but using its state.

It is also a decision of the user design, whether in the input event operations things are executed, which are dependending from consistent states (StateMachine should not switch between) or if this things are delegated to the StateMachine's thread.

5.11.5.5 Fast execution in one thread (interrupt) in a specific region

A region is due to definition in omg.org all states and transitions inside a nested state, especially in orthogonal (parallel) nested states, and also the top level region.

The StateMachine definition in omg.org does not presume that state switching should be so fast as possible. In omg.org chapter *14.2.3.9.1 The run-to-completion paradigm* the following text is written on end:

IMPLEMENTATION NOTE. Run-to-completion is often mistakenly interpreted as implying that an executing StateMachine cannot be interrupted, which, of course would lead to priority inversion issues in some time-sensitive systems. However, this is not the case; in a given implementation a thread executing a StateMachine step can be suspended, allowing higher-priority threads to run, and, once it is allocated processor time again by the underlying thread scheduler, it can safely resume its execution and complete its event processing.

It is only important, that during Run-to-completion the appropriated state situation where run-to-completion works is not changed. How many time it needs, and which thread does the execution, is not a topic of the state machine execution itself, it is a user's topic (how fast should be the control as a whole). Also if during Run to completion volatile conditions are changed, it should not be a problem.

Should run-to-complete be executed in only one non interruptable process for the whole state machine?

Consideration: Orthogonal regions are working anyway independent. **Another region as the own one should not switch before switching in the own (inner) region is finished**. It means, if it is ensured, that first only the inner region is executed, and only after them the inside not used event is applied to the outer regions, or Run to completion to the outer region is done, then it is possible that the regions can be handled in different threads or just in the interrupt for the fast execution and in other threads. If the inner region is the fast interrupt, then it is ensured. If the inner region is any other thread, and it can be interrupted by another thread with also state switch, it is bad.

The topic of dispersion execution of a state machine in a very fast thread (hardware interrupt) and some slower threads the following example should be used, from the `OFB_Presentation/src/StatePosCtrl`:

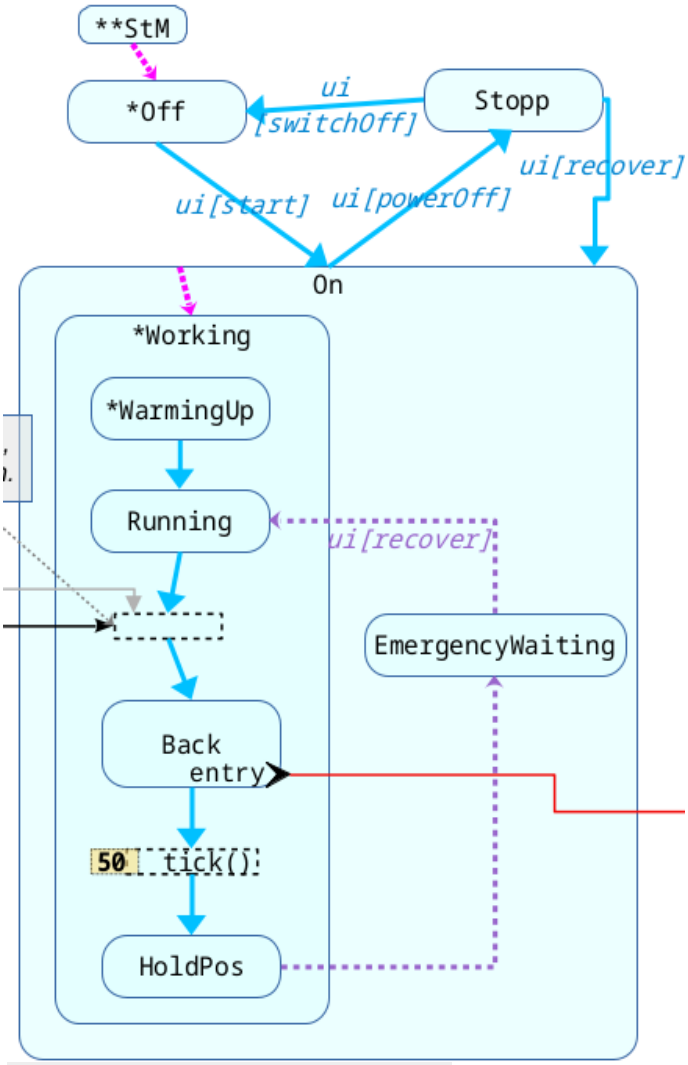


Figure 158: PositionCtrl_StMemg.png

The idea for this example is: On a position control system (robot) an emergency situation (collision or such) occurs, which switches in the fast positioning thread from `Running` to `Back`. This should make a fast back movement but only for a short time. After 50 ticks of this fast control step time (maybe 1 ms tick, 50 ms), the back movement is finished, and `HoldPos` is activated, all movements are stopped. This StateMachine can be more complex, for example also a fast recovering of working can be done, if more inputs are given. But if the situation is not clarified, the stop working or `HoldPos` is settled. It can be only recovered by a manual `ui` = *"user interface"* handling. But this is not done in the fast interrupt, no time for such slower stuff. The `ui` event is processed in

any other normal thread, may be in the back (main-) loop of a simple controller system.

5.11.5.5.1 Affiliation of the fast region to one event chain

If the OFB graphic contains transition inputs in the data flow drawn without event to the transition, then the appropriate region of the StateMachine is affiliated (assigned) to this specific event chain. This is shown in the following image. The gray drawn event input from `prepO` of the expression which delivers the condition (here switch, if the input `im >=70`) is drawn with the style `ofcDisabled`, not recognised as event input to the transition. But the output of the `imaxdetect` FBlock is in the ***data flow*** in this event chain. This is the information to affiliate this region to the event chain with the start event `step`, which is called as step operation in the fast controller interrupt.

With this affiliation, the StateMachine switches in this region immediately in the execution flow.

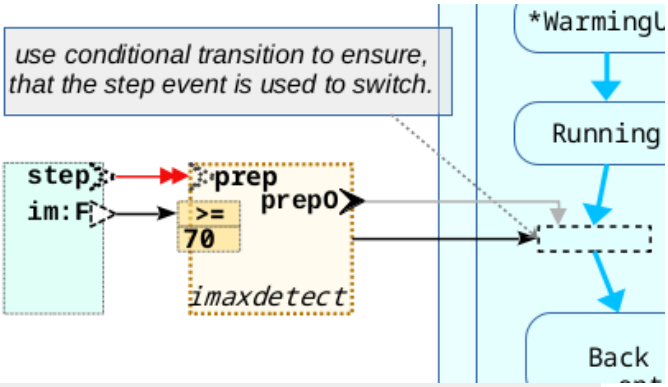


Figure 159: PositionCtrl_StMemg-CondEvchain.png

It means, if the condition `imaxdetect` occurs, immediately the new state `Back` is active, also exit, trans and entry actions are done, and the following algorithm in the event flow is based on the new StateMachine situation. It means the emergency handling is immediately, in the same step time. And this is necessary.

5.11.5.5.2 Further nested regions within the fast region

But what about a nuance. Supposed, the `Running` state is really a composite (Nesting) state with the following content:

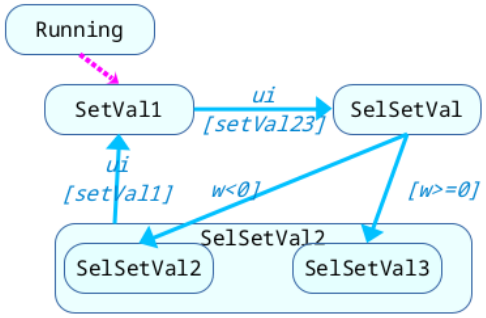


Figure 160: PositionCtrl_StMemg-RunningRegion.png

Some set values should be switched here due to a `ui` user interface command. This comes from a communication, and it is slower maybe also with effort for preparation.

If this is done in the slower thread, but the values (the states, prepared values in entry or transition actions) are used in the fast interrupt, there is a consistence problem. Also if the StateMachine for this sub region is more complex, it has partially switched for Run to completion, and then the interrupt comes and will be switch in its outer region, confusing may be occur.

The only one and simple solution is: If One region is assigned to an extra fast execution, also all inner regions are affiliated fort that.

It means, it is in the user's responsibility. If this shown Region is localised as inner region in the fast thread region, then its code is executed also in the step operation for the fast interrupt.

But there is another way, in OFB with very simple exchange the RegionState (the parent of the region, The parent state is changed from `Running` to `On`:

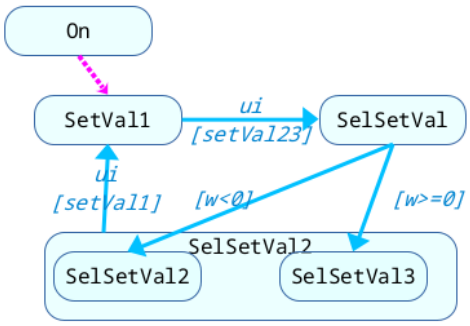


Figure 161: PositionCtrl_StMemg-ParallelSetVal.png

Now, this region is an orthogonal region in the state `On`, because `On` has now two default state, `Working` and `SetVal1`. The set value selection is now done in another tread, with more time as in the fast interrupt. But now, of course, this may need thread synchronization for the data

to get it **consistent**. It is done with the update concept, means `upd` alternating with `step` transports the data for set values to the updated values. The `upd` is called after `step` in the fast interrupt, but it copies only some values, after they are determined as consistent in the slow thread. this is a fast operation.

As conclusion the following rule is valid for OFB StateMachines:

The OFB StateMachines can have some Regions, which are cohesive inner Regions, which are affiliated to one specific **EventChain**, which is intended to be **fast executed**. This EventChain may be executed in a high prior thread or just in a cyclically (timer) interrupt in Embedded Control.

There can be **more as one such fast executed Regions** each in one EventChain, respectively fast thread/interrupt. This regions should be independent, as orthogonal Regions each other.

All other, not fast Regions in the StateMachine are executed in maybe several operations but called in only one thread, named **StateMachineSwitchThread**. This thread must never interrupt the fast executed EventChain.

There are synchronisation mechanism between both threads, to control State switching in the correct order.

5.11.5.5.3 Switch between fast and StateMachineSwitchThread

Back to the example Figure 141: PositionCtrl_StMemg.png page 21: It means the transition from `HoldPos` to `EmergencyWaiting` leaves the fast region, the `Working` region state goes to its finish pseudo state and the `ui` related region (operation) is informed. This performs the switch to `EmergencyWaiting` in the thread of this other region.

If the `ui` event comes with recover, then this action is done vice back to `Running`. First in the `ui` region the `EmergencyWaiting` state is left, and Working is reactivated, but remains in the first moment in the start pseudo state. But the information, “go to `Running`" (and not the default transition) is given.

The fast interrupt for positioning is still active, but does nothing (because the state `Working` was deactivated). If the information comes “go to `Running`”, then it finishes the state switch from the start pseudo state now to `Running`. The output values for motors etc. are yet given, and the robot works again, presumed with sensible set values for positioning.

If the fast execution is oriented to one or some adhesive regions, this regions can be executed immediately in the fastest thread (timer interrupt). If the transition leaves this adhesive regions, it is possible that first the states inside this regions go to the finish pseudo state. It means no state in the region is active, but the region state itself, which builds internally the region, remains yet active. The state switch is hence finished in this fast thread. The fast thread can no more change its region, because it is in the finish pseudo state and cannot be recovered in this state.

This finishing state switch creates an information (bit or boolean or event in the queue), which is communicate to the other maybe slower thread or also maybe just on another device, responsible to the outer region. The outer region contains the second part of the transition, here leave the region state `Working` and switch to `EmergencyWainting`.

In this scenario never an inconsistent state occurs.

What about the `ui[powerOff]`. That is not ultimately clarified in this simple example. Because switching power off during any movement may be not a good idea. There are physical storages, battery power, mechanical kinetics to considerate etc. It means the signal of power off should be received first in the fast interrupt to switch to a secure state (seen from the robot), means there are some more states to execute in the StateMachine in the `Working` region. Then only this state can be left. For the StateMachine it needs a specific event or signal for the Working region, and only if a state similar `Holdpos` is reached, then the region can be deactivated to go in the `Stopp` state in the `ui` related region. It is similar as the `Back->Holdpos` solution. But this more concise solution goes beyond the scope of this documentation.

How this is generated in code? For that some more considerations are necessary, see next chapters.

Another question is related also to this: What about, if StateMachines are dispersed over some hardware devices (with enough fast communication together), but seen as one StateMachine in the design. Then also the idea is helpfully, that regions should be independent. One region can be located on one device, the other region, orthogonal or in another state, can be located on another device. If the state switching does not left the independent regions, parallel (orthogonal) regions works independent, and for non orthogonal regions the following is valid: Only one region is active, the others cannot switch, because the region state (maybe on another device) is not set.

5.11.5.6 Detection and build affiliations of regions and event chains

This is now a question of the algorithm of the OFB translation. The affiliation of regions to event chains should be gathered from graphic (LibreOffice draw) information and first stored in Java data and outputted and reread from the intermediate files (fbd). In the form of Java data it is able to access from the code generation scripts (gTxt), and so it is generated in the executable target code C/++ or other) ready to compile.

First have a look on a part of output of the `.stm` File (shortened):

```
====Regions of StM: ==
====ui ix=0
  - CrossTransition EvChain=0 step ix=1
    trans_HoldPos_Working_On__EmergencyWai...
  ==SetVal1_On_Region default=SetVal1_On
    * SetVal1_On
    * SelSetVal
  ==StM default=Off_StM
    * Off_StM
    * On
====step ix=1
  - CrossTransition EvChain=8 ui ix=0
    trans_EmergencyWaiting_On__Running_....
  ==Working_On default=WarmingUp_Working_On
    * WarmingUp_Working_On
    * Running_Working_On
```

This log or report output is a mapping to the Java internal data for code generation. It is shortened for explanation, but related to the used example of Position Control with collision. It shows two RegionsEvchain, one for the `ui` event input, the other for `step`, the fast thread.

The Cross Transitions are shown, the regions with its default state are shown, and states (not all shown here) and transitions (not shown here). The data itself are stored in container, usable with `<:for:...` and `<&..` access in code generation.

This information is built in

```
>>CreateStmRegion_FBcl#prcRegionAffiliationToEvchain()
```

To solve the challenge to combine fast StateMachine switching in an interrupt with slower execution of other parts, the following considerations are necessary.

The classic UML StateMachine can have any different events, triggering the transitions. All these events are put in an event queue on its occurrence, and read out in the only one StateMachine's execution thread.

But in the OFB graphic using the IEC61499 event driven approach between Function Blocks, events are anyway present. Each FBlock can drive an event for state machine switching from its `evout` pin. **Other than** in implementations of IEC61499 **for automaton device processing**, for execution in a normal statement execution order (a function in C) **without using the event queue between each of the FBlocks**, an ***event chain*** is built which defines the order of statement of FBlock execution. The event chain has one **start event** on its input, it is either coming as input of the module from outer (it is an `evout` seen from the inner of the module), or it is emitted from any `evout` of the state machine itself or from another StateMachine in the same module.

The typical use case for fast execution is a conditional drawn StateMachine switching, with conditions on the appropriate transitions. The conditions are built in this fast thread, following the **start event of this event chain**.

With a side glance to the "condition triggered" part of the state machine for one or affiliated regions, the event used for trigger can be, or is often the same for all transitions of the region. For this situation the appropriate regions can be affiliated to this one event chain. The transitions can be triggered either from values (conditions) from signals built in the event chain itself, local values in the execution order or data flow. This values have an appropriate event, which is a consecutive event in the own event chain. This is the trigger for this transition. Or the other possibility, the transitions have not an event trigger, are really conditional, switching with the ***Run to completion*** approach. Then they have input signals, not coming from data flow relevant outputs, especially `Zout` variables. They are consistently set with an update event in other threads or in the same event chain execution (for example the fast controlling interrupt) from the step time before.

If the OFB graphic contains transition inputs in the data flow drawn without event to the transition, then the appropriate region of the StateMachine is affiliated (assigned) to this specific event chain. This is shown in the following image. The gray drawn event input from prepO of the expression which delivers

the condition (here switch, if the input im >=70) is drawn with the style ofcDisabled, not recognized as event input to the transition. But the output of the imaxdetect FBlock is in the data flow in this event chain. This is the information to affiliate this region to the event chain with the start event step, which is called as step operation in the fast controller interrupt.

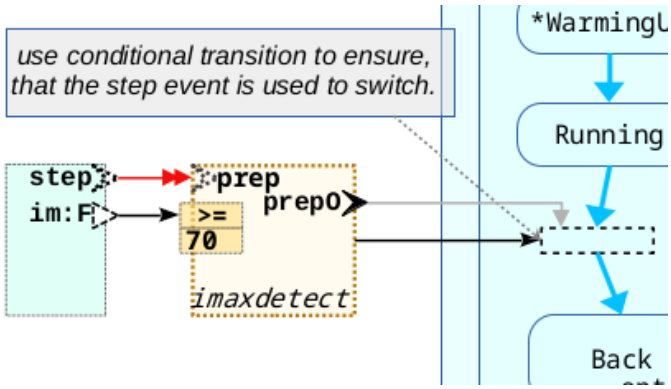


Figure 142: PositionCtrl_StMemg-CondEvchain.png

If the region's transitions have other really event triggered transitions, then the execution of this event chain containing the state execution need an event queue. The event queue is built in generated code with the filling operation, which selects only the appropriate event for this region(s). Then the execution of the step event chain operation starts with enqueue. Events which are not consumed are put forward in the other queues for other regions.

For the fast controlling thread it may be better to abstain from other events, to reduce calculation time, more using conditions accessible as data set from other threads. But this is a particular design decision.

If a region cannot be assigned to a deterministic event chain, because all transitions are either driven immediately by events, or driven by conditions not in the data flow (`Zout`, updated values), then the region is affiliated to that event chain which occurs most frequently.

In the example in image *Figure 158: PositionCtrl_StMemg.png*, page before, only the `ui` event is used.

Then the state execution is affiliated to this event, here `ui`. It means the state transitions can be executed all in the `ui` event chain operation by testing the particular guards.

But, if the event is not applied to any transition, then it is possible, that the event is need anywhere other in the whole state machine, in other event chains or in other event queues. It means, the event chain operation for this specific event, here `ui`, have a return information about using the event.

If other events are additional used, also here a queue is filled outside and enqueued in this event chain operation. Also here, if the event from the queue cannot be applied, it should be sent forward to the next queue for other regions. Whereby the order of regions, first the inner then the outer ones is regarded in decision which queue is used first.

5.11.5.7 The event pool for events to queue

I've had a bad experience: Events were generally allocated with `new` in C++ on necessity. Intrinsic, using dynamic data are sometimes forbidden for embedded software in runtime, but that was a given framework, outside of rules to considerate. And really, one time it was occurred, that the execution hangs, the memory was out, hardly to debug, and a queue was filled with each the same event. There was a very simple reason: The thread for enqueue had hang because any incidental software error.

Intrinsic, there are not the necessity of often only one event of a dedicated usage. Sometimes more event may be necessity for repeated sent events, but if that is more as 2..3, it indicates another software error.

On the other hand, there are not too much events necessary for all usage situation, maybe it is 100 for a comprehensive solution, with each 10..20 byte for data. 2 kByte is not a relevant size (for a comprehensive solution), and 10 event occurrences, 100 Byte is also not relevant for a small solution with a TMS320 controller with 0.5 kByte on chip.

That's why it is better to use a dedicated pool for each one event for each using. For the example on the pages before, only the `ui` event is given, coming supposed by a receiver driver from an *"Operation and Monitoring" (OaM)* devise. If the controller does not react to this event in a definitely time the communication has a not to fast minimal time (the human's operation velocity), then the not enqueued `ui` event can be simple removed from queue, and reused for the new *OaM* request. Use always the last, it is the last desire of the human, maybe to abort the process.

But the pool of necessary events should also automatically generated from the OFB graphic and that is still todo.

5.11.5.8 Data structure of an event, the event queue

The following are basic considerations about the event and its data, and the event queue.

General, the data which are related or appropriated to the event should be stored in an independent data structure to the event. This assures that the data are consistent with creation of the event till its consumption. If data are referenced, then it is sometimes possible, respectively it needs care about, that the data are not changed during the event is in queue. Furthermore, if events are transported, sent and received to and from other devices, the data consistence should be clarified. The data of the event from its own `struct` should be copied and load to and from the payload of communication telegrams.

In IEC61499 anyway it is handled with events. During the execution of one event chain operation as sequence of statements, the data are located in normal variable, in stack or in instance variable, as in normal sequential programming But for events, which leaves the focus of one execution thread, they should be copied. Which data, is related to the principles of the IEC61499. It means, the data structures for events can be code generated from the OFB graphic content.

General an event should have a basic `struct` on begin, which allows casting. A possible event definition (used in code generation from OFB) is:

```
typedef struct Event_OFB_emC_T {
  int32 id;
  int32 idDst;
} Event_OFB_emC_s;
```

And then a user defined event:

```
typedef struct EventuserXY_T {
  Event_OFB_emC_s base;
  float value1;
  int32 value2;
} EventuserXY_s;
```

A simple event queue can be built with two indices for read and write and a known size. Such an event queue has the following basic structure:

```
typedef struct EventQueue_OFB_T {
  uint16 nrofEntries;
  int16 volatile ctModify;
  int16 volatile ixRd, ixWr;;
} EventQueue_OFB_s;
```

xxx

The data buffer for the events is defined in an extra pool, or immediately after this structure. The buffer contains only the pointer to the `Event_OFB_emC_s*`, because the event instances have a different length.

On writing entries the `ixWr` is incremented and wraps on the end to 0. The buffer is filled, if the value of `ixRd` would be reached. The size of the event queue is limited on creation of it. For set the new value of `ixWr` concurrently in several threads the compare and swap approach should be used, or a simple interrupt disable. i`xRd` is also incremented after read out, if it is not equal `ixWr`. Because enqueue is always done in only one thread, compare and swap does not need to be used.

The really usable C sources are contained in the emC pool, see TODO not proper documented in the past ...

5.11.5.9 Code generation for StateMachines

With the last considerations in background, the following principles are valid and done:

a) StateMachine execution is an integrated part of the operations for the event chain.

b) On start of the event chain execution first cross transitions should be executed, which comes and are started in another region. For each this cross transitions, one bit is sufficient to inform about its necessity.

c) Following, the event queue should be enqueued for events, which are not part of the own event chain. For each enqueued event a switch-case of possible states and then check of its transitions should be done. Instead switch-case it is also possible to have each one state referred. Then its trans operation is called. If the event is not consumed, it should be taken in a possible following queue, which is part of the events organization data.

d) Then all ordinary Function Block operations should be executed in the data and event flow of the event chain operation.

e) If the data flow reaches a transition, then this transition should be immediately switch. As result a state change is expected, but first only as current state, not as updated state. It means it can be used as information for following processing with the `d` output on a state. The `q` output is not changed first.

f) If the same data flow reaches another transition too, that is in order immediately one after another, then only the first of this transitions is handled. Because it is only one event which should be consumed.

g) It is noticed in a local boolean variable, that at least one state transition is executed with the start event of the event chain.

h) What about another inner event in the data flow originally derived from the start event of the event chain. Is it a new event? But there is only one initializing event. Maybe this is a theoretical question because the design in graphic for one module should avoid too complicated situations. It is more simple to prevent using this second event by evaluation of the information, start event of the chain has triggered. Then the

Run to completion can be done at least on end of the event chains operation. Elsewhere Run to complete should be exexuted after each state switching immediately following. Which has also some advantages.

i) At last and at least one time the conditional transitions of this regions should be processed. That is the Run to complete as well as the conditional transition execution after the intrinsically inner transition because of the start event of the chain, as explained in *5.11.5.3 Merging the idea of event and condition driven state switch* page *182*.

j) On any state switching the information about cross transitions for the other event chain should be given if necessary.

The first decision is, using switch-case for state request, or using a pointer to the current state's data. Both is possible. it changes only the core, the state switch itself. The next examples uses the more known switch-case approach.

```
// Process state machine cross transition to this evchain:
if( thiz->stmCrossTrans_step_ui & (1<< kBitCROSStrans_EmergencyWaiting_On__Running_Work...)) {
  int catastrophicAbort = 10;
  int16 valMem = thiz->stmCrossTrans_step_ui;
  int16 valMemCmp;
  do {  // use compare and swap mechanism to reset this bit because prevent conflicts if ...
      valMemCmp = valMem;  // use last value to compare success
      int16 valSet = valMem & ~(1<< kBitCROSStrans_EmergencyWai...);  // only reset this bit
      valMem = compareAndSwap_AtomicUint16(&thiz->stmCrossTrans_step_ui, valMemCmp, valSet);
  } while(valMem != valMemCmp && --catastrophicAbort >0);  // repeat if write is not succes...
  ASSERT(catastrophicAbort >0);
  // executes switch state for this region, forced in regionsEvchain: ui
  thiz->stateStM[0] = thiz->stateStM[0] & ~mState_Working_On | kState_Running_Working_On;
}
```

Doe do-while compareAndSwap will ich durch ein Makro ersetzen, gleiche Wirkung im Code aber weniger zu lesen.

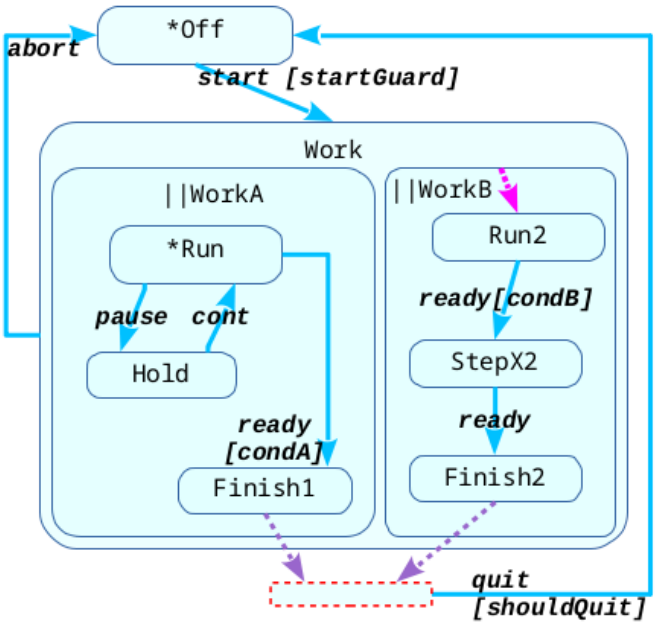
This is an example for b). Only a simple bit is used and quest to inform about the necessity of complete a cross transition. The reset of this bit is a little bit complicated. because, the new value for the whole memory word is written after read this, and in this very small time between reading and writing, the content on RAM may be changed. Another bit is set (for another cross transition), and this bit set is lost.

xxx

The `compareAndSwap` approach ensures, that a value is only then written to RAM if the expected value is there. See https://vishia.org/emc_/html/Base/Atomic_emC.html. The more simple solution may be a dedicated interrupt disable.

5.11.5.10 When an outgoing transition should switch

There are different cases. Look on a parallel state machine:



Stm/StateM1Parallel-OFB.png

Here a complex case is shown, a Join transition. The both parallel state regions are persist in the both `Finish` states. There is no reason to leave this states, if the event `quit` does not come. But this event is associated to the parent region.

On the next image right side the situation is another one. The state `Running` will be left by a condition which is related to the `stepf` event localized also in its own region. it means this state should be left immediately in this region. But the destination state `Finished` cannot be set, because its data are localized in another thread, because it is another region.

What is the situation in detail view of the eyes from the UML state machine's definition:

The comprehensive state `Working` remains active in the first moment where the state `Running` is left. But which state is then active inside the region `Working`? It is the not drawn implicit end state, the ***terminate pseudo state*** in terms of omg.org.

xxx

xxx

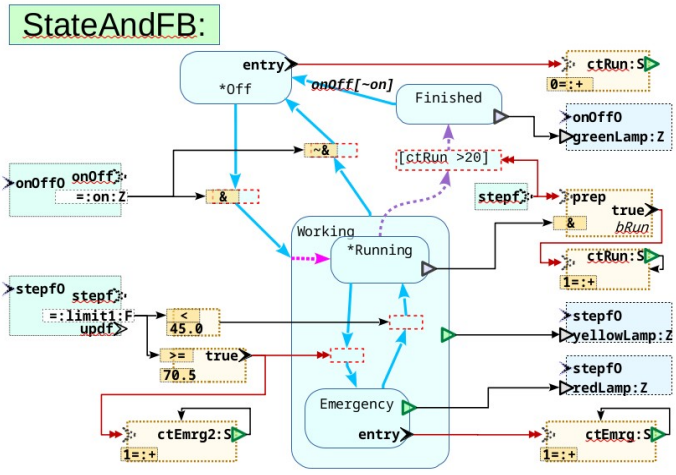


Figure 164: Stm/StateAndFB-OFB.png

With this description the whole state machine is determined though in the first time only the region of the state `Working` switches only from `Running` to the *terminate pseudo state*.

To complete the requested behaviour, now, the region of the whole state machine where `Working` and `Finished` are member of should get the information to switch from `Working` to `Finished`. This state transition is executed in the main operation (or thread) of this region, or more exact, for the >>`StM_Regions4EvChain_FBcl` instance, which combines more regions processed with the same event. The executin can be controlled by a specific event (if only an event queue is used), or just simple on start of the appropriate event operation if a specific bit is set. Last one is implemented in the code generation for Embedded C.

The state execution data for C(++) language contains one int variable with bits for all state switches between this two regions, or if necessary at least one variable for each inter-region switches. If there are 10

5.12 Execution order, Event and Data flow, Event chains and states

5.12.1 Event and Data flow

As also explained in chapter *5.6.2 GBlocks for each one function, data – event association*page *100*, events are associated to the data. In chapter *4.4 Event-Based Execution* on page *22*it is basically explained that events are used as execution control, instead of a sample time association of data pins. Then intrinsically the event flow or chain is responsible to the execution order. That is also defined in the IEC61499 norm.

Using the tools originally for IEC61499 automation control diagrams (4diac, see https://eclipse.def/4diac/), the event flow should be shown in the diagram. The next image shows a part of the used example in this chapters in 4diac:

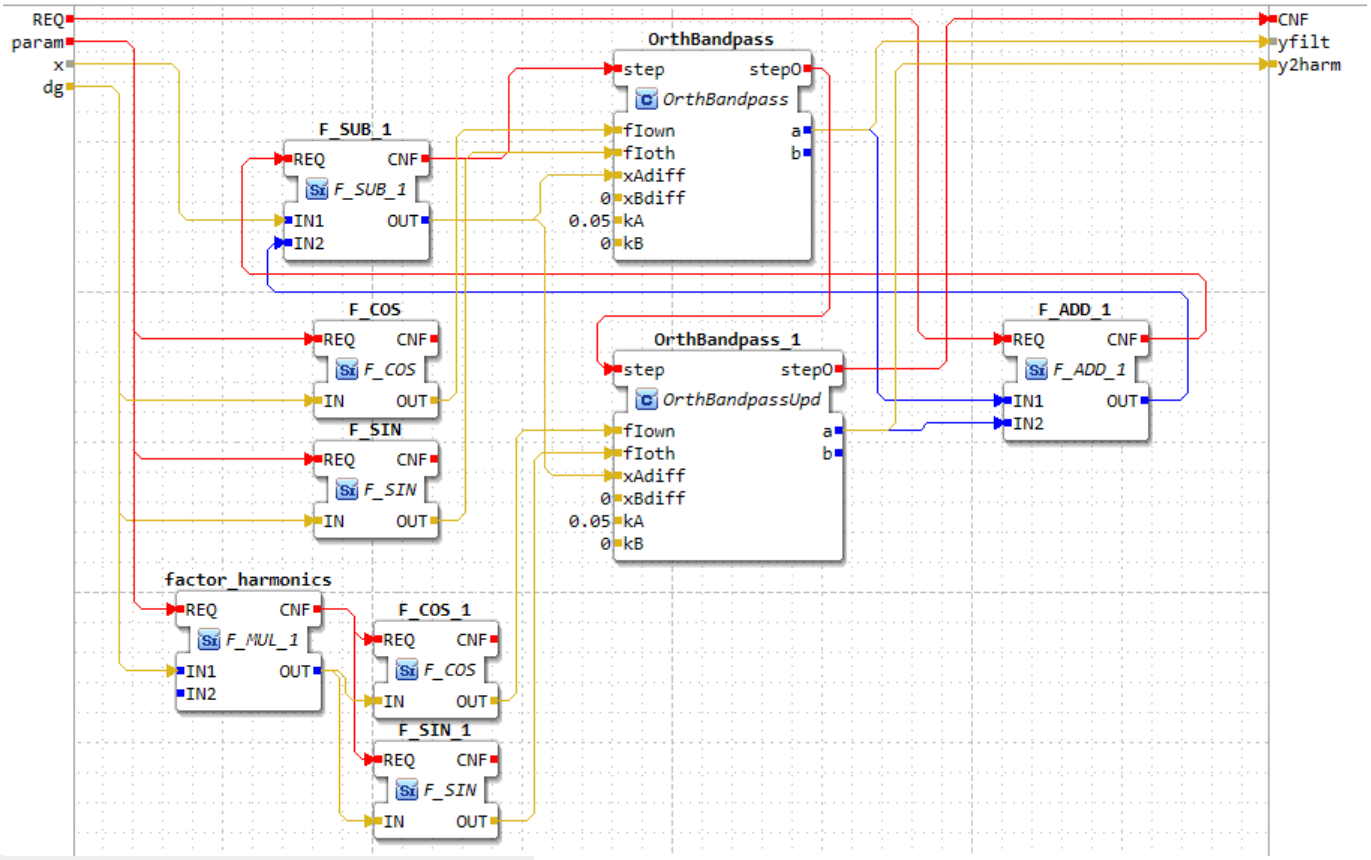


Figure 165: 4diac/OrthBandpassFilterAppl.png

The red connections are the event flow, the brown ones are data flow. The execution order depends only from the events. Here you see first the right `F_ADD_1` is executed, because firstly the outputs of the last step time should be added, then subtract from the x input in the `F_SUB_1` etc. The events should be wired manually thinking on the correct data flow. The data connections are only an information, from where get the data. But the association between data and event are also given here. The step event on the `OrthBandpass` is associated to the data `xAdiff`, `xBdiff` etc. The data are used if the input event comes, and the data are provided with the output event.

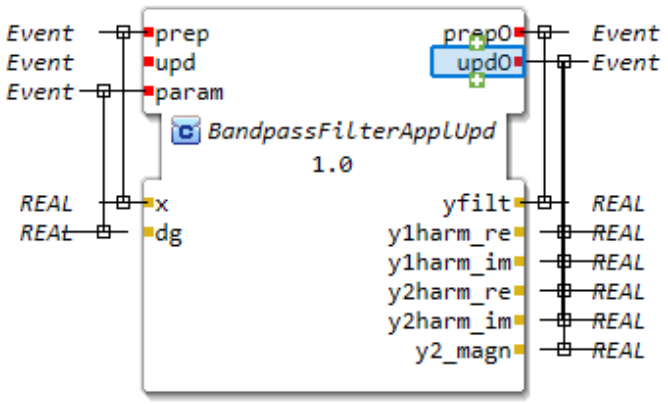


Figure 166: 4diac/OrthBandpassFilterApplUpd_ifc.png

The above shows the interface specification In 4diac for the module. You see all inputs and

output of the module, and the event-data association. The data pin `x` is associated to the event input `REQ`.

But, drawing also the event connections beside the data are a higher effort for the diagrams. If the data flow can be unique mapped to the event flow (as also mapped to the execution order in a given sample time in other FBlock tools such as Simulink), then the effort for draw is lower, and the diagrams are more related to familiar FBlock diagrams. Exact this is done in the OFB.

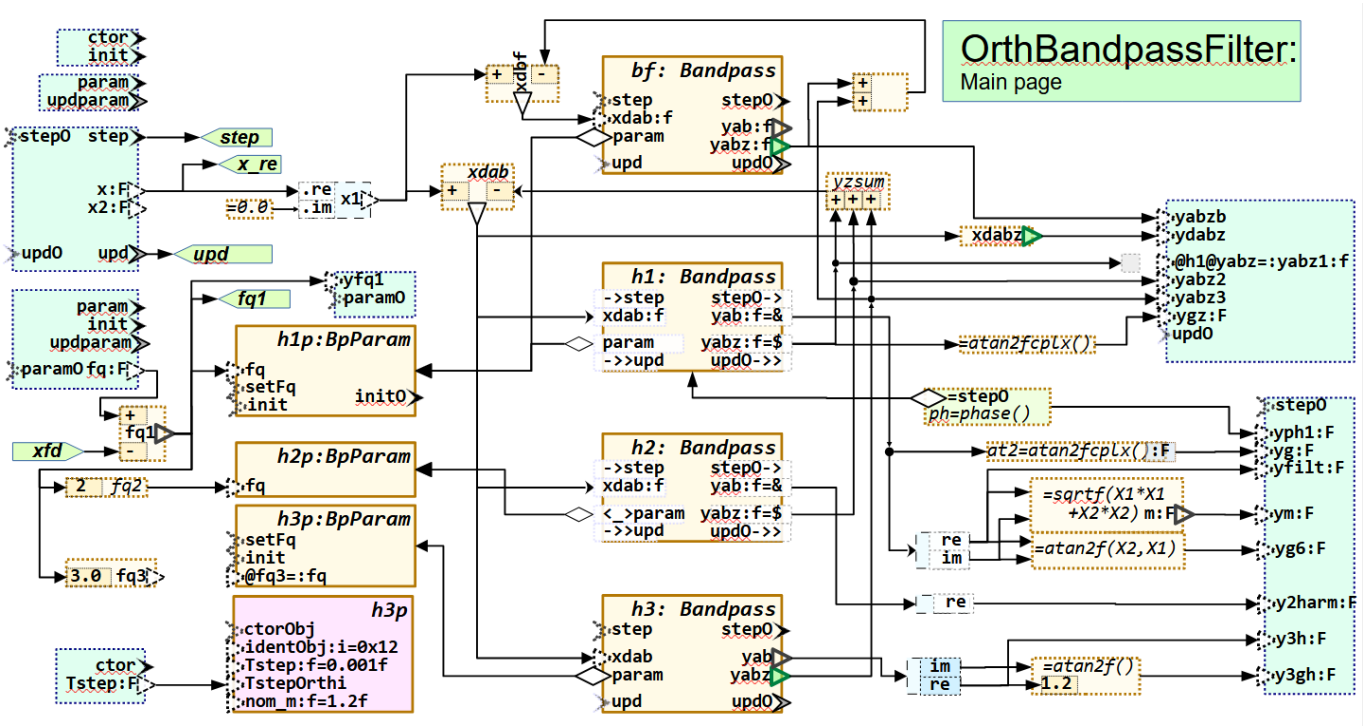


Figure 167: odg/OrthBandpassFilter.odg.png

This is the similar equivalent of the 4diac image left side () in OFB. The `REQ` event is here named `step`. Also here it is assigned to the data input `x`., compare to . Here the association between `step` and `x` is given because both are in the same `ofbModulePins` GBlock left side in pastel green. If the `step` event comes, `x` is offered with `step`. The data flow is used.

Because the `xdab` subtract expression needs the input data from `yzsum`, this is executed firstly before the `xdab` sub is executed, as result of the necessary data flow. It is automatically detected by evaluation of the data flow and results in the same event flow as in .

If the sub in `xdab` done, then the data are provided to the `h1`, `h2` etc. There is a `step` event input of this FBlocks related to its data input. It means the event input is used if the data are provided. It is accidental, that the name of the event `step` is the same as the modules `step`. Not the names of events are responsible for connection, the data flow is it. But of course the same event name is nearby because of similar functionality.

In the 4diac left it is manually decided, that the two FBlocks for the `OrthBandpass` (it is adequate to `h1`, `h2`) are executed one after another. This is a pragmatic but not necessary decision if only one thread is used. The automatically created event flow does not decide about sequences, instead the event is provided from `xdab` to all three `h1`, `h2`, `h3` parallel. This enables the possibility to executed this parts parallel for code generation, but also if usual known in some sequential source lines, if multi threading (multi core execution) is not used.

Parallel events needs often a ***Join_UFB***, a specific FBlock with joins events. All parallel both may be executed, then the ***Join_UFB*** reacts with its output event. Such Join mechanism are also known in 4diac, named there RND (comes from Rendezvous of events).

In OFB you can look to the generated fbd file for the Module. The fbd is a File in IEC61499 syntax and shows the automatic evaluated event flow. It looks like for the , parts from x to h1:

```
EVENT_CONNECTIONS
.....
step TO x1_X.prep;
x1_X.prepO TO x1.prep;
x1.prepO TO yzsum.prep;
yzsum.prepO TO xdab_X.prep;
xdab_X.prepO TO xdab.prep;
xdab.prepO TO h1.step;
h1.stepO TO d_17.prep;
d_17.prepO TO JOIN_stepO.J1;
```

later comes:

```
x1.prepO TO d_15.prep;
d_15.prepO TO xdbf_X.prep;
xdbf_X.prepO TO xdbf.prep;
xdbf.prepO TO bf.step;
bf.stepO TO JOIN_dqref_X_prep.J1;
```

This is the parallel event chain for the other FBlock `bf`. The `d_15` is the expression right of bf, without a definitive name, hence automatically named. But also the data connections are given in this file, and the definition of the FBlock:

```
FBS
...
d_15 : Expr_FBUMLgl( expr:='~+,+,+;,,;;' )
(* @1'0y=22:26, x=123..129 *);
```

In the FBS = Function BlockS definition part you see the constant input for the expression operators (see *5.8 Expressions inside the data flow (FBexpr)* page *128* and also as comment string some additional information, especially the position in the graphic page 1, y=22 mm, x)123 mm, so it is able to find in the graphic.

Also in the code generation this sequence of events is able to see, due to the sequence of statements. So you can check whether maybe specific drawing stuff is proper mapped to the event connections and hence sequence in code generation.

How the event connections are evaluated from the data flow, this is described as overview in chapter *5.12 Execution order, Event and Data flow, Event chains and states* page *196*. For details you can refer the sources of translation in Java, show log outputs etc. in debugging mode.

Events are also important for State machines. This is in the moment not in focus, but will be done in future.

If you are thinking to the Sequence Diagrams in UML, the origin idea of this sequence diagrams may be really the event communication. But as concession to code generation, which does not regard event thinking, it was broken down to *“operation sequences”*.

5.12.2 Event chains for each one operation, state variables

The example before in *Figure 167: odg/OrthBandpassFilter.odg.png* shows a module with one essential operation. The module has also a constructor, an init operation and update beside step, but not shown in this figure. This a little bit more complex module has more pages.

To explain event chains and operations lets look in a simple test example:

Empty page

5.13 Drawing and Source code generation rules

Table of Contents

C/++ is only one example for a target language but it is the most familiar, hence it is used her for description.

5.13.1 Writing rules in target language used from generated code from OFB

Often some core functions are offered, or they are anyway existing in the target language. Follow the idea of system levels, modules and black boxes, such functions are independently tested and documented (independent of an application) and can be really seen from the graphic level as *“black box”*, understandable what they do, but the inner operations are not topic of study, they are presumed as well.

Of course the provided functions in the target language should be proper to the source code generation of the ***OFB*** with whose event-data and the Object oriented concepts. That is usual possible with some wrappers around legacy software or, for Object Orientated C language, this concept is anyway proper.

Details of the following rules can be adapted in the templates for Code generation, see chapter *6.1.7 Templates for code generation* page . For the standard given templates for ***emC*** (*embedded multiplatform C/++*) it means:

- Data associated of one module with name `MyModule` should be assembled in a `struct` with the name `MyModule_s`. The trainling `_s` is used to differ the module’s identifier with the class name without `_s`, if C and C++ are mixed (may be recommended). Note: Use the `typedef` style

```
typedef struct MyModule_T {
  int32 myVariables;
} MyModule_s;
```

- The usable type is then only `MyModule_s`, and not `struct MyModule…` as often seen. It is more simple and obviously.

- You can have a class encapsulating the `struct` definition:

```
class MyModule : MyModule_s {
  inline void step (...) {....}
};
```

The class wraps the:

- C-language Object-Oriented Operations which should be written as:

```
void step_MyModule(MyModule_s* thiz, ....) {
.... }
```

- It means there are operations in C which are strongly related to the data with the data pointer named `thiz`. It is similar the C++ `this`, but written with `z` to allow mix with C++ and use a C++ Compiler for C files (which may be seen as recommended).

- The names should be `step_`, `upd_`, `init_`, `ctor_` following with the Module name, as default. That are the default names for the events automatically created and used, or specific names determined by the evin of the FBlock.

5.13.2 Life cycle of programs in embedded control: ctor, init, step and update

The OFB is first for embedded control programming with graphical support. For that speak about the life cycle.

Usual in embedded control programs does not use frequently allocated memory because of the possibility of fragmented memory, and also there is no process management which can free the whole memory if an application is closed. Normally an application is never closed. That's why allocation of memory is only usual on startup. All instances are prepared, and then the program runs till power off or reset. In rare cases specific applications are added on demand and also removed if there are no more necessary, with a may be specific memory allocation handling.

This is other than in PC programming, where a running program is a job, used on demand, finished and removed if it is no more necessary – or it hangs. An embedded application must never hang, it should run without restart also some years.

The OFB supports that thinking and regards three phases:

- `ctor`: This is an event or operation call to construct one FBlock either independently or with knowledge of values (data inputs) and other FBlocks (as aggregation) which are already constructed before. This means that the knowledge of data is consistently tree-like.

Because of specific handling of construction the operations for the constructions must start with `ctor` and other operations must not start with `ctor`. To fulfill this necessity for legacy code you can write simple wrappers (maybe as `#define` or as `inline`) which does not cause additional code.

```
#define ctor_MyModule(THIZ) \
legacyConstructionRoutine(...)
```

The often seen rule to write macro names only in upper case is of course not recommended here. Or better use the `inline` possibility available since C99 also for C language.

- `init`: A specific initial phase is necessary if there are circular dependencies between FBlocks. To fulfill a correct initialization one FBlock should be deliver proper initializing data, but this FBlock may depend also from other FBlocks. Then the initializing can be done only step by step. A proper example is: Aggregation between two FBlocks each other, maybe also to inner instances of these FBlocks (ports).

That's why the `init_MyModule(...)` operations are executed in a loop till all is ready. The basic form for that is:

```
ctor_FB1(&dataFB1, args);
ctor_FB2(&dataFB2, args, ... dataFB1);
//
bool bInitOk;
int ctAbortInit = 10;
do {
  bool bOkPart;
  bOkPart = init_FB1(&dataFB1, ... &FB2);
  bInitOk &= bOkPart;
  bOkPart = init_FB1(&dataFB1, ...&FB1);
  bInitOk &= bOkPart;
} while(!bInitOk && --ctAbortInit >=0);
if(ctAbortInit <0) {
  THROW(...) // a faulty state
}
```

As you see here (example) the `ctor_FB2` can use the `FB1` because it is always constructed, but not vice versa. But the `init_FBx` can use the (already existing, constructed) other FBlocks. The `init_` operation checks whether it has all necessities gotten from the other FBlocks, then it returns true. Else it returns false. The `init_` operations are all called one after another, in a proper but, not strong order. They are called repeatedly in this loop. But the loop is aborted if it needs too much iterations, which are intrinsically a result of a software error (any FBlock is not satisfied with the other ones). It means on `ctAbortInit <0` an emergency handling (search the cause) is necessary. The maximum number of necessary `init_` loops should not greater then the number of `init_FBlocks(...)` in the loop. Then also in a revers sensitive order called `init_FBlocks(...)` delivers the data from the last called to the first one.

Because of this specific handling, the operations for initialization must start with `init_` and other operations must not start with `init_`, or basically, the `init` event should be used for `init` in the graphic. To fulfill this necessity for legacy code you can write simple wrappers (maybe as `#define` or as `inline`) which does not cause additional code.

```
inline init_MyModule(MyModule_s* thiz, ...) {
legacyInitialization_Staterments(...)
}
```

- `prep` or `step`: This is the often cyclical called step routine for the sampling time. Such operations are often called immediately in interrupts. It is also possible to call lesser prior routines in a back loop of a simple controller organization without a specific RTOS (*RealTime Operations System*), or just also in a specific RTOS. *prep* comes from *prepare* in opposite to *update*.

- `upd` operation for *update*: In controller algorithm with often solves differential equations it is necessary first calculate the new state of all inner variables using the previous (old) state, and then update all states at ones.

If new and old variables are sometimes used confused, the results are often not entirely correct. With sensitive algorithms (e.g. filters) they are completely wrong. This is often not properly taken into account. The code generation of OFB respects this. The basic form of this is:

```
interrupt opeationOneStep (...) {
upd_FB1(&dataFB1, ...)
upd_FB2(&dataFB2, ...)
prep_FB1(&dataFB1, ... &FB1, &FB2)
prep_FB2(&dataFB1, ... &FB1, &FB2)
```

As you see, first all ***update*** are done for new states, using the current ones. Then ***prepare*** the new states to the current ones comes for the next step.

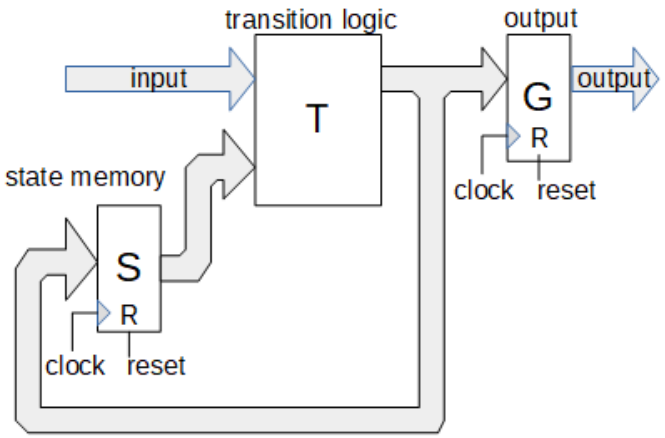


Figure 168: PrepUpd/q-input-trans-qout.png

This is similar also of D and Q on Flipflops in digital logic. As you see in the image for embedded control it is typically that the output has its own clock mechanism, it is the clock in the hardware. That's why the prep results should be used as new values for hardware output. The update state is to access the last values from the step time before. That's why update comes first. Sometimes it is revers in thinking.

The `upd` operations helps also for data consistence. If a whole update operation (consist of calling some `upd` operations for the inner FBlocks) are executed in a locked state (with mutex) or just in *disable interrupt* state for a simple non RTOS controller software, then interruptive routines gets always consistent data from its interrupted operations (tasks). The update operations usual should not need longer calculation times, because the do only copy data.

The `ctor`, `init`, `prep` or sometimes `step` and the `upd` are the basically existing events for execution. Regarded in the models by the user, regarded by source code generation.

5.13.3 Using events in the module pins and FBlocks, meaning in C/++

See chapter *4.4 Event-Based Execution* page

The events in an OFB diagram replaces on the one hand the often used "*sampling times*", on the other hand they are really events in an event controlled execution. But for code generation the execution of an event in a FBlock is one operation. That's the important rule.

But the events should not be elaborately shown and wired in the diagrams. Similar as associating sample times to data in other FBlock graphic tools, the events need primary only be given in the module's pin definition (style `ofbMdlPins`). Not only the wiring of events in the diagram (event connections) can be omitted, also events in FBlocks can be omitted, if the association with the data is unique.

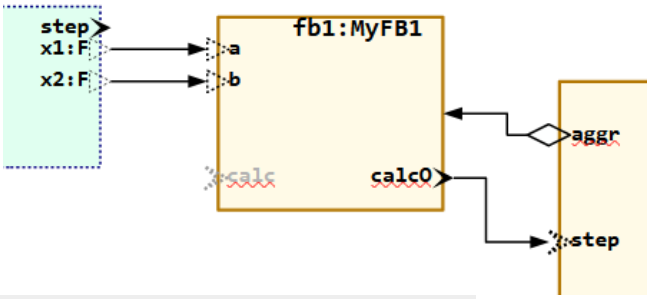


Figure 169: ExmplEvDeflt_calcOstep.png

Look for a not simple but should be obvious example in

- The both input values `x1` and `x2` are associated to a module input event `step`, usual the module gets a `step_..(..., float x1, float x2)` operation.

- The `fb1` has a named output event `calc0`. Hence for the input variables the input event, here drawn in gray as not active, is `calc`. The called operation is `calc_MyFB1(…)`. If the FBlock would not have any event designation, a `prep` event will be created as default.

- But notice, that an event – data association can also be drawn on another position of the graphic, proper to the rule "*Any element of the functionality can be shown more as one time in different contexts*" described in chapter *Error: Reference source not found* page . If the data inputs are associated to another event there, this is valid. Then the here shown `calc0` does not influence the input data association between `calc0` is an output event.

- For this example it is shown in the graphic that a called `calc_...(fb1...)`operation is followed by a `step_..(fb2...)` operation of the next FBlock because this is dedicated by the here shown event connection. In this special case the `fb1` has no data output which should elsewhere determine the calculation order (or just event connection). Hence it should be dedicated by the drawn event connection.

- The aggregation from the second `fb2` to the `fb1` needs an initialization. For that both FBlocks gets an `init` → `init0` event pair per default (as nowhere other it is dedicated in another way, just as default). The own address of the `fb1` as "port" output is related to the `init0`event, and the aggregation is related to the `init` event of the right FBlock.

- And also for construction a `ctor` and a `ctor0` event is associated to all FBlocks which are not expressions.

With this simple rules the code generation from OFB to C language in the default version (can be adapted, see TODO) is compatible with your basic function blocks in C language.

Then you don't need specific extra definitions outside of the Libre/Open Office graphic.

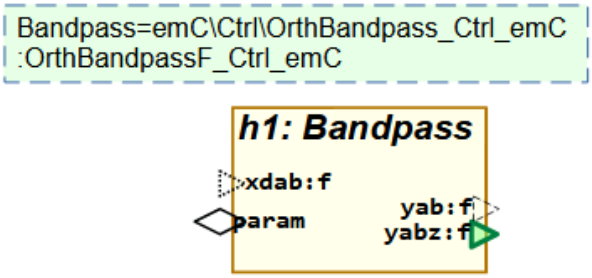


Figure 170: FBlockSimpleUsage.png

This is the only necessity in the graphic to use it together with the existing code in C/++ language:

- The green box is of style `ofbImport` and declares the alias `Bandpass` in the graphic as full Module type `OrthBandpass_Ctrl_emC` which is the module's name in C language (see ./../../../vishia/emc/html/Ctrl/OrthBandpass.html (http://www.vishia.org/emc/html/Ctrl/OrthBandpass.html).

- The input events `step`, `init`, `ctor` and the output events `step0` and `init0`, are automatically created because here events are not defined.

- Because at least one output with the graphic style `ofpZout...` is given, also the input event `upd` and the output event `upd0` is automatically defined.

- All data inputs are associated to the `step`, all data outputs which are not `ofpZout` are associated to `step0`. All `ofpZout` outputs are associated to `upd0`.

- All data inputs and outputs should be marked with the used types, here `F`for `float` and `f` for `complex_float`. This designation is only necessary ones if the FBlock is more as one time used.

- All aggregations, also associations are associated to the `init` event. They are inputs for the `init` event or just the `init_Module(thiz, param)`generated C operation though the

direction of the connection is to the referenced class, to initialize the reference.

- All Ports (not in example) with graphic style `ofpPort...` are associated to the `initO` event. They are outputs usable for other `init` inputs due to there reference connections.

5.13.4 More possibilities, definition of special events

If your target language module has more operations then the `ctor_…`, `init_…`and `step_…`, or you want to use another name instead for `step_…` then you can define your own events.

- TODO event with data in one block: It is for the data, an aggregation is not associated, it is associated to init.

- event in one block only with aggregation: It is instead init

- You can have more as one graphic block to show specific data and event relations.

TODO figures, program, test.

(empty page)

5.14 Showing processes

This chapter is not part of code generation yet, but a candidate. It describes a diagram kind, respectively parts inside a FBlock, which execution are done in an operation. Inclusively if, while, call.

(empty)

(empty page)

6 Setup and Explanation how does the script work

6.1 Converting the graphic – scripts for source code generation

As fast mentioned also in chapter *Error: Reference source not found*page *Error: Reference source not found* , one of the important capabilities is the generation of code in a proper target language.

The other approach is: storing the graphic in a unique proper readable textual representation, especially for versioning.

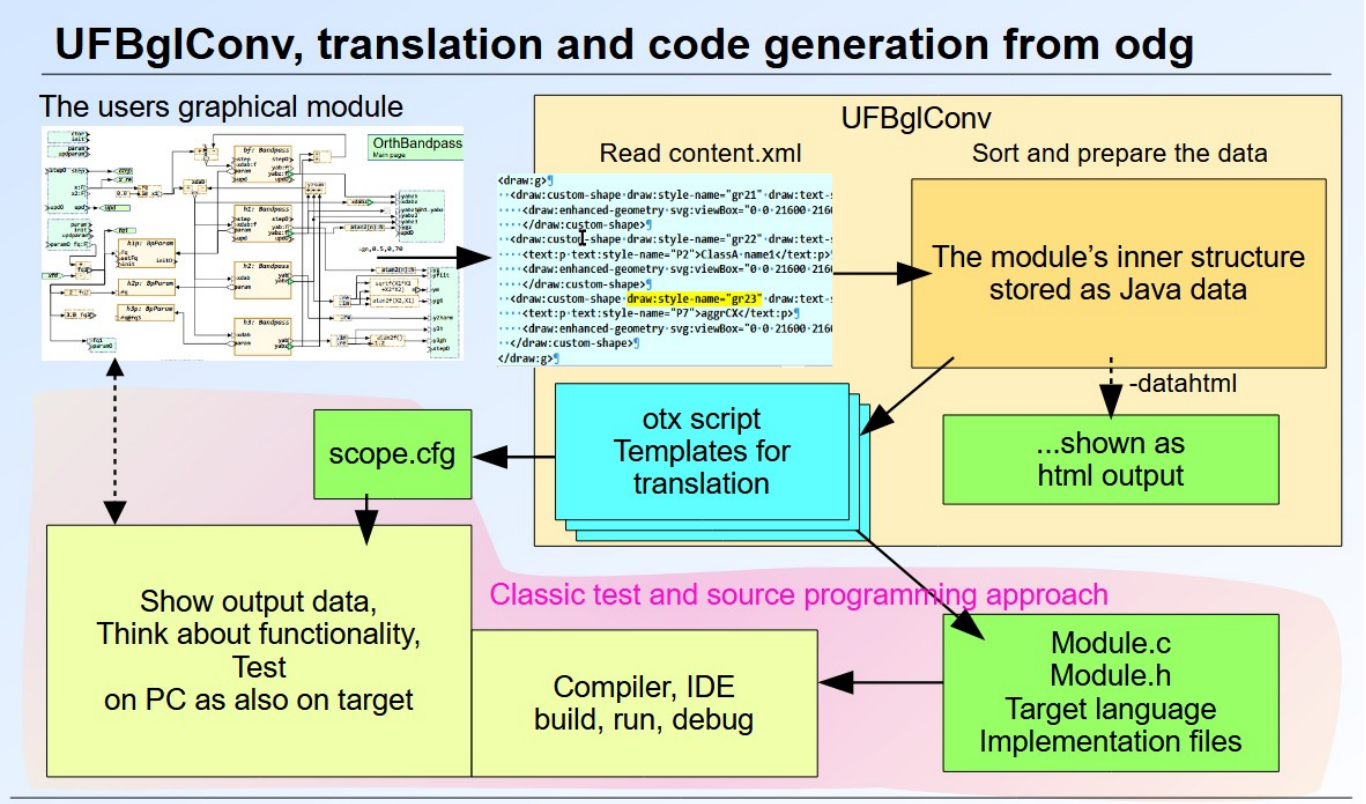


Figure 171: FBcl/OFBConvAndTestSlide.png

The slide above shows the working flow with OFBConv code generation. The classic approach is the magenta area on bottom side: Manually written code, test and compare with an only-documented module architecture and design. That is also valid, but supplemented with an automatically code generation from the graphical module, as shown on upper side in the slight. For code generation proper readable and adaptable templates are used as otx scripts.

The translation itself is done by a Java program. The necessary file is downloaded in the `tools` directory beside src, see TODO

To calling this translator, arguments are necessary, and also some directories should be prepared, and a textual difference viewer of the generated files in comparison to the version before may be called. Also a make of target software can immediately run.

This all is organised by either a batch file (MS-Windows) or a shell script (Linux, or on MS-Windows if for example MinGW is existing coming with a git installation). Both scripts, batch and shell does the same. But some commands are different.

The scripts can also include the compilation of the generated code till creating an executable, and start it. This is a end-to-end way from the graphic till its simulation, for simple and fast test.

6.1.1 Start scripts from the component

There are two centralised generation script for both OS-Systems in:

* `.../src/scripts/bat/genSrc_odg.bat`
* `.../src/scripts/shell/genSrc_odg.sh`

This scripts are called in the local existing bat or shell script file for the component to translated.

The original given script in OFB_Presentation regards the directory structure describe in *5.1.3 File tree structure of the examples* page *34*. The script can be adapted if the user want to have another file tree structure. Details are explained in 6.1.1 Common script for translation odg to target code page 188

Lets see on the more complex example for the

`.../src/ExmplPositionCtrlPID`

This directory for the component contains the `odg` graphic file:

`odg/ExmplPositionCtrlPID.odg`

configuration file to control the translation for target code generation:

`makeScripts/CcodeGenTarget.cfg`

A target related configuration file to control which header are necessary to include for C/++ code generation, and an adaption of names in graphic to the names really used n C/++ level:

`makeScripts/aliasHeader.target.cfg`

And last not least the both (Windows / Linux) translation scripts:

`makeScripts/genSrc_odg.sh`
`makeScripts/genSrc_odg.bat`

The scripts can be used to start manually from a command window or with double mouse clique from Explorer. They are also used if the macro and button is installed in LibreOffice to translate, see *6.1.4 Macro and Icon from LibreOffice Draw* page *222*.

Both scripts are general equal. The Windows batch file is the following:

include:../../+ExmplPositionCtrlPID/makeScripts/genSrc_odg.sh::$::96::::

```
#!/bin/sh:
## This script is called also from LibreOffice Draw with the %1 file.odg %2 %3 possible NOPAU...
## this shell script calls the common translator batch with proper arguments
## The SRCODG may be used in the TARGET.cfg as input file, or simple usage it is only a pattern:
export SRCODG="$1"
if test "$SRCODG" = ""; then export SRCODG="ExmplPositionCtrlPID.odg"; fi
## The current directory must be set to the 'SOURCE.wrk/src' where the COMPONENT and scripts ...
## It should be matching to the content of the TARGET.cfg
cd $(dirname $(realpath "$0"))/../..
## arument1 is the COMPONENT/makeScript/TARGET.cfg, argument2 is the path/to/build/COMPONENT ...
scripts/genOFB/genSrc_odg.sh +ExmplPositionCtrlPID/makeScripts/PCsim.args ../build $2 $3 $4
read -p "press Enter" INPUT
```

include:../../+ExmplPositionCtrlPID/makeScripts/genSrc_odg.bat::$::96::::

```
echo off:
REM This batch script is called also from LibreOffice Draw
REM with the %1 argument: The odg file,
REM and with the %2, %3, %4 arguments possible with "NODIFF" "NOPAUSE" "MAKE"
REM
REM manually calling without these arguments translates with diff view, with pause and not wi...
REM MAKE is the automatic calling of the C/++ make script CcodeGenPcSim.Zmake.jzTc.bat
REM which can be called even manually.
REM
REM This batch calls the common translator batch with proper arguments:
REM The SRCODG may be used in the gensrc_odg.args as input file,
REM set it for manually call without arguments (for double click):
set SRCODG=%1
if "%SRCODG%" == "" set SRCODG=ExmplPositionCtrlPID.odg
REM
REM argument1 is the path\to\TARGET.args, argument2 is the path\to\build\target
REM argument3, 4, 5 comes from args %2 %3 %4 possible from the LibreOffice Macro
REM current directory has to be ...\src necessary for arguemnts:
cd /D %~dp0\..\..
call scripts\genOFB\genSrc_odg.bat +ExmplPositionCtrlPID\makeScripts\PCsim.args ..\build %2 %...
REM
REM pause
```

The comment lines should be explaining, but may be too much here. The essence is:

1) The current directory is set to the directory of the component:

```
cd /D %~dp0\..\..
```

2) The common script is called with the configuration file as argument with the path from the just set current directory:

```
scripts\bat\genSrc_odg.bat ...CcodeGenTarget..
```

The second argument of this file is the `path/to/build` directory where build results are stored for this translated component.

The `SRCODG` environment variable is only necessary because used in the config script, see next chapter.

6.1.2 Configuration file for translation

include:../../+ExmplPositionCtrlPID/makeScripts/PCsim.args::$::96::::

```
## This is a command line argument script for Java, but place its role as configuration for ...:
## each arguments is one line
---is a commented argument for the java main routine
## The arguments starts after the line with label 'args'.
## This line defines the end line comment sequence ' ##', spaces before are also omit...javaArgs
args ##
---ifbd:path/to/Module.fbd            ## Read a textual stored modu...ifbd
     ##
-i:LibOFB_emC/odg/LibCtrl_emC.odg     ## The declaration (lib-) input odg file to transla...iodg
-im:PIDctrl_TsModulDef                ## Use The PID module definition from this file...im
-im:BandpassFilterModulDef            ## use from here the average module definiton module.
-im:Angle_Ctrl_FBtypedef              ## Some angle calculation FBtypes.
-cfg:LibOFB_emC/makeScripts/LibCtrl_emC.target.cfg     ## cfg for code generati...cfg
     ##
-i:$CMPN/odg/$SRCODG                   ## The input odg file to translate
---im:ModuleSpec                      ## select translation only for this modu...im
---xm:ModuleSpec                      ## alternatively read all but exclude some specific m...xm
-nameCmpn=..                          ## directory name builds $nameCmpn, as part of the ...cmpn
-cfg:$CMPN/makeScripts/ReplNamesHeader.cfg
     ##
-tplCode:@org.vishia.fbcl.writeFBcl.WriterCodegen:cHeader.otx     ## possible use other co...otx
-tplCode:@org.vishia.fbcl.writeFBcl.WriterCodegen:cImpl.otx
-tplCode:@org.vishia.fbcl.writeFBcl.WriterCodegen:cState.otx      ## has state machines
-tplCode:LibOFB_emC/makeScripts/scopeEthernetComm.otx ## code generation templates for th...gTxt
     ##
-dirGenSrc:$BUILDD/$nameCmpn/cpp/genSrc  ## The output directory for generated souce co...out
-dirCmpGenSrc:$nameCmpn/cpp/genSrc      ## directory for compare with source code generat...cmpr
-fbg:$BUILDD/$nameCmpn/cpp/genSrc/FBcl/    ## write raw content of each module to th...fbg
-dirFBcl:$BUILDD/$nameCmpn/cpp/genSrc/FBcl ## directory for generated FBcl fil...FBcl
-dirCmpFBcl:$nameCmpn/cpp/genSrc/FBcl   ## dir to compare generated FBcl fil...cmpr
-dirReport:$BUILDD/$nameCmpn/cpp/report ## output directory for some log files for data de...rep
-log:$LOG                            ## output file for l...log
-dirDbg:$BUILDD/$nameCmpn/cpp/report/dbg       ## output directory for some log files for ...dbg
     ##
-oxmltest                           ## possibility to write back the read content.x...xml
---oxmldatahtml                      ## possibility to write internal data as ht...dataHtml
---datahtml                          ## possibility to write the internal data in html
```

This is the configuration file as example for this `ExmplPositionCtrl`. It is not the simplest example and shows some common possible specifics.

Primary, this is intrinsic an argument file. Each line is one argument. In the calling common translation script it is used as

```
java ...  --@makeScripts/$SRCODG.args:args
```

All arguments are written after the line starting with `args ##`. and then each line is used for one argument. Arguments starting with `---` are commented out – for flexibility and offered as possibility.

`ifbd` `-ifbd:…` This line is commented here. It is the possible input of fbd or FBcl files, which describes the interface to used modules. Especially for modules which are present in the target language, not graphically drawn, can be inputted by an interface description in IEC61499 syntax (textual). This interface description may be simple proper to hand-written, but also an automatic translation from C-header files or other OFB modules translated before can be used.

`iodg` `-i:…` defines an `input.odg` file to translate. Here at first the common library `LibOFB_emC/odg/LibCtrl_emC.odg` is used, which defines the interface to given modules in C, in the `src_emC` component. More as one such argument are possible, hence more input `odg` files . A Module can have some pages in more input files, all they are summarised before code generation of the module.

The extension of the `-i:` file determines how to read it. `.odg` is LibreOffice, `.fbd` is a IEC61499 file. `.slx` should be for Simulink (yet TODO), all other graphic sources should/can be translated adequate if the translator supports it.

`im` `-im:…` If this option is used, only the named Modules (more as one possible) are translated from the odg file. This is usual for specific tests.

`xm` `-xm:…` This is the opposite to `-im:…`. If given and `-im:…` is not given, all modules are translated from the odg file, exclusively the here name modules. This is usual for specific tests.

`cfg` `-cfg:path/to/target.cfg` The `target.config` file contains some information about replacement of identifier and including headers or other imports. The `-cfg:...` arguments is valid for-each `-i:…` input file, hence module specific.

`cmpn` `-dirCmpn:..` This is a helper to have an internal environment variable `$scrCmpnDir` usable in following arguments with the name of the referenced directory.

The `..` means, name of the parent directory of the last `-i:…` input file.

`gTxt` `-tplCode:…` This is the path to gTxt (otx) files which controls the code generation. If this argument is not used, the internal files for C-code generation are used. If the argument is given, then all files should be given here. The internal files can be addressed as seen in the here given writing style: After `@` the Java class path in one of the jar files is used. The given `gTxt` file after colon is used from the package directory where the given class is located.

For this example an additional specific gTxt file `scopeEthernetComm.otx` is used to generate code for InterProcessCommunication.

`out` The three `-dirGenSrc:` `-dirFBcl:` `-dirReport:` describe where the output files should be stored. The name of the output files are name of the module in the `ofbTitle` shape in the graphic, with the proper extension given in code generation `otx` scripts or `.fbd`, etc.

`cmpr` This are directories for comparison the result to get a fast message whether it is the same. It is more for internal test.

`rep` `-dirReport:…` decides a directory for some report files of translation to fine explore what was happen. This is the data type propagation, the event connection due to data flow etc.

`log` `-log:…` If given then a log about translation with error messages for user is written to this file. If not given but rep is given, then the log file is written there as `log.txt`. The log is also written on console out. If `-silent` is given as argument, nothing is written to stdout, but as desired to the log file. `-silent` is only related to the stdout (on console).

The here used `$LOG` variable is set as environment variable in the calling script (see next chapter).

`dbg` `-dirDbg:…` is optional. It is more for inner debug files..

`odg` The option `-odg` forces output of a textual file which documents the internal graphic structure as text (not in IEC61499 syntax). In the necessary given `-dirReport:` directory. The advantage in opposite to an fbd file is: If a FBlock is more as one time drawn, all draw instances are reported. But the summary of the

FBlocks for its functionality is not contained there, it is in the fbd file.

`xml` That are some options for debug:

Here it is not so exact, TODO look in Java sources again and fine tuning

- An fbd file is output always if the `-dirFBcl:` directory is given.
- `-log` writes a log file for example with the execution order of data type propagation and event propagation in the given `-dirDbg:` directory.
- `-oxmltest` forces the output of the read `content.xml` file after reading (check of the correctness of `XmlReader`, or also look for details in the graphic file).
- `-oxmldatahtml` writes the read XML data (Java internals) in a readable html file.
- `-datahtml` writes the prepared module data (see chapter *6.1.3 Common script for translation odg to target code* page (Java internals) in a readable html file.

`SRCODG` If the script is called from the macor in LibreOffice (see 6.1.2 Marco and Icon from LibreOffice Draw page 149 then the %2 is proper set witth the odg file name. This is also if the script is called from another script. But if the script is called pure, and only one odg file exists in this content, then you can set its name as default.

6.1.3 Common script for translation odg to target code

The code generation from Open/LibreOffice odg files can be performed with the following batch or shell script. Both the batch and the shell are given one after another, so you can also see what's the difference in the scripts between Linux and Windows. A shell script can also run under Windows using for example MinGW. But the statements for symbolic link are different.

include:../../scripts/genOFB/genSrc_odg.bat::showArgs::96::::

```
REM This file is the batch file to call java for translation graphic to target source,
REM This file can be called per macro from LibreOffice Draw to execute translation and make.
REM The convention is: This script is located in '.../scripts/bat'
REM the current directory is the dir of the component.
REM
REM beside the component directory in src/%CMPNDIR%/odg/....
REM From the absolute directory of this called script, the component can be found.
REM expected arguments:
::echo currdir=%CD%
echo called=%0 %1 %2 %3 %4 %5
echo CD=%CD% : need to be ...\src
echo arg1=%1 : COMPONENT/makescript/TARGET.args
echo arg2=%2 : BUILD directory BUILDD
echo arg3=%3 : NODIFF or NOPAUSE or MAKE to prevent diff or pause
echo arg4=%4 : NODIFF or NOPAUSE or MAKE and optional start MAKE
echo arg5=%5 : NODIFF or NOPAUSE or MAKE
```

include:../../scripts/genOFB/genSrc_odg.sh::showArgs::96::::

```
## This file is the shell script to call java for translation graphic to target source,
## This file can be called per macro from LibreOffice Draw to execute translation and make.
## The convention is: This script is located in '.../scripts/bat'
## the current directory is the dir of the component.
##
## beside the component directory in src/%CMPNDIR%/odg/....
## From the absolute directory of this called script, the component can be found.
## expected arguments:
echo currdir=$PWD                                          ...CurrDIR
echo called=$0
echo 'arg1'=$1 : COMPONENT/makescript/TARGET.cfg           ...TARGET
echo 'arg2'=$2 : BUILD directory BUILDD                    ...BUILDD
echo 'arg3'=$3 : NODIFF or NOPAUSE to prevent diff or pause
echo 'arg4'=$4 : NODIFF or NOPAUSE to prevent diff or pause
```

The scripts are shorten in line, look to the originals.

This is the start of both scripts which shows given calling argument and the explanation of them.

CurrDIR shows the current directory but tests also whether it is proper. It tries to set the estimate current one if it is faulty. If not possible the script is aborted with an error message.

TARGET is the path to the target configuration file from the current directory.

BUILD is the path to the build directory from the current directory.

include:../../scripts/genOFB/genSrc_odg.bat::prepEnv::96::::

```
REM Prepare environment variable
set SCRIPTDIR=%~dp0
set TARGET=%~n1
for %%9 in (%~p1\..) do set CMPN=%%~n9
set CMPNDIR=%~dp1..
set BUILDD=%~dpnx2
if "%3" == "NOPAUSE" set NOPAUSE=NOPAUSE
if "%4" == "NOPAUSE" set NOPAUSE=NOPAUSE
if "%5" == "NOPAUSE" set NOPAUSE=NOPAUSE
if "%3" == "NODIFF" set NODIFF=NODIFF
if "%4" == "NODIFF" set NODIFF=NODIFF
if "%5" == "NODIFF" set NODIFF=NODIFF
if "%3" == "MAKE" set MAKE=MAKE
if "%4" == "MAKE" set MAKE=MAKE
if "%5" == "MAKE" set MAKE=MAKE
echo CD=%CD%
echo SCRIPTDIR=%SCRIPTDIR%
echo BUILDD=%BUILDD%
echo ARGS=%1
echo CMPNDIR=%CMPNDIR%
echo CMPN=%CMPN%
echo TARGET=%TARGET%
echo pause, diff, make=%NODIFF% %NOPAUSE% %MAKE%
```

include:../../scripts/genOFB/genSrc_odg.sh::prepEnv::96::::

```
## Prepare environment variable
SCRIPTDIR=$(dirname "$0")
TARGETCFG=$(basename "$1"); export TARGET=${TARGETCFG%%.*}
export CMPN=${1%%/*}                  ## the part till first / from $1
export CMPNDIR="$(dirname "$1")/.."
export BUILDD=$(realpath "$2")     ## before change the currd...BUILDD
if test "$3" = "NOPAUSE"; then export NOPAUSE="NOPAUSE"; fi
if test "$4" = "NOPAUSE"; then export NOPAUSE="NOPAUSE"; fi
if test "$3" = "NODIFF"; then export NODIFF="NODIFF"; fi
if test "$4" = "NODIFF"; then export NODIFF="NODIFF"; fi
echo CONFIG=$1
echo CMPNDIR=$CMPNDIR
echo CMPN=$CMPN
echo SRCODG=$SRCODG
echo TARGET=$TARGET
echo SCRIPTDIR=$SCRIPTDIR
echo BUILDD=$BUILDD
echo pause, diff=$NODIFF $NOPAUSE
```

This parts sets some environment variables which are used in the script itself, and possible also inside the 6.1.2 Configuration file for translation and inside the translator in Java.

Note for a small difference between Windows and Unix, which may not be exactly know by all users: In Windows the environment variables set with `set` are valid for the whole started process, also valid in the calling script after exit this script.

Where as in Unix/Linux, environment variables are never valid in the calling script (after finishing the current one), and are not valid in a called script, only valid in the own script, if they are set only with `name="value"`. If they are set with `export name="value"` then they are valid also in called scripts (but not in the calling script). The only one possibility to call a script to set environment variable is, not call the script, include it with the dot, or call with `source`, see for example in https://www.delftstack.com/howto/linux/bash-call-another-script/

Another important difference is: In Windows the `"..."` written surround the value are stored as part of the value. This is sometimes very stupid, depending on the usage. Write it without surrounding `""`! But care about spaces, also all spaces after the `=` till end of line (!) are part of the value of the environment variable. This is the fact since DOS and never changed.

Whereas in shell scripts for Unix / Linux, the surrounding `""` are not part of the value, surrounding with `""` is recommended.

Surround command line arguments with `""` plays the same role in Windows and Linux, they are general possible and necessary if the command line argument contains characters which have a special meaning in the command line interpretation, especially a space (separator for arguments) or also the `;` `>` etc.

Often this both aspects for "" are confused together.

Usage of the for statement to set CMPN in the batch file is also strange to understand. This is not a "for", Only the simple assignment of the content of the only one argument `(%~p1\..)` to the for variable `%%9` is essential. Only for "for" variables and for the arguments `%1`..., the special operations to get name, path etc. are applicable, see `help for` as command line help, see in german: https://de.wikibooks.org/wiki/Batch-Programmierung:_Erweiterungen_unter_Windows_NT

`CMPNDIR` is the **absolute path** to the directory where the component, `makeScripts`, `odg` etc. are located.

`CMPN` is the components name, derived from the first part of the given `arg1`. It is used to build the directory inside the build directory `BUILDD`

`TARGET` is used as part of designation of the target artefacts (generated files), especially for the log file name, and to look whether compilation is intended.

`SCRIPTDIR` is the absolute path to the own directory where this script is stored, to call scripts beside.

`BUILDD` is the absolute path to the given build directory as second argument `%2` or `$2` of this script.

`NODIFF` and `NOPAUSE` are set from the 3th and 4th argument, it controls whether the script is paused to show information, or it should be run without pause (for text), and whether the diff tool is used. See TODO generate.

include:../../scripts/genOFB/genSrc_odg.bat::BUILDD::96::::

```
REM check the BUILD Directory
if exist %2 goto :okBUILDD
  echo BUILD directory does not exists
  echo BUILDD=%BUILDD%
  echo currdir=%CD%
  echo create %2 ?
  if "%NOPAUSE%"=="" set /P INPUT=" [ y | n |] ENTER "
  if "%INPUT%"=="y" mkdir %2
  if not exist %2 (
    echo "BUILD directory not able to create, abort"
    pause
    exit /B 128
  )
:okBUILDD
```

include:../../scripts/genOFB/genSrc_odg.sh::BUILDD::96::::

```
## check the BUILD Directory
if ! test -d $2; then
  echo BUILD directory does not exists
  echo BUILDD=$BUILDD
  echo currdir=$PWD
  echo create $2 ?
  read -p " [ y | n |] ENTER " INPUT
  if test "$INPUT" = "y"; then mkdir -p $2; fi
  if ! test -d $2; then
    echo "BUILD Directory not able to create, abort"
    read -p "press any key" INPUT;
    exit 128;
  fi
fi
```

This part in the script checks, whether the given `BUILD directory` as parameter is existing. It is **not** created by default, because here it is sensible also to have symbolic links determined outside of this script, preferred using a RAM disk for build artifacts for faster work and be carefully with hard disk / SSD space. If the `BUILD directory` is not existing, then the script asks, and you can create the necessary desired linked directory for the `BUILD directory` by special handling outside.

Then press “n” or only ENTER. For the standard case, build should be created on hard disk as given, press “y” and ENTER. Then a `mkdir` is preformed. If this should be not successful (what ever the reason), then the script is finished with `exit`. An existing `BUILD directory` is necessary for further working.

If the `BUILD directory` exists, immediately or as symbolic link, all is ok, nothing is asked or done.

The `BUILD directory` is never cleaned here. Do it outside if necessary.

include:../../scripts/genOFB/genSrc_odg.bat::CurrDIR::96::::

```
REM ensure current dir is ok
if exist %1 goto :TargetOk
  if not exist scripts\genOFB; then cd %SCRIPTDIR%\..\..
  if not exist %1 (
    echo CD=%CD%
    echo arg1=%1
    echo faulty current directory, abort;
    pause
    exit /B 255
  )
  echo currdir changed: =$PWD
:TargetOk
```

include:../../scripts/genOFB/genSrc_odg.sh::CurrDIR::96::::

```
## ensure current dir is ok
if ! test -f $1; then                                    ...CurrDIR
  if ! test -d scripts/shell; then cd $(dirname $0)/../..; fi ## this is the expected dir
  if ! test -f $1; then
    echo Not found: $1
    echo possible faulty current directory, abort;
    exit 255;
```

```
  fi
  echo currdir changed: =$PWD
fi
```

This part checks whether the current directory is proper. It may be possible to call this script to execute, but the arguments are not exact related to the set current directory, or setting the current directory was forgotten. This can be a slip error which is able to repair here.

The repairing algorithm presumes, that both the component directory and the script directory are on the same level in the directory tree.

include:../../scripts/genOFB/genSrc_odg.bat::LOG::96::::

```
REM check and set log
set LOG=%BUILDD%\%CMPN%\cpp\genSrc\log\%CMPN%_%SRCODG%_%TARGET%.genSrc_OFB.log
echo LOG=%LOG%
REM Possibility to change some environment variable independent of this batch:
if exist %SCRIPTDIR%\..\WindowsEnv\EnvVar.bat call %SCRIPTDIR%\..\WindowsEnv\EnvVar.bat
if not exist "%BUILDD%\%CMPN%\cpp\genSrc\log" mkdir "%BUILDD%\%CMPN%\cpp\genSrc\log"
if not exist "%BUILDD%\%CMPN%\cpp\genSrc\log" (
  echo "Abort: Log cannot be created in %BUILDD%\%CMPN%\cpp\genSrc\log"
  pause
  exit /B 64
)
echo "Test LOG=%LOG% (should be set newly by Java execution) " >%LOG%
```

include:../../scripts/genOFB/genSrc_odg.sh::LOG::96::::

```
## check and set log
export LOG="$BUILDD/$CMPN/cpp/genSrc/log/${CMPN}_$TARGET.OFBconv.log"
echo LOG=$LOG
## Possibility to change some environment variable independent of this batch:
if test -f $SCRIPTDIR\EnvVar.sh; then . $SCRIPTDIR\EnvVar.sh; fi
if ! test -d $(dirname $LOG); then mkdir -p $(dirname $LOG); fi    ## creates also the .../ge...
if ! test -d $(dirname $LOG); then
  echo "Abort: Log cannot be created in $(dirname $LOG)"
  read -p "press any key" INPUT
  exit 64
fi
echo "Test LOG=$LOG (should be set newly by Java execution) " >$LOG
```

The set and check the log file is a little bit elaborately. Why: The log file is used also in the script. Hence the script is responsible to the log is proper.

Additionally the `EnvVar.bat` or `.sh` is called. This script can set the environment variables to other values, for example for test cases. Normally it should be empty. Also other actions can be performed there. It is outsourced from this script to ensure independency.

The EnvVar.sh is called with the dot operator, then set environment variables in this script are valid here.

After conditional calling this script the directory for the LOG file is created, and tested whether it is ok. Then also the LOG file itself should be able to create,

include:../../scripts/genOFB/genSrc_odg.bat::pause::96::::

```
REM output and pause for user to view what should be done:
echo ==OFB translator: %CMPN% ========================================
if "%NOPAUSE%"=="" pause
```

include:../../scripts/genOFB/genSrc_odg.sh::pause::96::::

```
## output and pause for user to view what should be done:
echo ==OFB translator: $CMPN  ========================================= ## <:.~Info.>
if test "$NOPAUSE%"==""; then read -p " ...Press ENTER..." VAR; fi  ...pause
```

This part is only responsible to show the information about target code generation in this script. The pause statement can be commented for daily usage, if the information are not necessary to view again.

include:../../scripts/genOFB/genSrc_odg.bat::Java::96::::

```
REM set the java class path in JCPVISHIA as central batch: SetJCP:
call %SCRIPTDIR%\..\WindowsEnv\SetJCP.bat
echo java -cp %JCPVISHIA% org.vishia.fbcl.OFBconv --@%1:args >>%LOG%
echo java -cp %JCPVISHIA% org.vishia.fbcl.OFBconv --@%1:args
java -cp %JCPVISHIA% org.vishia.fbcl.OFBconv --@%1:args
REM Note the java running process writes some stuff on this console, but also in the -log...LOG:
echo ... finished OFB translator
if ERRORLEVEL 1 (
  if "%NOPAUSE%"=="NOPAUSE" (
    echo ERROR java OFBconv exit with %ERRORLEVEL% >>%LOG%
  ) else (
    echo ERROR: %ERRORLEVEL%
    pause
  )
) else (
  REM no return error from java:
```

include:../../scripts/genOFB/genSrc_odg.sh::Java::96::::

```
## set the java class path in JCPVISHIA as central batch, using the dot base call for the she...
## Note: The script needs the directory where tools is found (parallel to src), .../src is th...
. scripts/LinuxEnv/SetJCP.sh $PWD/..                                          ...SetJCP:
echo java -cp $JCPVISHIA org.vishia.fbcl.OFBconv --@$1:args >>$LOG          ...LOG:
echo java -cp $JCPVISHIA org.vishia.fbcl.OFBconv --@$1:args
java -cp $JCPVISHIA org.vishia.fbcl.OFBconv --@$1:args                      ...Java:
## Note the java running process writes some stuff on this console, but also in the -log ...LOG:
echo ... finished OFB translator
if test $? -gt 0; then                                                      ...ErrorCheck
  if test "NOPAUSE" = "NOPAUSE"; then
    echo ERROR java OFBconv exit with $? >>$LOG
  else
    echo ERROR: $?
    read -p " ...Press ENTER..." VAR
  fi
else
  ## no return error from java:
```

That are the core lines to call the OFB translator as Java program.

`SetJCP` The called `SetJCP.bat` `.sh` sets the environment variable `JCPVISHIA` used as argument for `-cp`, the *Java Class Path*. This is outsourced (centralised) to this special script, because it is used for other scripts too. The jar files have a time stamp in the name, to ensure using the correct version. If the version in tools is changed, then only this scripts `SetJCP.bat`, `.sh` need to be adapted. It should be fixed only once.

`LOG` The Java call command is written to the log file. That is only if the Java call does not work. Then the log file contains the information about, to able to analyse afterwards. Elsewhere, only the current command line output messages are existing.

But, the log file is created newly if Java works. Then this start information are not relevant.

`Java` Java should be available on your system. All Java versions from Java-8 (Oracle), also OpenJDK can be used.

The argument `--@%1:args` or `--@%1:args` is the argument file for this Java call evaluated in the class >>`org.vishia.util.Arguments`. This is the `TARGET` configuration file, it's really an argument file. The part `:args` is the used start label.

`ErrorCheck`: The Java execution can return with an error code, if something is wrong. Then the log file should be evaluated for the reason. Here only the error message is stored additionally in the log, and shown on screen.

If it returns with 0, no error, then the else branch is used, and further actions are done, see next page.

include:../../scripts/genOFB/genSrc_odg.bat::fdiff::96::::

```
REM show file differences...fdiff
echo %CD%
if ""=="%NODIFF%" call %SCRIPTDIR%\..\WindowsEnv\fdiff.bat "%BUILDD%\%CMPN%\cpp\genSrc" "%C...
```

include:../../scripts/genOFB/genSrc_odg.sh::fdiff::96::::

```
## show file differences...fdiff
echo $PWD
if ! test "$NODIFF" = "NODIFF"; then scripts/LinuxEnv/fdiff.sh "$BUILDD/$CMPN/cpp/genSrc" "...
```

This two snippets for windows and linux after the end of execution of the Java call offers to possibility for a comparison of all new generate files with the given ones. The concept is, the given (settled generated files as artefact) are stored in the component's directory below `cpp/genSrc` for C/++ generation. That files can be used as input for an IDE (Integrated Development Environment) for the target compilation and test. The new generated files are first written in the `BUILDD` directory. With the file comparison tool new generated code lines can be visited, and the files should be synchronised with the settled ones. This is not to write or retain some special manual code changes. Do never influence automated generated code. It is to see what is changed because of changing in the graphic.

`fdiff` with calling `fdiff.bat` or adequate `fdiff.sh` a text diff tool should be started. Which tool is used, this depends on the choice in this script files itself. Using WinMerge is recommended, also on Linux using wine.

The diff tool is opened and offers the possibility to show which is changed in comparison to the last generation. It means the new result should be updated with the diff tool. This is very proper able to do using WinMerge.

More that, it is possible to decide that the sources for compilation and run are not taken from the just new generated directory (this is inside build), instead the currently updated sources can be used. Then anyway updating is necessary, but on the other hand, if the new generated sources are removed because they are on RAM disk in the temporary location, the last current files are existing for compilation without newly generation.

This approach has one advantage more: If it is seen that the new generated sources are the same without change, then they should not be updated. It means the target build system does not see a new time stamp, and can faster build. This is interesting on large projects with a lot of graphic generated files.

Also, if there are only changes in a few comment lines, updating can be omitted, with the same advantage. It should be updated later because the differences on view should be obviously.

Next the both files called in this script are shown:

include:../../scripts/WindowsEnv/fdiff.bat::$::96::::

```
REM this batch file should call a text diff view tool.:
REM Using WinMerge is a good decision.
echo currDir=%CD%
echo called=%0
echo arg1=%1
echo arg2=%2
REM some experience
::start cmd /C start cmd /C
::c:\Programs\WinMerge-2.12.4-exe\WinMerge.exe %1 %2
c:\Programs\WinMerge\WinMergeU.exe %1 %2
```

include:../../scripts/LinuxEnv/fdiff.sh::$::96::::

```
fdiff.sh $1 $2  ## should be in path, call the desired fdiff tool:
```

Generally the files in the directory `src/organizeScripts` can or should be adapted by the user situation. This is seen here for the WinMerge tool. There are some versions installed, and here it is selected which version to use. It may be also recommended instead calling a central `fdiff.bat` or similar which is able to find in the PATH (recommended).

It is similar for Linux. Here the decision is using the proven Windows-Tool WinMerge. This is called with `wine` (the Windows emulator on Linux), but the WinMerge is specific installed on the users location, it is not a standard tool in the linux distribution.

include:../../scripts/genOFB/genSrc_odg.bat::Zmake::96::::

```
  REM Compile the C code with Zmake if script exis...Zmake
  if NOT "%MAKE%" == "" call %CMPNDIR%\makeScripts\%TARGET%.Zmake.jzTc.bat
```

include:../../scripts/genOFB/genSrc_odg.sh::Zmake::96::::

```
  ## Compile the C code with Zmake if script exis...Zmake
  if test -f $CMPNDIR/makeScripts/$TARGET.Zmake.jzTc.sh; then $CMPNDIR/makeScripts/$TARGET.Zm...
```

The following lines allow immediately compilation. Here also `cmake` or any other script can be called. The compilation is optional, but it is optimal if immediately from the LibreOffice Draw graphic a running of the result executable should be visited. The examples uses the `curveView` application to view simulation results.

include:../../scripts/genOFB/genSrc_odg.bat::end::96::::

```
  REM post processing after Java OFB translation done.
)
:: if "%NOPAUSE%"=="" pause
exit /B 0
```

include:../../scripts/genOFB/genSrc_odg.sh::end::96::::

```
  ## post processing after Java OFB translation done.
fi
if test "NOPAUSE" != "NOPAUSE"; then read -p " ...Press ENTER..." VAR; fi
```

This is the real end of the script.

The last `pause` is commented here, because usual the “press ENTER” may be forgotten, the command window remain opened in background and a lot of such forgotten background windows are remaining.

Do it only uncomment, if some seems be faulty in the last lines called, to keep the window open.

empty

6.1.4 Macro and Icon from LibreOffice Draw

include:../../Templates_OFB/LOffcMacros/OFBwdiff-bat.xba.txt::$::90

```
REM  *****  BASIC  *****
REM https://ask.libreoffice.org/t/how-to-check-folders-name-by-macro/44665
Sub OFBwdiff
 dim sCurFileURL As String
 dim sCurFileSys As String
 dim sFolderSys As String
 dim aPaths as Variant
 dim odtName as String

 sCurFileURL = ThisComponent.getURL()
 sCurFileSys = ConvertFromURL(sCurFileURL)
 aPaths=Split(sCurFileSys,"\")
REM prepare the odtDir and odtName from file path in LibreOffic...<odtDir>
 odtDir = ""
 odtDirSep = ""
 For i = Lbound(aPaths) to Ubound(aPaths)-1
   odtDir = odtDir & odtDirSep & aPaths(i)
   odtDirSep = "\"
 Next i
 odtName = aPaths(Ubound(aPaths))
REM odtDir is the directory where the odt file is stored. It is necessary to change to ...
REM odtName is the file name with .odt as extension
REM activate the next line MsgBox for debugging this script.
REM MsgBox "Current Path : name: " & odtDir & " : " & odtName

REM call the proper script in the directory beside
REM 1. argument: A batch file to call
REM 2. argument: 1= Focus os window in standard size
REM 3. argument only one string argument for the batch separated with spaces for more e...
REM 4. optional argument true then libre office waits for finish shell (sync)
REM see https://help.libreoffice.org/6.4/en-US/text/sbasic/shared/03130500.html
REM String(1,34) is the " (quotation char) ASCII = 34 = 0x22, used for surround "odtDir"
REM more as one argument separated with space, arguments maybe surround with "dir/odt"
REM Shell starts the given batch with argument...<Shell>
Shell(odtDir &"\..\makeScripts\genSrc_odg.bat", 1, odtName )

End Sub
```

This is the macro as clipboard-copy from the macro-editor in LibreOffice, stored in the file `OFB_Presentation/src/Templates_OFB/LOffcMacros/OFBwdiff-bat.xba.txt`. There is also a file `OFBwrbat.xba` beside as XML-file which can put immediately in the suite of LibreOffice macro files.

`<odtDir>` The file path in LibreOffice is gotten via `ThisComponent.getURL()`. The next lines build a String for the directory path, separated from the name. The Linux version uses here / instead \.

`<Shell>` For the Windows version a simple batch file is called. Due to the second argument `1` it is executed in an new opened command window. So you can see what's happen.

For Linux this solution depends on which command window program is used, and this depends on the Window environment of the Linux system. There are a few possibilities. Fo a common solution xterm is used. The last lines of the Linux macro version stored in `OFB_Presentation/src/Templates_OFB/makeScripts/OFBwr-sh.xba.txt`. are:

include:../../Templates_OFB/LOffcMacros/OFBwdiff-sh.xba.txt::Shell::90

```
REM Shell starts the given batch with arguments: &lt;:@&lt;Shell&gt;.&gt...Shell
Shell("xterm -e " & odtDir &"/../makeScripts/genSrc_odg.sh", 1, odtName )
REM Shell("xterm")
REM Shell("xterm -hold -e ls")
End Sub
```

xterm is available as simple standard usual on all Linux systems. It needs the command string after the option -e, this is then the shell script.

How to create and edit this macro in LibreOffice:

Use the menu *"Tools – Macros – Organize Macros"*, press the [*Organize*] button, create the macro, and copy the text into. The macro will be stored in the LibreOffice data location which is in user/… on your PC as global user specific location or just in the data directory in a portable version.

`c:\Users\..user..\AppData\Roaming\LibreOffice\4\user\basic\Standard\OFBwr-bat.xba` is the location on a Windows PC,

`/home/..user../.config/libreoffice/4/user/basic/Standard/OFBwr-bat.xba` is the location on Linux-PC

`c:\Users\...user...\AppData\Roaming\LibreOffice\4\user\config\soffice.cfg\modules\swriter\toolbar\standardbar.xml` contains the toolbar outfit.

adequate directory in a portable version: `...\LibreOfficePortable.24.2\Data\settings\user\..`

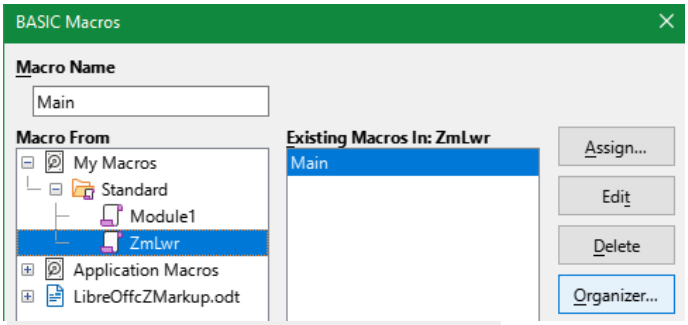


Figure 172: ZmLwr-Macro-create-edit.png

To get the button in the task bar, open "Tools – Customize", select the tab "Toolbar", select left side "Macros", search the macro and insert it for example here in the standard toolbar. For my LibreOffice usage I have reduced the amount of icons of the usual necessary ones, and that are all in *"Standard"*. The Toolbar can be adapted in LibreOffice in a real specified kind. The right image shows how a macro can be inserted in the toolbar.

This image right side shows more possibilities to set the proper toolbar with icons. For example the "Save" icon is proper to have, because here also the save status is shown. Whereas "Save as" is a more rarely used action, do not need a button because it can be start via menu.

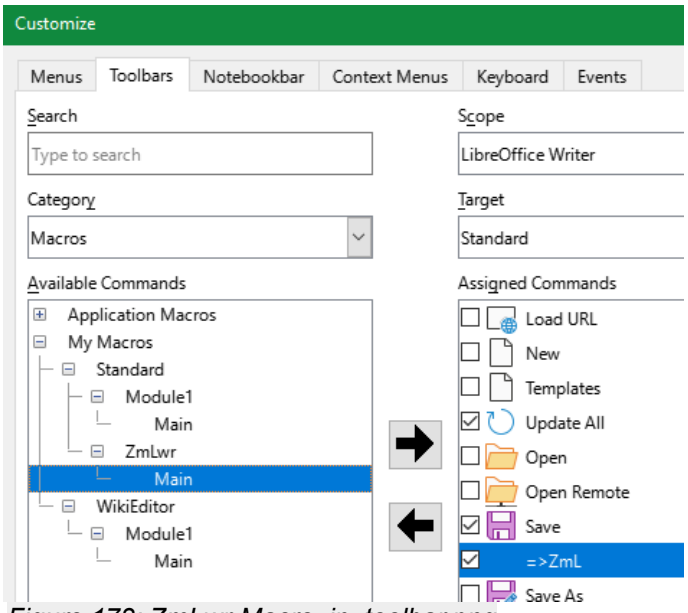


Figure 173: ZmLwr-Macro_in_toolbar.png

(empty line)

6.1.5 Handling of include in C/++ or import and real used type names

TODO

6.1.6 Error messages while translating

Generally, translation is continued if an error is found, to get the best usable result. But one error can cause other errors. That's why look for it.

`ERROR parse templates for codegen file: SCRIPTFILE.otx:java.text.ParseException: script is already existing, it is twice: otx: NAME`

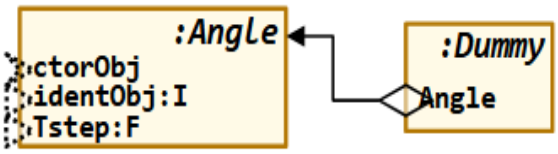
This is an error which occurs only if a generation otx script was changed. The reported script exists twice, maybe the first occurrence is in another script. The names of the sub scripts are unique over all scripts. The error is immediately output independent of usage of this sub script. Please fix it.

`ERROR other type given for the FBlock than existing already: FBtype FBtypeGiven`

This occurs if a name of a FBlock us used twice, another FBlock has the same name, but fortunately another FBtype, so that this graphic error is obviously. Note that the same instance of an FBlock can drawn in more as one Graphic Block (of course with the same FBtype), but therefore a confusion between FBlock names cannot be detected automatically.

`ERROR graphic FBtype has no THIS port: FBtype.name @xy`

Figure 174: odg/HowtoCreateTHIS.png



To access a FBlock as reference, for example via FBoper (see *5.8.11 FBoper, operation for a FBlock* page *151*) It is necessary that a formally port exists to refer it, with the name `THIS`. The port is created if an aggregation exists to it. If the FBtype is created in an extra *"Definition Diagram"*, it is necessary to do it there:

`INFO: do not generate target source code, non deterministic data types given: MDLNAME`

This message is an information if the module is really not for direct code generation, only if it is used in another module where the data types are determined by using. But if you expect that there are not non deterministic types, you should look in the generated report file `MDLNAME.dTypeUsg.txt` and look after the title: `=== FBlock.Pins with non deterministic DType ===` to see which pins are cause this behavior. Look especially for the module's data in and out.

6.1.7 Templates for code generation

The code generation is controlled by templates. Hence the adaption to any programming language and also to any rule set for a given programming language is possible.

The templates can be contained in more as one file. Any file contains the rule for some parts of code.

This chapter is to describe. TODO.

empty page

6.2 Code generation control with otx scripts

This otx scripts have a syntax described in: >>`OutTextPreparer`

This otx scripts have a syntax described in:

./../../Java/pdf/OutTextPreparer.pdf (www)

empty

6.3 Presentation of the graphic and results in files

Table of Contents

There are different files and format of files where the software drawn with graphic in OFB is presented:

- `*.odg`: The graphic file itself in LibreOffice odg format (Open Document Format)
- `report/*.fbg`: The read graphic data presented in a specific format, as raw data (without yet building of FBcl data as FBlock with pins, functions and connections)
- `FBcl/*.fbd`: The read and translated graphic data presented in a syntax near the IEC61499 standard (for automation devices). The event flow which is typically for IEC61499 diagrams is also typical for this OFB approach, and the other approaches are also proper as a *"Function Block connection language"*, as this textual graphic presentation can be seen.
- `genSrc/*.c, *.h`: The generated files for target source code are last the presentation of the graphic. If you change the graphic the results are seen there. But it is a wide way from graphic to these result files. That's why the both other intermediate formats may be important.

6.3.1 The original odg format (Overview)

An `*.odg` file is a zip file. You can look into with a zip presentation tool (use for example the Total Commander (https://www.ghisler.com/index.htm).

Internally the `content.xml` is the important file. it is XML and contains maybe readable the information in the graphic, inclusively the name of the styles, but not the appearance of predefined styles. They are defined in the `styles.xml`.

For example it is possible to synchronize styles from other files, (which are changed, improved, newer) by simple exchange the `styles.xml` file inside this zip archive. Of course you may be familiar with such things and have made a save copy. But it is possible a daily work.

The `content.xml` is read out from this OFBconv tool.

6.3.2 Graphic saved with the option The original odg format (Overview)

After reading and gathering the graphic, it can be saved in the gathered raw format to see differences in graphic from one to another version. This is done giving the option for the OFB converter see

```
-fbg
```

or also

```
-fbg:path/to/file-$(DATE)_$(TIME).fbg.txt
```

The last variant determines a dedicated store path for your own, and additional with the `$(DATE)` and `$(TIME)` the possibility to have a file name containing the current time stamp to get different versions. But think about to remove not necessary versions later.

The created file contains an overview and details of the read and interpreted graphic, offered as list of GBlocks (Graphic Blocks) and their pins with connection. That are 'raw data' because the association to FBlocks is already done, but the FBlocks are not completed. It looks like:

```
== FBlock in Graphic, Details:
```

```
@2'60(54..74, 50..60)  h1p =fb  ==FBlock== h1
  Pins:
  fbPinDst<---aggr--- fb=bf.param @2'90(92..9
  fbPinDst<---unspec--- demux=g_2_9_58.f @2'9
  Din= fq ('fq')  <--dataflow--- expr=e_2_4_6
  Evin= setFq ('setFq')
  Evin= init ('init')

@2'60(54..74, 63..69)  h2p =fb  ==FBlock== h2
  Pins:
  fbPinDst<---unspec--- demux=g_2_9_58.2 @2'9
  Din= fq ('fq')  <--dataflow--- expr=fq2.'<n
```

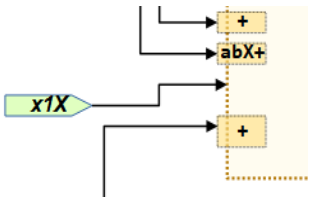
The output is cut here on right side. But the principle should be able to recognize. It contains the graphic position with page, coordinates in mm. The mm-value after the page is the vertical row, where some FBlocks are centralized in vertical order. This arrangement is important for the order of FBlocks and their pins. `@2,60(54..74, 66..69)` means, it is a GBlock on page 2, vertical row on 60 mm, with coordinates 54..74 mm in x and 63..69 mm in y. It is able to found on the graphic with this coordinates. `h2p` is the name of the also associated FBlock, the FBtype follows in the line (here not visible).

The pins follow in there graphic order or order determined by a `` `1 `` in the pin text, see 5.3.6 nrGpos, order of pins after grave page 40. The position is here not given, it should be able to find inside the GBlock. But the position of the connected pin(s) are given, to get an idea where the connection ends.

The connection kind is written in `<--dataflow--` as seen in the example. Here not the draw style is given (it would be `ofcDataflow`), instead the internal connection kind name is written out. That is similar for the kind of the GBlock. `=fb` is a GBlock of style `ofbFBlock`.

This information may be essential if you have a problem in the graphic. For example non connected connectors will be unfortunately not shown as ‘non connected’ This is a disadvantage of LibreOffice draw which may be fixed in the future:

Figure 175: NonConnectedConnector .png:



If you look on this image right side, the connect from bottom is not connected to the `[ + ]` pin. If you look exact, the point is not in the mid. But this is not an evidence, because it can be also in the mid and unconnected, or also on this position with a glue point. The result comes by shifting the pin in the near of the connector, the connector does not snaps on the glue point, only if the connector end point is moved, it snaps. The result is not visible.

Looking on the `file.fbg` output helps: The connection is missing there, but seen in the graphic. The only one explanation or hint is: Look whether it is connected.

In this image example above: The `[x1X]` Xref is connected to the whole GBlock, and this is shown in the `*.fbg` file as connection to the `fbPinDst`.

What does this helps? If you change anything in the graphic, you get a changed target code, but you have forgotten what you had changed in the graphic, then look here. Also for traceability what was changed in the past, by another people, this is helpful. It means this file should be a part of a version management system as presentation of the graphic.

A sensible approach to commit generated files is the following: Let it write on a temp location, but compare and copy it manually with or to a `/genSrcCmp/` location. Do not forgot it to copy, /or make it automatically by a script). Then commit the file from the `/genSrcCmp/` location if you have finished your work. Then you do not get scratch working versions in your repository.

Not all graphic changes force changed target code. But you can also study which changes forces target code changes. For example also changing in graphic position determines the order of pins in the target code or also the order of statements or operations.

6.3.3 The FBcl format or IEC61499, file.fbd

This format is mentioned some times in the description to explain internal data. The IEC61499 norm is here used as base, with some enhancements. The IEC61499 was developed from about 2005 as a new approach for automation control software, but it is not used frequently by the big players, instead they use since decades the IEC61131.

Using of events for execution control is one or the most important advantage of the approach which is standardized in IEC61499. This is a really proper idea, and hence also used for the OFB concept.

Another original approach of IEC61499 is the distribution of the drawn and presented graphic with Function Blocks in more distributed automation devices. This is automatically done by attribution of dedicated Function Blocks to specific devices. The necessary communication between this devices is then automatically determined by the implementation process. In IEC61131 this should be usual done manually by planning of communication as extra phase.

This is a short history of IEC61499.

The distribution of the generated target code to more as one devices may be also very interesting for the OFB concept and should be done also, later.

But now have a look to the appearance of a FBcl (IEC61499) file, its syntax:

include:../../+ExmplBandpassFilter/cpp/genSrc/FBcl/ArrayBandpassFilter.fbd::'F"VAR_INPUT'::45::5:-::'END_EVENT':+::+4:-::'END_EVENT':+

```
FUNCTION_BLOCK ArrayBandpassFilter  (*writ...
EVENT_INPUT
  ctor WITH Tstep, q1, qh, Tfd; (* kind=Ev...
  init WITH fq, updparamTHREAD, updTHREAD;...
  param WITH fq; (* kind=Evin@2 ~evdata=0x...
    ......
END_EVENT
EVENT_OUTPUT
  paramO WITH yfq1; (* kind=Evout@0 ~evdat...
  stepO WITH yph1, Yg, Yfilt, ym, yg6, Y2h...
    ......
END_EVENT
VAR_INPUT
```

Hint: The text after `include:` controls the immediately including of a source text, here the really created `ArrayBandpassFilter.fbd`. It means it is not written in the document, it is really. created by the OFBconv.

The FBcl file starts with the interface declaration, with the `EVENT...`. following by the variables `VAR` for input and output. The keyword `WITH` is IEC61499 conform and specifies the `VAR` variables, which are associated to the events.

The `(*...*)` is comment in IEC61499 and contains here some additional also internally information about the events, not used for evaluation, only used for information.

include:../../+ExmplBandpassFilter/cpp/genSrc/FBcl/ArrayBandpassFilter.fbd::'VAR_INPUT'#30::45::3:-::'END_VAR':+::+4:-::'END_VAR':+::+1:-

```
VAR_INPUT
  fq : REAL; (* kind=Din@0 ~evdata=0x6 ~in...
  x : REAL; (* kind=Din@1 ~evdata=0x10 ~in...
    ......
END_VAR
VAR_OUTPUT
  yfq1 : REAL; (* kind=vout@0 ~evdata=0x1 ...
  yabzb : CREAL; (* kind=zout@1 ~evdata=0x...
    ......
END_VAR
```

Here it is seen that all input variables are enhanced with `$event`. The `$` as a character inside the identifier is not admissible in IEC61499, but either the IEC61499 can be enhanced, or instead `$` two `__` can be written here. The meaning of this $event is: The local variables on input (style `ofpDout...` in the Module interface style `ofbMdlPins` are local valid, not for the whole module. That's why the name is enhanced with the associated event to distinguish same variable names in different event contexts. This variant of data visibility is not intended just in the IEC61499.

Also the type `CREAL` for `complex float` is not existing just in IEC61499, but the `REAL` is. The type names in this file follows all the intention of IEC61499. One of the important intention of data type names in IEC61499 is: Other than in C/++ and similar languages the numeric types `int` etc. are well distinguish from the bit types, which is also `int` in C-like languages. See *5.4 Data types* page *58*.

Further follows:

include:../../+ExmplBandpassFilter/cpp/genSrc/FBcl/ArrayBandpassFilter.fbd::'FBS'#30::45::3:-::'b3f':+::+6:-

```
FBS
  JOIN_dqref_X_dq : Join_OFB                ...
  JOIN_e_3_93_42_prep : Join_OFB            ...
    ......
  b3f : OrthBandpassF_Ctrl_emC( identObj:=...
  bf : OrthBandpassF_Ctrl_emC( kA:='3.0', ...
  dgI : VarZ_OFB                             (...
```

```
  dgI2 : VarZ_OFB                              ...
  dgI2_X : Expr_OFB( expr:='=+,,+;,*,;;', ...
  dgI_X : Expr_OFB( expr:='=+,,+;,*,;;', K...
```

This are the Function blockS contained in the FBcl file due to IEC61499 syntax. The `Join_OFB` are created from the OFBconv if events should be joined. In one implementation of IEC61499 there are named "`RND`" which comes from *"rendezvous"*

For some FBlocks you see an initializing expression after `:=`. This is IEC61499 conform.

The `Expr_OFB` is the common FBtype to implement expressions. In IEC61499 systems there are some standard FBlocks for that. The `Expr_OFB` is controlled by the `expr` input string in its functionality, see *5.8.12 How are expressions presented in IEC61499?* page *152*.

In IEC61499 there are three types of fbd files, this is one of them, the so named *"composite function block type"*. The others are the *"basic FBlock type"* which contains algorithm in Structure Text language and so named ECC (*Execution Control Charts*, State Machines). and the *"Service interface function block type*", which is also not used here.

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'VAR_OUTPUT'#30::45::3=-::'END_VAR'=+::+5=-::'fb3'=+::+2=-

```
VAR_OUTPUT
  ye1 : ARRAY [0..2] OF REAL; (* kind=Dout...
  ye2 : ARRAY [0..15] OF REAL; (* kind=Dou...
    ......
END_VAR
(*VAR     Note: That are the FBlocks VarX_...
  (*gain2 : ARRAY [0..2] OF REAL;( *  vout *)
  (*gf : REAL;( *  dout *)
  (*v1 : ARRAY [0..2] OF REAL;( *  dout *)
    ......
  fb3 : ARRAY [0..2] OF FBx_FB                 ...
  gain2 : VarV_OFB                             ...
```

This is an example from another FBcl file. The writing style `ARRAY[0...]` is IEC61499 conform for signals, but not sure for FBlocks. Starting with `0` is necessary for the target languages.

include:../../+BasicTest/cpp/genSrc/FBcl/ArraySlideDemux.fbd::'END_FBS'#30::45::4=-::'v1.prepO'=+::+6=-::'END_CONN'=+::+2=-

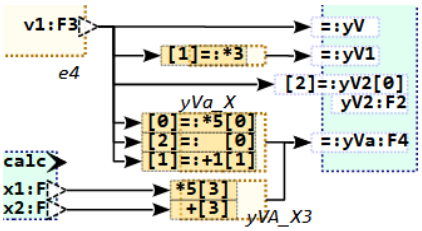
```
END_FBS
EVENT_CONNECTIONS
  calc TO e1.prep;  (* evChain=0,^1->; con...
  calc TO e3.prep;  (* evChain=0,^1->; con...
    ......
  v1.prepO TO e_5_128_97.prep;  (* evChain...
  v1.prepO TO yVa_X.prep;  (* evChain=0,^1...
  yVa_X.prepO TO JOIN_calcO.J11;  (* evCha...
  yVa_X3.prepO TO JOIN_calcO.J11;  (* evCh...
END_CONNECTIONS
```

```
DATA_CONNECTIONS
    ......
END_CONNECTIONS
END_FUNCTION_BLOCK
```

This are now the connections between pins of the FBlocks and pins of the interface. First `EVENT_CONNECTIONS`, then `DATA_CONNECTIONS`. With that all is described. The inner Functionality of a FBlock is described with designation of the type. The Type is either implemented direct in target code, or by another OFB diagram, or also maybe by another IEC61499 description. The data type are determined here, inclusively arrays. The connections are denominated here, that's all. From the FBcl file respectively the IEC61499 description all can manually tracked, and any tool can do code generation, with some independent of the functionality necessary environment information.

Some small divergences are existing to IEC61499 which may be possible to clarify in future. For example:

Figure 176: ArraySlideDemux_VectorAccExpr-2.png



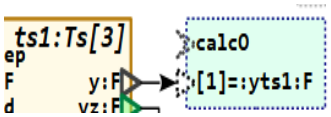
Array access to variables is part of the connection

```
v1.Y[2] TO yV2[0];
v1.Y[0] TO yVa_X.X1;
v1.Y[2] TO yVa_X.X2;
```

The mapping of the inputs of `yVa_X` to the output parts is clarified by the initializing String of this expression:

```
FBS ... yVa_X : ARRAY [0..3] OF Expr_OFB(
expr:='~+,@[0],@[0],@[1];...)
```

Figure 177: ArraySlideDemux_AccFBelem.png



Here the element `[1]` of the vectored FBlock is accessed, and then its output `y`, connected to `yts1`:

```
ts1[1].y TO yts1;
```

6.3.4 The original odg format (Overview)

One odg file

empty page

6.4 Zmake with acceleration of generated secondary source files

TODO

7 Overview show styles of this document

```
Simple code block
with some lines.
```

```
Cmd line
or file tree presentation
```

```
REM A windows batch file
or a shell script
```

REM A windows batch file

```
##Some configuation data
a = "test"
```

```
void javaOperation(float arg) {
  return;
}
```

```
void cppOperation(float arg) {
  return;
}
```

```
VARIABLES
  a AS float
##This is a otx script:
```

```
<:otx: VarV_UFB: evSrc, fb, evin, din>
<:if:din.isComplexDType()>
  thiz-><&fb.name()>.re = <&genExprTermD...
  thiz-><&fb.name()>.im = <&genExprTermD...
<:else>
  thiz-><&fb.name()> = <&genExprTermDin(...
<.if><: >
<.otx>
```

```
VARIABLES
  a AS float
```

`Code`, `ccode`: And `here is simple code`

`CodeCmd`, `cCmd`: this is a `cmd call arguments` example

`CodeScript`, `cS`: a `part of a script`

the `small form` (?)

`CodeCfg`: `cCfg`: `config data`

and some configuration data

and also `javaOperation` with arguments

#also C or C++ language `cppOperation()` given

`CodeZbnf` and `cZ` for `Zbnf syntax`

and

`CodeOtx`.and `cOtx` for otx scripts

A nomination of a style, this is

A `Marker` with style `cM` should be demonstrative

wait what is `cV`?

`CodeFBcl` and `cFBcl` for `VARIABLES` in a IEC614499 source